OAshi S.à r.l.
10, Rue du Bocksberg
L-6614 Wasserbillig
Luxembourg
Tel: +352 26 71 30 1
E-Mail: info@oashi.com

Wissensbasis | Knowledge Base | Base de connaissances : MyTISM Framework

MyTISM - Handbuch für Anwender, Administratoren und Entwickler

Zielgruppe | Target Group | Groupe cible:
Anwender, Power-User, Administratoren und Entwickler.

Ziel | Goal | Objectif:
Das universelle MyTISM-Handbuch.

Stand | Date: 2026-06-03

Inhaltsverzeichnis

Vorwort

Willkommen in der Welt von MyTISM.

Der tägliche Umgang mit einer umfangreichen Software erfordert verlässliche Werkzeuge und ein Handbuch, das klare Antworten liefert. Dieses Dokument dient als zentrale „Single Source of Truth“ für alle Akteure im System. Um Sie bestmöglich auf Augenhöhe abzuholen, kombiniert dieses Handbuch zwei didaktische Prinzipien: einen logisch aufbauenden Lesefluss und die sogenannte „progressive Tiefe“.

Jedes technologische Thema beginnt mit den funktionalen Grundlagen und der Bedienung der Benutzeroberfläche für den operativen **Anwender**. Darauf aufbauend werden erweiterte Konfigurationsmöglichkeiten (wie Eingabemaskendesign oder komplexe Abfragen) für erfahrene **Power-User** und **MyTISM-Administratoren** erläutert. Den Abschluss eines jeden Kapitels bildet ein tiefer architektonischer Einblick in die Backend-Mechanismen, APIs und Server-Strukturen für **Entwickler** und **Server-Administratoren**. Das Handbuch erlaubt es Ihnen somit, in jedem Thema exakt so tief einzutauchen, wie es für Ihre tägliche Rolle erforderlich ist.

Um komplexe Themen greifbar zu machen, verzichten wir im vorderen Teil der Kapitel bewusst auf trockene Theorie. Stattdessen demonstrieren wir Ihnen die Kernfunktionen praxisnah anhand eines durchgehenden Beispielprojekts: einer fiktiven Krankenhausverwaltung. Je tiefer und technischer die Struktur wird, desto mehr weicht diese Metapher einer präzisen Fachsprache für unsere IT-Experten.

Dieses Handbuch ist Ihr verlässlicher Begleiter, der Ihnen vom ersten Klick an zur Seite steht und später gleichzeitig als Nachschlagewerk dient.

Chapter 1. Einführung und Grundlagen

1.1. Was bedeutet MyTISM?

Der Name MyTISM steht als offenes Akronym für „My Tool Is My…“ – wahlweise ergänzt durch Begriffe wie Solution, Key to Success oder Inspiration. Hinter diesem Namen verbirgt sich ein umfassendes Baukastensystem für Softwareanwendungen. Anstatt unzählige kleine Einzellösungen mühsam miteinander zu verknüpfen, bietet MyTISM eine zentrale Plattform. Sie vereint die sichere Speicherung von Daten, die Verarbeitung von Geschäftsregeln und die grafische Benutzeroberfläche auf Ihrem Bildschirm in einem durchdachten Gesamtsystem. Das System wird von der OAshi S.à r.l. entwickelt und kontinuierlich betreut.

1.2. Historie und Motivation

Die ursprüngliche Idee zu MyTISM entstand im August 2000 aus einer konkreten Frustration im Alltag von Softwareentwicklern. Wenn ein System damals erweitert werden sollte – beispielsweise um ein neues Feld für die „Mobilnummer“ in einer digitalen Kundenakte –, bedeutete dies einen enormen, fehleranfälligen Aufwand. Entwickler mussten die Tabellenstrukturen im Hintergrund händisch ändern, den Programmcode mühsam anpassen und die Masken auf dem Bildschirm komplett neu zeichnen. MyTISM wurde mit der Vision erschaffen, genau diese starre und zeitaufwändige Trennung aufzuheben. Das System behandelt Informationen konsequent als greifbare „Objekte“ anstatt als nackte Datensätze. Wenn heute ein neues Feld im System definiert wird, weiß MyTISM automatisch, wie es gespeichert, verarbeitet und auf dem Bildschirm angezeigt werden muss. Das macht die Weiterentwicklung der Software nicht nur extrem schnell, sondern reduziert auch Fehler auf ein absolutes Minimum.

1.3. Das Schichtenmodell (3-Tier-Architektur)

Damit das System auch bei Tausenden von Benutzern absolut stabil und übersichtlich bleibt, ist es im Hintergrund in drei strikt getrennte Schichten unterteilt. Die erste Schicht ist die Präsentationsebene (Frontend), also die Masken, Fenster und Menüs, die Sie täglich auf Ihrem Bildschirm sehen und bedienen. Die zweite Schicht ist die zentrale Verarbeitungsebene (Middleware), ein Server, der im Hintergrund alle Berechnungen durchführt, Automatismen steuert und Ihre Zugriffsrechte prüft. Die dritte Schicht ist das sichere Datenfundament (Backend), eine Datenbank, in der alle Ihre eingegebenen Informationen dauerhaft und strukturiert gespeichert werden. Der große Vorteil dieser strikten Trennung: Wenn das Aussehen einer Bildschirmmaske geändert werden soll, bleiben die Verarbeitungsebene und das Datenfundament davon völlig unberührt. Das macht MyTISM für Anwender leicht zu bedienen und für die IT-Abteilung extrem

wartungsfreundlich und zukunftssicher.

1.4. Wichtige Grundbegriffe

Um Ihnen den Einstieg in die Arbeitsweise von MyTISM zu erleichtern, haben wir die wichtigsten Grundbegriffe in einer kurzen Übersicht zusammengefasst. Stellen Sie sich das System nicht einfach nur als simple Datensammlung vor, sondern als ein intelligentes, hochgradig vernetztes Informationsnetzwerk. Auch wenn die Basis an eine klassische Tabellenkalkulation erinnert, agiert MyTISM weitaus dynamischer: Daten existieren hier nicht starr nebeneinander, sondern interagieren aktiv miteinander und reagieren auf komplexe Geschäftsregeln. Die folgenden Definitionen helfen Ihnen dabei, diese dynamische Struktur schnell zu verinnerlichen.

Begriff	Erklärung
Solstice	Die zentrale, grafische Kommandozentrale auf Ihrem Bildschirm. Hier greifen Sie im Arbeitsalltag auf das gesamte Datennetzwerk zu, pflegen Informationen und steuern Ihre operativen Prozesse.
Entität (Entity)	Ein eigenständiger, strukturierter Datenbereich im System. Wenn Sie an eine herkömmliche Tabelle denken, ist die Entität das gesamte Tabellenblatt (z. B. der Bereich „Kunden“), das hier jedoch intelligent mit dem restlichen System kommuniziert.
Attribut (Attribute)	Eine spezifische Eigenschaft oder ein konkretes Eingabefeld innerhalb einer Entität. In der Tabellen-Denkweise entspricht dies einer einzelnen Spaltenüberschrift (z. B. das Feld „Vorname“ oder „Kundennummer“).
Relation (Beziehung)	Das aktive Bindeglied zwischen Entitäten. Es handelt sich um Querverweise, die dem System beispielsweise sagen: „Dieser spezifische Kunde aus Datenbereich A ist der Empfänger dieser konkreten Rechnung aus Datenbereich B“.
BO (Business Object)	Ein konkreter, einzelner Datensatz, der im System existiert. Das ist exakt eine vollständig ausgefüllte „Zeile“ Ihrer Daten, die nun als interaktives Objekt durch die Software verarbeitet werden kann.
Ldel (Logical Delete)	Der systemweite Schutzmechanismus vor versehentlichem Datenverlust. Löschen Sie ein Objekt, wird dieses nicht physisch zerstört, sondern vom System lediglich als „gelöscht“ markiert (Soft-Delete). Dadurch wird es in der Regel systemweit unsichtbar, oder in Ausnahmefällen durchgestrichen dargestellt

Begriff	Erklärung
Schema	Der technische Bauplan oder die fundamentale „DNA“ der Software. Hier legt der Entwickler fest, welche Entitäten, Attribute und Relationen im gesamten Netzwerk existieren. Das Schema kann allerdings auch zur Laufzeit dynamisch um weitere virtuelle Entitäten oder Attribute erweitert werden.
Strukturelemente	Der Sammelbegriff für alle sichtbaren Bausteine Ihrer Benutzeroberfläche. Dazu gehören fertige Eingabeformulare, dynamische Suchfilter (Lesezeichen) oder maßgeschneiderte Druckvorlagen (Reports).

Chapter 2. Architektur, Schema & Modularisierung

2.1. Grundlagen & Konzepte

Das Fundament jeder MyTISM-Anwendung ist der zentrale Bauplan, das sogenannte Schema. Dieses in XML verfasste Dokument definiert die gesamten Datenstrukturen, Eigenschaften und relationalen Beziehungen des Systems. Der enorme Architekturvorteil von MyTISM besteht darin, dass das Framework diesen Bauplan dynamisch auswertet und daraus vollautomatisch die passenden Datenbanktabellen sowie die Basis-Masken für die Benutzeroberfläche generiert. Entscheidet die Klinikleitung beispielsweise, dass künftig bei jedem Patienten die Blutgruppe erfasst werden muss, genügt ein einziger deklarativer Eintrag im Schema. Das System erzeugt daraufhin selbstständig die notwendige Datenbankspalte im Backend und das entsprechende Eingabefeld in der digitalen Patientenakte des Clients.

2.1.1. Entitäten, Attribute und Vererbung

Die realen Geschäftsobjekte werden im Schema als Entitäten abgebildet, welche auf physische Tabellen referenzieren. Wir unterscheiden dabei zwischen eigenständigen Hauptobjekten, wie einem Patienten oder einem Arzt, und abhängigen Unterobjekten, wie einer einzelnen Blutdruckmessung innerhalb einer Behandlungsakte. Die Eigenschaften dieser Objekte werden als Attribute definiert. Eine technische Besonderheit bilden virtuelle Attribute, deren Werte nicht physisch in der Datenbank gespeichert, sondern zur Laufzeit dynamisch berechnet werden. Die tagesaktuelle Aufenthaltsdauer eines Patienten ist ein solches virtuelles Attribut, das bei jedem Aufruf live aus dem Aufnahmedatum errechnet wird und somit niemals veralten kann.

Um redundante Definitionen zu vermeiden, unterstützt das Schema eine echte objektorientierte Vererbung. So kann eine allgemeine Basis-Entität wie „Medizinisches Ereignis“ grundlegende Attribute wie das Datum und den behandelnden Arzt zentral vorgeben. Konkrete Entitäten wie „Operation“ oder „Visite“ erben von dieser Basis und erweitern sie lediglich um ihre jeweils spezifischen Fachfelder. Ist eine Basis-Entität im Schema als „abstrakt“ markiert, dient sie rein der Vererbung und kann vom Anwender nicht als eigenständiger Datensatz angelegt werden.

2.1.2. Löschkonzepte und Datenhaltung

Aus Gründen der Nachvollziehbarkeit und Compliance, die besonders bei Behandlungsdaten essenziell sind, löscht MyTISM Datensätze standardmäßig nicht physisch. Löscht ein Anwender ein Objekt über die Benutzeroberfläche, wird dieses über

das sogenannte Soft-Delete-Verfahren (Ldel) in der Datenbank lediglich als unsichtbar markiert. Ein unwiderruflicher Hard-Delete, der die Daten physisch von der Festplatte tilgt, erfolgt erst durch automatisierte Hintergrunddienste nach Ablauf definierter Aufbewahrungsfristen.

2.1.3. Relationen, Module und Offline-Betrieb

Entitäten existieren im System selten isoliert, sondern sind über Relationen miteinander verknüpft. Die Architektur erzwingt dabei bidirektionale Beziehungen. So ist jederzeit auswertbar, welche Medikamente ein bestimmter Patient aktuell erhält und umgekehrt, welchen Patienten ein bestimmtes Medikament verabreicht wurde. Die Anwendungsarchitektur ist zudem konsequent modular aufgebaut, sodass fachliche Erweiterungen – wie etwa ein integriertes IT-Ticketsystem – als gekapselte Module an das Kernschema angedockt werden können.

Für Ausfallsicherheit und mobile Einsätze bietet das System eine tiefgreifende Offline-Fähigkeit. Der Laptop eines Notarztes im Rettungswagen kann durch eine lokale MyTISM-Instanz völlig autark betrieben werden. Sobald das Gerät wieder über eine Netzwerkverbindung zum Hauptsystem verfügt, synchronisiert der Client alle erfassten Offline-Daten vollautomatisch mit dem zentralen Server.

2.2. Benutzeroberfläche & Bedienung

Während reguläre Anwender primär in den fertigen Formularen arbeiten, müssen Power-User oftmals tiefe Datenstrukturen abteilungsübergreifend verstehen und filtern.

2.2.1. Das EntityTool zur Schema-Visualisierung

Die Datenmodelle wachsen in Enterprise-Projekten erfahrungsgemäß schnell zu komplexen Netzwerken heran. Ein einzelner Patient ist mit Aufnahmen verknüpft, diese wiederum mit Behandlungen, Diagnosen und Medikamenten, welche ihrerseits feingranulare Dosierungsanweisungen enthalten. Um bei solch tief verschachtelten Informationsbäumen den Überblick zu behalten, stellt das Framework für Power-User und Fachadministratoren das Werkzeug `entityTool` bereit. Dieses Tool generiert aus dem aktuellen Zustand eines laufenden Servers heraus eine interaktive, grafische Visualisierung des gesamten Schemas. Die Darstellung kann bequem im Web-Browser betrachtet und durchsucht werden. Sie dient als essenzielle Arbeitsgrundlage, um komplexe Lesezeichen-Filter oder Auswertungen über mehrere Relationen hinweg fehlerfrei zu konzipieren.

2.3. Erweiterte Konfiguration (XML-Schema)

Die erweiterte Konfiguration der Benutzeroberfläche und des Systemverhaltens erfolgt zentral durch den Entwickler im XML-Schema.

2.3.1. Navigation und Ordnerstruktur

Die visuelle Positionierung von Formularen im Navigationsbaum des Clients wird über das `<Folder>`-Tag gesteuert. Dieses Tag wirkt sich auf alle nachfolgenden Deklarationen aus, bis ein neues `<Folder>`-Tag definiert wird.

Es existieren syntaktisch keine verschachtelten `<Folder>`-Tags im XML-Schema. Tieferliegende Ordnerstrukturen müssen stattdessen zwingend über absolute Pfadangaben im Attribut deklariert werden.

```
<Folder path="Medizinische_Akte/Laborwerte"/>
```

2.3.2. Darstellung von Entitäten und Listen

Die visuelle Repräsentation eines Objekts in der Software wird primär über das `<ui>`-Tag der jeweiligen Entität gesteuert. Das Attribut `description` definiert hierbei, wie das Objekt in Suchfeldern und Listen als lesbarer Text formatiert wird.

Das CBOFormat für die textuelle Darstellung

Für diese Formatierungen nutzt das System das sogenannte `CBOFormat`. Es transformiert rohe Business Objekte anhand ihrer Attributwerte in Zeichenketten. Diese Formatieurng kann systemweit verwendet werden, von der grafischen Benutzeroberfläche bis hin zur Generierung von Reports und REST-APIs.

Die Syntax mischt feste Textkonstanten flexibel mit dynamischen Attributwerten. Statischer Text wird dabei in einfache Anführungszeichen eingeschlossen, während Attribute direkt adressiert werden.

Einfaches Syntax-Beispiel im Schema

```
'Abteilung: 'Name' ['Id']'
```

Ein konkreter Datensatz des Typs `Fachabteilung` mit dem Namen „Kardiologie“ und der ID `10046689` wird durch dieses Format zur Zeichenkette „Abteilung: Kardiologie [10046689]“ transformiert.

Verhalten von Listen und Tabellen

Aus Performance-Gründen werden verknüpfte Listen standardmäßig erst beim aktiven Aufruf durch den Anwender geladen. Für sehr kleine, statische Auswahllisten kann dieses Verhalten über das Attribut `loadImmediate="true"` überschrieben werden, um ein

flüssigeres Arbeiten zu ermöglichen.

```
<ui loadImmediate="true"/>
```

Das Sicherheitsattribut `linkOnly="true"` verhindert das direkte Neuanlegen von Objekten aus Kontextmenüs heraus. Anwender können in diesem Fall nur bestehende Datensätze verknüpfen, jedoch keine neuen Entitäten über die Auswahlliste generieren.

Die initiale Sortierung von Tabellen wird über `defaultSorting` im Format `Spaltenname:Sortierrichtung` konfiguriert.

```
<ui defaultSorting="Name:ASC Beschreibung:DESC"/>
```

Die Mehrfachsortierung nach zwei oder mehr Spalten, wie im obigen Beispiel gezeigt, erfordert eine erweiterte Lizenz in MyTISM.

Über den `defaultSelectionFilter` lässt sich die Ergebnismenge von Auswahllisten durch feste OQL-Bedingungen hart einschränken. Dieser Filter greift serverseitig und kann vom Anwender auf der Oberfläche nicht umgangen werden.

```
<ui defaultSelectionFilter="'Inaktiv = NULL OR NOT Inaktiv'"/>
```

2.3.3. Widgets und Dateneingabe

Das Verhalten einzelner Eingabefelder (`<attr>`) lässt sich über sogenannte UI-Tipps detailliert justieren. Um ein mehrzeiliges Textfeld anstelle eines standardmäßigen, einzeiligen Eingabefeldes zu erzwingen, wird der UI-Tipp `Area` verwendet.

```
<ui tips="Area"/>
```

Bei booleschen Entscheidungen (Ja/Nein) erlaubt die Datenbank oftmals einen dritten, undefinierten Zustand (`NULL`). Dieser unbestimmte Zustand kann auf der Oberfläche durch den Tipp `triState:false` explizit verboten werden, um eine klare Entscheidung vom Anwender zu erzwingen.

```
<ui tips="triState:false"/>
```

Alternativ steht für diesen Zweck der dedizierte Custom-Datentyp `type="Boolean2VL"` zur

Verfügung, der den Einsatz des UI-Tipps obsolet macht.

Das Attribut `mandatory="true"` im `<ui>`-Tag definiert ein Feld theoretisch als Pflichtfeld. Da dieses Feature vom Solstice-Client derzeit jedoch de facto ignoriert wird, müssen Pflichtfelder zwingend über Formular-Regeln (Rules) abgebildet werden.

Die Client-GUI schneidet bei String-Eingaben standardmäßig überflüssige Leerzeichen am Anfang und Ende des Textes automatisch ab. Für spezifische Formate, bei denen Leerzeichen zwingend erhalten bleiben müssen, kann diese Automatik deaktiviert werden.

```
<ui autotrim="false"/>
```

2.3.4. Datei-Anhänge, Persistenz und DSGVO

Entitäten, deren Daten nur zur Laufzeit aggregiert werden und niemals in der Datenbank gespeichert werden sollen, können von der Persistenz ausgeschlossen werden.

```
<db persistent="false"/>
```

Die Speicherung großer Dateianhänge (BLOBs) wird über das Attribut `streamResource="true"` im `<db>`-Tag aktiviert. Um den Speicherplatzbedarf zu optimieren, kann die automatische Historisierung für diese Binärdaten gezielt abgeschaltet werden.

```
<db streamResource="true"/>
<db noStreamResourceHistory="true" />
```

Spezielle, nur lokal relevante Daten wie Lagerbestände lassen sich über das Attribut `forbidDirectChanges="true"` vor jeglicher manueller Manipulation über den Client schützen.

```
<db forbidDirectChanges="true"/>
```

Wenn `forbidDirectChanges="true"` gesetzt ist, dürfen Änderungen an diesen Objekten ausschließlich durch systeminterne Hintergrundprozesse erfolgen. Solche geschützten Objekte werden vom Framework architektonisch rigoros nicht mehr zwischen den MyTISM-Nodes im

> Cluster synchronisiert!

Die Einhaltung datenschutzrechtlicher Vorgaben (DSGVO) wird zentral im XML-Schema deklariert. Hierfür stellt das Schema dezidierte Tags wie `<GDPRDataCategory>`, `<GDPRBusinessInterest>`, `<GDPRProcessingPurpose>`, `<GDPRProcessingLegalBasis>`, `<GDPRLaw>` und `<GDPRRetentionPurpose>` zur Verfügung. Ein Hintergrunddienst im Applikations-Server berechnet anhand dieser deklarativen Tags vollautomatisch die physischen Löschfristen für die entsprechenden Datensätze im System.

2.4. Architektur, Backend & Deep-Dive

Für Server-Administratoren und Backend-Entwickler bildet das in XML deklarierte Schema die essenzielle Basis der gesamten Datenbank- und Programmarchitektur.

2.4.1. Servereinstellungen und Boot-Verhalten

Die fundamentale Serverkonfiguration erfolgt in der zentralen Datei `mytism.ini` innerhalb des Abschnitts `[DBMan]`.

```
schemaFile=/.demo/schema/schema.xml
```

Bei großen lokalen Entwicklungsdatenbanken verzögern die routinemäßigen Datenprüfungen den Serverstart oft erheblich. Diese systemweiten Integritätsprüfungen lassen sich über spezifische Parameter konfigurativ abschalten, um den Boot-Vorgang massiv zu beschleunigen.

```
noMetaDataCheck=1
noInitialDataCheck=1
noIntegrityCheck=1
noIntegrityDoubleIdChecks=1
noIntegrityBLOBChecks=1
```

Alternativ schließt der Parameter `integrityCheckEntitiesToExcludeFromNToMAndDoubleIdCheck` gezielt einzelne, datenintensive Entitäten von diesen Prüfungen aus.

2.4.2. Scaffolding und abstrakte Entitäten

MyTISM nutzt für die Persistenzschicht das Prinzip des Scaffolding. Aus den Deklarationen

im Schema erzeugt der Build-Prozess vollautomatisch Java-Basisklassen. Diese Klassen kapseln die gesamte Persistenzlogik sowie alle grundlegenden Getter- und Setter-Methoden. Das Framework generiert zusätzlich dedizierte NN-Getter (`get*NN()`), um skalare `NULL`-Werte auf der Datenbankebene typsicher im Java-Code abzufangen.

Entitäten können über das XML-Attribut `abstract="true"` explizit als abstrakte Basisklassen deklariert werden.

Fehlen bei einer als abstrakt deklarierten Entität konkrete Subentitäten im Schema, geben der Schemagenerator und der Server die Warnung `Entity … is "abstract", but has no concrete subentities.` aus.

Der Build bricht mit einer `AttributeDefinitionException` ab, falls ein Attribut in einer abgeleiteten Klasse redundant definiert wird.

Die 3-Tier-Architektur von MyTISM darf nicht mit dem klassischen MVC-Paradigma gleichgesetzt werden. MyTISM stellt architektonisch primär das Model dar, während View und Controller vollständig im jeweiligen Client gekapselt sind. Ein direkter Import von bestehenden, fremden SQL-Schemata ist aufgrund des strikten Generierungsansatzes nicht vorgesehen.

Um redundante Pfadangaben zu vermeiden, wird das Basis-Package für die Code-Generierung global im einleitenden `<Schema>`-Tag definiert.

```
<Schema version="@ProjectName@ Schema built @BUILT@"
defaultPackage="com.klinikum.bo.management">
```

Die Namen der Entitäten müssen aus internen Gründen projektübergreifend zwingend eindeutig bleiben. Es darf in der gesamten Schema-Landschaft nur exakt eine Entität mit einem spezifischen Namen existieren.

2.4.3. Custom-Code und Modulsystem

Für Entitäten mit spezifischer Geschäftslogik muss der Code-Generator angewiesen werden, eine manuell erweiterbare Java-Klasse zu erzeugen.

```
<code custom="true"/>
```

Zusätzliche Funktionsbereiche lassen sich über das Modulsystem einbinden, welches im Schema inklusive des jeweiligen Providers deklariert wird.

```
<ModuleProvider name="oashi" path="/com/oashi"/>
<Module name="core" provider="oashi"/>
```

Die Modularisierung wird auf Code-Ebene durch ein intelligentes Vererbungssystem abgebildet. MyTISM sammelt den Code aus verschiedenen Modulen und baut daraus automatisch eine strikte Vererbungskette aus Aspekt-Klassen auf.

Damit diese Vererbungshierarchie fehlerfrei kompiliert, muss die Custom-Klasse zwingend mit dem Makro `@ENTITY [Klassenname]@` annotiert werden. Fehlt diese Annotation, baut das System am Modulsystem vorbei und korrumpiert die Architektur. Eine Compiler-Meldung „Object cannot be null“ deutet meist auf einen fehlenden leeren Standardkonstruktor hin oder auf die fehlerhafte direkte Instanziierung einer abstrakten Entität.

2.4.4. Relationen und Rückwärtsbeziehungen

Eine zentrale Architekturregel besagt, dass alle Relationstypen (n-1, 1-n, n-m) in MyTISM in der Regel bidirektional definiert sind. Fehlt im XML-Schema die explizite Deklaration einer solchen `<backRelation>`, generiert das System diese Rückrelation jedoch völlig automatisch im Hintergrund.

```
<attr name="Station" type="Station" relation="n-1">
  <backRelation name="ZugehoerigeStationen"
singular="ZugehoerigeStation"/>
</attr>
```

In seltenen Ausnahmefällen, wie massiven Performance-Engpässen in rein unidirektional geplanten Schnittstellen, kann diese Automatik durch das Attribut `ignoreReverseRelations="true"` unterdrückt werden.

2.4.5. Transaktionen, Abfragen und Frapping

Die Datenbankarchitektur von MyTISM verlangt eine strikte Graphenstabilität innerhalb einer laufenden Transaktion. Dieses essenzielle Konzept wird als „Frapping“ bezeichnet. Es besagt, dass eine identische Datenbank-ID innerhalb einer Transaktion immer auf dieselbe Java-Instanz im Arbeitsspeicher verweisen muss. Laden zwei Abfragen innerhalb desselben Vorgangs zufällig dasselbe Objekt, stellt das Frapping sicher, dass Modifikationen synchron

auf exakt derselben Instanz erfolgen.

Backend-Entwickler rufen Daten klassischerweise über `queryBO()` ab, was einen echten OQL-Request an die Datenbank sendet. Sollen speicherschonend zuerst die lokalen Caches geprüft werden, ist die Methode `tx.getBOsByAttrs()` die performantere Wahl.

Beim expliziten Frapping über `tx.frapBOFromCache(bo)` und beim Inkludieren eines BOs in eine Transaktion mutiert das Framework das Objekt nicht, sondern gibt die gefrappte Instanz zurück. Wird dieser Rückgabewert im Code ignoriert, greift das Frapping nicht, was zu schwer auffindbaren Bugs führt.

```
def drei = tx.includeBO(tx.getBO(3))
// KORREKT: Mit dem Rückgabewert weiterarbeiten
drei.geschlecht = tx.frapBOFromCache(cFemale)
```

Temporäre Attribute, die nicht persistiert werden sollen, können dynamisch über die Transient Properties Map an Objekte gehängt werden. Damit diese bei Transaktionsabbrüchen (Rollbacks) korrekt zurückgesetzt werden, müssen zwingend transaktionsgebundene Properties genutzt werden.

2.4.6. Virtuelle Attribute und Arrays im Backend

Bei der Implementierung virtueller Attribute gelten strenge Vorgaben. Methoden, die im Schema definierte virtuelle Attribute bereitstellen, müssen zwingend mit der `@Override` -Annotation versehen werden.

In der Getter-Methode eines virtuellen Attributs darf niemals eine laufende Transaktion (`tx`) vorausgesetzt werden. Da Objekte auch in Lesezeichen ohne aktiven Transaktionskontext instanziiert werden, darf zur Datenbeschaffung hier ausschließlich der BOLoader (`getBOLoader()`) genutzt werden.

Virtuelle Attribute können als lokal zu cachen markiert werden, um performancekritische Berechnungen zu minimieren. Dabei muss die zugehörige Getter-Methode zwingend mit einem führenden Unterstrich benannt werden. Da Arrays in Java stets veränderlich sind, dürfen sie bei der Nutzung von gecachten virtuellen Attributen niemals direkt als Referenz aus dem Getter zurückgegeben werden.

2.4.7. Lifecycle-Hooks und Code-Regeln

Der Lebenszyklus eines Business Objects durchläuft vor dem Speichern spezifische Hooks

des Interfaces `de.ipcon.db.core.SaveAwareI`. Die Methode `verifyOnClient()` dient ausschließlich lokalen Plausibilitätsprüfungen auf dem Client, ohne teure Datenbankzugriffe auszulösen. Serverseitig wird zunächst `beforeVerifyOnServer()` für sehr schnelle Datenmanipulationen aufgerufen.

Die eigentlichen Datenprüfungen unter Einbezug der Datenbank erfolgen strikt in der Methode `verifyOnServer()`. Ausschließlich hier darf bei erkannten Integritätsfehlern eine `SaveVetoException` geworfen werden. Haben alle Instanzen das Speichern genehmigt, stößt das System abschließend `afterVerifyOnServer()` für asynchrone Nachberechnungen an, wobei hier keine Werte mehr am Objekt verändert werden dürfen.

Ein kritischer Randfall sind dezentrale Systeme. Steht `tx.isSyncMode()` auf `true`, wird die Transaktion gerade über das Netzwerk synchronisiert. In diesem Fall dürfen im Hook `verifyOnServer()` absolut keine Seiteneffekte auftreten, Daten verändert oder Exceptions geworfen werden.

Um verschachtelte Code-Blöcke zu vermeiden, sollte jeder Hook mit einem ressourcenschonenden „Early Exit“ beginnen.

```
@Override
method verifyOnServer(nodeNumber = Long, user = Benutzer, tx =
Transaction)
   super.verifyOnServer(nodeNumber, user, tx)
   if tx.isDeletedOrNotInvolved(this) then
      return
```

Das Überschreiben von `isReadOnly`-Methoden erfordert hohe Disziplin. Trifft die eigene logische Bedingung für den Leseschutz nicht zu, muss zwingend die Super-Klasse über `super.isReadOnly(a)` aufgerufen werden.

```
import de.ipcon.schema.AttributeI

@ENTITY Person@

@Override
method isReadOnly(a = AttributeI) returns boolean
   if not getAktivNN() and a.getName() <> "Aktiv" then
      return 1
```

```
    return super.isReadOnly(a)
```

Beim Überschreiben von Settern muss stets als erster Befehl der Wert über `super.setXYZ(xyz)` gesetzt werden, bevor die eigene Logik greift. Getter von persistenten Attributen sollten nicht manuell überschrieben werden und dürfen keinesfalls Exceptions werfen, da dies die Benutzeroberfläche sofort zum Absturz bringt.

Beim endgültigen Entfernen von Objekten aus einer n-m-Relation darf niemals z. B. `setMedikamente(null)` aufgerufen werden. Es muss zwingend über den Iterator der Relation iteriert und explizit `removeMedikament(m)` aufgerufen werden.

Ein Objekt-Identitätsvergleich für boolesche Werte darf niemals über Konstrukte wie `== Boolean.TRUE` erfolgen; stattdessen ist konsequent `.booleanValue()` zu nutzen.

Innerhalb von Konstruktoren dürfen niemals asynchrone Threads gestartet werden, da hier auftretende Fehler stillschweigend verschluckt werden.

2.4.8. System-Updates und Unit-Testing

Verändert sich das Datenmodell derart, dass MyTISM es nicht automatisch anpassen kann, müssen Updates zwingend über versionierte Skripte verteilt werden. Der Update-Prozess durchläuft zwei isolierte Phasen. Die Methode `runUpdateScriptsStage1()` führt native SQL-Skripte aus, bis sie auf das erste `.orm`-Skript stößt. Erst danach führt `runUpdateScriptsStage2()` die objektrelationalen `.orm`-Skripte aus, welche regulär neue Datensätze innerhalb einer Transaktion anlegen können.

Die Ausführung der Stage 2 passiert ausschließlich auf dem autoritativen Hauptserver. Auf dezentralen Sync-Knoten werden `.orm`-Skripte lediglich protokolliert, aber niemals ausgeführt. Setzen native SQL-Skripte auf Daten auf, die zuvor in `.orm`-Skripten erzeugt wurden, stürzt der Sync-Knoten ab.

Die API `UpdateHandlerTools` bietet wichtige Hilfsmethoden für das Schreiben dieser Skripte.

```
import de.ipcon.db.update.UpdateHandlerTools

if (UpdateHandlerTools.checkTableExists('ezaehler') \
     && checkColumnExists(table: 'zaehler', column:
'spannungsebene')) {
  log.info('Starting moving column spannungsebene from zaehler to
ezaehler.')
```

```
  stmt.executeUpdate("ALTER TABLE ezaehler add column
spannungsebene_bkup bigint")
  UpdateHandlerTools.dropColumn('zaehler', 'spannungsebene',
stmt)
}
```

Wird die Update-API in `@CompileStatic`-annotierten Groovy-Skripten genutzt, dürfen keine benannten Parameter übergeben werden. Alle Aufrufe müssen streng positioniert erfolgen, wie beispielsweise `UpdateHandlerTools.checkTableExists('bo', stmt)`.

Die Performance bricht massiv ein, wenn ein ORM-Skript neue Tabellen anlegt, Daten migriert und sofort riesige OQL-Abfragen absetzt, da die erforderlichen Datenbank-Indizes erst nach einem weiteren Serverneustart greifen.

Für das Schreiben von Unit-Tests für komplexe Transaktionen bietet das Framework mächtige Mocking-Möglichkeiten über Interceptoren an.

```
protected void setUp() {
   tx = bol.createTransaction()
   def queryArztbesuche = "Arztbesuch a where not Ldel and
BehandelnderArzt.Name = $1"
   bol.addQueryInterceptor(queryArztbesuche,
queryInterceptorArztbesuche as TestQueryInterceptorI)
}
```

Chapter 3. Lesezeichen und Datenabfragen (OQL)

3.1. Grundlagen & Konzepte

Lesezeichen (Bookmarks) bilden für Anwender den zentralen Einstiegspunkt in das MyTISM-Ökosystem. Es handelt sich um systemweit gespeicherte, vordefinierte Ansichten, die strukturierte Daten in der Benutzeroberfläche als Listen oder Tabellen darstellen. In unserem fiktiven Krankenhaus-Projekt könnte ein Lesezeichen beispielsweise als Startbildschirm dienen und alle Patienten auflisten, die aktuell auf einer bestimmten Station aufgenommen sind. Über interaktive Filter am Kopf dieser Tabellen lassen sich die Datenmengen weiter einschränken, für Massenänderungen markieren oder zur externen Weiterverarbeitung exportieren. Für Ad-hoc-Auswertungen lässt sich jedes Lesezeichen in einen interaktiven Pivot-Modus umwandeln. So kann visualisiert werden, wie sich die Altersstruktur der Patienten über die verschiedenen Stationen hinweg verteilt.

Die technische Basis dieser Lesezeichen bildet die Object Query Language (OQL). OQL ist eine proprietäre Abfragesprache, die als Übersetzungsschicht zwischen dem Anwender und der relationalen Datenbank fungiert. Der größte architektonische Vorteil liegt in der vollständigen Abstraktion des logischen Datenmodells. Wer nach einem Patienten sucht, muss nicht wissen, in welchen verschachtelten Datenbanktabellen dessen Adressdaten, Notfallkontakte oder Diagnosen physisch abgelegt sind. Das System fragt lediglich das übergeordnete Objekt „Patient“ ab, woraufhin OQL diese Anfrage vollautomatisch in SQL-Befehle übersetzt. Diese Architektur kapselt die Persistenzschicht ab und bewahrt Nutzer vor dem manuellen Schreiben von SQL-Code.

Ergänzend zu den strukturierten OQL-Abfragen bietet MyTISM eine integrierte Volltextsuche (FTS). Diese ermöglicht schnelle, textuelle Suchen über alle passenden Attribute der Entität, die im Lesezeichen dargestellt wird. Bei der Eingabe eines Begriffs in das Suchfeld eines Lesezeichens durchsucht das System den serverseitig erstellten Index. Um diesen Suchindex performant zu halten, sind technische oder nicht-textuelle Daten – wie boolesche Werte, Zahlen, Datumsangaben (und natürlich rein virtuelle Attribute) – standardmäßig von der Indexierung ausgeschlossen. Das Zusammenspiel aus strukturierten OQL-Abfragen und der Freitextsuche macht das Auffinden von Informationen hocheffizient.

3.2. Benutzeroberfläche & Bedienung

Die Interaktion mit Datenbeständen beginnt in der Benutzeroberfläche von Solstice primär in der Suchleiste eines Lesezeichens. Power-User können hier ihr Wissen über das Datenmodell nutzen, um Ad-hoc-OQL-Klauseln direkt in der GUI einzugeben, anstatt nur

die Volltextsuche oder vorgefertigte Filterkomponenten zu verwenden. Hierfür muss die Eingabe im Suchfeld zwingend mit einer öffnenden eckigen Klammer `[` eingeleitet werden. Das System bietet hierbei eine Autovervollständigung: Tippt der Nutzer beispielsweise `[ a.` und drückt die Tastenkombination „Strg + Leertaste“, erscheint eine alphabetische Liste aller verfügbaren Attributnamen.

Diese strukturierte OQL-Suche lässt sich nahtlos mit der klassischen Volltextsuche kombinieren. Wird die OQL-Abfrage mit einer schließenden eckigen Klammer `]` beendet, interpretiert das System den restlichen Text automatisch als Volltext-Suchbegriff und verknüpft beide Bedingungen zwingend mit einem logischen `AND`. Ein Eingabebeispiel für diese kombinierte Suche lautet: `[ a.Name = 'Sara' ] Grippe`. Im Kontext des Krankenhauses sucht diese Eingabe strukturiert nach einer Patientenakte mit dem exakten Namen „Sara“, während im restlichen Volltext das Wort „Grippe“ vorkommen muss.

Für gefilterte Ergebnismengen stehen in der Software dedizierte Tastenkürzel zur schnellen Datenanalyse zur Verfügung. Mit „Alt-P“ lässt sich die aktuelle Liste direkt in den interaktiven Pivot-Modus umstellen, um Daten visuell zu aggregieren. Für die Weiterverarbeitung in externen Programmen exportiert „Alt-E“ die angezeigten Daten als CSV, während „Alt-X“ einen formatierten Excel-Export generiert.

3.3. Erweiterte Konfiguration (XML)

Die Konfiguration von Lesezeichen erfolgt durch Administratoren über deren XML-Parameter. Auf der Ebene der strukturellen Elemente lassen sich Lesezeichen über dedizierte Attribute im `<Table>`-Tag detailliert steuern. Das Attribut `explicitStart="true"` verhindert, dass große Datenmengen sofort beim initialen Öffnen eines Lesezeichens geladen werden; die Abfrage erfolgt erst nach manuellem Bestätigen durch die Enter-Taste oder F5. Soll die Abfrage hingegen sofort ausgeführt werden, erzwingt `loadImmediate="true"` das Laden der Liste ohne weitere Benutzerinteraktion. Ein festes Limit für die maximale Anzeige von Objekten in der Tabelle wird über `maxRows="100000"` definiert. Das Attribut `showDeleted="true"` bewirkt, dass auch Datensätze im Lesezeichen angezeigt werden, die über das `Ldel`-Flag als gelöscht markiert sind.

Zusätzlich zum Ladeverhalten lässt sich das Interaktionsverhalten der Tabelle anpassen. Ein Doppelklick auf einen Listeneintrag öffnet standardmäßig den dazugehörigen Datensatz. Soll bei einem Doppelklick stattdessen direkt ein verknüpftes Objekt geöffnet werden, leitet das Attribut `openProperty="Patient"` im `<Table>`-Tag dieses Verhalten auf die gewünschte Relation um. Soll zwingend ein abweichendes Formular erzwungen werden, wird das Standardverhalten mittels Groovy-Skripten überschrieben. Hierfür wird die Standard-Aktion innerhalb des `<Table>`-Elements durch `<Action cmd="openSelected" merge="true">` ersetzt. Das dazugehörige Skript im `<onAction>` -Tag greift zunächst über `ftx['tblMain'].getFirstSelectedObject()` auf das in der

Tabelle markierte Objekt zu. Anschließend ermittelt es die Entität der Relation und erzwingt das Öffnen über die API-Methode `ctx.openForm()`. Dieses Vorgehen umgeht die standardmäßige Formular-Priorisierung und garantiert, dass Nutzer exakt die vorgesehene Maske sehen.

```
<Action cmd="openSelected" merge="true">
  <onAction language="groovy"><![CDATA[
    // 1. Die aktuell ausgewählte Verordnung aus der Tabelle
holen
    def verordnung = ftx['tblMain'].getFirstSelectedObject()

    if (verordnung.Patient == null) {
       ftx.toast('Kein Patient verknüpft.')
       return
    }

    // 2. Entität der verknüpften Patientenakte ermitteln
    def ent = ctx.schema.getEntityForObject(verordnung.Patient)

    // 3. Das gewünschte Formular suchen (hier das
Standardformular)
    def form = ctx.getForms(ent).find()

    // 4. Den Patienten explizit mit dem gewählten Formular
öffnen
    if (form != null) {
       ctx.openForm(form, verordnung.Patient)
    } else {
       ftx.toast('Kein passendes Formular gefunden.')
    }
  ]]></onAction>
</Action>
```

Den strukturellen Rahmen für die eigentliche Datenbankabfrage bildet das `<Query>` -Element. Das Setzen von `type="Text"` generiert vollautomatisch das notwendige SQL-Präfix der Aggregatsfunktion für das jeweilige Basis-Objekt. Für völlig freie Abfragen muss `type="Free"` oder `type="Raw"` gewählt und die vollständige OQL-Query im XML vorgegeben werden. Um die Volltextsuche über den Standard hinaus auszudehnen und Attribute von lediglich indirekt verknüpften Objekten in die Lesezeichen-Suche

einzubeziehen, wird das Tag `<addProperty>` verwendet.

```
<Query type="Text">
  <addProperty>Arzt.Name</addProperty>
  <addProperty>Patient.Person.Name</addProperty>
</Query>
```

Die dynamische Filterung der Ergebnismenge wird über `<filter>`-Tags gesteuert, die in der Benutzeroberfläche durch Trennlinien gruppiert werden können.

```
<separator text="Filterkategorie #1"
icon="/20x20png/ForwardAll.png" fontSize="+10%" fontStyle="BOLD"
gradientStartColor="180 180 240"/>
```

Feste Filter, die permanent im Hintergrund greifen und in der GUI nicht sichtbar sind, werden konsequent ohne das `type`-Attribut definiert. Ein XML-Beispiel für einen Filter, der nur Datensätze anzeigt, die jünger als einen Monat sind, sieht wie folgt aus:

```
<filter><![CDATA[ age(Crea) < "1 month" ]]></filter>
```

Bei regulären, für den Endanwender sichtbaren Filtern werden die dynamischen Benutzereingaben im verpflichtenden Kindelement `<clause>` stets durch zwei geschweifte Klammern referenziert und so in die OQL-Query injiziert.

```
<filter type="string" title="Dokumentnummer" cols="30">
  <clause>Dokumentnummer = "{}"</clause>
</filter>
```

Wird ein Filter in der GUI leer gelassen, injiziert das Tag `<ifEmpty>` eine automatische Fallback-Klausel, um beispielsweise standardmäßig nur offene Datensätze anzuzeigen. Soll ein Filter standardmäßig gelöschte Datensätze ausblenden, wird hierfür klassischerweise die Klausel `<ifEmpty>NOT Ldel</ifEmpty>` genutzt. Boolesche Filter vom Typ `type="bool"` erscheinen in der Software als Checkboxen, die bei der Konfiguration zwingend alle drei logischen Zustände (`<ifTrue>`, `<ifFalse>`, `<ifNull>`) explizit abarbeiten müssen. Das XML-Beispiel für einen solchen Filter lautet:

```
<filter type="bool" title="nur männlich">
```

```
  <ifTrue>Geschlecht.Tid = "MAENNLICH"</ifTrue>
  <ifFalse>Geschlecht.Tid = "WEIBLICH" or Geschlecht.Tid =
"NA"</ifFalse>
  <ifNull>Geschlecht = null</ifNull>
</filter>
```

Spezifische Datumsfilter (`type="date"`) können über das Attribut `strictFormat` strenge Eingabeformate erzwingen. Wird zusätzlich das Attribut `replace="true"` gesetzt, formatiert die GUI abweichende Datumseingaben automatisch in das erforderliche Zielformat um. Eine noch weitreichendere Vorverarbeitung von Eingaben erlaubt das Tag `<inputPreprocessor>`. Mittels Groovy-Skripten im XML kann die Systemvariable `input` beliebig manipuliert werden, bevor sie endgültig an die OQL-Engine übergeben wird. Ein Beispiel zur Umwandlung einer kommagetrennten Liste von IDs in ein OQL-konformes Format lautet:

```
<inputPreprocessor>input.split(',').collect{ "'${it.trim()}'"
}.join(',')</inputPreprocessor>
```

Für hierarchische Abhängigkeiten nutzt MyTISM dynamische Auswahllisten. Diese Multiple-Choice-Filter beziehen ihre Werte nicht aus statischen Listen, sondern zur Laufzeit via `<choiceQuery>` direkt aus der Datenbank. Ein untergeordneter Filter kann durch das Attribut `dependsOn="NameDesErstenFilters"` vom zuvor ausgewählten Wert des Elter-Filters abhängig gemacht werden. Die vom Nutzer referenzierte ID wird dabei über Platzhalter wie `{Id}` oder `{Filtername}` in die abhängige Abfrage injiziert.

```
<filter type="multipleChoice" title="$R{Wirkstoff}"
dependsOn="Medikamentengruppe">
  <choiceQuery query="Wirkstoff bo WHERE Not Ldel ORDER BY Name"
               dependsOnQuery="Wirkstoff bo WHERE Not Ldel AND
Medikamentengruppe.Id = {Medikamentengruppe} ORDER BY Name">
    Wirkstoff = {Id}
  </choiceQuery>
</filter>
```

Bei einer leeren Auswahl des Benutzers in abhängigen Filtern (`dependsOnQuery`) setzt das System derzeit strikt den Text `NULL` als fixen String ein. Es gibt in MyTISM derzeit keine native Mechanik, um abweichende Klauseln bei nicht gewählten Werten automatisch zu

generieren. Als Workaround muss manuell eine ODER-Klausel in die Abfrage integriert werden (z. B. `or '{Medikamentengruppe}' = 'NULL'`), um Fehler bei der Evaluierung zu vermeiden.

Komplexe OQL-Szenarien erfordern oftmals, dass Filter bestimmten Bedingungsgruppen zugewiesen werden, was über das Attribut `group` definiert wird. Diese Zuweisung ist essenziell für `<template>`-Konstruktionen mit Makros wie `{UnionAll}`, wenn für unterschiedliche Subentitäten abweichende Attribute durchsucht und die Bedingungen isoliert angewendet werden müssen.

Sollen in einem Lesezeichen exklusive Attribute aus unterschiedlichen Subentitäten gefiltert werden, schlägt der reguläre OQL-Parser fehl. Dem Filter-Tag muss in diesem Fall eine Dummy-Klausel (z. B. `<clause>false</clause>`) mitgegeben werden, während die echte Filterlogik im `<template>`-Tag via `{Union}`-MEX-Makros abgehandelt wird.

Wird in Abfragen nach Objekten mit einem bestimmten Software-Interface gesucht, nutzt das System im Hintergrund oft das Makro `WithInterface`. Um zu verhindern, dass zukünftig neu programmierte Untertypen automatisch aus der GUI ausgeblendet werden, muss dem entsprechenden Filter zwingend das Attribut `excludeOtherInterfaces="false"` angefügt werden.

Hinweis für Power-User und Entwickler: Eine vollständige Dokumentation aller Mengenoperatoren und Backend-Schnittstellen finden Sie im Kapitel „OQL-Referenzhandbuch“.

Chapter 4. Benutzeroberfläche & Formularengine (de.ipcon.form)

4.1. Grundlagen & Konzepte

Der Solstice-Client bildet als native Java-Swing-Applikation das visuelle Herzstück des Systems. Die Benutzeroberfläche ist kompromisslos auf eine performante und fehlerfreie Datenerfassung im professionellen Arbeitsalltag ausgelegt. Um maximale Effizienz zu gewährleisten, lässt sich das gesamte System vollständig über die Tastatur bedienen. Umfangreiche Tastenkürzel (Shortcuts) beschleunigen den Arbeitsfluss von erfahrenen Anwendern signifikant. Zu den wichtigsten systemweiten Hotkeys zählen `F2` für das schnelle Speichern und `F3` für das Speichern mit anschließendem Schließen der Maske. Mit `ESC` lässt sich ein Vorgang sofort abbrechen und die Maske schließen, wobei bei ungespeicherten Änderungen stets eine Sicherheitsabfrage erscheint. Die Taste `F4` öffnet Auswahllisten und Popups, `F5` aktualisiert Tabellen oder Lesezeichen, und `STRG+F` aktiviert die globale Suche.

Die Oberfläche gliedert sich primär in einen hierarchischen Navigationsbaum auf der linken Seite und einen flexiblen Hauptarbeitsbereich auf der rechten Seite. Der Client unterstützt nativ einen Mehrfachfenstermodus. Dadurch können Anwender problemlos mehrere Datensätze, wie beispielsweise zwei unterschiedliche Patientenakten, parallel nebeneinander vergleichen. Zudem ist die Oberfläche barrierefrei und ergonomisch konzipiert, sodass alle Funktionen auch bei visuellen Einschränkungen optimal erfassbar bleiben.

4.1.1. Was sind Strukturelemente?

„Strukturelemente“ ist der systemweite Oberbegriff für alle UI-Elemente und Masken, mit denen Anwender Daten anzeigen, suchen oder manipulieren. Während reguläre Anwender diese Masken im Alltag lediglich bedienen, können Administratoren sie im Hintergrund über Konfigurationsdateien versionieren und detailliert anpassen. Das Framework unterscheidet dabei fünf primäre Typen von Strukturelementen:

- **Lesezeichen (Bookmarks):** Dienen der strukturierten Tabellen- oder Listenansicht von Datenmengen, wie beispielsweise einem durchsuchbaren Medikamentenkatalog. Sie verbergen standardmäßig alle Datensätze, die im System durch das Flag `ldel` als gelöscht markiert wurden.
- **Formulare:** Die eigentlichen Eingabemasken für die Detailansicht und Bearbeitung. Sie definieren die exakte Anordnung der UI-Komponenten, von simplen Textfeldern bis hin zu komplexen Kalendern und Baumansichten.
- **Schablonen (Templates):** Agieren als Blaupausen für die Neuanlage von Datensätzen.

Sie definieren zwingend den Typ des zu erzeugenden Objekts, rufen das zugehörige Formular auf und belegen definierte Standardwerte vor.

- **Reports:** Transformieren Daten aus der Datenbank in druckbare Formate, beispielsweise für die Generierung von Arztbriefen oder Barcode-Etiketten. Die technische Dokumentation hierzu befindet sich im separaten Kapitel zur Reporting-Engine.
- **Codebausteine:** Reine Entwickler-Elemente, die als wiederverwendbare Fragmente arbeiten, um redundanten Code zu vermeiden. Ein modularer Baustein wie die „Adresse“ kann so völlig identisch in verschiedene Hauptformulare eingebunden werden.

Neben individuell erstellten Masken existieren vordefinierte Basis-Elemente, die tief im Systemkern verankert sind. Diese tragen in der Benutzeroberfläche üblicherweise den Namenszusatz „(Vorgebaut)“, um sie als unveränderliche System-Elemente zu kennzeichnen. Im Gegensatz zu regulären Objekten können diese Elemente im Navigationsbaum weder verschoben noch verlinkt werden. Für eigene Anpassungen müssen diese System-Elemente zwingend zuvor manuell kopiert werden.

4.1.2. Kontext und Formularauswahl

Die Formularauswahl in MyTISM arbeitet strikt kontextbezogen. Für ein und dasselbe Datenbank-Objekt können abhängig vom jeweiligen Anwendungsfall oder der Benutzerrolle völlig unterschiedliche Formulare existieren. Öffnet ein Anwender einen Datensatz, ermittelt die Engine das passende Formular anhand einer fest definierten Regelhierarchie:

1. Prüfung der generellen Verfügbarkeit des Formulars für die spezifische Benutzergruppe.
2. Auswertung des höchsten zugewiesenen Prioritäts-Wertes.
3. Ermittlung des passgenauesten BO-Typs innerhalb der Vererbungshierarchie.
4. Alphabetische Sortierung nach dem Formularnamen.
5. Fallback auf die interne ID des Formulars.

4.2. Benutzeroberfläche & Bedienung

Power-User können die Benutzeroberfläche auch ohne Programmierkenntnisse weitreichend an die eigenen operativen Prozesse anpassen.

4.2.1. Navigationsbaum und Aliase

Der Navigationsbaum auf der linken Seite ist das zentrale Steuerungselement des Clients.

Über das Tastenkürzel `STRG+L` können Power-User hier sehr schnell sogenannte Aliase von bestehenden Einträgen erzeugen, um individuelle Arbeitsbereiche zusammenzustellen.

Bei diesen Aliasen handelt es sich ausschließlich um Verknüpfungen und nicht um eigenständige Kopien. Jede Änderung an einem über einen Alias geöffneten Strukturelement überschreibt sofort das Original, was bei unbedachter Nutzung systemweite Masken korrumpieren kann.

4.2.2. Globale Parameter und Variablen

Systemweite Vorgaben für die Benutzeroberfläche können direkt im Navigationsbaum unter „Admins/Mytism/Benutzerverwaltung/Variablen“ gesteuert werden. Sollen beispielsweise Tooltips in Tabellen systemweit deaktiviert werden, wird der Wert der Variable `tables.showTooltips` angepasst. Diese globalen Parameter lassen sich präzise für einzelne Benutzer oder ganze Benutzergruppen überschreiben. Um bei einer großen Anzahl an Benutzern den Ordner „Alle Benutzer“ übersichtlich in alphabetische Unterordner zu gruppieren, können die Variablen `users.view.group.minElements` und `users.view.group.maxElements` beispielsweise auf den Wert 5 gesetzt werden.

4.3. Erweiterte Konfiguration (XML)

Für tiefgreifende Anpassungen des Formular-Layouts nutzen Administratoren ein XML-basiertes Markup-Konzept.

4.3.1. Struktur-Synchronisation und Dateisystem

Strukturelemente können über das Tool „Struktur-Synchronisation“ (oder „DateiSystemSync“) als XML-Dateien in das lokale Dateisystem exportiert werden. Der

Aufruf dieses Tools erfolgt im Client über den Menüpunkt „Administration → System-Objekte syncen“. Dieser XML-Export ist essenziell, da er eine professionelle Versionskontrolle der gesamten Oberfläche ermöglicht.

Damit ein Strukturelement exportiert werden kann, muss in seinen Eigenschaften zwingend ein Dateiname (bzw. relativer Pfad) hinterlegt sein. Die Schaltfläche „Dateiname vorschlagen“ generiert dabei automatisch einen passenden Namen, der die Ordnerstruktur des Elterpfads logisch widerspiegelt.

In der Sync-GUI stehen Administratoren verschiedene Aktionen zur Verfügung. Die Liste der Elemente lässt sich filtern und verschiedene Detailstufen für Log-Meldungen sind zuschaltbar. Die Aktion „Vergleichen“ aktualisiert die Liste und gleicht den Stand zwischen der Datenbank und dem Dateisystem für einen manuellen Überblick ab. Die Aktion „Alles synchronisieren“ importiert oder exportiert automatisch alle Elemente basierend auf ihrem Status und speichert die finalen Änderungen in der Datenbank.

Dies ermöglicht zwei klassische Deployment-Workflows. Beim Export passen Administratoren bestehende Elemente direkt im Live-System an und exportieren diese in das lokale Dateisystem, um die neue Version in der Versionskontrolle zu sichern. Beim Import entwerfen Entwickler neue Masken auf einem separaten Testsystem und übertragen die XML-Dateien anschließend auf den Live-Server. Die Synchronisations-GUI erkennt diese als neue oder geänderte Dateien und importiert die neue Version sicher in die Live-Datenbank. Eine Funktion „Sync automatisch durchführen“ zur selbsttätigen Überwachung existiert zwar, gilt jedoch als fehleranfällig und wird im professionellen Betrieb selten genutzt.

Damit der System-Scanner die exportierten Dateien korrekt als Strukturelemente erkennt, gelten zwingende Suffix-Regeln für Dateinamen. Diese lauten `.frm.xml` für Formulare, `.bkm.xml` für Lesezeichen, `.tpl.xml` für Schablonen, `.bst.xml` für Codebausteine und `.als.xml` für Aliase. Eine technische Sonderrolle nehmen Reports ein, die im Rahmen der Synchronisation separat in zwei Dateien verwaltet werden.

Jede dieser XML-Dateien besteht aus einem Wurzel-Element, das dem Typ des Strukturelements entspricht, also beispielsweise `<Formular>` oder `<Schablone>`. Dieses Wurzel-Element besitzt essenzielle Attribute zur Systemregistrierung. Das Attribut `Name` definiert die Beschriftung in der GUI. Das Attribut `ElterPfad` definiert den logischen Ordner-Pfad im Navigationsbaum. Fehlen Ordner in diesem Pfad, legt MyTISM sie beim Import automatisch an. Das Attribut `Prioritaet` bestimmt, welches Element standardmäßig geöffnet wird, falls für einen Datensatz mehrere Masken existieren. Der Standardwert des Systems ist hierbei meist `-50`. Zuletzt gibt es das Attribut `Tid` für die technische ID, die jedoch vom System automatisch vergeben wird und manuell niemals gesetzt werden sollte.

Unterhalb des Wurzel-Elements definieren spezifische Kind-Elemente den Inhalt. Das Tag `<Beschreibung>` dient als interne Notiz für Entwickler und wird Endanwendern nicht angezeigt. Das Tag `<BOTyp>` legt fest, an welche Entität dieses Strukturelement gebunden ist. Das Tag `<Parameter>` bildet den Kernbereich, in dem das detaillierte Layout des Formulars oder die Abfrage des Lesezeichens hinterlegt wird. Zuletzt steuert das Tag `<Gruppen>`, welche Benutzergruppen das Element überhaupt sehen dürfen.

Dieses Tag wird bei vorgebauten Elementen aus dem Dateisystem beim Einlesen oft vom System ignoriert. Vorgebaute Elemente werden standardmäßig und sicherheitshalber ohnehin nur der Administratoren-Gruppe zugewiesen.

Strukturelemente, die direkt vom Schema-Generator erstellt oder als vordefinierte Standardelemente angelegt wurden, besitzen das interne Flag `IstAutomatik=true`. Vordefinierte Standardelemente sind im Live-Betrieb als absolut unveränderlich zu betrachten. Verändert ein Benutzer

> ein solches Formular in der GUI, wird es beim nächsten Update unwiderruflich vom System überschrieben.

Neben echten persistierten Daten existiert die virtuelle Entität „BX“, für die spezielle BX-Formulare angelegt werden. Sie werden primär genutzt, um komplexe Suchmasken oder interaktive Dashboards abzubilden. Diese BX-Objekte halten lediglich temporäre Werte zur Ablaufsteuerung oder zur Ein- und Ausgabe im Arbeitsspeicher, die niemals fest in der Datenbank gespeichert werden.

4.3.2. Formular-Layouts und Widgets

Im `<Parameter>`-Bereich eines Formulars (`.frm.xml`) wird das visuelle Layout mit hochspezialisierten UI-Tags aufgebaut. Das grundlegende Layout-Konzept basiert auf einem intern entwickelten Java-GridLayout.

> Dieser Abschnitt bietet einen konzeptionellen Überblick über die wichtigsten Bausteine der XML-Formularengine. Die vollständige API-Dokumentation aller verfügbaren XML-Tags, Attribute und Groovy-Bindings finden Sie im separaten Handbuch-Abschnitt „Referenz: XML-Formularelemente & Widgets“ in TEIL 5.

Die exakte Positionierung der Elemente kann optional explizit über die Attribute `e-x` und `e-y` gesteuert werden, wobei das Präfix „e-“ für „Element“ steht. Um beispielsweise zwei Felder auf derselben horizontalen Zeile zu platzieren, wird `e-y="same"` genutzt. Der Umbruch auf die nächste Zeile ist das Standardverhalten, kann aber auch explizit mit `e-y="next"` erzwungen werden.

Die Breite und Höhe von Feldern steuern Administratoren über die Attribute `minSize` und `prefSize`. Die Maßeinheit „c“ steht hierbei für die Zeichenbreite und ist die zu favorisierende Vorgabe. Die Syntax erwartet dabei stets ein Tupel aus horizontaler und vertikaler Ausdehnung.

Zusammengehörige Felder lassen sich zur besseren Strukturierung durch das Tag `<Border>` optisch einrahmen. Dieses Element sollte jedoch sparsam eingesetzt werden, da zu viele ineinander verschachtelte Rahmen das Formular optisch stark belasten. Der gezielte Einsatz von simplen `<Label/>`-Tags für Zwischenüberschriften führt in komplexen Masken oftmals zu einem ruhigeren UI-Design. Das Container-Element `<View>` dient hingegen als unsichtbarer Basis-Container, der Kind-Elemente in einem Raster gruppiert.

```
<Border etched="true" title="Stammdaten">
  <View columns="2" externalVGap="8"> ... </View>
</Border>
```

Für dynamische Masken steht das Tag `<visibleIf>` zur Verfügung. Damit lassen sich ganze Formularbereiche zur Laufzeit über Groovy-Skripte ein- oder ausblenden. Dies kann mit einem `<ToggleButton>` kombiniert werden, um auf- und zuklappbare Bereiche zu konstruieren. Wechselt der Nutzer den Button, wertet ein anschließendes Skript die Sichtbarkeit neu aus und klappt den definierten Bereich auf.

Für die eigentliche Dateneingabe bietet MyTISM einen umfassenden Katalog an Widgets. Das Standard-Eingabefeld für Strings oder Zahlen ist `<Text>`. Es lässt sich über `selectAllWhenFocused="true"` so konfigurieren, dass der gesamte Text beim Fokussieren markiert wird. Über `lineWrap="true"` wird ein weicher Zeilenumbruch erzwungen. Sollen in der Akte mehrzeilige Notizen gerendert werden, muss zwingend `class="ITextArea"` deklariert werden. Für strukturierte Inhalte oder Log-Ausgaben steht das `<Editor>`-Tag mit Syntax-Highlighting bereit. Das Tag `<StyledText>` fungiert als Rich-Text-Editor für formatierte Fließtexte. Dieser Editor lässt sich über `onlyTextFormattingActions="true"` strikt auf reine Textformatierungen limitieren. Das simple `<Label>` dient der reinen Anzeigebeschriftung.

```
<Label text="Wichtiger Hinweis:" fontSize="+100%"
fontStyle="bold" />
<Text property="Name" fontSize="+50%" fontStyle="italics" />
```

Für boolesche Werte steht das Tag `<CheckBox>` zur Verfügung. Dieses Widget kann über `triState="true"` auch drei Zustände annehmen. Eine visuelle Alternative ist der `<ToggleButton>`, der als Knopf zwischen zwei Zuständen wechselt und über `falseText` beziehungsweise `trueIcon` dynamisch sein Aussehen ändert. Das Tag `<ComboBox>` erzeugt ein klassisches Dropdown-Menü. Dieses verbietet über `chooseOnly="true"` Freitext-Eingaben und erlaubt über `nullable="true"` leere Auswahlen. Spezifisch für Zeit- und Datumsangaben existiert der `<DateChooser>`. Er rendert ein Kalender-Icon zur Datumsauswahl, dessen Formatierung über das Attribut `format` exakt gesteuert wird. Sollen stattdessen Zeitdauern erfasst werden, kombiniert der `<SimpleTimespanChooser>` ein Zahlenfeld mit einem Dropdown für die jeweilige Zeiteinheit.

```
<DateChooser property="Geburtsdatum" format="MEDIUM_"
editable="false"/>
<SimpleTimespanChooser property="Wartezeit"
defaultUnit="minutes"/>
```

Für die Einbindung relationaler Datenstrukturen existieren hochspezialisierte Widgets. Das Tag `<Table>` zeigt 1:n-Relationen direkt als eingebettete Liste im Formular an. Soll ein vorhandenes Datenbankobjekt verknüpft werden, rendert `<Popup>` ein Suchfeld inklusive

Such- und Neuanlage-Icons. Für hierarchische Strukturen eignet sich das `<Tree>`-Tag hervorragend. Für Ressourcenplanungen existiert das komplexe Gantt-Element `<Scheduler>`.

```
<Popup property="ZustaendigerArzt" popupSize="800, 15c">
   <Table columns="Name | Abteilung"/>
</Popup>
<Tree entity="Station" childrenProperty="Zimmer"
parentProperty="Elter"/>
```

Das Verhalten all dieser Eingabefelder lässt sich detailliert und dynamisch anpassen. Das Setzen des Flags `disabled="true"` verhindert das Editieren und graut das Feld optisch permanent aus. Das Flag `editable="false"` verhindert das Editieren ebenfalls, belässt das Feld jedoch in seiner regulären optischen Darstellung. Um Formatierungen auf reinen Anzeigetexten zu nutzen, imitiert `disguiseAsLabel="true"` das Styling eines klassischen Labels. Für komplexe Geschäftslogiken lassen sich Felder mit den Skript-Tags `editableIf` und `alsoMandatoryIf` zur Laufzeit steuern. So kann beispielsweise ein Feld dynamisch zum Pflichtfeld gemacht werden, sobald spezifische Bedingungen eintreten.

```
<Text property="Begruendung">
  <!-- Wird beim Aktualisieren der GUI stetig neu evaluiert -->
  <alsoMandatoryIf cached="false" language="groovy">
    !rootBO.StandardMedikamentAusgewaehlt
  </alsoMandatoryIf>
</Text>
```

Definiert das zugrundeliegende XML-Schema der Entität ein Attribut in der Datenbank bereits hart als Pflichtfeld (`mandatory="true"`), kann diese Restriktion niemals über ein `alsoMandatoryIf` in der GUI umgangen werden.

Das Tag `<Element>` dient in Formularen oft als feingranularer Wrapper, um das Labeling von Widgets zu überschreiben. Das Setzen von `hideForNullBO="true"` sorgt beispielsweise dafür, dass ein verknüpftes Element komplett aus der Maske verschwindet, wenn das zugrundeliegende Objekt den Zustand `null` hat. Da sich Datenbank-Schemata über die Jahre weiterentwickeln, ist das Attribut `missingPropertiesPolicy` im Wurzel-Knoten essenziell. Es steuert mit den möglichen Werten `error`, `ignore` oder `log` exakt, wie das Formular reagieren soll, wenn ein im XML-Layout referenziertes Attribut im zugrundeliegenden Datenmodell fehlt.

4.3.3. Reiter (Tabs) und Ladeverhalten

Ausladende Formulare werden klassischerweise in Reiter unterteilt, wofür das Tag `<TabbedView>` mit seinen `<Tab>`-Kindern genutzt wird. Aus Gründen der Performance werden Inhalte von Tabs standardmäßig erst dann vom Server geladen, wenn der Nutzer den Reiter explizit anklickt. Soll der Inhalt jedoch bereits beim initialen Öffnen der Maske geladen werden, muss dieses Ladeverhalten explizit auf `lazy="false"` geändert werden.

```
<Tab title="Zu öffnen" name="tabPosten" lazy="false">
```

Soll ein Tab programmatisch beim Öffnen des Formulars via Groovy-Skript fokussiert werden, muss dieser Tab zwingend mit `lazy="false"` definiert sein. Andernfalls existiert die GUI-Komponente zum Zeitpunkt des Skriptaufrufs noch nicht, was unweigerlich zu einer Ausnahme führt.

4.3.4. Aktionen, Berechtigungen und Uploads

Interaktionen werden als Aktionen deklariert und flexibel in Toolbars gerendert. Mit dem Attribut `toolBar` lassen sich Buttons in die Standard-Toolbar der Tabelle einfügen. Durch den Wert `topMdiOnly` werden sie in der Haupt-Toolbar des Client-Fensters platziert. Die Engine trennt bei Aktionen strikt zwischen reiner Sichtbarkeit über `<availableOn>` und Klickbarkeit über `<enabledOn>`.

```
<Action cmd="pullInfo" name="Info ziehen"
icon="/20x20png/Check.png">
  <enabledOn
language="groovy">bo.hatAlleNotwendigenDaten()</enabledOn>
  <availableOn
language="groovy">user.istMitgliedVonAdmins()</availableOn>
  <onAction language="groovy"><![CDATA[ ... Logik ...
]]></onAction>
</Action>
```

Um Standardfunktionen von Tabellen zu verbergen, wird die entsprechende Action mit einem leeren Tag überschrieben.

```
<Action cmd="newElement"/>
```

In allen Formular-Panels kann nativer Drag-'n'-Drop-Support für Dateien konfiguriert

werden. Das folgende Skript prüft vor dem Upload sicherheitshalber, ob der Datensatz bereits gespeichert wurde, und validiert anschließend das Dateiformat.

```
<onDrop language="groovy"><![CDATA[
  if (tx.isActive() && tx.isInvolved(rootBO)) {
     ctx.showError('Bitte erst speichern')
     return
  }
  def imageFilter = new Bild.DateiArtImageFileFilter(tx)
  data.each { file ->
    if (!imageFilter.accept(file)) {
        ctx.showError("Dateiart von Datei '${file.name}' wird
nicht unterstützt.")
        return
    }
    createXRayImage(file)
  }
  ftx.refreshForms()
]]></onDrop>
```

4.3.5. Tabellen in Formularen

Listen innerhalb von Formularen werden im `<Parameter>`-Bereich durch das Tag `<Table>` definiert. Die Darstellung der Spalten kann hierbei direkt über das Attribut `columns` gesteuert werden, wobei oftmals die Kurznotation zum Einsatz kommt. Die Definitionen der einzelnen Tabellenspalten werden in dieser Notation zwingend durch ein Pipe-Zeichen (`|`) voneinander getrennt. Innerhalb einer Spalte werden zusätzliche Eigenschaften durch ein Komma abgetrennt. Spalten, die in diesem String nicht explizit deklariert werden, blendet das System in der GUI automatisch aus. Soll die Spaltenüberschrift vom reinen Attributnamen abweichen, wird der neue Titel in einfache Anführungszeichen direkt hinter den Namen geschrieben. Breitenangaben erfolgen kompakt durch einen Zahlenwert plus Einheit, wobei neben Pixeln vor allem die Zeichenbreite (`c`) favorisiert wird.

Die Kurznotation unterstützt die komplexe Mehrfachsortierung. Wenn nach mehreren Spalten sortiert werden soll, wird die Priorität direkt als Zahl an das Schlüsselwort `asc` oder `desc` angehängt. Um bei polymorphen Tabelleninhalten den exakten Typ des Datensatzes anzuzeigen, wird zudem das System-Attribut `Bot.Name` als Spalte deklariert.

```
<Table entity="Ereignis"
```

```
       columns="Bot.Name '$R{EreignisTyp}' | Anfang, desc | Ende,
desc2 | Patient"
       loadImmediate="true" />
```

Diese Kurzschreibweise hat jedoch technische Grenzen, wenn eine Spalte tiefgreifende Modifikationen erfordert. In diesen Fällen muss zwingend auf das ausführliche XML-Tag `<Column>` ausgewichen werden.

```
<Table entity="Rechnung">
  <Query type="Text"/>
  <View>
    <!-- Auch hier bestimmt sortLevel="X" exakt die Reihenfolge
-->
    <Column property="Belegdatum" sort="DESC" sortLevel="1"/>
    <Column property="Patient" sort="ASC" sortLevel="2"/>
  </View>
</Table>
```

Die dynamische Filterung dieser formulinternen Tabellen wird exakt wie bei regulären Lesezeichen über `<filter>`-Tags innerhalb des `<Query>`-Elements gesteuert. Die Filterlogik reicht dabei von simplen Text- und Datumsfiltern bis hin zu dynamischen und kaskadierenden Dropdowns.

4.3.6. Schablonen-Konfiguration

Schablonen (`.tpl.xml`) definieren die strikten Regeln für die Anlage neuer Datensätze. Im `<Parameter>`-Bereich können über das Tag `<newInstance>` Skripte hinterlegt werden, die das neu erzeugte Objekt beim Klick auf „Neu“ mit Standardwerten vorbelegen. Das Kind-Tag `<Formular>` steuert dabei präzise, welches Formular sich direkt nach der Initialisierung für den Anwender öffnen soll.

```
<Schablone Name="Neue Benachrichtigung"
ElterPfad="/Admins/Alarme">
  <BOTyp Name="MyTISMBenachrichtigungsAuftrag"/>
  <Formular Name="Benachrichtigungs-Formular"/>
  <Parameter>
    <Schablone>
      <newInstance language="groovy"><![CDATA[
        // 'tx' ist die aktuelle Transaktion, 'ctx' der
```

```
ClientContext
        ba = tx.include(new MyTISMBenachrichtigungsAuftrag());
        ba.setAbsender(ctx.getSession().getUser()); // Absender
vorbelegen

        bv = tx.include(new MyTISMBenachrichtigungsVorlage());
        bv.setIstEinweg(true);
        ba.setVorlage(bv);

        return ba; // Zwingend das neue BO zurückgeben
      ]]></newInstance>
    </Schablone>
  </Parameter>
</Schablone>
```

4.3.7. Codebausteine zur Wiederverwendung

Um komplexen Layout-Code nicht mehrfach kopieren zu müssen, können komplette GUI-Ansichten in Codebausteine (`.bst.xml`) ausgelagert werden. Ein Formular kann diesen Baustein über das `<Include>`-Tag referenzieren. Der XML-Parser des Systems ersetzt den `<Include>`-Befehl beim Laden exakt durch den im referenzierten Codebaustein hinterlegten Inhalt.

```
<Tab title="Allgemein" scrollable="true">
  <View>
    <Include
name="/Admins/$R{MyTISM}/$R{Alarme}/$R{X}/Allgemein.elem"/>
  </View>
</Tab>
```

Zudem unterstützen diese Codebausteine auch dynamische Übergabeparameter. Diese werden im `<Include>`-Tag als Attribute definiert und im Baustein selbst via `$IP{…}` variabel aufgelöst.

```
<Include name="codebaustein" attrWert="zwei"/>
```

4.3.8. Maskierung und Anzeige über CBOFormat

Innerhalb der XML-Konfiguration von Strukturelementen steuern Power-User und Administratoren die Objektdarstellung primär über das Attribut `format`. Dieses Attribut nimmt einen validen `CBOFormat`-Ausdruck entgegen und kommt bei einer Vielzahl von Oberflächenkomponenten zum Einsatz.

Auswahllisten und ComboBoxen nutzen das Attribut, um festzulegen, wie ein referenziertes BO in der Dropdown-Liste für den Anwender dargestellt wird.

Einbindung in einer ComboBox

```
<ComboBox property="Patient" format="'Id: ' Name" />
```

In Tabellenspalten (`<Column>`) wird das Werkzeug eingesetzt, um verknüpfte Relationen innerhalb einer einzelnen Zelle ansprechend aufzubereiten. Zusätzlich steht hier das Attribut `toolTipFormat` zur Verfügung. Dieses definiert eine alternative Maskierung, die dem Anwender beim Verweilen mit dem Mauszeiger über der Zelle als Tooltip eingeblendet wird.

Einbindung in einer Tabellenspalte mit Tooltip

```
<Column property="Pflegestufe" format="Name"
toolTipFormat="'Interne ID: ' Id" />
```

Labels und Textfelder nutzen das `CBOFormat` als dynamische Formatierungsschablone für die reine Datenanzeige. Die zwingende Voraussetzung für die korrekte Evaluierung des Formats ist das gleichzeitige Setzen des Attributs `property`.

Auch im Kalender- und Scheduler-Modul steuert das Format das visuelle Erscheinungsbild. Innerhalb des `<Scheduler>`-Elements wird dem `contactMapper` über das Attribut `format` ein entsprechender Ausdruck übergeben, um die Beschriftung von Kontakten und Terminen im Zeitraster zu manipulieren.

Einbindung im Scheduler-Modul

```
<Scheduler property="Termine">
   <contactMapper format="Nachname ', ' Vorname" />
</Scheduler>
```

4.4. Architektur, Backend & Deep-Dive

Die Formularengine verknüpft das XML-basierte Layout direkt mit der Datenbankebene des Backends.

4.4.1. Core-Architektur und Immutabilität

Neben den projektspezifischen Strukturelementen existieren vordefinierte Basis-Elemente aus dem System-Core. Diese Core-Elemente liegen im Quellcode-Verzeichnis unter `/nrx/de/ipcon/db/core/resources` und werden beim Build-Prozess fest in die zentrale Kernel-JAR einkompiliert. Ein Hintergrunddienst liest diese XML-Dateien bei jedem Serverstart aus und legt sie in der Datenbank an. Architektonisch sind diese System-Elemente als strikt unveränderlich zu betrachten. Sollen Änderungen an diesen Core-XMLs wirksam werden, müssen Entwickler die Anpassungen direkt im Code vornehmen. Anschließend muss die Marker-Datei `.checked-initialdata` im Serververzeichnis gelöscht und das System neu gestartet werden.

4.4.2. Context-Management: Client vs. Formular

Die Architektur unterscheidet strikt zwischen dem globalen `ClientContextI` (`ctx`) und dem formularspezifischen `FormContextI` (`ftx`). Der globale Kontext verwaltet systemweite Aktionen, wie das Speichern laufender Transaktionen oder das Aufrufen globaler Dialoge. Zudem steuert der globale Kontext GUI-Aufrufe, wie das Öffnen eines definierten Lesezeichens über `ctx.openView()`. Der formularspezifische Kontext ist hingegen ausschließlich für Detailaktionen innerhalb einer spezifischen Maske zuständig. Die Engine injiziert in alle GUI-Skripte standardmäßig die Variablen `ctx`, `ftx`, `tx` sowie das aktuelle Business Object `bo`.

Es ist architektonisch zulässig, den globalen Client-Kontext aus dem aktuellen Formular-Kontext abzurufen. Der umgekehrte Weg ist jedoch strikt untersagt, da Zugriffe auf andere Formulare zu schwerwiegenden Nebeneffekten in der Benutzeroberfläche führen.

Beim Zugriff auf Eigenschaften des Form-Kontexts aus Groovy heraus sollte stets die ausführliche Getter-Schreibweise genutzt werden. Groovy bevorzugt bei Properties andernfalls fehlerhaft interne Methoden wie `isRoot()`, was zu unerwarteten Werten bei der Code-Evaluierung führt.

4.4.3. Breadcrumb-Navigation und Architektur

Die Breadcrumb-Navigation ermöglicht den kontextbasierten Zugriff auf Objekte innerhalb einer verschachtelten Formular-Hierarchie. Dadurch können Skripte bei uneindeutigen Relationen den logischen Pfad der Datenstruktur rückwärts auflösen. Dieser

Navigationszustand wird exklusiv für den aktuellen Thread in einem lokalen Speicher vorgehalten. Das System verzichtet bewusst auf ein vererbbares `ThreadLocal`, um zu verhindern, dass asynchron abgespaltene Threads die Historie erben und Race Conditions auslösen. Ein interner Zähler dient als Verschachtelungsschutz und stellt sicher, dass bei kaskadierenden Aufrufen stets der äußerste Einstiegspunkt den Start definiert.

Die API des `BreadcrumbFinders` bietet spezifische Methoden zur Durchsuchung der Hierarchiekette. Die Methode `findParentBO()` liefert das direkte Eltern-Objekt, während `findFirstBOOfType()` das erste Objekt eines definierten Typs sucht. Bevor Backend-Komponenten auf Schema-Attribute zugreifen, muss der Kontext initialisiert und abschließend bereinigt werden. Um Speicherlecks zu verhindern, ist hierbei zwingend das Try-Finally-Muster anzuwenden.

```
method syncImpl()
   // 1. Kontext betreten
   FormContextI.enterBreadcrumb(getFtx())

   do
      // 2. Zugriff auf das Schema
      currBOValue = getSchema().getValueAsString(bo,
getDisplayProperty())
      // weitere Logik...
   finally
      // 3. Kontext zwingend verlassen
      FormContextI.exitBreadcrumb(getFtx())
   end
end
```

4.4.4. Lebenszyklus und Action-Hooks

Die Formular-Engine basiert auf einem streng synchronisierten Model-View-Konzept. Über Action-Hooks klinken sich Entwickler direkt in den Datenaustausch zwischen Datenbank und Benutzeroberfläche ein. Die Engine injiziert innerhalb dieser XML-Skript-Tags automatisch feste Kontext-Variablen: `bo` (aktuelles Business Object), `rootBO` (Basis-BO der Maske), `fe` (spezifisches `FormElementI`), `ftx`, `tx` und den Lade-Kontext `bol`.

Der Hook `<onRefresh>` löst aus, wenn Daten aus dem Model in die GUI geladen werden. Der Hook `<onSync>` feuert beim Zurückschreiben der Eingaben aus der GUI in das zugrundeliegende BO. Dies geschieht typischerweise beim Verlassen eines Eingabefeldes oder unmittelbar vor dem Speichern des Formulars.

```
<Text property="Beschreibung">
  <onRefresh language="groovy">
    ftx.toast('Lade Befund aus der Datenbank in die GUI.')
  </onRefresh>
  <onSync language="groovy">
    ftx.toast('Speichere geänderten Befund von der GUI ins
Model.')
  </onSync>
</Text>
```

Für direkte Interaktionen in Listen existiert der Hook `<onAfterSelectValue>`, der unmittelbar nach der Selektion eines Tabellenwerts feuert. Das folgende Beispiel nutzt diesen Hook, um eine versehentliche Mehrfachselektion programmatisch zu unterbinden.

```
<Table property="Diagnosen" columnSelectionAllowed="false"
autoSelectLast="true">
  <onAfterSelectValue language="groovy"><![CDATA[
    // Prüfe über das injizierte FormElementI (fe), ob mehrere
Zeilen markiert wurden
    if (fe.hasMultipleSelectedObjects()) {
      // Setze die Selektion hart auf die erste markierte Zeile
zurück
      fe.selectedLine = fe.firstSelectedLine
      ftx.toast("Bitte immer nur eine einzige Diagnose
auswählen!")
    }
  ]]></onAfterSelectValue>

  <Column property="Name"/>
</Table>
```

Zusätzlich existieren die Hooks `<onFocusGained>` und `<onFocusLost>`, die beim Betreten und Verlassen eines Eingabefeldes auslösen.

4.4.5. Virtuelle Attribute und GrooqlFilter

Virtuelle Eigenschaften berechnen sich zur Laufzeit dynamisch und können für Optimierungszwecke gecacht werden.

```
<virtualProperty entity="BO" name="TeureSumme" type="Long"
cached="true">
  <get>return bo.BOLoader.queryBO("sum(Id) from BO a where not
Ldel").find()</get>
</virtualProperty>
```

Innerhalb virtueller Attribute darf zum Nachladen von Daten ausschließlich der `BOLoader` und niemals direkt eine Transaktion aufgerufen werden. Da Java-Arrays grundsätzlich veränderlich sind, darf ein gecachter Rückgabewert eines virtuellen Array-Attributs niemals direkt modifiziert werden. Die generierten Getter-Methoden von persistenten Attributen dürfen keinesfalls überschrieben werden, da dies das GUI-Formular blockiert.

In der XML-Deklaration virtueller Eigenschaften dienen Breadcrumbs häufig dem Zugriff auf übergeordnete Kataloge im Datenbaum.

```
<!-- Suche nach einem spezifischen Katalog-Typ im Baum -->
<virtualProperty entity="Medikament" name="Bestand"
type="LagerBestand" relation="n-1">
  <get>
    def stock =
Breadcrumb.findFirstBOOfType(Apothekenkatalog)?.lager
    if (!stock) {
      return null
    }
    bo.getBestandInLager(stock)
  </get>
</virtualProperty>
```

Ein weiterer Anwendungsfall ist das Filtern von Listen basierend auf dem Root-Kontext, der meist durch die „BX“-Entität abgebildet wird. Das System greift hierbei auf das Haupt-Konfigurationsobjekt zu, um globale Einstellungen für die Auswertung heranzuziehen.

```
<virtualProperty entity="Mitarbeiter" name="TimeEntriesForWeek"
type="ZeiterfassungsEintrag" relation="1-n">
  <get><![CDATA[
    import de.ipcon.tools.date.DateTimeTools

    def bx = Breadcrumb.findFirstBOOfType(BX)
```

```
    def startOfWeek = bx.FirstDayOfWeek
    // ... Berechnungslogik ...
  ]]></get>
</virtualProperty>
```

Das Konzept `GrooqlFilter` verbindet Formularmasken direkt mit OQL-Abfragen. Ein Groovy-Skript prüft Objekte im Arbeitsspeicher und wird vom Framework gleichzeitig in eine korrespondierende Datenbankabfrage transformiert.

4.4.6. Validierung im Client und Deadlocks

Für Konsistenztests auf der Benutzeroberfläche existiert die Hook-Methode `verifyOnClient()`. Diese dient der reinen clientseitigen Validierung von Eingabeformaten, bevor Daten an den Server übertragen werden. Innerhalb dieser Methode besteht eine akute Deadlock-Gefahr. Es dürfen hier niemals asynchrone GUI-Interaktionen wie Bestätigungsdialoge aufgerufen werden. Für fachliches Feedback muss stattdessen eine `SaveVetoException` geworfen werden, deren Nachricht das System fängt und als Dialog anzeigt.

4.4.7. GUI-Threading und Hintergrundprozesse

Beim GUI-Threading darf der Event-Dispatcher-Thread (EDT) der Swing-Oberfläche niemals durch Datenbank-Operationen blockiert werden. Für die asynchrone Ausführung komplexer Berechnungen im Hintergrund stellt das Framework die Utility-Klasse `SPU` bereit.

```
SPU.offEDT(myComponent, () -> calculateData())
   .thenAccept(data -> SPU.onEDTv(myComponent, () ->
updateUI(data)));
```

Muss ein Prozess auf eine Antwort warten, ohne die Benutzeroberfläche einzufrieren, initiiert das System einen sekundären Loop.

```
SPU.offEDTAndWait(myComponent, () -> saveToDb());
```

Für temporäre, rein GUI-gesteuerte Auflistungen müssen Dummy-Objekte mit flüchtiger ID und zugewiesenem `BOLoader` initialisiert werden.

```
Quertabelle tmp = new Quertabelle()
```

```
tmp.setTempId()
tmp.setBOLoader(bo.getBOLoader())
tmp.setName("Januar")
```

4.4.8. Fortgeschrittenes Rendering und Workarounds

Formulare können für dynamisches GUI-Styling HTML-Fragmente evaluieren, um spezifische Werte farblich hervorzuheben. Dies kommt häufig in Tabellenzellen zum Einsatz, bei denen ein virtuelles Attribut im `<get>`-Block einen formatierten HTML-String zurückgibt.

```
return """<html><body style="background: ${->color}">${-
>bo.getName()}</body></html>"""
```

Werden in diesen HTML-Skripten Icons über das `<img>`-Tag geladen, müssen in aktuellen Java-Versionen die Attribute `height` und `width` zwingend deklariert werden.

Mittels des `<renderer>`-Tags lässt sich das Zeichnen von Tabellenzellen durch Groovy-Skripte vollständig überschreiben. Wird ein solcher Custom-Renderer implementiert, muss der visuelle Zustand im `else`-Zweig zwingend zurückgesetzt werden. Da die Rendering-Engine GUI-Zellen beim Scrollen recycelt, würden visuelle Modifikationen andernfalls auf falsche Datensätze übertragen.

```
<renderer>

renderer.setHorizontalAlignment(javax.swing.SwingConstants.CENTER
)
  if (isSelected) {
     renderer.setText("\u2714")
  } else {
     renderer.setText(null)
  }
</renderer>
```

Der programmatische Zugriff auf GUI-Elemente erfolgt über den Array-Operator des `FormContexts`.

```
def bos = ftx['$TabellenName'].getSelectedObjects() as List
```

Schalten Skripte die Sichtbarkeit von Elementen dynamisch um, schlägt ein sofortiges Neuladen (Refresh) oft fehl, da der Zeichenprozess noch nicht abgeschlossen ist. Zur Vermeidung dieser Race-Condition wird der Update-Befehl über die Klasse `SPU` asynchron in die Queue delegiert.

```
SPU.onEDTv ftx, { -> ftx['v_error_report'].ftx.refreshForms() }
```

Für dynamische Masken können XML-Codebausteine zur Laufzeit programmgesteuert geladen und geparst werden.

```
def arguments = [employeeId:String.valueOf(mitarbeiterId)]
def bpathStr = '/Admins/Codebaustein/employee_row.element'
def codebaustein =
ctx.getCodebausteinStorage().get(BenanntPath.ofMultipathString(bp
athStr), null)
def parameter = codebaustein.parameter.replaceFirst(/<Include>/,
'')
return codebaustein.replaceArguments(parameter, [bpathStr],
arguments)
```

Für das Debugging lassen sich Konsolenausgaben direkt in Skript-Blöcken platzieren, welche im Client über die „Groovy-Konsole“ ausgelesen werden.

Fortgeschrittene Maskierung und Zahlenformatierung über CBOFormat

Für Anwendungsfälle im Backend-Code oder in Transformationsskripten bietet das `CBOFormat` eine erweiterte Syntax zur Formatierung. Diese Logik wird benötigt, wenn Daten für JSON-Antworten aufbereitet oder innerhalb von Hintergrundprozessen maskiert werden müssen. Um numerische Attribute mit Formatierungsmustern zu versehen, greift eine spezifische geschweifte Klammer-Syntax.

Komplexe Zahlenformatierung im Skript

```
def formatString = ".{Betrag{,#0.00}} 'EUR'"
def formatiertesErgebnis = rechnungsBO.describe(formatString)
```

Der Ausdruck `.{<Attribute>{,#0.00}}` wendet die definierte Formatmaske direkt auf den internen Dezimalwert an. Der anschließende Text in einfachen Anführungszeichen wird als statisches Suffix angehängt. Diese Methode evaluiert die Regeln direkt auf Java-Ebene und vermeidet temporäre String-Konkatenationen im Skript.

Chapter 5. Reporting-Engine

5.1. Grundlagen & Konzepte

Die Reporting-Engine transformiert strukturierte Datenbankinhalte in ansprechende, exportierbare Dokumente. Typische Anwendungsfälle im klinischen Umfeld sind die automatisierte Generierung formaler Entlassungspapiere, fortlaufender Bestandslisten oder Namensetiketten. Im Gegensatz zu klassischen Textverarbeitungen arbeitet das System mit einer strikten Trennung von Design und Inhalt. Ein Report agiert als dynamische Schablone mit fest definierten Platzhaltern für Logos, Texte und Tabellen. Erst zur Laufzeit greift die Engine auf die Datenbank zu und integriert die aktuellen Objektinformationen in dieses Layout.

5.1.1. Datenverarbeitung und Rendering

Die Engine verarbeitet die bereitgestellten Daten nacheinander. Sie läuft über die ausgewählten Hauptobjekte und löst dabei die definierten Relationen hierarchisch auf. Die Strukturierung großer Datenmengen erfolgt über logische Gruppen, welche redundante Daten zusammenfassen und optische Abschnitte – wie etwa den Wechsel zur nächsten Station – generieren.

Als primäre Rendering-Engine dient eine modifizierte und um Rich-Text-Fähigkeiten erweiterte Version des Open-Source-Tools JasperReports. Reports lassen sich standardmäßig in diverse Formate wie PDF, HTML, CSV, Excel oder RTF exportieren oder inklusive Client-Vorschau ausdrucken. Für unformatierte, stark textbasierte Dokumente, wie beispielsweise seitenlange psychiatrische Gutachten, steht alternativ eine leichtgewichtige AsciiDoc-Engine zur Verfügung.

5.1.2. Objektorientierte Datenbeschaffung

Klassische Reporting-Tools erfordern oftmals hochkomplexe und fehleranfällige SQL-Abfragen direkt im Report-Layout, um Datengruppen oder Sortierungen zu erzeugen. MyTISM verzichtet im Layout vollständig auf direkte Datenbankabfragen. Die Datengrundlage wird stattdessen über eine XML-basierte Definition fest am jeweiligen Objekttyp (Entität) verankert. Diese Definition steuert zentral, welche Relationen aufgelöst und in welcher Reihenfolge die Daten an das Layout übergeben werden. Soll eine vollständige Patientenakte generiert werden, wird der Report an der Entität „Patient“ verankert, woraufhin das System dann etwa die chronologische Behandlungs-Historie automatisch bereitstellt.

5.1.3. Bänder-Architektur (Bands)

Das Layout-Konzept von JasperReports basiert nicht auf statischen Tabellen, sondern auf sogenannten Bändern (Bands). Das Kernelement ist das Detailband, welches für jeden einzelnen Datensatz – beispielsweise für jedes Medikament einer Inventarliste – wiederholt gerendert wird. Dieses Detailband wird von Kopf- und Fußbändern (Header und Footer) umschlossen. Diese äußeren Bänder steuern Seitenumbrüche oder Gruppenwechsel, um etwa am Ende einer Liste die summierten Behandlungskosten abzugrenzen.

5.1.4. Ausführungsmodi und Typen

Das System unterscheidet konzeptionell zwischen dynamischen Ad-hoc-Reports und eigenständigen Reports. Ad-hoc-Reports werden vom Anwender direkt aufgerufen, indem er beispielsweise spezifische Patienten in einem Lesezeichen markiert und den Report über das Kontextmenü für exakt diese Treffermenge anfordert. Eigenständige Reports ermitteln ihre Datengrundlage hingegen völlig selbstständig anhand einer fest im Report vorkonfigurierten OQL-Abfrage, ohne dass eine vorherige Benutzerauswahl erforderlich ist.

Darüber hinaus wird zwischen Einzel-Reports und Listen-Reports unterschieden. Ein Einzel-Report fokussiert sich auf die Details eines spezifischen Objektes, wie den individuellen Arbeitsvertrag eines Arztes. Ein Listen-Report arbeitet mehrere Objekte iterativ ab, wie beispielsweise die aggregierte Liste aller diensthabenden Ärzte einer Nachtschicht.

5.1.5. Platzhalter: Felder, Variablen und Parameter

Um die abstrakten Datenmodelle in das Dokument zu überführen, nutzt die Engine spezifische Platzhalter. Felder greifen direkt auf die Attribute der übergebenen Business Objects zu. Variablen dienen systemseitigen Berechnungen zur Laufzeit, wie beispielsweise für Seitenzahlen oder aggregierte Rechnungssummen. Parameter ermöglichen die interaktive Dateneingabe durch den Benutzer vor der Generierung, um etwa den Auswertungszeitraum eines Schichtplans einzugrenzen. Zusätzlich übernehmen Lokalisierungs-Makros die automatische Übersetzung statischer Textbausteine.

5.1.6. Subreports und Vererbung

Subreports erlauben es, andere Reports als wiederverwendbare Module in einen Hauptreport einzubetten. Dies ist besonders nützlich, um Daten aus völlig unterschiedlichen Entitäten in einem einzigen Dokument zusammenzuführen, beispielsweise beim Einbetten einer Medikamentenliste in die Patientenakte. Dank der Systemarchitektur steht ein Report, der für eine Basis-Klasse definiert wurde, durch Polymorphie automatisch auch allen abgeleiteten Unterklassen zur Verfügung. Die Sichtbarkeit jedes Reports bleibt dabei stets strikt an das zentrale Rechtesystem gekoppelt.

CBOFormat-Integration in JasperReports

Um komplexe Formatierungslogiken direkt im XML-Layout des Reports zu vermeiden, unterstützt die Engine das MyTISM-spezifische `CBOFormat`. Adressblöcke, Patientennamen oder komplexe Datentypen können so systemweit einheitlich und lesbar ausgegeben werden. Um das Format im Report auf das aktuelle Hauptobjekt anzuwenden, wird das Basis-BO über das reservierte Feld `$F{THIS}` innerhalb des Ausdrucks angefordert.

Anwendung des Formats in einem Jasper-Textfeld

```
<textField>
    <reportElement x="0" y="0" width="200" height="20"/>
    <textElement/>
    <textFieldExpression><![CDATA[$F{THIS}.describe("Nachname ',
' Vorname")]]></textFieldExpression>
</textField>
```

Durch diese saubere Kapselung greift der Report direkt auf die zentralen Formatierungsroutinen des Systems zu. Verweist das Ende einer Attributkette in einem Report nicht auf einen einfachen Wert (Skalar), sondern auf ein komplexes Business Object, wendet das System automatisch das definierte Standardformat dieses BOs an. Nachträgliche Änderungen an diesem Standardformat wirken sich dadurch ohne manuellen Eingriff direkt auf alle zugehörigen Reports aus.

5.2. Benutzeroberfläche & Bedienung

Für Power-User beginnt die Verwaltung und Bereitstellung von Reports direkt in der grafischen Benutzeroberfläche Solstice.

5.2.1. Anlage im Client

Die Erstellung eines neuen Reports erfolgt im Client typischerweise über die vorgebaute Schablone im Navigationsbaum. Jeder Report wird dabei fest an einen spezifischen BO-Typ gebunden, wie beispielsweise an die Entität „Vertrag“. Eine konfigurierbare, numerische Priorität steuert die Anzeigereihenfolge in den Kontextmenüs. Bei etwaigen Konflikten gewinnt stets der speziellste BO-Typ oder die höchste Priorität.

Durch das Setzen der Checkbox „Auch für Unterklassen des BO-Typs nutzbar“ wird ein Basis-Report polymorph für alle abgeleiteten Sub-Entitäten freigeschaltet. Ein auf der Basisklasse „Vertrag“ definierter Bericht steht somit automatisch auch für spezialisierte Typen wie „Arztvertrag“ oder „Wartungsvertrag“ zur Verfügung.

Im selben Formular werden die zulässigen Druckziele konfiguriert, wie etwa eine PDF-

Vorschau oder der direkte physische Druck. Sind in den Einstellungen explizite Sprachen hinterlegt, limitiert das System die L10n-Übersetzung im finalen Druckdialog strikt auf diese sprachliche Auswahl.

5.2.2. Ausführungsmodi in der GUI

Zwei zentrale Flags im Formular steuern das Ausführungsverhalten der Rendering-Engine. Ist die Option „ist eine Liste“ aktiviert, iteriert der Report in einem einzigen Durchlauf über eine gesamte Menge markierter Objekte, anstatt nur ein einzelnes Element zu verarbeiten. Dieses Verhalten ist essenziell für aggregierte Übersichtsberichte, wie etwa tabellarische Auflistungen mehrerer Datensätze.

Das Flag „ist eigenständig“ bewirkt hingegen, dass der Report die aktuelle GUI-Selektion des Anwenders vollständig ignoriert. Die Datengrundlage wird in diesem Fall autark über eine fest integrierte OQL-Abfrage ermittelt.

Für das System-Debugging durch Power-User bieten die Formulare dedizierte, schreibgeschützte Reiter. Die Reiter „Verarbeitete Parameter“, „Verarbeitete Report-Definition“ und „Verarbeitete Anker-Definition“ visualisieren den fertig aufgelösten XML-Quellcode unmittelbar vor der Übergabe an die Rendering-Engine.

5.3. Erweiterte Konfiguration (XML)

Für tiefgreifende Anpassungen der Anker-Definitionen und der Layout-Logik nutzen Administratoren das direkte XML-Design. Exportierte Reports bestehen im Dateisystem physisch aus zwei separaten Dateien, wobei die Hauptdatei stets das Suffix `.rpt.xml` trägt.

5.3.1. Schema-Konfiguration (`schema.xml`)

Globale Standardwerte für Automatik-Reports, wie die generelle Seitenausrichtung oder bevorzugte Schriftgrößen, werden in der Datei `schema.xml` über das Element `<report>` definiert.

```
<report title="Analyse" orientation="Portrait"
fontSizeNormal="10" fontSizeBig="14"/>
```

Zusätzlich lassen sich spezifische Steuerungsattribute direkt an den einzelnen Entitäts-Attributen deklarieren. Dadurch können interne IDs auf automatischen Reports standardmäßig ausgeblendet, Spaltenbreiten angepasst oder Sortierungen hartkodiert werden.

Werden in einer Entität mehrere Attribute vom selben Typ deklariert, kann es beim Kompilieren der `schema.xml` zu einer Exception durch Namenskollisionen bei den implizit erzeugten Rückrelationen kommen. In diesem Fall muss das Attribut `backRelation` manuell umbenannt oder die Rückrelation mittels `ignoreReverseRelations="true"` explizit unterdrückt werden.

5.3.2. XML-Konfiguration: Die Anker-Definition (`<set>`)

Die Anker-Definition steuert die Datenbeschaffung und ersetzt direkte SQL-Queries im Layout. Im einfachsten Fall definiert das Root-Element lediglich den fundamentalen Startpunkt für die Abfrage.

```
<set entity="Vertrag"/>
```

Das Auflösen von 1:n- oder n:m-Relationen erfolgt zwingend über das Tag `<many>`, welches die referenzierten Objekte sortiert an das Layout übergibt. Durch die Vergabe eines Alias (`alias="P"`) lassen sich die einzelnen Elemente der Relation später im Layout gezielt über das Feld `$F{P}` ansprechen.

```
<set entity="KrankenhausRechnung">
    <many property="Posten" alias="P">
        <sort ascending="true" byProperty="Position"/>
    </many>
</set>
```

Daten-Gruppierungen erfordern eine strikte Synchronisation zwischen dem Jasper-Layout und der Anker-Definition. Die Sortierung im `<many>`-Tag des Ankers muss exakt der `groupExpression` der Layout-Gruppe entsprechen, da die Engine die Daten sequenziell verarbeitet. Bei einem Gruppenwechsel löst die Engine automatisch den „Group Footer“ aus, um beispielsweise aggregierte Zwischensummen zu drucken. Bleibt die `groupExpression` leer, iteriert die Gruppe nicht über Daten, sondern verschachtelt lediglich Layout-Bereiche, was als Workaround für extrem lange Reports dient.

Virtuelle Eigenschaften können direkt innerhalb des Ankers spezifisch für den jeweiligen Report deklariert werden.

```
<virtualProperty name="WirkstoffeAlsString" entity="Rezept">
```

```
<get>de.ipcon.tools.TextTools.join(getWirkstoffe().values())</get
>
</virtualProperty>
```

Eine syntaktisch fehlerhafte Ankerdefinition führt beim Speichern oder Rendern nicht zwingend zu einer expliziten Fehlermeldung. Dies resultiert oftmals in einem stillen Fehler, der sich ausschließlich durch leere Ergebnismengen im finalen Dokument äußert.

5.3.3. Eigenständige Abfragen und Filter

Für eigenständige Reports wird die Abfrage direkt im XML über das Tag `<Query type="Text">` deklariert. Diese Abfrage befüllt das systemseitige Variablen-Array `$P{BOS}` bei der Ausführung automatisch mit den ermittelten Objekten. Über das Attribut `fieldWidth` kann die Breite des zugehörigen Suchfeldes in der GUI gesteuert werden. Für komplexe Filterungen bietet das Tag `<transform-script>` die Möglichkeit, Abfrageresultate programmatisch zu modifizieren, bevor sie an die Engine übergeben werden.

```
<Query type="Text" entity="Medikament" fieldWidth="20">
    <filter><![CDATA[Bestand < Mindestmenge]]></filter>
    <transform-script language="groovy">
        /* Optionale Modifikation der Resultate hier */
    </transform-script>
</Query>
```

5.3.4. Benutzerdialoge & Parameter (`<parameter>`)

Sollen Eingabeparameter vor der Ausführung als interaktiver Auswahldialog präsentiert werden, muss das Attribut `isForPrompting="true"` gesetzt sein.

```
<parameter name="Stichtag" isForPrompting="true"
class="java.util.Date">
    <property name="format" value="MEDIUM_"/>
</parameter>
```

Über das Property `choiceScript` lassen sich im Dialogfenster dynamische Dropdown-Listen generieren. Die mögliche Eingabe wird durch `chooseOnly="true"` strikt auf die

vorgegebenen Werte limitiert. Das Property `nullable="true"` steuert, ob die Dropdown-Auswahl leer gelassen werden darf.

```
<parameter name="GruppierenNach" isForPrompting="true"
class="java.lang.String">
    <property name="choiceScript"
value="model.addEntry('$R{Station}');
model.addEntry('$R{Fachbereich}');"/>
    <property name="chooseOnly" value="true"/>
    <property name="nullable" value="true"/>
</parameter>
```

Über das Attribut `rawInputDefinition` kann ein komplettes MyTISM-Auswahl-Popup inklusive OQL-Filtern direkt in den Dialog injiziert werden. Hierbei müssen sämtliche XML-Sonderzeichen streng als Entities escaped werden, um die Syntax nicht zu brechen.

```
<parameter name="BehandelnderArzt" isForPrompting="true"
class="de.ipcon.db.core.Benutzer">
    <property name="rawInputDefinition" value="&lt;Popup
property=&quot;VBO&quot;&gt;&lt;Table&gt;&lt;Query
type=&quot;Text&quot; entity=&quot;Benutzer&quot;&gt;
&lt;filter&gt;NOT AnmeldungVerweigern OR AnmeldungVerweigern =
null&lt;/filter&gt; &lt;/Query&gt;&lt;Columns&gt;Name,
ASC|Beschreibung&lt;/Columns&gt;&lt;/Table&gt;&lt;/Popup&gt;"/>
</parameter>
```

Parameternamen in Reports dürfen niemals Punkte enthalten. Dieses Trennzeichen bricht die Skriptauswertung der Engine sofort und führt zum Abbruch der Generierung.

5.3.5. Jasper-Layouts: Bilder, Diagramme und Formatierungen

Um eine einheitliche Typografie zu gewährleisten, können in der XML-Definition übergreifende Schriftarten deklariert werden.

```
<reportFont name="Klinik_Standard" isDefault="true"
fontName="Arial" size="11"/>
```

Standardmäßig werden Datenmengen in Detail-Bändern vertikal untereinander aufgefaltet. Sollen Listeninhalte stattdessen horizontal nebeneinander dargestellt werden, muss das Attribut `printOrder="Horizontal"` im Root-Tag gesetzt werden. Die verfügbare Seitenbreite muss hierbei exakt auf die konfigurierte `columnWidth` und das optionale `columnSpacing` aufgeteilt werden. Die Summe aus Spaltenbreite und Spaltenabstand, multipliziert mit der Anzahl der Elemente, darf die verfügbare Druckbreite der Seite niemals überschreiten.

Bilder müssen initial als reguläres Business Object vom Typ „Bild“ importiert und als „BildPosten“ verknüpft werden. Für Vektorgrafiken (SVG) ändert sich die deklarierte Klasse des Parameters, und in der `imageExpression` muss ein `BatikRenderer`-Objekt instanziiert werden.

```
<parameter name="Krankenhauslogo" isForPrompting="false"
class="de.ipcon.db.core.Bild"/>

<imageExpression
class="net.sf.jasperreports.engine.JRRenderable"><![CDATA[new
net.sf.jasperreports.renderers.BatikRenderer($P{Krankenhauslogo})
]]></imageExpression>
```

Die vergebenen Namen von eingebundenen BildPosten dürfen systemweit niemals Bindestriche enthalten. Dieses Zeichen wird bei der Namensauflösung fehlinterpretiert und stört die Skriptauswertung der Bildverarbeitung.

Zur visuellen Aufbereitung aggregierter Daten nutzt die Engine das Tag `<barChart>`. Die Datenübergabe erfolgt über ein gekapseltes `categoryDataset`, welches in verschiedenen `categorySeries` definiert, wie die Achsen und Balken zu rendern sind.

```
<barChart>
  <chart>
    <reportElement key="barChart-1" x="0" y="0" width="500"
height="200"/>
  </chart>
  <categoryDataset>
    <dataset>
      <datasetRun subDataset="VerbrauchsDaten">
        <dataSourceExpression><![CDATA[new
de.ipcon.db.report.BOsDataSource($P{BOS} as BO[],
```

```
$P{REPORT})]]></dataSourceExpression>
      </datasetRun>
    </dataset>
    <categorySeries>
      <seriesExpression><![CDATA["Verbrauch in
Einheiten"]]></seriesExpression>

<categoryExpression><![CDATA[L10n.formatDateNT($F{FuerMonat},
'MMMM')]]></categoryExpression>

<valueExpression><![CDATA[$F{Verbrauch}]]></valueExpression>
    </categorySeries>
  </categoryDataset>
  <barPlot>
    <plot>
      <seriesColor seriesOrder="0" color="#005A9C"/>
    </plot>
  </barPlot>
</barChart>
```

Werden in den Layouts Hintergrundfarben definiert, muss das Attribut `mode="Opaque"` zwingend gesetzt sein, da die Farbe andernfalls transparent bleibt. Laufzeitsummen werden über das Attribut `calculation="Sum"` deklariert.

```
<variable name="Gesamtkosten" class="java.math.BigDecimal"
calculation="Sum">

<variableExpression><![CDATA[$F{POSTEN}.Betrag]]></variableExpres
sion>
</variable>
```

Für die Darstellung von dynamischen Seitenzahlen im Format „Seite X von Y“ existiert ein spezifischer architektonischer Workaround. Zwei Textfelder werden nebeneinander platziert und werten die Systemvariable `$V{PAGE_NUMBER}` zu unterschiedlichen Rendering-Zeitpunkten aus.

```
<textField evaluationTime="Now">
    <textFieldExpression class="java.lang.String"><![CDATA["Seite
```

```
${$V{PAGE_NUMBER}} von " ]]></textFieldExpression>
</textField>
<textField evaluationTime="Report">
    <textFieldExpression
class="java.lang.Integer"><![CDATA[$V{PAGE_NUMBER}
]]></textFieldExpression>
</textField>
```

Wird ein Fließtext in der PDF-Datei abgeschnitten, muss das Attribut `isStretchWithOverflow="true"` gesetzt werden, damit sich das Feld dynamisch nach unten ausdehnen kann. Um in solchen Layouts Überlappungen zu vermeiden, verschiebt `positionType="float"` nachfolgende Felder automatisch nach unten.

Um unerwartete weiße Flächen im gerenderten PDF zu vermeiden, müssen ausblendbare Textfelder zwingend das Attribut `isRemoveLineWhenBlank="true"` erhalten. Zusätzlich muss die definierte Höhe des Bänders exakt der maximalen Höhe der umschlossenen Elemente entsprechen.

Bei horizontal auffaltenden Bändern kann ein ungewollter Treppeneffekt entstehen, wenn leere Felder ausgeblendet werden. Da das Entfernen einer Zelle die komplette Zeile über die gesamte Reportbreite anhebt, verschieben sich die nachfolgenden Spalten sukzessive nach oben.

5.3.6. Einbindung von Subreports im Layout

Das Einbinden von untergeordneten Subreports in einen Hauptreport erfordert zwingend die Definition von exakt zwei Parametern im Hauptdokument. Ein Parameter wird für die Jasper-Engine benötigt, der andere referenziert das zugrundeliegende Business Object des Reports.

```
<parameter name="LaborwerteSubreport" isForPrompting="false"
class="net.sf.jasperreports.engine.JasperReport"/>
<parameter name="LaborwerteSubreportBO" isForPrompting="false"
class="de.ipcon.db.core.Report"/>
```

Subreports lassen sich in der Engine architektonisch nicht drehen. Hat der Hauptreport das Format „Portrait“, kann ein eingebetteter Subreport nicht

im „Landscape“-Modus gerendert werden.

Wird ein Subreport auf eine Liste von Objekten angewendet, ist die Systemvariable `$P{BOS}` standardmäßig `null`. Die zugrundeliegende Liste muss explizit als Parameter in den Subreport übergeben werden.

Die Sortierung im Detail-Bereich eines Subreports funktioniert nicht durch die Übergabe einer vorsortierten BO-Liste. Die Sortierung muss zwingend über ein `<sort>`-Tag in der Ankerdefinition des Subreports erzwungen werden.

Bei mehrfacher Schachtelung müssen alle tieferliegenden Reports und virtuellen Eigenschaften bereits im Strukturelement des Hauptreports definiert sein. Alle Report-Einstellungen und Parameter müssen manuell über alle Hierarchie-Ebenen durchgereicht werden.

5.3.7. Externe Layouting-Tools und DTD

Zum grafischen Bearbeiten der Layouts darf ausschließlich das externe Tool iReport in der Version 2.0.5 verwendet werden. Die Kompatibilität muss in iReport strikt auf „JasperReports 2.0.0 - 2.0.1“ limitiert werden, da neuere XML-Formate nicht unterstützt werden.

Um in Entwicklungsumgebungen eine Autovervollständigung für das Jasper-XML zu gewährleisten, kann eine DTD-Deklaration eingebunden werden.

```
<!DOCTYPE jasperReport PUBLIC "//JasperReports//DTD Report
Design//EN"
"http://jasperreports.sourceforge.net/dtds/jasperreport.dtd">
```

Meldet die DTD-Validierung den Fehler `No such accessible method: addElement() on object`, fehlt keine Methode im System. Dieser Fehler weist darauf hin, dass ein XML-Tag in einer falschen Reihenfolge auf Geschwisterebene platziert wurde.

5.3.8. Codebausteine und Struktursynchronisation

Um redundanten XML-Quelltext zu vermeiden, nutzt das System Codebausteine. Dies sind eigenständige Strukturelemente, in die wiederkehrende Layout-Fragmente zentral ausgelagert werden. Die Definition des Quelltextes wird vom Wurzelelement

`<Codebaustein>` umschlossen. Diese Bausteine lassen sich per Referenz über das Tag `<Include>` in andere Reports einbinden. Sollen dynamische Werte übergeben werden, werden diese als Parameter deklariert und innerhalb des Bausteins über die Syntax `$IP{…}` aufgelöst.

```
<Codebaustein>
    <textField>
        <textFieldExpression><![CDATA["Klinik: " +
$IP{KlinikName}]]></textFieldExpression>
    </textField>
</Codebaustein>
```

```
<Include hideComment="true" file="Pfad/Zum/Codebaustein">
    <property name="KlinikName" value="St. Johannes Hospital"/>
</Include>
```

5.3.9. Die AsciiDoc-Alternative (Narrative Reports)

Die AsciiDoc-Engine dient der Generierung stark textlastiger, narrativer Dokumente. Ein AsciiDoc-Report wird über ein XML-Skelett definiert, das zwei wesentliche CDATA-Blöcke enthält: das visuelle Erscheinungsbild (`PDFTheme`) und den Inhalt (`Asciidoc`).

Das Styling erfolgt zentral im YAML-Format innerhalb des `PDFTheme`-Blocks. Über Spalten-Definitionen positionieren Sie Logos im Header und lassen das System Seitenzahlen im Footer generieren.

```
# Auszug aus dem <PDFTheme> für Briefkopf und Footer
header:
  height: 110pt
  padding: [36pt,0,10pt,0]
  columns: <30% >40% >30%
  recto:
    left:
      content: image:$I{'KlinikLogo'}[pdfwidth=120pt,
align="left"]
    right:
      content: 'Abteilung Kardiologie'
footer:
```

```
  height: 70pt
  recto:
    center:
      content: '- Seite {page-number} -'
```

Der Inhalt des `<Asciidoc>`-Tags wird vom System als ein einziger Groovy-GString evaluiert. Der Block beginnt als Groovy-Skript zur Definition lokaler Variablen und Konstanten. Am Ende des Skripts geben Sie den Textkörper als evaluierten GString über `return """\ … """` zurück.

```
// 1. Logik-Teil: Variablen und Konstanten vorbereiten
def patientenName = bo.describe('Familienname')
def behandelnderArzt = bo.BehandelnderArzt?.describe("('Dr.
'Name2' ')(Name1)")
def datumHeute = L10n.formatDateNT(new Date(), 'dd.MM.yyyy')

// 2. Text-Teil: Den eigentlichen Report als formatierten GString
zurückgeben
return """\
:notitle:

Der Patient ${patientenName} wurde am ${datumHeute} von
${behandelnderArzt} untersucht.
"""
```

Der GString kann unterbrochen werden, um Relationen iterativ als Tabellenzeilen zu generieren und das Ergebnis anschließend zusammenzufügen.

```
// Den GString für eine dynamische Tabelle unterbrechen und
zusammenkleben
""" +
(bo.Medikamente.values().collect { med ->
"""| ${med.Wirkstoff} | ${med.Dosierung} | ${med.Hinweis}"""
}.join('\n')) + """
```

Für fixe Layout-Strukturen, wie Unterschriftenblöcke, werden randlose, unsichtbare Tabellen genutzt. Für die Einrückung von Texten empfiehlt sich die Deklaration eigener Variablen für geschützte Leerzeichen am Dokumentenanfang.

```
// Eigene Variablen für saubere Einrückungen definieren
:indent1: {nbsp}
:indent4: {indent1}{indent1}{indent1}{indent1}

// Randlose Tabelle für den Unterschriften-Block nutzen
[frame="none" cols="<40%,^20%,>40%" grid="none"]
|===
|Datum: ${datumHeute}                              ||Unterschrift
des Arztes
|
||image:$I{bo.BehandelnderArzt.Unterschrift}[]
|Ort: ${bo.KlinikStandort}
||${behandelnderArzt}
|===
```

Da der AsciiDoc-Parser reservierte Zeichen nutzt, müssen MyTISM-spezifische Erweiterungen teilweise manuell über Pass-Macros geschützt werden:

- Um HTML-Formatierungen direkt im Text zu erzwingen, nutzen Sie das Inline-Pass-Macro:

  ```
  +++<font size=".5em">Kleine Wörter</font>+++
  ```

- Wenn Sie AsciiDoc-Variablen innerhalb eines HTML-Pass-Macros verwenden, muss zwingend die Syntax `pass:a[…]` genutzt werden:

  ```
  :red: #ff0000

  pass:a[<color rgb="{red}">Fehler!</color>]`
  ```

Sollen Sternchen innerhalb eines fettgedruckten Textes erscheinen, müssen diese mit Pluszeichen geschützt werden (`+*+`).

Bilder werden mittels `$I{…}` integriert, wobei der Ausdruck zwischen den Klammern als Groovy-Code ausgewertet wird.

```
// Dynamisches Wasserzeichen basierend auf dem BO-Status
background-image: $I{bo.isDraft() ? 'EntwurfWasserzeichen' :
```

```
'NO_IMG'}
```

Die AsciiDoc-Engine unterstützt derzeit kein natives Listen-Reporting über mehrere voneinander unabhängige Objekte. Dieses Verhalten erfordert in AsciiDoc manuelles Scripting im Layout.

5.4. Architektur, Backend & Deep-Dive

Das Backend abstrahiert die Datenbeschaffung bei der serverseitigen Generierung von Reports vollständig. Das System interpretiert die XML-basierte Anker-Definition, um den Objektgraphen über das ORM-Framework in den Speicher zu laden.

5.4.1. Der Report-Lifecycle für Entwickler

Der Lifecycle der Engine gliedert sich in vier strikte Phasen. In der Design-Phase wird der Blueprint im Designer erstellt. Während der Kompilierungs-Phase wandelt die Engine das XML in eine prozedurale Java-Klasse um, wobei Syntaxfehler zum Abbruch führen. In der Befüllungs-Phase iteriert die Engine zeilenweise durch den Datenstrom der `BOsDataSource`. Zuletzt transformiert die Export-Phase die befüllte Report-Instanz in das finale Zielformat.

5.4.2. Programmatischer Aufruf (`PrintingServices`)

Reports können von Backend-Entwicklern interaktionsfrei in Hintergrundskripten oder serverseitigen Actions generiert werden. Die zentrale API für diese Automatisierung ist die Klasse `de.ipcon.db.report.PrintingServices`. Um den Druckprozess einzuleiten, muss das Report-Objekt über seine TID und einen neuen Datenbank-Transaktionskontext geladen werden.

In interaktiven Actions kann die Client-API-Methode `ctx.showStringInputDialog()` genutzt werden, um Parameter zur Laufzeit abzufragen. Benutzereingaben können vor der Übergabe typsicher in reguläre Java-Datumsobjekte konvertiert werden.

```
import de.ipcon.db.core.Report
import de.ipcon.db.report.PrintingServices
import de.ipcon.db.core.BO
import de.ipcon.tools.DateTimeTools

// Interaktive Parameter-Abfrage im Client vor der Ausführung
def station = ctx.showStringInputDialog("Für welche Station soll
die Liste generiert werden?", "Parameter")
```

```
def stichtag = L10n.parseDate("2026-05-11", "yyyy-MM-dd")
def monatsanfang = DateTimeTools.getFirstDayOfMonth(stichtag,
false)

// Report-Objekt über TID in einer neuen Transaktion laden
def reportTID = "MCS_MEDIKAMENTEN_BESTAND"
def report = Report.byTid(ctx.getNewTransaction(), reportTID)

if (!report) {
   throw new IllegalArgumentException("Es muss zwingend ein
Report mit der TID '$reportTID' existieren.")
}

// Parameter-Map und die abzufragende BO-Liste vorbereiten
def params = ['Station': station, 'Stichtag': stichtag,
'Monatsanfang': monatsanfang]
def medikamente = ntx.queryBO("Medikament bo where not Ldel and
Bestand < Mindestmenge")

// Report-Engine triggern und das BO-Array explizit casten
PrintingServices.printReport(report, params, medikamente as BO[])
```

5.4.3. Bindings und Variablen-Scopes im Backend

Die Engine erzwingt in den XML-Skript-Blöcken strenge Variablen-Bindings und isolierte Kontext-Scopes. Im Scope der Anker-Query steht ausschließlich die Variable `loader` zur Verfügung. Innerhalb der Definition von virtuellen Eigenschaften sind die Variablen `bo` und `log` injiziert. Bei Listen-Reports stellt die Engine das Basis-Array der iterierten Objekte über den Parameter `$P{BOS}` bereit.

In Report-Expressions dürfen niemals einzeilige Java-Kommentare verwendet werden. Da die Report-Generierung intern prozedurale Java-Klassen erzeugt, bricht dieser Code um und verursacht Syntaxfehler auf Backend-Ebene.

Beim Aufruf virtueller Properties im Report darf niemals der reguläre Java-Getter genutzt werden. Es muss zwingend die property-basierte Kurzschreibweise verwendet werden, da das Backend andernfalls beim Parsen eine Exception wirft.

Beim Zugriff auf temporäre Eigenschaften (Transient Properties) ist die Kurzschreibweise streng verboten und führt zu Abstürzen. In Blöcken wie `<addVirtualProperty>` muss zwingend der vollständige Getter verwendet werden.

In den Definitionen für `<addVirtualProperty>` dürfen keine Klasseneinschränkungen oder Typisierungen programmiert werden, da dies zu Parsing-Fehlern führt.

Innerhalb der Getter-Methoden von virtuellen Attributen darf im Report-Code niemals eine neue Datenbank-Transaktion aufgerufen werden. Zum Nachladen von Daten muss stets ausschließlich der angebundene Loader verwendet werden.

5.4.4. Datenquellen (`BOsDataSource`) und Diagramm-Customizer

Um Datenmengen zielgerichtet an JFreeChart-Diagramme oder Subreports zu übergeben, muss der Backend-Entwickler den Datenstrom manuell in eine `BOsDataSource` kapseln.

```
// Initialisierung einer BOsDataSource für die Laborwerte des
Patienten
new de.ipcon.db.report.BOsDataSource($F{THIS}.Laborwerte,
$P{LaborwerteSubreportBO})
```

Sollen JFreeCharts visuell angepasst werden, kann der Code über eine dedizierte Java-Klasse programmatisch modifiziert werden. Diese Customizer-Klasse muss zwingend von `JRAbstractChartCustomizer` erben.

```
import net.sf.jasperreports.engine.JRAbstractChartCustomizer;
import net.sf.jasperreports.engine.JRChart;
import org.jfree.chart.JFreeChart;

public class BettenauslastungCustomizer extends
JRAbstractChartCustomizer {
   public void customize(JFreeChart chart, JRChart jasperChart) {
       // Spezifische Backend-Logik für erweiterte Diagramm-
Manipulation
   }
```

```
}
```

Für komplexe Sortierungen innerhalb von Crosstab-Buckets kann ein maßgeschneiderter Comparator in das Report-XML injiziert werden. Für Layout-Anpassungen von Kreuztabellen bietet JasperReports Attribute wie `whenNoDataCell` oder `crosstabCell[isBlankWhenNull]`.

```
<comparatorExpression><![CDATA[new OrderBy({
it.getTransientProperty('someProperty')
})]]></comparatorExpression>
```

Werden Kreuztabellen im direkten Detail-Band des Reports gerendert, führt dies zu einem schweren Berechnungsfehler in der Generierungs-Pipeline. Als Workaround muss die Crosstab zwingend über ein eigenständiges `subDataset` entkoppelt werden.

5.4.5. Asciidoc-Reporting: Evaluation und BLOB-Integration

Die alternative AsciiDoc-Engine evaluiert den gesamten Inhalt des `<Asciidoc>`-Tags als einen einzigen Groovy-GString. Dem Report übergebene Parameter werden im AsciiDoc-Skript automatisch in Groovy-Variablen mit dem Präfix `param_` transformiert.

Die Verarbeitung erfolgt in zwei getrennten Stufen. Zuerst löst das System den GString auf, berechnet Variablen und fügt normale Datenfelder ein. Im letzten Schritt werden spezielle Bildausdrücke ausgewertet. Dies erlaubt es, die Kriterien für Bilder dynamisch über Variablen zusammenzusetzen, bevor die Bilddatei geladen wird. Bilder, die als binäre BLOBs in der Datenbank liegen, können nativ evaluiert und in den Text gerendert werden.

```
// Fetch dynamic blob images via OQL from the database inside
Asciidoc
$I{bo.getBOLoader().queryBO('Bild a WHERE NOT Ldel and Name = ${-
> "Unterschrift_$bo.BehandelnderArzt.Nachname"}')}
```

5.4.6. Spezifische Warnhinweise & Edge Cases (Backend)

Eine `IllegalArgumentException` mit der Meldung „Invalid parameter 'xyz' given“ schützt das System vor invaliden Parametern, die in der XML-Definition nicht deklariert wurden. Ein Sonderfall dieser Exception tritt auf, wenn ein Benutzer keine ausreichenden Leserechte auf die Argumente

eines eingebundenen Codebausteins besitzt.

Werden Report-Variablen beim Druck durchgehend als `null` evaluiert, fehlt meist die Initialisierung außerhalb des Detail-Bereichs. Dieses Verhalten wird durch die Definition einer `initialValueExpression` korrigiert.

Ein L10n-Caching-Fehler bei Übersetzungen im Report entsteht durch einen Plural-Key mit fehlerhaft hinterlegtem Singular-Wert in der Datenbank. Dieser Fehler überdauert das reguläre Löschen der System-Caches und muss zwingend manuell via SQL-Update bereinigt werden.

Tritt beim Rendern von Reports mit eingebetteten Vektorgrafiken eine `ClassNotFoundException` oder `NotSerializableException` auf, signalisiert dies einen Konflikt im Jasper-Virtualizer. Um den Virtualizer zu umgehen, muss der Parameter `PrintingServices.AVOID_TEMP_FILES` an den Report übergeben werden.

Für das serverseitige Debugging fehlschlagender Report-Ausgaben werden die rohen Zwischenergebnisse unter dem Pfad `/tmp/MyTISM_ReportFiles/` abgelegt. Das System speichert dort temporäre PDF-Dateien sowie bei AsciiDoc-Reports den reinen Textkörper und das finale YAML-Styling.

Chapter 6. Lokalisierung (L10n) & Mehrsprachige Daten

6.1. Grundlagen & Die Zwei-Säulen-Architektur

MyTISM ist als vollständig mehrsprachiges System konzipiert. Die Architektur trennt dabei bei der Bedienung konsequent zwischen der Programmsprache und der Datensprache. Um diese Trennung technologisch sauber abzubilden, basiert das System auf einer strikten Zwei-Säulen-Architektur. Diese beiden Säulen lösen grundlegend unterschiedliche Anwendungsfälle und nutzen isolierte technologische Fundamente:

- **Säule 1: Die Datenlokalisierung (Datensprache / L10nString):** Diese Säule ist für die Verwaltung dynamischer, vom Benutzer erfasster Daten direkt in der Datenbank zuständig (z. B. medizinische Befunde oder Diagnosetexte). Sie wird über das XML-Datenschema gesteuert, speichert Schlüssel-Wert-Paare in PostgreSQL-`hstore`-Strukturen und stellt über virtuelle Aliase (`L10nedString`) eine transparente Zugriffsschicht bereit.
- **Säule 2: Die UI- und Systemlokalisierung (Programmsprache / Resource Bundles):** Diese Säule steuert alle Oberflächen- und Systemtexte der Benutzeroberfläche wie Schaltflächen, Menüeinträge, Tabellenüberschriften, Fehlermeldungen und System-Benachrichtigungen. Obwohl diese Texte zentral in Bündeln vordefiniert sind, können sie durch Parameter-Interpolation und Platzhalter zur Laufzeit hochgradig dynamisch zusammengesetzt werden. Sie basiert auf dem klassischen Java-Ressourcenkonzept und nutzt entweder dateibasierte `.properties`-Dateien (im CVS-Repository) oder dynamische, in der Datenbank verwaltete Übersetzungsbündel (`L10nBundle`).

Durch diese technologische Trennung kann die Benutzeroberfläche beispielsweise auf Spanisch bedient werden, während die medizinischen Akten gleichzeitig in deutscher Datensprache bearbeitet werden.

MyTISM überwindet die starren Limitierungen klassischer Java-Anwendungen durch einen hybriden Ansatz. Das System vereint statische Textdateien mit einer hochperformanten Datenbank-Persistenz. Übersetzungsdaten können live im laufenden System über die grafische Oberfläche editiert oder neu angelegt werden. Wird ein übersetzter Warnhinweis angepasst, ist diese Änderung sofort für alle Nutzer wirksam, ohne dass der Server aktualisiert und neu gestartet oder Verbindungen getrennt werden müssen.

6.2. Säule 1: Datenlokalisierung (Datensprache & Schema-Integration)

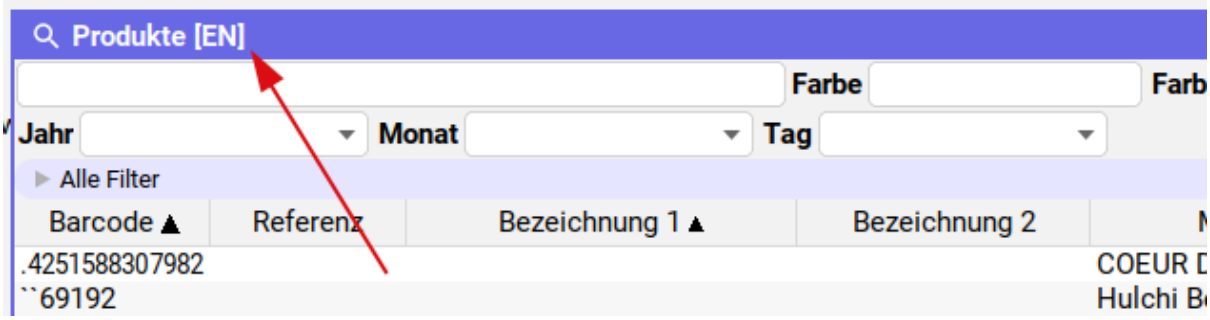
6.2.1. Benutzeroberfläche & Bedienung der Datensprache

Die aktive Datensprache bestimmt, in welcher Sprache die Werte in Lesezeichen, Listen und

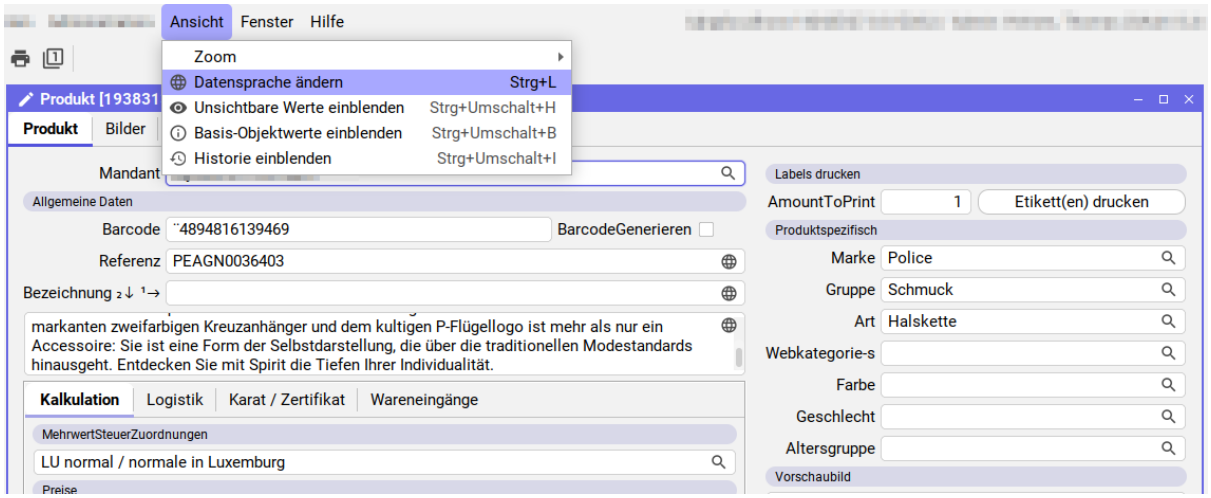
Formularen angezeigt werden. Sie lässt sich über das Menü „Ansicht → Datensprache ändern“, den Shortcut „Strg+L“ oder das Globus-Symbol in der Werkzeugleiste anpassen.

Wenn Sie einen Datensatz öffnen, werden die Texte zunächst in der aktuell aktiven Sprache angezeigt.

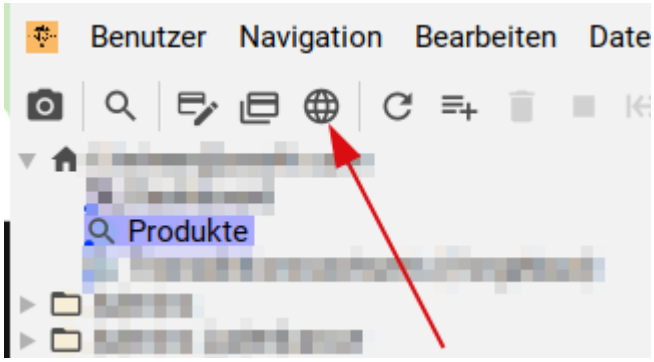
Weicht die aktive Datensprache von der Programmsprache ab, wird sie zur visuellen Kontrolle stets im Fenstertitel jedes Formulars und Lesezeichens in eckigen Klammern (z. B. „[DE]“) angezeigt.



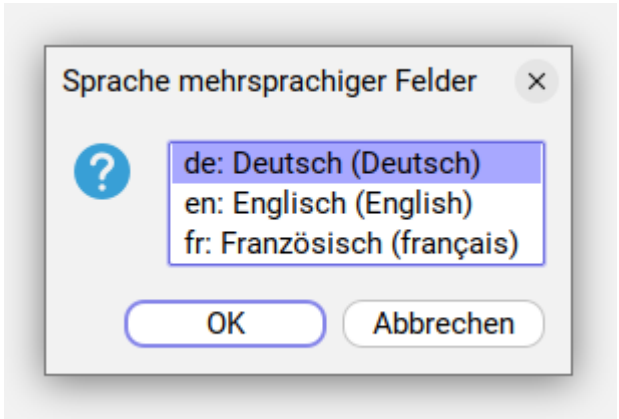
Der Menüeintrag „Datensprache ändern“ im Menü „Ansicht“ öffnet den Dialog zur Sprachauswahl.



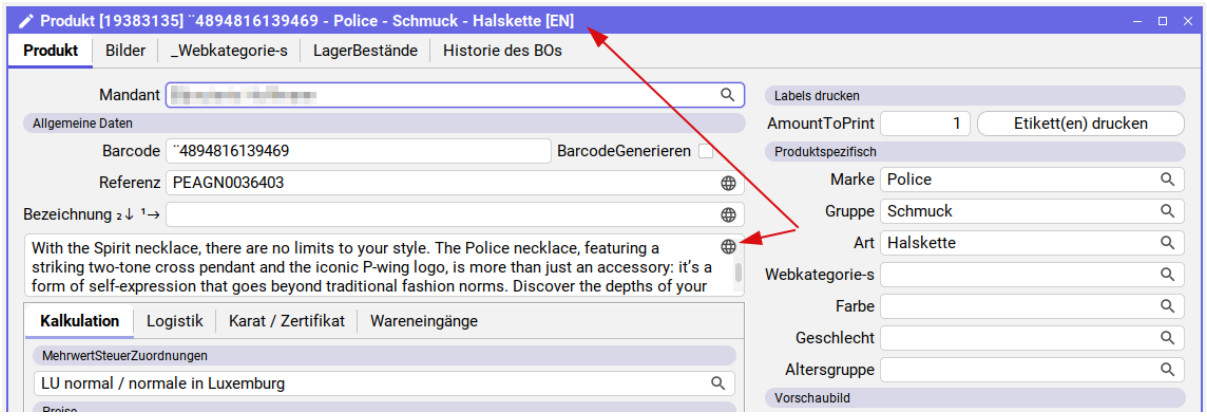
Ebenso erreichen Sie diesen Dialog über den Action-Button mit dem Globus-Symbol in der Werkzeugleiste.



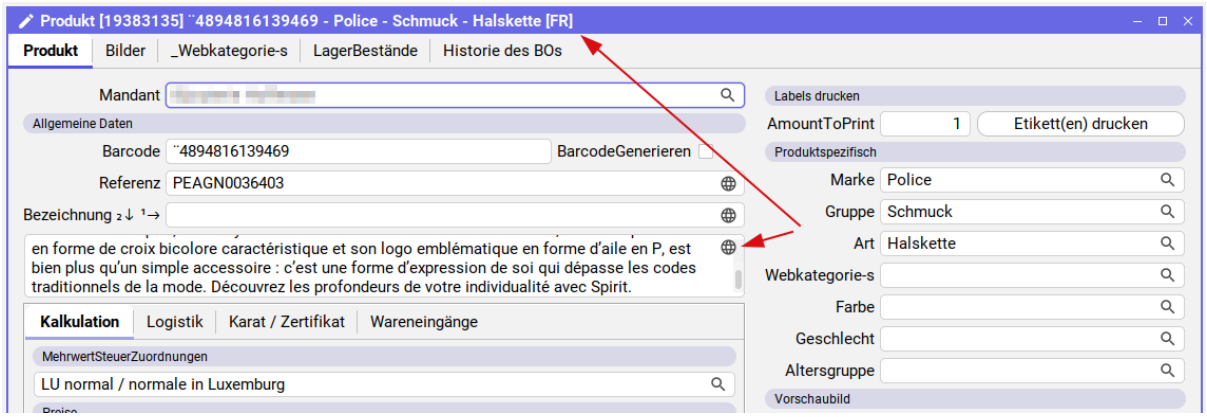
Daraufhin öffnet sich ein separates Dialogfenster zur Sprachauswahl.



Wählen Sie in diesem Dialog die gewünschte Zielsprache aus und bestätigen Sie Ihre Eingabe. Sobald Sie die Sprache gewechselt haben, aktualisiert sich der Indikator im Fenstertitel. Gleichzeitig laden alle mehrsprachigen Felder im Formular oder Lesezeichen automatisch die Texte der neu gewählten Sprache.



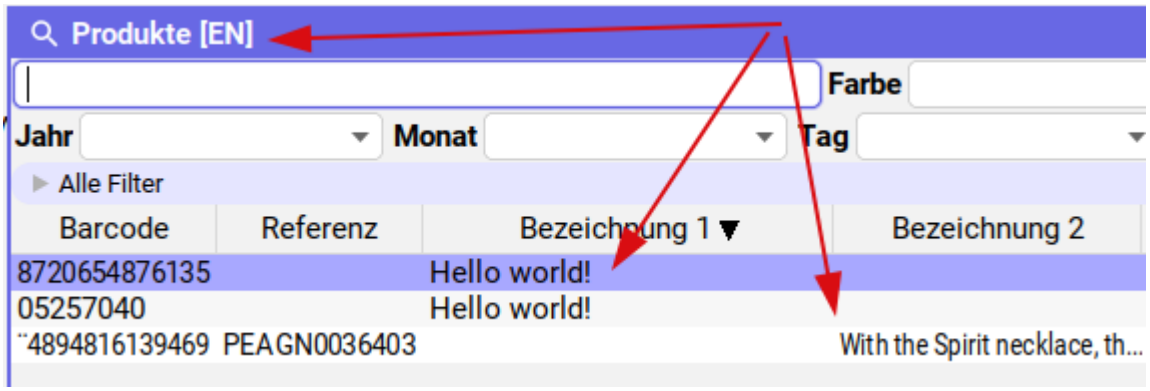
Wechseln Sie die Sprache erneut, passen sich der Fenstertitel und die Inhalte sofort wieder entsprechend an.



Alle Eingaben, die Sie in mehrsprachigen Feldern tätigen, werden automatisch unter dieser neuen Sprache gespeichert.

Lesezeichen passen die angezeigten Werte in den Spalten für mehrsprachige Felder vollautomatisch an die aktuell gewählte Datensprache an. In der deutschen Spracheinstellung werden Ihnen die entsprechenden Daten somit auf Deutsch angezeigt.

Wechseln Sie über das Globus-Symbol oder das Ansicht-Menü in den englischen Modus, werden die Daten im Lesezeichen entsprechend mit ihren englischen Übersetzungen angezeigt.



Wird ein Datensatz aus einem englischsprachigen Lesezeichen geöffnet, stellt das System das resultierende Formular automatisch auf Englisch ein. War der Datensatz bereits in einem anderen Fenster geöffnet, rückt dieses in den Vordergrund und übernimmt sofort die neu gewählte Spracheinstellung. Dies gewährleistet eine durchgängige Konsistenz zwischen Lesezeichen und Formular, sodass Sie auch bei einem Sprachwechsel stets die richtigen Einträge pflegen.

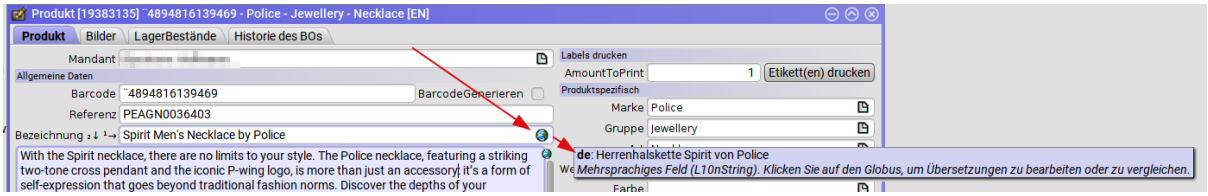
Um diese Pflege im Formular zu erleichtern, müssen Sie auf einen Blick erkennen können, welche Datenfelder überhaupt übersetzbar sind. Nicht jedes Feld in einem Formular ist nämlich mehrsprachig, da beispielsweise eine Artikelnummer oder ein Preis sprachunabhängig ist. Zur schnellen Erkennung sind alle übersetzbaren Felder mit einem Globus-Symbol markiert.

Das System passt die Darstellung des Symbols automatisch an die Art des Eingabefeldes an, um das Formular übersichtlich zu halten und den Lesefluss nicht zu stören.

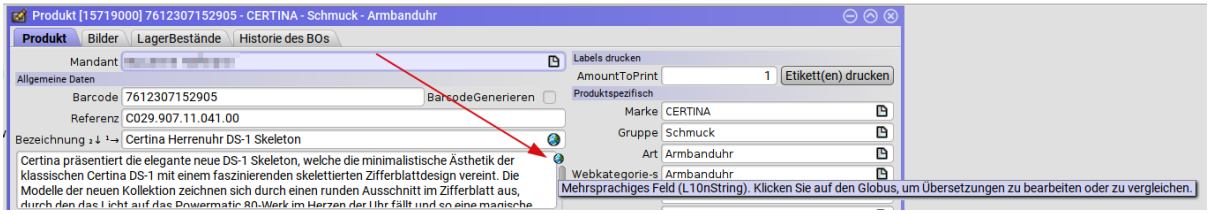
Bei einzeiligen Textfeldern wird der Globus platzsparend am rechten Rand direkt neben dem Eingabefeld platziert, ohne den eingegebenen Text zu verdecken.

Bei mehrzeiligen Textbereichen (Text Areas) befindet sich der Globus dezent rechts oben neben dem Eingabebereich. Dadurch wird kein vertikaler Platz verschwendet und das Symbol bleibt auch beim Herunterscrollen sichtbar.

Ein Hover über das Globus-Symbol blendet direkt den Text der Standardsprache in einem Tooltip ein, sofern ein solcher existiert. Zusätzlich liefert der Tooltip am unteren Rand direkt einen praktischen Hinweis zur Bedienung des Symbols.



Sollte für das Feld noch kein Text in der Standardsprache hinterlegt worden sein, weist der Tooltip stattdessen nur explizit auf die Funktion des Symbols hin.

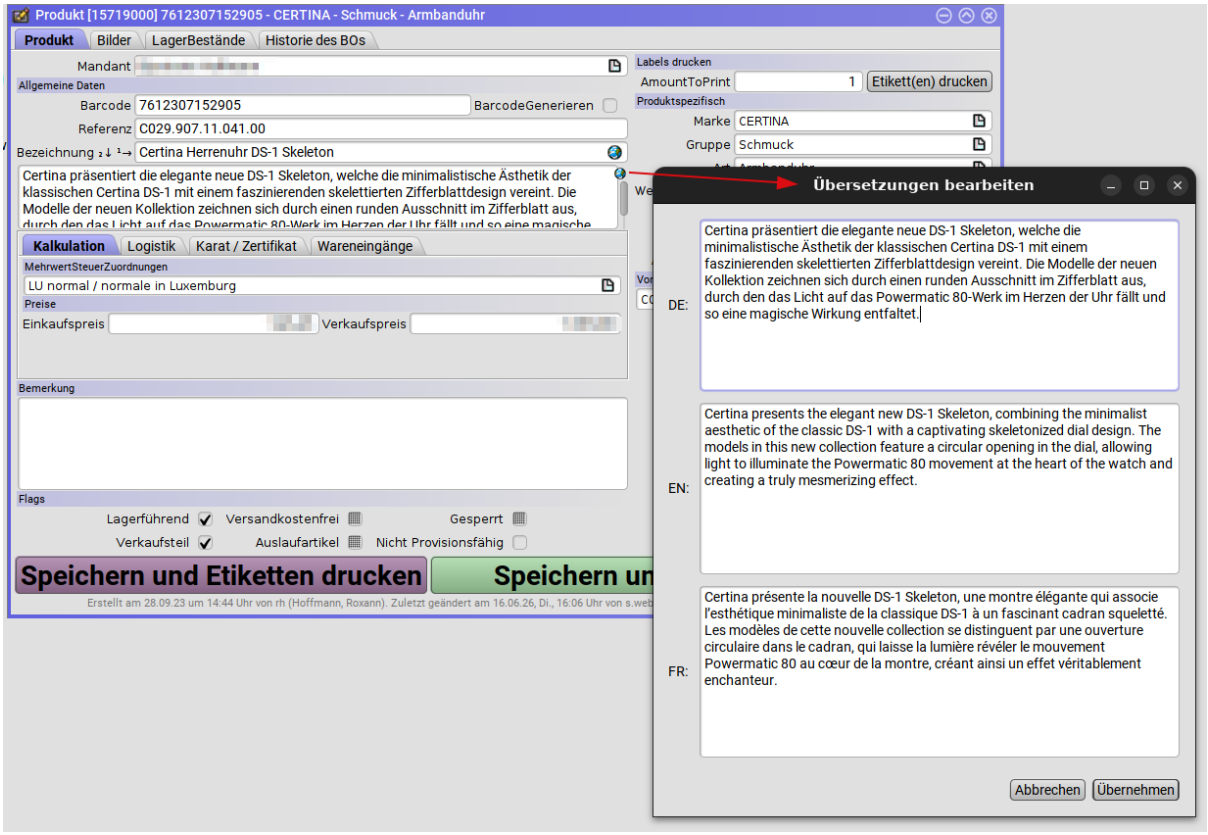


6.2.2. Die zwei Wege der Übersetzungspflege (Workflows)

Für die Erfassung und Pflege mehrsprachiger Daten unterstützt MyTISM zwei unterschiedliche, sich ergänzende Arbeitsabläufe, die für verschiedene Alltagsszenarien optimiert sind.

Der erste Workflow („Globale Erstanlage“) eignet sich ideal für die vollständige Erfassung neuer Datensätze. Hierbei stellen Sie die globale Datensprache der Maske einmalig auf die gewünschte Zielsprache ein, um anschließend alle Felder des Formulars nacheinander in einem Rutsch für diese Sprache zu befüllen.

Der zweite Workflow („Gezielte Detailkorrektur“) kommt bei der Bearbeitung bereits bestehender Daten zum Einsatz. Müssen Sie lediglich ein einzelnes Feld (wie die Artikelbeschreibung) in mehreren Sprachen korrigieren oder ergänzen, wäre das globale Umschalten der gesamten Maske zu aufwändig. Für diese punktuelle Pflege öffnen Sie über einen Klick auf das Globus-Symbol eines Feldes direkt den interaktiven Schnell-Editor.



Dieser Schnell-Editor ist für einen effizienten und flexiblen Arbeitsablauf konzipiert:

- **Strukturierte Übersicht:** Die verfügbaren Sprachen werden zur schnellen Orientierung immer konsistent sortiert (z. B. nach „DE“, „EN“, „FR“) untereinander aufgelistet.
- **Fokussierte Modalität:** Der geöffnete Dialog blockiert immer nur das aktuell zugehörige Dokument. Sie können den Editor jederzeit zur Seite schieben und parallel in anderen geöffneten Lesezeichen oder Formularen der Applikation arbeiten, ohne ihn schließen zu müssen. Klicken Sie versehentlich auf das blockierte Ursprungsdokument, rückt sich der Editor automatisch wieder in den Vordergrund, um Ihnen den fehlenden Fokus zu signalisieren.
- **Dynamische Größenanpassung:** Die Eingabebereiche passen ihre Höhe automatisch an die Länge des längsten vorhandenen Textes an. Bei sehr langen Texten (z. B. ausführlichen Artikelbeschreibungen) schalten sich automatisch Scrollbars ein, und die Gesamtgröße des Fensters wird sicher gedeckelt, damit der Dialog niemals die Bildschirmgrenzen sprengt.
- **Direktes Editieren:** Sobald sich der Dialog öffnet, wird der Cursor automatisch im ersten Textfeld platziert, sodass Sie sofort lostippen können. Die Navigation zwischen den Sprachen funktioniert dabei nahtlos per Tastatur über die `Tab`-Taste (vorwärts) bzw. `Shift+Tab` (rückwärts).
- **Änderungen übernehmen:** Ein Klick auf die Schaltfläche „Übernehmen“ (oder das Drücken von `Strg+Enter`) überträgt alle geänderten Texte zurück in die Maske. Die Daten werden im Objekt aktualisiert und erst bei der nächsten regulären Speicherung der Hauptmaske final in die Datenbank geschrieben.
- **Sicheres Abbrechen & Warnsystem:** Möchten Sie Ihre Änderungen verwerfen, können Sie den Dialog über die Schaltfläche „Abbrechen“, das Fensterkreuz („X“) oder durch Drücken der `Escape`-Taste schließen. Haben Sie bereits Texte eingetippt oder verändert, fängt das System den Schließvorgang automatisch ab und bittet um eine explizite Bestätigung. Dies verhindert, dass aufwändige Übersetzungsarbeiten durch einen versehentlichen Klick verloren gehen.

6.2.3. OQL-Filter für Power-User

Für die Auswertung mehrsprachiger Daten in Lesezeichen stellt die Object Query Language (OQL) erweiterte Filter-Funktionen bereit. Da mehrsprachige Felder in der Datenbank intern als Schlüssel-Wert-Paare verarbeitet werden, bietet OQL leistungsstarke Spezial-Operatoren, um diese zu durchsuchen.

Direkter Zugriff auf eine spezifische Sprache

Um explizit im Text einer ganz bestimmten Sprache zu suchen, hängen Sie das gewünschte Sprachkürzel in eckigen Klammern und einfachen Anführungszeichen direkt an den Feldnamen an. Dieser direkte Zugriff liefert einen einfachen Textwert zurück, auf den Sie

alle regulären OQL-Operatoren (=, !=, <, >, ilike, matches) anwenden können.

```
// Exakter Treffer in der deutschen Übersetzung
[Diagnosen['de'] = 'Fieber']

// Wildcard-Suche (ilike) in der englischen Übersetzung
[Diagnosen['en'] ilike '%fever%']

// Überkreuz-Vergleich in mehreren Sprachen mit der OQL-
Kurzschreibweise
[ANY OF (Diagnosen['de'], Diagnosen['en']) = 'Fieber']
```

Globale Suche über alle Sprachen

Wenn Sie nicht wissen, in welcher Sprache ein Begriff hinterlegt wurde, können Sie gezielt alle Werte eines Feldes durchsuchen. Hierfür stellen die mehrsprachigen Felder das virtuelle Suffix .values zur Verfügung. In Kombination mit Text-Operatoren wie ilike oder like übernimmt der OQL-Smart-Parser automatisch die Umwandlung in die korrekte Datenbankabfrage.

```
// Sucht den Begriff in allen vorhandenen Übersetzungen der
Diagnose
[Diagnosen.values ilike '%covid%']
```

Existenzprüfungen und Mengenvergleiche

Zusätzlich zum .values-Suffix existiert das .keys-Suffix, um gezielt die vorhandenen Sprachschlüssel eines Feldes zu prüfen. Zusammen mit den Mengen-Operatoren HAS und LACKS können Sie exakt analysieren, welche Übersetzungen gepflegt wurden und welche fehlen.

```
// Prüfung, ob für diese Diagnose eine deutsche Übersetzung
existiert
[Diagnosen.keys HAS 'de']

// Prüfung auf eine definitiv fehlende französische Übersetzung
[Diagnosen.keys LACKS 'fr']

// Findet Diagnosen, die sowohl eine deutsche als auch eine
```

```
englische Übersetzung haben
[Diagnosen.keys HAS ALL OF ['de', 'en']]

// Findet Diagnosen, in denen der Wert 'Fieber' oder 'Fever' in
irgendeiner Sprache vorkommt
[Diagnosen.values HAS ANY OF ['Fieber', 'Fever']]
```

Eine OQL-Abfrage auf `= null` bei mehrsprachigen Feldern ist logisch mehrdeutig und führt zu Fehlern. Die Abfrage `[Wirkstoffe['de'] = null]` trifft fälschlicherweise sowohl dann zu, wenn der deutsche Text explizit gelöscht wurde, als auch, wenn der Schlüssel „de“ überhaupt nicht existiert. Um auf das Fehlen einer Sprache zu filtern, muss der Mengenoperator `LACKS` in Kombination mit dem `.keys`-Suffix verwendet werden. Für den Spezialfall, bei dem ein Schlüssel existiert, dessen Wert aber auf `null` gesetzt wurde, muss zwingend `[Wirkstoffe.keys HAS 'de' AND Wirkstoffe['de'] = null]` genutzt werden.

Transparente Suche über das virtuelle Alias

Zusätzlich legt das System für jedes mehrsprachige Feld automatisch ein virtuelles Attribut an, sofern im Datenschema der Parameter `defaultLocaleAlias` konfiguriert wurde. Dieses Alias-Attribut verhält sich in OQL-Abfragen exakt wie ein klassisches, einsprachiges Textfeld und durchsucht immer automatisch die Texte in der von Ihnen aktuell gewählten Datensprache. Wurde bei der technischen Umstellung eines Feldes auf Mehrsprachigkeit der alte Feldname als Alias beibehalten (beispielsweise „Wirkstoff“ statt „Wirkstoffe“), müssen Sie nichts weiter tun. Alle Ihre alten Abfragen und bereits gespeicherten Lesezeichen-Filter funktionieren nach dem Update nahtlos und völlig transparent weiter.

```
// Transparente Suche über das Alias in der aktuell aktiven
Datensprache
[Wirkstoff ilike '%Ibuprofen%']
```

6.2.4. Generierter Java-Code für L10nString- und L10nedString-Attribute

Für jedes im XML-Schema definierte mehrsprachige Feld generiert das System spezifische Zugriffsmethoden in den Entity-Klassen. Dabei wird technisch strikt zwischen dem persistenten physischen Map-Attribut (Typ `L10nString`, z. B. `Bezeichnungen1` im Plural) und dem virtuellen Alias-Attribut (Typ `L10nedString`, z. B. `Bezeichnung1` im Singular) unterschieden.

Zugriff auf das physische Attribut (L10nString)

Das physische Attribut hält die tatsächliche `Map<String, String>` im Arbeitsspeicher, welche in PostgreSQL als `hstore` abgebildet wird. Folgende Methoden werden für das physische Attribut generiert, um die Übersetzungen direkt zu verwalten:

- **Getter:** `public Map<String, String> getBezeichnungen1()`
  Gibt eine unmodifizierbare Ansicht der Map mittels `Collections.unmodifiableMap()` zurück.
- **Putter:** `public String putBezeichnungen1(String key, String value)` sowie `public String putBezeichnungen1(Map.Entry<String, String> entry)`
  Fügt eine Übersetzung für einen spezifischen Sprachschlüssel (z. B. „de“ oder „en_US“) hinzu oder überschreibt eine bestehende. Diese Methode übernimmt das vollständige Dirty-Checking, stößt bei aktiver Transaktion die Änderungserfassung an und inkrementiert die Version des Business-Objekts. Sie gibt den vorherigen Wert für diesen Sprachschlüssel zurück oder `null`, falls noch kein Eintrag existierte.
- **Dropper:** `public String dropBezeichnungen1(String key)`
  Entfernt den Eintrag für den angegebenen Sprachschlüssel aus der Map, stößt die Änderungserfassung an und gibt den gelöschten Text zurück.
- **Standard-Setter:** `public void setBezeichnungen1(Map<String, String> map)`
  Dient dem vollständigen Kopieren einer Map, indem intern über alle Einträge der übergebenen Map iteriert und für jeden `putBezeichnungen1(entry)` aufgerufen wird.

Zugriff auf das virtuelle Alias-Attribut (L10nedString)

Wird ein `defaultLocaleAlias` definiert, erzeugt das Framework ein virtuelles, flaches Text-Attribut. Dieses vereinfacht den lesenden und schreibenden Zugriff auf die Texte unter automatischer Berücksichtigung des aktuellen Kontextes:

- **Lokalisierter Getter mit Locale-Objekt:** `public String getBezeichnung1(Locale locale)`
  Liefert den übersetzten Wert mit Fallback-Logik (z. B. von „de_DE“ auf „de“) über die Hilfsklasse `L10n.getLocalizedWithFallback()`.
- **Lokalisierter Getter mit Sprachkürzel:** `public String getBezeichnung1(String localeString)`
  Verhält sich identisch zum Locale-Getter, akzeptiert jedoch das Sprachkürzel direkt als String.
- **Standard-Getter:** `public String getBezeichnung1()`
  Gibt den Text für das aktuell aktive Standard-Locale zurück, welches über `getBOLoader().getDefaultLocale()` bezogen wird.
- **Standard-Setter:** `public void setBezeichnung1(String value)`

Schreibt den übergebenen Text für das aktuell aktive Standard-Locale des Benutzers in die zugrunde liegende Map, indem es intern `putBezeichnungen1(defaultLocale, value)` aufruft.

> Es existiert kein lokalisierter Setter auf dem virtuellen Alias-Attribut (z. B. `setBezeichnung1(String value, Locale locale)`). Möchten Sie als Entwickler eine spezifische Übersetzung programmatisch schreiben, müssen Sie zwingend den Putter des physischen Plural-Attributs aufrufen (z. B. `putBezeichnungen1("fr", "valeur")`). Der Aufruf des Standard-Setters `setBezeichnung1("Wert")` schreibt immer ausschließlich in die aktuell aktive Datensprache des Loaders / Schemas.

Ermittlung des Standard-Locales und UI-Ereignissteuerung (Deep-Dive)

Auf Systemebene wird das für einen Ladeprozess aktive Standard-Locale dynamisch über den `BOLoaderI` ermittelt. Die Methode `getDefaultLocale()` folgt dabei einer klar definierten, mehrstufigen Kaskade, um eine robuste Fallback-Sicherheit im System zu garantieren.

Die Auflösung des Locales erfolgt in drei priorisierten Stufen:

- **Explizites Schema-Locale:** Zuerst wird geprüft, ob im aktiven `InstrumentingSchemaI` ein spezifisches Locale (über `setLocale()`) hinterlegt wurde.
- **Datenbank-Fallback:** Ist dort kein Wert definiert, fällt das System auf das aktuell als Standard definierte `L10nLocale` zurück.
- **JVM-Standardwert:** Als letzte Instanz dient das globale Standard-Locale der virtuellen Java-Maschine (JVM), welches über `L10n.getDefaultLocale()` bezogen wird.

Das `InstrumentingSchemaI` deklariert das Registrierungs- und Benachrichtigungssystem für dieses Kaskaden-Locale. Eine Änderung der Datensprache über `setLocale(Locale)` löst ein Änderungsereignis aus, welches an alle registrierten Instanzen des Typs `SchemaLocaleChangeListenerI` propagiert wird. Das System benachrichtigt diese Listener nacheinander über die Methode `schemaLocaleChanged(oldLocale, newLocale)`.

Damit sich Formularelemente und Lesezeichendaten dynamisch aktualisieren, klinkt sich die Client-Architektur an zentralen Stellen des Ladezyklus in diesen Ereignisstrom ein:

- **Lesezeichen-Ausführung:** Beim Laden eines Lesezeichens registriert sich der Form-Kontext am zugehörigen `InstrumentingSchemaI`.
- **Formular-Öffnung:** Beim direkten Editieren eines Datensatzes wird der Formular-Kontext ebenfalls als Beobachter am Schema registriert, um Feldinhalte live anzupassen.

- **Desktop-Komponenten:** Desktop-Elemente, die von `AbstractClientDesktopElement` erben, prüfen nach der Initialisierung über die Hilfsmethode `anyContextHasLocalizedAttribute()`, ob überhaupt lokalisierte Daten im aktuellen Schema vorhanden sind. Ist dies der Fall, melden sie sich ebenfalls am Änderungs-Listener an.

Über diese Ereignis-Kaskade wird sichergestellt, dass jede Änderung der aktiven Datensprache unmittelbar an die grafische Benutzeroberfläche übertragen wird. Dies ermöglicht eine durchgängige Konsistenz und ein verzögerungsfreies Rendering aller betroffenen Eingabemasken.

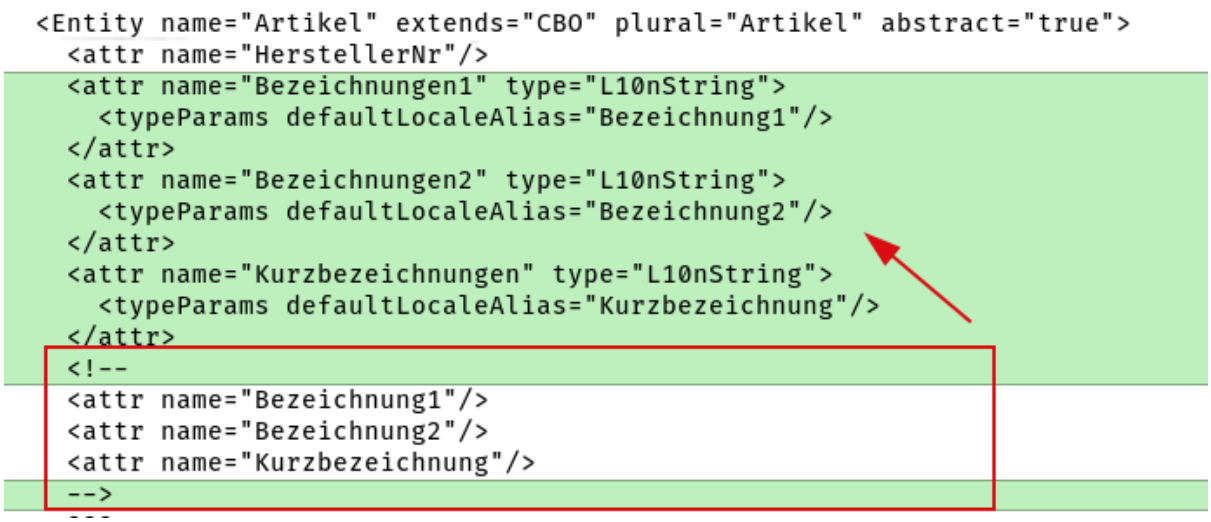
6.2.5. Datenbank-Migrationen (Hstore)

Die nachträgliche Umstellung eines bestehenden Textfeldes auf Mehrsprachigkeit erfordert eine technische Anpassung und ein Update des MyTISM-Systems. Dieser Vorgang umfasst die Änderung im zugrunde liegenden Datenschema sowie eine begleitende Migration der bereits vorhandenen Daten in der Datenbank.

1. Anpassung des XML-Schemas

Zunächst muss der Datentyp des betroffenen Attributs in der Schema-Definition von einem regulären Textfeld auf den Typ `L10nString` geändert werden. In der Praxis wird hierbei oft ein neuer Attributname (z. B. im Plural) vergeben, um die neue mehrsprachige Datenstruktur logisch von der alten, flachen Textspalte zu trennen. Über den Parameter `defaultLocaleAlias` wird ein Alias für den transparenten Lesezugriff definiert, der idealerweise exakt dem alten Feldnamen entspricht. Dadurch bleibt bestehender Programmcode, der auf dieses Feld zugreift, weiterhin kompatibel und erhält automatisch den Wert der jeweiligen Standardsprache.

```
<attr name="Bezeichnungen1" type="L10nString">
  <typeParams defaultLocaleAlias="Bezeichnung1"/>
</attr>
```



2. Datenmigration mittels Update-Skript

Mehrsprachige Felder werden in PostgreSQL intern als Schlüssel-Wert-Paare (Hstores) abgebildet. Die Migration von Bestandsdaten erfolgt im SQL-Update-Skript über die dedizierte Methode `UpdateHandlerTools.migrateStringToHstore()`. Diese Methode kopiert die Daten aus der alten Tabellenspalte und fügt sie in der neuen Spalte unter einem von Ihnen definierten Standard-Sprachschlüssel (z. B. `de`) ein.

```
// Migriert die Spalte 'wirkstoff' nach 'wirkstoffe' beim
Medikament und setzt Deutsch ('de') als Standard-Key
UpdateHandlerTools.migrateStringToHstore('medikament',
'wirkstoff', 'wirkstoffe', 'de', stmt)
```

Die Methode sorgt standardmäßig auch dafür, dass die Zielspalte automatisch in der Datenbank angelegt wird, falls der Schema-Generator im Vorfeld noch nicht gelaufen ist. Nach erfolgreicher Migration der Daten werden die alten, nun obsoleten Textspalten automatisch während des Metadatenchecks beim Serverstart aus der Datenbank entfernt.

6.3. Säule 2: UI- & System-Lokalisierung (Resource Bundles)

6.3.1. Unterstützte Bereiche für Mehrsprachigkeit

Die Lokalisierung der Benutzeroberfläche und programmseitiger Ausgaben wird an zahlreichen Stellen systemweit unterstützt. Neben dem direkten programmgesteuerten Zugriff können Platzhalter im Format `$R{key}` definiert werden, die das System vor der Darstellung zur Laufzeit durch die passende Übersetzung des aktiven Locales ersetzt.

Diese Platzhalter und Übersetzungen werden in folgenden Bereichen angewendet:

- **Benutzer-Skripte und -Einstellungen im Client Solstice:**
  - In den XML-basierten Benutzer-Login-Skripten (z. B. `<Configuration><Locale>de</Locale></Configuration>`).
  - In XML-Definitionen für Plugins innerhalb der Benutzer-Voreinstellungen.
  - In XML-Definitionen für Standardwerte (Defaults) in den Benutzer-Voreinstellungen.
  - In Report-Definitionen.
  - In Parametern von Formularen, Schablonen (Templates) und Lesezeichen.
- **Interne Client-Funktionalität:**
  - Automatische Übersetzung bei der Ausgabe des Namens (`Name`) und des Elter-Pfades (`ElterPfad`) von Objekten des Typs `Benannt` im Navigationsbaum.
  - Übersetzung im Rahmen des `PolymorphicTemplateSelectionTreeModel`.

Bei der Entwicklung in XML-Dateien (z. B. Formularen) lässt sich gezielt nach Stellen suchen, die für Übersetzungen vorbereitet sind oder `$R` -Platzhalter aufnehmen können. Typische Suchmuster hierfür sind Attribute wie `title="`, `label="` oder `text="`.

6.3.2. Schlüssel- und Pfad-Auflösung (Resolution-Kaskade)

Für die Lokalisierung halten Instanzen des Typs `L10nPackProviderI` (primär `de.ipcon.tools.L10n` auf Server- und Hilfsebene sowie `de.ipcon.db.AbstractClient` im Client) benannte Sprachpakete (`L10nPack`) im Speicher bereit. Diese Pakete gruppieren Textbausteine unterschiedlicher Sprachen unter eindeutigen Identifikatoren.

Wird ein `$R{key}`-Platzhalter aufgelöst oder `L10n.msg()` aufgerufen, ermittelt MyTISM automatisch eine Liste der beteiligten und relevanten Objekte des aktuellen Kontextes:

- Beim direkten Aufruf von `L10n.msg()` wird automatisch die aufrufende Klasse ermittelt und als einziges beteiligtes Objekt herangezogen.
- Bei Platzhaltern in Formular- oder Report-Parametern werden sowohl die Klasse des UI-Strukturelements (z. B. `Formular.class`) als auch die Klasse des im Formular dargestellten Business-Objekts (BO) übergeben.

Basierend auf diesen Objekten wird eine Liste relevanter Sprachpakete (`L10nPacks`) erstellt und kaskadierend nach dem Schlüssel durchsucht. Der erste gefundene Treffer wird zurückgeliefert.

Die Vererbungshierarchie der Paketsuche

Die Benennung und Strukturierung der Sprachpakete folgt standardmäßig der Paket- und Klassenhierarchie von Java. Sucht beispielsweise ein Objekt der Klasse `de.ipcon.form.FText` nach einem Schlüssel, wird der Pfad hierarchisch von der spezifischen Klasse über die Vererbungslinien bis zum Basispaket aufgelöst.

Das System durchsucht die Pakete in exakt dieser Reihenfolge:

1. `de.ipcon.form.FText` (die konkrete Klasse)
2. `de.ipcon.form.FPanel` (Superklasse von `FText`)
3. `de.ipcon.form` (Paket der Klasse)
4. `de.ipcon` (Übergeordnetes Paket)
5. `de` (Basis-Namespace)

Dies gilt gleichermaßen für Module: Auch Modul-Bündel werden hierarchisch entlang der Vererbungshierarchie durchsucht. Dadurch werden Redundanzen in den BO-Bündeln von Projekten, welche dieselben Module verwenden, vollständig vermieden.

Gibt es für eine Klasse einen registrierten Pfad-Compiler (Implementierung von `L10nPathCompilerI`, wie z. B. `FormularPathCompiler` in `Formular.nrx`), bestimmt dieser explizit, welche Sprachpakete für die Klasse einbezogen werden. Die interne Generierung dieser Pfadliste wird über `L10n.compilePath()` gesteuert.

Format- und Zeichen-Einschränkungen

Für eine fehlerfreie Verarbeitung gelten strikte syntaktische Regeln für Bezeichner:

- **Bündel- und Paketnamen (`L10nPack`):** Dürfen ausschließlich Buchstaben, Zahlen, Unterstriche (`_`), Bindestriche (`-`) und Punkte (`.`) enthalten. Der Punkt `.` dient dabei systemweit als reserviertes Trennglied zum Aufteilen der Namespaces.
- **Schlüsselbezeichner (`L10n-Keys`):** Dürfen ausschließlich Buchstaben, Zahlen sowie die Zeichen `_`, `-`, `.`, `~` und Slashes (`/`) enthalten.

Web-Besonderheiten (Grails & Cauldron)

In Web-Projekten auf Basis von Grails oder Cauldron existiert historisch bedingt kein automatisches Mapping von Entitäten oder Klassen auf Sprachpakete. Hier gelten folgende Besonderheiten:

- **Grails-Anwendungen:** Hier wird häufig mit einem einzigen globalen Sprachbündel namens `Grails` gearbeitet. Dieses kann zur besseren Strukturierung in logische

Unterbereiche aufgeteilt werden (z. B. `Grails.checkout` oder `Grails.user.settings`).

- **Cauldron-Projekte:** Hier herrscht prinzipiell freie Namenswahl für Sprachpakete. Für Neuentwicklungen wird jedoch dringend empfohlen, sich ebenfalls an das standardmäßige Paketnamensschema zu halten.

Vereinheitlichung im L10nCache und die L10nPack-Speicherarchitektur

Obwohl die Lokalisierungsdaten der Programmsprache aus zwei unterschiedlichen physischen Quellen stammen – statischen `.properties`-Dateien im Dateisystem und dynamischen Datenbank-Einträgen –, vereinheitlicht das System diese zur Laufzeit im Arbeitsspeicher. Beim Systemstart und beim Laden der Ressourcen liest das Framework die `.properties`-Dateien ein und überführt deren Inhalte intern in dieselbe logische Bündel-Struktur, die auch für Datenbank-Ressourcen (`L10nBundle`, `L10nResource` und `L10nEntry`) genutzt wird. Diese konsolidierten Daten werden gemeinsam im globalen `L10nCache` in Form von `L10nPack`-Objekten im RAM vorgehalten. Für den programmatischen Zugriff ist die physische Herkunft eines Textes somit vollkommen transparent. Dieser hybride In-Memory-Lookup garantiert, dass statische Standardübersetzungen aus dem CVS und dynamisch im laufenden Betrieb editierte Texte über exakt dieselbe API konsistent aufgelöst werden. Zudem ermöglicht dies, dass in der Datenbank hinterlegte Übersetzungen die statischen Standardwerte aus den `.properties`-Dateien im Arbeitsspeicher nahtlos und konfliktfrei überschreiben können.

Technische Implementierung und Speicheroptimierung (L10nPack)

Um den Speicherbedarf der Anwendung zu minimieren, weicht die Klasse `L10nPack` bewusst von Standard-Java-Klassen wie `java.util.HashMap` ab. Anstelle einer Map nutzt `L10nPack` intern zwei parallel laufende Arrays: ein Array für die Schlüssel und ein Array für die Werte. Dadurch werden die zusätzlichen Objekt-Pointer, die bei einer klassischen `HashMap` für jeden Node anfallen, vollständig vermieden. Dies spart angesichts der enormen Anzahl an String-Objekten im Lokalisierungssystem signifikant Arbeitsspeicher ein.

Um trotz des Verzichts auf eine Map-Struktur maximale Zugriffsgeschwindigkeiten zu garantieren, werden die Schlüssel im internen Array stets sortiert abgelegt. Die Suche nach einem Übersetzungseintrag erfolgt hochperformant über eine binäre Suche (Binary Search).

Zusätzlich ist die Klasse `L10nPack` für eine extrem schnelle Serialisierung und Deserialisierung direkt auf Datenebene optimiert. Diese effiziente Speicherabbildung auf Datenträgern ist die technische Voraussetzung dafür, dass L10n-Bündel im Cache über `java.lang.ref.WeakReference` gehalten werden können, ohne dass das wiederholte Einlesen bei GC-Bereinigungen zu nennenswerten Performance-Einbußen führt. Aus Gründen der Speicherverwaltung und GC-Effizienz ist auf Ebene einzelner `L10nPack`

-Instanzen explizit kein Vererbungs- oder Kaskadierungspfad (Parent Fallback) implementiert. Dies erlaubt es dem Garbage Collector (GC), nicht mehr stark referenzierte, einzelne `L10nPack`-Objekte rückstandslos freizugeben. Die Auflösung von Hierarchien und das bedarfsgerechte Nachladen fehlender Schlüssel werden stattdessen vollständig an übergeordnete Manager-Komponenten delegiert.

6.3.3. Metadaten-Modularisierung & Schema-Synchronisation

Das Lokalisierungssystem speichert und synchronisiert Übersetzungsdaten aus zwei physischen Quellen, die zur Laufzeit im L10nCache konsolidiert werden:

1. Statische `.properties`-Dateien im Dateisystem des Quellcodes, welche beim Build-Vorgang in die Anwendungs-JARs verpackt werden.
2. Dynamische `L10nBundle`-Objekte direkt aus der Datenbank, welche zur Laufzeit in den Server-Cache (`L10nCache`) geladen werden.

Metadaten-Aufteilung nach Modul-Herkunft

MyTISM trennt die Lokalisierung von Schema-Metadaten (wie Entitäts- und Attributnamen) strikt nach deren Herkunft (Core, Modul oder Projekt). Entitäten registrieren im System ihre genaue Herkunft sowie die Information, ob für sie eine benutzerdefinierte Klasse im Modul oder Projekt existiert.

Der Schema-Check ist in dieser Hinsicht streng ausgelegt: Soll eine benutzerdefinierte Klasse geladen werden, muss das Attribut `custom="true"` explizit in dem Modul oder Projekt definiert sein, in dem die Klasse tatsächlich deklariert ist. Eine bloße Definition „irgendwo“ im Projekt reicht nicht aus.

Deaktivierung des automatischen Startup-Scaffoldings

In älteren Versionen generierte der Server bei jedem Systemstart automatisch deutsche Standard-Übersetzungen direkt im BO-Bündel des Projekts. Dieses Verhalten wurde vollständig deaktiviert.

Durch diese Deaktivierung wird Folgendes sichergestellt:

- Der Server generiert beim Starten keine redundanten Einträge für Schema-Entitäten und -Attribute mehr im BO-Bündel des Projekts.
- Der interne L10n-Cache bleibt frei von unvollständigen Standardwerten, sodass die hierarchische Lookup-Kaskade mit Fallback auf die BO-Bündel der Module fehlerfrei funktioniert.
- Übersetzungen für in Modulen definierte Entitäten und Attribute verbleiben sauber in den jeweiligen Modul-Bündeln, anstatt das Projekt-Bündel zu überladen.

Manuelle Schema-Synchronisation und Redundanz-Vermeidung

Da die automatische Generierung beim Serverstart entfällt, müssen Schema-Änderungen (hinzugefügte oder entfernte Entitäten und Attribute) manuell synchronisiert werden. Hierzu steht in der GUI eine dedizierte Synchronisations-Aktion bereit, über welche die Übersetzungen eingepflegt werden können.

Um Redundanzen im BO-Bündel des Projekts zu minimieren, gilt folgende Regelung:

- Für das BO-Bündel des Projekts wird standardmäßig nur dann eine Ressource für einen Schlüssel angelegt, wenn dieser weder im Core noch in einem Modul existiert.
- Sollen Standard-Übersetzungen aus dem Core oder einem Modul im Projekt überschrieben werden, muss das entsprechende BO-Bündel für diese Entität und der zugehörige Ressourcenschlüssel manuell im Projekt angelegt werden.
- Es wird vorausgesetzt, dass die Core- und Modul-Übersetzungen den qualitativen Standard definieren und als primäre Quelle dienen.

Verzeichnisstruktur und Konventionen für Projekt-Dateien

Um Konflikte mit den automatischen Bereinigungsroutinen des Servers zu vermeiden, müssen Entwickler eine strikte Trennung bei manuell angelegten Übersetzungsdateien einhalten:

- **Bündel für Schema-Entitäten (BO-Übersetzungen):** Diese Daten werden in Dateien nach dem Schema `nrx/[…Projektverzeichnis…]/resources/l10n/[…Projekt-Package…].bo_[ISO-Kürzel]` (z. B. `.bo_de.properties`) abgelegt und beim Serverstart automatisch importiert. In diesen Dateien dürfen sich **ausschließlich** Schlüssel-Wert-Paare für tatsächlich im Schema existierende Entitäten und deren Attribute befinden. Sollten sich hier „überzählige“ oder freie Schlüssel befinden, verbleiben diese ungenutzt, weshalb freie Texte dennoch zwingend in separaten Dateien ohne das `.bo_`-Präfix im Namespace abgelegt werden sollten.
- **Freie Texte (Formulartitel, freie GUI-Beschriftungen):** Diese Texte müssen zwingend in separaten Dateien ohne das `.bo_`-Präfix im Namespace abgelegt werden, beispielsweise unter `nrx/[…Projektverzeichnis…]/resources/l10n/[…Projekt-Package…]_[ISO-Kürzel]` (z. B. `_de.properties`). Diese Daten sind vom automatischen Schema-Bereinigungsprozess ausgenommen und bleiben dauerhaft erhalten.

Die Initialisierung und Steuerung dieses Import-Verhaltens erfolgt in der Methode `L10nBundle.initEnvironment()`.

6.3.4. XML-Konfiguration & UI-Makros

In den XML-Konfigurationsdaten von Lesezeichen, Formularen oder Schablonen werden statische Bildschirmtexte mit dem Makro `$R{Schluessel}` lokalisiert.

```
<Table entity="Ereignis" columns="Bot | Anfang '$R{Anfang}', desc
| Ende '$R{Ende}', desc | Patient '$R{_Patient}'"/>
```

Ist der L10n-Schlüssel exakt identisch mit dem Attributnamen der Entität, generiert das System das Übersetzungslabel vollautomatisch, sodass das `$R{…}`-Makro im XML entfallen kann. Lesezeichen-Filter unterstützen die Übersetzung ihrer Titel über dasselbe Makro, wie in `<filter type="bool" title="$R{Erledigt}"/>`.

Sollte für einen Aufruf keine gültige Übersetzung gefunden werden, zeigt das System stattdessen den rohen L10n-Schlüsselnamen in der Benutzeroberfläche an. Schlüssel müssen daher zwingend als lesbare, „sprechende Namen“ definiert werden, damit fehlende Übersetzungen eindeutig zugeordnet werden können.

Nicht maskierte Anführungszeichen oder Apostrophe in Übersetzungsdateien können das Rendern von XML-Strukturen zerstören, da die UI-Engine diese fälschlicherweise als XML-Steuerzeichen interpretiert. Einfache Hochkommas müssen in Bündeln zwingend durch eine mehrfache Schreibweise escaped werden (z. B. `Patient''''s file`).

6.3.5. Verwaltung von Übersetzungs-Bündeln

Die Verwaltung aller UI-Übersetzungstexte erfolgt im Navigationsbaum unter „Admins → MyTISM → Lokalisierung → Lokalisierungs-Bündel“. Bündel sollten für eine saubere Bereichstrennung dediziert nach Fachbereichen angelegt werden (z. B. `de.example.bo.Geschlecht`).

Legt ein Administrator ein völlig neues Bündel manuell über die GUI an, muss zwingend das Häkchen bei „Vorgeladen“ (Preload) gesetzt und die Pfadposition auf 0 konfiguriert werden.

Wird ein Lokalisierungs-Bündel in der GUI erstmalig angelegt und auf „Vorgeladen“ gesetzt, werden die Einträge vom Client oft nicht sofort gefunden. In diesem speziellen Fall der Neuanlage ist zwingend ein Neustart des Servers erforderlich, um den Cache sauber zu initialisieren.

Für groß angelegte Massenübersetzungen durch externe Dienstleister lassen sich diese dynamischen Sprachbündel als Textdatei im `.properties`-Format exportieren. Die Export-Logik nutzt die Option „Locale 2“ für die Zielsprache, während die Option „Locale 1“ in der generierten Datei lediglich als auskommentierte Referenzsprache für den Übersetzer dient. Dabei sind die Standardwerte für den Export von Bündeln (wie Locales und Pfade) intelligent voreingestellt. Die physischen L10n-Bündeldateien verzichten zudem beim Export auf das Wrapping von Textwerten.

```
#de.example.bo.Gefahrenstufe|fr|UTF-8
# :noTabs=true:mode=makefile:
# Please do not change the above lines in ANY WAY!
# Leicht = slight
Leicht =
```

Die GUI-Maske für `L10nBundle` wurde zudem erweitert, um das Auffinden von Duplikaten in Core, Modulen oder im Projekt zu erleichtern und somit die langfristige Wartbarkeit zu stärken.

Bei Quertabellen berechnen die virtuellen Attribute `L10nName` und `L10nBeschreibung` ihre Übersetzungen vollautomatisch basierend auf den regulären Feldern für Name und Beschreibung über `msg(getName(), [Object this])`.

Bei dynamischen Bündeln in der Datenbank dürfen niemals Unterstriche in Bündelnamen verwendet werden. Das native Java-Resource-Backend interpretiert jede Zeichenfolge nach einem Unterstrich fälschlicherweise als Sprachkürzel, wodurch das Bündel im Cache unauffindbar wird.

Die Zeichenfolge `.bo.` im Bündelnamen führt zu kritischen Ladefehlern, falls die entsprechende Entität im Projekt nicht existiert. Als Workaround sollten in solchen Fällen Bündelnamen ohne die Zeichenfolge `.bo.` genutzt werden.

L10n-Schlüsselnamen dürfen systemweit ausschließlich Buchstaben, Zahlen sowie die Zeichen `_`, `-`, `.`, `~` und `/` enthalten. Für die Bezeichner kompletter Sprachpakete (`L10nPack`) sind die Sonderzeichen Tilde (`~`) und Slash (`/`) strikt untersagt.

6.3.6. Programmatische UI-Lokalisierung (Java, NetRexx & Groovy)

Auf Backend-Ebene stellt die Klasse `de.ipcon.tools.L10n` die primäre API für alle programmatischen Lokalisierungsprozesse der Programmsprache bereit.

```
@ENTITY Arzt uses L10n@
```

Durch diesen Import steht die Methode `msg()` nativ für lokalisierte Exception-Messages zur Verfügung.

```
if eighteenYearsBefore.before(dob) then
   signal SaveVetoException(msg('exc.PatientNotOfAge'))
```

Für die Übergabe einzelner Textparameter existieren bequeme Kurzschreibweisen.

```
msg('frage.Entlassen', 'Dr. Schmidt')
```

Übersetzungsstrings mit hochdynamischen Parameterstrukturen werden systemintern via `java.text.MessageFormat` aufgelöst.

```
method getBehandlungsZusammenfassung() returns String
   return msg('msg.BehandlungsZusammenfassung', [-
      Object Integer.valueOf(getSymptome().values().size()),-
      getBehandlungskosten()-
   ]);
```

Die dazugehörige `.properties`-Datei nutzt die `choice`-Logik von Java für Fallunterscheidungen und Pluralbildung.

```
msg.BehandlungsZusammenfassung=Symptome ({0, choice,
0#keine|0<{0} erfasst}) {1, choice, 0#|0<Kosten: {1}}
```

In Groovy muss bei der Übergabe dynamischer Parameter zwingend ein expliziter Cast auf ein Array vom Typ `Object[]` erfolgen. Ohne diesen Cast wählt die dynamische Typisierung die falsche Methodensignatur aus und wirft zur Laufzeit einen Fehler.

Soll ein UI-Text völlig unabhängig von der aktuell eingestellten Oberflächensprache des angemeldeten Benutzers in einer fest definierten Sprache gerendert werden (z. B. für automatisierte Export-Dokumente), muss das gewünschte `Locale` explizit an die API übergeben werden.

```
String text = L10n.msg("befund.Titel", null, null, true,
Locale.GERMAN)
```

L10n und das Anführungszeichen bzw. Apostroph

Die unbedachte Verwendung von Standard-Anführungszeichen oder Apostrophen in Übersetzungen (sowohl in `.properties`-Dateien als auch in Datenbank-Bündeln) führt regelmäßig zu kritischen UI-Ladefehlern. Werden diese übersetzten Werte in dynamisch generierte XML-Texte injiziert (besonders häufig bei Tabellenspalten-Definitionen oder Attribut-Labels), interpretieren nachgelagerte Parser diese fälschlicherweise als XML-Steuerzeichen. Dies beendet XML-Attribute vorzeitig und führt zu unvollständig gerenderten Oberflächen.

Zur Vermeidung dieser Fehler gelten folgende Entwickler-Richtlinien:

- **Nutzung typografischer Sonderzeichen:** Anstelle der einfachen, schreibmaschinenbasierten Zeichen `'` und `"` sollten konsequent die typografisch korrekten Zeichen `’` (erreichbar über `AltGr` + `#` bzw. `AltGr` + `'`), sowie `„` (`AltGr` + `V`) und `“` (`AltGr` + `B`) verwendet werden. Diese Zeichen besitzen keine steuernde Wirkung in XML-Dokumenten und verhindern Darstellungsfehler zuverlässig.
- **Maskierung von einfachen Hochkommas:** Müssen einfache Hochkommas für Java-Klassen (z. B. `MessageFormat`) oder System-Bündel zwingend als Standard-Apostroph geschrieben werden, sind diese in den `.properties`-Dateien und Datenbank-Einträgen durch Verdopplung zu maskieren (z. B. `Patient''s file` statt `Patient’s file`).

6.3.7. Unit-Tests & Validierung von Systemtexten

In automatisierten Unit-Tests dürfen Systemmeldungen der Programmsprache niemals über harte Strings verglichen werden. Der erwartete Text muss stets dynamisch über das L10n-System aufgelöst werden, da der Test andernfalls auf Maschinen mit abweichendem Standard-Locale fehlschlägt.

```
errMsg = L10n.msg(Patient.L10N_KEY_MISMATCHED_PARAMS, [paramA,
paramB] as Object[], [Patient] as Object[]);
```

Es muss zwingend die Variante der `msg()`-Methode mit drei Parametern genutzt werden, um durch die Übergabe des Entitäts-Objekts den korrekten Suchpfad zu gewährleisten.

Chapter 7. Alarmsystem & Benachrichtigungen

7.1. Grundlagen & Konzepte

MyTISM trennt die Überwachung von Daten und das Versenden von E-Mails konzeptionell in zwei völlig eigenständige Systemkomponenten. Das Alarmsystem fungiert als Sensor, der beispielsweise kontinuierlich auf das Eintreffen neuer, kritischer Laborergebnisse wartet. Das Benachrichtigungssystem stellt die Kommunikations-Infrastruktur dar, vergleichbar mit einem Pager-Netzwerk, das Nachrichten zustellt.

Diese strikte Entkopplung ermöglicht den autarken Betrieb beider Komponenten. Nachrichten können manuell versendet werden, ohne dass ein Alarm im Hintergrund läuft. Umgekehrt kann das Alarmsystem unsichtbare Hintergrundskripte ausführen, ohne eine Benachrichtigung zu versenden.

7.1.1. Verhalten bei Teilausfällen (Queuing-Mechanismus)

Die entkoppelte Architektur definiert klares Systemverhalten bei Teilausfällen:

Nur Alarmsystem aktiv:
Ist das Alarmsystem aktiv, das Benachrichtigungssystem jedoch deaktiviert, werden Alarme weiterhin überwacht. Auslösende Alarme generieren `MyTISMBenachrichtigungsAuftrag`-Objekte, die unbearbeitet in einer Warteschlange in der Datenbank verbleiben. Wird das Benachrichtigungssystem reaktiviert, arbeitet es diese Warteschlange ab und stellt die Nachrichten nachträglich zu.

Nur Benachrichtigungssystem aktiv:
Ist das Benachrichtigungssystem aktiv, das Alarmsystem jedoch deaktiviert, werden keine automatischen, alarmbasierten Aufträge generiert. Benachrichtigungsaufträge, die auf anderem Wege im System eingehen (z. B. manuell oder per Skript), werden jedoch sofort verarbeitet. Bei einer späteren Reaktivierung des Alarmsystems werden verpasste Auslösungen – abhängig von der individuellen Konfiguration des Alarms – nachträglich nachgeholt.

7.1.2. Die vier Alarmtypen (Das „Was“ und „Warum“)

MyTISM stellt vier spezifische Alarmtypen bereit.

Der Einfache Termin

Dieser Alarm löst unabhängig von konkreten Datenobjekten zu einem fest definierten

Zeitpunkt aus. Dies kann einmalig oder wiederkehrend nach einem Cron-Muster geschehen (z. B. eine kalendarische Erinnerung an eine wöchentliche Chefvisite).

Der BO-basierte Termin (BBT)

Dieser Alarm überwacht eine dynamische Menge konkreter Geschäftsobjekte (BOs) und berechnet für jedes Objekt einen individuellen Auslösezeitpunkt. Ein Anwendungsfall ist die dynamische Generierung jährlicher Erinnerungen an Routineuntersuchungen für alle Patientenakten.

Der Hinweis

Dieser reaktive Alarm feuert sofort bei Eintritt eines definierten Ereignisses (Erstellen, Ändern oder Löschen eines Objekts). Fällt beispielsweise ein Medikamentenbestand im Lager unter einen definierten Schwellenwert, löst der Hinweis unmittelbar aus.

Die Wiedervorlage (WV)

Die Wiedervorlage reagiert auf das Ausbleiben von Änderungen oder Ereignissen innerhalb einer definierten Frist (Inaktivität). Wird einem neu aufgenommenen Patienten in der Notaufnahme nicht innerhalb von 30 Minuten ein Arzt zugewiesen, löst das System den Alarm aus.

7.1.3. Funktionsweise der Objekt-Überwachung

Um die zu überwachende Objektmenge gezielt einzuschränken, nutzt das System sogenannte BOMasken. Eine BOMaske filtert die Datenbasis vor, sodass ein Alarm beispielsweise ausschließlich für Patienten auf der Intensivstation greift.

Bei allen Terminen lässt sich eine Vorwarnzeit konfigurieren, um Benachrichtigungen mit definiertem Vorlauf zum eigentlichen Ereignis zu versenden. Bei Hinweisen und Wiedervorlagen definieren Auslösekriterien präzise, auf welche Attribut-Änderungen das System reagiert (z. B. „wird gesetzt auf Wert kleiner als“).

7.1.4. Meldewege und Empfänger (Das „Wie“ und „Wer“)

Die primären Meldewege von MyTISM sind klassische E-Mails sowie Direkt-Benachrichtigungen im grafischen Solstice-Client. Zusätzlich unterstützt das System die Anbindung echter Hardware-Smartphones als SMS-Gateways.

Empfänger können Benutzer, Benutzergruppen oder dynamische Adressen wie die „Adresse des Patienten“ sein. Das System garantiert technisch eine strikte Deduplizierung: Ein Empfänger erhält pro Alarmauslösung exakt eine einzige Benachrichtigung, selbst wenn er über mehrere Wege adressiert wurde.

7.2. Bedienung & Konfiguration

Dieser Abschnitt beschreibt die Konfiguration für Power-User, von Frontend-Einstellungen bis hin zu Alarmformularen in der GUI.

7.2.1. Empfang von Nachrichten im Client

Für den Empfang von Alarmmeldungen im MyTISM-Client muss im XML-Nutzerprofil des Anwenders zwingend ein entsprechendes Plugin konfiguriert sein. Mit dem Parameter `silent="yes"` sammelt das Plugin neue Meldungen unaufdringlich in der Taskleiste. Die Konfiguration `silent="no"` öffnet hochpriorisierte Popups sofort im Vordergrund auf dem Bildschirm.

```
<Plugin
class="de.ipcon.form.notification.ClientNotificationManager"
silent="yes"/>
```

7.2.2. Benutzereinstellungen und Adressen

In der Benutzeroberfläche des Benutzerformulars lassen sich Adressen und kryptografische Präferenzen definieren:

Einfache Konfiguration:
E-Mail-Adressen, die im Reiter „Einfache Benachrichtigungskonfiguration“ eingetragen werden, generieren im Hintergrund automatisch `MyTISMAdresseEmail`-Objekte.

PGP-Konfiguration:

Im Reiter „Benachrichtigungssystem“ → „Adressen“ kann der öffentliche Schlüssel für verschlüsselte Nachrichten hinterlegt werden. Der Import erfolgt aus der lokalen Datei `/.gnupg/pubring.gpg` oder über einen Keyserver (Variable `pgpKeyServer`).

Kryptografische Präferenzen:
Es kann festgelegt werden, ob verschlüsselte/signierte Mails gewünscht sind („Nie“, „Wenn möglich“, „Zwingend“, „Standard“) oder ob das ältere Inline-Format erzwungen werden soll.

Adressen-Priorisierung und Fallback:
Empfänger-Ziele (`MyTISMAdresseSolstice` oder `MyTISMAdresseEmail`) werden über das Attribut `Position` priorisiert. Der Versand erfolgt der Reihe nach, bis die erste Zustellung erfolgreich ist; nachfolgende Adressen dienen als Fallback.

Das Flag „WeiterAuchWennErfolgreich“, welches parallelen Versand erzwingen soll, führt aufgrund eines bekannten Bugs zu unzuverlässigem

> Routing. Es wird dringend empfohlen, pro Benutzer nur eine einzige E-Mail-Adresse zu konfigurieren.

7.2.3. Alarme verwalten, aktivieren und testen

Die Verwaltung der Alarme erfolgt unter „Admins → MyTISM → Alarme“. Mitglieder der Systemgruppe „Admins Alarmsystem“ besitzen vollen Zugriff.

Aktivierung:
Neue Alarme werden im Status „vorbereitet“ (deaktiviert) angelegt und erst durch das Setzen der Checkbox „Alarm ist aktiv“ scharfgeschaltet.

Testmodus:
Ist das Flag „Testmodus“ aktiv, werden keine Benachrichtigungen versendet und keine `AlarmAusloesung`-Objekte angelegt. Das System schreibt lediglich Log-Einträge zur Simulation.

> Systemkritische Hintergrund-Aktionen werden auch im Testmodus real ausgeführt. Einfache Termine werden nach simulierter Auslösung gelöscht. Die für BO-basierte Termine und Wiedervorlagen benötigten Status-Objekte (`BOBasierterTerminStatus` / `WiedervorlageStatus`) werden in der Datenbank fortgeschrieben.

7.2.4. Gemeinsame Eigenschaften aller Alarme

Alle Alarme verfügen über folgende grundlegende Felder:

Name / Beschreibung:
Kurzname und Hilfetext, welcher dynamisch in Benachrichtigungen verwendet werden kann.

Empfänger (CC / BCC):
Angabe von CC- und BCC-Empfängern unter Berücksichtigung der systemweiten Deduplizierung.

Benachrichtigungsvorlage:
Referenz auf die GSP-Vorlage zur Definition von Layout und Inline-Bildern.

Alte Alarme nur auslösen wenn nicht älter als:
Definiert im Reiter „Erweitert“, wie weit verpasste Auslösungen (z. B. nach einem Ausfall) in der Vergangenheit liegen dürfen, um nachgeholt zu werden.

Verantwortlicher:
Benutzer, der bei System- oder Skriptfehlern des Alarms benachrichtigt wird. Seine E-Mail-

Adresse dient bei ausgehenden Mails als Absender. Ohne expliziten Verantwortlichen wird der interne System-User des Alarmsystems als Absender genutzt.

Bei Ausfall benachrichtigen:
Optionale Gruppe, die bei Skript- oder Systemfehlern zusätzlich informiert wird.

Überwachung starten ab:
Datum in der Zukunft, ab dem die Überwachung scharfgeschaltet wird.

7.2.5. Der „Einfache Termin“

Der „Einfache Termin“ löst zu einem festen Zeitpunkt unabhängig von Datenobjekten aus. Er kann einmalig („… am/um“) oder wiederkehrend („… wiederholen nach Muster“) per Cron-Syntax konfiguriert werden. Das System unterstützt klassische Cron-Muster (z. B. `*/5` für alle fünf Minuten) sowie proprietäre Makros (z. B. `@weekly` für Sonntag 0:00 Uhr oder `4/@lastOfM` für den letzten Mittwoch im Monat).

Vorwarnzeit:
Die Benachrichtigung kann gezielt vor dem Ereignis ausgelöst werden (z. B. `15m`).

System-Verhalten: Einmalige Termine werden nach ihrer Auslösung gelöscht.

7.2.6. Der „BO-basierte Termin“ (BBT)

Ein BO-basierter Termin überwacht eine Menge von Geschäftsobjekten und berechnet individuelle Auslösezeitpunkte. Die Objektmenge wird über eine BOMaske definiert. Der Auslösezeitpunkt wird über zwei Wege bestimmt:

1. **Auslese-Attribut:** Ein Datums-Attribut des Objekts wird ausgelesen.
2. **Auslöse-Skript:** Ein Groovy-Skript berechnet dynamisch ein `java.util.Date`.

Das Auslösedatum wird nur bei der Alarmerstellung, beim Start der Überwachung des Objekts sowie bei expliziten Änderungen am Datums-Attribut oder am Skript neu berechnet. Wird ein virtuelles, zeitabhängiges Attribut verwendet, bemerkt das Alarmsystem Änderungen im Hintergrund nicht und arbeitet mit dem veralteten, gecachten Datum.

Neuterminierung nach Auslösung:
Standardmäßig löst ein BBT pro Objekt nur einmal aus. Ist das Flag „Alarm bleibt auch nach Auslösung weiterhin aktiv“ gesetzt, wird nach der Auslösung ein neuer Zeitpunkt berechnet. Das Skript muss hierbei zwingend ein Datum in der Zukunft zurückliefern, um Endlosschleifen zu verhindern.

7.2.7. Der „Hinweis“

Ein Hinweis reagiert auf aktive Ereignisse in der Datenbank ohne Echtzeit-Garantie (Latenz im Sekundenbereich). Für blockierende, sofortige Reaktionen muss die Methode `verifyOnServer()` der jeweiligen Objekt-Klasse genutzt werden.

Einfache Hinweis-Ereignisse:

- **... erzeugt wurde:** Löst aus, wenn ein neues, passendes Objekt erstellt wird.
- **... geändert wurde:** Löst bei beliebigen Feld- oder Relationsänderungen am Objekt aus.
- **... gelöscht wurde:** Löst beim Löschen aus.
- **... erschienen ist:** Löst aus, wenn ein existierendes Objekt so modifiziert wird, dass es neu in die Maske passt.
- **... verschwunden ist:** Löst aus, wenn ein Objekt so geändert wird, dass es die Maskenkriterien nicht mehr erfüllt.

Hinweis-Ereignisse über Auslösekriterien:

Auslösekriterien definieren präzise Änderungen über drei Eigenschaften:

- **Attribut:** Das zu überwachende, persistente Datenfeld (virtuelle Attribute werden nicht unterstützt).
- **Änderungstyp:** Z. B. „wird gesetzt auf Wert kleiner als“ oder „wird in irgendeiner Weise geändert“.
- **Wert:** Statischer Vergleichswert für den Änderungstyp.

Mehrere Kriterien sind standardmäßig mit OR verknüpft, können aber über die Checkbox „Alle Kriterien müssen zutreffen“ mit AND verknüpft werden. Zusätzlich kann gefiltert werden, welcher Benutzer oder ob ein Mitglied einer Gruppe die Änderung durchgeführt hat.

7.2.8. Die „Wiedervorlage“ (WV)

Wiedervorlagen reagieren auf Inaktivität und prüfen, ob definierte Kriterien innerhalb einer Wartezeit **nicht** eingetreten sind.

Inaktivitätszeit:
Wartezeit bis zur Auslösung (z. B. 2d).

Neuterminierung nach Kriterienerfüllung:
Ist das Flag „Alarm bleibt auch nach Kriterienerfüllung weiterhin aktiv“ gesetzt, läuft die Überwachung nach einem eingetroffenen Ereignis weiter.

Neuterminierung nach Auslösung:
Ist das Flag „Alarm bleibt auch nach Auslösung weiterhin aktiv“ gesetzt, feuert die Wiedervorlage nach Ablauf der Inaktivitätszeit in wiederkehrenden Intervallen neu.

7.2.9. Manuelles Versenden

Manuelle Benachrichtigungen werden über die Schablone „MyTISM-Benachrichtigungsauftrag ohne Vorlage (Vorgebaut; Versenden)“ an Benutzer, Gruppen oder andere `NotificationReceiverCollectionI` versendet. Die Schablone enthält Felder für Empfänger, Betreff, Text, Zeichensatz, Priorität und Anhänge.

7.2.10. L10n-Spracheinstellungen (Benutzer)

Platzhalter (`$R{…}`) in Text und Betreff werden durch das L10n-System übersetzt. Das Benachrichtigungssystem ermittelt die Sprache (Locale) in folgender Reihenfolge:

1. Die Locale am Benachrichtigungsauftrag selbst.
2. Eine über `getPreferredLocale()` definierte Locale des Empfängers.
3. Die Standard-Locale des Systems.

Das Attribut `BevorzugtesLocale` einer `MyTISMAdresse` wird vom System ausschließlich ausgewertet, wenn diese Adresse direkt als Empfänger eingetragen wurde. Bei indirekter Adressierung (z. B. über den Benutzer) wird das Attribut ignoriert.

7.3. System-Verwaltung

Dieser Abschnitt behandelt administrative Aufgaben wie Rechteverwaltung und fortgeschrittenes Tracing.

7.3.1. Solstice-Integration und Berechtigungen

Damit Popups im Solstice-Client angezeigt werden, muss das Plugin im XML-Profil des Benutzers eingetragen sein:

```
<Plugin
class="de.ipcon.form.notification.ClientNotificationManager"
silent="yes"/>
```

Damit Benachrichtigungen geladen werden, benötigen Benutzergruppen Leserechte auf die Schema-Klassen der Datei `core-benachrichtigung.xml` (u. a.

`MyTISMBenachrichtigung`, `MyTISMBenachrichtigungsAuftrag`, `MyTISMBenachrichtigungsVorlage`). Um eigene Nachrichten zu verfassen, sind zusätzlich Erstellen- und Schreibrechte erforderlich.

7.3.2. Globaler Catchall-Empfänger

Über die Einstellungen-Variable `notifications.catchall` kann ein globaler Standardempfänger hinterlegt werden. Dieser wird genutzt, wenn der eigentliche Empfänger gelöscht wurde, „Anmeldung verweigern“ aktiv ist oder keine konkreten Adressen hinterlegt sind.

Dieser funktionale Catchall-Empfänger ist strikt von der netzwerkseitigen Catchall-Routing-Regel für Mailserver zu trennen.

7.3.3. Dienst für fehlgeschlagene Aufträge

Der Hintergrunddienst `NotifyAdminsOfFailedBenachrichtigungsAuftragsService` informiert Administratoren über endgültig fehlgeschlagene Benachrichtigungsaufträge. Der Verteiler wird über die Variable `notification.recipients.mytismBenachAuftragFailed` als komma-separierte Liste konfiguriert. Die Vorlage liegt unter `/nrx/de/ipcon/resources/services/bs/NotifyAdminsOfFailedBenachrichtigungsAuftrags.bs.xml`.

7.3.4. Tracing: Benachrichtigungen für spezifische Objekte finden

Administratoren können über OQL-Ausdrücke in der Suchleiste von Lesezeichen prüfen, ob Benachrichtigungen für spezifische Objekte erzeugt wurden.

Generelle Benachrichtigungsaufträge:

- Lesezeichen: `/Admins/MyTISM/Benachrichtigungen/MyTISM-Benachrichtigungsaufträge (Vorgebaut)`
- Suchbegriff: `[exists(within KontextBOs b where b.BO.Id = <objekt-id>) or exists(within AnhangBOs c where c.BO.Id = <objekt-id>)]`

Alarmspezifische Aufträge:

- Lesezeichen: `/Admins/MyTISM/Alarme/Ausloesung/AlarmBenachrichtigungsauftraege`
- Suchbegriff:

`[a.(AlarmAusloesungFuerBO)FuerAusloesung.BO.Id = <objekt-id> or exists(within AnhangBOs c where c.BO.Id = <objekt-id>)]`

7.4. System-Administration & Troubleshooting

Dieser Abschnitt richtet sich an System-Administratoren und beschreibt die Konfiguration über die `mytism.ini`, Netzwerk-Infrastruktur sowie das Troubleshooting des SMS-Gateways.

7.4.1. Aktivierung des Alarmsystems und des Benachrichtigungssystems (mytism.ini)

Beide Systeme werden über die zentrale Konfigurationsdatei `mytism.ini` gesteuert.

```
[Alarme]
activateAlarme=if_possible

[Notifications]
activateNotifications=if_possible
```

- **Alarmsystem:** Erlaubte Werte sind `if_possible` und `mandatory`. In Cluster-Umgebungen darf das Alarmsystem aus technischen Gründen ausschließlich auf dem autoritativen Hauptserver aktiviert sein.
- **Sync-Events:** Der historische Parameter `handleSyncEvents` ist veraltet. Er wird vom System ignoriert und kann aus der `mytism.ini` entfernt werden.
- **Benachrichtigungssystem:** Der Wert `activateNotifications` wird permanent überwacht. Änderungen zur Laufzeit starten oder stoppen das System automatisch. In Cluster-Umgebungen muss das Benachrichtigungssystem auf allen Knoten aktiv sein.

7.4.2. Deaktivierung des Benachrichtigungssystems

Um das Versenden aller Benachrichtigungen (insbesondere E-Mails) systemweit zu stoppen, muss das Benachrichtigungssystem in der `mytism.ini` deaktiviert werden:

```
[Notifications]
activateNotifications=never
```

In Cluster-Umgebungen muss dieser Schalter auf allen sendefähigen Cluster-Knoten auf `never` gesetzt werden. Bereits an den konfigurierten

> Mailserver (MTA) übertragene E-Mails werden von diesem dennoch zugestellt.

7.4.3. Spam-Vermeidung (Rate Limiting)

Die Drosselung wird pro Übertragungskanal konfiguriert:

```
[Notifications.Email]
sendingRateLimitCheckDurationInSeconds=10
sendingRateLimitMaxSendingCount=10

[Notifications.Solstice]
sendingRateLimitCheckDurationInSeconds=60
sendingRateLimitMaxSendingCount=40
```

Fehlen diese Parameter, greifen die Standardwerte (`sendingRateLimitCheckDurationInSecondsDefault=60`, `sendingRateLimitMaxSendingCountDefault=120`), was maximal 120 Benachrichtigungen innerhalb von 2 Minuten erlaubt.

Um das Rate Limiting vollständig abzuschalten, muss mindestens ein Parameter auf den Wert `-1` gesetzt werden:

```
[Notifications.Email]
sendingRateLimitCheckDurationInSeconds=-1
sendingRateLimitMaxSendingCount=-1

[Notifications.Solstice]
sendingRateLimitCheckDurationInSeconds=-1
sendingRateLimitMaxSendingCount=-1
```

> Diese Anpassungen müssen zwingend auf allen sendefähigen Cluster-Knoten vorgenommen werden. Bereits gedrosselte Nachrichten können aufgrund des Wiederholungsmechanismus noch bis zu 30 Minuten verzögert zugestellt werden.

7.4.4. Schutz vor veralteten Nachrichten (Maximales Alter)

```
[Notifications]
```

```
maxAgeOfNoticationInDays=3
```

Ist der Parameter gesetzt, führt das System vor jedem Versand eine millisekundengenaue Prüfung durch. Ist eine Nachricht zum Sendezeitpunkt mindestens eine Millisekunde älter als die definierte Anzahl an Tagen, wird sie verworfen.

7.4.5. Verschlüsselung und digitale Signatur (OpenPGP)

Auf Serverumgebungen mit Java 8 oder älter müssen für die Ausführung von OpenPGP die „Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files“ installiert sein.

```
[Notifications]
encrypt=if_possible
sign=if_possible
privateKeyFile=/.<project>/.gnupg/secring.gpg
```

- `encrypt`: Erlaubte Werte sind `never`, `if_possible` (Standard) und `mandatory` (bricht bei fehlendem Empfänger-Schlüssel mit Fehler ab).
- `sign`: Erlaubte Werte sind `never`, `if_possible` (Standard) und `mandatory` (bricht bei fehlendem System-Schlüssel ab, Benachrichtigungssystem wird deaktiviert).

Erzeugung des System-Schlüssels via GnuPG (Linux)

Bei allen `gpg`-Aufrufen muss zwingend der Parameter `--homedir` angegeben werden, um eine Vermischung mit dem Schlüsselring des Linux-Benutzers zu vermeiden.

1. Erstellen Sie ein temporäres Verzeichnis: `mkdir /tmp/gnupg`
2. Generieren Sie den Schlüssel: `gpg --homedir /tmp/gnupg --gen-key`
3. Gültigkeit: `0` (verfällt nie).
4. Name und E-Mail eingeben (sollte mit der `from`-Adresse übereinstimmen).
5. **Wichtig:** Vergeben Sie **keine** Passphrase, da MyTISM diese beim automatisierten Zugriff nicht übergeben kann. Der Schutz muss rein über restriktive Dateirechte erfolgen.
6. Kopieren Sie die generierte Datei `/tmp/gnupg/secring.gpg` an den Pfad aus der `mytism.ini` (`privateKeyFile`).
7. Exportieren Sie den öffentlichen Schlüssel für die Empfänger:

`gpg --homedir /tmp/gnupg --export --armor <schlüssel-id>`

Weitere E-Mail-Adresse hinzufügen

1. Starten Sie den Editierungsmodus: `gpg --homedir /tmp/gnupg --edit-key <schlüssel-id>`
2. Geben Sie am Prompt `adduid` ein, tragen Sie die Daten ein und speichern Sie mit `save`.

7.4.6. Mailer-Konfiguration (E-Mail-Einstellungen)

```
[Mailer]
smtpHost=mail.example.com:587
useTLS=1
from=MyTISM-System <mytism@example.com>
authMethod=smtp_auth
username=smtpauthname
password=smtpauthpasswort
useInlinePGP=0
suppressMsgID=0
checkAddress=support@oashi.com
```

- `authMethod`: Zulässige Werte sind `none`, `pop_before_smtp` (erfordert `username`, `password`, optional `POPBeforeSMTPHost`) und `smtp_auth`. Numerische Werte (0, 1, 2) sind veraltet und unzulässig.
- `useInlinePGP`: `0` (Standard) verwendet MIME-Format, `1` erzwingt das veraltete Inline-PGP-Format.
- `suppressMsgID`: Administratoren können über `1` erzwingen, dass MyTISM keinen eigenen „Message-ID“-Header generiert, welcher andernfalls Servernamen und Datum verraten würde (Standard ist `0`).
- `checkAddress`: Sendet beim Serverstart automatisch eine Test-E-Mail zur Überprüfung der SMTP-Anbindung.

Konfiguration mehrerer Mailserver (E-Mail-Routing-Regeln)

Sollen E-Mails abhängig von Kriterien über unterschiedliche Mailserver verschickt werden, deklarieren Sie mehrere Mailer-Sektionen (`[Mailer.Name]`):

```
[Mailer.Alt1]
from=mytism@example.com
```

```
smtpHost=mail.example.com

[Mailer.Alt2]
from=mytism@someOtherServer.org
smtpHost=smtp.someOtherServer.org
useTLS=1
```

Die Zuordnung erfolgt über E-Mail-Routing-Regeln in der GUI. Diese filtern ausgehende Nachrichten anhand von Absender, Empfänger oder Betreff mittels regulärer Ausdrücke. Damit eine Regel greift, müssen alle definierten Filter zutreffen (Ausnahme: Bei mehreren Empfängern muss der Filter auf mindestens einen zutreffen).

Jedes System verfügt über eine implizite „Catchall"-Regel, die dem Hauptserver zugewiesen ist und die Standardkonfiguration `[Mailer]` anspricht. Fehlt eine referenzierte Konfiguration in der `mytism.ini`, wird der Versand blockiert.

In Cluster-Umgebungen arbeitet jeder Knoten die Regeln nach Position ab. Trifft eine Regel zu und ist dem Knoten zugewiesen, versendet dieser die E-Mail; andernfalls wird die Bearbeitung für diesen Knoten abgebrochen.

7.4.7. Kontrolle und Fehlersuche bei Benachrichtigungen

Treten beim Versenden Fehler auf, erhalten Aufträge den Statuscode 3 (`WAITING_FOR_RETRY`) und das System versucht in 30-Minuten-Intervallen einen erneuten Versand. War die Versendung erfolgreich, wird der Status auf 4 (`SUCCESSFULLY_SENT`) gesetzt. Fehlgeschlagene Solstice-Popups bei abgemeldeten Benutzern lösen keinen Retry aus.

Administratoren prüfen fehlgeschlagene Aufträge im Lesezeichen „Admins/MyTISM (Vorgebaut)/Benachrichtigungen/MyTISM-Benachrichtigungsaufträge (Vorgebaut)" über den Filter „Mit 'echten' Fehlern".

7.4.8. Troubleshooting: Android SMS Gateway

Das SMS-Gateway basiert auf der Android-App „Android SMS Gateway" und wird über ein physisches Smartphone betrieben.

1. Allgemeine System- und Netzwerkprüfung

- Prüfen Sie das `daily.log` auf Initialisierungsfehler von `SMSNotificationHandler` oder `AndroidSMSGatewayClient`.
- Für detailliertes Logging tragen Sie Folgendes in die `log4j.conf` ein:

```
log4j.logger.de.ipcon.db.notification.SMSNotificationHandler=DEBUG
log4j.logger.de.ipcon.messaging.sms.smsgateway=DEBUG
```

- Vergewissern Sie sich, dass `[Notifications.SMS]` und `[Notifications.SMS.AndroidSMSGateway]` in der `mytism.ini` aktiv sind.
- Stellen Sie sicher, dass das Smartphone eine WLAN-Verbindung besitzt und der OpenVPN-Tunnel („oashi-infra“) aktiv ist.
- Prüfen Sie, ob die ASG-App auf dem Handy geöffnet, „Local server“ aktiv und die App „online“ ist (die IP muss mit OpenVPN übereinstimmen).
- Testen Sie die Erreichbarkeit der App vom Server aus: `curl -X GET -u <username> http://<gatewayHost>:8080/health`
- **Voodoo-Reset:** Kommentieren Sie die Sektion `[Notifications.SMS]` in der `mytism.ini` aus, speichern Sie, entfernen Sie die Kommentarzeichen und speichern Sie erneut, um das Subsystem neu zu starten.

2. Fehleranalyse beim SMS-Versand

- Erstellen Sie manuell einen Benachrichtigungsauftrag über die Schablone „MyTISM-Benachrichtigungsauftrag ohne Vorlage (Vorgebaut; Versenden)“.
- Deaktivieren Sie „Nur Benutzer oder Gruppen“, um eine `MyTISMAdresseSMS` direkt als Empfänger auszuwählen.
- Senden Sie den Auftrag und prüfen Sie den Status im zugehörigen Lesezeichen `/Admins/MyTISM/Benachrichtigungen/MyTISM-Benachrichtigungsaufträge (Vorgebaut)`.
- **Status bleibt auf „2: In sending process“:** Der Webhook-Rückkanal ist gestört (siehe Punkt 4).
- **Status wechselt auf „5: Failed ultimately“ mit SocketTimeoutException:** Der OpenVPN-Tunnel ist vermutlich zusammengebrochen: Prüfen und ggfs. Voodoo-Reset durchführen.

3. Fehleranalyse beim SMS-Empfang

- Senden Sie eine SMS an das Android-Handy.
- Prüfen Sie, ob eine `EingehendeBenachrichtigung` erzeugt wurde.
- Fehlt das Objekt, ist der Webhook-Rückkanal blockiert.

4. Fehleranalyse des Webhook-Rückkanals (Routing)

- Prüfen Sie über `http://<gatewayHost>:8080/webhooks`, ob genau vier Webhooks registriert sind.
- Testen Sie den Rückkanal durch einen GET-Request an den Webhook-Listener (dieser wird absichtlich mit einem HTTP-Fehler quittiert):
  - Lokale Entwickler-Instanz: `curl http://ws-<ENTWICKLER-RECHNER-NAME>:<webhooksListenAtPort>/smsgatewaywebhooks/smsdelivered`
  - Produktiv-Instanz: `curl http://localhost:<webhooksListenAtPort>/smsgatewaywebhooks/smsdelivered`
  - Externer Test: `curl http(s)://<webhooksSendToHost>:<webhooksSendToPort>/smsgatewaywebhooks/smsdelivered`
- Bei erfolgreichem Routing müssen im `/var/log/nginx/access.log` zeitnah zwei POST-Einträge mit Statuscode `202` erscheinen:
  - `POST /smsgatewaywebhooks/smssent HTTP/2.0` (Zustellung an App erfolgt).
  - `POST /smsgatewaywebhooks/smsdelivered HTTP/2.0` (Erfolgreich über das Mobilfunknetz versendet).

7.5. Architektur, Backend & Deep-Dive

Dieser Bereich beleuchtet programmatische Schnittstellen, BOMasken-Performance und potenzielle Zirkelbezüge für Entwickler.

7.5.1. Performance-Optimierung und BOMasken-Typen

BOMasken mit reinen Groovy-Skripten belasten das System, da jedes Objekt einzeln ins RAM geladen wird (O(N)). Bevorzugen Sie `OQLBOMasken` oder `GrooqlBOMasken`, da diese direkt auf Datenbankebene filtern.

Müssen Skripte verwendet werden, sind „Early Exits“ vorgeschrieben. Prüfen Sie billige Attribute immer zuerst, bevor teure Abfragen (Many-Relationen) ausgewertet werden:

```
// Check cheap local boolean flag first
if (bo.Ldel) {
   return false
}
// String comparison is slightly more expensive
if (bo.Name == null || bo.Name != 'Desired Name') {
```

```
   return false
}
// Querying many-relations is highly expensive and must be the
last step
def hasPreferredMember = bo.Mitglieder.find { it.istBevorzugt }
if (!hasPreferredMember) {
   return false
}
return true
```

7.5.2. Datenbank-Architektur und die „bas“-Tabelle

Die PostgreSQL-Systemtabelle `bas` verwaltet Auslösezustände für BBTs und Wiedervorlagen und wird direkt über SQL-Ebene gepflegt.

Spaltenstruktur der Tabelle `bas`*:*

- `alarm`: ID des Alarms.
- `alarmbot`: Abstrakter Basistyp (`BOBasierterTermin` oder `Wiedervorlage`).
- `bo`: ID des überwachten Objekts.
  Sonderfall: Für jede Wiedervorlage darf es maximal einen einzigen Eintrag geben, bei dem `bo` gleich `NULL` ist (wenn die WV das Erstellen neuer BOs überwacht).
- `bot`: BOT-ID des überwachten Objekts.
- `datumstart`: Berechnete Ankerzeit für die Auslösung.
- `active`: Status der Auslösung. Erlaubte Kombinationen:
  1. `datumstart != NULL` & `active = true` (Aktiv, steht zur Auslösung an).
  2. `datumstart != NULL` & `active = false` (Erledigt).
  3. `datumstart == NULL` & `active = false` (Deaktiviert).

     Hinweis: Der Zustand `active == NULL` stellt einen Fehlerzustand bei der Datumsberechnung dar.
- `crea` / `lmod`: Zeitstempel des Eintrags.

7.5.3. Troubleshooting: BO-basierte Termine (BBT)

Hat ein BBT für ein Objekt nicht ausgelöst:

1. **Erwarteten Zeitpunkt berechnen:** Ermitteln Sie das theoretische Auslösedatum (inklusive Vorwarnzeit).

2. **Prüfen auf `AlarmAusloesung`:** Existiert das Objekt, schlug die nachgelagerte Aktion (z. B. das Benachrichtigungsskript) fehl.
3. **Prüfen der `bas`-Tabelle:** Suchen Sie den Eintrag mit `alarm = <BBT.Id>` und `bo = <BO.Id>`. Fehlt der Eintrag, passte das Objekt nie auf die BOMaske, oder es wurde nachträglich geändert und wieder gelöscht. Suchen Sie im Log nach `deleted because BO no longer fits the alarm.`.
4. **Prüfen des Alarmsystems:** War der Alarm aktiv (`BBT.Aktiv = true`) und lief das System zum Sendezeitpunkt? (Log: `Alarm system successfully initialized and ready.`).

Steuerung der Hintergrund-Initialisierung

Wird die BOMaske geändert, triggert dies eine asynchrone Neuinitialisierung der Status-Objekte. Dieser Prozess kann bei großen Datenbeständen durch Deaktivierung des Alarmsystems in der `mytism.ini` (`activateAlarme=never`) gestoppt werden. Wird das System reaktiviert (`if_possible`), setzt die Engine den Prozess automatisch am Abbruchpunkt fort.

7.5.4. Backend-Hooks und der Auslöse-Lifecycle

Bei der Alarmauslösung wird zuerst die Methode `trigger()` der Alarm-Klasse aufgerufen. Liefert diese `true`, wird die Auslösung als abgearbeitet betrachtet; Benachrichtigungsskripte und Standard-Benachrichtigungen werden übersprungen. Standardklassen liefern `false` zurück.

Benachrichtigungsskript

Wird ein Skript ausgeführt, stehen folgende Systemvariablen zur Verfügung:

- `api`: Das API-Objekt.
- `alarm`: Das Alarm-Objekt.
- `dateNow`: Auslösezeitpunkt.
- `log`: Der Logger.
- `idB0`: Objekt-ID.
- `bot`: Business Object Type.
- `bt`: Auslösende Transaktion (nur Hinweis und Wiedervorlage).
- **Transaktions-Singleton:** Über `api.getTransaction()` bezogene Transaktionen liefern innerhalb eines Skript-Kontexts immer dieselbe Instanz.
- **Rückgabewert:** Liefert das Skript `true`, werden Standard-Benachrichtigungen unterdrückt.

- **Skript-Exception:** Der gesamte Ablauf bricht ab und es werden keine Benachrichtigungen gesendet.

GSP-Templates und injizierte Variablen

In den GSP-Vorlagen stehen standardmäßig zur Verfügung:

- `benutzer` / `empfaenger`: Adressierter Empfänger.
- `alarm`: Auslösendes Alarm-Objekt.
- `dateNow`: Auslösedatum (`java.util.Date`).
- `api`: `TemplateScriptAPI`.
- `bo`: Auslösendes Geschäftsobjekt (nur BBT, Hinweis und Wiedervorlage)..
- `bt`: Auslösende Transaktion (nur Hinweis und Wiedervorlage).

Variablen in Auslösekriterien und -skripten

Bei Kriterien-Skripten injiziert das System:

- `valueNew` / `valueOld` / `valueCompare` (Als Java-Objekt).
- `schema` (Schema-Objekt).
- `attribute` / `type` (Attribut und CBO-Typ).
- `kriterium` / `bp` / `log`.

Im globalen Auslöseskript stehen zur Verfügung:

- `bo` (Geändertes Objekt).
- `schema` (Schema-Objekt).
- `bp` (Transaktionsschritt).
- `kriterium` (Kriterium-Objekt).
- `log` (Logger).

```
// Check if the medication dosage exceeds the defined critical
threshold
if (bo.getDosageInMg().intValue() >
bo.getMaxDosageInMg().intValue()) {
   // Trigger the alarm only if the doctor has not granted a
specific exception
   if (!bo.getExceptionGranted().booleanValue()) {
      return true
```

```
    }
}
return false
```

7.5.5. Programmatische Dateianhänge (DataSourceConvertibleI)

Sollen E-Mails programmatisch Dateien angehängt werden, muss die Klasse das Interface `de.ipcon.messaging.email.DataSourceConvertibleI` implementieren.

Es muss zwingend die Methode `getDataSource(context = DataSourceContext)` implementiert werden. Die parameterlose Variante `getDataSource()` ist obsolet.

Alternativ kann die Klasse `MyTISMAnhangBOEintrag` verwendet werden, um ein lesbares PDF zu generieren.

Wird der Anhang über ein Groovy-Skript gesteuert:

- `return null`: Kein Anhang.
- Rückgabe einer `Map`: Paralleles Anhängen mehrerer Objekte.

```
// Attach multiple clinical objects to the email dispatch
Map<String, Object> map = new HashMap<>()
map.put("Patient File", bo)
map.put("Attending Doctor", bo.getBehandelnderArzt())
return map
```

7.5.6. Programmatische Aufträge (Fluent API & Builder)

Zur Erstellung von Aufträgen steht eine Fluent API zur Verfügung (alternativ auch ein Builder-Pattern):

```
def notificationOrder = api.createBenachrichtigungsAuftrag()
      .setSender(sender)
      .addRecipientEmail('develop@oashi.com')
      .addRecipientEmailBCC('team@oashi.com')
      .setReplyToAddress(replyTo)
      .setL10nLocale(L10nLocale.byLocale(Locale.ENGLISH))
```

```
api.sendNotification(notificationOrder)
```

Für Aufträge ohne Vorlage existieren Setter wie `.setSubjectSource` und `.setSubjectSourceIsFixed`. Wird `api.sendNotification` aufgerufen und der Auftrag ist nicht neu oder wurde mit abweichender Transaktion erstellt, wirft das System eine `IllegalStateException`.

7.5.7. Reihenfolge der Versendung

Aufträge werden nach strikten Regeln versendet:

1. Höhere `Priorität` wird zuerst verarbeitet.
2. Bei gleicher Priorität gilt das FIFO-Prinzip nach Erstellungsdatum (`BeauftragtAm`).
3. Bei gleichem Datum entscheidet die interne Einreihung.

7.5.8. Architektonischer Schutz vor Endlosschleifen

Bedingte Alarme ignorieren Änderungen an Entitäten wie `Alarm`, `AlarmAusloesung`, `MyTISMBenachrichtigung` und `MyTISMBenachrichtigungsauftrag` (inklusive aller Subklassen), um kaskadierende Endlosschleifen zu verhindern.

Werden in einem Benachrichtigungsskript andere Business-Objekte manipuliert, kann dies andere Alarme triggern und bei unsauberem Design zu Zirkelbezügen führen.

7.5.9. Cluster-Synchronisation

- **Alarmsystem:** Darf ausschließlich auf dem autoritativen Server-Knoten laufen (`authoritative=1`), ansonsten statt Start Warnung im Log.
- **Benachrichtigungssystem:** Muss zwingend auf allen Knoten aktiv sein, da Popups im Solstice-Client nur lokal für den angemeldeten Benutzer generiert werden können.

Chapter 8. DSGVO & Datenlebenszyklus

8.1. Grundlagen & Konzepte

Patientendaten unterliegen strengen rechtlichen Vorgaben. MyTISM automatisiert den Datenlebenszyklus und garantiert bei entsprechender Konfiguration die Einhaltung der Datenschutz-Grundverordnung (DSGVO), insbesondere von Artikel 6 zur Rechtmäßigkeit der Verarbeitung. Statt Löschfristen manuell in Tabellen zu pflegen, definiert eine zentrale Konfigurationsdatei (das Schema-XML), welche Daten aus welchem Grund und wie lange gespeichert werden dürfen. Ein Hintergrunddienst scannt die Datenbank periodisch und berechnet für jeden Datensatz das frühestmögliche Löschdatum, sodass das System den Datenschutz rechtssicher und vollautomatisch umsetzt.

8.1.1. Datenlöschung: Weiches vs. hartes Löschen

Das System unterscheidet architektonisch zwischen zwei Arten der Löschung. Das weiche Löschen (Soft Delete) markiert Datensätze über die Benutzeroberfläche im Hintergrund lediglich als „gelöscht“, entfernt sie jedoch nicht physisch. Dies ermöglicht die sofortige Wiederherstellung bei fehlerhaften Eingaben, erfüllt jedoch nicht die gesetzlichen Anforderungen zur Datenvernichtung.

Für die gesetzeskonforme Vernichtung existiert das harte Löschen (Purge). Ist eine gesetzliche Aufbewahrungsfrist abgelaufen, erzwingt dieser Vorgang das unwiderrufliche physische Entfernen des Datensatzes aus der Datenbank.

Der Purge-Vorgang vernichtet Daten endgültig. Es existiert kein systeminterner Papierkorb für hart gelöschte Objekte. Eine Wiederherstellung ist ausschließlich über externe, physische Datenbank-Backups möglich.

8.1.2. Klassifizierung und Lösch-Vetorecht

Die DSGVO-Architektur basiert auf ineinandergreifenden Parametern:

- **Datenkategorie:** Klassifiziert die Art der Daten (z. B. sensible Gesundheitsdaten).
- **Verarbeitungszweck:** Definiert den legitimen Erhebungsgrund (z. B. medizinische Behandlung).
- **Aufbewahrungsfrist:** Legt länderspezifische Regeln und Speicherzeiträume fest.

Ein Hintergrunddienst berechnet die Löschfristen ressourcenschonend asynchron, um zeitkritische Prozesse wie das Speichern medizinischer Befunde nicht zu blockieren. Geringfügige Fristüberschreitungen werden dabei toleriert.

Zusätzlich greift das Lösch-Vetorecht: Entfällt der primäre Verarbeitungszweck eines Datensatzes, kann ein verknüpfter Datensatz die Löschung blockieren. So verhindert eine unbezahlte, abrechnungsrelevante Rechnung die Löschung der verknüpften Patientenakte.

Zur Fehlervermeidung vernichtet das System fällige Datensätze nicht am exakten Stichtag. Eine systemweite Sicherheitsmarge von standardmäßig 30 Tagen ab dem berechneten Löschdatum gleicht Ungenauigkeiten aus und verhindert Datenverluste durch administrative Fehler.

8.2. Benutzeroberfläche & Bedienung

Die DSGVO räumt Patienten weitreichende Rechte ein (Artikel 15 bis 22), die im Klinikalltag organisatorische und manuelle Eingriffe erfordern.

- **Art. 15 (Auskunftsrecht):** Verarbeitungszwecke sind im System hinterlegt; die eigentliche Auskunftserteilung muss jedoch manuell zusammengestellt werden. Ein automatisierter Export existiert nicht im Core.
- **Art. 16 (Recht auf Berichtigung):** Fehlerhafte Daten werden regulär über die Standard-Formulare der GUI korrigiert.
- **Art. 17 (Recht auf Vergessenwerden):** Vollautomatisierte Funktionen für sofortige Löschanfragen existieren nicht. Fordert ein Patient die Löschung, muss das früheste Löschdatum manuell überschrieben und die Neuberechnung für diesen Datensatz deaktiviert werden.
- **Art. 18 & 21 (Einschränkung und Widerspruch):** Spezifische Attribute zur systemseitigen Unterstützung (z. B. `GDPRObjectionDate`) existieren im Standard nicht und müssen organisatorisch gelöst werden.
- **Art. 19 & 20 (Mitteilungspflicht und Datenübertragbarkeit):** Mitteilungen an Dritte und der strukturierte Datenexport müssen manuell erfolgen.
- **Art. 22 (Automatisierte Entscheidungen):** Der Core trifft keinerlei automatisierte Einzelentscheidungen (Profiling).

8.3. Erweiterte Konfiguration (XML & Services)

Die Berechnung und Überwachung der Fristen wird administrativ über Business Services (BS) konfiguriert.

8.3.1. DSGVO-Berechnungsdienste über die GUI steuern

Zwei Dienste überwachen den Datenlebenszyklus:

1. `GDPRRetentionInitService`: Berechnet initial das früheste Löschdatum für neu

erfasste Datensätze.

2. `GDPRRetentionUpdateService`: Aktualisiert diese Fristen in regelmäßigen Intervallen.

Diese Dienste müssen nach einer Neuinstallation manuell konfiguriert werden. Der Administrator muss zwingend die Systembenutzer `MT_BS_GDPR_RETENTION_INIT` und `MT_BS_GDPR_RETENTION_UPDATE` anlegen und der Systemgruppe `RG_Services` zuweisen.

Die Dienste werden über die reguläre BS-Maske angelegt und mit einem CronJob versehen. Die XML-Konfigurationen für das `<Sync>`-Feld finden sich unter `/nrx/de/ipcon/resources/services/bs/`.

XML-Snippet für den Init-Service:

```
<Sync>
  <mytism-connection url="socket://localhost"
user="MT_BS_GDPR_RETENTION_INIT" pass=""/>
  <script language="groovy">
    import de.ipcon.db.gdpr.GDPRRetentionInitService
    new GDPRRetentionInitService(api, api.rcl).runService()
  </script>
</Sync>
```

XML-Snippet für den Update-Service:

```
<Sync>
  <mytism-connection url="socket://localhost"
user="MT_BS_GDPR_RETENTION_UPDATE" pass=""/>
  <script language="groovy">
    import de.ipcon.db.gdpr.GDPRRetentionUpdateService
    new GDPRRetentionUpdateService(api, api.rcl).runService()
  </script>
</Sync>
```

8.4. System-Administration (Server-Ebene)

Die finale Datenvernichtung erfolgt tief im System-Backend auf Server-Ebene.

8.4.1. Konfiguration der finalen Datenvernichtung (Der Purger)

Der Task zur endgültigen Löschung (`de.ipcon.db.Purger`) wird ausschließlich im Abschnitt `[Purger]` der Datei `mytism.ini` gesteuert. Das System erzwingt die 30-Tage-Karenzzeit. Als Sicherheitsmechanismus berechnet der Purger das erwartete Löschdatum unmittelbar vor dem Löschvorgang zwingend neu. Nur bei exakter Übereinstimmung mit dem in der Datenbank gespeicherten Datum wird der Datensatz gepurged. Abweichungen aktualisieren lediglich den Datenbank-Eintrag.

```
[Purger]
activatePurger=1
initialDelayDays=7
delayDays=7
```

- `activatePurger`: Aktiviert (`1`) oder deaktiviert (`0` / fehlend) den Purger-Task.
- `initialDelayDays`: Wartezeit in Tagen vor dem ersten Start nach Systemstart. Ein Wert `≤ 0` startet den Task sofort (Standard: 7).
- `delayDays`: Wartezeit in Tagen zwischen Task-Ausführungen (Standard: 7).

Jede manuelle Änderung in der `mytism.ini` bricht einen aktiven Purger-Task sofort ab. Der Task wird neu eingeplant, wobei die `initialDelayDays`-Wartezeit wieder vollständig von vorne beginnt.

8.5. Architektur, Backend & Deep-Dive

Die technische DSGVO-Architektur basiert im Backend auf spezifischen XML-Elementen und Datenbank-Interfaces.

8.5.1. Globale Platzierung: Zwecke, Interessen & Gesetze

Wiederverwendbare rechtliche Konstrukte werden im XML-Schema als globale Elemente definiert. Elemente wie `<GDPRProcessingPurpose>`, `<GDPRLaw>`, `<GDPRDataCategory>`, `<GDPRBusinessInterest>`, `<GDPRProcessingLegalBasis>` und `<GDPRRetentionPurpose>` werden zwingend als direkte Unterelemente des Wurzelknotens `<Schema>` platziert und niemals innerhalb von Entitäten verschachtelt.

Ein `<GDPRBusinessInterest>` repräsentiert das legitime Geschäftsinteresse. Der `<GDPRProcessingPurpose>` definiert den Verarbeitungszweck, der über `<LegalBasis>` eine gesetzliche Grundlage referenziert.

```
<Schema>
  <GDPRBusinessInterest id="LegalCompliance"/>
  <GDPRProcessingPurpose id="ContractualPerformance">
    <LegalBasis id="ContractualPerformance"/>
  </GDPRProcessingPurpose>
  <GDPRProcessingLegalBasis id="ContractualPerformance">
    <Law id="EU_DSGVO_6_1_b"/>
  </GDPRProcessingLegalBasis>
</Schema>
```

Für das referenzierte Gesetz (`<GDPRLaw>`) verlangt das Attribut `country` zwingend den ISO 3166-1 alpha-2 Ländercode. Über `url` kann ein Weblink zur Online-Ressource hinterlegt werden.

```
<GDPRLaw
  id="EU_DSGVO_6_1_b"
  paragraph="Art. 6 Abs. 1 lit. b DSGVO"
  country="EU"
  url="https://dsgvo-gesetz.de/art-6-dsgvo/"/>
```

Diese Bausteine werden abschließend in einer `<GDPRDataCategory>` gebündelt.

```
<GDPRDataCategory id="ContractData">
  <BusinessInterest id="LegalCompliance"/>
  <ProcessingPurpose id="ContractualPerformance"/>
  <RetentionPurpose id="LegalCompliance"/>
</GDPRDataCategory>
```

8.5.2. Richtlinien zur Aufbewahrung (Data Retention Policies)

Die `<GDPRDataRetentionPolicy>` legt Fristen und Regeln für eine Datenkategorie fest. Die Angabe der gesetzlichen Grundlage über ein `<Law>`-Tag ist zwingend erforderlich.

Das System erzwingt eine strikte Bindung: Das Land des referenzierten `<GDPRLaw>`-Elements muss zwingend mit dem `country`-Attribut der Policy übereinstimmen.

Die Dauer (`period`) wird case-insensitive angegeben (z. B. "10y"). Der Startzeitpunkt (`start`) kann `calendar_year_end`, `business_year_end` oder `custom` sein. Auslöser (`trigger`) sind Ereignisse wie `termination` (Vertragsende), `creation` oder `last_transaction`. Das Datenprüfintervall auf Löschfähigkeit wird über `evaluationFrequency` definiert. Dokumentierend sind `lastAssessmentDate` (letzte rechtliche Bewertung) und `rightsAndFreedomsAssessed` (ob Betroffenenrechte und -freiheiten bei Fristfestlegung formell abgewogen wurden).

```
<GDPRDataRetentionPolicy
  country="DE"
  category="ContractData"
  period="10y"
  start="custom"
  trigger="termination"
  evaluationFrequency="annually"
  lastAssessmentDate="2024-08-28"
  rightsAndFreedomsAssessed="true">
  <Law id="DE_BGB_195"/>
</GDPRDataRetentionPolicy>
```

Wenn `start="custom"` gewählt wird, bestimmt das System das Startdatum über den Attribut-Pfad, der in der Entität im Tag `<gdpr retentionStartDatePath="…">` konfiguriert wurde.

8.5.3. DSGVO-Metadaten direkt an der Entität verankern

Entitäten müssen zwingend mit einem `<gdpr>`-Tag ausgestattet werden, welches die Entität mit einer Datenkategorie verknüpft. Das Element `<affectedPerson>` referenziert die betroffene Person. Ohne expliziten `retentionStartDatePath` gilt das Erstellungsdatum als Startpunkt.

```
<Entity name="Behandlungsvertrag">
  <gdpr dataCategory="ContractData"
retentionStartDatePath="Crea">
    <affectedPerson path="VertragsPartnerKunde.AbstraktePerson"/>
  </gdpr>
</Entity>
```

Das Element `<retentionVeto>` definiert Verknüpfungen zu Datensätzen, die ein Lösch-

Veto einlegen können.

```
<Entity name="Krankenhausrechnung">
  <gdpr dataCategory="InvoiceData">
    <affectedPerson path="Patient.AbstraktePerson"/>
    <retentionVeto path="BuchungsKonto.AbstraktePerson"/>
  </gdpr>
</Entity>
```

8.5.4. Datenbank-Interfaces & Zeitstempel

Überwachte Entitäten müssen das Core-Interface `GDPRRelevantBaseI` implementieren. Zusätzlich stellt die Klasse `de.ipcon.db.core.GDPRRelevantI` Hilfsmethoden bereit.

Das Interface erzwingt auf Datenbankebene zwei Attribute:

- `GDPREarliestDeletionDate`: Speichert das Löschdatum (Uhrzeit hart auf 00:00 Uhr gesetzt).
- `GDPREarliestDeletionDateLmod`: Protokolliert Modifikationen der Frist millisekundengenau.

Die Sicherheitsmarge von 30 Tagen ist in der Backend-Konstante `GDPRTools.SAFETY_MARGIN_DAYS_BEFORE_DELETION` verankert.

8.5.5. Deep-Dive: Fristenberechnung in den Diensten

Die Klassen `GDPRRetentionInitService` und `GDPRRetentionUpdateService` im Package `de.ipcon.db.gdpr` erben von `GDPRRetentionService` und arbeiten chronologisch.

Der **Initialisierungsdienst** liest die Konfiguration aus dem Schema-XML und identifiziert alle relevanten Entitäten. Er berechnet das frühestmögliche Löschdatum und speichert dieses sowie den Zeitstempel der Berechnung ab.

Der **Aktualisierungsdienst** operiert iterativ in konfigurierten Zeitintervallen. Er berechnet das Datum für relevante Entitäten komplett neu. Nur bei einer Abweichung zum aktuell gespeicherten Datum wird ein Schreibvorgang ausgelöst.

8.5.6. Historische Logs bereinigen & Soft-Delete Filter

Nach dem Purgen eines Datensatzes bleiben historische Transaktions-Logs in der Datenbank erhalten. Diese Objekte (`OldValue` und `NewValue`)

> können weiterhin sensible Klartextdaten enthalten. Entwickler müssen diese historischen Felder programmatisch auf `null` setzen oder die Einträge proaktiv löschen, um DSGVO-Verstöße zu verhindern.

Da Soft-Delete-Objekte physisch verbleiben, müssen Entwickler in eigenen OQL-Abfragen zwingend den Zusatz `WHERE NOT Ldel` verwenden. Wird dies vergessen, verarbeitet das System weich gelöschte Daten, was zu kritischen Fehlinformationen führt.

Chapter 9. Rechtesystem & Benutzerverwaltung

9.1. Grundlagen & Konzepte

Das Rechtesystem von MyTISM garantiert die Vertraulichkeit sensibler Daten und ermöglicht gleichzeitig einen reibungslosen Betriebsablauf. Im Gegensatz zu komplexen Einzelberechtigungen setzt das Framework auf ein strikt rollenbasiertes Modell. Rechte werden nicht an individuelle Personen vergeben, sondern ausschließlich an Benutzergruppen gekoppelt. Einem Benutzer werden Arbeitswerkzeuge und Datenzugriffe automatisch durch die Zuweisung zu fachlichen Gruppen (z. B. „Chirurgen“ oder „Verwaltung“) bereitgestellt.

Standardmäßig existiert eine Systemgruppe „Admins“ sowie eine allgemeine Gruppe „Benutzer“. Ein sicherheitskritischer Anker ist die Systemgruppe `RG_Solstice_Login`. Ausschließlich Mitglieder dieser Gruppe sind berechtigt, sich an der grafischen Benutzeroberfläche des Clients anzumelden.

9.1.1. Sicherheit durch die Positivliste

Das Rechtesystem basiert auf dem Architektur-Konzept der Positivliste. Im Grundzustand verfügt ein Benutzer über keinerlei Rechte im System; er sieht weder Daten noch Formulare. Zugriffsberechtigungen müssen explizit und gezielt gewährt werden.

Jede Berechtigung basiert auf vier Parametern:

1. **Wer:** Die berechtigte Gruppe (z. B. „Pflegepersonal“).
2. **Was:** Die betroffene Datenkategorie oder das konkrete Objekt.
3. **Wie:** Die zulässige Aktion (Lesen, Schreiben, Erstellen, Löschen, oder Ablehnung davon).
4. **Warum:** Eine deklarative Bemerkung, die bei explizit verweigerten Zugriffen als Begründung im UI ausgegeben wird.

9.2. Benutzeroberfläche & Bedienung

Das Rechtesystem steuert dynamisch die Sichtbarkeit und Bedienbarkeit der Benutzeroberfläche. Besitzt ein Nutzer auf Datenbankebene keine Leserechte für bestimmte Objekte, blendet das System die darauf basierenden Lesezeichen und Schablonen im Navigationsbaum automatisch aus.

Benutzer im Navigationsbaum werden bei einer großen Anzahl vom System automatisch

alphabetisch gruppiert. Wird der Zugriff auf ein angefordertes Objekt oder Formular verweigert, gibt die Benutzeroberfläche die zentral hinterlegte Bemerkung der Rechtezuweisung als Hinweistext aus.

9.3. Verwaltung & XML-Konfiguration

Die Steuerung der Zugriffsrechte, Sichtbarkeiten und GUI-Strukturen erfolgt zentral durch Administratoren.

9.3.1. Rechtezuweisungen & UI-Sichtbarkeit

Die Definition von Berechtigungen erfolgt im Formular der jeweiligen Gruppe im Reiter „Rechte-Zuweisungen“. Über sogenannte BO-Masken wird definiert, auf welche Datenbestände zugegriffen werden darf. Das System erlaubt hierbei eine feingranulare Steuerung auf Spaltenebene. Über Konfigurationsfelder können Rechte auf einzelne Attribute isoliert werden, um beispielsweise den Lesezugriff auf eine Patientenakte zu gewähren, während das Attribut „Diagnose“ für diese Gruppe verborgen oder schreibgeschützt bleibt.

Um Befugnisse hart zu entziehen, wird das Flag „Ablehnen“ bei der Aktion gesetzt. Dieses Flag wirkt als absolutes Veto, das sämtliche Rechte überschreibt, die dem Nutzer eventuell durch andere Gruppenmitgliedschaften gewährt wurden. Das Flag „Ablehnen aufheben“ deaktiviert ein solches Veto für privilegierte Benutzergruppen gezielt. Technische Voraussetzung hierfür ist, dass die aufhebende Zuweisung exakt dieselbe BO-Maske und dieselben Aktions-Flags besitzt wie die ursprünglich verbietende Zuweisung.

Änderungen an Gruppen oder Rechtezuweisungen greifen aktuell noch nicht sofort im laufenden Client-Betrieb. Um die neuen Rechte zu erhalten, müssen sich die betroffenen Benutzer zwingend ab- und wieder anmelden.

In den Ordnereinstellungen lässt sich die Sichtbarkeit für bestimmte Gruppen definieren; eine leere Liste beschränkt den Zugriff standardmäßig auf Administratoren. Innerhalb von Strukturelementen steuert die Relation „Gruppen“ (im XML über das Tag `<Gruppen>`) die Sichtbarkeit im Navigationsbaum, unabhängig von den eigentlichen Datenrechten.

```
<Gruppen>
  <Gruppe Name="Admins"/>
  <Gruppe Name="Onkologen"/>
</Gruppen>
```

Zusätzlich kann jedem Strukturelement ein Groovy-Skript zur dynamischen Sichtbarkeitssteuerung angehängt werden, in das die Variablen `user` und `struktur` injiziert werden.

```
return user.istMitgliedVon("Chefaerzte") ||
struktur.getName().contains("Freigegeben")
```

9.3.2. Benutzerpflege & Globale Variablen

Die Verwaltung von Anwendern erfolgt im Navigationsbaum unter „Admins → MyTISM → Benutzerverwaltung“. Werden systemweite GUI-Verhaltensweisen über „Einstellungen-Variablen“ konfiguriert und an Benutzer oder Gruppen gebunden, greift eine harte Priorisierung.

Damit die Priorisierungskette evaluiert wird, muss in der Definition der Variable das Flag „Ueberschreibbar“ gesetzt sein. Ist dieses Flag nicht aktiv, ignoriert das System alle benutzer- oder gruppenspezifischen Zuweisungen und erzwingt global den Standardwert. Ist das Flag aktiv, gilt folgende Priorität:

1. Explizit für den Benutzer definierter Wert.
2. Definierter Wert der Gruppe.
3. Bei konkurrierenden Werten aus mehreren Gruppenmitgliedschaften gewinnt der Wert der Gruppe mit der niedrigsten Datenbank-ID.

9.3.3. Administrative Strukturierung großer Datenmengen

Um bei tausenden Systembenutzern die Übersicht zu wahren, gruppiert MyTISM den Ordner „Alle GUI-Benutzer“ anhand globaler UI-Parameter automatisch.

- `users.view.groupingStart`: Startgrenze für die Gruppierung (Standard: 30).
- `users.view.enableGrouping`: Globale Aktivierung der automatischen Ordnung.
- `users.view.group.forceAlphaNumBreak`: Erzwingt einen harten Trennschnitt zwischen numerischen und alphabetischen Logins.
- `users.view.group.maxElements`: Maximales Limit an Benutzern pro Unterordner (Standard: 15).
- `users.view.group.minElements`: Mindestanzahl an Benutzern pro Unterordner vor Neuanlage (Standard: 5).
- `users.view.group.numericRange`: Schrittweite zur Gruppierung numerischer Logins (Standard: 10).

- `users.view.group.maxSubgroups`: Maximalanzahl an gemischten Untergruppen pro Ordner (Standard: 4).

In Gruppenskripten darf aktuell ausschließlich Beanshell (kein Groovy) verwendet werden. Es werden keine Kontext-Variablen injiziert; lediglich die Methode `log()` steht zur Generierung von Log-Ausgaben zur Verfügung.

9.4. Architektur, Backend & Deep-Dive

Auf Architekturebene wird das Rechtesystem durch Hooks in den Business Objects und performante Filtermechanismen abgesichert.

9.4.1. Programmatische Hooks: `isReadOnly` und `isMandatory`

Während GUI-Rechte statisch konfiguriert werden, erlauben Hooks in der BO-Klasse dynamische Evaluierungen zur Laufzeit. Schreibrechte werden in einer fest definierten Hierarchie geprüft: Zuerst wird das Schema-XML ausgewertet (z. B. `readonly="true"` oder virtuelles Attribut). Gibt diese Basis-Prüfung `false` zurück, evaluiert die Engine den `PermissionHandler`, um Gruppenrechte und BO-Masken zu prüfen.

Über die Methode `isReadOnly()` lassen sich Attribute programmatisch sperren.

```
import de.ipcon.schema.AttributeI

@ENTITY Patient@
method isReadOnly(a = AttributeI) returns boolean
   if getEntlassenNN() and ATT_Hauptdiagnose == a.getName() then
      return 1
   return super.isReadOnly(a)
```

In der BO-Klasse lautet die Methode `isReadOnly` (großes O), im Interface `AttributeI` hingegen `isReadonly` (kleines o). Der Aufruf von `super.isReadOnly(a)` ist obligatorisch, da ohne ihn die serverseitige Validierung im `PermissionHandlerI` fehlschlägt.

Analog erzwingt die Methode `isMandatory()` Pflichtfelder programmatisch auf Backend-Ebene.

9.4.2. Hochperformantes Filtern: Grooql & Two-Step-Filtering

Wird in einer BOMaske ein Skript hinterlegt, injiziert der Server zur Laufzeit die Variablen `bo`, `schema`, `entity`, `att`, `maske`, `user` und `log` in den Auswertungskontext.

Einfache Skript-Masken sind ineffizient, da das Skript für jedes Objekt bei jedem Datenzugriff im Arbeitsspeicher iterativ ausgewertet werden muss. Die Lösung ist das Two-Step-Filtering der `GrooqlBOMaske`: Im ersten Schritt wird das Skript auf SQL-Ebene in eine OQL-Abfrage transformiert, um eine Datenbankvorfilterung zu erzielen. Im zweiten Schritt führt die Methode `fits()` auf Java-Ebene die finale Evaluierung im Speicher durch.

Die `OQLBOMaske` verlagert die Filterung exklusiv in die Datenbank. Sie nutzt keine Groovy-Skripte, sondern leitet direkte OQL-Klauseln aus dem Feld `WhereClauses` ab. Aufgrund der nativen Evaluierung auf Datenbankebene ist die `OQLBOMaske` für systemkritische Hintergrunddienste stets die performanteste Architektur-Wahl.

Chapter 10. Hintergrunddienste & Initialdaten

Automatisierte Prozesse entlasten das System und die Anwender im Arbeitsalltag. Bevor ein System produktiv genutzt werden kann, müssen systemweite, statische Rahmenbedingungen definiert sein. Diese feststehenden Basisinformationen werden in MyTISM als „Initialdaten“ bezeichnet. Ergänzend dazu übernehmen „Hintergrunddienste“ (Business Services) wiederkehrende Routineaufgaben völlig automatisch. Beide Konzepte zusammen gewährleisten einen stabilen und wartungsarmen Systembetrieb.

10.1. Grundlagen & Konzepte

Um redundante Eingaben und inkonsistente Stammdaten zu vermeiden, liefert MyTISM unveränderliche Initialdaten wie Ländercodes oder Maßeinheiten fest im Systemkern mit. Ein Coredata-Generator baut diese Basis beim ersten Serverstart vollautomatisch auf. Er generiert den initialen Administrator-Benutzer sowie fertige Eingabemasken, Lesezeichen und Berichte für Standard-Datentypen.

Zur Vermeidung von Datenwidersprüchen gilt das Prinzip der zentralen Datenhaltung. Ausschließlich der autoritative Hauptserver darf Initialdaten erzeugen und verwalten. Für die Offline-Fähigkeit mobiler Endgeräte läuft auf diesen Geräten ein vollwertiger lokaler MyTISM-Server im Hintergrund. Die Datensynchronisation erfolgt ausschließlich zwischen diesen Server-Knoten und niemals direkt zwischen den Benutzeroberflächen. Sobald ein lokaler Client-Server wieder über eine Netzwerkverbindung verfügt, synchronisiert er die Initialdaten passiv und automatisch vom Hauptserver.

10.1.1. Hintergrunddienste

Hintergrunddienste agieren proaktiv und autonom im System. Sie ermöglichen beispielsweise den automatisierten nächtlichen Import von Systemdaten, den selbstständigen Versand von Benachrichtigungen oder die kontinuierliche Überwachung von Beständen. Diese Auslagerung von Routineaufgaben in den Systemhintergrund reduziert menschliche Fehler und entlastet die Anwender im operativen Betrieb signifikant.

10.2. Benutzeroberfläche & Bedienung

Die Verwaltung und Überwachung von Hintergrunddiensten (Business Services, BS) erfolgt für Administratoren direkt in der grafischen Benutzeroberfläche des Solstice-Clients. Die entsprechende Maske befindet sich im Navigationsbaum unter „Admins → MyTISM → Interna → BSs“. Zur Fehleranalyse bei Dienstausfällen bietet das Formular jedes Business

Services einen dedizierten Reiter „Stacktrace“. Dieser Reiter protokolliert den exakten technischen Fehlerbericht des jeweils letzten fehlerhaften Skript-Durchlaufs.

10.3. Verwaltung & XML-Konfiguration

Die Neuanlage und präzise Taktung von Hintergrunddiensten erfolgt über die administrative Eingabeschablone für Business Services.

10.3.1. Verwaltung von Hintergrunddiensten

Jeder Business Service erfordert zwingend die Angabe eines Verantwortlichen sowie einen beschreibenden Namen. Im Feld „Bei Ausfall benachrichtigen“ kann eine Administrator-Gruppe definiert werden, die bei Skriptfehlern alarmiert wird. Für reguläre Skripte muss im Feld „Java-Klasse“ der Standardwert `de.ipcon.db.sync.ScriptService` hinterlegt werden. Ein „Business Node“ repräsentiert in der Architektur die physische Server-Instanz beziehungsweise den Netzwerkknoten.

Ein Business Service muss im Reiter „Dienst“ zwingend mindestens einem Business Node zugewiesen werden. Bleibt dieses Feld leer, wird das Skript im System nicht ausgeführt.

Die zeitliche Ausführungsvorschrift wird über ein XML-Snippet in klassischer Cron-Syntax gesteuert. Das folgende Beispiel konfiguriert eine tägliche Ausführung um exakt 15:10 Uhr:

```
<ExecutionPolicy>
  <CronJob interrupt="false">
    <Commands>
      <Command>10 15 * * *</Command>
    </Commands>
  </CronJob>
</ExecutionPolicy>
```

Der Parameter `interrupt="true"` erzwingt den Abbruch eines noch laufenden Prozesses desselben Jobs bei der nächsten fälligen Ausführung. Der Standardwert `false` erlaubt parallele Überschneidungen, beispielsweise bei langlaufenden Datenimporten.

Neben der Cron-Syntax sind erweiterte Makros zulässig: `@yearly` (1. Jan, 0:00 Uhr), `@monthly` (Erster des Monats), `@weekly` (Sonntag, 0:00 Uhr), `@daily` (Mitternacht) und `@hourly` (volle Stunde). Ein Intervall wie `*/5` im Minuten-Feld definiert „alle 5 Minuten“. Schrittweiten wie `0-23/2` im Stunden-Feld definieren „jede zweite Stunde“. Spezielle Wochentags-Definitionen wie `@lastOfM` oder `@lastOfY` sind ebenfalls möglich (z. B.

`4/@lastOfM` für den letzten Mittwoch des Monats).

Eine Ausführungsvorschrift kann den Dienst auch als ununterbrochenen Dauerläufer deklarieren, was für Echtzeit-Schnittstellen erforderlich ist:

```
<ExecutionPolicy>
  <ExecutionPolicyKeepRunning/>
</ExecutionPolicy>
```

Die Logik des Dienstes wird im Reiter „Script“ in einem XML-Knoten hinterlegt. Der Groovy-Code wird in ein `<Sync>`-Tag sowie einen CDATA-Block eingebettet.

```
<Sync>
  <mytism-connection url="socket://localhost"
user="MT_BS_VITALWERTE" pass="aPassword"/>
  <script language="groovy"><![CDATA[
    tx = api.getNewTx()
    // Implement custom script logic here
  ]]></script>
</Sync>
```

10.4. System-Administration (Server-Ebene)

Server-Administratoren können Dienste zu Testzwecken jederzeit manuell über die Kommandozeile ausführen:

```
./mytism run-bs /Pfad/zum/Script
```

10.4.1. Konfiguration der Initialdaten (.initialdata.xml)

Fest vorgegebene Systemwerte werden als XML-Dateien (`.initialdata.xml`) in den Modul-Ordnern deklariert und beim Serverstart automatisch importiert. Die Grundstruktur definiert die zu befüllenden Datenbankspalten.

```
<initialdata>
  <data>
    <column attr="Name"/>
```

```
    <column attr="Beschreibung" replaceEscaped="true"/>
  </data>
</initialdata>
```

Bei der Spaltendefinition im `<column>`-Tag ist keine explizite Datentypangabe erforderlich. Alle Werte im Daten-Block müssen in doppelte Anführungszeichen gesetzt und durch Semikolons getrennt werden. Das Parsing ist intern an die deutsche System-Locale gebunden; für Fließkommazahlen muss daher zwingend das Komma als Dezimaltrennzeichen verwendet werden. Für mehrzeilige Inhalte müssen Zeilenumbrüche als `\n` escaped und im `<column>`-Tag das Attribut `replaceEscaped="true"` gesetzt werden.

Über ein integriertes Groovy-Skript lässt sich das Update-Verhalten steuern:

```
<initialdata>
  <script language="groovy"><![CDATA[
    importAndUpdateExisting()
  ]]></script>
</initialdata>
```

Die Methode `start()` importiert ausschließlich fehlende Werte und ignoriert bestehende Datensätze (Standard). Die Methode `importAndUpdateExisting()` importiert fehlende Werte und überschreibt bestehende Datensätze zwingend. Die Methode `updateExistingOnly()` aktualisiert ausschließlich bereits vorhandene Einträge.

Beim Import aus kommagetrennten Strings müssen vorkommende Anführungszeichen verdoppelt werden. Folgt direkt nach einem verdoppelten Anführungszeichen ein Semikolon, bricht der interne Parser jedoch fehlerhaft ab.

10.5. Architektur, Backend & Deep-Dive

Die Zuverlässigkeit des verteilten Systems basiert auf einer robusten Backend-Orchestrierung. Dieser Abschnitt beleuchtet die programmatischen Mechanismen, APIs und Infrastruktur-Konzepte für Entwickler.

10.5.1. Business Services API & Architektur

Hintergrunddienste werden im Backend als Subklassen der Basisklasse `de.ipcon.db.sync.ScriptService` implementiert. Die Engine injiziert implizit folgende

API-Methoden und Variablen in die Skriptumgebung:

- `getBO(id)`: Lädt ein Business Object gezielt anhand seiner Datenbank-ID.
- `query("OQL")`: Führt komplexe Suchanfragen über die Object Query Language aus.
- `newTx()`: Erstellt einen isolierten Transaktions-Kontext für atomare Datenbankoperationen.
- `save()`: Speichert getätigte Änderungen asynchron in die PostgreSQL-Datenbank.
- `log`: Zugriff auf den System-Logger zur Protokollierung von Statusmeldungen.
- `args`: Array der beim Aufruf übergebenen Kommandozeilen-Parameter.

Aus Performance-Gründen nutzen Hintergrunddienste kein aktives Event-System, sondern basieren auf einem ressourcenschonenden Polling-Mechanismus. Datenschutz-Dienste lauschen nicht auf Änderungen, sondern pollen relevante Datensätze in ihren festgelegten CronJob-Intervallen auf neuen Berechnungsbedarf.

10.5.2. Initialdaten-Skripting

Komplexe Initialdaten-Szenarien erfordern Groovy-Logik innerhalb des `<script>`-Tags.

```
<initialdata>
  <script language="groovy"><![CDATA[
    // Check if the table 'Mitarbeiter' is currently empty
    def count = tx.queryBO("SELECT count(bo.Id) FROM Mitarbeiter
bo WHERE Not Ldel").get(0) as Long

    if (count.longValue() == 0) {
      start() // Only create new entries if table is empty
    } else {
      importAndUpdateExisting() // Otherwise, overwrite existing
data
    }
  ]]></script>
</initialdata>
```

Beim Skripten von Initialdaten gelten zwei architektonische Restriktionen. Erstens kann die Spalte `vattrName` für Core-Entitäten nicht verwendet werden, da für diese Module keine projektspezifischen Custom-Klassen mit den erforderlichen Konstanten generiert werden können. Zweitens unterstützt der XML-Parser das direkte Initialisieren von 1:n- oder n:m-Relationen nativ nicht. Solche komplexen Verknüpfungen müssen als Workaround manuell

über Groovy-Closures im Skript-Block iteriert und relationiert werden.

Chapter 11. Web-Server (Cauldron) & REST-APIs

11.1. Grundlagen & Konzepte

Der integrierte Webserver Cauldron fungiert als zentrale Kommunikationsschnittstelle des MyTISM-Frameworks. Er ermöglicht die standardisierte Interaktion externer Applikationen und Clients mit dem Backend und ist als leichtgewichtiges Subsystem direkt in die Framework-Architektur eingebettet.

Cauldron erfüllt primär zwei Aufgaben: Zum einen stellt er die interne „Deploy-Seite“ als klassische Weboberfläche bereit, über die Administratoren Dokumentationen einsehen und den Solstice-Client bereitstellen. Zum anderen dient er als moderne REST-API für Drittsysteme zum strukturierten Datenaustausch mit dem Backend. Der Webserver verarbeitet drei Inhaltstypen: Groovlets (Schnittstellenlogik), GSP-Templates (serverseitiges HTML) und statische Ressourcen.

Hinsichtlich der Datenpräsentation unterstützt die Architektur zwei Konzepte: Beim serverseitigen Rendering (Server-Side Rendering) generiert das System vollständige HTML-Seiten, was sich für zustandslose Übersichten oder Dashboards eignet. Beim clientseitigen Rendering (Client-Side Rendering) agiert Cauldron als zustandsloser Datenlieferant und übermittelt strukturierte Payloads im JSON-Format. Die visuelle Aufbereitung erfolgt autark durch clientseitige Frameworks (z. B. Single Page Applications).

Ein architektonischer Kernvorteil ist die hohe Durchsatzkapazität. Cauldron verzichtet auf starre Obergrenzen für die Dateigröße von Uploads oder die Länge von JSON-Nachrichten, um extrem große Datenmengen (z. B. BLOBs) performant zu übertragen. Um eine Überlastung durch Request-Spitzen zu verhindern, greifen restriktive Schutzmechanismen auf Netzwerkebene, die überzählige Anfragen drosseln oder abweisen.

Der Zugriff auf Web-Ressourcen und API-Endpunkte erfordert zwingend eine Authentifizierung über die regulären Anmeldedaten. Das System evaluiert die Gruppenzugehörigkeiten und blockiert den Zugriff, sofern keine Mitgliedschaft in Systemgruppen wie „Admins“, „RG_Solstice_Login“ oder „RG_Deploy“ vorliegt.

11.2. Benutzeroberfläche & Bedienung

Für normale Anwender agiert Cauldron unsichtbar im Hintergrund; sie rufen lediglich die Startadresse auf, um den Client herunterzuladen. Die Bereitstellung dieser Dienste obliegt Administratoren.

Power-User, die über Tools wie cURL dynamische Daten aus Lesezeichen-APIs abfragen,

müssen eine Formatierungsregel beachten: Werden mehrere Filter-Parameter via Kaufmanns-Und (&) übergeben, muss die URL in der Kommandozeile zwingend in doppelte Anführungszeichen gesetzt werden, da das Zeichen andernfalls als Befehlsende interpretiert wird.

11.3. Verwaltung & Server-Konfiguration

Dieser Abschnitt richtet sich an Administratoren, welche das System verwalten und absichern.

11.3.1. Die Deploy-Seite (`mytism.ini`)

Die Bereitstellung der Deploy-Seite erfolgt zentral in der `mytism.ini`. Fehlt die Sektion `[DeploySite]`, greift das System auf die Basiswerte der `jetty.xml` zurück. Der Parameter `useCauldron` steuert das Architekturmodell (`0` für isolierte Deploy-Instanz, `1` für Bündelung mit APIs).

Der Parameter `requireAuthentication` regelt den zwingenden Web-Login. Bei `1` muss sich ein Benutzer authentifizieren und berechtigt sein. Bei `0` ist die Seite öffentlich erreichbar, jedoch blendet MyTISM den Reiter „Dokumentation“ aus Sicherheitsgründen vollständig aus.

Wird das systeminterne `deploy`-Verzeichnis auf dem Dateisystem lediglich als symbolischer Link (Symlink) angelegt, blockiert der Basis-Webserver Jetty den Zugriff aus Sicherheitsgründen und liefert einen 500er-Fehler zurück.

11.3.2. Rate Limiting und Quotas

Um den Server vor Überlastung zu schützen, greifen weitreichende Netzwerklimits. Die Parameter `backlog` bzw. `tlsBacklog` definieren das harte Limit der ServerSocket-Warteschlange (Standard: 10), um abgewiesene Verbindungen schnellstmöglich zu verwerfen.

Das System wartet maximal 210.000 Millisekunden (gesteuert durch `maxWaitForAuth`) auf die Anmeldung eines Clients, bevor die Verbindung getrennt wird.

Zusätzlich greift eine zweistufige IP-Quota: Der Parameter `softMaxUnauthedBCHPerIP` fungiert als weiche Grenze (Standard: 50 unauthentifizierte Verbindungen derselben IP). Ab diesem Wert wird jede weitere Anfrage um den Faktor `delayFactorUnauthedBCH` (Standard: 200ms) verzögert. Der Parameter `hardMaxUnauthedBCHPerIP` blockiert bei 150 parallelen Verbindungen derselben IP sofort alle weiteren Versuche.

```
[DeploySite]
host=localhost
port=18080
--tlsHost=
--tlsPort=
requireAuthentication=0
--reauthEveryXDays=90
```

11.3.3. Nginx, Proxy-Routing & HTTPS

In Produktionsumgebungen wird Cauldron hinter einem Reverse-Proxy (Nginx) betrieben. Der Parameter `host` in der `mytism.ini` muss dann zwingend auf `localhost` gesetzt werden, damit die Applikation nicht nach außen offensteht.

Im Multi-Node-Clusterbetrieb werden weder Session-Daten noch der Applikations-Cache zwischen den Knoten synchronisiert. Daher muss im `upstream`-Block der Nginx-Konfiguration zwingend `ip_hash` (Sticky Sessions) aktiviert werden, um einen Split-Brain-Zustand und den Verlust der Authentifizierung zu vermeiden.

Statische Frontend-Dateien sollten direkt über Nginx ausgeliefert werden. Für WebSockets müssen in der Konfiguration Upgrade-Header zugelassen werden.

```
http {
   upstream cauldron {
      ip_hash; # Zwingend zur Vermeidung von Session-Verlusten
      server 127.0.0.1:8081 fail_timeout=120s;
      server 127.0.0.1:8082 fail_timeout=120s;
   }

   server {
      location /deploy/sessions {
         proxy_pass http://cauldron;
         proxy_http_version 1.1;
         proxy_set_header Upgrade $http_upgrade;
         proxy_set_header Connection "Upgrade";
      }

      location /bookmarks {
         proxy_pass http://cauldron;
```

```
      }

      # Auslieferung statischer Dateien (SPA) über Nginx
      location / {
         root /.m/react;
         try_files $uri /index.html;
      }
   }
}
```

Ein Reverse-Proxy muss zwingend den HTTP-Header `X-Forwarded-For` mitsenden. Andernfalls erkennt das System jeden Web-Zugriff fälschlicherweise als lokalen Aufruf und hebelt weitreichende Sicherheitsrichtlinien aus.

Für eine verschlüsselte HTTPS-Verbindung müssen Administratoren in der Backend-Definition (`cauldron.conf`) Zertifikatsdaten hinterlegen.

```
connectors {
   http 8081, host: 'localhost'
   https 8444, keyStore:'./tlscert.p12', storeType:'PKCS12',
keyStorePassword:'nix', keyManagerPassword:'nix'
}
```

11.3.4. JWT-Tokens für Lesezeichen generieren

Die spezielle Bookmark-API dient als dedizierte Schnittstelle, um Lesezeichen dynamisch an Drittanwendungen auszuliefern. Für die Validierung zustandsloser Tokens (JWT) muss in der `mytism.ini` ein Schlüsselverzeichnis hinterlegt werden.

```
[Api]
keyDir=/.<PROJ_DIR>/api-keys/
```

Diese Zugriffstokens werden administrativ über das Kommandozeilen-Tool generiert und kapseln die interne Gruppen-ID als Claim.

```
(export PROJ_HOME="/.kis" && java -cp "${PROJ_HOME}/deploy/*"
```

```
de.ipcon.db.api.bookmarks.BookmarkCLI create-token <groupTid>)
```

11.4. Architektur, Backend & Deep-Dive

Dieser Abschnitt richtet sich an Entwickler, die eigene Schnittstellen im System implementieren. Da Cauldron tief im Datenbankmanager (DBMan) integriert ist, sind Startup-Race-Conditions ausgeschlossen. Logik-Skripte werden erst beim Erstaufruf kompiliert und im Arbeitsspeicher gecacht (In-Memory Hot-Reloading). Groovlets müssen dabei nicht alle Basisklassen manuell importieren, da das System automatisch Default-Includes wie sämtliche Schema-Packages bereitstellt. Externe Bibliotheken (JAR-Dateien) können jedoch nicht dynamisch zur Laufzeit nachgeladen werden, was zwingend einen vollständigen Neustart des MyTISM-Hauptservers erfordert.

11.4.1. API-Konfiguration und Hot-Reloading (`cauldron.conf`)

Die Konfiguration projektspezifischer REST-APIs erfolgt über die Datei `cauldron.conf`. Sobald die Datei existiert, werden programmierte Änderungen zur Laufzeit überwacht und sofort wirksam.

Der `application`-Block repräsentiert einen isolierten Context. Ein kritischer Parameter für interaktive Frontends ist der `sessionStore`: Da Cauldron bei Änderungen einen internen Neustart durchführt, gehen Benutzersitzungen verloren, sofern sie nicht über den `sessionStore` gesichert werden.

```
application {
   path '/example'
   resources '/zfs/home/patrick/Projects/helloworld'
   sessionStore '/zfs/home/patrick/Projects/helloworld/sessions'
   routing {
      get '/hello', forward: '/hello.groovy'
      get '/patient/@id', forward: '/patient.groovy?id=@id'
   }
}
```

Beim Speichern der `cauldron.conf` startet Cauldron sofort neu. Laufende Anfragen brechen hart ab, weshalb manuelle Reloads niemals zur Stoßzeit erfolgen dürfen. Wird die Datei in einem neuen Projekt erstmalig angelegt, greift das Hot-Reloading noch nicht; der Hauptserver muss einmal neu gestartet werden.

11.4.2. Routing und Endpunkte

Das API-Routing folgt dem CRUDL-Paradigma. Verzeichnisse müssen englische Substantive im Plural sein (z. B. `/patients`). URLs müssen der Konvention `/<Ressourcenname-Plural-Englisch>/<id|action|nix>` folgen. Jede HTTP-Aktion muss als separates Groovlet vorliegen: `create.groovy`, `read.groovy`, `update.groovy`, `list.groovy` und `delete.groovy`. Die Methode GET dient ausschließlich dem Abfragen von Ressourcen und darf niemals sensible Daten in der URL transportieren.

Das Routing wird von oben nach unten abgearbeitet. Neben HTTP-Methoden (`get`, `post`) existiert das Verb `ws` (WebSockets) und das universelle Verb `all`. Dynamische URL-Bestandteile werden durch Platzhalter (`@`) deklariert; reguläre Ausdrücke durch einen Doppelpunkt (`:`).

Das Verb `ignore:` weist Cauldron an, einen passenden Request stillschweigend abzubrechen. Spezifische HTTP-Statuscodes werden über `error:` gesendet (es existiert kein Mechanismus für Custom-Fehlerseiten).

Eigenprogrammierte Filter müssen zwingend das Java-Interface `javax.servlet.Filter` implementieren.

```
filter '/upload', script: '/MyTISMAuthFilter.groovy',
requireAnyOfGroups: ['Admins', 'RG_Solstice_Login']
```

Zudem lässt sich ein Validierungs-Closure für Login-Pflichten erzwingen.

```
// Redirect users without an active session to the login page
get "/**", forward: "login.groovy", validate: { return
session?.getAttribute('user') == null }
```

Um die interne URL-Falle bezüglich `templates` und `groovlets` zu unterbinden (Default-Handling), muss am Ende des Routings zwingend ein globaler Catchall-Filter (`all ':/.*', error: 'NOT_IMPLEMENTED'`) konfiguriert werden.

11.4.3. Das 4-Phasen-Modell und Serialisierung

Die Entwicklung eines Servlets folgt einem 4-Phasen-Modell:

1. Manuelle CSRF-Validierung und JWT-Token-Prüfung.
2. Validierung der gesendeten Payloads (Einkommende Datenströme müssen hartcodiert

als UTF-8 ausgelesen werden).

3. Geschäftslogik der Datenbank-Interaktion.
4. Serialisierung der Datenmodelle (klassischerweise über Helferklassen wie `JsonModel` und `JsonResponseModel`).

```
import de.ipcon.cauldron.mvc.JsonModel
import de.ipcon.cauldron.mvc.JsonResponseModel

Map data = JsonModel.fromRequest(request)
JsonResponseModel jsonResponse = new JsonResponseModel(response)

if (!data.name) {
   jsonResponse.setBadRequestError('"Name" must be specified.')
   jsonResponse.send()
   return
}
```

Datentransformation mittels CBOFormat

Das `CBOFormat` kann in Phase 4 direkt zur Formatierung der ausgelieferten JSON-Payloads eingesetzt werden, um die Implementierung von Formatierungslogik auf Client-Seite zu vermeiden.

```
def patientenListe = tx.queryBO("Patient p where p.Status =
'AKTIV'")
def payload = patientenListe.collect { patient ->
   [
      id: patient.id,
      display: patient.describe("Nachname ', ' Vorname ' (' Id
')'"),
      abrechnung: patient.describe(".{KontoSaldo{,#0.00}} 'EUR'")
   ]
}
```

Besonderheiten bei der Array-Serialisierung

Fehlt bei einem Array-Index der Wert, rendert das System diesen als `null` (ohne Anführungszeichen). Um zwingend ein String-Escaping zu erzwingen, kann der Parameter

`useComponentsGUIText` auf `true` gesetzt werden.

Für „Early Exits“ kann ein Wert zurückgegeben werden, der zwar ignoriert wird, aber die Skript-Ausführung bei Fehlern sofort abbricht.

11.4.4. Injizierte Variablen und Sicherheitsaspekte

Cauldron injiziert die Variablen `params` und `headers`. Es existieren keine hartcodierten Obergrenzen für HTTP-Header. Über `application.properties` erhält der Entwickler Zugriff auf eine `ConcurrentHashMap` für applikationsweites In-Memory-Caching. Für die Datenausgabe werden `html` und `json` injiziert; Variablen werden beim HTML-Rendering nicht automatisch escaped (Gefahr von Cross-Site Scripting).

Um Code-Duplizierung zu vermeiden, kann eine mit `@groovy.transform.BaseScript(Superklasse)` annotierte abstrakte Superklasse verwendet werden. Die Annotation muss auf einer Package-Deklaration oder einem expliziten Import-Statement platziert werden.

11.4.5. Datenbank-Verbindungen (`dbm`) und Limits

Die Variable `dbm` repräsentiert die zentrale Datenbank-Verbindung. Das hart konfigurierte Limit für gleichzeitige Datenbankverbindungen liegt bei 128 (Timeout: 10.000 Millisekunden). Jede aktive Verbindung wird nach maximal 300.000 Millisekunden zerstört.

Daten müssen zwingend über `queryBO()` abgefragt werden (direkte String-Konkatenation für SQL-Befehle ist verboten).

`dbm` ist ein nicht-cachender Loader; Entitäten mit derselben ID erzeugen unterschiedliche Java-Instanzen. Für komplexes Caching muss der `dbm` über einen `CachingBOLoader` gekapselt werden.

```
// Use CachingBOLoader safely
def kontakt = new
de.ipcon.db.core.CachingBOLoader(dbm).queryBO("Kontakt k WHERE
NOT Ldel ORDER BY Crea DESC LIMIT 1").find()
```

Das manuelle Starten eines `DBManLocalBOLoader` im Web-Umfeld hebelt das Transaktionsmanagement aus und ist strikt verboten.

11.4.6. Transaktionsmanagement (`tx`)

Sollen Objekte in eine neue Transaktion überführt werden, muss `tx.frapBOFromCache()`

genutzt und das Ergebnis zwingend neu zugewiesen werden. Das direkte Überführen eines Objekts via `include()` in eine zweite Transaktion wirft eine harte Exception.

In Cauldron wirft die Änderung eines Business Objects ohne vorheriges `tx.includeBO()` eine Exception (im Gegensatz zum Solstice-Client, der eine Persistierung lautlos ignoriert).

```
import de.ipcon.db.DBMan.SaveResult

SaveResult savePatient = dbm.save('Saving new patient', {
Transaction tx ->
   Patient patient = tx.includeNew(Patient.class, Rufname:
params.rufname)
})

if (!savePatient.wasSuccessful()) {
   log.error(savePatient.saveOrFunctionException)
   response.setStatus(500)
   return params
}
```

Kollidierende Schreibzugriffe durch langlaufende Operationen führen zu `LockNotGrantedException` und extremem Log-Spam, was eine aktive Log-Rotation erfordert.

Neu erstellte und direkt wieder gelöschte BOs können als leere Hüllen in der Datenbank persistieren und verbrauchen sofort persistente Datenbank-IDs.

11.4.7. Lokales Setup & SPA-Integration

Für Entwicklungszwecke (z. B. mit React) kann Cauldron über `export CAULDRON_ENVIRONMENT="development"` konfiguriert werden. Bei der Einbindung über Symlinks müssen deklarierte Pfade ggf. relativ umgeschrieben werden. Ein lokaler Nginx muss so konfiguriert werden, dass er bei unbekannten Pfaden auf die `index.html` des Frontends zurückfällt.

```
server {
   root /.m/react;

   location / {
```

```
        try_files $uri /index.html;
    }
}
```

11.4.8. Legacy-Architektur & Testing

In XML-Definitionen sind Attribute wie `displayProperty` (ersetzen durch `property`), `displayFormat` (`format`) und `leftClass` (`leftEntity`) veraltet. `displayClass` ist gestrichen. Standardwerte werden über `initDefaults()` definiert, nicht mehr über `default="…"`. Das Attribut `showFtsPopup` bei Lesezeichen ist obsolet und im Backend ist die parameterlose Variante der Methode `getDataSource()` weggefallen. Für Tests wird heute der `TestBOLoader` mit nativem `queryInterceptor` anstelle der ausgemusterten Klasse `MockTransaction` verwendet.

Chapter 12. System-Updates & Serverstart

12.1. Grundlagen & Konzepte

System-Updates und strukturelle Datenbankänderungen erfordern in verteilten Systemen ein sicheres und automatisiertes Vorgehen. Wenn sich Datenmodelle ändern – beispielsweise durch die Aufteilung eines Textfeldes in separate Felder – bergen manuelle Eingriffe in die Datenbank ein hohes Risiko für Datenverlust und Inkonsistenzen. MyTISM nutzt stattdessen versionierte Update-Skripte. Diese Skripte definieren vorab exakt, wie Bestandsdaten verlustfrei in neue Strukturen überführt werden.

Dieses Vorgehen ist essenziell für dezentrale Offline-Knoten (wie Laptops für Außeneinsätze), die nicht permanent mit dem Hauptserver verbunden sind. Manuelle Datenbankbefehle aus der Zentrale würden diese Geräte nicht erreichen. Die Update-Skripte werden hingegen in das Software-Update integriert. Sobald ein Offline-Knoten eine Netzwerkverbindung herstellt, gleicht er sich mit dem Hauptserver ab und führt anstehende Skripte vollautomatisch und strikt chronologisch aus. Dies eliminiert manuelle Eingriffe durch Techniker vor Ort und erhöht die Ausfallsicherheit des gesamten Netzwerks drastisch.

12.2. Benutzeroberfläche & Bedienung

Die Einrichtung neuer, synchronisierender Serverknoten beginnt zwingend in der grafischen Benutzeroberfläche des Hauptservers. Im Formular des neu angelegten Business-Nodes exportieren Administratoren die Konfigurationsdatei `.init-syncaccount`. Diese Datei enthält die Login-Daten inklusive eines automatisch generierten Passworts für das Synchronisationskonto. Sie wird in das Projektverzeichnis des neuen Knotens gelegt, damit sich dieser beim ersten Start automatisch an der Zentrale authentifizieren kann.

Für jeden physischen Knotenpunkt muss zwingend ein eigener Synchronisations-Benutzer verwendet werden. Beim Export der `.init-syncaccount`-Datei generiert das System jedes Mal ein neues Passwort und überschreibt das vorherige in der Datenbank. Wird die Datei für denselben Benutzer erneut exportiert, verlieren alle zuvor eingerichteten Knoten, die diesen Benutzer verwenden, sofort ihren Zugang.

12.3. Verwaltung & Server-Konfiguration

Der Serverstart von MyTISM lässt sich administrativ über das Dateisystem und spezifische Server-Parameter steuern.

12.3.1. Server Dateistruktur Übersicht

Der MyTISM Server lebt für gewöhnlich vollständig in seinem eigenen Projektverzeichnis. Übersicht über die Standarddatei- und Ordnerstruktur.
Je nach Projekt und Server können weitere Dateien notwendig und vorhanden sein.

```
/.project/
 |-- backups/ ①
 |-- cauldron/ ②
 |-- cauldron.conf ③
 |-- .checked-[firstnodestart, initialdata, integrity, metadata,
sync] ④
 |-- .init-[keygen, streamcopy] ④
 |-- .mytism_session ⑤
 |-- deploy/ ⑥
 |-- dumps/ ⑦
 |-- filesRoot/ ⑧
 |-- lib/ ⑨
 |-- logs/
 |   |-- daily.log ⑩
 |   |-- maintenance.log ⑪
 |-- log4j.conf ⑫
 |-- mytism ⑬
 |-- mytism.ini ⑭
 |-- run/ ⑮
 |-- script-log.conf ⑯
```

```
|-- secrets.p12 ⑰
```

1. Der Zielordner für die täglichen Datenbank-Backups
2. (Optional) Projektspezifische Web-Endpunkte des Cauldron-Servers
3. (Optional) Konfigurationsdatei des Cauldron-Servers
4. siehe Trigger-Dateien im Dateisystem
5. Konfiguration der JVM, siehe Server-Konfiguration
6. Enthält den eigentlichen Code für Server, Client, Dokumentation, …
7. (Optional) Zielordner für Heap- und Stackdumps des Servers
8. Enthält die BLOBs des Servers, d.h. die importierten Dateien und Bilder
9. Enthält interne, zum Serverstart benötigte Skripte
10. Serverlog, wird täglich rotiert
11. Log-Datei der täglichen Wartung, wie Erzeugung des Backups
12. Konfigurationsdatei des Server-Loggings, siehe Server-Konfiguration
13. Ausführbares Skript, für Serverstart und weitere Tools
14. Server Konfiguration, siehe Server-Konfiguration
15. Automatisch verwaltete Laufzeitumgebung des Servers
16. (Optional) Abweichende Logging-Konfiguration für *run-bs* Skripte
17. Intern vom Server verwendete Zertifikate

12.3.2. Trigger-Dateien im Dateisystem

Das Vorhandensein bestimmter leerer Systemdateien im Projektverzeichnis signalisiert dem Server, dass ressourcenintensive Initialisierungen bereits in der Vergangenheit erfolgreich durchgeführt wurden. Administratoren können diese Dateien löschen, um beim nächsten Neustart eine zwingende Prüfung oder Neuanlage zu erzwingen.

- `.checked-firstnodestart`: Zeigt an, ob die Datenbank auf den initialen Start eines synchronisierenden Knotens vorbereitet wurde. Fehlt diese Datei, räumt ein Backend-Mechanismus die Transaktions-Logs so weit auf, dass der Sync-Prozess seinen Ansatzpunkt findet.
- `.checked-metadata`: Vergleicht das XML-Schema mit den SQL-Definitionen. Das Löschen erzwingt ein tiefgreifendes Update aller Tabellendefinitionen.
- `.checked-integrity`: Startet die kritische Integritätsprüfung (z. B. auf fehlerhafte Fremdschlüssel). Das Löschen erzwingt eine Reparatur, was bei großen Datenbanken

extrem lange dauern kann.

- `.checked-initialdata`: Überprüft die Existenz generierter Strukturelemente (Auto-Formulare, Basis-Benutzer, Standard-Gruppen) und erzeugt diese bei Fehlen neu. Echte Initialdaten aus `.initialdata.xml`-Dateien werden jedoch erst neu geladen, wenn zusätzlich `.checked-metadata` entfernt wird.
- `.checked-sync`: Enthält die IDs der zuletzt übertragenen Business Transactions (BTs) und existiert ausschließlich auf synchronisierenden Servern.
- `.init-keygen`: Fehlt diese Datei, leert der synchronisierende Server beim Start vollautomatisch seine lokale `bi`-Tabelle in der Datenbank.
- `.init-streamcopy`: Fehlt diese Datei, gleicht der synchronisierende Server alle fehlenden Datei-Anhänge (BLOBs) vollständig mit dem Hauptserver ab.

Wird eine lokale synchronisierende Instanz aus einem Backup neu aufgesetzt, müssen zwingend alle `.checked-*`-Dateien sowie `.init-keygen` und `.init-streamcopy` gelöscht werden. Nach dem erfolgreichen Neustart ist ein letzter manueller Schritt erforderlich: Ein Administrator muss eine beliebige Datenänderung im System vornehmen und speichern. Dieser Speichervorgang erzeugt einen initialen lokalen Log-Eintrag, ohne den der Synchronisations-Mechanismus seinen Startpunkt nicht findet und unweigerlich mit der Meldung „Server has no logs from us yet, so we can't check." abbricht.

12.3.3. Boot-Optionen & Systemverhalten (`mytism.ini`)

Über den Abschnitt `[DBMan]` in der Datei `mytism.ini` lässt sich das Startverhalten feinjustieren. Langwierige System-Checks können durch die Zuweisung des Wertes `1` explizit unterdrückt werden.

```
noMetaDataCheck=1
noInitialDataCheck=1
noIntegrityCheck=1
noIntegrityDoubleIdChecks=1
noIntegrityBLOBChecks=1
```

Das vollständige Abschalten des Integritätschecks mittels `noIntegrityCheck=1` ist hochgradig gefährlich. Dieser Check führt bei Schema-Updates essenzielle automatische Reparaturen durch, wie das Aktualisieren von Basis-Objekt-Typen (BOT) nach dem Umzug von Entitäten im Datenmodell. Ohne diese Korrektur drohen unwiderruflich

> fehlerhafte Transaktionshistorien.

Sollen sehr große Tabellen aus Performancegründen vom Doppel-ID-Check ausgeschlossen werden, können diese kommagetrennt definiert werden.

```
integrityCheckEntitiesToExcludeFromNToMAndDoubleIdCheck=(Patient,
Arzt, Messwert)
```

Um eine lokale synchronisierende Instanz für Testzwecke zu betreiben, muss der Server explizit als nicht-autoritativ konfiguriert werden.

```
[DBMan]
authoritative=0
url=jdbc:postgresql://localhost:5432/<dbname>-syncnode
filesRoot=/.<projektkuerzel>-syncnode/filesRoot

[Sync]
url=socket://localhost:4242?compress=zlib9
user=Admin
pass=<passwort>
```

12.3.4. Logging & Fehlersuche

Der Datenbankmanager (`DBMan`) loggt den Startvorgang und die Skript-Ausführungen in der Datei `daily.log` im Verzeichnis `/.<projektkuerzel>/logs/`. Die Log-Level werden zentral in der Datei `log4j.conf` konfiguriert.

```
# Monitoring aller ausgeführten OQL- und SQL-Queries für tiefe
Einblicke
log4j.logger.de.ipcon.MyTISMQueryMonitoring=DEBUG

# Empfohlener Logger zur Analyse von Abhängigkeitsproblemen
log4j.logger.de.ipcon.db.core.UnresolvedReferencesResolver=DEBUG
```

12.4. Architektur, Backend & Deep-Dive

Die Zuverlässigkeit von System-Updates hängt von der technischen Orchestrierung der Skripte und der präzisen Abfolge von Datenbankmigrationen ab.

12.4.1. Infrastruktur: Business Nodes (BN) & Business Units (BU)

Die verteilte Systemarchitektur basiert auf physischen Rechner-Knotenpunkten (Business Nodes) und organisatorischen Einheiten (Business Units). Ist beim Start in der `mytism.ini` und der `.init-syncaccount` keine `nodeID` hinterlegt, sucht die Methode `DBMan.assureServerBN` automatisch nach einem Node, der zum Hostnamen passt, oder legt diesen an und persistiert die ID in der `mytism.ini`.

Business Units (BU) sind abstrakte Entitäten zur Generierung autarker Nummernkreise für eindeutige Identifikationsmerkmale (z. B. Rechnungsnummern). Ist der Maximalwert eines Bereichs überschritten, liefert die Abfrage `null` zurück, was von der aufrufenden Logik zwingend abgefangen werden muss.

Business Units dürfen zwingend nur serverseitig in der Methode `verifyOnServer()` zugewiesen werden, um ID-Kollisionen zu verhindern.

```
setBelegNr(BU.nextValueAsString(getClass().getName() +
'.BelegNr', tx, nodeNumber))
```

Bei der Implementierung muss der Synchronisations-Status berücksichtigt werden. Über `tx.isSyncMode()` muss geprüft werden, ob die Transaktion gerade durch einen Netzwerk-Sync empfangen wird. Ist dies der Fall, dürfen im Code absolut keine Seiteneffekte (wie weitere Datenänderungen oder Exceptions) ausgelöst werden, da dies die Integrität der Netzwerksynchronisation zerstört.

12.4.2. Der Coredata-Generator als Bootstrapper

Der Coredata-Generator (`de.ipcon.schema.generators.CoreData`) orchestriert den strukturellen Systemstart im Backend. Er befüllt die globale Tabelle der Basis-Objekt-Typen (BOT), legt den initialen Administrator an und generiert Basis-Formulare, Sammelordner, Standard-Druckziele sowie Standard-Reports.

Zusätzlich importiert der Generator vorgebaute Strukturelemente aus dem Projektpfad `de/ipcon/db/core/resources` anhand der kompilierten Liste `ResourceIndex`. Der Import erfordert spezifische Dateiendungen (`.bkm`, `.frm`, `.tpl`, `.rpt`, `.bst`) sowie den exakten Elementtyp als XML-Root-Knoten.

Werden vorgebaute XML-Ressourcen nachträglich angepasst, reicht ein einfacher Serverneustart nicht aus. Damit die Änderungen wirksam werden, muss zwingend die Trigger-Datei `.checked-initialdata` gelöscht werden.

12.4.3. Lebenszyklus von Update-Skripten (Stage 1 & Stage 2)

Die Klasse `DBMan` koordiniert die chronologische Ausführung von System-Updates. Update-Skripte unterliegen einer strikten Namenskonvention (`YYYY-MM-DD-XXX_Migration.orm` oder `.sql`), um die Ausführungsreihenfolge technisch zu erzwingen. Fehlt diese Konvention, ist die Ausführung undefiniert und führt zu Abhängigkeitsfehlern. Erfolgreich ausgeführte Skripte werden in der internen Datenbanktabelle `bupd` protokolliert.

Bei einer leeren, neuen Datenbank überspringt das Backend historische Migrationen und loggt diese lediglich als erledigt. Auf bestehenden Systemen greift ein zweistufiger Prozess:

Stage 1:
Das System führt ausschließlich SQL-Skripte aus. Stößt es auf das erste `.orm`-Skript, pausiert die Skriptausführung. Der Schema-Generator gleicht nun die Datenbank an das aktuelle XML-Modell an. Ungefährliche Aktionen (z. B. neue Spalten) werden sofort ausgeführt; zerstörerische Aktionen (Spaltenlöschungen, Typänderungen, neue Indizes) werden blockiert und aufgeschoben.

Stage 2:
Das System führt die `.orm`-Skripte aus, welche die volle Persistenzschicht und interne Validierungslogik der Objekte nutzen. Wirft ein Skript eine `ServerRestartRequiredException` oder folgt auf ein `.orm`-Skript ein weiteres SQL-Skript, erzwingt das Backend einen kontrollierten Neustart und setzt die Abarbeitung danach nahtlos fort. Unbehandelte Exceptions führen zum harten Abbruch des Serverstarts.

Erst nach erfolgreichem Abschluss aller Skripte führt das System die aufgeschobenen, zerstörerischen Schemaanpassungen aus.

Wartungsmodus:
Ein versehentlicher Zugriff von Anwendern auf inkonsistente Datenzustände während eines Updates ist architektonisch ausgeschlossen. Der Server öffnet seine Netzwerk-Ports erst am Ende des gesamten Startvorgangs. Für den laufenden Betrieb signalisiert das Event `onServerLocked` den Clients, wenn der Server aktiv gesperrt wurde.

12.4.4. API: UpdateHandlerTools & Skripting

Bei schemaverändernde Migrationen hilft die Klasse `de.ipcon.db.update.UpdateHandlerTools`.

```
import de.ipcon.db.update.UpdateHandlerTools
```

```
// 1. Safety backup of the data via plain SQL
stmt.executeUpdate("SELECT * INTO medikament_backup FROM
medikament")

// 2. Recursively drop the column 'wirkstoff', including all
inherited tables
UpdateHandlerTools.dropColumn('medikament', 'wirkstoff', stmt)
```

Bei der Umbenennung ganzer Entitäten muss die spezialisierte Methode `renameEntity` genutzt werden.

```
// Safely rename an entire entity including its BOT metadata
UpdateHandlerTools.renameEntity('tbl_patient_alt',
'tbl_patient_neu', '@BOPACK@', 'PatientAlt', 'PatientNeu', DBMan,
stmt)
```

Die Methode `renameEntity` ändert keine Vorkommnisse des alten Namens in dynamischen Groovy-Skripten (Benachrichtigungen, Alarme, BOMasken). Diese müssen manuell via Volltextsuche gefunden und korrigiert werden.

Das System stellt implizite Standard-Imports zur Verfügung und erlaubt über das injizierte `stmt`-Objekt die Ausführung nativer SQL-Befehle.

```
import java.sql.ResultSet

// Native read access via statement object
ResultSet rs = stmt.executeQuery("SELECT id, wirkstoff FROM
medikament_backup")

// Clean up the database natively after ORM migration
stmt.executeUpdate("DROP TABLE medikament_backup")
```

Vor komplexen Migrationen müssen programmatische Existenz-Prüfungen genutzt werden.

```
// Compact existence checks in standard ORM scripts
if (checkTableExists('tbl_patient') && checkColumnExists(table:
```

```
'tbl_patient', column: 'blutgruppe')) {
    log.info('Executing migration...')
}
```

Werden diese Prüfungen in `@CompileStatic`-annotierten Groovy-Methoden verwendet, dürfen zwingend keine benannten Parameter übergeben werden. Die Prüfungen müssen statisch über die Klasse aufgerufen und das Statement-Objekt muss mitgegeben werden.

12.4.5. Systemprüfungen & Reparatur-Logik

Der Integrity-Check repariert beim Serverstart verwaiste Relationen und korrupte Transaktionshistorien, beispielsweise wenn sich Entitätstypen im Schema geändert haben oder weggefallen sind. Reparatur-Transaktionen erscheinen im Log legitimerweise als gelöscht (`Ldel`). Sie werden regulär über das Netzwerk synchronisiert, bleiben in GUI-Lesezeichen jedoch absichtlich unsichtbar.

Bricht der stark parallelisierte Integrity-Check bei sehr großen Datenbanken ab, liegt dies an zu strengen Betriebssystem-Limits („Too many open files“) oder aufgebrauchtem Shared Memory. Administratoren müssen diese Limits sowie die PostgreSQL-Konfiguration entsprechend erhöhen.

Backend-Entwickler können Systemprüfungen aus einem Update-Skript heraus für den nächsten Neustart erzwingen.

```
// Programmatically trigger system checks for the next restart
DBMan.triggerInitialDataCheck()
DBMan.triggerIntegrityCheck()
DBMan.triggerMetaDataCheck()
```

12.4.6. Edge Cases & Architektonische Gefahren

Kein automatischer Rollback: MyTISM bietet keinen automatischen Rollback-Mechanismus für fehlgeschlagene `.orm`-Skripte der Stage 2. Wirft ein Skript einen Fehler, wird der Serverstart abgebrochen. Zur Wiederherstellung muss zwingend ein physisches Datenbank-Backup eingespielt und alle `.checked-*`-Triggerdateien müssen gelöscht werden. Bei zerstörenden Skript-Operationen müssen Entwickler manuelle Backup-Tabellen innerhalb des Skripts anlegen.

Absturzgefahr auf Sync-Nodes: ORM-Skripte werden ausschließlich auf dem Hauptserver ausgeführt. Greift ein chronologisch direkt nachfolgendes SQL-Skript auf Objekte zu, die das ORM-Skript gerade erst erschaffen hat, stürzen synchronisierende Instanzen beim Start ab. Solche SQL-Skripte dürfen auf Sync-Nodes erst nach vollständigem Abschluss der Netzwerksynchronisation manuell via `psql` abgesetzt werden.

Zerstörerische Schemaänderungen (Race-Condition): Soll ein SQL-Skript eine alte Spalte umbenennen oder sichern, darf datumstechnisch niemals ein `.orm`-Skript davor einsortiert sein. Das ORM-Skript triggert den Schema-Generator; fehlt die Spalte im neuen XML-Schema, löscht der Generator sie rigoros aus der Datenbank, bevor das SQL-Skript die Daten retten kann.

Fehlende Datenbank-Indizes (Performance-Kollaps): Wird in einem SQL-Skript eine neue Tabelle angelegt, Daten migriert und im selben Skript komplex lesend darauf zugegriffen, bricht die Performance massiv ein. Neue Datenbank-Indizes werden architektonisch bedingt erst nach einem erneuten Serverneustart am Ende des Update-Zyklus generiert.

Die SQL-INSERT-Falle (Schleichende Datenkorruption): In reinen SQL-Update-Skripten darf niemals das Konstrukt `INSERT INTO … SELECT * FROM` verwendet werden. Da die physische Spaltenreihenfolge in der Datenbank aufgrund von Schema-Erweiterungen oft von der logischen Definition abweicht, drohen Datenkorruptionen. Spaltennamen müssen immer explizit angegeben werden.

Werden Tabellen manuell via SQL verschoben und der Integrity-Check ist deaktiviert, korrumpiert die Transaktionshistorie. Metadaten müssen in diesem Fall zwingend manuell repariert werden.

```
-- Repair transaction history metadata after manual schema moves
UPDATE produktNew SET bot = $botNewEntity where bot =
$botOldEntity;
UPDATE bp SET botid = $botNewEntity WHERE boid IN (SELECT id FROM
$oldEntity);
UPDATE bt SET bot = $botNewEntity WHERE bot = $botOldEntity;
```

Chapter 13. Deployment & Client-Launch (Dawn)

13.1. Grundlagen & Konzepte

Dawn ist ein in-house entwickelter Installer und Runtime-Launcher für den MyTISM-Client (Solstice) sowie dessen technische Infrastruktur. Die Anwendung ersetzt fehleranfällige Verteilungsmechanismen wie „Java Web Start“ und optimiert den Bereitstellungszyklus durch die Nutzung nativer HTTP/HTTPS-Downloads.

Ein zentraler administrativer Vorteil ist, dass für die initiale Bereitstellung und für fortlaufende Updates keinerlei lokale Administrator-Rechte auf den Zielsystemen erforderlich sind. Anstelle eines manuellen Client-Rollouts durch die IT-Abteilung rufen Anwender die zentrale Deploy-Seite im Intranet auf. Der bereitgestellte Installer (ca. 3 MB) lässt sich direkt ausführen. Die Ziel-URL des Servers ist im Dateinamen kodiert, wodurch Dawn den zuständigen Server beim Start automatisiert detektiert. Ein Whitelisting der Binärdatei durch den Microsoft Defender verhindert fälschliche Heuristik-Blockaden auf Windows-Systemen.

Dawn unterstützt aktuelle Windows-Umgebungen, 64-Bit-Linux-Distributionen und ARM-basierte Single-Board-Computer. Unter Windows wird eine 32-Bit-Stub-Executable ausgeliefert, die eine 64-Bit-Architektur selbstständig erkennt und die native 64-Bit-Java-Laufzeitumgebung nachlädt. Für macOS muss weiterhin noch Java Web Start verwendet werden.

Architektonisch isoliert Dawn das Java Runtime Environment (JRE) vollständig vom restlichen Betriebssystem. Die spezifische Laufzeitumgebung wird gekapselt für die MyTISM-Instanz im Hintergrund ausgeführt. Dies eliminiert systemweite JRE-Abhängigkeiten und Wechselwirkungen mit anderen lokalen Anwendungen. Die Versionspflege erfolgt serverseitig; Sicherheits-Patches werden beim Client-Start vollautomatisch appliziert.

Die Integrität des Launch-Prozesses ist kryptografisch abgesichert. Sämtliche ausführbaren Dawn-Komponenten und die mitgelieferte JRE verfügen über digitale Signaturen und Zeitstempel (signiert mit demselben Entwicklerzertifikat). Dawn validiert bei der Initialisierung seine eigene Integrität. Wird auf dem Server eine neuere, gültig signierte Version detektiert, führt das System ein automatisches Selbstupdate im Hintergrund durch.

13.2. Benutzeroberfläche & Bedienung

Die Deploy-Seite ist als primärer Berührungspunkt bewusst einfach gehalten. Power-User

müssen in Multi-Server-Umgebungen einen Edge Case beachten: Da die lokal installierte Dawn-Version durch den zuletzt besuchten Server aktualisiert wird, kann es beim Wechsel zwischen Projekten zu Versionskonflikten kommen. Tritt eine Inkompatibilität auf, wird dieser Fehler beim initialen Start des Solstice-Clients als Meldefenster ausgegeben.

13.3. Verwaltung & Server-Konfiguration

Die Bereitstellung und Konfiguration der Deploy-Seite wird administrativ gesteuert.

13.3.1. Steuerung der Deploy-Seite (`mytism.ini`)

Die Konfiguration der Download-Seite erfolgt in der Sektion `[DeploySite]` der `mytism.ini`. Fehlt die Sektion, greift das System auf die internen Einstellungen der `jetty.xml` zurück.

```
# Basic setup for deploy site with authentication
[DeploySite]
host=localhost
port=8080
tlsHost=
tlsPort=8443
useCauldron=1
requireAuthentication=1
reauthEveryXDays=90
```

Der Parameter `requireAuthentication` regelt den Login-Zwang. Steht der Wert auf `1`, müssen sich Benutzer mit dem Solstice-Passwort legitimieren und zwingend Mitglied der Gruppen „Admins“, „RG_Solstice_Login“ oder „RG_Deploy“ sein. Bei `0` ist die Seite öffentlich zugänglich; der Reiter „Dokumentation“ wird aus Sicherheitsgründen ausgeblendet, es sei denn, der Aufruf erfolgt über „localhost“.

Der Parameter `reauthEveryXDays` definiert die Gültigkeit des Dawn-Tokens (Standard: 90 Tage) für den Zugang zur Deploy-Seite. Dies betrifft nicht den Login im Solstice-Client; eine gecachte Applikation kann offline problemlos gestartet werden.

Wenn das Verzeichnis `deploy` auf dem Server als Symlink angelegt wird, blockiert Jetty den Zugriff. Der Client meldet einen HTTP 500 Fehler („Could not get resource "/" via ServletContext“). Es muss zwingend ein physisches Verzeichnis genutzt werden.

Der Parameter `useCauldron=1` weist das System an, die interne Webserver-Instanz für die

Deploy-Seite mitzunutzen. Ein Wert von `0` startet isoliert einen separaten Mini-Server.

Beim Einsatz eines Reverse-Proxys (wie Nginx) muss die originale Client-IP an den Webserver durchgereicht werden. Andernfalls wertet Cauldron jeden externen Zugriff fälschlicherweise als privilegierten lokalen Aufruf (localhost).

Im Nginx muss der HTTP-Header zwingend gesetzt werden:

```
# Forward original client IP to backend server
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
```

13.3.2. Benutzerdefiniertes Client-Logging

Für ein individuelles Client-Logging muss im Home-Verzeichnis des Benutzers ein Konfigurationsordner angelegt werden, der dem Java-Paketnamen entspricht (z. B. `~/.com.oashi.m`). Die dort abgelegte Datei `client-log.conf` wird von Dawn erkannt und anstelle der Standardkonfiguration angewendet.

13.4. System-Administration (Server-Ebene)

Dieser Abschnitt richtet sich an Administratoren, die das Dateimanagement auf den Zielsystemen sowie Massen-Rollouts verantworten.

13.4.1. Verzeichnisstruktur und Speicher-Management

Dawn splittet Verzeichnisse zur Speicheroptimierung. Die Executable und Desktop-Verknüpfungen liegen im Profilverzeichnis (Windows: `%USERPROFILE%\appdata\roaming\`, Linux: `~/.local/share/dawn/`). Der Cache (JRE und JAR-Module) wird ausschließlich lokal abgelegt (Windows: `%USERPROFILE%\appdata\local\dawn\`, Linux: `~/.cache/dawn/`), unterteilt in die Ordner `app/` und `jre/`.

Bei automatisierten Massen-Rollouts in Unternehmensnetzwerken greift keine Bandbreiten-Kontrolle. Starten hunderte Clients gleichzeitig, wird die Netzwerkinfrastruktur ungedrosselt mit der vollen Last der Downloads konfrontiert.

13.4.2. Kommandozeilen-Tools (CLI)

Dawn lässt sich über die Kommandozeile fernsteuern, was Headless-Aufrufe via SSH

ermöglicht. Der Befehl `dawn help` listet alle Optionen auf.

Die Dawn-CLI dokumentiert derzeit keine verlässlichen Exit-Codes. Fehlgeschlagene Ausführungen (z. B. Netzwerkprobleme) können von CI/CD-Pipelines nicht ohne Weiteres über einen Exit-Code ausgewertet werden.

Der Befehl `remove -headless` entfernt die Applikation aus dem Cache. Verschwindet der letzte referenzierende Hardlink auf eine JAR-Datei, wird diese endgültig gelöscht. Für eine restlose Bereinigung müssen die Verzeichnisse unter `AppData` (respektive `~/.cache` und `~/.local`) manuell gelöscht werden. Zur Speicheroptimierung diene die Parameter `-run -keep` und `-log-keep` (Aufbewahrungsfrist für Run-Verzeichnisse, Client-Logs und interne Dawn-Logs in Sekunden).

Dawn ist installationsfrei und hinterlässt daher weder versteckte Registry-Keys, noch geplante Windows-Aufgaben oder anderweitige systemweite Spuren.

13.4.3. Troubleshooting & Edge Cases

Startet die Applikation nicht und die Kommandozeile bleibt stumm, muss Dawn mit dem Parameter `-sync-run` gestartet werden, um Fehler direkt in der Konsole auszugeben.

Bricht die Applikation unter Windows sofort ab, hat oft ein Drittanbieter-Antivirenprogramm die Java-Umgebung (z. B. `jit.dll`) zerstört. Das Verzeichnis `jre` im lokalen Cache muss gelöscht und die Applikation neu gestartet werden.

Blockiert der Windows Defender die `dawn.exe` trotz Whitelisting, muss ein Signatur-Update über die Kommandozeile erzwungen werden (`MpCmdRun.exe -removedefinitions -dynamicsignatures` und `-SignatureUpdate`), bevor Dawn neu heruntergeladen werden kann.

Wird das Desktop-Icon nicht erstellt, müssen in Konflikt stehende `*.ds` -Dateien im `app`-Verzeichnis gelöscht oder der Parameter `-force -create-icon` verwendet werden.

Unter Ubuntu wird das Desktop-Icon nach einer Headless-Installation oft als nicht vertrauenswürdig blockiert. Lösung: `gio set /home/<user>/Desktop/oashi-dawn-de_acme@project_acme_com.desktop "metadata::trusted" true`. Zudem muss die heruntergeladene Dawn-Datei zwingend manuell

ausführbar gemacht werden (`chmod +x`).

Dawn übernimmt keine Windows-System-Proxy-Einstellungen und bietet keine Proxy-Kommandozeilenparameter. Es müssen direkte HTTP/HTTPS-Routen zum Server existieren. Zudem übernimmt Dawn keine Root-Zertifikate aus dem Windows-Speicher; abgelaufene TLS-Zertifikate auf dem Server blockieren den Download (Workaround: `-ignore-tls-errors`).

13.5. Architektur, Backend & Deep-Dive

Die Backend-Architektur des Caching und der Speicherdeduplizierung ist hochgradig optimiert.

13.5.1. Pile-Verzeichnis und Hash-Deduplizierung

Alle Binärdateien werden isoliert im Pile-Verzeichnis (`~/.cache/dawn/pile/`) abgelegt. Der Dateiname generiert sich aus dem Originalnamen und Teilsegmenten des SHA256-Hashes, um Dateinamenskollisionen über Server hinweg auszuschließen. Werden identische Standard-Bibliotheken in unterschiedlichen Projekten mit projektspezifischen Keys signiert, verändert sich der Hash; Dawn behandelt diese als unterschiedliche Artefakte.

13.5.2. Hardlinks vs. Softlinks

Zur Speicherschonung erzeugt Dawn im Laufzeitverzeichnis keine physischen Kopien, sondern Inode-Hardlinks, die auf die Originaldateien im Pile referenzieren. Hardlinks wurden gewählt, da Softlinks unter Windows Administrator-Rechte erfordern. Dies dedupliziert den Speicherverbrauch lokal innerhalb eines Benutzerprofils.

Auf Multi-User-Systemen (z. B. Terminalserver) existiert kein globaler Shared Cache. Caches liegen exakt so oft auf der Festplatte, wie es aktive Benutzerprofile gibt. Manifest-Dateien (`*.ds`) und die `log4j.conf` bilden eine Ausnahme und werden physisch kopiert.

Datei-Explorer summieren die Größen von Hardlinks oft fehlerhaft auf, was die Illusion einer Platzverschwendung erzeugt.

Seit Juni 2025 ist die Kompatibilität mit dem Andrew File System (AFS) netzwerkseitig gesichert.

13.5.3. Asynchrone Garbage Collection

Die Garbage Collection läuft asynchron. Dawn startet die Java Virtual Machine (JVM) und pausiert den eigenen Aufräum-Thread kurzzeitig, um Systemressourcen während der Bootphase zu schonen. Anschließend löscht der Prozess verwaiste Binärdaten im Pile und tote Laufzeitverzeichnisse.

13.5.4. Manifest-Generierung (`solstice.dst`) und Rollbacks

Die finale DawnSpec (`solstice.ds`) existiert auf dem Server physisch nicht; ein Backend-Servlet generiert sie in Echtzeit aus der Schablone `solstice.dst`. Bei Updates berechnet das Servlet neue Hashes, die der Client erkennt. Neue Third-Party-Bibliotheken müssen händisch in die `solstice.dst` eingetragen werden.

Die korrekte serverseitige Generierung lässt sich durch direkten Abruf via `wget https://project.acme.com/deploy/solstice.ds` überprüfen.

Ein automatisierter, globaler Downgrade durch einen Netzwerkbefehl existiert nicht. Rollbacks erfolgen über den regulären Update-Pfad: Die fehlerhafte Bibliothek wird auf dem Server durch die ältere Version überschrieben.

13.5.5. Self-Healing und Binary-Updates

Durch den Abgleich von Dateigrößen und Hashes schließt die Architektur Starts mit fragmentierten Dateien aus. Abgebrochene Downloads müssen jedoch von vorne beginnen (keine HTTP Range Requests).

Erkennt Dawn eine neuere Version seiner selbst, überschreibt sich das Binary automatisch. Der aktuell initiierte Boot-Vorgang wird jedoch noch mit der alten Version abgeschlossen; die neue Logik greift ab dem nächsten Start. Ein Downgrade der Dawn-Executable ist ausgeschlossen.

Chapter 14. Systembetrieb, Troubleshooting, Entwicklungsumgebung & Testing-Architektur

14.1. Grundlagen & Konzepte

Der reibungslose Systembetrieb von MyTISM garantiert die permanente Verfügbarkeit kritischer Daten. Troubleshooting umfasst in diesem Kontext das schnelle und sichere Eingreifen bei unerwarteten Abstürzen oder Netzwerkabbrüchen, wie etwa bei asynchronen Datenlieferungen von zuvor offline betriebenen Clients. Das System bietet hierfür mächtige Reparaturwerkzeuge, um Inkonsistenzen und Datenkollisionen sicher aufzulösen. Eine strenge Qualitätssicherung ist unter dem Leitsatz „No tests? No trust.“ zentral in der Entwicklungsphilosophie verankert. Updates durchlaufen vor dem produktiven Einsatz tausende automatisierte Tests, welche operative Prozesse simulieren und die Software-Reaktionen validieren.

14.2. Lokales Setup und Entwicklungsumgebung

Die nachfolgenden Konfigurationsschritte beschreiben den Aufbau und die Verwaltung der lokalen Systemumgebung für Entwickler und Administratoren.

14.2.1. Runtime-Infrastruktur

Für die Softwareentwicklung, das Erstellen von Migrationsskripten und das System-Troubleshooting ist der Aufbau einer autarken lokalen Laufzeitumgebung zwingend erforderlich. Die fundamentale Basis besteht aus einer lokalen PostgreSQL-Datenbankinstanz sowie einem aktuellen Java Development Kit (JDK). Nach der Konfiguration der Systemvariablen auf dem Host-System wird der Applikationsquellcode aus dem zentralen Repository in das lokale Projektverzeichnis ausgecheckt.

14.2.2. Datenbank-Initialisierung

Ein frisch ausgechecktes System erfordert im ersten Schritt die Bereitstellung einer leeren Datenbankstruktur. Das zentrale Kommandozeilen-Tool von MyTISM wertet hierzu die Konfigurationsparameter im Projektverzeichnis aus und legt eine entsprechende Instanz in der PostgreSQL-Datenbank an. Die Ausführung erfolgt über das Terminal:

```
./mytism init-db
```

14.2.3. Konfigurations-Management

Die Laufzeitkonfiguration des lokalen Applikationsservers ist über drei zentrale Steuerdateien organisiert:

1. **`mytism.ini`:** Die primäre Konfigurationsdatei. In der Sektion `[DBMan]` werden Verbindungs-URLs, Ports und Zugangsdaten zur PostgreSQL-Datenbank typsicher hinterlegt.
2. **`log4j.conf`:** Steuert das globale Logging-Verhalten und die Ablaufprotokollierung des Applikationsservers.
3. **`.mytism_session`:** Eine versteckte Datei im Projektverzeichnis zur persistenten Definition benutzerspezifischer Umgebungsvariablen der Sitzung, wie Heap-Zuweisungen der JVM oder Garbage-Collection-Parameter.

14.2.4. Start-Modi des Applikationsservers

Der lokale Server unterstützt verschiedene Ausführungsmodi. Um den Server vom Terminal zu entkoppeln und als Hintergrunddienst auszuführen, wird der Parameter `start` verwendet:

```
./mytism start
```

Für das interaktive Debugging können Log-Ausgaben im Vordergrundmodus direkt in das Terminalfenster (stdout) geleitet werden:

```
./mytism server
```

Dieser Initialisierungsprozess bootet die Middleware-Komponenten, validiert das XML-Schema, registriert die autonomen Hintergrunddienste und öffnet die Ports für eingehende Client-Verbindungen.

14.2.5. Client-Bootstrapping im Entwicklungsmodus

Sobald der Server Verbindungen akzeptiert, kann der Solstice-Client gestartet werden. Das primäre Werkzeug für die lokale Entwicklung ist das Bash-Skript `start-client`. Es stellt eine Verbindung via Dawn her und bietet erweiterte Debugging-Optionen:

```
./start-client [-h|-c HOST] [USERNAME] [PASSWORD]
```

Die integrierte Hilfe dokumentiert die Parameter zur Verbindungs- und Agentensteuerung:

```
Starts the Solstice client with a debugging connection to a local
server.
Syntax: ./start-client [-h|-c HOST] [USERNAME] [PASSWORD]
options:
  -h          This help
  -c          The scheme, hostname and port of the deploy site to
connect to. Default: http://localhost:8080
  -d          Enable debugging agent on default port: 5005
  -D [port]   Enable debugging agent on the specified port
```

Für Legacy-Umgebungen steht das Skript `start-client-jar` zur Verfügung, welches den Client als isolierte Java-Anwendung außerhalb der Dawn-Infrastruktur startet. Es synchronisiert benötigte JAR-Bibliotheken via `rsync` in das Verzeichnis `run_solstice` und konfiguriert die JVM mit optimierten Memory-Parametern. Der Server-Parameter wird automatisch formatiert; die Verbindung erfolgt standardmäßig komprimiert und verschlüsselt.

Die initiale Authentifizierung an frisch aufgesetzten Instanzen ohne eingespieltes Datenbank-Backup erfolgt über das Standard-Konto „Admin“. Damit weitere Benutzer auf die Solstice-Schnittstelle zugreifen können, müssen diese administrativ der Systemgruppe `RG_Solstice_Login` zugewiesen werden.

14.3. System-Administration (Server-Ebene)

Start-Skripte und Konfigurationen sind übersichtlich im Projektverzeichnis gebündelt, um den Server-Betrieb trivial zu halten.

14.3.1. Konfigurationsdateien und Logging

Der Basisbetrieb wird primär durch die Dateien `mytism.ini`, `log4j.conf` und `.mytism_session` gesteuert. Das übergeordnete Server-Logging wird zentral in der `log4j.conf` konfiguriert (z. B. `log4j.rootLogger=INFO, stdout, daily`). Für ein granular gesteuertes, lokales Client-Logging wird eine separate Datei namens `client-log.conf` im Projektverzeichnis angepasst.

Die Datei `.mytism_session` darf nach einem Absturz auf keinen Fall gelöscht oder geleert werden. Es handelt sich hierbei nicht um einen flüchtigen Session-Speicher, sondern um eine elementare, statische Konfigurationsdatei. Sie definiert maschinenspezifische Umgebungsvariablen wie RAM-Zuweisungen oder die Begrenzung von Backup-Threads persistent.

14.3.2. Betriebssystem-Limits und Zeitumstellung

Brechen automatisierte Tests oder Serverprozesse mit der Exception „too many open files“ ab, limitieren die File-Descriptoren des Linux-Betriebssystems den Durchlauf. Administratoren müssen als `root` den Eintrag `fs.file-max = 2097152` in die Datei `/etc/sysctl.conf` einfügen und mittels `sysctl -p` dauerhaft anwenden.

Bei der Taktung von Hintergrunddiensten via Cron-Job ist die Zeitumstellung (Daylight Saving Time) zwingend zu berücksichtigen. Ereignisse, die in die Zeitstunden von Daylight-Saving-Korrekturen fallen, werden architekturseitig stillschweigend nicht oder mehrfach ausgeführt. Dies muss bei der Planung systemkritischer Tasks einkalkuliert werden.

14.4. Datenbank-Troubleshooting & Reparatur

Ein elementarer Bestandteil des Systembetriebs ist die Wiederherstellung von synchronisierenden Server-Knoten nach Abstürzen oder Netzwerkabbrüchen (Split-Brain).

14.4.1. Cache-Bereinigung und Sync-Wiederherstellung

Vor dem Start eines synchronisierenden Servers nach dem Einspielen eines physischen Backups müssen administrative Bereinigungen im Dateisystem vorgenommen werden. Das rigorose Löschen aller generierten `.checked-*`-Dateien zwingt den Server zu einer vollständigen Neuprüfung. Die Datei `.checked-sync` enthält die IDs der zuletzt übertragenen Business Transactions und muss zwingend entfernt werden. Ebenso ist die Datei `.init-keygen` zu löschen, damit das System beim Neustart die lokale `bi`-Tabelle leert und den Sync-Ansatz neu initialisiert. Das Entfernen von `.init-streamcopy` erzwingt einen sauberen Abgleich der BLOBs im `filesRoot`-Verzeichnis, woraufhin der Hauptserver fehlende BLOBs automatisch überträgt.

14.4.2. Behebung von Doppel-ID-Kollisionen

Kommt es durch fehlerhafte Sync-Vorgänge zu ID-Kollisionen in polymorphen Relationen (z. B. zwei Objekte mit exakt derselben ID), ist ein strenger Reparatur-Ablauf auf dem autoritativen Server zwingend einzuhalten:

```
# 1. Den Server kontrolliert stoppen und ein initiales Backup
anlegen
./mytism stop_mytism
./mytism backup

# 2. Das DoubleIdFixer-Tool aufrufen, um das SQL-Reparatur-Skript
zu generieren
./mytism run de.ipcon.db.tools.DoubleIdFixer .oashi --repairAll
fix_double_ids.sql

# 3. Alle Cache- und Check-Dateien im Verzeichnis löschen
rm .checked-*

# 4. Das generierte SQL-Skript direkt auf der Datenbank ausführen
psql -U postgres DATENBANKNAME < fix_double_ids.sql

# 5. Die Basis-Instanz-Tabelle (bi) zwingend komplett leeren
psql -U postgres DATENBANKNAME -c "delete from bi;"

# 6. Server kurz durchstarten für die internen Aufräumarbeiten
./mytism start_mytism

# 7. Danach erneut stoppen und ein frisches Backup für die
ausstehenden Sync-Knoten ziehen
```

14.4.3. Boot-Parameter zur Fehlerumgehung

Um den Serverstart zu beschleunigen oder Check-Routinen notfallmäßig zu umgehen, bietet die Datei `mytism.ini` spezifische Flags im Abschnitt `[DBMan]`. Zudem lassen sich extrem große Tabellen gezielt vom zeitaufwendigen Doppel-ID-Check ausnehmen.

```
[DBMan]
# Suppress standard checks for faster startup during
troubleshooting
noMetaDataCheck=1
noInitialDataCheck=1
noIntegrityCheck=1
```

```
# Exclude specific, extremely large tables from N:M and Double-ID
checks
integrityCheckEntitiesToExcludeFromNToMAndDoubleIdCheck=(BP, BT,
Messwert)
```

Der Systemstart beinhaltet einen automatischen Check auf verwaiste Dateien (Orphaned BLOBs) im `filesRoot`-Verzeichnis. Verzögert dieser Prozess den Start extrem, lässt sich der Check über das Flag `noIntegrityBLOBChecks=1` in der `mytism.ini` temporär deaktivieren.

14.5. Architektur, Backend & Testing

MyTISM-Tests sind architektonisch oftmals eine zielgerichtete Mischung aus Unit- und Integrations-Tests, welche durch den `TestBOLoader` orchestriert werden.

14.5.1. Test-Isolation und Graphenstabilität

Modul-Tests erben zumeist direkt von der Klasse `GroovyTestCase`. Damit diese verlässliche Ergebnisse liefern, müssen sie absolut unabhängig voneinander und in völliger Isolation laufen. Um Cache-Lecks und Wechselwirkungen zwischen aufeinanderfolgenden Tests zu verhindern, müssen am Ende jedes Tests zwingend die Transaktion geschlossen und statische Caches geleert werden.

```
protected void tearDown() {
   tx.close()
   TBO.flushCaches()
}
```

Schlagen Unit-Tests fehl, weil Strukturen in der In-Memory-Datenbank nicht gefunden werden, fehlt in der `setUp()`-Methode oftmals der Initialisierungsaufruf. Damit `byTid` -Methoden den Cacheloader korrekt aktualisieren können, muss zwingend `tx.injectInitialDataFor(TBO.class)` ausgeführt werden.

Ein weiteres Kernkonzept für die Graphenstabilität im Arbeitsspeicher ist das Frapping („Festzurren“). Für alle Business Objects innerhalb einer Transaktion gilt die Regel, dass bei identischer Datenbank-ID auch die Java-Instanz im Speicher zwingend identisch sein muss. Die Methode `tx.frapBOFromCache(bo)` mutiert das Objekt nicht in-place, sondern garantiert durch Rückgabe der gefrappten Instanz die strikte Eindeutigkeit des Objektgraphen.

```
// Korrektes Frappen und Identitätsprüfung im Test
def krankenakte = tx.includeBO(tx.getBO(5))
krankenakte.geschlecht = tx.frapBOFromCache(tx.getBO(12)) // z.
B. referenziertes Geschlecht-BO
assert tx.frapBOFromCache(tx.getBO(12)).is(tx.getBO(12)) //
Objekt-Identität sicherstellen
```

14.5.2. Lokalisierung (L10n) in automatisierten Tests

In automatisierten Tests dürfen Fehlermeldungen niemals gegen hartcodierte Strings geprüft werden. Das System muss L10n-Schlüssel dynamisch auflösen, da Tests auf Servern mit abweichender OS-Locale andernfalls unweigerlich fehlschlagen. Für den exakten Abgleich in Assertions muss zwingend die Aufruf-Variante der Übersetzungs-Methode mit drei Parametern genutzt werden.

```
def errMsg = L10n.msg(Krankenakte.L10N_KEY_MISMATCHED_PARAMS,
[paramA, paramB] as Object[], [Krankenakte] as Object)
```

14.5.3. Edge Cases und Logging-Gefahren für Entwickler

Das Logging innerhalb des Codes birgt spezifische Gefahren, die zwingend umschifft werden müssen.

In automatisierten Tests darf keinesfalls `Log4jHelper.createDefaultLogger()` aufgerufen werden. Dieser Eingriff überschreibt die `WARN`-Voreinstellung aus dem Skript `compile-and-run-tests.groovy` und überflutet das Terminal massiv mit `INFO`-Logs.

Mit diesem Kapitel schließen wir den administrativen und architektonischen Betrieb von MyTISM (Teil 4) ab. Sie haben nun das Rüstzeug, um das System in hochkritischen Szenarien sicher zu warten und zu reparieren. Im folgenden **TEIL 5: Referenzen** finden Entwickler und Power-User das vollständige **OQL-Referenzhandbuch** für komplexe Datenbankabfragen sowie die verbindlichen **Best Practices und Coding Conventions** für die Entwicklung im MyTISM-Ökosystem.

Chapter 15. OQL-Referenzhandbuch

Die Object Query Language (OQL) ist das technologische Herzstück des Datenzugriffs innerhalb des MyTISM-Ökosystems. Dieses Referenzhandbuch dient als tiefgehende Dokumentation für Entwickler und Administratoren und behandelt syntaktische Grundlagen, Objekt-Relationen, Performance-Analyse und MEX-Erweiterungen.

Diese Referenz wird kontinuierlich um weitere Details und Praxisbeispiele ergänzt. Bei spezifischen Fragen steht das Support-Team unter https://mytism.com/#contact zur Verfügung.

15.1. Teil 1: OQL-Grundlagen

Zielgruppe: Entwickler und fortgeschrittene Administratoren

Dieser Teil legt das gemeinsame Fundament für die programmatische Abfrage. Hier lernen Sie die grundlegende Syntax, Datentypen und die Navigation durch komplexe Objekt-Relationen kennen.

15.1.1. Einführung in die Object Query Language (OQL)

Was ist OQL?

OQL ist eine objektorientierte Sprache zur direkten Abfrage von Datenstrukturen aus der Datenbank. Im Gegensatz zu klassischem SQL abstrahiert OQL die relationale PostgreSQL-Datenbankschicht vollständig und wandelt objektorientierte Aufrufe automatisch in relationale SQL-Queries um. Dies reduziert Boilerplate-Code, verhindert Logikfehler bei Tabellen-Joins und schirmt Entwickler von der Persistenzschicht ab. Spezielle `OQLTools` können OQL-Queries zur Analyse jederzeit in die ausführbare SQL-Version umwandeln.

Abweichung vom Standard

Die verwendete OQL-Version ist auf MyTISM maßgeschneidert und implementiert den offiziellen OQL-Standard bewusst nicht exakt, sondern ergänzt ihn um proprietäre Optimierungen.

Anwendungsbereiche im Framework

OQL begegnet Anwendern in folgenden Kontexten:

Filterung in Lesezeichen für Endanwender Die Nutzung von OQL in GUI-Lesezeichen und Suchfeldern wird im **Kapitel „Lesezeichen, Datenabfragen (OQL) & Volltextsuche“** behandelt.

Nutzung im Programmcode (Backend) und Masken Im Backend setzen Entwickler OQL-Queries ab, um Business Objects (BOs) speicherschonend in den RAM zu laden. Dies geschieht architektonisch sauber über das `Transaction`-Objekt oder den `BOLoader`. OQL bildet zudem die technologische Basis für `OQLBOMasken` zur extrem performanten Vorfilterung großer Datenmengen.

15.1.2. Grundlegende Syntax und Aufbau einer Query

Die vier Kernelemente einer Abfrage

Queries bestehen aus vier essenziellen Elementen:

1. Aggregatsfunktion
Sie definiert den Rückgabewert. Struktur: `[SELECT] <value> FROM [ONLY] <Entity> a`. Verfügbare Methoden: * `count(*)`: Gibt die exakte Anzahl der Ergebnisse als Liste zurück. * `max(Position)` / `min(Position)`: Liefern höchste oder kleinste Werte. * `avg(Position)`: Berechnet den Mittelwert (kann Kommastellen enthalten). * `sum(Position)`: Berechnet die Gesamtsumme.

Elementanzahl vorab ermitteln

Wird `count(*)` testweise vor eine Query eingefügt, lässt sich performant die zu erwartende Elementanzahl ermitteln, ohne Objekte in den Arbeitsspeicher zu laden.

2. Ausdrücke / Bedingungen (WHERE-Klausel)
Ein boolescher Ausdruck, eingeleitet durch `WHERE`, der die Ergebnisse einschränkt. Bedingungen lassen sich durch `AND`, `OR`, `NOT` und Klammern `()` verknüpfen (`AND` bindet stärker als `OR`). Standard-Operatoren umfassen Identität (`=`), Ungleich (`!=`), Größenvergleiche (`<`, `>`, `<=`, `>=`) und Textvergleiche (`like`, `ilike`).

Best Practice für gelöschte Objekte

Direkt nach dem `WHERE` sollte bei regulären Abfragen `NOT Ldel` folgen, um gelöschte Objekte aus der Ergebnismenge auszuschließen.

Beispiele für WHERE-Klauseln

```
Patient p WHERE NOT Ldel
Patient p WHERE Stationaer = false
Patient p WHERE NOT Ldel AND NOT Stationaer
Patient p WHERE NOT Ldel AND (Stationaer = null OR Stationaer)
```

3. Antwortbegrenzung (LIMIT)

Limitiert die Ergebnismenge aus Performancegründen über das Schlüsselwort `LIMIT`.

Beispiel für ein Limit

```
Medikament m where not Ldel ORDER BY Id LIMIT 1
```

4. Sortierung (ORDER BY)
Definiert die lexikalische oder numerische Reihenfolge der Ergebnisse.

Beispiele für Sortierungen

```
Medikament m where not Ldel ORDER BY Id LIMIT 1
order by a.crea limit 100
```

SELECT-Anweisungen und Projektionen

Explizites vs. Implizites SELECT
Das explizite `SELECT` fragt komplette Objekte unter Angabe eines Variablenbezeichners ab.

Explizites SELECT

```
SELECT p FROM Patient p
```

Das implizite `SELECT` (Kurzform) lässt `SELECT <value> FROM` weg und ist die effizienteste und am häufigsten verwendete Form in MyTISM.

Implizites SELECT

```
Patient p
```

Rückgabe einzelner Attribute
Um Bandbreite zu sparen, kann ein einzelnes Attribut projiziert werden. Das Schlüsselwort `DISTINCT` entfernt Duplikate performant direkt auf Datenbankebene.

Beispiele für die Rückgabe einzelner Attribute

```
Hersteller FROM Medikament m WHERE (count(*) WITHIN Wirkstoffe w)
> 10
Bezeichnung FROM Medikament m WHERE (count(*) WITHIN Wirkstoffe
w) > 10
SELECT DISTINCT Medikament.PZN FROM Verordnung v WHERE NOT Ldel
```

```
AND Dosis > 0
```

Rückgabe mehrerer Attribute als Array
Wenn alle abzufragenden Attribute denselben Datentyp aufweisen, können diese in eckigen Klammern gruppiert und als Array zurückgegeben werden.

Erfolgreiches Array-Select (gleicher Basistyp)

```
SELECT [Medikament.Id, Dosis] FROM Verordnung v WHERE NOT Ldel
AND Dosis > 0
```

Fehlerhaftes Array-Select (unterschiedliche Typen)

Eine kombinierte Abfrage wie `SELECT [Medikament.PZN, Dosis] FROM Verordnung v` wirft einen Fehler, da `PZN` ein Text (`String`) und `Dosis` eine Kommazahl (`BigDecimal`) ist.

Das ONLY-Schlüsselwort

Das `ONLY`-Schlüsselwort stellt sicher, dass bei einer Abfrage strikt nur Objekte des explizit angegebenen Entitätstyps zurückgeliefert werden. Ohne `ONLY` greift der Polymorphismus und bezieht abgeleitete Unterklassen mit ein.

Performance-Vorteil durch ONLY

Sucht man nur das Basis-Objekt, ist `ONLY` wesentlich effizienter als eine nachträgliche Typüberprüfung im Code, da PostgreSQL signifikant weniger Sub-Tabellen verknüpfen muss.

Beispiele zur Unterdrückung von Polymorphismus

```
ONLY Mitarbeiter m
Mitarbeiter m where exists(ONLY Arzt a where not Ldel and
Mitarbeiter = m)
```

15.1.3. Datentypen, Operatoren und Filterbedingungen

Werttypen und ihre Formatierung

Text / Strings
Alphanumerische Zeichen und Texte müssen zwingend von einfachen (`'`) oder doppelten (`"`) Anführungszeichen umschlossen werden (z. B. `"Aspirin"`, `"01234"`).

Sicherheitsrisiko durch String-Konkatenation

Dynamische String-Konkatenationen im Backend-Code öffnen Tür und Tor für SQL-Injections. Es ist zwingend eine sichere Parameterübergabe via Platzhalter zu verwenden.

Numerische Werte
Ganzzahlen und Gleitkommazahlen werden ohne Anführungszeichen angegeben. Die Formatierung erfolgt nach der internationalen U.S.-Notation (Punkt als Dezimaltrenner, keine Tausendertrennzeichen). Beispiel: `123.24` oder `-1234.928482`.

Da Text-Attribute nicht via SQL-Cast direkt in Zahlen umgewandelt werden können, wird die PostgreSQL-Funktion `TO_NUMBER` im Pass-through genutzt.

Beispiel für TO_NUMBER

```
Patient p WHERE TO_NUMBER(Zimmernummer, '9999') > 200
```

Hstore (Key-Value Dictionaries)
OQL unterstützt Hstore-Spalten, die als flexible Schlüssel-Wert-Paare fungieren (z. B. zur Speicherung lokalisierter Texte) und in Abfragen wie virtuelle Objekte behandelt werden.

Boolesche Werte
Erlaubte Werte sind `true`, `false` und `null` (ohne Anführungszeichen). Explizite Vergleiche wie `Abgerechnet = true` können im Code elegant auf `Abgerechnet` verkürzt werden. `Abgerechnet = false` wird als `not Abgerechnet` abgekürzt.

Randfall bei Null-Werten

Die Abfrage auf den fehlenden Wert `Abgerechnet = null` lässt sich logisch nicht abkürzen und muss vollständig ausgeschrieben werden.

Operatoren für den einfachen Vergleich

- Identität (`=`) und Ungleich (`!=`): Prüfen auf absolute Übereinstimmung oder Ungleichheit.
- Größenvergleiche (`<`, `>`, `<=`, `>=`): Prüfen numerische Werte oder Datumsvergleiche.
- `between ... and ...`: Prüft, ob ein Attribut innerhalb eines inklusiven Bereichs liegt. `Id between 2 and 7` ist semantisch identisch zu `Id >= 2 and Id <= 7`.

Beispiele für Größen- und Bereichsvergleiche

```
Herzfrequenz <= 120
Aufnahmedatum > '2025-12-01'
```

```
Verordnung v where not Ldel and Tagesdosis between 100 and 500
```

Rechnen mit Datumswerten

Die Funktion `now()` liefert das aktuelle Systemdatum inklusive Uhrzeit. Zeitintervalle können addiert werden.

Syntax-Besonderheit bei Intervallen

Datumswerte und relative Intervalle müssen logisch immer addiert werden (`+`). Eine direkte Subtraktion (z. B. `now() - '7d'`) funktioniert nicht. Für die Vergangenheit muss ein negativer String addiert werden (z. B. `now() + '-7d'`).

Rückgabewert (Projektion) modifizieren

```
(Crea + '1d') from BO a where Id = 1234 limit 1 // + 1 Tag
(Crea + '-1d') from BO a where Id = 1234 limit 1 // - 1 Tag
```

Text- und Mustersuche

- `like` / `ilike`: `like` beachtet strikt die Groß- und Kleinschreibung. `ilike` ignoriert diese und sollte aus Performancegründen bevorzugt werden. Das Prozentzeichen `%` dient als Wildcard.
- Reguläre Ausdrücke (`matches` und `imatches`): Vergleichen Texte mit Regex-Mustern.

Warnhinweise für Regex im GUI-Umfeld

Regex-Vergleiche sind auf PostgreSQL-Ebene langsam und sollten die Ausnahme bleiben. Geschweifte Klammern `{}` (z.B. Längenangaben) werden in GUI-Lesezeichen nicht korrekt unterstützt. Schließende eckige Klammern `]` müssen mit einem Backslash escaped werden (`\\]`).

Logische Verknüpfungen

Bedingungen lassen sich durch `AND`, `OR` und `NOT` zusammensetzen. `NOT` wird vor `AND` und `AND` vor `OR` evaluiert. Runde Klammern `()` heben die Standard-Priorisierung auf.

Multi-Attribut-Vergleiche

- `ANY OF`: Erzeugt eine ODER-Verknüpfung der Werte oder Attribute.
- `ALL OF`: Erzeugt eine UND-Verknüpfung.
- `ANY DEFINED OF`: Erzeugt eine ODER-Verknüpfung ergänzt um einen Check auf `!=`

`null`.

Beispiele für Kurzschreibweisen

```
ANY OF (Nachname, Vorname) = 'Müller'
ANY OF (Nachname, Vorname) = ANY OF ('Müller', 'Schmidt')
```

Der IN LIST (...) Operator
Prüft, ob ein einzelnes Attribut in einer kommaseparierten Liste von Werten enthalten ist. Der Operator ist performanter als `ANY OF()` und unterstützt Text-Operatoren wie `ilike`.

Verwendung im GUI-Lesezeichen

```
[ p.Alter in list(18, 30, 50, 65)
[ m.Fachbereich ilike list('%chirurg%', '%kardio%')
```

Fallstricke bei Parameterlisten

Der Ausdruck `Id in list($1)` erlaubt als Parameter `$1` immer nur eine einzige, konkrete ID und keine Liste von Objekten. Die korrekte Übergabe an Prepared Statements wird in Teil 3 beschrieben.

Umgang mit Null-Werten

Der Wert `null` bedeutet, dass ein Wert noch nicht befüllt wurde. Soll dieser leere Zustand als gültiges Ergebnis zugelassen werden, muss er explizit eingeschlossen werden (z. B. `Stationaer = null OR NOT Stationaer`).

Vermeidung von Fallstricken bei ANY OF und null-Werten

Die Kurzschreibweise `any of (Stationaer, NOT Stationaer)` überspringt kommentarlos Entitäten, bei denen `Stationaer = null` gesetzt ist. Wird `null` als Element einer Parameter-Liste an `ANY OF` oder `IN` übergeben, wird dieser auf Datenbankebene oft stillschweigend ignoriert.

OQL-Methoden

- `lower(<value>)` / `upper(<value>)`: Wandelt Text temporär in Klein- oder Großbuchstaben um. Bei einfachen Vergleichen sollte stattdessen der performantere `ilike`-Operator bevorzugt werden.

15.1.4. Objekt-Navigation und Relationen

Navigation über einfache Attributketten (n:1)

Der Zugriff auf verknüpfte Eltern-Objekte erfolgt standardmäßig über die Punkt-Notation. Ist die Entität im Kontext eindeutig, kann der Variablenname optional weggelassen werden (`Patient p WHERE NOT Ldel` ist äquivalent zu `Patient p WHERE NOT p.Ldel`).

Implizite Existenzprüfung

Die Bedingung `Notfallkontakt.Id < 200` impliziert auf Datenbankebene automatisch `Notfallkontakt != null`. Eine logische Negation wie `NOT Notfallkontakt.Id < 200` ignoriert kompromisslos alle Objekte ohne Notfallkontakt. Die Existenz sollte daher explizit via `Notfallkontakt = null` geprüft werden.

Logische Negation bei leeren Relationen

Die Negation `NOT Notfallkontakt.Id < 200` ignoriert zusätzlich alle Objekte ohne Notfallkontakt. Der technische Grund hierfür ist, dass der gesamte Ausdruck für leere Relationen auf der PostgreSQL-Ebene gar nicht erst ausgewertet werden kann. Soll die Existenz oder das Fehlen des Notfallkontakts explizit geprüft werden, geschieht dies am sichersten via `Notfallkontakt = null`.

Optionale Relationen abfragen mit dem Fragezeichen-Operator (?)

Sollen Objekte berücksichtigt werden, bei denen die Relation leer (`null`) ist, wird das Fragezeichen `?` vor dem Punkt verwendet. Der Ausdruck `Notfallkontakt?.Ldel = null` entspricht implizit `(Notfallkontakt = null or Notfallkontakt.Ldel = null)`.

Typ-Casting in Attributketten

Spezifische Attribute von Subentitäten lassen sich bei Basisklassen nur durch explizite Casts adressieren. Die zwingende Syntax lautet: `Variablenname.(Typ)Relation.Attribut`.

Syntax-Vorgabe bei Casts

Die Angabe des Variablennamens (z. B. `p`) ist beim Cast zwingend erforderlich. Der Ausdruck `(Typ)` muss immer direkt auf einen Punkt folgen.

Erfolgreiche Cast-Aufrufe

```
p.(Patient)Notfallkontakt.(Patient)Notfallkontakt.Stationaer
```

Semantischer Unterschied beim Vergleich mit Null

`a.(Patient)AbstraktePerson.Geburtsdatum = null` liefert nur Objekte mit verlinktem Patienten, deren Datum `null` ist. `a.(Patient)AbstraktePerson?.Geburtsdatum = null` liefert zusätzlich alle Objekte, deren Relation komplett leer ist oder auf einen anderen Typ verweist.

Casts in Verbindung mit der ID

Ein Cast an der vorletzten Stelle eines Navigationspfades wird ignoriert, wenn das nachfolgende Attribut die `.Id` ist. Das Snippet `a.(Patient)AbstraktePerson.Id > 0` liefert alle abstrakten Personen zurück. Als Workaround muss stattdessen auf `.Ldel` geprüft werden: `not a.(Patient)AbstraktePerson.Ldel`.

Abfrage von Many-Relationen (1:n und n:m)

Existenzprüfungen mit EXISTS(...) und NOT EXISTS(...)

`EXISTS(<Subquery>)` prüft effizient, ob in einer Menge verknüpfter Objekte mindestens ein Objekt den Filtern entspricht. Subqueries müssen zwingend in Klammern gesetzt werden.

Performance und der Verdopplungseffekt

Existenzprüfungen über implizite Inner-Joins (z. B. `Mitarbeiter m where Abrechnungen.Id > 0`) duplizieren das Basis-Objekt für jedes Kind-Element. Dies verfälscht globale `count(*)`-Queries massiv.

Das Schlüsselwort WITHIN

Das `WITHIN`-Schlüsselwort initiiert Subqueries tief innerhalb von Many-Relationen. Die Syntax verlangt das `WITHIN`-Schlüsselwort, den pluralisierten Attributnamen der Relation und einen lokalen Variablennamen.

Beispiel für die WITHIN-Syntax

```
Mitarbeiter m where not exists(within Abrechnungen b where not
Ldel)
```

Mengen-Operationen (count(*) WITHIN)

Das `count(*)`-Präfix kann mit einer `WITHIN`-Subquery kombiniert werden, um die Anzahl verknüpfter Kind-Elemente zu ermitteln und für mathematische Vergleiche zu nutzen.

Filterung auf spezifische Subentitäten innerhalb von Many-Relationen

Sollen nur bestimmte Subtypen durchsucht werden, kann die Cast-Syntax auch bei der

`WITHIN`-Definition eingesetzt werden.

Gecastete WITHIN-Subquery

```
exists(within m.(Arzt)Abrechnungen b where Personalnummer = $1)
```

Randfall bei der Klassennamen-Prüfung

Restriktionen über den Klassennamen (z. B. `where b.Bot.Name = 'Arzt'`) bergen Risiken, da Unterklassen nicht berücksichtigt werden.

Best Practice und Performance-Tipp (ONLY)

Eine effiziente Methode zur Abfrage spezifischer Subentitäten ist die Nutzung der expliziten Rückrelation in Kombination mit dem `ONLY` -Keyword. Das unbedachte Verketten von `NOT EXISTS` und `WITHIN` in Verbindung mit einem Cast verursacht massive Performanceprobleme und ist ein Anti-Pattern. Es ist stattdessen zwingend die Rückrelation zu nutzen.

Performanter Ausschluss über die Rückrelation

```
Mitarbeiter m where not exists(Arzt a where Mitarbeiter = m)
```

15.1.5. Arrays und Key-Value-Strukturen verarbeiten

Grundlegende Syntax und Typ-Casting

Arrays werden in OQL als simple Literale in eckigen Klammern und durch Kommata getrennt angegeben (z. B. `Laborwerte = [5.2, 42.0]`). OQL unterstützt dabei ein vollautomatisches Typ-Casting.

Positions- und Schlüsselzugriff

Der Zugriff auf einen Array-Wert erfolgt über seine Position, die in OQL zwingend 1-basiert ist. Zugriffe auf nicht existierende Indizes werfen keine Fehler.

Gefahr von „Off-by-one“-Fehlern

In Java und Groovy beginnen Arrays mit Index 0. In nativen OQL-Queries (z. B. im `<filter>` oder `BOLoader`) muss zwingend auf die 1-basierte Position geachtet werden (`Position = Index + 1`).

Die Index-Notation akzeptiert auch Strings als Schlüssel für Hstore-Felder.

Beispiele für direkten Zugriff und Wildcards

```
Details['blutgruppe'] = 'A+'
Details['allergien'] ilike '%penicillin%'
ANY OF (Details['blutgruppe'], Details['rhesusfaktor']) = 'A+'
```

Strikte Typisierung bei Array- und Hstore-Indizes

Der Index-Zugriff verlangt zwingend einen Integer, einen String (in Anführungszeichen) oder einen Slice. Unmaskierte Schlüsselnamen oder boolesche Werte (`true` / `false`) führen sofort zu einer `OQLSyntaxException`.

Array-Funktionen und Operatoren

Slicing: Ausschneiden von Teilmengen OQL ermöglicht das Ausschneiden von Bereichen. Die Syntax lautet `[Startposition..Endposition]` (inklusive der Grenzen, ohne Leerzeichen). Fehlt die Start- oder Endposition, beginnen bzw. enden die Slices implizit am Array-Rand.

Native SQL-Funktionen: array_length
Die Funktion `array_length([Array], [Dimension])` dient als Pass-through, um Arrays auf eine exakte Gleichheit zu prüfen und Teilmengen auszuschließen.

Prüfung auf exakte Array-Gleichheit

```
[ Laborwerte = 5.2 and Laborwerte = 42.0 and
array_length(Laborwerte, 1) = 2
```

Prüfen auf Vorhandensein: IN und ANY
* `IN`: Ist ein einzelner Wert identisch in einer Liste enthalten. * `ANY`: Erlaubt zusätzlich auch Größer- oder Kleiner-Vergleiche (`>`, `<`) auf die Elemente der übergebenen Liste.

Beispiele für IN und ANY

```
// Erfolgreiche IN-Prüfung (sofern die Variable $1 das Array [1,
2] ist)
1 in $1

// Die Ziffer 1 ist in der angegebenen Menge vorhanden
1 = ANY(1, 2, 3, 4, 5)
```

```
// Wahr, da die Ziffer 5 größer als die 1 und die 4 ist
5 > ANY(1, 4, 6)
```

Mengenprüfungen: CONTAINS, CONTAINEDBY, OVERLAPS
* `CONTAINS`: Das linke Array ist die vollständige Obermenge des rechten Arrays. * `CONTAINEDBY`: Das linke Array ist eine Teilmenge des rechten Arrays. * `OVERLAPS`: Mindestens ein Element aus einem Array kommt im anderen vor. Mögliche Doppler werden bei diesen Prüfungen vollständig ignoriert.

Beispiele für Mengenprüfungen

```
// Wahr: Die rechte Menge ist vollständig im linken Array
enthalten
[1, 2, 3] contains [1, 2]

// Falsch: Die Ziffer 3 fehlt auf der linken Seite
[1, 2] contains [1, 2, 3]

// Fehler: Eine komplett leere Menge ist als Argument nicht
erlaubt
[1, 2] contains []

// Wahr: Das linke Array ist vollständig im rechten enthalten
[1, 2] containedby [1, 2, 3]

// Wahr: Vorhandene Doppler werden bei der Prüfung ignoriert
[1, 1, 2] containedby [1, 2, 3]

// Wahr: Die Ziffer 2 ist in beiden Mengen vorhanden
[1, 2] overlaps [2, 3]

// Wahr: Die 3 ist in beiden Arrays vorhanden
[1, 2, 3] overlaps [3, 4, 5]
```

Hstore-Funktionen und virtuelle Arrays

Hstore-Felder verfügen über die virtuellen Suffixe `.keys` und `.values`, welche alle Schlüssel oder Werte als durchsuchbares Array repräsentieren.

Vereinheitlichte Mengen-Operatoren

- `HAS` / `LACKS`: Prüft, ob ein Element existiert oder fehlt.
- `HAS ALL OF` / `HAS ANY OF`: Prüfen die Schnittmengen.
- `LACKS ALL OF` / `LACKS ANY OF`: Prüfen das Fehlen von Teilmengen.

Beispiele für Existenz- und Mengenprüfungen

```
// Existenzprüfung für bestimmte Hstore-Keys
Details.keys HAS 'blutgruppe'
Details.values HAS 'A+'

// Prüfung auf komplett fehlende Schlüssel
Details.keys LACKS 'rhesusfaktor'

// Komplexe Mengenvergleiche
Details.keys HAS ALL OF ['blutgruppe', 'rhesusfaktor']
Details.values HAS ANY OF ['A+', 'Penicillin']
```

Der Parser erkennt die virtuellen Arrays bei Wildcard-Suchen (`ilike`, `matches`) automatisch, wodurch explizite Subqueries entfallen (z.B. `Details.values ilike '%Suchbegriff%'`).

Randfälle und Best Practices

Silent Failures bei NULL in Arrays

PostgreSQL verwirft `NULL`-Werte bei Operatoren wie `ANY OF` oder `IN` stillschweigend. Die GString-Erweiterung `${→ ['Bla', null]}` verliert dadurch den Null-Check. Als performanter Workaround zur Prüfung auf `NULL` im Array muss die native Funktion `array_position(Attributname, NULL) IS NOT NULL` verwendet werden.

NULL-Werte vs. Fehlende Schlüssel (Hstore)

Die Abfrage `Details['allergien'] = null` ist wahr, wenn der Schlüssel auf `null` steht ODER der Schlüssel gar nicht existiert. Um strikt auf das physische Fehlen zu prüfen, muss `Details.keys LACKS 'allergien'` verwendet werden.

15.2. Teil 2: Fortgeschrittene Praxisbeispiele & MEX-Erweiterungen

Zielgruppe: Power-User und Administratoren

Dieser Teil behandelt komplexe Anwendungsfälle, dynamische Ergebnisfindung und MEX-Präprozessor-Erweiterungen.

15.2.1. Komplexe OQL-Praxisbeispiele

Wie finde ich Duplikate?

Die Suche nach versehentlich doppelt angelegten Objekten (z. B. Medikamente) lässt sich effizient über eine `exists()`-Subquery realisieren.

Suche nach doppelten Medikamenteneinträgen

```
[ Medikament m where not Ldel and exists(Medikament b where not
b.Ldel and b != m and b.Bezeichnung = m.Bezeichnung)
```

Um das älteste Original aus der Treffermenge auszuschließen und nur die Duplikate anzuzeigen, wird der Erstellungszeitpunkt (`Crea`) in den Vergleich einbezogen.

Anzeige nur der jüngeren Dubletten

```
[ exists(Medikament b where not b.Ldel and b != m and
b.Bezeichnung = m.Bezeichnung AND b.Crea < m.Crea)
```

Filterung auf das jeweils aktuellste BO einer Many-Relation

Um eine Hauptentität basierend auf dem Status ihres aktuellsten verknüpften Elements zu filtern, wird die Aggregatsfunktion `max()` in Kombination mit `WITHIN` genutzt.

Patientenakten mit aktuellster Untersuchung im August 2025

```
[ Patientenakte a where not Ldel and
 exists(Untersuchung b where not Ldel and
        b.Patientenakte = a and
        b.Datum >= '2025-08-01' and
        b.Datum <  '2025-09-01' and
        b.Datum = (max(Datum) WITHIN a.Untersuchungen c where
not Ldel)
```

```
        )
```

15.2.2. MEX: MyTISM-Erweiterungen für OQL (Präprozessor)

MEX fungiert als vorgeschalteter Präprozessor, der fehlende ORM-Features (z. B. Unions, Subclass-Casting, Prefetching) transparent nachrüstet und komplexes natives SQL vermeidet. MEX besteht aus geschweiften Klammern `{}`, die serverseitig schrittweise ausgewertet und ersetzt werden.

Unbehandelte Klammerblöcke

Bleiben nach der Auswertung unbehandelte Klammerblöcke im Quelltext übrig, stoppt das System und gibt eine harte Fehlermeldung zur Identifizierung aus.

Zusammenfassen von Ergebnissen

Die Befehle `{Union}` und `{UnionAll}` fassen die Ergebnisse isolierter OQL-Queries in einem Roundtrip zusammen (aktuell ohne automatische Dublettenerkennung).

Beispiel für Backend-Abfrage mit UnionAll

```
SELECT a FROM de.mytism.hospital.Mitarbeiter a WHERE
a.BOTyp.Name="Arzt"
{UnionAll SELECT a FROM de.mytism.hospital.ExterneKraefte a WHERE
a.BOTyp.Name="Arzt"}
```

Ein „Tag“ (z. B. `@Stationaer`) kann einem Union-Resultat zugewiesen werden, um die Herkunft zu identifizieren. Das Tagging ermöglicht die gezielte Anwendung von Groovy-Code auf die Ergebnismenge mittels `<transform-script>`. Das Attribut `onTag` steuert, für welche Resultate das Skript aufgerufen wird (Default: `*` = alle Tags). Variablen im Skript sind `bo` (das aktuell iterierte Objekt) und `tag` (der ermittelte Union-Tag).

Performance im Transform-Skript

Da das Transform-Skript potenziell für sehr viele Objekte aufgerufen wird, darf die Laufzeit nur minimal sein. Zeitaufwändige Berechnungen oder Datenbank-Ladeoperationen sind verboten.

Komplexes Union Tagging und Transformation im XML

```
<Query type="Text">
  <template>
```

```
    only BO a where 'dummy'!='for Tag'
    {Union @Stationaer p.Aufnahmen from Patientenakte {=where}
{=constraints}}
    {Union @Ambulant p.Behandlungen from Patientenakte {=where}
{=constraints}}
  </template>

  <!-- Transform-Skript speichert den Herkunfts-Tag in einem
virtuellen Attribut 'Behandlungsart' -->
  <transform-script language="groovy" onTag="*">
    bo.Behandlungsart = tag
  </transform-script>
</Query>
```

Dynamische Listen und Filter

- `{BOTIdList of super|sub [without self] <Klassenname>}` fügt eine Liste von BOT-IDs direkt in die Query ein. Aus Performancegründen sollte stattdessen wenn möglich direkt auf den Typ gefiltert werden.
- `{IdList [projection] from <Klassenname> where <OQL-Bedingung>}` transformiert eine Subquery in eine ID-Liste.

Typ- und Interface-Prüfungen

Der Ausdruck `{WithInterface <Package>}` ermöglicht die Abfrage von Klassen, die ein bestimmtes Java-Interface implementieren, und liefert eine Liste passender BOT-IDs.

Komplexität bei GUI-Filtern

Wird `WithInterface` in Lesezeichen-Queries genutzt, muss zwingend für jeden Objekttyp eine eigene `<clause group="Typname">` angelegt werden. Neue Untertypen werden ausgeblendet, es sei denn, das Flag `excludeIfNoClauseForEntity="false"` wird an die `<filter>` -Definition angehängt.

Weitere MEX-Funktionen

- `{Fulltext matches <Suchbegriff>}` bettet die Volltextsuche ein (architektonisch veraltet/deprecated).
- `{Prefetch <Relation>}` erzwingt das direkte, unlazy Mitladen ausgewählter Many-Relationen. Komplexe Attribut-Ketten über mehrere Relationen werden derzeit noch

nicht unterstützt.

15.3. Teil 3: Backend, Architektur & Performance

Zielgruppe: Backend-Entwickler

Dieser Teil behandelt die performante Ausführung von OQL im Backend, die Graphenstabilität im RAM sowie Best Practices zur Performance-Optimierung.

15.3.1. OQL in der Programmierung (Entwickler-Sicht)

Absetzen von OQL im Backend

Die Methode `tx.queryBO()` ersetzt Platzhalter wie `$1` oder `$2` durch die übergebenen Parameter und übernimmt das Escaping automatisch. Für das Laden von BOs über Attribut-Wert-Paare ist `getBOsByAttrs` (Groovy-Alias: `getExistingBOs`) performanter, da es lokale Caches nutzt.

Sicherheit und Parameter

Gefahr durch String-Konkatenation

Die direkte String-Konkatenation in Queries ist extrem anfällig für SQL-Injections und führt zur `Little Bobby Tables warning`.

`queryBO` agiert als Prepared Statement, verhindert SQL-Injections zuverlässig und ist durch wiederverwendbare Execution-Pläne performanter.

Lösung für Index-Verschiebung bei Arrays

Wird ein Array als Parameter an `$1` übergeben, packt der Parser das Array fälschlicherweise aus (Index-Verschiebung). Das Array muss im Code erneut gekapselt werden: `[['A', 'B']].toArray()` oder elegant über die GString-Erweiterung: `${→ ['A', 'B']}`.

Zudem müssen bei `queryBO` alle Objekte serialisierbar sein; Platzhalter dürfen niemals in einfachen Anführungszeichen stehen (`'$1'`).

Die Groovy GString-Erweiterung

Die Syntax `${→ variablenname}` übergibt Queryparameter direkt als Closures innerhalb von GStrings, wodurch die manuelle Nummerierung entfällt.

Gefahr einer OutOfMemoryException

Die Methode `List queryBO(GString)` lädt alle Ergebnisse in den RAM.

> Für große Datenmengen oder Batch-Prozesse muss zwingend `Iterator query(GString)` (lazy loading) verwendet werden, um Abstürze zu verhindern.

Fatale Folgen eines frühen Castings

Methoden, die GStrings kapseln, müssen zwingend den Rückgabetyp `GString` und niemals `String` aufweisen. Eine implizite Evaluierung zu `String` zerstört die Struktur (z. B. `from $1`) und führt zu Laufzeitfehlern.

Objekt-Identität und Frapping

Das „Frapping“ garantiert, dass der Objektgraph innerhalb einer Transaktion stabil bleibt: Besitzen zwei BOs dieselbe Datenbank-ID, müssen sie exakt dieselbe Java-Instanz im Speicher sein.

Die Methode `tx.frapBOFromCache(bo)` bindet ein Objekt sicher in den Graphen ein oder gibt die bereits verankerte Instanz zurück.

Graphenstabilität und Mutationen

`frapBOFromCache()` mutiert das übergebene Objekt niemals „in-place“. Der Rückgabewert der Methode muss zwingend neu zugewiesen und weiterverwendet werden. Ein Ignorieren des Rückgabewerts führt zu fatalen Bugs bei der Persistierung.

OQL in Unit-Tests isolieren

Mit dem `TestBOLoader` können Tests im RAM ohne echte Datenbankverbindung simuliert werden.

Veraltete Methodik

Die Klasse `MockTransaction` ist fehleranfällig und darf nicht mehr verwendet werden.

Stattdessen stellt der `TestBOLoader` den `queryInterceptor` bereit, um bekannte Query-Zeichenfolgen durch vorbereitete Fixture-Objekte abzufangen. Um Cache-Lecks zu verhindern, müssen im `tearDown()` die Transaktion geschlossen, Caches geleert (`TBO.flushCaches()`) und `bol.clearQueryInterceptors()` aufgerufen werden.

Datenzugriff auf BOs in Reports

In JasperReports wird die Feld-Klammer-Syntax `$F{THIS}` für den Zugriff auf das Basis-Objekt genutzt.

Gefahr bei virtuellen Eigenschaften in Reports

Bei virtuellen Eigenschaften darf in `<textFieldExpression>` niemals die klassische Java-Getter-Syntax (`$F{THIS}.getAttribut()`) verwendet werden, da die physischen Methoden nicht existieren. Es muss zwingend der direkte Eigenschaftsname verwendet werden (`$F{THIS}.Attribut`).

OQL in Masken und Skripten

Die `OQLBOMaske` bietet das Attribut `WhereClauses`. Sie wendet die Filter direkt bei der Datenbankabfrage an und reduziert die in den RAM zu ladende Menge hochgradig effizient.

Grooql kombiniert OQL und In-Memory-BOMasken („Two-Step-Filtering“). Das Skript wird in OQL transformiert, die Datenbank liefert eine Obermenge, die dann lokal über die `fits()` -Methode nachgefiltert wird. Grooql bietet spezifische Datumsfunktionen (`.thatDay()`, `.thatMonth()`, `.addYear()`, `.subDay()`).

Code-Snippets für Datumsabfragen in Grooql-Filterskripten

```
// Abfrage auf ein exaktes Jahr (Alle Dokumente aus dem Jahr
2025)
Aufnahmedatum.year = 2025
// Alternativ mit Getter
Aufnahmedatum.getYear() = 2025

// Abfrage auf neuere Datensätze
Aufnahmedatum.year > 2025

// Kombinierte Abfrage: Patientenakte von 2025
Nachname.startsWith("Müll") && Aufnahmedatum.getYear() = 2025
```

Abgrenzung und Performance

Die `GrooqlBOMaske` erfordert einen In-Memory-Auswertungsschritt. Wenn die Filterlogik rein in OQL abbildbar ist, ist die `OQLBOMaske` aus Performancegründen zwingend vorzuziehen.

15.3.2. Performance, Analyse und Best Practices

Die Abfragen analysieren („Back to SQL“)

`EXPLAIN` und `EXPLAIN ANALYZE` liefern den berechneten Query-Plan sowie die

tatsächlichen Ausführungszeiten von PostgreSQL, basierend auf nicht zwingend aktuellen Statistiken. Diese Befehle können OQL-Abfragen direkt vorangestellt werden. Die Text-Ausgabe lässt sich durch `EXPLAIN FORMAT text` erzwingen.

Das Werkzeug `OQLTools.toSQL` transformiert eine OQL-Query in natives SQL für tiefergehende Analysen. Für die grafische Visualisierung von Metriken wird `explain.depesz.com` empfohlen (Datenschutz beachten!).

Die Aktivierung des Loggers `de.ipcon.MyTISMQueryMonitoring=DEBUG` liefert detaillierte Informationen zu OQL- und SQL-Queries direkt im Server-Log.

Anti-Patterns und deren Vermeidung

- **Riesige, inline generierte IN-Listen**:
  Blähen Abfragen auf und verursachen lange Parsing-Zeiten. Der `IN`-Operator muss zwingend mit Platzhaltern (z. B. `${→ idListe}`) verwendet werden, um die Liste als Parameter zu übergeben.
- **Doppler bei count()-Queries***:
  Entstehen durch implizite Inner-Joins über Many-Relationen. Die Nutzung von `EXISTS` und `WITHIN` verhindert, dass Basis-Objekte für jedes Kind-Element dupliziert werden.
- **Leere Ergebnisse durch fehlende LEFT JOINS**:
  Optionale Relationen in `OR`-Verknüpfungen führen beim standardmäßigen Inner-Join zum Verwerfen der Zeile, wenn die Relation leer ist. Der Fragezeichen-Operator `?` vor dem referenzierten Attribut (z. B. `Medikament?.PZN`) erzwingt einen Left Join und verhindert dies.

Chapter 16. Referenz: XML-Formularelemente & Widgets

16.1. Grundlagen der Referenz

Dieses Kapitel dient als Nachschlagewerk für das XML-basierte Formular-Design in MyTISM (`.frm.xml`). Es listet alle verfügbaren XML-Tags, deren Attribute, Standardwerte (fettgedruckt) und die injizierten Groovy-Variablen alphabetisch auf.

16.1.1. Globale Skriptvariablen (FPanel Bindings)

Alle Eingabe- und Strukturkomponenten leiten sich architektonisch von `FPanel` ab. In sämtlichen Skript-Blöcken (wie `<onAction>`, `<onRefresh>`, `<visibleIf>`, `<editableIf>`) stehen standardmäßig folgende injizierte Variablen zur Verfügung:

Variablenname	Klasse/Interface	Definition	Beschreibung
`ctx`	ClientContextI	`ftx.getCtx()`	Client-weiter Kontext, wird z. B. zum Öffnen von Dialogen oder Formularen benutzt.
`user`	Benutzer	`ftx.getCtx().getSession().getUser()`	Der im Client aktuell angemeldete Benutzer.
`ftx`	FormContextI	`ftx`	Kontext des Formulars, in dem das Element eingesetzt ist. Wird gebraucht, um andere Formularelemente anzusprechen.
`bo`	BO	`ftx.getBO()`	Das zugrundeliegende BO des Elements (kann vom `rootBO` abweichen).

Variablenname	Klasse/Interface	Definition	Beschreibung
`tx`	Transaction	`ftx.getRoot().getTransaction()`	Die Transaction, mit der das `rootBO` geladen wurde (wenn `bol instanceof Transaction`).
`fe`	FormElementI	`fe`	Das konkrete GUI-Element selbst (als `FormElementI`).
`bol`	BOLoaderI	`ftx.getRoot().getBOLoader()`	Der Loader, mit dem die Daten des Strukturelements geladen wurden.
`rootBO`	BO	`ftx.getRoot().getBO()`	Das zugrundeliegende BO des gesamten Formulars.

16.2. Action

Definiert eine interaktive Aktion, die als Schaltfläche oder Kontextmenü-Eintrag gerendert werden kann.

Name	Erlaubte Werte	Beschreibung
`acceleratorKey`	String: z. B. `"ENTER"`, `"control shift F5"`, …	Tastatur-Shortcut zur direkten Auslösung der Aktion.
`accKey`	siehe `acceleratorKey`	Alias für `acceleratorKey`.
`animation`	Boolean: **true**, false	Legt während der Ausführung eine Ladeanimation über das Formular.
`cmd`	String	Funktionsname, der z. B. von Buttons gerufen werden kann.
`contextMenu`	Boolean: true, **false**	Bestimmt, ob die Aktion im Standard-Kontextmenü des Elements erscheint.

Name	Erlaubte Werte	Beschreibung
`formElementSync`	Boolean: **true**, false	Synchronisiert die Formulardaten zwingend vor der Ausführung der Aktion.
`icon`	String: z. Bsp. icon="20x20/New.gif", icon="image/remove_red_eye.svg" oder icon="image/remove_red_eye.svg@5085dc" (mit Farbangabe in Hex; nur für SVGs verfügbar)	Pfad zum gewünschten Icon.
`initialState`	Boolean: true, false	Wird zu Boolean Action mit dem angegebenen Anfangszustand. Dieser schaltet bei jeder Ausführung der Action um. Wird aktuell nur vom `ToggleButton` unterstützt.
`local`	Boolean: true, false	Erzwingt das Auftauchen der Action am umgebenden Element, anstatt sie an die nächst höhere Border zu propagieren.
`menu`	String	Name des übergeordneten Menüpfads, in dem die Aktion platziert werden soll.
`merge`	Boolean: true, **false**	Führt Actions zusammen oder überschreibt diese.
`mnemonicKey`	String	Aktiviert ein Windows/Linux-Tastaturkürzel (Alt + Buchstabe) für Menüs.
`name`	String	Name der Action. Wenn nicht angegeben, gleich `cmd`. Dient standardmäßig als Button-Titel.

Name	Erlaubte Werte	Beschreibung
`offEDT`	Boolean: true, **false**	Führt die Action in einem neuen Thread aus, um ein Blockieren der GUI zu verhindern.
`priority`	int: **0**	Bestimmt die Sortierungs-Reihenfolge innerhalb von Menüs oder Symbolleisten.
`progressShowDelay`	int: **1000**	Wartezeit in Millisekunden, bevor die Ladeanimation gestartet wird.
`restoreFocus`	Boolean: **true**, false	Setzt den Fokus nach Ausführung auf das zuvor aktive Element zurück.
`shortDescription`	String	Kurze Beschreibung, die als Tooltip angezeigt wird.
`showLabel`	Boolean: true, **false**	Zeigt den Namen der Action unterhalb eines ggf. vorhandenen Icons an.
`smallIcon`	String	Pfad zu einem kleineren Icon für kompakte Toolbar-Darstellungen.
`toolBar`	Leerer String	Fügt die Action der Standard-ToolBar neben einer Tabelle beziehungsweise der obersten Toolbar (bei topMdiOnly) hinzu.
`topMdiOnly`	Boolean: true, false	Drückt die Action zwingend auf die oberste Toolbar des Clients (beziehungsweise des Objektfensters im MDI-Modus).

16.2.1. availableOn

Liefert ein hier angegebenes Skript `true` zurück, wird die Action angezeigt.

Die Bedingung für `availableOn` wird nur ein einziges Mal evaluiert, wenn die Komponente initial generiert wird.

16.2.2. enabledOn

Die Action bleibt permanent sichtbar, ist jedoch nur klickbar, wenn das angegebene Skript `true` zurückliefert. Andernfalls wird sie ausgegraut.

Im Gegensatz zu `availableOn` wird `enabledOn` bei jedem Statuswechsel der GUI laufend neu evaluiert.

16.2.3. initialState

Erfüllt die gleiche Aufgabe wie das Attribut `initialState`, nutzt jedoch ein Skript zur Bestimmung des Anfangszustandes.

16.2.4. longDescription

Dient der Anzeige von formatiertem Text (auch HTML) zur tiefergehenden Erklärung der Action.

```
<Action cmd="resetData" name="Daten resetten"
shortDescription="Daten zuruecksetzen" toolBar="" accKey="control
R">
  <onAction language="groovy">rootBO.doMagic()</onAction>
  <longDescription><![CDATA[<html>
    Folgende Voraussetzungen müssen erfüllt sein, damit die Daten
der selektierten Zeilen zurückgesetzt werden können:
    <ul>
      <li>Der Benutzer ist Mitglied der "Verwaltung" oder ein
ADMIN</li>
      <li>Es ist mindestens eine Zeile ausgewählt</li>
    </ul>
  </html>]]></longDescription>
</Action>
```

16.2.5. onAction

Definiert das Skript, welches für die jeweilige Action ausführt wird.

16.3. BooleanInputComponent

Abstrakte Basis-Komponente für alle binären Eingabeelemente. Erbt alle Attribute und

Subelemente von FInputPanel.

Name	Erlaubte Werte	Beschreibung
`class`	String	Vollqualifizierter Java-Klassenname einer benutzerdefinierten Implementierung.
`displayProperty`	DEPRECATED	siehe `property`
`property`	String: Property accessor. Beispiele: property="Buch.Autor.Alter"; property="."	Das Attribut der aktuell betrachteten Entität, das im Kontext dieses Elementes verwendet werden soll. Im zweiten Beispiel wird das BO des aktuellen Formkontexts als Property gesetzt.
`text`	String	Die Beschriftung, die neben oder auf dem Widget gerendert wird.

16.4. Border

Zieht einen dekorativen Rahmen um Formularbereiche. Erbt alle Attribute und Subelemente von FPanel.

Name	Erlaubte Werte	Beschreibung
`bevel-highlight`	Farbangabe. Bitte entweder als „#rrggbbaa“ oder „r,g,b,a“, „r g b a“ oder eine Farbkonstante der java.awt.Color, z. B. YELLOW angeben. Farbnamen mit Postfix „ISH“ werden in Richtung Weiß verschoben (Mittelwert der einzelnen Farbwerte und 255). Der Alphawert ist optional. Die einzelnen Werte sind bei den beiden letzteren Varianten entweder Float-Werte von 0.0..1.0 (bei 1 bitte 1.0 angeben!) oder Integer-Werte von 0..255. Bitte nur die eine Sorte Werte verwenden. Oder für Random-Farbe: random.	Farbe für abgeschrägte Hervorhebungen.
`bevel-highlightInner`	Farbangabe. #rrggbbaa, r,g,b,a, r g b a. Alpha optional. Beispiele: #14f900, 255,255,0,0, 0.5 0.4 0.3 0.2, GREEN, YELLOWISH	Innere Hervorhebungsfarbe.
`bevel-highlightOuter`	Farbangabe. #rrggbbaa, r,g,b,a, r g b a. Alpha optional. Beispiele: #14f900, 255,255,0,0, 0.5 0.4 0.3 0.2, GREEN, YELLOWISH	Äußere Hervorhebungsfarbe.
`bevel-shadow`	Farbangabe. #rrggbbaa, r,g,b,a, r g b a. Alpha optional. Beispiele: #14f900, 255,255,0,0, 0.5 0.4 0.3 0.2, GREEN, YELLOWISH	Schattierungsfarbe.

Name	Erlaubte Werte	Beschreibung
`bevel-shadowInner`	Farbangabe. #rrggbbaa, r,g,b,a, r g b a. Alpha optional. Beispiele: #14f900, 255,255,0,0, 0.5 0.4 0.3 0.2, GREEN, YELLOWISH	Innere Schattierungsfarbe.
`bevel-shadowOuter`	Farbangabe. #rrggbbaa, r,g,b,a, r g b a. Alpha optional. Beispiele: #14f900, 255,255,0,0, 0.5 0.4 0.3 0.2, GREEN, YELLOWISH	Äußere Schattierungsfarbe.
`bevel-type`	`LOWERED`, `RAISED`	Abschrägung des Rahmens.
`beveled`	`highlight`, `highlightInner`, `highlightOuter`, `shadow`, `shadowInner`, `shadowOuter`	Bestimmt den Stil der Abschrägung.
`etched-highlight`	Farbangabe. #rrggbbaa, r,g,b,a, r g b a. Alpha optional. Beispiele: #14f900, 255,255,0,0, 0.5 0.4 0.3 0.2, GREEN, YELLOWISH	Hervorhebung. Muss zwingend zusammen mit `etched-shadow` verwendet werden.
`etched-shadow`	Farbangabe. #rrggbbaa, r,g,b,a, r g b a. Alpha optional. Beispiele: #14f900, 255,255,0,0, 0.5 0.4 0.3 0.2, GREEN, YELLOWISH	Schattierung. Muss zwingend zusammen mit `etched-highlight` verwendet werden.
`etched-type`	String: **`LOWERED`**, `RAISED`	Stellt die Begrenzung als Vertiefung oder Erhöhung dar.
`etched`	Boolean: true, **false**	Aktiviert den geätzten Rahmen-Stil.
`line-color`	Farbangabe. #rrggbbaa, r,g,b,a, r g b a. Alpha optional. Beispiele: #14f900, 255,255,0,0, 0.5 0.4 0.3 0.2, GREEN, YELLOWISH	Farbe für den Rahmen-Typ `line`.
`line`	Boolean: true, false	Aktiviert einen einfachen, flachen Linienrahmen.

Name	Erlaubte Werte	Beschreibung
`title-justification`	String: **`LEFT`**, `CENTER`, `RIGHT`	Horizontale Positionierung des Titels.
`title-position`	String: `NORTH`, `SOUTH`, **`WEST`**, `EAST`	Vertikale Positionierung des Titels am Rahmen.
`title`	String	Überschrift für die durch die Border abgegrenzten Inhalte.
`toolBar-background`	Farbangabe. #rrggbbaa, r,g,b,a, r g b a. Alpha optional. Beispiele: #14f900, 255,255,0,0, 0.5 0.4 0.3 0.2, GREEN, YELLOWISH	Hintergrundfarbe der Toolbar.
`toolBar-floatable`	Boolean: true, false	Erlaubt das Losreißen der Toolbar vom Rahmen.
`toolBar-layout`	String	Bestimmt den Layoutmanager der Toolbar.
`toolBar-orientation`	String: `HORIZONTAL`, **`VERTICAL`**	Ausrichtung der Toolbar.
`toolBar-position`	String: `NORTH`, `SOUTH`, **`WEST`**, `EAST`	Platzierung der Toolbar. Wird automatisch mit `toolBar-orientation` gekoppelt.
`toolBar-rollover`	Boolean: true, false	Aktiviert visuelles Rollover-Feedback für Toolbar-Buttons.
`topMdi`	Boolean: true, false	Drückt die Symbolleiste direkt in das übergeordnete MDI-Fenster.

Verwenden Sie für `etched` und `line` ausschließlich die Booleschen Werte `true` oder `false`. Nutzen Sie bei `beveled` die konkreten Stilnamen, um Parsing-Konflikte zu vermeiden.

16.5. Button

Eine klassische interaktive Schaltfläche. Erbt alle Attribute und Subelemente von FPanel.

Name	Erlaubte Werte	Beschreibung
`action`	String	`cmd`-Attribut der Action, die bei Klick auf den Button ausgeführt wird.
`background`	Farbangabe. #rrggbbaa, r,g,b,a, r g b a. Alpha optional. Beispiele: #14f900, 255,255,0,0, 0.5 0.4 0.3 0.2, GREEN, YELLOWISH	Hintergrundfarbe.
`defaultButton`	Boolean: true, **false**	Bei **true** wird der Button aktiviert, wenn der Container den Fokus besitzt und die ENTER-Taste gedrückt wird.
`foreground`	Farbangabe. #rrggbbaa, r,g,b,a, r g b a. Alpha optional. Beispiele: #14f900, 255,255,0,0, 0.5 0.4 0.3 0.2, GREEN, YELLOWISH	Schriftfarbe.
`hAlign`	String: `LEFT`, `CENTER`, `RIGHT`	Horizontale Text- und Icon-Ausrichtung.
`icon`	String: z. Bsp. icon="20x20/New.gif", icon="image/remove_red_eye.svg" oder icon="image/remove_red_eye.svg@5085dc" (mit Farbangabe in Hex; nur für SVGs verfügbar)	Pfad zum gewünschten Icon.
`initialFocus`	Boolean: true, **false**	Setzt den initialen Fokus beim Öffnen des Formulars auf dieses Element.
`multiClickThreshold`	Integer (**750ms**)	Spezifiziert den Zeitraum, innerhalb dessen mehrfaches Klicken als ein einziger Klick interpretiert wird.
`text`	String	Beschriftung des Buttons.

Name	Erlaubte Werte	Beschreibung
`vAlign`	String: TOP, CENTER, BOTTOM, z. B. vAlign="TOP"	Bestimmt die vertikale Ausrichtung des Textes innerhalb des Elements. Die Höhe des Elements muss größer sein als eine normale Zeilenhöhe, was z. B. durch Setzen des Attributs prefSize erreicht werden kann.

16.6. Canvas

Ein freier Zeichenbereich für benutzerdefinierte Swing-Grafiken oder Overlays. Erbt alle Attribute und Subelemente von FPanel.

Name	Erlaubte Werte	Beschreibung
`toolTipText`	String.	Text, der angezeigt wird, wenn man den Mauszeiger über das Element hält.

16.7. Chart

Komponente zur grafischen Darstellung von Diagrammen via JFreeChart. Unterstützte Diagrammtypen umfassen unter anderem `timeSeriesChart`, `stackedXYAreaChart`, `scatterPlot`, `xYAreaChart`, `xYLineChart`, `xYStepAreaChart` und `xYStepChart`. Erbt alle Attribute und Subelemente von FPanel.

Name	Erlaubte Werte	Beschreibung
`closed`	Boolean: true, **false**	Schließt die Datenreihen automatisch.
`print`	Boolean: **true**, false	Blendet die Druckoption im Kontextmenü ein.
`property`	String: Property accessor. Beispiele: property="Buch.Autor.Alter"; property="."	Verknüpft das Diagramm mit einer Daten-Relation.
`props`	Boolean: **true**, false	Blendet Einstellungs-Optionen im Kontextmenü ein.

Name	Erlaubte Werte	Beschreibung
`save`	Boolean: **true**, false	Blendet die Speicher-Option (Bild-Export) im Kontextmenü ein.
`toolTips`	Boolean: **true**, false	Aktiviert interaktive Tooltips an den Datenpunkten.
`zoom`	Boolean: **true**, false	Aktiviert die interaktive Zoom-Möglichkeit über die Maus.

16.7.1. buildScript

Das Groovy-Skript zur Generierung der Diagrammdaten. Der `JFreeChartBuilder` wird als Variable `builder` an das Skript übergeben. Innerhalb der Closure steht das `anchor`-Objekt zur Verfügung, welches das Formular-BO referenziert.

```
<Chart>
  <buildScript><![CDATA[
  builder.xYLineChart(title:'oneTitle', xAxisLabel:"EK",
yAxisLabel:'VK') {
    def zes = anchor.ZeiterfassungsEintraege
    antiAlias=true
    borderVisible=false
    borderPaint='#c0c0c0'
    plot {
      tableXYDataset() {
        rows = { (0..zes.size()-1).each{ it } }
        x = { it }
        series(name: anchor.kontakt.describe()) {
          values = { zes.values().getAt(it).Anwesenheitsdauer }
          stroke = 2
          paint = '#4c1e67'
        }
      }
    }
  }
  ]]></buildScript>
</Chart>
```

16.7.2. onClick

Leitet aufbereitete Click-Events auf Elemente innerhalb der Chart weiter. Existiert diese Subnode, werden klickbare Elemente visuell hervorgehoben. Verfügbare Variablen im Skript sind `row`, `column` (zur Identifizierung in BarCharts) und `section` (in Ring- oder PieCharts).

Um die Daten des angeklickten Elements weiterzuverarbeiten, empfiehlt es sich, eine Map mit den Schlüsseln im Kontext-Binding des Groovy-Skripts (z. B. im `onConstruction` der View) zu hinterlegen. Alternativ können die Schlüssel oder Labels aber auch als Filter interpretiert werden und als Query in einem zu öffnenden Lesezeichen gesetzt werden.

```
<Chart>
  <buildScript>[ ... ]</buildScript>
  <onClick><![CDATA[
    import de.ipcon.tools.date.DateTimeTools

    def bkm = ctx.getBOLoader().getBOByAttr(Lesezeichen, 'Tid',
'MCS_Rechnungen')
    // re-construct the month range from the column label
    def month =
DateTimeTools.getFirstDayOfMonth(L10n.parseDate(column, 'MM/yy'),
true)
    def presetQuery = "[Belegdatum >=
'${L10n.formatISODate(month)}'"

    ctx.openView(bkm, [query: presetQuery.toString()])
  ]]></onClick>
</Chart>
```

16.8. CheckBox

Ein Auswahlfeld für binäre Zustände. Erbt alle Attribute und Subelemente von BooleanInputComponent.

Name	Erlaubte Werte	Beschreibung
`triState`	Boolean: **true**, false	Wenn hier „true“ gesetzt ist, kann das Element drei Zustände annehmen: true, false und null. Ansonsten nur true und false. Bei `false` entfällt der undefinierte Null-Zustand.

16.9. ComboBox

Ein klassisches Dropdown-Auswahlfeld. Erbt alle Attribute und Subelemente von FInputPanel.

Name	Erlaubte Werte	Beschreibung
`autoSelect`	String: `first`, `last`, `firstNonNull`	Definiert, welches Element initial als Default ausgewählt wird.
`chooseOnly`	Boolean: true, **false**	Verbietet bei `true` freie Texteingaben in das Feld.
`format`	String (CBOFormat)	Legt die Formatierung der Anzeige fest (z. B. `'Id: 'Name`).
`nullable`	Boolean: **true**, false	Verbietet bei `false` eine leere (`null`) Auswahl.
`nullChoiceTitle`	String	Der Platzhalter-Text für eine leere Auswahl.
`selectEntity`	String	Name der Entität, deren gesamte Objekte zur Auswahl angeboten werden.
`selectOutOf`	String	Name der spezifischen Relation, aus der Objekte angeboten werden.
`sortBy`	String	Name des Attributes für die Sortierung der Liste.
`whereClause`	String (OQL)	Optionaler OQL-Filter zur strikten Einschränkung der Auswahl.

Name	Erlaubte Werte	Beschreibung
`showId`	Boolean: true, **false**	Zeigt die ID jedes Elements in eckigen Klammern an.
`suppressDuplicatesInNonRelationMode`	Boolean: true, **false**	Verhindert optische Duplikate durch Anzeige der IDs in eckigen Klammern.

16.9.1. choiceScript

Ein Groovy-Skript, das eine Map für statische, hartcodierte Optionen zurückgibt.

```
<ComboBox property="FilterZeitraum" e-label="$R{Zeitraum}"
chooseOnly="true" nullable="false">
  <choiceScript>
    ['letzten 2 Tage'    : '2Tage',
     'letzten 7 Tage'    : '7Tage',
     'Alle'              : 'Alle']
  </choiceScript>
</ComboBox>
```

16.10. DateChooser

Eingabefeld zur Datumsauswahl über einen ausklappbaren Kalender. Erbt alle Attribute und Subelemente von FTextInputComponent.

Name	Erlaubte Werte	Beschreibung
`autoHideButton`	Boolean: true, **false**	Blendet das Kalendersymbol automatisch aus, sobald das Feld schreibgeschützt wird.
`columns`	Integer	Legt die physische Zeichenbreite des Textfeldes fest.
`format`	String: `LONG_`, `MEDIUM_`, `SHORT_`, `dd/MM/YYYY`	Bestimmt die Formatierung des angezeigten Datums.
`popupHeight`	Integer (px, c, em, dlu)	Bestimmt die Höhe des Kalender-Popups.

Name	Erlaubte Werte	Beschreibung
`popupWidth`	Integer (px, c, em, dlu)	Bestimmt die Breite des Kalender-Popups.
`timeZoneProperty`	String	Übersteuert die im Schema definierte Zeitzone für die Anzeige.

16.11. Editor

Code-Editor mit Syntax-Hervorhebung. Erbt alle Attribute und Subelemente von FInputPanel.

```
<Editor property="Bemerkung" mode="patch"/>
```

Name	Erlaubte Werte	Beschreibung
`columns`	int: **60**	Spaltenbreite des Editors.
`electricScroll`	int: **3**	Sichert, dass beim Scrollen stets x Zeilen über und unter dem Cursor sichtbar bleiben.
`focusable`	Boolean: **true**, false	Entzieht dem Feld den Fokus.
`initialFocus`	Boolean: true, **false**	Setzt den initialen Fokus beim Öffnen des Formulars auf dieses Element.
`maxUndos`	int: **500**	Maximale Anzahl der Undo-Schritte im Speicher.
`mode`	String: **`xml`**, `groovy`, `log`, `patch`	Definiert die Sprache für die Syntax-Hervorhebung.
`rows`	int: **10**	Angezeigte Zeilenanzahl.
`text`	String	Belegt den Editor mit statischem Text vor.

16.12. Element

Dekorativer und logischer Struktur-Wrapper für Formularfelder. Erbt alle Attribute und Subelemente von FPanel.

Name	Erlaubte Werte	Beschreibung
`autoCreate`	Boolean: true, **false**	Erzeugt das verknüpfte BO der Relation automatisch, falls es beim Laden `null` ist.
`autoHide`	Boolean: true, **false**	Bei `false` wird das Element auch dann ausgegraut angezeigt, wenn das Attribut nicht aufgelöst werden kann.
`hideForNullBO`	Boolean: **true**, false	Blendet das Widget komplett aus, wenn das übergeordnete BO `null` ist.
`hSpaceDist`	Double: **-1**	Bestimmt die restliche Platzverteilung bei aktivem `rightFill`. Bei 0.5 werden Elemente zentriert.
`label`	String	Die Beschriftung links über oder vor dem umschlossenen Feld.
`labelBackground`	Farbangabe. Bitte entweder als „#rrggbbaa“ oder „r,g,b,a“, „r g b a“ oder eine Farbkonstante der java.awt.Color, z. B. YELLOW angeben. Farbnamen mit Postfix „ISH“ werden in Richtung Weiß verschoben (Mittelwert der einzelnen Farbwerte und 255). Der Alphawert ist optional. Die einzelnen Werte sind bei den beiden letzteren Varianten entweder Float-Werte von 0.0..1.0 (bei 1 bitte 1.0 angeben!) oder Integer-Werte von 0..255. Bitte nur die eine Sorte Werte verwenden. Oder für Random-Farbe: random.	Hintergrundfarbe des Labels.

Name	Erlaubte Werte	Beschreibung
`labelForeground`	Farbangabe. #rrggbbaa, r,g,b,a, r g b a. Alpha optional. Beispiele: #14f900, 255,255,0,0, 0.5 0.4 0.3 0.2, GREEN, YELLOWISH	Schriftfarbe des Labels.
`mode`	String: `FREE_FIELD`, **`FREE_LABEL`**, `LABEL_ON_TOP`, `DEFAULT`	`FREE_FIELD`: Feld füllt Raum bis zum rechten Rand. `FREE_LABEL`: Ausrichtung an der Kante des längsten Labels. `LABEL_ON_TOP`: Label steht über dem Feld.
`no-label`	Boolean: true, **false**	Unterdrückt die Generierung des Labels vollständig.
`rightFill`	Double	Prozentuale Streckung des Feldes zum rechten Rand (0.0 bis 1.0).
`rows`	Integer	Zeilenhöhe im Grid-Layout.
`transactionControl`	Boolean: true, **false**	Kapselt das Feld in eine eigenständige Datenbank-Transaktion, getrennt vom Hauptformular.
`x` / `y`	Integer	Explizite x/y-Koordinaten im Raster des Containers.

16.13. Email

Eingabefeld für E-Mail-Adressen. Erbt alle Attribute und Subelemente von FTextInputComponent.

16.14. FInputPanel (abstrakt)

Zentralisiert die Validierungslogik und Pflichtfeld-Prüfungen für interaktive Eingabekomponenten. Erbt alle Attribute und Subelemente von FInputComponent.

Name	Erlaubte Werte	Beschreibung
`initialFocus`	Boolean: true, **false**	Setzt den initialen Fokus beim Öffnen des Formulars auf dieses Element.

16.14.1. alsoMandatoryIf

Macht ein Eingabeelement zur Laufzeit dynamisch zum Pflichtfeld.

```
<alsoMandatoryIf language="groovy">
   ctx.currentUser.istMitgliedVon("Pflichtgruppe")
</alsoMandatoryIf>
```

Standardmäßig ist das Skript gecacht. Für eine Re-Evaluierung bei Refreshs muss `cached="false"` gesetzt werden.

```
<alsoMandatoryIf cached="false" language="groovy">
   !rootBO.KundeWillDatenNichtNennen
</alsoMandatoryIf>
```

Im Schema hart definierte Pflichtfelder (`mandatory`) können über `alsoMandatoryIf` niemals ausgehebelt werden.

16.15. FPanel (abstrakt)

Der absolute Urvater aller Layout- und Eingabe-Komponenten im XML-Design.

Name	Erlaubte Werte	Beschreibung
`debug`	Boolean: true, **false**	Zeichnet rote Hilfslinien zur Layout-Korrektur.
`editable`	Boolean: **true**, false	Bei „false“ kann innerhalb dieses Elements kein Feld mehr editiert werden.
`implied`	Boolean: true, **false**	Kennzeichnet implizit generierte Elemente.
`l10nBundle`	DEPRECATED	Ohne Funktion.
`maximumSize` / `maxSize`	Tupel: (horizontal, vertikal). Beispiele: maxSize="4c, 5c"; maxSize="4c,"; maxSize=",5c"	Maximale Layout-Ausdehnung.

Name	Erlaubte Werte	Beschreibung
`minimumSize` / `minSize`	Tupel: (horizontal, vertikal). Beispiele: minSize="4c, 5c"; minSize="4c,"; minSize=",5c"	Gibt die minimale Größe des Elements mit horizontalem und vertikalem Wert an und ersetzt die ansonsten automatische Größenberechnung. Diese würde das Element auf das Minimum an Platz für dessen Inhalt setzen. Beide Werte sind jeweils optional.
`missingPropertiesPolicy`	String: **`error`**, `ignore`, `log`	Verhalten bei im Schema fehlenden Properties.
`name`	String	Interner Referenzname für programmatische Griffe via `ftx['Name']`.
`preferredSize` / `prefSize`	Tupel: (horizontal, vertikal). Beispiele: prefSize="8c, 6c"; prefSize="8c,"; prefSize=",6c"	Gibt die bevorzugte Größe des Elements mit horizontalem und vertikalem Wert an und ersetzt die ansonsten automatische Größenberechnung. Diese würde das Element auf das Minimum an Platz für dessen Inhalt setzen. Beide Werte sind jeweils optional.
`scrollable`	Boolean: true, false; Oder Richtungsbeschränkung, z. B. scrollable="VERTICAL_ONLY"	Scrollbar für dieses Element ein- oder ausschalten beziehungsweise auf eine Richtung beschränken.

Subelemente:

16.15.1. editableIf

Bestimmt über ein Skript, ob das Element editiert werden darf.

```
<Text property="Beschreibung">
  <editableIf
language="groovy">bo.kannEditiertWerden</editableIf>
```

```
</Text>
```

16.15.2. dropAllowedIf

Prüft, ob ein Drag-and-Drop-Vorgang auf diesem Element zulässig ist.

16.15.3. onAfterSetValue

Wird ausgeführt, nachdem ein Wert programmatisch oder durch den Benutzer im Element gesetzt wurde.

16.15.4. onBeforeSave / onAfterSave

Lifecycle-Hooks, die unmittelbar vor oder nach dem Speichern des umgebenden Formulars ausgelöst werden.

16.15.5. onConstruction

Skript, das exakt einmal während der Instanziierung der GUI-Komponente aufgerufen wird.

16.15.6. onDrop

Behandelt Dateien, die per Drag-and-Drop auf das Element gezogen wurden.

```
<onDrop language="groovy">
  def result = Datei.importFileFromDnD(ftx, tx, files, rootBO)
  ftx.getRoot().refreshForms()
</onDrop>
```

16.15.7. onFocusGained / onFocusLost

Wird beim Setzen oder Verlassen des Eingabe-Fokus ausgeführt.

16.15.8. onMDIOpen / onMDIClose / onMDIActivate / onMDIDeactivate

Spezifische Hooks zur Steuerung und Reaktion auf MDI-Fenster-Events (Öffnen, Schließen, Aktivieren, Deaktivieren).

16.15.9. onRefresh / onSync

`onRefresh` lädt Model-Daten in die View. `onSync` schreibt View-Daten zurück ins Model.

16.15.10. script

Universeller Block zur Deklaration wiederverwendbarer Methoden und lokaler Variablen, die exklusiv für dieses Element gelten.

16.15.11. visibleIf

Steuert die Sichtbarkeit des Elements zur Laufzeit.

```
<Text property="Beschreibung">
  <visibleIf language="groovy">bo.istSichtbar</visibleIf>
</Text>
```

Name	Erlaubte Werte	Beschreibung
`leftEntity`	String: Entitätsname	Die verknüpfte Entität muss eine Subklasse der angegebenen Entität sein. Kann Skript-Inhalt dieses Tags ersetzen oder ihn als Konjunktion ergänzen.
`language`	String: groovy, **beanshell**	Die zu verwendende Skriptsprache.
`leftClass`	DEPRECATED	siehe `leftEntity`
`never`	Boolean: true, **false**	Bei `true` wird das Element niemals angezeigt.
`notForLeftEntity`	String	Negierte Form der Klasseneinschränkung.
`notForRightEntity`	String	Negierte Form der Ziel-Klasseneinschränkung.
`property`	String	Überprüft ein Property direkt.
`rightEntity`	String	Das verknüpfte Objekt muss von dieser Klasse sein.

16.16. FTextInputComponent (abstrakt)

Abstrakte Basisklasse, die fundamentale Eingabeeigenschaften für alle Text-Input-Komponenten bereitstellt. Erbt alle Attribute und Subelemente von FInputPanel.

16.17. Image

Widget zur Anzeige von statischen Bildern oder Grafiken. Erbt alle Attribute und Subelemente von FPanel.

Name	Erlaubte Werte	Beschreibung
`displayProperty`	DEPRECATED	siehe `property`
`property`	String: Property accessor. Beispiele: property="Buch.Autor.Alter"; property="."	Das Attribut der aktuell betrachteten Entität, das im Kontext dieses Elementes verwendet werden soll. Im zweiten Beispiel wird das BO des aktuellen Formkontexts als Property gesetzt.
`scaleToFit`	Boolean: true, **false**	Skaliert das Bild automatisch auf die Elementgrenzen.
`transparentBG`	Boolean: true, **false**	Aktiviert die Transparenz für den Bildhintergrund.

16.18. Label

Ein reines Text-Anzeige-Widget, das HTML und Farbverläufe rendert. Erbt alle Attribute und Subelemente von FPanel.

Name	Erlaubte Werte	Beschreibung
`arc`	Integer >= 0	Radius für abgerundete Ecken (0 bis 100).
`asyncRefresh`	Boolean: true, false, **null**	Erzwingt asynchronen Refresh bei stark variierendem HTML-Inhalt.

Name	Erlaubte Werte	Beschreibung
`background`	Farbangabe. Bitte entweder als „#rrggbbaa“ oder „r,g,b,a“, „r g b a“ oder eine Farbkonstante der java.awt.Color, z. B. YELLOW angeben. Farbnamen mit Postfix „ISH“ werden in Richtung Weiß verschoben (Mittelwert der einzelnen Farbwerte und 255). Der Alphawert ist optional. Die einzelnen Werte sind bei den beiden letzteren Varianten entweder Float-Werte von 0.0..1.0 (bei 1 bitte 1.0 angeben!) oder Integer-Werte von 0..255. Bitte nur die eine Sorte Werte verwenden. Oder für Random-Farbe: random.	Hintergrundfarbe.
`class`	String	Eigene Java-Implementierungsklasse.
`clickable`	true, **false**	Macht das Label klickbar (öffnet bei BO-Properties das Standardformular).
`disabledIcon`	String	Alternatives Icon für den schreibgeschützten Zustand.
`displayFormat`	DEPRECATED	siehe `format`
`displayProperty`	DEPRECATED	siehe `property`
`font`	String	Schriftart.
`fontSize`	String: +X%	Gibt an, um wie viel Prozent die Schrift vergrößert werden soll.
`fontStyle`	String: z. B. fontStyle="bold", fontStyle="italics" oder fontStyle="BOLD", fontStyle="italics"	String: z. B. fontStyle="bold", fontStyle="italics" oder fontStyle="BOLD", fontStyle="italics"

Name	Erlaubte Werte	Beschreibung
`foreground`	Farbangabe. Bitte entweder als „#rrggbbaa“ oder „r,g,b,a“, „r g b a“ oder eine Farbkonstante der java.awt.Color, z. B. YELLOW angeben. Farbnamen mit Postfix „ISH“ werden in Richtung Weiß verschoben (Mittelwert der einzelnen Farbwerte und 255). Der Alphawert ist optional. Die einzelnen Werte sind bei den beiden letzteren Varianten entweder Float-Werte von 0.0..1.0 (bei 1 bitte 1.0 angeben!) oder Integer-Werte von 0..255. Bitte nur die eine Sorte Werte verwenden. Oder für Random-Farbe: random.	Farbangabe. Bitte entweder als „#rrggbbaa“ oder „r,g,b,a“, „r g b a“ oder eine Farbkonstante der java.awt.Color, z. B. YELLOW angeben. Farbnamen mit Postfix „ISH“ werden in Richtung Weiß verschoben (Mittelwert der einzelnen Farbwerte und 255). Der Alphawert ist optional. Die einzelnen Werte sind bei den beiden letzteren Varianten entweder Float-Werte von 0.0..1.0 (bei 1 bitte 1.0 angeben!) oder Integer-Werte von 0..255. Bitte nur die eine Sorte Werte verwenden. Oder für Random-Farbe: random.
`format`	String (CBOFormat)	Dynamische Formatierungsschablone. Setzt `property` zwingend voraus.
`gradientStartColor`	Farbangabe	Anfangsfarbe für lineare Farbverläufe.
`gradientStartPosition`	String: `NORTH`, `SOUTH`, `WEST`, `EAST`	Ausrichtung des Verlaufs-Starts.
`gradientStopColor`	Farbangabe	Endfarbe für lineare Farbverläufe.
`gradientStopPosition`	String: `NORTH`, `SOUTH`, `WEST`, `EAST`	Ausrichtung des Verlaufs-Endes.
`hAlign`	String: `LEFT`, `CENTER`, `RIGHT`	Horizontale Text-Ausrichtung.
`hTextPosition`	String	Position des Textes im Verhältnis zum Icon.

Name	Erlaubte Werte	Beschreibung
`html`	Boolean: true, **false**	Erlaubt die direkte Interpretation von HTML-Tags. Aktiviert bei `href` automatisch `clickable`.
`icon`	String: z. Bsp. icon="20x20/New.gif", icon="image/remove_red_eye.svg" oder icon="image/remove_red_eye.svg@5085dc" (mit Farbangabe in Hex; nur für SVGs verfügbar)	Pfad zum gewünschten Icon.
`iconColor`	Farbangabe	Explizite Einfärbung für SVG-Vektorgrafiken.
`iconTextGap`	Integer	Abstand in Pixeln zwischen Icon und Text.
`openProperty`	String	Bestimmt eine abweichende Property, die beim Anklicken geöffnet werden soll (setzt `clickable="true"`).
`padding`	String	Pixel-Innenabstand: Oben, Links, Unten, Rechts (z. B. `20,30,20,30`).
`property`	String: Property accessor. Beispiele: property="Buch.Autor.Alter"; property="."	Das Attribut der aktuell betrachteten Entität, das im Kontext dieses Elementes verwendet werden soll. Im zweiten Beispiel wird das BO des aktuellen Formkontexts als Property gesetzt.
`text`	String	Statischer Anzeigetext (erlaubt HTML bei `html="true"`).
`textWhileLoading`	String	Platzhalter-Text während asynchroner Hintergrundprozesse.

Name	Erlaubte Werte	Beschreibung
`toolTipText`	String.	Text, der angezeigt wird, wenn man den Mauszeiger über das Element hält.
`vAlign`	String: TOP, CENTER, BOTTOM, z. B. vAlign="TOP"	Bestimmt die vertikale Ausrichtung des Textes innerhalb des Elements. Die Höhe des Elements muss größer sein als eine normale Zeilenhöhe, was z. B. durch Setzen des Attributs prefSize erreicht werden kann.
`vTextPosition`	String: `TOP`, `CENTER`, `BOTTOM`	Vertikale Textposition zum Icon.

16.18.1. Format

Deklariert ein alternatives CBOFormat direkt als XML-Element anstelle des Attributs.

16.18.2. Text

Definiert den anzuzeigenden Text in einem XML-Element für mehrzeilige oder komplexe HTML-Strukturen. Bei HTML muss das `<html>`-Tag ohne vorangestellte Leerzeichen exakt am Anfang stehen.

```
<Label>
  <Text><![CDATA[<html>
    <body>Ein Text als HTML.<br/>
      Damit Sonderzeichen nicht codiert werden, wird CDATA
genutzt.
    </body>
  </html>]]></Text>
</Label>
```

16.18.3. onClick

Klickbare Labels reagieren auf `clickable="true"`, auf HTML-Links oder auf ein explizites `onClick`-Skript, welches die Parameter `ftx`, `fe` und `event` entgegennimmt.

16.19. PDFViewer

Widget zur nativen Anzeige und Annotation von PDF-Dokumenten im Formular. Erbt alle Attribute und Subelemente von FPanel.

Name	Erlaubte Werte	Beschreibung
`annotationColorEdit`	Farbangabe. #rrggbbaa, r,g,b,a, r g b a. Alpha optional. Beispiele: #14f900, 255,255,0,0, 0.5 0.4 0.3 0.2, GREEN, YELLOWISH Standard: **#000000**	Editor-Farbe für Anmerkungen.
`annotationColorSaved`	Farbangabe. #rrggbbaa, r,g,b,a, r g b a. Alpha optional. Beispiele: #14f900, 255,255,0,0, 0.5 0.4 0.3 0.2, GREEN, YELLOWISH Standard: **FF0000**	Farbe für gespeicherte PDF-Anmerkungen.
`annotationFont`	String: **Helvetica-18**	Schriftart für Anmerkungen.
`bufferedImage`	Boolean: true, **false**	Puffert das Bild speicherintensiv vor, um das Rendering unter Windows zu beschleunigen.
`e-no-label`	Boolean: true, **false**	Unterdrückt das automatische Label des Elements.
`enableAnnotations`	Boolean: true, **false**	Aktiviert interaktive PDF-Anmerkungen.
`fitOnPage`	Boolean: true, **false**	Passt das Dokument an die Seitenhöhe an.
`fitWidth`	Boolean: **true**, false	Passt das Dokument an die Seitenbreite an.
`forceAntialiasing`	Boolean: **true**, false	Erzwingt pixelgenaue Kantenglättung.
`lowQuality`	Boolean: true, **false**	Reduziert die Render-Qualität zugunsten der Performance.

Name	Erlaubte Werte	Beschreibung
`property`	String: Property accessor. Beispiele: property="Buch.Autor.Alter"; property="."	Verweist auf den BLOB-Datenstrom der PDF-Datei.
`scaleBicubic`	Boolean: true, **false**	Aktiviert bilineare Filterung beim Skalieren.
`scaleToFit`	Boolean: true, **false**	Alias für `fitOnPage`.
`zoomValue`	Double: **0.05**	Standard-Zoom-Schrittfaktor.

16.19.1. onAnnotationAdded / onAnnotationChanged / onAnnotationRemoved / onAnnotationSelected

Skripte, die bei entsprechenden Annotations-Ereignissen im PDF-Viewer feuern.

16.20. Popup

Eingabefeld mit Such- und Erstellungs-Icons zur Verknüpfung von relationalen Objekten. Erbt alle Attribute und Subelemente von FInputPanel.

Name	Erlaubte Werte	Beschreibung
`align`	String	Textausrichtung im Feld.
`autoEdit`	Boolean: true, **false**	Erlaubt das Editieren physisch abhängiger Attribute direkt in der Detailview.
`columns`	Integer	Zeichenbreite des Eingabefeldes.
`displayFormat`	DEPRECATED	siehe `format`
`displayFormatDivider` / `displayFormatPostfix` / `displayFormatPrefix`	String	Formatierungs-Schablonen für die Anzeige.
`displayProperty`	DEPRECATED	siehe `property`
`displaySort`	String	Sortier-Attribute für die Schnelleingabe.
`editable`	Boolean: **true**, false	Sperrt das Feld gegen textuelle Eingaben.

Name	Erlaubte Werte	Beschreibung
`fallBackProperty`	String	Fallback-Attribut, falls die primäre Property `null` ist.
`fontsize`	String: +X%	Gibt an, um wie viel Prozent die Schrift vergrößert werden soll.
`fontStyle`	String: z. B. fontStyle="bold", fontStyle="italics" oder fontStyle="BOLD", fontStyle="italics"	String: z. B. fontStyle="bold", fontStyle="italics" oder fontStyle="BOLD", fontStyle="italics"
`foreground`	Farbangabe. Bitte entweder als „#rrggbbaa“ oder „r,g,b,a“, „r g b a“ oder eine Farbkonstante der java.awt.Color, z. B. YELLOW angeben. Farbnamen mit Postfix „ISH“ werden in Richtung Weiß verschoben (Mittelwert der einzelnen Farbwerte und 255). Der Alphawert ist optional. Die einzelnen Werte sind bei den beiden letzteren Varianten entweder Float-Werte von 0.0..1.0 (bei 1 bitte 1.0 angeben!) oder Integer-Werte von 0..255. Bitte nur die eine Sorte Werte verwenden. Oder für Random-Farbe: random.	Schriftfarbe.
`format`	String (CBOFormat)	Formatierung des BOs im Textfeld.
`lazy`	Boolean: true, false	Initialisiert Daten erst beim Klick.
`lookupCaseSensitive`	Boolean: true, **false**	Setzt die Schnelleingabe-Suche auf case-sensitive.
`lookupProperty`	String	Komma-separierte Attribute zur Durchsuchung bei der Schnelleingabe.

Name	Erlaubte Werte	Beschreibung
`lookupStartingWith`	Boolean: true, false	Sucht ausschließlich am Anfang des Wortes.
`lookupSubstring`	Boolean: **true**, false	Sucht nach Teilstrings im Wort.
`nullChoiceTitle`	String	Text bei leerer Auswahl.
`offerCopyBeforeEdit`	Boolean: true, false	Bietet dem Benutzer vor Änderungen das Anlegen einer Kopie an.
`openFormTid`	String	Formular-TID für den Doppelklick-Aufruf.
`openProperty`	String	Abweichende Property für den Doppelklick-Aufruf.
`popupAlign`	String	Ausrichtung des Popups zum Eingabefeld.
`popupHeight` / `popupWidth` / `popupSize`	String (px, c)	Dimensionen des resultierenden Such-Popups.
`showEntityName`	Boolean: **true**, false	Zeigt den Entitätsnamen im Such-Kopf an.
`showNewAction`	Boolean: true, **false**	Blendet die Neuanlage-Schaltfläche ein.
`showSelectAction`	Boolean: **true**, false	Blendet das Such-Icon ein.
`subentitiesToExclude`	String	Schließt Unterentitäten von der polymorphen Suche aus.
`templateSource`	String	Vorlage für die Schnelleingabe.
`usePolymorphySelectionTree`	Boolean: true, false	Aktiviert einen Baum zur Typ-Auswahl bei polymorphen Suchen.

16.21. Scheduler

Komplexe Komponente zur Ressourcenplanung (Dienstpläne, Schichtplanung) entlang einer Zeitachse. Unterstützt die Integration von `<Border>` und `<DetailView>`. Die minimale Konfiguration erfordert die Attribute `property`, `range`, `itemClass`, `groupClass` sowie ein valides `dataMapper`-Skript. Erbt alle Attribute und Subelemente von FPanel.

Terminologie und Rollen:

Begriff	Beispiele	Verwendung
Item	ZeiterfassungsEintrag, Termin	Dies sind die entlang einer Zeitachse anzuordnenden Objekte.
Contact	Mitarbeiter, Maschine	Items werden nach Contacts gruppiert. Um den Scheduler benutzen zu können, muss in der Many-Relation mindestens eine Instanz vorhanden sein.
Group	Abteilung, Mandant	Groups ermöglichen es, Contacts visuell zu gruppieren.
Holiday	Feiertag	Dienst ausschließlich der Konfiguration und Anzeige von (read-only) Feiertagen im Kalender.

Name	Erlaubte Werte	Beschreibung
`autoRefresh`	Boolean: true, **false**	Aktualisiert Datenänderungen anderer Clients automatisch zur Laufzeit.
`allowOverlaps`	Boolean: true, **false**	Erlaubt zeitliche Überschneidungen von Items auf derselben Ressource.
`background`	Farbangabe. #rrggbbaa, r,g,b,a, r g b a. Alpha optional. Beispiele: #14f900, 255,255,0,0, 0.5 0.4 0.3 0.2, GREEN, YELLOWISH	Farbgebung für obere Zeit-Header.
`datatipFormat`	String: **`date time duration`**, `date time`, `date`, `time duration`, `time`, `duration`, `none`	Bestimmt das Tooltip-Format für Items.

Name	Erlaubte Werte	Beschreibung
`foreground`	Farbangabe. #rrggbbaa, r,g,b,a, r g b a. Alpha optional. Beispiele: #14f900, 255,255,0,0, 0.5 0.4 0.3 0.2, GREEN, YELLOWISH	Schriftfarbe.
`gridColor`	Farbangabe. #rrggbbaa, r,g,b,a, r g b a. Alpha optional. Beispiele: #14f900, 255,255,0,0, 0.5 0.4 0.3 0.2, GREEN, YELLOWISH	Gitterfarbe im Kalender. Wenn nicht angegeben, wird eine aufgehellte `headerColor` verwendet.
`groupClass`	String	Klasse für die Rolle der „Groups“ (z. B. Abteilung).
`headerColor`	Farbangabe. #rrggbbaa, r,g,b,a, r g b a. Alpha optional. Beispiele: #14f900, 255,255,0,0, 0.5 0.4 0.3 0.2, GREEN, YELLOWISH	Header-Farbe. Wenn nicht angegeben, wird ein aufgehellter `background` verwendet.
`itemClass`	String	Klasse für die Rolle der „Items“ (Termine).
`noNavigation`	Boolean: true, **false**	Unterdrückt Kalender-Navigationsschaltflächen.
`property`	String	Verknüpfte Relation für die Contacts (Ressourcen).
`range`	String: `Day`, `Week`, **`Month`**	Dargestellter Zeitbereich.
`snapUnit`	String: **`Day`**, `Hour`, `Minute`	Steuert die Mindestgröße neu angelegter Items und um welche minimale Zeiteinheit ein Item bewegt / verändert werden darf. (Default: 1 Day)
`type`	String: **`Resources`**, `Timetable`, `SingleMonth`	Kalender-Ansichtstyp.
`viewOnly`	Boolean: **true**, false	Schützt den Kalender gegen interaktives Verschieben.
`workHours`	String: **6-22**	Hebt Kernarbeitszeiten farblich hervor.

16.21.1. dataMapper

Das Skript konfiguriert geladene BOs, darzustellende Daten und Verknüpfungen sowie erlaubt die Beeinflussung des Verhaltens und der Darstellung.

Als Parameter wird ein `mapper` vom Typ `FSchedulerBOMapper<ItemClass, ContactClass, GroupClass` übergeben, wobei die "GroupClass" optional ist und `Void` sein kann, wenn keine Gruppen genutzt werden. "ContactClass" entspricht hierbei dem Typ der Relation, die im XML-Attribut "property" gesetzt wurde. "ItemClass" muss explizit via XML-Attribut "itemClass" angegeben worden sein.

Zusätzlich verfügbare Parameter sind: `Transaction tx`, `ClientContextI ctx`, `FormContextI ftx` und `FScheduler fe`.

Am `mapper` existieren die Methoden `itemMapper`, `contactMapper` und `groupMapper`, um die entsprechenden Rollen zu konfigurieren.
An diesen "Sub-Mappern" wiederum können wahlweise Attribute oder Funktionen hinterlegt werden.

Optional kann ein `holidayMapper` hinzugefügt werden, wenn Feiertage angezeigt werden sollen.

```
<dataMapper><![CDATA[
   mapper.itemMapper()
       .displayText('Lohnart.L10nName')
       .start('KommtGueltig') // !Mandatory! ①
       .end('GehtGueltig') // !Mandatory! ②
       .contacts('Mitarbeiter') // !Mandatory!
       .groups('Mitarbeiter.Abteilung')
       .lockedIf { it.getLohnart()?.istNormal() } ③
       .resizingAllowedIf { false } ④
       .background { Color.decode('#f2e3a6') } ⑤

   mapper.contactMapper()

.format("(AbstraktePerson.Name2|left(1))(AbstraktePerson.Name1|le
ft(1))") ⑥
       .groups('Abteilung') ⑦
```

```
    mapper.groupMapper()
        .name('L10nName')


    mapper.holidayMapper(Feiertag) ⑧
        .date('Datum') ⑨
        .displayText('L10nName')
        .filter(" EXISTS (WITHIN Laender l WHERE NOT l.Ldel AND
l.Tid = 'LUXEMBOURG')") ⑩
]]></dataMapper>
```

① **itemMapper**: Damit Items bei Bedarf aus der Datenbank nachgeladen werden können, müssen `start` und `end` auf persistente Attribute verweisen.

② **itemMapper**: Um neue Items zu erzeugen und bestehende zu editieren, müssen die Attribute oder Relationen von `start`, `end` und `contacts` beschreibbar sein.

③ **itemMapper**: Über `lockedIf` kann das Editieren einzelner Items blockiert werden. Es ist auch möglich, ein Boolean-Attribut zu übergeben.

④ **itemMapper**: Über `resizingAllowedIf` kann gesteuert werden, ob es möglich ist, die Dauer von Items zu bearbeiten oder ob die Dauer nach der Neuanlage fix bleibt. Per default ist resizing möglich.

⑤ **itemMapper**: Die Methoden `background` und `foreground` steuern die Farbgebung von Text und Hintergrund einzelner Items.

⑥ **contactMapper**: Um zu bestimmen, wie ein Contact angezeigt werden soll, kann entweder über `format` ein CBOFormat oder über `name` ein Attribut oder eine Funktion angegeben werden.

⑦ **contactMapper**: Werden "Groups" verwendet, bestimmt `groups` welche Gruppen geladen werden. Neue Items müssen ihrem „Contact“ (und ggfs. dessen „Group“) korrekt zugeordnet sein, oder sie sind **nicht sichtbar**.

⑧ **holidayMapper**: Um die Feiertags-Funktionalität hinzuzufügen, erwartet die Funktion `holidayMapper` eine Angabe der Klasse, welche die Feiertags-Objekte repräsentiert.

⑨ **holidayMapper**: Wird nur ein Datum angegeben, wird dieses implizit als ganzer Tag interpretiert. Alternativ können aber auch `start` und `end` angegeben werden.

⑩ **holidayMapper**: Um ausschließlich die relevanten Feiertage anzuzeigen, reicht es in den meisten Projekten, eine zusätzliche Filterbedingung mitzugeben, welche die Resultate auf die Feiertage eines Landes oder Bundeslandes einschränkt. Für komplexere Szenarien können `query` (zum gezielten Laden) und/oder `contacts()` (zum Verlinken) verwendet werden.

Es ist möglich, an dem Mapper über `itemMapper().query()` eine Funktion zu hinterlegen, die alle „Items“ im Zeitraum der Parameter `LocalDateTime start` und `LocalDateTime end` für die „Contacts“ mit `Set<Long> contactID` lädt. Dies kann sinnvoll oder sogar notwendig sein, wenn mit nicht-persistenten Objekten gearbeitet wird, um trotzdem effizient Daten zu laden. Wird ein expliziter Query implementiert, und die Ansicht ist editierbar, so ist es notwendig, neu erzeugte und ungespeicherte BOs im Rahmen des Query-Scripts mit dem Query-Resultat zusammen zurückzugeben, damit diese beim „Umblättern“ nicht verloren gehen.

Alle weiteren hier aufgelisteten Subelemente enthalten Groovy-Scripte und sind mit Ausnahme des `dataMapper`-Skripts optional.

Name	Variablen / Rückgabetyp	Beschreibung	Default

`validIf`	`Transaction tx`
	`FormContextI ftx`
	`ItemClass item`
	`ContactClass contact`
	`LocalDateTime start`
	`LocalDateTime end`
	Rückgabetyp : `boolean`
		Wird aufgerufen, unmittelbar nachdem der Benutzer versucht hat, ein neues Item anzulegen oder ein Item zu modifizieren, aber bevor diese Neuanlage oder Modifikationen in einer Transaktion aufgezeichnet bzw. auf ein BO angewandt werden. Im Falle einer Neuanlage ist `item = null`.
		Gibt das Skript `false` zurück, wird die Modifikation abgebrochen, ohne einen Fehler zu werfen.
			Per Default unimplementiert. Es wird angenommen, dass jede Modifikation / Neuanlage zulässig ist.

Name	Variablen / Rückgabetyp	Beschreibung	Default
`newItem`	`FormContextI` ftx `Transaction` tx `ContactClass` contact `LocalDateTime` start `LocalDateTime` end Rückgabetyp : `ItemClass`	Wird aufgerufen, wenn der Benutzer die Neuanlage eines Items in der Ansicht via Mouse Drag abgeschlossen hat und `validIf` `true` war oder übersprungen wurde. Die Werte des von `newItem` erzeugten Objekts werden zurück in die Ansicht gesynct, d.h. die übergebenen Zeiten müssen nicht zwangsweise die finalen Zeiten sein. **CAUTION:** Es ist wichtig, dass das erzeugte Item mit dem übergebenen Contact (und dessen Group, wenn verwendet) verknüpft wird, da andernfalls das Item aus der Ansicht verschwindet.	Per Default wird versucht, ein neues Objekt von dem Typ, der in `itemClass` angegeben wurde, zu erzeugen. Start, Ende und Contact werden über die im Mapper definierten Attribute oder Setter-Funktionen gesetzt und müssen daher beschreibbar sein.
`onItemClick`	`ItemClass` item	Das Script wird bei Doppelklick auf ein Item ausgeführt.	Per Default öffnet sich das angeklickte Item in einem für den Benutzer verfügbaren und präferierten Formular.

Name	Variablen / Rückgabetyp	Beschreibung	Default
`onContactClick`	`ContactClass` contact	Das Script wird bei Doppelklick auf einen Contact ausgeführt.	Per Default öffnet sich der angeklickte Contact in einem für den Benutzer verfügbaren und präferierten Formular.

Ausführliches Beispiel: Das nachfolgende Beispiel zeigt eine Verwendung des Schedulers mit der nicht-persistenten Entität `ZEKumuliertFuerTag`. Da nicht-persistente Entitäten nicht in eine Transaction inkludiert und nicht neu geladen werden können, ist es notwendig, Skripte für `newItem` und `onItemClick` zu verwenden. Das `validIf` Skript verhindert, dass Zeiteinträge an Feiertagen und Wochenenden angelegt oder dorthin bewegt werden.

```
<Scheduler property="Mitarbeiter" range="Week" viewOnly="false"
itemClass="ZEKumuliertFuerTag" groupClass="Abteilung"
background="#587ac2" foreground="#ffffff">
  <dataMapper><![CDATA[
    import java.awt.Color

    mapper.itemMapper()
        .displayText { i ->
          i.getBasierendAufZEs().values()
            .findAll { !it.isDeleted() }
            .collect { it.lohnart?.l10nName }
            .unique().join(', ')
        }
        .start('Start') // writeable vattr on ZEKumuliertFuerTag
        .end('Ende') // writeable vattr on ZEKumuliertFuerTag
        .contacts('Mitarbeiter')
        .groups('Mitarbeiter.Abteilung')
        .lockedIf { i -> i.getBasierendAufZEs().values().any {
it.istExplizitGestempelt() || it.lohnart?.istNormal() } }
        .resizingAllowedIf { false }
        .background { ZEKumuliertFuerTag ze ->
          return ze.getLohnart().getColor()
```

```
        }
        .query({ start, end, contactIDs ->
           // use the query option here, since we can't load non-
persistent objects from database, but we can load the persistent
objects they are based on
           // thus, the most efficient solution is to load the
underlying objects from database and transform them
           def startDate = DateTimeToolsNG.toDateOfSystem(start)
           def endDate = DateTimeToolsNG.toDateOfSystem(end)
           final def query = 'ONLY ZeiterfassungsEintrag ze WHERE
ze.Mitarbeiter.Id IN LIST($1) AND ErsterZEFuerTag = NULL' +
               ' AND COALESCE(ze.Kommt, ze.KommtKorrigiert) <= $3
AND' +
               ' (COALESCE(ze.Geht, ze.GehtKorrigiert) >= $2 OR
COALESCE(ze.Geht, ze.GehtKorrigiert) = NULL)'

           def zes = tx.queryBO(query, [contactIDs, startDate,
endDate] as Object[]) as Collection<ZeiterfassungsEintrag>
           def kumul = zes.collect { it.getZEKumuliertFuerTag() }

           // we need to re-add the new entries that were
previously created in the tx (if any)
           for (BO newBO : tx.getNewBOs()) {
             if (newBO instanceof ZeiterfassungsEintrag &&
!(newBO instanceof MultiZeiterfassungsEintrag)) {
               if (((ZeiterfassungsEintrag)
newBO).ersterZEFuerTag == null) {
                 kumul.add(newBO.getZEKumuliertFuerTag())
               }
             }
           }
           return kumul
         })

    mapper.contactMapper()
        .format("(AbstraktePerson.Name1',
')(AbstraktePerson.Name2)")
        .groups('Abteilung')
```

```
    mapper.groupMapper()
        .name('L10nName')
  ]]></dataMapper>
  <newItem><![CDATA[
      import de.ipcon.db.core.BO
      import de.ipcon.tools.date.DateTimeTools
      import java.time.ZoneId
      import java.util.concurrent.TimeUnit

      def e = (Mitarbeiter) contact
      def d =
Date.from(start.atZone(ZoneId.systemDefault()).toInstant())
      def salaryType = Lohnart.forHomeOffice(tx)

      createNewZEsForDay = { Mitarbeiter employee, Date day,
Lohnart lohnart ->
        // [imagine some code creating persistent time entries
for the given day]
      }

      def newItem = createNewZEsForDay(e, d, salaryType)
      return newItem.getZEKumuliertFuerTag()
  ]]></newItem>
  <onItemClick><![CDATA[
      import de.ipcon.form.MDIManagerI

      def ntx = ctx.getNewFormTransaction()
      // load the persistent entity which builds the np-entity so
we have something to frap
      def frappedEntry =
ntx.getBO(item.getBasierendAufZEs().values().find().getId())
      // open the np-object now that it's been built with the
right tx
      def cumul = frappedEntry.getZEKumuliertFuerTag()
      def form =
ctx.getFormByTid('MCS_ZEITERFASSUNGSEINTRAG_S_TAG',
cumul.getClass().simpleName)
      // and finally open the form
      ctx.openForm(form, ntx, cumul, null, null, null, false,
```

```
MDIManagerI.VIEWTYPE_WIZARD, null, false, false, false, /*
doNotIncludeNotIncludedNewBOsWithTxAsLoader = */ true)
  ]]></onItemClick>
  <validIf><![CDATA[
      import java.time.DayOfWeek
      import java.time.ZoneId

      // This isValid check checks if the employee is allowed to
work on the day we try to set
      def employee = (Mitarbeiter) contact
      def d =
Date.from(start.atZone(ZoneId.systemDefault()).toInstant())
      def contract = employee.getGueltigerArbeitsvertrag(d)

      def weekDay = start.getDayOfWeek()
      if (weekDay == DayOfWeek.SATURDAY) {
      return contract == null || contract.istSamstag()
      } else if (weekDay == DayOfWeek.SUNDAY) {
      return contract == null || contract.istSonntag()
      }

      // holidays
      def handler = contract?.getNiederlassung()?.getBundesland()
?: contract?.getNiederlassung()?.getLand()
      if (handler != null && contract != null &&
!contract.istFeiertag()) {
      return !handler.isFeiertag(d)
      }

      return true
 ]]></validIf>
</Scheduler>
```

16.22. SimpleDurationChooser

Eingabefeld für Zeitspannen (Typ `Duration`), bestehend aus Zahlenfeld und Einheiten-Dropdown. Erbt alle Attribute und Subelemente von FTextInputComponent.

Name	Erlaubte Werte	Beschreibung
`format`	String	Formatierung der angezeigten Zeitspanne. Aktuell unbenutzt.
`defaultUnit`	String: `years`, `months`, `weeks`, `days`, `hours`, `minutes`, `seconds`	Vorausgewählte Zeiteinheit.
`enabled`	Boolean: true, false	Setzt das Element explizit auf bearbeitbar.

16.23. SimpleTimespanChooser

Eingabefeld für Zeitspannen (Typ `Timespan`), bestehend aus Zahlenfeld und Einheiten-Dropdown. Erbt alle Attribute und Subelemente von SimpleDurationChooser.

Name	Erlaubte Werte	Beschreibung
`align`	String: `LEFT`, `CENTER`, `RIGHT`, `LEADING`, `TRAILING`	Textausrichtung.
`fallBackProperty`	String	Fallback-Property.
`fontsize`	String: +X%	Gibt an, um wie viel Prozent die Schrift vergrößert werden soll.
`fontStyle`	String: z. B. fontStyle="bold", fontStyle="italics" oder fontStyle="BOLD", fontStyle="italics"	String: z. B. fontStyle="bold", fontStyle="italics" oder fontStyle="BOLD", fontStyle="italics"
`rows`	int: **4**	Zeilenhöhe.
`selectAllWhenFocused`	Boolean: true, *false*	Wenn das Form-Element den Fokus bekommt, wird der gesamte Inhalt selektiert.

16.24. StyledText

HTML-basierter Rich-Text-Editor mit reduzierter Formatierungs-Toolbar. Erbt alle Attribute und Subelemente von FInputPanel.

Name	Erlaubte Werte	Beschreibung
`property`	String: Property accessor. Beispiele: property="Buch.Autor.Alter"; property="."	Das Attribut der aktuell betrachteten Entität, das im Kontext dieses Elementes verwendet werden soll. Im zweiten Beispiel wird das BO des aktuellen Formkontexts als Property gesetzt.
`displayProperty`	DEPRECATED	siehe `property`
`onlyTextFormattingActions`	Boolean: true, **false**	Zeigt ausschließlich Fett, Kursiv und Unterstrichen in der Toolbar an.
`excludeTableActions`	Boolean: true, **false**	Verbirgt Tabellen-Schaltflächen in der Toolbar.
`columns`	int: **20**	Spaltenbreite.
`rows`	int: **4**	Zeilenhöhe.
`readOnly`	Boolean: true, **false**	Sperrt das Feld gegen Eingaben.
`selectAllWhenFocused`	Boolean: true, *false*	Wenn das Form-Element den Fokus bekommt, wird der gesamte Inhalt selektiert.

16.25. Tab

Ein einzelner Reiter innerhalb einer `TabbedView`. Erbt alle Attribute und Subelemente von FPanel.

Name	Erlaubte Werte	Beschreibung
`background`	Farbangabe. Bitte entweder als „#rrggbbaa“ oder „r,g,b,a“, „r g b a“ oder eine Farbkonstante der java.awt.Color, z. B. YELLOW angeben. Farbnamen mit Postfix „ISH“ werden in Richtung Weiß verschoben (Mittelwert der einzelnen Farbwerte und 255). Der Alphawert ist optional. Die einzelnen Werte sind bei den beiden letzteren Varianten entweder Float-Werte von 0.0..1.0 (bei 1 bitte 1.0 angeben!) oder Integer-Werte von 0..255. Bitte nur die eine Sorte Werte verwenden. Oder für Random-Farbe: random.	Hintergrundfarbe des Reiters.
`editable`	Boolean: **true**, false	Bei „false“ kann innerhalb dieses Elements kein Feld mehr editiert werden.

Name	Erlaubte Werte	Beschreibung
`foreground`	Farbangabe. Bitte entweder als „#rrggbbaa“ oder „r,g,b,a“, „r g b a“ oder eine Farbkonstante der java.awt.Color, z. B. YELLOW angeben. Farbnamen mit Postfix „ISH“ werden in Richtung Weiß verschoben (Mittelwert der einzelnen Farbwerte und 255). Der Alphawert ist optional. Die einzelnen Werte sind bei den beiden letzteren Varianten entweder Float-Werte von 0.0..1.0 (bei 1 bitte 1.0 angeben!) oder Integer-Werte von 0..255. Bitte nur die eine Sorte Werte verwenden. Oder für Random-Farbe: random.	Schriftfarbe des Titels.
`grabFocus`	Boolean: **true**, false	Fokus-Einstellung.
`lazy`	Boolean: **true**, false	Lädt den Inhalt des Reiters erst bei Klick auf den Tab.
`title`	String	Die sichtbare Beschriftung des Reiters.
`toolTipText`	String.	Text, der angezeigt wird, wenn man den Mauszeiger über das Element hält.

16.25.1. onShowingTab / onHidingTab

Werden ausgeführt, sobald der Tab in der Benutzeroberfläche aktiviert oder deaktiviert wird.

16.26. TabbedView

Container-Element für Reiter-Navigationen (Tabs). Erbt alle Attribute und Subelemente von FPanel.

Name	Erlaubte Werte	Beschreibung
`antiAlias`	Boolean: true, **false**	Kantenglättung für Reiter-Texte.
`conflictPolicy`	String: **`IGNORE`**, `ASK`	(Nur im Root) Konfliktbehandlung bei parallelen Speicherungen (`IGNORE`: Merge erzwingen, `ASK`: Benutzer fragen).
`height` / `width`	Integer (px, c)	Dimensionen der View.
`ignoreOtherLocalTransactionSaves`	Boolean: true, **false**	(Nur im Root) Verhindert das Nachziehen von lokalen Fremd-Speicherungen im Formular.
`rotateLabels`	Boolean: **true**, false	Dreht Reiter-Texte bei seitlicher Platzierung.
`tabLayoutPolicy`	String	Layoutmanager der Reiter-Leiste.
`tabPlacement`	String: `BOTTOM`, `TOP`, `LEFT`, `RIGHT`	Platzierung der Reiterkarten.
`useMaximumHeight` / `useMaximumWidth`	Boolean: true, **false**	Streckt die View maximal.

16.27. Table

Rendert Datenlisten, Relationen oder Lesezeichen als durchsuchbare Tabelle. Erbt alle Attribute und Subelemente von FPanel.

Name	Erlaubte Werte	Beschreibung
`alternateCellBackground`	Farbangabe	Hintergrundfarbe aller geraden Zeilen.
`asyncModel`	Boolean: **true**, false	Asynchrones Ladeverhalten.
`autoRefresh`	Boolean: true, **false**	Triggert automatische Updates bei Änderungen.
`additionalAutoRefreshEntities`	String	Komma-separierte Entitätsnamen zur zusätzlichen Live-Aktualisierung.

Name	Erlaubte Werte	Beschreibung
`autoSelectFirst`	Boolean: **true**, false	Selektiert automatisch den ersten Eintrag nach dem Laden.
`autoSelectLast` / `autoSelectNone`	Boolean	Selektionsverhalten beim initialen Laden.
`cellBackground`	Farbangabe	Hintergrundfarbe ungerader Zeilen.
`columns`	String	Kompakte Spaltennotation.
`columnSelectionAllowed`	Boolean: **true**, false	`false` wählt bei Klick stets die gesamte Zeile aus.
`createInDetailView`	Boolean: true, **false**	Erlaubt die Erstellung verknüpfter Relationsobjekte in der Detailview.
`dependent`	Boolean: true, false	Kennzeichnet Objekte als physisch unselbstständig.
`easyEdit`	Boolean: **true**, false	Doppelklick öffnet den Bearbeitungsmodus der Zelle.
`editableDetailView`	Boolean: true, **false**	Wird benötigt, um Objekte aus virtuellen Relationen in der Detailview zu editieren, selbst wenn übergeordnete Attribute schreibgeschützt sind.
`entity`	String	Entitätstyp für die freie Tabellensuche.
`explicitStart`	Boolean: true, **false**	Erfordert manuelle Bestätigung per F5/Enter für den Lade-Start.
`freeSearch`	Boolean: **true**, false	Blendet das generelle Freitext-Suchfeld ein.
`horizontalScrollBarPolicy` / `verticalScrollBarPolicy`	String: `ALWAYS`, `AS_NEEDED`, `NEVER`	Scrollbar-Verhalten.

Name	Erlaubte Werte	Beschreibung
`initialFocus`	Boolean: true, **false**	Setzt den initialen Fokus beim Öffnen des Formulars auf dieses Element.
`intercellSpacingX` / `intercellSpacingY`	Integer	Pixel-Abstände zwischen Zellen.
`itemProperty`	String	Aktiviert Sortierungs- und Verschiebeschaltflächen.
`linkOnly`	Boolean: true, **false**	Verhindert Neuanlagen in der Tabelle.
`loadImmediate`	Boolean: true, **false**	Startet den Ladevorgang sofort beim Öffnen.
`maxRowHeight` / `rowHeight`	Integer	Zeilenhöhen.
`maxRows`	Integer (**100000**)	Limitierung der geladenen Zeilen (SQL-LIMIT).
`missingPropertiesPolicy`	String: **`error`**, `ignore`, `log`	Verhalten bei fehlenden Spalten-Properties.
`openFormTid`	String	Formular-TID zum Öffnen der Zeilenobjekte.
`openProperty`	String	Property für den Doppelklick-Aufruf.
`parentEntity`	String	Parent-Verknüpfung.
`preferredVisibleRows`	Integer (**5**)	Anzahl initial sichtbarer Zeilen.
`reloadBOsWhenOpening`	Boolean: true, **false**	Erzwingt einen frischen Datenbankabruf beim Öffnen von Tabelleneinträgen (Doppelklick), anstatt RAM-Caches zu nutzen.
`resizeMode`	String: `ALL_COLUMNS`, `LAST_COLUMN`, `NEXT_COLUMN`, `SUBSEQUENT_COLUMNS`, `OFF`	Spaltenbreiten-Verhalten bei Größenänderung des Fensters.
`rowSelectionAllowed`	Boolean: **true**, false	Erlaubt die Selektion kompletter Zeilen.

Name	Erlaubte Werte	Beschreibung
`showEntityName`	Boolean: **true**, false	Zeigt den Namen der Entität im Kopf an.
`showHorizontalLines` / `showVerticalLines`	Boolean	Blendet Gitterlinien ein.
`singleClickEdit`	Boolean: true, **false**	Einfacher Klick öffnet den Zelleneditor.
`sortForSelected`	Boolean: true, **false**	Sortiert selektierte Zeilen nach oben.
`subentitiesToExclude`	String	Schließt Unterentitäten von der Anzeige aus.
`usePolymorphySelectionTree`	Boolean: true, false	Aktiviert einen polymorphen Baum für die Typ-Auswahl einer neuen Instanz, die sich aus den verfügbaren Schablonen in derselben Baumstruktur speist, wie sie unter dem Admin-Knoten des Navigationsbaums verknüpft sind.
`viewOnly`	Boolean: true, **false**	Deaktiviert jegliche Interaktionen (Verlinken, Erstellen, Löschen).

16.27.1. Column

Einzelne Spaltendefinition innerhalb der Tabelle.

Name	Erlaubte Werte	Beschreibung
`property`	String: Property accessor. Beispiele: property="Buch.Autor.Alter"; property="."	Das Attribut der aktuell betrachteten Entität, das im Kontext dieses Elementes verwendet werden soll. Im zweiten Beispiel wird das BO des aktuellen Formkontexts als Property gesetzt.
`displayProperty`	DEPRECATED	siehe `property`
`format`	String (CBOFormat)	Formatierung verknüpfter Relationen.

Name	Erlaubte Werte	Beschreibung
`toolTipFormat`	String (CBOFormat)	Tooltip-Formatierung.
`title`	String	Spalten-Überschrift.
`width`	Integer (px, c)	Spaltenbreite.
`vAlign`	String: TOP, CENTER, BOTTOM, z. B. vAlign="TOP"	Bestimmt die vertikale Ausrichtung des Textes innerhalb des Elements. Die Höhe des Elements muss größer sein als eine normale Zeilenhöhe, was z. B. durch Setzen des Attributs prefSize erreicht werden kann.
`cellBackground` / `alternateCellBackground`	Farbangabe. Bitte entweder als „#rrggbbaa“ oder „r,g,b,a“, „r g b a“ oder eine Farbkonstante der java.awt.Color, z. B. YELLOW angeben. Farbnamen mit Postfix „ISH“ werden in Richtung Weiß verschoben (Mittelwert der einzelnen Farbwerte und 255). Der Alphawert ist optional. Die einzelnen Werte sind bei den beiden letzteren Varianten entweder Float-Werte von 0.0..1.0 (bei 1 bitte 1.0 angeben!) oder Integer-Werte von 0..255. Bitte nur die eine Sorte Werte verwenden. Oder für Random-Farbe: random.	Hintergrundfarben für Zeilen.
`cellForeground` / `alternateCellForeground`	Farbangabe. #rrggbbaa, r,g,b,a, r g b a. Alpha optional. Beispiele: #14f900, 255,255,0,0, 0.5 0.4 0.3 0.2, GREEN, YELLOWISH	Schriftfarben für Zeilen.
`deferredRendering`	Boolean: true, **false**	Ohne Funktion.

Name	Erlaubte Werte	Beschreibung
`debug`	Boolean: true, **false**	Aktiviert erweitertes Logging.
`caching`	Boolean: **true**, false	Cacht berechnete Zellwerte.
`style`	String: z. B. fontStyle="bold", fontStyle="italics" oder fontStyle="BOLD", fontStyle="italics"	Schriftstil.
`justification`	String: **LEFT**, `CENTER`, `RIGHT`	Horizontale Textausrichtung.
`sort`	`ASC`, `DESC`, `NONE`	Sortierungsrichtung.
`sortLevel`	int: **0**	Priorität bei Mehrfachsortierung.
`rendererClass`	String	Klassenname eines eigenen CellRenderers.

headerRenderer und renderer

Überschreibt die Zeichen-Logik via Groovy (instanziiert neue `FTableScriptedColumnRenderer`).

```
<Column property="Enabled">
   <renderer>
      import java.awt.Color

renderer.setHorizontalAlignment(javax.swing.SwingConstants.CENTER
)
      if (!value) {
         renderer.setBackground((row&amp;amp;1) == 0 ? new Color(234,
176, 176) : new Color(240, 200, 200))
         renderer.setText('\u2717')
      } else {
         renderer.setBackground((row&amp;amp;1) == 0 ? new Color(199,
234, 176) : new Color(207, 226, 186))
         renderer.setText('\u2714')
      }
   </renderer>
</Column>
```

16.27.2. DetailView

Der angedockte Detailbereich einer Tabelle.

Wird die DetailView in einem Lesezeichen verwendet, muss in der `<Table>` das Attribut `viewOnly="true"` gesetzt sein. Für die Änderung von Attributen aus Relationen müssen `onAfterSetValue`-Hooks implementiert werden, um die Tabelle erzwungen zu aktualisieren.

```
<Table property="Buecher" columns="Titel | Erscheinungsjahr |
Autor.Familienname">
  <DetailView name="autor">
    <Border etched="true" title="Details zum Autor">
      <View scrollable="true">
        <Element label="Familienname">
          <Text property="Autor.Familienname">
            <onAfterSetValue
language="groovy">ftx['autor'].getBO().bumpVersion()</onAfterSetV
alue>
          </Text>
        </Element>
      </View>
    </Border>
  </DetailView>
</Table>
```

Name	Erlaubte Werte	Beschreibung
`adjustableSplit`	Boolean: true, **false**	Erlaubt das Verschieben der Trennlinie.
`position`	String: `NORTH`, **`SOUTH`**, `WEST`, `EAST`	Platzierung an der Tabelle.
`resizeWeight`	Float: **0.0**	Gewichtung bei Größenänderung.
`scrollable`	Boolean: true, false; Oder Richtungsbeschränkung, z. B. scrollable="VERTICAL_ONLY"	Scrollbar für dieses Element ein- oder ausschalten beziehungsweise auf eine Richtung beschränken.

16.27.3. MultipleChoiceFilterGUI

Spezifisches Widget für Filterkaskaden innerhalb von Tabellen-Abfragen.

Name	Erlaubte Werte	Beschreibung
`preselectIdx`	int	Bestimmt den initial vorselektierten Index im Dropdown.
`sort`	`ASC`, `DESC`, `NONE`	Sortierungsreihenfolge der Optionen.

16.28. Text

Standard-Eingabefeld für alphanumerische Informationen. Erbt alle Attribute und Subelemente von FTextInputComponent.

Name	Erlaubte Werte	Beschreibung
`align`	String	Textausrichtung.
`background`	Farbangabe. Bitte entweder als „#rrggbbaa“ oder „r,g,b,a“, „r g b a“ oder eine Farbkonstante der java.awt.Color, z. B. YELLOW angeben. Farbnamen mit Postfix „ISH“ werden in Richtung Weiß verschoben (Mittelwert der einzelnen Farbwerte und 255). Der Alphawert ist optional. Die einzelnen Werte sind bei den beiden letzteren Varianten entweder Float-Werte von 0.0..1.0 (bei 1 bitte 1.0 angeben!) oder Integer-Werte von 0..255. Bitte nur die eine Sorte Werte verwenden. Oder für Random-Farbe: random.	Hintergrundfarbe.

Name	Erlaubte Werte	Beschreibung
`class`	String	Eigene Implementierungsklasse (z. B. `ITextArea` für Mehrzeiligkeit).
`disabled`	Boolean: true, **false**	Sperrt das Feld und graut es aus.
`disguiseAsLabel`	Boolean: true, **false**	Ahmt ein einfaches Label nach (erlaubt Formatierung).
`font`	String	Schriftart.
`foreground`	Farbangabe. Bitte entweder als „#rrggbbaa“ oder „r,g,b,a“, „r g b a“ oder eine Farbkonstante der java.awt.Color, z. B. YELLOW angeben. Farbnamen mit Postfix „ISH“ werden in Richtung Weiß verschoben (Mittelwert der einzelnen Farbwerte und 255). Der Alphawert ist optional. Die einzelnen Werte sind bei den beiden letzteren Varianten entweder Float-Werte von 0.0..1.0 (bei 1 bitte 1.0 angeben!) oder Integer-Werte von 0..255. Bitte nur die eine Sorte Werte verwenden. Oder für Random-Farbe: random.	Schriftfarbe.
`format`	String	Formatschablone.
`lineWrap`	Boolean: **true**, false	Bricht Text an der Kante um.
`password`	Boolean: true, **false**	Maskiert die Eingabe als Passwort.
`roundingFormat`	String	Rundungsverhalten.
`rows`	int: **4**	Zeilenhöhe (`ITextArea`).

Name	Erlaubte Werte	Beschreibung
`selectAllWhenFocused`	Boolean: true, *false*	Wenn das Form-Element den Fokus bekommt, wird der gesamte Inhalt selektiert.
`syncOnWait` / `syncOnWaitDelay`		Ohne Funktion.
`tabSize`	int: **4**	Tabulator-Schrittweite.
`translationAvailable`	DEPRECATED	Ohne Funktion.
`wrapStyleWord`	Boolean: **true**, false	Bricht Text bevorzugt an Whitespace-Grenzen um.

16.29. ToggleButton

Ein interaktiver Umschaltknopf für boolesche Attribute. Erbt alle Attribute und Subelemente von BooleanInputComponent.

Das Attribut `trueText` fungiert semantisch als exakter Alias für `text`. Damit der Text zwischen `trueText` und `falseText` wechselt, muss an der Action `initialState` gesetzt und im `onAction`-Skript `action.setSelectedState(bool)` aufgerufen werden.

Name	Erlaubte Werte	Beschreibung
`action`	String	`cmd`-Attribut der verknüpften Action.
`background`	Farbangabe. #rrggbbaa, r,g,b,a, r g b a. Alpha optional. Beispiele: #14f900, 255,255,0,0, 0.5 0.4 0.3 0.2, GREEN, YELLOWISH	Hintergrundfarbe.
`foreground`	Farbangabe. #rrggbbaa, r,g,b,a, r g b a. Alpha optional. Beispiele: #14f900, 255,255,0,0, 0.5 0.4 0.3 0.2, GREEN, YELLOWISH	Schriftfarbe.

Name	Erlaubte Werte	Beschreibung
`selectedColor`	Farbangabe. #rrggbbaa, r,g,b,a, r g b a. Alpha optional. Beispiele: #14f900, 255,255,0,0, 0.5 0.4 0.3 0.2, GREEN, YELLOWISH	Hintergrundfarbe im gedrückten Zustand (Schattierung).
`fontSize`	String: +X%	Gibt an, um wie viel Prozent die Schrift vergrößert werden soll.
`fontStyle`	String: z. B. fontStyle="bold", fontStyle="italics" oder fontStyle="BOLD", fontStyle="italics"	String: z. B. fontStyle="bold", fontStyle="italics" oder fontStyle="BOLD", fontStyle="italics"
`hAlign` / `vAlign`	String	Ausrichtungen auf der Schaltfläche.
`multiClickThreshold`	Integer (**750ms**)	Klick-Schwellenwert zur Spam-Vermeidung.
`trueIcon` / `falseIcon`	String: z. Bsp. icon="20x20/New.gif", icon="image/remove_red_eye.svg" oder icon="image/remove_red_eye.svg@5085dc" (mit Farbangabe in Hex; nur für SVGs verfügbar)	Zustandsspezifische Icons.
`text` / `trueText` / `falseText`	String	Zustandsspezifische Beschriftungen.

16.30. Tree

Hierarchischer Auswahlbaum zur Verknüpfung von relationalen Verzeichnisstrukturen. Erbt alle Attribute und Subelemente von FInputPanel.

Name	Erlaubte Werte	Beschreibung
`childrenProperty`	String	Name des Attributs für Kinder-Objekte.
`displayClass`	DEPRECATED	Ohne Funktion.
`displayProperty`	DEPRECATED	siehe `property`

Name	Erlaubte Werte	Beschreibung
`entity`	String	Die Entität, deren Instanzen im Baum dargestellt werden.
`filter`	String	Derzeit vom Tree nicht unterstützt.
`format`	String	Das Attribut für die Beschriftung der Knoten (Default: `ui description`).
`freeSearch`	Boolean: **true**, false	Ohne Funktion.
`height` / `width`	Integer	Dimensionen des Baums.
`parentProperty`	String	Name des Attributs für Eltern-Objekte.
`restrictToEntity`	String	Schränkt die Auswahl auf Subtypen ein.
`usePolymorphySelectionTree`	Boolean: true, **false**	Aktiviert die polymorphe Baumauswahl.

16.31. Uri

Eingabefeld zur Erfassung und Validierung von Web-Links (URLs). Erbt alle Attribute und Subelemente von FTextInputComponent.

Name	Erlaubte Werte	Beschreibung
`autoaddProtocol`	Boolean: **true**, false	Ergänzt fehlende Protokolle (z. B. `http://`) automatisch.

16.32. View

Unsichtbarer Layout-Basiscontainer, der Kind-Elemente in einem Grid-Raster anordnet. Erbt alle Attribute und Subelemente von FPanel.

Name	Erlaubte Werte	Beschreibung
`autoHideElements`	Boolean: **true**, false	Blendet Elemente aus, falls deren Properties nicht auflösbar sind.
`border`	String	Optionaler Rahmen-Stil.

Name	Erlaubte Werte	Beschreibung
`columns`	Integer	Teilt den Inhalt auf die angegebene Anzahl Spalten auf.
`conflictPolicy`	String: **`IGNORE`**, `ASK`	(Nur im Root) Konfliktbehandlung bei parallelen Speicherungen (`IGNORE`: Ignoriere Konflikte und übernehme/merge Änderungen, `ASK`: Prüfe auf Konflikte und frage Benutzer).
`defaultRightFill`	Double (**1**)	Prozentuale Streckung der umschlossenen Felder.
`delegateToParent`	Boolean: true, **false**	Übernimmt Spalten-Einstellungen der übergeordneten View.
`externalHGap` / `externalVGap`	Integer (px, c)	Äußerer Abstand zum View-Rand.
`height` / `width`	Integer	Dimensionen der View.
`hideElementsForNullBO`	Boolean: **true**, false	Blendet Kindelemente aus, wenn das Formular-BO `null` ist.
`ignoreOtherLocalTransactionSaves`	Boolean: true, **false**	(Nur im Root) Ignoriert lokale Fremdspeicherungen.
`internalHGap` / `internalVGap`	Integer (px, c)	Innerer Abstand zwischen den Zeilen und Spalten.
`tint`	<Farbwert>[@<Intensität>], Intensität ist default 0.1; Beispiele: tint="#ff0000 @ 0.1"; tint="GREENISH"	Färbt den Hintergrund der View ein.
`useMaximumHeight` / `useMaximumWidth`	Boolean: true, **false**	Streckt die View maximal im Container.

Chapter 17. Developer Reference, Best Practices & Syntax

Hier sind die verbindlichen, systemweiten Entwickler-Richtlinien für MyTISM.

17.1. MyTISM Best Practices

Die Softwareentwicklung im MyTISM-Umfeld erfordert ein Höchstmaß an Qualität, Zuverlässigkeit und Wartbarkeit. Dieser Abschnitt definiert die verbindlichen Best Practices für alle MyTISM-Entwicklerinnen und -Entwickler. Er zeigt bewährte Architektur-Ansätze auf, die zwingend dazu beitragen sollen, effizienten und extrem robusten Code zu schreiben. Von der Planung über die Gestaltung bis hin zur Implementierung bilden diese Vorgaben das Fundament unserer Systementwicklung. Diese Best Practices sind das Ergebnis langjähriger Erfahrung und sollen dabei helfen, Code zu produzieren, der den höchsten Anforderungen der modernen Softwarearchitektur gerecht wird.

Ergänzend zu diesem internen Regelwerk wird die Lektüre des Buchs Effective Java, 3rd Ed. dringend empfohlen.

17.1.1. Programmierung

Grundlegendes

Grundprinzipien der objektorientierten Programmierung

Bitte beim Programmieren diese fünf Prinzipien (**SOLID**) beachten:

- **Single-Responsibility-Prinzip** oder **Prinzip der eindeutigen Verantwortlichkeit**, kurz SRP: Jede Klasse hat nur eine fest definierte Aufgabe zu erfüllen. In einer Klasse sollten lediglich Funktionen vorhanden sein, die direkt zur Erfüllung dieser Aufgabe beitragen.
- **Open-Closed-Prinzip** oder **Prinzip der Offen- und Verschlossenheit**, kurz OCP: Software-Einheiten (Module, Klassen, Methoden) sollten sowohl offen (für Erweiterungen) als auch verschlossen (für Modifikationen) sein. Eine Erweiterung im Sinne des Open-Closed-Prinzips ist beispielsweise die Vererbung. Diese verändert das vorhandene Verhalten der Einheit nicht, erweitert aber die Einheit um zusätzliche Funktionen oder Daten. Überschriebene Methoden verändern auch nicht das Verhalten der Basisklasse, sondern nur das der abgeleiteten Klasse. Folgt man darüber hinaus dem Liskovschen Substitutionsprinzip, verändern auch überschriebene Methoden nicht das Verhalten, sondern nur die Algorithmen.
- **Liskovsches Substitutionsprinzip** oder **Ersetzbarkeitsprinzip**, kurz LSP: Ein Programm, das Objekte einer Basisklasse T verwendet, muss auch mit Objekten der

davon abgeleiteten Klasse S korrekt funktionieren, ohne dabei das Programm zu verändern.

- **Interface-Segregation-Prinzip** oder **Schnittstellenaufteilungsprinzip**: Zu große Schnittstellen sollen in mehrere Schnittstellen aufgeteilt werden, falls implementierende Klassen unnötige Methoden haben müssen. Nach erfolgreicher Anwendung dieses Entwurfprinzips würde ein Modul, das eine Schnittstelle benutzt, nur die Methoden implementieren müssen, die es auch wirklich braucht.
- **Dependency-Inversion-Prinzip** oder **Abhängigkeits-Umkehr-Prinzip**, kurz DIP: Module höherer Ebenen sollten nicht von Modulen niedrigerer Ebenen abhängen. Beide sollten von Abstraktionen abhängen. Abstraktionen sollten nicht von Details abhängen. Details sollten von Abstraktionen abhängen.

12 Goldene Regeln für das Refactoring von Java-Code

Hier sind 12 essenzielle Regeln, die beim Refactoring von Java-Code beachtet werden sollten, um die Wartbarkeit und Qualität zu verbessern:

1. **Understand Before You Refactor**:
   Bevor unverstandener Code geändert wird, muss dieser gründlich gelesen und der Ausführungsfluss nachvollzogen werden. Alle Verwendungen (`Strg + Alt + H` in IntelliJ, im Zweifel anschl. zusätzlich Volltextsuche) müssen identifiziert werden, um unerwartete Fehler zu vermeiden. Vorsicht ist angebracht: „Don't touch what you don't understand."
2. **Start with Unit Tests, Not Code**:
   Bevor Refactoring-Maßnahmen ergriffen werden, sollte eine ausreichende Testabdeckung des Codes sichergestellt sein, um dessen Verhalten zu validieren, selbst wenn es nur grundlegende Unit-Tests sind. Bei Legacy-Code sind Charakterisierungstests zur Verifizierung des aktuellen Verhaltens oder Approval-Tests zur Erfassung und zum Vergleich von Ausgaben nach Änderungen besonders empfehlenswert. Ohne Vertrauen in die Tests („No tests? No trust.") sollte im Idealfall kein Refactoring vorgenommen werden.
3. **Rename Like Your Life Depends On It**:
   Klarheit in der Benennung hat oberste Priorität. Namen sollten präzise den Zweck einer Funktion widerspiegeln. Beispielsweise macht die Umbenennung von `process()` in `validateAndDispatchInvoice()` die Funktion sofort ersichtlich.
4. **Extract Methods Aggressively**:
   Lange Methoden sollten konsequent in kleinere, überschaubare Einheiten zerlegt werden. Die Funktion „Extract Method" (`Strg+Alt+M` in IntelliJ, leider wird NetRexx nicht unterstützt) ist ein mächtiges Werkzeug, um die Lesbarkeit und Wartbarkeit des Codes erheblich zu verbessern.
5. **Follow the „Boy Scout Rule"**:

Der Code sollte stets sauberer hinterlassen werden, als er vorgefunden wurde. Dies bedeutet, bei jeder Berührung einer Datei kleine Verbesserungen vorzunehmen, wie das Umbenennen von Variablen, das Hinzufügen von Kommentaren oder das Entfernen veralteter TODOs. Es ist jedoch ratsam, diese kleinen Verbesserungen in einem separaten Commit festzuhalten, um sie von der eigentlichen Funktionalitätsänderung zu trennen und die Nachvollziehbarkeit des Haupt-Commits zu gewährleisten.

6. **Break Dependencies Mercilessly**:
   Starke Kopplungen im Code, insbesondere in Legacy-Systemen mit globalem Zustand, statischen Aufrufen oder zirkulären Referenzen, müssen reduziert werden. Die Einführung von Interfaces ist ein effektives Mittel, um Abhängigkeiten zu verwalten und die Flexibilität zu erhöhen.

7. **Write Logs That Speak Human**:
   Während des Refactorings sollte das Logging optimiert werden, um Fehler aussagekräftiger zu gestalten und Kontextinformationen zu liefern, anstatt generischer Meldungen wie „Error in Service“. Eine klares Logging ist entscheidend für die spätere Fehlersuche.

8. **Use the „WTF per Line“ Metric**:
   Code muss für andere Entwickler sofort verständlich sein, ohne zusätzliche mündliche Rückfragen oder Erklärungen. Ist dies nicht der Fall, sind aussagekräftige Kommentare und präzise JavaDoc-Dokumentation zwingend erforderlich. Bei extremer Komplexität kann eine Neufassung des Codes notwendig sein. Vor einer Neufassung sollte jedoch Rücksprache gehalten werden, da „kompliziert“ manchmal „optimiert“ bedeutet, was dann ebenfalls entsprechend dokumentiert werden muss.

9. **Don’t Just Delete - Replace with a Warning**:
   Alte, scheinbar nicht mehr genutzte Konstanten, Methoden oder Klassen sollten niemals einfach so direkt entfernt werden. Stattdessen ist es bewährte Praxis, sie zunächst mit der Annotation `@Deprecated` zu kennzeichnen. Eine zusätzlich in den Logs ausgegebene Warnung beim Aufruf von veralteten Funktionen hilft maßgeblich dabei, verbleibende Abhängigkeiten und letzte Verwendungen in Produktionssystemen frühzeitig zu erkennen und abzufangen, bevor der Code entfernt wird. Dies ermöglicht einen kontrollierten Übergang und minimiert das Risiko unerwarteter Fehlfunktionen.

10. **Re-review Your Own PRs (Pull Requests) After a Coffee Break**:
    Eine Überprüfung des eigenen Codes nach einer kurzen Pause, mit „frischen Augen“, hilft, Tippfehler, unnötige Komplexität oder übertriebenes Over-Engineering zu erkennen und zu korrigieren.

11. **Refactor One Responsibility at a Time**:
    Dienste mit mehreren Verantwortlichkeiten (z. B. Datenbankzugriffe, API-Aufrufe, XML-Erstellung, Benachrichtigungsversand) sollten in separate, spezialisierte Dienste aufgeteilt und einzeln refaktoriert werden. Das Entwirren von „Spaghetti-Code“ sollte dabei schrittweise erfolgen, eine „Nudel“ nach der anderen. ;-)

12. **If It Ain’t Broke, Don’t Rewrite It**:
    Ein vollständiges Neuschreiben von Code („Big Rewrite“) ist riskant und nur bei gravierenden, unbehebbaren Bugs oder massiver Entwicklungsblockade gerechtfertigt. Selbst dann sind eine umfassende Testabdeckung sowie ausreichend Zeit für die Implementierung, das Debugging und den späteren Support, der sich durch potenziell eingeschlichene neue Fehler ergeben könnte, absolute Voraussetzungen. Andernfalls ist es besser, das „Monster“ (den problematischen Code) zu isolieren oder durch gezieltes Refactoring mit einer neuen, sauberen Schicht zu umhüllen, anstatt ein potenziell noch größeres Monster (Problem) zu schaffen.

Diese Liste wurde inspieriert durch einen Blog post von Kavya.

Statische Methoden

Statische Methoden werden oft als "Abkürzung" genutzt, sind aber im Kontext von sauberer Softwarearchitektur kritisch zu betrachten, da sie harte Kopplung erzwingen.

Das Problem der Kopplung

Ein statischer Aufruf wie `UserService.isValid(user)` bindet den Aufrufer untrennbar an die konkrete Klasse `UserService`. Dies verhindert den Einsatz von Interfaces und verstößt gegen das Dependency Inversion Principle.

Testbarkeit

Statische Methoden sind "Hard-Wired". Während Instanzmethoden via Dependency Injection leicht durch Mocks ersetzt werden können, erfordern statische Methoden komplexe Bytecode-Manipulation (z.B. PowerMock), was die Test-Suite instabil und langsam macht.

Mythos Compiler-Performance (Inlining)

Es wird oft argumentiert, dass static schneller sei, da der Aufruf via invokestatic keinen Vtable-Lookup benötigt. **Realität:** Moderne JIT-Compiler (Just-In-Time) nutzen Class Hierarchy Analysis (CHA). Wenn eine Instanzmethode nicht überschrieben wird, "devirtualisiert" der Compiler den Aufruf automatisch und führt das gleiche Inlining durch wie bei statischen Methoden. Der Performance-Vorteil von static ist in 99% der Fälle vernachlässigbar.

Wann static dennoch erlaubt ist

- Utility-Funktionen: Rein mathematische oder transformierende Funktionen ohne Seiteneffekte (z. B. `Math.max()`, `StringUtils.isEmpty()`).
- Factory-Methoden: Zur Erzeugung von Instanzen (z. B. `List.of()`, `Optional.of()`).

- Statische Interface-Methoden (Java 8+): Wenn eine Hilfsfunktion logisch untrennbar zum Interface gehört, aber keinen Objektzustand benötigt.

Goldene Regel für statische Methoden

Wenn eine Methode als Instanzmethode implementiert werden kann, dann tue es auch. Statisch sollte nur sein, was absolut zustandslos ist und niemals durch eine andere Logik ersetzt werden muss.

Hardcoding vs. eigenschaftsbasierter Ansatz

Das Antipattern, bei dem Code auf Daten in der Datenbank fest verdrahtet ist, wird als „Hardcoding“ bezeichnet. Es wird im Allgemeinen nicht empfohlen, sich auf bestimmte feste Werte in der Datenbank zu verlassen. Stattdessen sollten alternative Mechanismen wie eine Konfigurationsdatei, eine *.initialdata.xml-Datei oder ähnliche Ansätze genutzt werden, um diese Daten zu speichern und im Code zu verwenden.

Wenn Entscheidungen aufgrund des Zustands in der Datenbank getroffen werden müssen, sollten diese niemals auf der Grundlage spezifischer Daten wie dem Namen, der Id oder der Tid eines Objekts erfolgen. Stattdessen sollte ein „eigenschaftsbasierter“ Ansatz („property based approach“) verfolgt werden. Das bedeutet, Entscheidungen sollten auf allgemeinen Eigenschaften, Merkmalen oder erfüllten Bedingungen des Objekts beruhen, da dieser Ansatz flexibler und anpassungsfähiger ist.

Durch die Anwendung dieses Ansatzes kann generischer und wiederverwendbarer Code erstellt werden, der eine breite Palette von Eingaben und Szenarien verarbeiten kann. Außerdem ist solcher Code leichter wartbar und testbar.

System zum Markieren von Todos

Kommentare, die mit dem Schlüsselwort **FIXME** eingeleitet werden, können verwendet werden um Ideen zu notieren, die den Code verbessern. Diese sollten auch verwendet werden, um noch zu erledigende Sonderfälle, u.ä., zu markieren und zu beschreiben.

Ein solcher Kommentar beginnt mit **FIXME**, gefolgt vom aktuellen Datum und Namenskürzel, als Präambel. Die Reihenfolge zwischen Datum und Kürzel ist frei. Für das Datum wird das ISO Format bevorzugt (e.g. 2023-02-13).

Mehrzeilige FIXMEs halten sich an die Einrückungsvorschriften von normalen Kommentaren. Dadurch lassen sich mehrere Kommentare für die gleiche Codestelle leicht voneinander unterscheiden. Zudem erkennen und highlighten verschiedene Editoren solche Kommentare, was deren Lesbarkeit fördert.

FIXMEs sollten möglichst nur in einer eigenen Zeile vor dem betroffenen Code stehen, damit die CVS Historie für diese Zeile(n) nicht unnötig überladen wird, was Recherchen

unnötig erschwert. Zudem lassen sich so einfacher weitere FIXMEs und Antwortkommentare hinzufügen.

Relevanz eines FIXMEs

Das früher benutzte, etwas saloppe System aus `FIXME` für wichtige/große Sachen und `FIXMAY` für kleinere Sachen wird heute nicht mehr verwendet, findet sich jedoch noch in älterem Code. Wir verwenden inzwischen ein etwas strukturierteres und abgestufteres System, das sich aus dem Schlüsselwort `FIXME` und einer Anzahl von Ausrufezeichen zusammensetzt:

- FIXME!!! - Das muss unbedingt noch gefixt/gemacht/ergänzt werden, sonst funktioniert gar nichts. Darf eigentlich nur vorkommen während man noch am Entwickeln ist.
- FIXME!! - Da läuft in vielen Fällen noch was schief oder gar nicht bzw. ist sehr suboptimal gelöst; muss i.d.R. noch verbessert werden, bevor der Code ausgeliefert werden kann.
- FIXME! - Sollte noch behandelt werden; z. B. ein eher selten auftretender Fall, der noch nicht funktioniert, wenn er denn mal auftritt. Kommt auf Projekt/Situation an, ob man das erstmal so lässt oder nicht :-)
- FIXME - Könnte man mal was dran machen; funktioniert zwar immer, ist aber nicht besonders schön oder etwas unperformant/suboptimal gelöst; l10n-Todos; Kosmetik; etc.

Vorteile gegenüber dem alten System:

- Man hat **immer** das Schlüsselwort `FIXME`, welches wohl von einigen Tools (z. B. CVSspam) erkannt und benutzt wird.
- Die Schreibweise ist recht intuitiv. Auch, wenn man nichts von diesem System weiß, erkennt man selbst als Außenstehender leicht, wie wichtig oder weniger wichtig ein FIXME ist.
- Man kann auch mit einer einfachen Textsuche immer genau die Stellen finden, die man haben will; also z. B. nur sehr dringende/wichtige Todos (dann sucht man z. B. nach „FIXME!! “) oder auch weniger wichtige Sachen (suchen nach „FIXME! “) oder einfach alles (suchen nach „FIXME “).

Zusätzlich muss jedes FIXME auch mit dem Benutzerkürzel und einem Timestamp ausgestattet werden, z.B. *FIXME! TH, 2016-11-25: Ein Beispieltext.*

Vorteile:

- Nachfragen können zielgerichtet an die richtige Person gerichtet werden.
- Es ist ohne lange Recherche in der CVS-History ersichtlich, wer das FIXME erstellt hat

und seit wann es besteht.

Es ist vorteilhaft wenn längere FIXME-Texte eine einleitende Zeile nach obigem Format, quasi eine Titelzeile haben, und der weiterführende Text um eine Stufe eingerückt darunter notiert wird, mit entsprechenden Umbrüchen nach ca. 60 Zeichen. So bekommt man beim Scannen des Codes schnell einen Eindruck über das FIXME und kann die Details bei Bedarf lesen oder einfach überlesen.

Beispiel:

```
   /* FIXME! TH, 2020-07-21: Refactor this to avoid code
duplication with method doMagic()
         The code duplication here is currently necessary due to
limitations in NetRexx.
         Once we have a cross-compiler, this can and should be
refactored. */
   method jumpThroughHoops()
```

Namenskonventionen für Metadaten und Tids

Die Tid dient als textueller Identifikator zusätzlich zur globalen Id und wird häufig direkt im Code referenziert.

Für den Inhalt von neu hinzugefügten Tids und vergleichbaren Metadaten gelten strikte Konventionen, um eine einheitliche Datenbasis zu garantieren:

- Bezeichner müssen zwingend in englischer Sprache verfasst sein.
- Es ist ausschließlich Großschreibung (CAPS) ohne Umlaute zu verwenden.
- Sonderzeichen sind untersagt; es dürfen nur alphanumerische Zeichen (A-Z, 0-9) genutzt werden.
- Leerzeichen müssen konsequent durch Unterstriche (`_`) ersetzt werden.

Beispiele für korrekte Tid-Werte

- `PREMIUM_CUSTOMER`
- `STANDARD_TARIFF`
- `MAJOR_CUSTOMER`

Diese Konvention stellt die Konsistenz zwischen Datenbankinhalt und Quellcode-Konstanten sicher und minimiert Fehler bei String-Vergleichen.

Exceptions

Fehlermeldungen müssen sachlich formuliert werden und den Fehlerfall verständlich für den normalen Benutzer darlegen. In Fehlermeldungen muss immer auch das Objekt genannt werden, das den Fehler ausgelöst hat, da die Fehlermeldung im Log sonst oft wertlos ist. Es dürfen keine Ausrufezeichen oder Fragezeichen-Ausrufezeichen-Kombinationen verwendet werden, da diese den Benutzer „anschreien“. Fragezeichen sind nur für Benutzerinteraktionen zu verwenden und nicht in Fehlermeldungen; eine Ausnahme stellt der Zusatz „Meinten Sie vielleicht…?“ oder ähnliche Formulierungen dar.

Fehlermeldungen sollten grundsätzlich über das Lokalisierungssystem (L10n) in Englisch, Deutsch und idealerweise auch in Französisch bereitgestellt werden. Bitte darauf achten, vollständige Sätze zu übersetzen, anstatt einzelne Satzteile oder Wörter. Die Grammatik kann in verschiedenen Sprachfamilien erheblich von der Grammatik germanischer oder romanischer Sprachen abweichen, sodass das Zusammensetzen von Satzteilen nicht immer funktioniert.

Statt einer RuntimeException soll die de.ipcon.tools.IRuntimeException verwendet werden. Bei der Ausgabe des Stacktrace von IRuntimeExceptions werden „unnütze“ Stackframes ausgefiltert. Beispielsweise fallen interne Reflection-Aufrufe und Groovy-Hilfsmethoden weg (siehe auch Commit vom 19.02.2015 08:29). Die resultierenden Stacktraces sind kürzer und lesbarer und beinhalten die wirklich interessanten Informationen.

Fehlermeldungen, die in der GUI angezeigt werden, sollten auch für weniger technisch versierte Benutzer verständlich sein. Technische Details sollten besser ins Log geschrieben werden, während dem Benutzer ein allgemeinerer Text mit der Bitte um Meldung an uns angezeigt wird.

Konstanten

Vermeidung von „Constant Interfaces“

In Java besteht die Möglichkeit, Konstanten sowohl in Interfaces als auch in Klassen zu definieren. Eine frühere Praxis bestand darin, manche Interfaces ausschließlich als Container für Konstanten zu verwenden, die dort standardmäßig `public static final` sind. Dies ist als „Constant Interface Antipattern“ bekannt (siehe „Effective Java, 3rd Ed., Item 22, p. 107“ und Wikipedia) und weist einige Nachteile auf.

Wenn man Konstanten exportieren will, gibt es mehrere sinnvolle Möglichkeiten:

- Wenn die Konstanten eng mit einer vorhandenen Klasse oder einem Interface verknüpft sind, sollten sie direkt in der entsprechenden Klasse oder dem Interface definiert werden.
- Wenn die Konstanten am besten als Elemente eines Enum-Typs zu betrachten sind, dann sollten sie mit einem Enum-Typ exportiert werden. Enums sind in Java typsicher,

effizient und erweiterbar.
Enums müssen in Java definiert werden, da NetRexx zwar die Verwendung von Enums unterstützt, jedoch nicht deren Definition.

- Andernfalls sollten die Konstanten in einer Hilfsklasse exportiert werden, die als `public final` markiert ist und über einen privaten Konstruktor verfügt. Auf diese Weise wird sichergestellt, dass die Hilfsklasse nicht instanziiert werden kann.
  Ein Beispiel dafür findet sich im Core in der Klasse `de.ipcon.messaging.email.MIMECharsets`.

Interfaces sollten nur für die Definition von Typen verwendet werden und nicht lediglich dem Export von Konstanten dienen.

Weitere Patterns und Anti-Patterns für Konstanten

Gute Artikel mit Patterns und zu vermeidenden Anti-Patterns in Bezug auf Konstanten findet man auf baeldung.com oder in diesem Artikel von Rakesh Prajapati, sowie in diesem Stack Overflow Thread.

Im Folgenden werden einige wichtige Patterns und Anti-Patterns aufgezählt.

Patterns für Konstanten

- **Konstantenklassen:** Das Gruppieren verwandter Konstanten in einer eigenen Klasse fördert die Organisation und Übersichtlichkeit.
- **`constant` (Java: `final static`) Schlüsselwort:** Die Verwendung von `constant` bzw. in Java `final static` stellt sicher, dass der Wert einer Konstante nach der Initialisierung nicht mehr verändert werden kann und Konstanten ohne Instanziierung der Klasse direkt über den Klassennamen angesprochen werden können.
- **Zugriffsmodifikatoren:** Die Sichtbarkeit von Konstanten sollte durch geeignete Zugriffsmodifikatoren (`public`, `private`, `inheritable` (bzw. in Java `protected`) kontrolliert werden.
- **Benennungskonventionen:** Konstantennamen sollten aussagekräftig sein und in Großbuchstaben mit Unterstrichen zur Trennung von Wörtern geschrieben werden (z.B. `MAX_VALUE`, `DEFAULT_TIMEOUT`).

Anti-Patterns für Konstanten

- **„Magische Zahlen“:** Das Verwenden von literalen Zahlenwerten im Code ohne Erklärung erschwert die Lesbarkeit und Wartbarkeit. Stattdessen sollten Konstanten mit aussagekräftigen Namen verwendet werden.
- **Konstanten-Interfaces:** Das Definieren von Konstanten in Interfaces kann zu Namensraumkollisionen und Verwirrung führen. Es ist besser, Konstanten in Klassen

oder Enums zu kapseln.

- **Globale Konstanten:** Übermäßiger Gebrauch globaler Konstanten kann die Modularität und Testbarkeit des Codes beeinträchtigen. Konstanten sollten möglichst nah an ihrer Verwendungsstelle definiert werden.
- **Unveränderliche Collections als Konstanten:** Das Deklarieren einer unveränderlichen Collection als Konstante verhindert nicht, dass die Elemente innerhalb der Collection verändert werden.
- **`null` als Konstantenwert:** Die Verwendung von `null` als Wert einer Konstante kann zu Verwirrung und unerwarteten `NullPointerExceptions` führen. Es ist besser, spezielle „leere“ Objekte oder, noch besser, Enums mit expliziten Werten zu verwenden, um das Fehlen eines Wertes darzustellen. Enums bieten zusätzliche Typsicherheit.

Datentypen

Strings

Vergleiche von Tids mit konstanten Strings

Vergleiche von Tids eines Objekts mit konstanten String-Codes sind im Code oftmals unnötig kompliziert formuliert. Es reicht, die Konstante auf die linke Seite zu stellen und mit „==“ den strict String check von NetRexx zu verwenden. Dadurch entfällt die Möglichkeit einer NPE und man muss außerdem nicht die (teurere) NN-Methode auf dem Objekt rufen.

Beispiel:

```
   if getTidNN().equals(TID_BLA) then
      return
```

wird zu:

```
   if TID_BLA == getTid() then
      return
```

Queries und L10n-Schlüssel in Konstanten ablegen

Das Ablegen von Queries und Lokalisierungsschlüssel in Konstanten wird empfohlen und bietet mehrere Vorteile:

- **Schneller Zugriff und Wartbarkeit:** Das Hinterlegen von hartkodierten Werten als Konstanten am Anfang einer Klasse oder an einem zentralen Ort ermöglicht einen schnellen Zugriff und erleichtert die Wartung des Codes. Falls sich diese Werte ändern,

müssen sie nur an einer Stelle aktualisiert werden, was die Wahrscheinlichkeit von Fehlern reduziert.

- **Dokumentation:** Konstanten können mit erklärenden Kommentaren oder Java-Dokumentationen versehen werden, was die Verständlichkeit des Codes erheblich verbessert. Teammitglieder können leicht nachvollziehen, wofür die Konstanten verwendet werden, und wie sie sich auf die Funktionalität des Programms auswirken.

Queries

Die Zusammenfassung von Queries an einer zentralen Stelle bietet weitere Vorteile:

- **Wiederverwendbarkeit und Mustererkennung:** Das Sammeln aller Queries einer Klasse an einer Stelle ermöglicht die einfache Wiederverwendung von Teilen von Queries und das Erkennen von ähnlichen oder sich wiederholenden Mustern. Dies fördert die Effizienz bei der Erstellung und Wartung von Queries.
- **Fehlererkennung:** Fehler in Queries können leichter entdeckt werden, da sie sich direkt untereinander befinden. Abweichungen von bestimmten Mustern oder fehlende Klauseln sind einfacher zu erkennen, was die Fehlererkennung und -behebung erleichtert.
- **Verbesserte Testbarkeit:** Die Verwendung von Konstanten für Queries erleichtert die Integration von Queries in Unit-Tests und Query-Interceptors. Die Queries sind leicht zugänglich und können effektiv dokumentiert werden, was die Testbarkeit und Qualitätssicherung des Codes verbessert.

L10n-Schlüssel

Die Verwendung von Konstanten für Lokalisierungsschlüssel (L10n-Schlüssel) bietet folgenden weiteren Vorteil:

- **Abgleich mit Ressourcendateien:** Die Verwendung von Konstanten erleichtert den Abgleich der im Code verwendeten L10n-Schlüssel mit den entsprechenden Ressourcendateien für Übersetzungen. Dies fördert die Konsistenz und ermöglicht eine einfachere Aktualisierung von Übersetzungen, da es leichter wird, die benötigten Schlüssel in den Übersetzungsdateien zu finden und sicherzustellen, dass sie korrekt und vollständig sind.

Zusammenfassend führt die Verwendung von Konstanten für L10n-Schlüssel zu einer besseren Wartbarkeit, Fehlererkennung und Testbarkeit des Codes. Sie verhindert das Aufblähen des Codes durch riesige Inline-Strings oder kryptische Zeichenfolgen und erleichtert den Umgang mit Übersetzungen, was insgesamt die Qualität und Zuverlässigkeit des Codes verbessert.

Test auf leeren bzw. nicht leeren String

NetRexx

In NetRexx wurde im Code oft ein Vergleich der folgenden Art verwendet:

```
if s1 = '' then
   iterate
if s1 = '' then
   doMagic(s1)
if s2 \= '' then
   iterate
if s2 <> '' then
   doSomeMoreMagic(s2)
```

Generell ist es keine gute Idee solche Vergleiche durchzuführen, denn Strings haben einen Längen-Zähler, der von der Methode `isEmpty()` direkt auf Gleichheit mit *0* getestet wird, was schneller und effizienter ist als ein Vergleich von zwei Strings via equals; vor allem in Hotspots kann dies zu Geschwindigkeitsverbesserungen führen. Außerdem führt NetRexx bei der Variante mit `=` bzw. `<>` auch noch einen case-insensitive Vergleich durch und entfernt umschließenden Whitespace implizit, was dem Programmierer evtl. nicht immer bewusst war bei der ursprünglichen Programmierung und zusätzlichen Aufwand bei der Ausführung bedeutet. Während des Refactoring sollte überlegt werden, ob ein trimmen nötig ist und vorher vergessen wurde oder umgekehrt.

Nach einem Refactoring würde der oben stehende Code dann so aussehen:

```
if s1.isEmpty() then
   iterate
if s1.trim().isEmpty() then
   doMagic(s1)
if not s2.isEmpty() then
   iterate
if not s2.trim().isEmpty() then
   doSomeMoreMagic(s2)
```

Die Codebase von core und Modulen wurde bereits umgestellt.

Bei zukünftiger Entwicklung bitte nur noch die Variante mit `.trim().isEmpty()` verwenden bzw. ggfs. ohne Trimmen. Analog kann die Methode

`de.ipcon.tools.TextTools isNullOrEmpty(…)` verwendet werden, welche die entsprechenden Aufrufe zusammenfasst.

String-Konkatenation

NetRexx

Der de.ipcon.tools.StringBuilderPool gibt für `StringBuilderPool.getInstance()` einen `StringBuilder` zurück. `StringBuilder` sind unsynchronisiert.

Wenn eine konstante Anzahl an Text konkateniert wird, kann der `StringBuilder` auch direkt verwendet werden.

Wird mit Schleifen gearbeitet, sollte ein `StringBuilder` aus dem Pool genommen werden. Der StringBuilderPool hält bereits `StringBuilder` mit größerer interner Datenstruktur vor und vermeidet dadurch das interne Vergrößern, das auf Dauer sehr teuer ist.

Nach dem Konkatenieren werden die genutzten Ressourcen durch `StringBuilderPool.finalToString()` freigegeben. Erfolgt dies nicht, entsteht ein memory leak.

Der Java-native `StringBuffer` kann in der Regel vermieden werden.

Groovy

String-konkatenation in Groovy ohne Plus und mit Interpolation ist vorzuziehen, da es performanter ist:

```
def a = 'ich'
"$a bin ein gutes Beispiel"
a + 'bin Keines'
```

Maps

Umgang mit LazyMapI-s

Im Interface LazyMapI gibt es die statischen Methoden `isNullOrEmpty(Map)`, `isEmpty(Map)` und `size(Map)`. Diese Methoden zu rufen ist in manchen Fällen sinnvoll, in anderen Fällen jedoch eher nachteilig.

Nachteilig ist es, wenn man - nach der Prüfung auf Leerheit oder Abfrage der Größe - eine Operation auf der Map ausführt, die auf jeden Fall ein unlazy auslöst. Dazu gehören u.a. `isEmpty()`, `keySet()`, `entrySet()`, `values()`.

Der Grund dafür ist, dass der `LazyMapI.isEmpty(Map)` call für den Fall „leere Map“ genauso teuer ist wie ein unlazy. Ein anschließendes `values()` z.B. zum Iterieren über eine leere Map tut dann eh nichts mehr und kostet nichts, so dass ein Check und early return hier unnötig ist. Wenn die Map andererseits nicht leer ist, kann man das unlazy (via `Map#size` oder `Map#isEmpty`) in diesen Fällen auch direkt auslösen, weil es sonst zwei Abfragen gäbe: einmal lazy und danach noch einmal unlazy für das Iterieren.

Von Vorteil ist es, wenn man nur die Information benötigt, ob eine Map leer ist oder wie groß sie ist, um z.B. die Anzahl zurückzugeben oder bestimmte Dinge nur für leere oder nicht-leere Maps auszuführen, ohne die Map selbst (immer) zu verwenden (z.B. wenn man raus springt, wenn eine Map nicht leer ist oder diese Information anderweitig nutzt).

Klassen und Methoden

Empfohlene Größen

optional

Soft-Limits für die Länge von Klassen und Methoden sind als hilfreiche Richtlinien gedacht, um Teammitglieder bei der Organisation des Codes in klar strukturierte Einheiten zu unterstützen.

Die empfohlenen Soft-Limits lauten wie folgt:

- maximal eine „Bildschirmseite“ (ca. 70-80 Zeilen) für Methoden
- maximal etwa 500 Zeilen für Klassen

Diese Soft-Limits sind keine strikten Regeln, sondern dienen als Leitlinien. Es kann Situationen geben, in denen es sinnvoll ist, von diesen Grenzen abzuweichen, sofern die Lesbarkeit und Wartbarkeit des Codes dadurch nicht beeinträchtigt werden.

Diese Grenzen sollen als Gedankenanstoß dienen, um mögliche Refactorings in Erwägung zu ziehen, wenn eine Codeeinheit diese Grenzen überschreitet. Das Auslagern von thematisch zusammengehörigem Code in sogenannte „Aspekt-Klassen“ ist z.B. eine bewährte Strategie zur Verbesserung der Code-Struktur und Lesbarkeit. Durch die Aufteilung des Codes in kleinere Einheiten wird dieser modularer, einfacher zu testen und leichter zu warten.

Klassen

Es ist nicht empfehlenswert, mehr als eine Java Klasse in einer Quelldatei zu deklarieren. Die Standardkonvention für Java-Code besagt, dass jede Java-Klasse in einer separaten Quelldatei deklariert werden sollte. Dies hat mehrere Vorteile:

- Es macht den Code leichter zu lesen und zu verstehen, da jede Klasse in ihrem eigenen Kontext steht.

- Es erleichtert die Wartung des Codes, da Änderungen an einer Klasse nur in einer Datei vorgenommen werden müssen.
- Es verbessert die Testbarkeit des Codes, da jede Klasse separat getestet werden kann.

Es gibt jedoch einige Fälle, in denen es sinnvoll sein kann, mehr als eine Java Klasse in einer Quelldatei zu deklarieren. Dazu gehören:

- Private Klassen und Interfaces: Private Klassen und Interfaces sind nicht von anderen Klassen sichtbar und können daher in derselben Quelldatei deklariert werden wie die Klasse, mit der sie verbunden sind.
- Kleine, eng miteinander verbundene Klassen: Wenn zwei Klassen sehr klein sind und eng miteinander verbunden sind, kann es sinnvoll sein, sie in derselben Quelldatei zu deklarieren.
- Sonderfall für NetRexx: Dependent Classes sind eine Möglichkeit, Java-Minor-Klassen zu implementieren. Solch eine Klasse wird in derselben Quelldatei definiert wie die Klasse, von der sie abhängt.

Methoden

Arrow-Anti-Pattern und Early Exit Strategie

Das "Arrow-Anti-Pattern" beschreibt Code-Strukturen, die durch tiefe Verschachtelungen von bedingten Anweisungen eine Pfeilform annehmen. Diese Struktur erhöht die kognitive Last massiv, da beim Lesen ein mentaler Stack mitgeführt werden muss, um den aktuellen Status der Bedingungen zu verfolgen.

Als Best Practice gilt die Anwendung von "Early Exits" mittels "Guard Clauses". Dabei wird die Methode so früh wie möglich verlassen, sobald ein Ergebnis feststeht oder eine Ausnahme (Exception) ausgelöst werden muss. Dies reduziert die Verschachtelungstiefe idealerweise auf eine Ebene ("Flat Code"). Einmal abgehandelte Spezialfälle können gedanklich sofort abgehakt werden, was die Wartbarkeit und Lesbarkeit verbessert.

Beispiel für schwer lesbaren Code (Arrow-Anti-Pattern):

```
   public Integer getSomething() {
      def result
      try {
         def tmp = getMyTempValue()
         if (tmp < 0) {
            result = tmp * 2
         } else {
            def tmp2 = getAnotherValue()
```

```
            if (tmp2 < 5) {
               result = tmp * tmp2 / 5
            } else {
               if (tmp2 > 10) {
                  result = tmp - 25 / 17
               } else {
                  result = tmp2 + 2
               }
            }
         }
      } catch (IllegalArgumentException e) {
         result = -1
      }
      return result
   }
```

Beispiel für optimierten Code mittels Early Exits:

```
   public Integer getSomething() {
      try {
         def tmp = getMyTempValue()
         if (tmp < 0) {
            return tmp * 2
         }
         def tmp2 = getAnotherValue()
         if (tmp2 < 5) {
            return tmp * tmp2 / 5
         }
         if (tmp2 > 10) {
            return tmp - 25 / 17
         }
         return tmp2 + 2
      } catch (IllegalArgumentException e) {
         return -1
      }
   }
```

Warnung vor Fehl-Optimierung im `finally`-Block

Eine Reduzierung der Code-Verschachtelung darf niemals durch ein `return` innerhalb eines `finally`-Blocks erkauft werden. Ein `return` im `finally`-Block fungiert in Java (und Groovy) als absolut vorrangige Anweisung, die den bisherigen Kontrollfluss vollständig überschreibt. Jegliche im `try`- oder `catch`-Block geworfene oder weitergereichte Exception wird durch dieses `return` unwiederbringlich verworfen ("Exception Swallowing"). Technisch erfolgt dies durch das Leeren des internen Exception-Registers des Threads durch die JVM. Dies resultiert in einem "Silent Failure": Die Methode gibt den Kontrollfluss an den Caller (ggfs. inkl. einem regulären Rückgabewert) zurück, während ein kritischer Fehler, der eigentlich zum Programmabbruch oder einer Fehlermeldung in der GUI hätte führen müssen, spurlos verschwindet.

Methoden mit Array, Collection oder Map als Rückgabewert

In der Regel sollten Methoden, die ein Array, eine Collection oder eine Map als Rückgabewert liefern, keinen `null`-Wert zurückgeben, sondern ein Array der Länge Null, eine leere Collection oder eine leere Map. Hier sollten am besten entsprechende Konstanten oder statische Hilfsmethoden wie `Collections#emptyMap()` genutzt werden, da das Bauen neuer leerer Instanzen unnötig Speicher und Performance kostet.

Das Hauptargument für diese Vorgehensweise ist, dass sonst zusätzlicher Code in den Callern erforderlich ist, um den Rückgabewert `null` korrekt zu verarbeiten. Zudem ist die Rückgabe von `null` fehleranfälliger, da die Person, die den Caller schreibt, vergessen könnte, den Code für Sonderfälle zu schreiben, um den `null`-Wert korrekt zu behandeln. Ein solcher Fehler kann jahrelang unbemerkt bleiben, da solche Methoden in den meisten Fällen in der Praxis dann doch ein oder mehrere Objekte zurückgeben.

Ferner sollte man es vermeiden, `null`-Werte in Arrays, Collections oder Maps zu stecken, da der Caller meist ebenfalls nicht davon ausgeht, dass beim Iterieren solche Werte auftauchen.

Wie immer gibt es natürlich Ausnahmen, wo es durchaus nützlich und gut ist, `null` als möglichen Rückgabewert einer solchen Methode zu verwenden oder `null`-Entries zu haben.

Methoden mit bool-Rückgabewert nicht unnötig verkomplizieren

Logische Vergleiche sollten nicht unnötig verbos über mehrere Zeilen gezogen werden und dadurch den Code schwerer und langwieriger zu lesen machen.

Beispiel für Code, der vermieden werden sollte:

```
if not erledigt then
   return 0
else
   return 1
```

Einfacher:

```
return erledigt
```

Das hat natürlich auch seine Grenzen. Das folgende Beispiel sollte man nicht unbedingt komplett zusammen fassen, weil es dann ggfs. schwerer zu lesen wird:

```
if not erledigt then
   return 0
if ignorieren then
   return 1
return default
```

Dennoch kann man es etwas einfacher machen:

```
if not erledigt then
   return 0
return ignorieren or default
```

Für „Spezialisten“ aber auch so:

```
return erledigt and (ignorieren or default)
```

Eine größere Vereinfachung auf eine Zeile kann auch bei größeren Ausdrücken je nachdem trotzdem einfacher und schneller zu erfassen sein. Das kommt auch auf die Terme im Ausdruck und die Variablennamen an. Es gibt hier keine definitive Regel, bis auf die ganz offensichtlichen Fälle im allerersten Beispiel.

Wenn der Ausdruck oder Teile davon zu schwer zu lesen sind, könnte das auch darauf hinweisen, dass man Teile davon in eine Tool-Methode in das entsprechende Objekt auslagern sollte (z.B. die Existenz einer Rechnung

> abprüfen oder sowas), dann fällt es oftmals zusammen auf sowas wie „not isDeleted() and existsInvoice()“, was dann wiederum super einfach zu verstehen ist.

Noch ein real-life Beispiel dazu:

alt:

```
   if not super.isStammdatenuebernahmePossible() then
      return 0
   executeUebernahme = not isEmpfZaehldatenMandatory()
   if not executeUebernahme then
      executeUebernahme = alleZaehlerstaendeEmpfangen()
   if not executeUebernahme then
      return 0
   return 1
```

einfacher:

```
   return super.isStammdatenuebernahmePossible() and (not
isEmpfZaehldatenMandatory() or alleZaehlerstaendeEmpfangen())
```

Der umgebaute Code liest sich relativ eindeutig als „die Superklasse muss bereits sagen, dass die Stammdatenübernahme möglich ist, und außerdem sollen die Zähldaten entweder keine Pflicht sein oder alle Zählerstände sind bereits empfangen“. Das ist viel einfacher zu verstehen, als der 8-zeilige Block darüber, der zudem indirekt doppelte Verneinung verwendet.

Keine Änderungen an Parametern im Verlauf einer Methode

Es ist generell keine empfehlenswerte Praxis, einen übergebenen Parameter im Verlauf einer Methode zu ändern, da dies mehrere Probleme mit sich bringen kann:

- Es kann den ursprünglichen Wert der Variablen verändern, insbesondere wenn es sich um call-by-reference handelt. Dadurch können unerwartete Seiteneffekte auftreten, die schwer nachvollziehbar sind.
- Die Lesbarkeit und Wartbarkeit des Codes kann beeinträchtigt werden und unnötige Verwirrung verursachen.
- Es kann Refactoring erschweren, da Änderungen innerhalb der Methode unerwartete Auswirkungen auf andere Teile des Codes haben können.

- Durch Mehrfachzuweisungen kann es schwierig und zeitaufwändig sein, den genauen Wert des Parameters beim Lesen des Codes zu verfolgen.

Es ist ratsam, in solchen Fällen eine lokale Variable zu verwenden und den bereits vorhandenen Code entsprechend zu refaktorieren.

Es gibt jedoch einige Ausnahmen, die geduldet werden:

- Die Anpassung des Werts ganz am Anfang der Methode, z. B. durch einen Null-Check mit anschließender Initialisierung auf einen Standardwert, um sicherzustellen, dass der Parameter einen gültigen Wert hat.
- Für Code in Hotspots, d.h. in performanzkritischen Teilen, kann die direkte Änderung von Parametern toleriert werden, da moderne JVMs Parameter in Registern speichern und der Zugriff darauf schneller ist als auf lokale Variablen.

Wenn eine solche Ausnahme zur Anwendung kommt, muss dies im entsprechenden Code ausreichend dokumentiert werden, um ungewolltes Refactoring durch andere Teammitglieder zu verhindern.

Überlange Zeilen durch Variablen verkürzen

Für lange if-Bedingungen ist es eine gute Praxis zu überprüfen, ob Teilausdrücke vorher getrennt evaluiert und in Variablen gespeichert werden können, um überlange Zeilen zu vermeiden. Dies kann die Lesbarkeit des Codes erheblich verbessern.

Es ist jedoch wichtig, dabei mögliche Auswirkungen auf die Performance zu berücksichtigen. In einigen Fällen könnten komplizierte Ausdrücke sonst immer evaluiert werden, selbst wenn sie aufgrund von Shortcut-Evaluierung nicht immer berechnet werden müssten. Daher sollte die Optimierung der Lesbarkeit nicht auf Kosten der Leistung gehen.

Die Entscheidung, Teilausdrücke in Variablen zu speichern, sollte daher abhängig von den spezifischen Anforderungen des Codes und der Performance getroffen werden. In Fällen, in denen die Verbesserung der Lesbarkeit klar im Vordergrund steht und die Leistungsauswirkungen minimal sind, ist das Vorwegnehmen von Teilausdrücken in Variablen eine empfohlene Vorgehensweise.

Das folgende Beispiel ist schwer verständlich:

Beispiel

```
   if gas = null -
         or (gas.booleanValue() and offerGroup <=
GasAngebotsgruppe) -
         or (not gas.booleanValue() and offerGroup <=
```

```
StromAngebotsgruppe) then
      return 1
   ...
```

Es könnte wie folgt wesentlich besser lesbar geschrieben werden:

verbessertes Beispiel

```
   isGas = offerGroup <= GasAngebotsgruppe and gas <> null and
gas.booleanValue()
   isEle = offerGroup <= StromAngebotsgruppe and gas <> null and
not gas.booleanValue()
   if gas = null or isGas or isEle then
      return 1
    ...
```

Kein Objekt-Identitätsvergleich mit Boolean-Konstanten

Ein Identitätsvergleich mit `Boolean.TRUE` oder `Boolean.FALSE` ist zu vermeiden, da es gefährlich sein könnte, `==` zu verwenden, da das zu testende Boolean-Objekt möglicherweise nicht von einer der Konstanten stammt, sondern als `new Boolean(boolValue)` konstruiert wurde, auch wenn dies nicht empfohlen wird.

Alternativ kann man für Boolean-Attribute die NN-Methode verwenden: für einen Vergleich mit `true` direkt, für einen Vergleich mit `false` unter Ausschluss des null-values via `not getAttributnameNN(1)`, da ein null-value so auf true vorgegeben wird und wegen des `not` dann im Ergebnis `false` resultiert (nur ein explizites `false` des zu testenden Boolean-Objekts führt so zu einem insgesamten `true` des Ausdrucks).

Ähnlich kann man für beliebige andere Boolean-Objekte vorgehen, indem man die `de.ipcon.tools.NullCheckTools#NN(Boolean)`-Methode verwendet, ebenfalls mit dem Standard-Default-Wert `1` (true) für Vergleiche gegen true oder `not` und Default-Wert `1` (true) für Vergleiche gegen explizites false unter Ausschluss von null.

Für Groovy empfiehlt sich für Vergleiche gegen true die Groovy truth zu verwenden und für Vergleiche gegen explizit false ein Vergleich gegen null mit anschl. && auf `!value`.

Keine Threads in Konstruktoren starten

Das Anti-Pattern, bereits im Konstruktor einen Thread zu starten, der den state des Objekts benutzt, sollte vermieden werden. Gründe (u.a.):

- Auftretende Fehler werden von der JVM in eine `InvocationTargetException` gewrappt, was die Fehlerbehandlung unnötig kompliziert gestaltet.
- Es begünstigt schwer zu lokalisierende Timing-Bugs, wenn der Thread auf Variablen zugreift, die ggfs. noch gar nicht (vom Konstruktor einer Subklasse) oder nicht vollständig initialisiert wurden.
- Es ist intransparent (ggfs. ist eine Kapselung aber sogar erwünscht).

Eine mögliche Lösung dafür ist es, den Thread explizit über einen Methodencall von außen auf dem neu konstruierten Objekt zu starten. Falls man diese Internals verstecken möchte, so bietet sich die Verwendung einer Factory an, welche die Konstruktion und das Starten des Threads zusammen und korrekt für jede Subklasse erledigt und somit nach außen hin kapselt.

Arbeiten mit BOs

Frapping und Graphenstabilität

Frapping sorgt dafür, dass der Objektgraph intern innerhalb einer Methode eindeutig ist. Dies ist zwingend erforderlich, da der Programmcode auf Java-Objekten agiert, bei denen mehrere Instanzen dasselbe Business Object (BO) repräsentieren können.

Ziel des Frappings ist es, die sogenannte „Graphenstabilität“ zu garantieren. Für alle BOs `a` und `b` innerhalb einer Transaktion muss gelten: Haben zwei BOs dieselbe Identifikationsnummer (ID), dann muss es sich um exakt dieselbe Java-Instanz im Speicher handeln.

```
(a.Id = b.Id) => (a = b)
```

Das Frapping muss stets so lokal wie möglich stattfinden. Das bedeutet: Eine Instanzmethode sorgt selbst dafür, dass die ihr übergebenen Parameter-BOs aus demselben BO-Graphen stammen wie ihr eigenes BO. Es darf nicht ungeprüft vorausgesetzt werden, dass übergebene Parameter bereits im korrekten BO-Graphen vorliegen. Eine Ausnahme von dieser Regel bilden Setter-Methoden, die den übergebenen Wert direkt setzen und darauf vertrauen müssen, dass der Wert zuvor auf Aufruferseite korrekt gefrappt wurde.

> Fehler, die durch die Missachtung der Graphenstabilität entstehen, sind im produktiven Betrieb extrem schwer zu isolieren und zu beheben. Sie führen oft zu unvorhersehbarem Verhalten bei nachgelagerten Datenbank- oder Cache-Operationen.

Beim Frappen eines BOs werden sowohl das BO selbst als auch alle damit verbundenen

Relationen bei Bedarf neu geladen. Alle in nicht-persistenten Attributen (`npattr`) gesetzten Werte bleiben beim Frapping erhalten. Auch transiente Werte, die via `BO#setTransientProperty(String, Object)` am BO hinterlegt wurden, bleiben erhalten, sofern sie serialisierbar sind. Befinden sich unter diesen transienten Werten wiederum BOs, werden diese ebenfalls gefrappt.

Temporäre Objekte werden niemals im Server-Cache gespeichert. Daher ist strikt darauf zu achten, dass temporäre BOs, die per `BO#setTransientProperty` oder als `npattr`-Wert an einem zu frappenden BO abgelegt wurden, bereits über einen passenden Loader verfügen. Andernfalls schlägt das Frapping unweigerlich fehl.

Technische Notwendigkeit

Wird ein BO per Query geladen, werden alle Skalare und n:1-Relationen dieses BOs in den Speicher geladen.

Angenommen, ein BO `a` vom Typ `Patient` ist über eine n:1-Relation via `Patient#getGesetzlicheKasse()` mit einem BO `b` vom Typ `Krankenkasse` verknüpft. Der Typ `Patient` besitzt eine weitere n:1-Relation via `Patient#getAbrechnendeKasse()`, die ebenfalls auf dieselbe logische Objektinstanz `b` verweist. Es gilt somit auf Datenbankebene: `a.GesetzlicheKasse.Id == a.AbrechnendeKasse.Id`.

Würde das Objekt `b` nicht automatisch vom Backend gefrappt, hätten Änderungen an `b` über den `GesetzlicheKasse`-Getter keinerlei Auswirkungen auf die Instanz, die über den `AbrechnendeKasse`-Getter erreicht wird, obwohl es sich fachlich um dieselbe Krankenkasse handelt.

Ein weiteres Problem betrifft Transaktionsgrenzen. Wird ein BO `a` über die Transaktion 1 geladen und soll modifiziert werden, muss es in diese Transaktion mittels `include()` eingebunden werden, damit Änderungen aufgezeichnet werden. Liegt nun eine zweite Java-Instanz `kk` vom Typ `Krankenkasse` vor, die jedoch über eine andere Transaktion geladen wurde, darf dieses Objekt nicht direkt über `a.setGesetzlicheKasse(kk)` zugewiesen werden. Vielleicht wurde es vor Stunden im Cache gespeichert oder berechnet. Stattdessen muss die Instanz über den Cache der Zieltransaktion gefrappt werden: `a.setGesetzlicheKasse(tx.frapFromCache(kk))`.

Die Methode `frapFromCache()` löst hierbei zwei Kernprobleme der Systemarchitektur.

1. Sie stellt die Identitätssicherung bereit: Ist das BO der Transaktion bereits bekannt, erkennt `frapFromCache()` dies und gibt die bereits existierende Java-Instanz zurück.
2. Sie regelt die Isolierung und Nebenläufigkeit: Ein BO könnte in mehreren Transaktionen und Threads gleichzeitig verwendet werden. Würden alle Threads auf derselben Java-Instanz operieren und eine Transaktion eine temporäre Änderung

vornehmen, hätte das direkten Einfluss auf alle anderen Transaktionen im System. Das Frappen erhöht somit die Nebenläufigkeit.

Praxis-Szenario I: Transaktionsgrenzen und include()

Das folgende Groovy-Beispiel demonstriert das Verhalten von Instanzen beim Einbinden in eine Transaktion innerhalb der Verwaltung von Fachabteilungen. Die Fachabteilungen sind als Quertabelle hinterlegt und können mittels OQL angefragt werden.

```
def tx1 = newTx()
def tx2 = newTx()
```

Nach diesen Zeilen stehen zwei Transaktionen zur Verfügung. Möchte man nun beispielsweise eine Fachabteilung aus der Datenbank laden, kann man dies über eine der beiden Transaktionen tun. In dem nachfolgenden Code wird zweimal das gleiche BO vom Typ `Fachabteilung` geladen. Dabei wird zunächst nur `f1` der Transaktion `tx1` hinzugefügt. Die letzten Zeilen dienen zur Ausgabe der verfügbaren BOs `f1` und `f2`.

```
def f1 = tx1.queryBO("Fachabteilung f where not Ldel ORDER BY Id
LIMIT 1").get(0)
def f2 = tx2.queryBO("Fachabteilung f where not Ldel ORDER BY Id
LIMIT 1").get(0)
f1 = tx1.include(f1)

println "BO f1: $f1 \n\t BOL: ${f1.getBOLoader()} \n\t Tx:
${f1.getTransaction()}"
println "BO f2: $f2 \n\t BOL: ${f2.getBOLoader()} \n\t Tx:
${f2.getTransaction()}"
```

Die Konsole liefert folgende Ausgabe:

```
BO f1: Fachabteilung[10046689]@4e46a0a1
    BOL: Transaction@4686004d#p0#n0#b1#r1
    Tx: Transaction@4686004d#p0#n0#b1#r1
BO f2: Fachabteilung[10046689]@6702b1c7
    BOL: Transaction@13d2285e#p0#n0#b0#r0
    Tx: null
```

Wie in der Ausgabe zu sehen ist, sind `f1` und `f2` zwei Instanzen des BOs mit der Id `10046689`. Ebenfalls ist zu sehen, dass beide Instanzen mit unterschiedlichen `BOLoader` -Instanzen geladen wurden. Da zuvor die Instanz `f1` in die Transaktion `tx1` eingebunden wurde, wird auch die zugehörige Transaktion angezeigt. Bei `f2` ist dieser Wert `null`, da `f2` noch keiner Transaktion hinzugefügt wurde.

Nachfolgend wird auch `f2` der Transaktion `tx1` hinzugefügt.

```
f2 = tx1.include(f2)

println "BO f1: $f1 \n\t BOL: ${f1.getBOLoader()} \n\t Tx:
${f1.getTransaction()}"
println "BO f2: $f2 \n\t BOL: ${f2.getBOLoader()} \n\t Tx:
${f2.getTransaction()}"
```

Die Ausgabe dieses Codes ist wie folgt:

```
BO f1: Fachabteilung[10046689]@4e46a0a1
    BOL: Transaction@4686004d#p0#n0#b1#r1
    Tx: Transaction@4686004d#p0#n0#b1#r1
BO f2: Fachabteilung[10046689]@4e46a0a1
    BOL: Transaction@4686004d#p0#n0#b1#r1
    Tx: Transaction@4686004d#p0#n0#b1#r1
```

Aufgrund der Tatsache, dass `f1` sowie `f2` das gleiche BO mit der Id `10046689` beinhalten, prüft die Methode `include()`, ob die Transaktion das BO bereits in ihrem Graphen hat. In diesem Fall gibt sie dieses existierende BO zurück. In der Log-Ausgabe sieht man deutlich, dass es sich nun um dieselbe Java-Instanz handelt.

Wenn man nun aber `f1` oder `f2` der Transaktion `tx2` direkt hinzufügen möchte, kommt das Frappen ins Spiel. Da die Objekte bereits im BO-Graphen von `tx1` enthalten sind, können diese nicht ohne Weiteres einem fremden BO-Graphen hinzugefügt werden. In einem solchen Fall würde eine `TransactionIncludeException` geworfen werden. Um das Objekt nun trotzdem der Transaktion `tx2` hinzuzufügen, muss das Objekt neu geladen werden. Durch den Aufruf der Methode `tx2.frapBOFromCache()` wird intern überprüft, ob das gewünschte BO bereits über die Transaktion `tx2` geladen wurde.

```
def f1 = tx1.queryBO("Fachabteilung f where not Ldel ORDER BY Id
LIMIT 1").get(0)
```

```
def f2 = tx1.queryBO("Fachabteilung f where not Ldel ORDER BY Id
LIMIT 1").get(0)

f1 = tx1.include(f1)
f2 = tx1.include(f2)

println "Before Frapping..."
println "BO: $f1 \n\t BOL: ${f1.getBOLoader()} \n\t Tx:
${f1.getTransaction()}"
println "BO: $f2 \n\t BOL: ${f2.getBOLoader()} \n\t Tx:
${f2.getTransaction()}"

f1 = tx2.include(tx2.frapBOFromCache(f1))

println "\nAfter Frapping..."
println "BO f1: $f1 \n\t BOL: ${f1.getBOLoader()} \n\t Tx:
${f1.getTransaction()}"
println "BO f2: $f2 \n\t BOL: ${f2.getBOLoader()} \n\t Tx:
${f2.getTransaction()}"
```

Die Ausgabe zeigt den Zustand vor und nach dem Frappen:

```
Before Frapping...
BO f1: Fachabteilung[10046689]@4e46a0a1
    BOL: Transaction@4686004d#p0#n0#b1#r1
    Tx: Transaction@4686004d#p0#n0#b1#r1
BO f2: Fachabteilung[10046689]@4e46a0a1
    BOL: Transaction@4686004d#p0#n0#b1#r1
    Tx: Transaction@4686004d#p0#n0#b1#r1

After Frapping...
BO f1: Fachabteilung[10046689]@6702b1c7
    BOL: Transaction@13d2285e#p0#n0#b1#r1
    Tx: Transaction@13d2285e#p0#n0#b1#r1
BO f2: Fachabteilung[10046689]@4e46a0a1
    BOL: Transaction@4686004d#p0#n0#b1#r1
    Tx: Transaction@4686004d#p0#n0#b1#r1
```

Ein weiteres wichtiges Verhalten zeigt sich, wenn ein BO angefragt wird, das der Transaktion noch komplett unbekannt ist, d.h. noch nicht über diese Transaktion geladen wurde. Hierbei fordert die Methode eine neue Instanz an.

```
def f1 = tx1.queryBO("Fachabteilung f where not Ldel ORDER BY Id
LIMIT 1").get(0)
f1 = tx1.include(f1)
println "Before Frapping: ${f1}"
f1 = tx2.include(tx2.frapBOFromCache(f1))
println "After Frapping: ${f1}"
```

```
Before Frapping: Fachabteilung[10046689]@7da8c505
After Frapping: Fachabteilung[10046689]@6102ac74
```

In der Ausgabe ist zu sehen, dass `f1` vor dem Frappen eine andere Instanzadresse hat als nach dem Frappen. Die zurückgegebene Instanz wurde vorher nirgends über `tx2` angefragt, weshalb eine neue Instanz intern geladen und übergeben wird.

Praxis-Szenario II: Graphenstabilität im Datenmodell

Die folgenden Beispiele gehen von diesem einfachen Schema aus:

```
<Entity name="Patient" extends="CBO" plural="Patienten">
  <attr name="Pflegestufe" type="Pflegestufe" relation="n-1"/>
</Entity>

<Entity name="Pflegestufe" extends="Quertabelle"
plural="Pflegestufen"/>
```

Jedem Patienten kann genau null bis eine Pflegestufe zugeordnet werden. Es wird davon ausgegangen, dass bereits zwei Pflegestufen-Objekte im System existieren: „Pflegestufe 1“ (Name = ’Stufe 1’, Id = 1) und „Pflegestufe 5“ (Name = ’Stufe 5’, Id = 5).

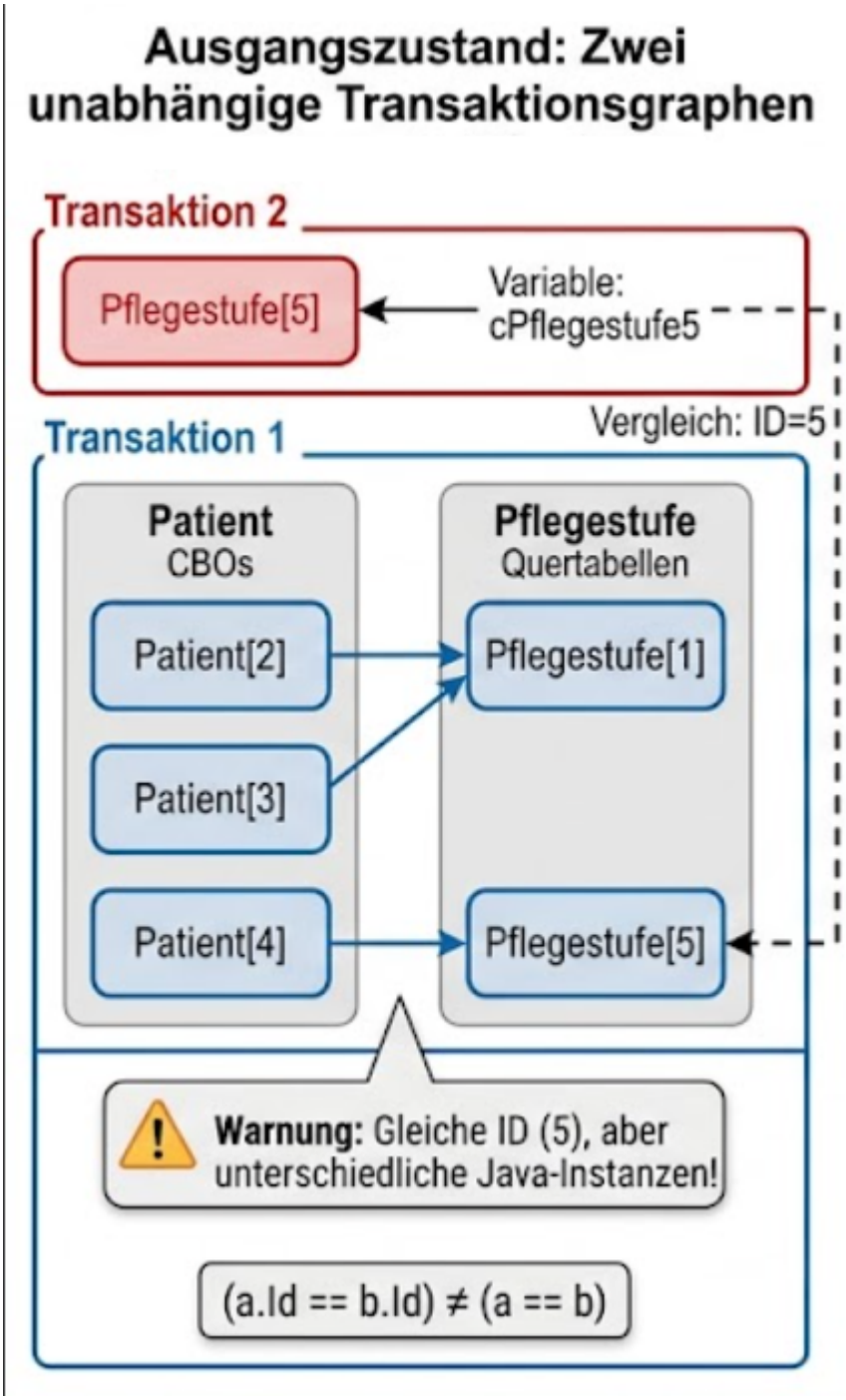
Es liegt folgende Ausgangssituation vor:

Eine Transaktion `tx2` enthält eine eigene Version von `Pflegestufe[5]`, nachfolgend über die Variable `cPflegestufe5` referenziert.

Gleichzeitig existiert eine Transaktion `tx`, in der die Patienten mit den IDs 2, 3 und 4

geladen wurden. Patient 2 und Patient 3 haben `Pflegestufe[1]` gesetzt. Patient 4 hat als Pflegestufe das Objekt mit der ID 5 gesetzt. Das Objekt `Pflegestufe[5]` in `tx` ist ein anderes Java-Objekt als `cPflegestufe5`, enthält zurzeit jedoch die gleichen Daten.

Solange nur Getter verwendet werden, können neue Objekte in Relationen dynamisch nachgeladen werden. Diese sind dann jedoch immer demselben Objektgraphen zugeordnet wie das Ausgangsobjekt. Man sagt daher auch, eine Transaktion ist graphenstabil.



Patient 2 und 3 verweisen nicht nur beide auf eine Pflegestufe mit Id 1, sondern sogar auf die gleiche Java-Objekt-Instanz. Unter allen geladenen Objekten in einer Transaktion ist eine Id höchstens einmal vertreten.

```
assert tx.getBO(1).name == 'Stufe 1'

def pflegestufe = tx.includeBO(tx.getBO(1))
pflegestufe.name = 'Akutpflege'
assert tx.getBO(2).pflegestufe.name == 'Akutpflege'
assert tx.getBO(3).pflegestufe.name == 'Akutpflege'
assert tx.getBO(1).name == 'Akutpflege'
```

Ändert man den Inhalt dieses Objektes, dann gilt diese Änderung sofort für alle Zugriffspfade. Angenommen, in der Datenbank gibt es einen weiteren Patienten mit Id 6, der ebenfalls die Pflegestufe mit Id 1 hat. Wird dieses Objekt nun in die Transaktion geladen, verlinkt das System es direkt mit der aktuellen, modifizierten Version des Objektes im Speicher.

```
assert tx.getBO(6).pflegestufe.name == 'Akutpflege'
```

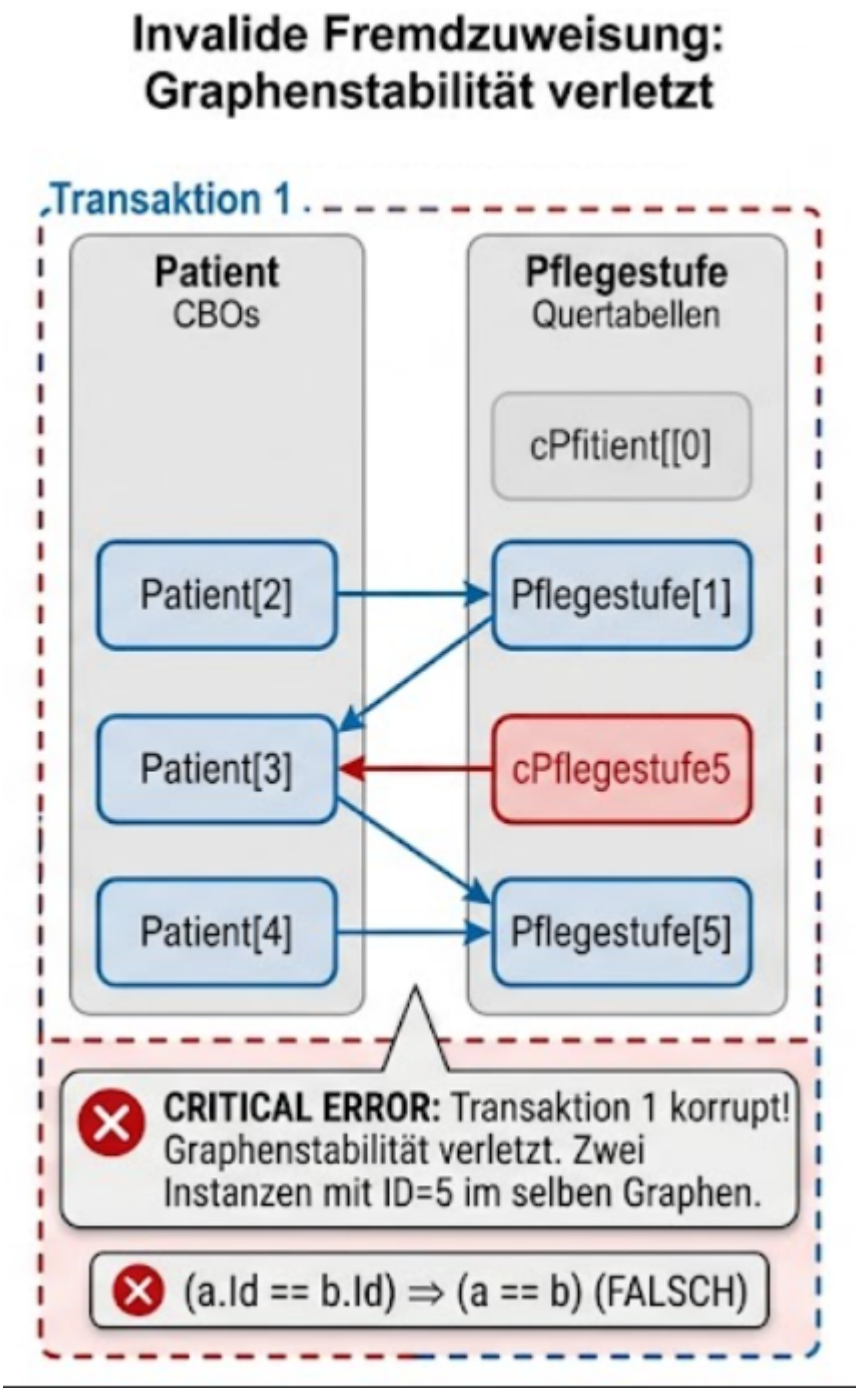
Anti-Patterns und Fallstricke

Kommen wir auf das obige Szenario zurück, bei dem die Variable `cPflegestufe5` die `Pflegestufe[5]` repräsentiert, welche mit `tx2` geladen wurde. Ein BO mit dieser Id ist in `tx` bereits bekannt, da es `Patient[4]` zugeordnet ist.

Nachfolgend werden die drei häufigsten Fehlermuster beschrieben, die im Rahmen von Code-Reviews zwingend abzuweisen sind.

1. Verletzung der Graphenstabilität durch direkte Fremdzuweisung

```
def patientDrei = tx.includeBO(tx.getBO(3))
patientDrei.pflegestufe = cPflegestufe5
```



Dieses Vorgehen führt zu schwerwiegenden Fehlern im System. Die Pflegestufe von `patientDrei` wurde auf eine Objektinstanz gesetzt, die nicht zu dieser Transaktion gehört. Der Setter von Pflegestufe ruft implizit auch die Methode `addPflegestufe(patientDrei)` auf `cPflegestufe5` auf, um die Rückrelation zu pflegen. Die transaktionslokale Instanz von `Pflegestufe[5]` innerhalb von `tx` erfährt jedoch nichts von dieser Änderung. Der Graph aller in der Transaktion geladenen Objekte ist nicht mehr stabil, da die IDs der erreichbaren Objektinstanzen im Speicher nicht mehr eindeutig sind.

2. Ignorieren des Rückgabewerts (Der „No-Op“-Fehler)

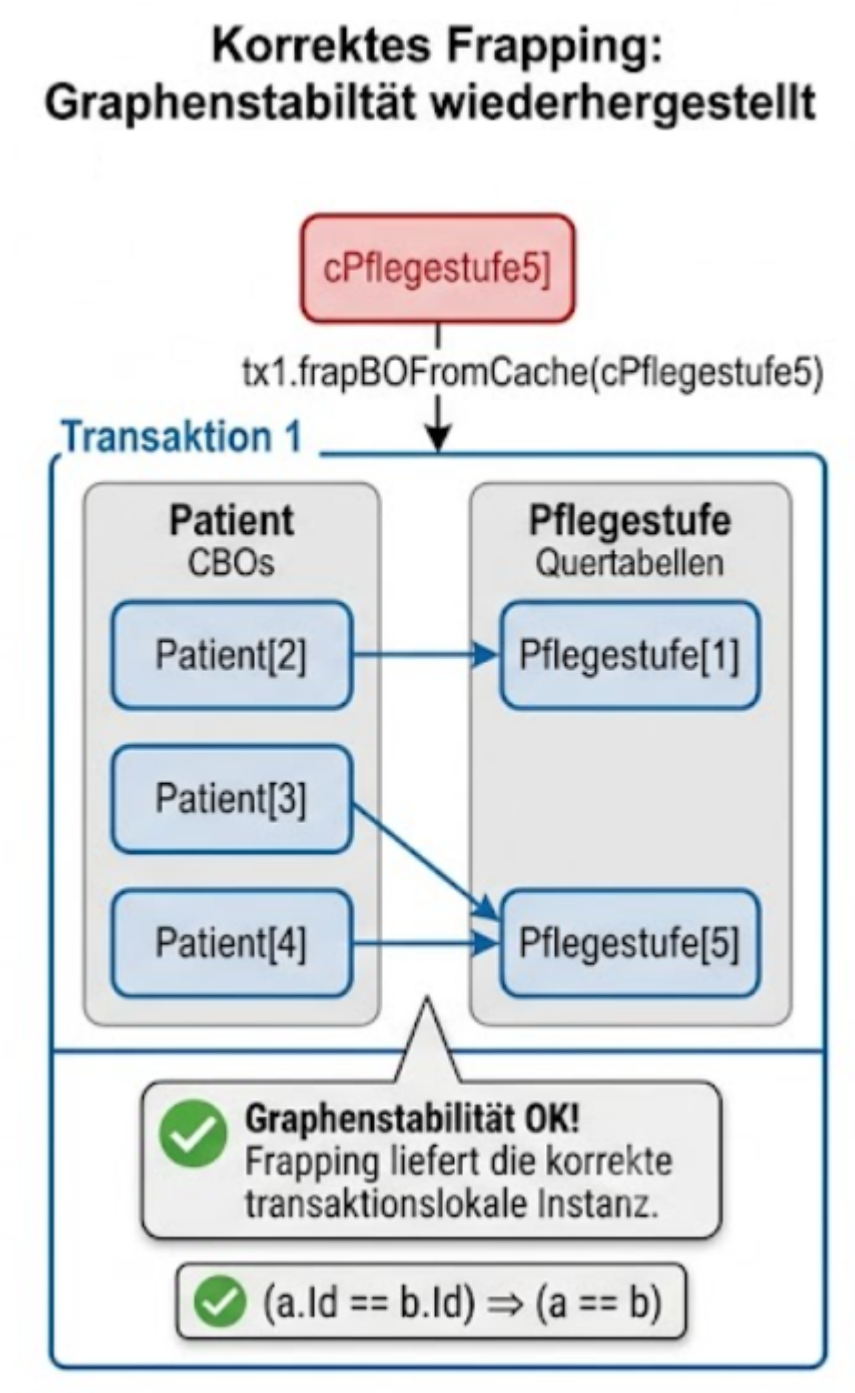
Ein häufiger Trugschluss ist die Annahme, dass `frapBOFromCache()` das übergebene Objekt direkt modifiziert.

```
def patientDrei = tx.includeBO(tx.getBO(3))
tx.frapBOFromCache(cPflegestufe5)
patientDrei.pflegestufe = cPflegestufe5
```

Hier wird das Objekt zwar gefrappt, jedoch wird der Rückgabewert fälschlicherweise ignoriert. Das zurückgelieferte Objekt ist nicht das gleiche, das übergeben wurde. Die Instanzvariable `cPflegestufe5` wurde nicht verändert und zeigt weiterhin auf das transaktionsfremde Objekt. Dieser Code ist vom Verhalten identisch zum vorherigen fehlerhaften Beispiel.

Korrekte Umsetzung: Der Rückgabewert muss zwingend aufgefangen und zugewiesen werden.

```
def patientDrei = tx.includeBO(tx.getBO(3))
patientDrei.pflegestufe = tx.frapBOFromCache(cPflegestufe5)
```



Frappen ist also nicht nur notwendig, weil bereits ein Objekt mit dieser Id im Graphen bekannt sein könnte, sondern um jedem Graphen sein eigenes Objekt zuzuweisen. Ist `Pflegestufe[5]` der Transaktion noch komplett unbekannt, erstellt sie sich beim Frappen automatisch eine transaktionslokale Kopie des Objektes.

3. Inflationäre Nutzung („Over-Frapping“)

Das folgende Beispiel stellt zwar sicher, dass auf jeden Fall gefrappt wurde, jedoch auch in Fällen, in denen es absolut nicht nötig gewesen wäre.

```
def patientDrei = tx.includeBO(tx.getBO(3))
patientDrei.pflegestufe = tx.frapBOFromCache(tx.getBO(5))
patientDrei.pflegestufe =
tx.frapBOFromCache(tx.getBO(4).pflegestufe)
def patientVier = tx.getBO(4)
patientDrei.pflegestufe =
tx.frapBOFromCache(patientVier.pflegestufe)

assert tx.frapBOFromCache(tx.getBO(5)).is(tx.getBO(5))
```

Als Folge dieses Vorgehens wird der Quellcode schlechter lesbar und unnötig unübersichtlich. Da `Pflegestufe[5]` und `Patient[4]` bereits mit dieser Transaktion geladen wurden, findet `frapBOFromCache()` ohnehin genau dasselbe Objekt in der Transaktion wieder und gibt es unverändert zurück. Hier gilt immer die strikte Objektidentität im Speicher, weshalb das Frappen redundant ist.

Temporäre BOs

Anstatt für temporäre BOs manuell `setBOLoader` und `setTempId` zu rufen muss `BO.newInstance` benutzt werden.
Dies stellt sicher das die Setup-Reihenfolge korrekt ist, da der BOLoader Einfluss auf die temporäre Id haben kann `BackendQueryAnsweringBOLoaderI`).

Der Parameter `includeIfTx` sollte für temporäre BOs false sein, damit diese nicht in der (möglichen) Transaction eingebunden werden.

Aspekt-Unterklassen für BOs

Falls eine Entity-Implementierung zu groß wird, können Teile des Codes in sogenannte „Aspektklassen“ ausgelagert werden. Es sollte immer nur thematisch zusammengehöriger Code in eine Aspektklasse ausgelagert werden.

Das Konzept der Aspekt-Klassen ist eine Eigenheit von MyTISM und sollte nicht mit dem Konzept der aspektorientierten Programmierung verwechselt werden. Diese dient zur Verwendung generischer Funktionalitäten über mehrere Klassen hinweg (Cross-Cutting Concern), bzw. der Trennung der Geschäftslogik von der Programmlogik.

Werden mehrere Aspektklassen für eine Entität benötigt, ist darauf zu achten, dass die Vererbungshierarchie sinnvoll ist (s. auch Kapitel „Projekt Build“ in der Einarbeitung). Von mehreren (Aspekt-)Klassen benötigter Code muss also weiter oben in der Hierarchie

angesiedelt sein, um für all diese Klassen verfügbar zu sein. Dies ist gleichzeitig auch ein Nachteil des Konzepts der Aspektklassen - eine solch strukturell bedingte Aufteilung kann „künstlich“ wirken und unhandlich sein.

Beispiel - Aufteilung der Entität `LieferSimulation`:

```
   /*
    * This class was split up into several classes reflecting
certain aspects
    * due to its huge size ("->" means "extends"):
    *
    * LieferSimulation.nrx
    *    -> LieferSimulationConfigurationAspects.nrx
    *    -> LieferSimulationSchemaAspects.nrx
    *       [...]
    */
   @ENTITY LieferSimulation EXTEND ConfigurationAspects@

   /*
    * LieferSimulationConfigurationAspects.nrx
    */
   @ENTITY LieferSimulation EXTEND SchemaAspects@

   /*
    * LieferSimulationSchemaAspects.nrx
    */
   @ENTITY LieferSimulation@
```

Anwendung: Da die Aspektklassen nicht im Schema hinterlegt sind, muss in der Entity-Implementierung manuell mittels des Keywords `EXTEND` abgeleitet werden. Es ist immer der Entitätsname der im Schema hinterlegten Entität anzugeben. Der echte Name der Aspektklasse spiegelt sich nur im Dateinamen wider. Aus dem obigen Beispiel werden (u.a.) folgende Klassen generiert:

```
   /*
    * LieferSimulation.nrx
    */
   class LieferSimulation extends
         LieferSimulationConfigurationAspects abstract
```

```
   /*
    * LieferSimulationConfigurationAspects.nrx
    */
   class LieferSimulationConfigurationAspects extends
         LieferSimulationBase abstract
```

(Statische) Hilfsmethoden a la „byTid(...)“

(Statische) Hilfsmethoden a la „getInstanceByTid(...)“ / „getByTid(...)“ sind als deprecated zu markieren und auf „byTid(...)“ umzustellen.

Wenn zeitlich möglich soll der Code komplett auf die neuen „byTid(...)“-Methoden umgestellt und die alten Methoden entfernt werden, sofern man sich sicher ist, alle Stellen (SEs, Dienste, BO-Masken, ...) korrekt umgestellt zu haben.

Noch besser ist es, für häufig benutzte Instanzen von Objekten statische „for...“-Methoden zu bauen, anstatt die Tid-Konstanten zu verwenden; diese rufen in der Regel die „byTid“-Methode und ein Exponieren der Tid-Konstanten der Klasse nach außen ist dann nicht mehr nötig.

Wenn zeitlich möglich sollte direkt auf die „for...“-Methoden umgestellt werden; die Anmerkungen oben bzgl. Vollständigkeit gelten natürlich ebenso.

Beispiel:

```
   properties inheritable
      TID_GERMANY = 'DE'

   method byTid(bol = BOLoaderI, tid = String) returns Land
static
      return Quertabelle.getValue(bol, 'Land', tid, 'Tid')

   method forGermany(bol = BOLoaderI) returns Land static
      return byTid(bol, TID_GERMANY)
```

Der Aufruf von anderen Klassen aus verkürzt sich damit folgendermaßen:

```
    method doSomething()
       bol = getBOLoader()
       de1 = Land.byTid(bol, Land.TID_GERMANY)
```

```
        de2 = Land.forGermany(bol)
```

tl;dr: In besagtem Anwendungsfall bitte nur noch „byTid(…)“- und „for…“-Methoden bauen und verwenden. Altlasten ggfs. umbauen oder mindestens als *deprecated* markieren.

Override-Annotation für Methoden für virtuelle Attribute

Alle Methoden, die virtuelle Attribute implementieren, sollten mit einem `@Override` annotiert werden. Für alle Schema-definierten virtuellen Attribute, die keine Aggregate sind, wird im generierten Code der Entity („Base“-Klasse) eine abstrakte Methode erzeugt, die durch den Compiler eine Implementierung erzwingt. Durch die Annotation als `@Override` fällt es dann zusätzlich auch auf, wenn man ein virtuelles Attribut aus dem Schema entfernt, aber die zugehörige implementierende Methode noch in der Klasse verbleibt.

Verwendung von `npattr` und `BO#setTransientProperty(String, Object)`

Eigenschaften und Verwendung nicht-persistenter Attribute (`npattr`)

Nicht-persistente Attribute (`npattr`) sind im Schema definierte Attribute einer Entität, die zum Speichern von temporären Werten am BO benutzt werden können.

Sie verhalten sich auf der Client-Seite im Prinzip genau wie persistente Attribute (getter und setter werden automatisch generiert, Automatik-Strukturelemente beinhalten XML-Code für sie etc.) und sind insofern auch typensicher. Ihr Wert wird beim Frapping erhalten, sofern - im Falle von BOs - der Loader des Werts konsistent mit dem des Objekts ist.

Die gesetzten Werte werden niemals zum Server übertragen und daher auch nicht in der Datenbank gespeichert.

Ihr Einsatzzweck beschränkt sich weitgehend auf Fälle, in denen man Werte nur sehr kurzzeitig vom Benutzer in der UI eingeben oder ihm diese anzeigen lassen will und der Zugriff via Schema oder generierter getter/setter einen geringeren Programmieraufwand bedeutet. Als Werte sind nur Schema-bekannte Typen möglich.

> Im MyTISM-Core werden `npattr` z.B. für die Auswahl von Test-BOs an Alarmen verwendet.
> Ein weiteres Beispiel ist die Auswahl der Locale1 und Locale2 am L10n-Bundle-Objekt.
> Weiterhin werden `npattr` auch zur Konfiguration der Filter im Dateisystem-Sync eingesetzt.
> Als letztes Beispiel sei angeführt, dass auch die Auswahl der Sprache und des Druckziels im Druckdialog via `npattr` am Report zwischengespeichert werden.

Eigenschaften und Verwendung der transient Properties-Map am BO

Die Methode `BO#setTransientProperty(String, Object)` dient ebenfalls zum Speichern von temporären Werten am BO, jedoch eher dynamisch und nicht statisch am Schema definiert. Beim Setzen des Werts kann zusätzlich angegeben werden, dass der gespeicherte Wert bei allen Änderungen an diesem Objekt verworfen werden soll (optionaler Parameter `versioned`). Außerdem kann ein Wert bedingt nur dann gesetzt werden, wenn die aktuelle Versionsnummer des BOs mit der übergebenen Nummer übereinstimmt (optionaler Parameter `explicitVersion`).

Die einfache Version von `BO#setTransientProperty(String, Object)` ohne zusätzliche Parameter sowie `BO#getTransientProperty(String)` können auch sehr reibungslos implizit via Schema-Zugriff aufgerufen werden, indem man dem gewünschten Namen der transient Property einen Unterstrich („_“) voranstellt. Dies ist besonders einfach im Falle von Groovy-Skripten möglich, da im BO ein Mechanismus implementiert ist, der die propertyMissing-Calls abfängt und auf das Schema umleitet. So ist eine Schreibweise in der Form `bo._CachedValue = 5` sowie `if (5 = bo._CachedValue)` möglich, was die Verwendung in Skripten und insbesondere `Scripted Attribute` getter oder setter Implementationen erheblich vereinfacht.

Ihr Wert wird beim Frapping erhalten, sofern der Wert serialisierbar ist und - im Falle von BOs - der Loader des Werts konsistent mit dem des Objekts ist. Wenn sie nicht serialisiert werden können (z.B. awt- oder swing-Objekte), werden sie durch einen Null-Wert ersetzt und eine Warnung über den Vorgang wird protokolliert.

Ihr Einsatzzweck ist recht universell, da man diese Werte auch in der UI verwenden kann, sofern man sie via `Scripted Attribute` exponiert. Eine Verwendung zum Zwischenspeichern von Resultaten, zum vorübergehenden Vorhalten eines Zustands und vor allem zum Caching ist üblich. Als Werte sind prinzipiell alle Java-Objekte möglich, wie an der Signatur der Methode zu erkennen ist - daher ist die Verwendung auch nicht typensicher.

Im MyTISM-Core werden transient Properties z.B. für die „Badges“ am BO verwendet, die zur Markierung von Objekten für Queries Verwendung finden.
Ein weiteres Beispiel ist der Einsatz zur Zwischenspeicherung des ausgewählten Reports oder der zu druckenden Objekte im Druckdialog.
Als letztes Beispiel sei erwähnt, dass auch der Cache der BLOBs an Objekten eine transient Property verwendet.

Sollte ein cached virtual attribute in einem Report nicht den gewünschten Caching-Effekt zeigen, so liegt das meist daran, dass man mit einer BO-Instanz ohne filled cache (also ohne Wert in der transient property) in den Report eingestiegen ist. Der Report baut ggfs. einen neuen

CachingBOLoader mit einem InstrumentedSchema und frappt die zu druckenden BOs mit dem neuen Loader. In der frapped Instanz werden zwar alle transient properties beim deep-clone übernommen, aber wenn der Wert noch nicht existierte, so nützt das nichts. Beim Drucken wird der Cache dann zwar gefüllt, aber der nächste Druckvorgang bekommt wieder die BO Instanz ohne filled cached und berechnet den Wert dann wieder teuer und langwierig neu. Abhilfe schafft hier, dass man den getter des cached virtual attribute einmal ruft bevor man den Druckvorgang auslöst, z.B. im Code der entsprechenden Action.

Es gibt auch noch die Transaction-Properties, die tx-sensitiv (d.h. gegenüber Rollbacks, Savepoints etc.) sind. Somit sind diese sehr gut zum Caching von Zwischenergebnissen geeignet, die „layered“ wieder zum Vorschein kommen sollen, wenn ein Savepoint zurückgerollt wird. Eventuelle Zwischenergebnisse sind so im Gegensatz zu einfacheren, via npattr oder transienten Properties implementierten Caches nicht einfach weg. Dies kann in Bezug auf die Leistung einen großen Unterschied machen.
Dafür funktionieren die beiden letztgenannten Cache-Implementierungsarten jedoch auch ohne eine Transaktionsbindung.

Korrekte Verwendung von Savepoints

Wenn man einen Savepoint nicht mehr braucht, da man mit dem Teil, den man evtl. zurücksetzen möchte, fertig ist, sollte man diesen immer per `commit()` abschließen und somit invalidieren. Im finally schlägt das dann ggfs. auf, wenn eine Exception ausgelöst wurde und das `commit()` daher nicht gerufen worden ist, und dort kann man dann einen `rollback()` veranlassen, sofern der Savepoint für `isValid()` true zurück gibt.

Falsch:

```
def sp = tx.savepoint('bla')
try {
   // ...
} catch (IRuntimeException ire) {
   sp.rollback()
   throw ire
}
```

Richtig:

```
def sp = tx.savepoint('bla')
try {
   // ...
   sp.commit()
} finally {
   if (sp.isValid()) {
      sp.rollback()
   }
}
```

Besonders kritisch ist die falsche Variante, wenn die Transaction lange offen bleibt und z.B. im loop dauernd neue Savepoints erstellt aber nie committet werden und nur im Falle einer Exception rollbacked werden. Es wird unnötig viel Speicher allokiert und erst wieder beim finalen Speichern der Transaction oder ihrem `close()` freigegeben.

Achtung, bei einem rollback werden auch alle include()-s rückgängig gemacht, die man innerhalb des Savepoints gemacht hatte. D.h. wenn man einen Savepoint eröffnet, darin BOs includet und den Savepoint dann rollbacked, werden danach Änderungen an solchen (nach dem rollback nicht mehr includeten) BOs nicht mehr aufgezeichnet, es sei denn, man includet diese erneut. Wenn man das nicht weiß / beachtet, könnte das zu Bugs führen, da die Aufzeichnung nicht aktiviert ist.

verifyOn(Client|Server) sowie die before- und after-Varianten

Allgemeines

Jegliche verifies sowie die before- und after-Varianten laufen sowohl auf dem Solstice-Client als auch auf dem Server immer single-threaded ab.

Die Reihenfolge der Klassen ist nicht festgelegt und von daher beliebig, d.h. man kann sich in einer verify-Methode von Klasse X nicht darauf verlassen, dass die gleiche verify-Methode von Klasse Y bereits vorher gelaufen ist.

Wenn die Id von neuen BOs bereits bekannt sein und benutzt werden soll, um sie z.B. als Teil eines generierten Texts an einem Attribut zu speichern, so muss dies auf der Serverseite geschehen, da nur dort die Id bereits gezogen wurde. Das BO hat in allen 3 verify-Phasen bereits die vom Server zugeteilte Id gesetzt, so dass diese sowohl im before- als auch im afterVerifyOnServer verwendet werden kann.

Die serverseitigen verifyOn-Methoden prüfen keine Benutzerrechte ab,

d.h. hier sind - im Gegensatz zu den clientseitigen Methoden - beliebige Änderungen und recalcs möglich, ohne dem Benutzer spezielle Rechte einräumen zu müssen.

Ein neu erstelltes BO bekommt natürlich nur dann die gezogene Id, wenn der Speicherprozess erfolgreich war und nicht durch ein SaveVeto oder andere Exceptions unterbrochen wurde.

Client bedeutet hier wirklich nur Solstice-Client. Andere „Clients“, z.B. Services via BS oder run-bs, führen diese verifies nicht aus.

Dialoge

Aus den verify-Methoden des Clients heraus sollen nur modale Dialoge erstellt werden. Grund ist, dass beim Speichern viele verify-Methoden gerufen werden und der Benutzer sonst mit einer Unmenge an Dialogfenstern gleichzeitig überschüttet werden könnte.

Einsatzzwecke für die verschiedenen Varianten

Da es immer wieder Unsicherheiten gibt, was in das verifyOn* und was in das beforeVerifyOn* bzw. in das afterVerifyOn* gehört, hier entsprechenden Konventionen dazu:

beforeVerifyOn*

Dinge, die zu Unstimmigkeiten im verifyOn* führen könnten, müssen vor dem verifyOn* abgestellt werden, und somit das Objekt zur Verifikation vorbereitet werden. Nur so kann das verifyOn* dann auf einem korrekt auskalkulierten und (hoffentlich) integeren Objekt seine Arbeit tun.

Dazu gehören beispielsweise recalcs, die Daten aufgrund des neuen Zustands des Objekts aufbereiten oder neuberechnen.

Dies gilt auch für cached properties, die auf der Serverseite basierend auf dem aktuellen Zustand unter Einbezug der Änderungen in der Transaktion rekalkuliert werden. So kann man bei Bedarf auch den neuen Wert einer cached property noch im verify auf Korrektheit oder Konsistenz mit anderen Werten überprüfen.

Ebenfalls in der beforeVerifyOnServer-Phase müssen Nummern aus BUs gezogen und gesetzt werden, z.B. Belegnummern, EANs, Kundennummern, etc., damit sichergestellt ist, dass es keine doppelten Nummern gibt und diese Nummern in der verify-Phase für weitere Prüfungen auch bereits gesetzt sind.

verifyOn*

Im verifyOn* dürfen nur reine Checks und *keine Modifikationen an Daten* vorgenommen werden, weder auf der Client- noch auf der Serverseite - das ergibt sich bereits aus dem Namen der Methode „verify“, Datenänderungen wären schließlich nicht im Methodennamen erwähnte „Seiteneffekte“.

Historisch gewachsen gibt es noch diverse Codestellen, die im verify doch Änderungen vornehmen. Dieser Code sollte nach und nach refaktoriert werden.

Checks, die nur lokal modifizierte Daten betreffen, können im verifyOnClient ausgeführt werden.

Alle Checks, die einen ggfs. parallel durch andere Benutzer geänderten Datenstand mit berücksichtigen sollen, müssen im verifyOnServer durchgeführt werden, da nur dort der aktuelle Datenstand (exkl. evtl. noch nicht gesyncter Daten von anderen Knoten) vorliegt.

afterVerifyOn*

Im afterVerifyOn* sollten nur noch Dinge ausgeführt werden, die - nachdem das verifyOn* fehlerfrei lief - den verifizierten Zustand nicht wieder kaputt machen, sondern nur noch unkritische Dinge am Objekt ergänzen.

Dazu gehört beispielsweise das Mitführen einer Historie.

Umgehung der Datenvalidierung in Transaktionen

Die Klasse `Transaction` bietet Administratoren die Möglichkeit, die client- und serverseitigen Verifizierungsprozesse (verify hooks) gezielt zu deaktivieren. Dies geschieht über die Hilfsmethoden `disableClientVerification` und `disableServerVerification`.

Die Verwendung dieser Funktionen wird als Bemerkung in der BT dokumentiert, um den Grund für eventuelle spätere Probleme durch Dateninkonsistenzen nachvollziehen zu können.

Beispiele:

- Import von Daten, die nicht den vorgegebenen Formaten entsprechen.
- Vorübergehende Änderung von Datensätzen, die im regulären Betrieb nicht zulässig wären.
- Vermeidung von Aktionen, die vor oder nach der Validierung in den entsprechenden before- oder after-hooks ausgelöst werden (z.B. Exporte, Benachrichtigungen).

Die Umgehung der Validierung sollte nur in **Ausnahmefällen** und mit **äußerster Vorsicht** erfolgen. Die Daten, die ohne Validierung gespeichert werden, sollten manuell auf **Korrektheit und Konsistenz** geprüft werden, um spätere Probleme zu vermeiden.

L10n

Müssen im Zuge von manuell angelegten Strukturelementen, o.ä. im L10n-System entsprechende Ressourcen angelegt werden, so sind diese in einem sogenannten Custom-Bundle einzutragen.
Für das projektspezizische BO-Bundle gibt es eine Art „Selbstheilungsmechanismus“, der beim Serverstart sämtliche in diesem Bundle befindlichen Ressourcen gegen das zugrundeliegende Schema abgleicht und diejenigen Ressourcen raus löscht, für die es keine Entsprechnung (Ordner, Entität, Attribut-Singular bzw. -Plural) im Schema gibt.

Common Pitfalls

- Keine Unterstriche in Bundle-Namen verwenden
  (Nachstehende Erklärung ist vielleicht noch etwas holperig und darf gerne präzisiert werden.)
  Anhand des Unterstrichs wird die jeweilige Sprache „ermittelt“. Beim Server-Start landen die jeweiligen Einträge in sprachen-eigenen Caches. Die Einträge eines Bundles namens „foo_bar“ landen also im Sprachen-Cache „bar“. Der Client fragt für seine aktuelle Locale (oder die gewählte Report-Sprache als Report-Locale) in die Caches für die Sprache „en“ oder „fr“, aber niemals in den Cache für „bar“.
  Problemfall war, dass Änderungen am Wert immer erst nach einem Server- und Client-Restart sichtbar waren.
  Dieser Konstrukt ist nicht auf unserem Mist gewachsen, sondern kommt von Java selbst.
- „.bo.“ im Bundle-Namen kann zu „Merkwürdigkeiten“ führen
  Auch hier war das Problem, dass Änderungen am Wert immer erst nach einem Server- und Client-Restart sichtbar waren. Der Bundle-Namen lautete „lu.PROJEKT.bo.Gruppe“, eine Entität, die es in besagtem Projekt nicht gab (sondern vom Core her kam). Nachdem der Bundle-Name auf „lu.PROJEKT2“ geändert wurde, waren Wert-Änderungen sofort sichtbar/abrufbar.

GUI

Grundlegende Dinge, die beim Umgang mit dem EDT (Event Dispatcher Thread) in der GUI zu beachten sind

- den UI-Status nicht außerhalb des EDT ändern
- keine langwierigen Operationen auf dem EDT ausführen

- keine Locks auf dem EDT acquiren
- den Zustand der Components nicht während des paintings ändern

Die SPU-Klasse (Swing Process Utility)

Die SPU-Klasse versucht, ein architektonisches Problem bei der Arbeit mit Swing zu lösen.

Das Erste, was jeder bei der Verwendung von Swing auf die harte Tour lernt, ist, dass man GUI-Objekte nur innerhalb des EDT (Event Dispatcher Thread) manipulieren sollte. Jedes Painting (Neuzeichnen) und jede Benutzerinteraktion findet dort statt. Die einzige Möglichkeit, Hintergrund- und/oder Multithreading zu nutzen, besteht darin, direkt auf den Daten zu arbeiten, da das sichtbare Update wiederum den EDT benötigt, um gerendert zu werden.

Das bedeutet für unser Framework, dass benutzerdefinierte Skripte laufen und „Dinge“ über eine öffentliche API tun können, wobei diese APIs sowohl innerhalb als auch außerhalb des EDT aufgerufen werden können. Und was noch schlimmer ist: Die für den Benutzer sichtbare API sollte unabhängig davon immer gleich aussehen. Anfangs führte dies zu öffentlichen Methoden, die - nachdem sie erkannt hatten, welcher Fall vorliegt - den Prozess entweder in Hintergrund-Threads oder in den EDT auslagerten. Das Fehlen einer `CompletableFuture`-Implementierung zu dieser Zeit ließ irgendwie keine wirklich elegante Wahl.

Um die Dinge weiter zu verkomplizieren, waren API-Aufrufe aus einem Skript manchmal gegenseitig von der Reihenfolge abhängig, in der sie abgesetzt wurden. Die Tatsache, dass einige davon in die Hintergrundverarbeitung gingen und andere in den EDT - wobei die Benutzer nicht wussten, was wohin ging (wie sollten sie auch...) - führte dazu, dass ein Refresh und der Versuch, den Fokus auf ein gerade neu eingeführtes Objekt zu setzen, oft nicht funktionierten.

Wir könnten diesen ganzen Ärger lösen, indem wir alles Single-Threaded machen. Wir haben sogar APIs mit 'sync'-Flags, um den Aufrufer auf das Ergebnis warten zu lassen. Dies ist jedoch anfällig für Deadlocks, wenn die aufgerufene Methode vom EDT aufgerufen wird und ihrerseits `invokeAndWait` auf einer anderen ausführt.

Aber es gibt viele Möglichkeiten, sich Probleme einzuhandeln, indem jede Komponente ihr eigenes Konzept hat (z.B. Tabelle vs. Text). Es ist einfach zu leicht, Operationen sequenziell falsch zu ordnen oder sich unwissentlich auf zufälliges Timing zu verlassen.

Die SPU-Klasse wird das hoffentlich ein für alle Mal beheben.

Wie?

Zunächst gibt es das Konzept von Warteschlangen (Queues) für Komponenten. Wenn eine Komponente aus vielen verschiedenen Delegaten zusammengesetzt ist, muss man

sicherstellen, dass sie alle dasselbe Objekt als `Queue ID` an diese API übergeben. Die `Queue ID` repräsentiert eine Warteschlange, die verwendet wird, um die Dinge in die korrekte Reihenfolge zu bringen, in der die Ausführung erfolgen soll.

Die Ordnung innerhalb der Warteschlange erfolgt in der Reihenfolge, in der sie eingereicht wurden. Noch offen ist, ob zukünftig das Konzept von „abhängigen Sub-Prozesse“ eine sinnvolle Ergänzung sein könnte.

Technische Referenz

Hier sind die wichtigsten Regeln und Konventionen für die Arbeit mit der SPU-Klasse (s.a. JavaDocs):

Das Queue-ID (qid) Konzept:

- Jedes Objekt kann als `qid` dienen (meistens eine UI-Komponente).
- **Wichtig:** Wenn `qid` als `null` übergeben wird, wird der Task als unabhängig betrachtet und sofort (bzw. konkurrent) ausgeführt.
- Alle Tasks mit derselben `qid` werden „zusammen verwaltet“.

Namenskonventionen der Methoden:

- `offEDT…`: Führt Code in einem Hintergrund-Thread aus.
- `onEDT…`: Führt Code auf dem Swing Event Dispatch Thread aus.
- `…v` (Suffix): Steht für "Void" (akzeptiert `Runnable` oder `Consumer` ohne Rückgabewert).
- `…AndWait`: Blockiert den Aufrufer bis zur Fertigstellung.
  - **Hinweis:** Dank Verwendung von `SecondaryLoop` ist dies auch aus dem EDT heraus sicher, ohne die UI einzufrieren.

Anwendungsbeispiele:

```
// Fire-and-Forget im Hintergrund
SPU.offEDTv(myComponent, () -> performHeavyCalculation());

// Berechnung im Hintergrund, dann UI-Update (Chaining)
SPU.offEDT(myComponent, () -> calculateData())
    .thenAccept(data -> SPU.onEDTv(myComponent, () ->
updateUI(data)));

// Blockierender Aufruf aus einer Button-Action (Sicher)
// Friert das Repainting der UI nicht ein dank SecondaryLoop
```

```
SPU.offEDTAndWait(myComponent, () -> saveToDb());
```

Wichtiger Hinweis für Unit-Tests:

Beim Testen dieser Klasse ohne echte GUI (Headless) kann es passieren, dass der sekundäre EDT zu früh herunterfährt. Dies führt dazu, dass Tests für `onEDTAndWait` fehlschlagen. *Lösung:* Einen `javax.swing.Timer` mit einem Timeout < 1000ms im Test erstellen (siehe `SPUTest` für Beispiele), um den EDT am Leben zu erhalten.

BOs und die GUI - Welcher Code gehört eigentlich wohin?

Generell kann man sagen, dass Code, der ausschliesslich dazu dient, die UI zu beeinflussen oder zu steuern, nicht in die BOs gehört. Dagegen sollte jegliche Business-Logik nur in den BOs programmiert werden.

Wenn UI-Actions BOs manipulieren, so gehört der dazugehörige Code in der Regel immer auch in die BOs selbst und nicht in den Skript-Code der Actions.

So kann man den Code auch für automatisierte Programmteile wiederverwenden und muss den Code nicht neu schreiben oder aufwändig später vom Skript in die BOs migrieren. Zudem ist der Code dann auch mit JUnit-Tests absicherbar und wird vom Compiler geprüft, was in der UI natürlich nicht der Fall ist.

Der Code für UI-Actions kann durchaus auch Objekte zur Steuerung der UI übergeben bekommen für den Aufruf aus der UI selbst. Hier sollte man darauf achten, immer sehr generische Interfaces wie BasicDialogI oder TransactionProviderI zu verwenden und den Code so zu gestalten, dass er auch ohne einen BasicDialogI funktioniert. Dies kann man so umsetzen, dass man z.B. ohne BasicDialogI stattdessen einfach nur loggt oder sich den Wert aus einer UI-User-Rückfrage anders besorgt bzw. einen default annimmt.

In der Entwicklungsphase oder der Anfangsphase eines Features kann es durchaus praktisch sein, den Code zuerst in der UI in Skripten vorliegen zu haben, damit man ihn ohne Updates ändern und anpassen kann. Allerdings sollte dieser Code, sobald das Feature stabil ist, auf jeden Fall in die BOs umgezogen werden.

Client-Context und Form-Context

Während es unproblematisch ist, sich aus dem Form-Context heraus den Client-Context zu besorgen(`ftx.getCtx()`), sollte man such von dem Client-Context niemals den Form-Context beschaffen(`ctx.getFtx()`). Dies liegt daran, dass man vom Client-Context aus mit `getFtx()` den globalen Form-Context bekommt, auf dem die gewünschten Aktionen entweder keine oder unbeabsichtigte Folgen haben können.

17.1.2. Dokumentation

Javadoc

Für neue Methoden und Klassen soll immer direkt auch Javadoc erstellt werden. Wenn man Methoden oder Klassen refaktoriert, soll auch fehlendes Javadoc ergänzt oder bestehendes aktualisiert werden.

Bekommt die zu beschreibende Methode Argumente übergeben, so ist darauf zu achten, dass im Javadoc die Einheit beschrieben wird um fehlerhafte Aufrufe zu verhindern (Parameter „*delay*“ vom Typ „long“ `⇒` 2000s dauern länger als 2000ms).

Wenn Javadocs von einer Superklasse oder einem implementierten Interface ererbt werden und daher in der Klasse keine Javadocs erstellt werden sollen, fügt man den Kommentar `inherit-doc` davor, damit niemand auf die Idee kommt, doppelte Javadocs zu erstellen.

Wir benutzen absichtlich kein leeres Javadoc mit nur `{@inheritDoc}` darin, da dies den Code aufbläht und die Javadocs zudem sogar verschlechtert (vgl. diesen Blog-Eintrag dazu).

AsciiDoc

Das für die (MyTISM- und Projekt-) Dokumentationsdateien verwendete Format ist AsciiDoc (Endung: `*.ad`). Grundlegende Informationen zu den Ablageorten für Dokumentationsdateien und zur Generierung bzw. Einbindung in den Buildprozess sind im Hitchhiker zu finden.

Beim Schreiben der Dokumentation gilt die Regel „ein Satz pro Zeile“, da diese Technik mehrere Vorteile bietet. Diese Vorteile sowie andere Empfehlungen und Best Practices sind im Dokument link:https://asciidoctor.org/docs/asciidoc-recommended-practices beschrieben.

Es empfiehlt sich, vor allem wenn Bilder eingefügt wurden, nicht nur die automatisch gerenderte HTML-5 Datei (etwa durch Ansehen der *.ad-Datei im Browser) zu kontrollieren, sondern auch die generierten HTML- und PDF-Dateien:

Bilder

- Bilder müssen im *.png oder *.jpg-Format vorliegen (FIXME zumindest werden aktuell nur diese Formate während des Builds ins Image-Unterverzeichnis des Docs-Verzeichnisses kopiert).
- In der Regel sollten Blockimages (`image::`) und keine Inline-Images (`image:`) verwendet werden, da es sein kann, dass bestimmte Features / Attribute nicht für Inline-Images unterstützt werden. (In den meisten Fällen ist dies korrekt; Inline-Images

sind ohnehin nur für Bilder, die von Fliesstext umgeben sein sollen, vorgesehen).

Besonderheiten bei der PDF-Generierung aus AsciiDoc-Dateien

- Um sicherzustellen, dass die sogenannten „Admonition-Icons“ gerendert werden, ist das Attribut „icons“ zusammen mit dem Wert „font“ im Document header anzugeben: `icons: font`(dies gilt nicht nur für die Generierung der PDFs, sondern auch der HTML-Dateien).
- Direkt am Bild muss in dem Imagelink nachstehenden eckigen Klammern `pdfwidth` spezifiziert werden; keine oder eine einfache Breitenangabe via `width` genügt hier nicht. Beispiel:
  `image::images/ticket/modules-ticket_mailfetcher.png[pdfwidth=100%]`
- Es kann sein, dass die `pdwidth` nicht ausgelesen werden kann, wenn in den eckigen Klammern zusätzlich ein Alternativtext angegeben ist (→ FIXME nochmals testen)

17.1.3. Unit-Tests

Neue Tests sollten in Groovy implementiert werden. Existierende Tests können von NetRexx in Groovy übersetzt werden, wenn es die Zeit zulässt oder größere Anpassungen darin nötig sind.

Testklassen sollten nach der Klasse die sie testen benannt werden und den Postfix „Test“ erhalten.

Tests müssen nicht unbedingt Compilestatic sein, da sie kurz nach dem Compilieren eh ausgeführt werden und etwaige Fehler kurz danach zur Laufzeit auffallen. Zudem erlaubt CompileDynamic auch den Zugriff auf private Variablen, was im Zuge mancher Tests praktisch sein kann.

Allgemeine Vorlage für neue Tests bei Projekten, die das Modulsystem verwenden

```
package @BOPACK@

import de.ipcon.db.core.Transaction
import de.ipcon.db.testing.TestBOLoader
import org.apache.log4j.Logger


class Example@MODULETAG@Test extends GroovyTestCase {

   TestBOLoader bol = TestBOLoader.of('@PRJPACK@')
   Transaction tx
```

```
   Logger log = Logger.getLogger(Example@MODULETAG@Test.class)


   /**
    * Setup transactions, gets executed before each test
    */
   protected void setUp() {
      tx = bol.createTransaction()
      // injects existing initialdata and calls initEnvironment
for those classes (should reset static variable caches)
      // TBO is Mandatory to update the Cacheloader, used by
several 'byTid' or 'byName' Methods.
      tx.injectInitialDataFor TBO.class
      // create needed support structure

      // simluateSave, that way it looks like that's the all just
cached values.
      tx.simulateSave()
   }


   /**
    * Cleaning up now unused data, gets executed after each test
    */
   protected void tearDown() {
      /* fixes id clashing when running multiple tests due to
caching,
         we need to clear all BOs linked in static variables, as
those can leak to other tests. */
      tx.close()
      TBO.flushCaches()
   }


   //-----------------------------------------------------------
   // Tests SaveVetos
   //-----------------------------------------------------------
```

```
   /**
    * Some descriptive Comment, the method name starts with
'test'
    * and should contain the word 'should' for description
    */
   void
testDescriptiveMethodNameShouldExplainExpectedFunctionality() {
      //additional setup
      //the actual test
      //compare to expected result
   }
}
```

Es darf auf keinen Fall ein Default Logger konfiguriert werden via `Log4jHelper.createDefaultLogger()`, da dies 1. nicht mehr erforderlich ist und 2. die auf `WARN` voreingestellte Protokoll-Ebene sonst untergraben würde, die in `compile-and-run-tests.groovy` gesetzt wird (dies führt dann zur Ausgabe aller INFO-Log-Nachrichten bei der Ausführung aller Tests!)

Weitere Infos zur Konfiguration des Loggings beim Bauen siehe Einarbeitungsdokumentation.

L10n in Tests

Um L10n-Nachrichten in Tests zu vergleichen, bitte keine festen Strings benutzen, sondern die Keys immer vom L10n-System auflösen lassen. Andernfalls kann es sein, dass der Test auf einer Maschine mit einer anderen Default-Locale fehlschlägt. Zum Testen lohnt es sich, den Key als Konstante in der NetRexx-Klasse zu exponieren und dann im Test gegenzuprüfen, beispielsweise so:

```
errMsg = L10n.msg(MeinBO.L10N_KEY_MISMATCHED_PARAMS, [paramA,
paramB] as Object, [MeinBO] as Object)
```

Bitte darauf achten, dass die 3-Parameter Variante von L10n.msg benutzt wird, da sonst die Klasse und die zugehörige L10n-Ressource nicht gefunden und der Key im Test nicht aufgelöst wird.

Troubleshooting

Wenn Tests fehlschlagen, weil die Initialdaten in Tests von sehr großen Projekten fehlen, sollte man die Logausgabe auf ein „too many open files“ durchsuchen. Abhilfe schafft hier der Eintrag des Werts `fs.file-max = 2097152` in die Datei `/etc/sysctl.conf` und ein anschließendes Neuladen via `sysctl -p ` als root. Weiterführende Infos und Troubleshooting dazu sind auch in der „Linux Basics“ Doku zu finden im Abschnitt „Too many open files in system“.

Common Pitfalls

- Wenn man den Verdacht hat, dass sämtliche Tests einer Datei gar nicht ausgeführt werden, sollte man die Gross-/Kleinschreibung des Dateinamens der Test-Klasse überprüfen (siehe die zugehörige file mask in der Test-Section der build.xml)
- Ebenfalls sollte man in Test-Methoden darauf achten, dass die Methode *queryBO* in Groovy als zweiten Parameter ein *Object[]* erwartet. Daher am besten explizit nach *Object[]* casten, ansonsten kann Groovy eine Liste als Object-Array mit einem Object (in diesem Fall die Liste) interpretieren und würde *toString()* auf der kompletten Liste anstatt den einzelnen Elementen aufrufen.
- Wenn man die Werte von Konstanten in Interfaces verändert, dann reicht es nicht, das Interface neu zu compilieren. Man sollte dann auf jeden Fall einen clean rebuild machen (oder zumindest von dem Teil des Projekts, der das Interface benutzt), ansonsten ändern sich die Konstanten nicht, da sie beim Übersetzen in die Klassen kopiert wurden.
- Fehlermeldungen der Art „Objekt mit Id 100 sollte ein Mandant sein, habe aber ein Währung-Objekt gefunden“ weisen auf Id-clashes hin.
  - ist die Variable lastSimulatedId static?
  - Wird in der tearDown()-Methode `TBO.flushCashes()` und das `simulateSave()` aufgerufen? Das löscht den Cache und verhindert, dass Objekte eines Tests Einfluss auf den nächsten Test haben.
- Ein in den initialdata hinterlegtes Objekt, z. B. `EURO` bei Aufruf von `Waehrung.getEuroInstance()` kann nicht gefunden werden, `tx.getExistingBO(Waehrung, Tid:'EURO')` funktioniert aber.
  - Hier wurde vergessen TBO in der setUp()-Methode neu zu initialisieren. Der TBO-Cache enthält wegen flushCaches() keine Objekte mehr, hat aber noch den alten CacheLoader hinterlegt, der Objekte von der neu erstellten Transaktion nicht kennt.
- Dezimalwerte in der initialdata (z.B. `1.5`) werden mit Punkt (`.`) als Dezimaltrennzeichen falsch importiert, da die Nachkommastellen abgeschnitten werden (aus `1.5` wird `1`).
  - Aktuell verwendet das Parsing fix die deutsche Locale. Daher muss das Komma (`,`) als Dezimaltrennzeichen genutzt werden. S.a. Ticket 197181238.

- Sollte beim Rendern eines Reports (via `createPDFFile()`) eine `ClassNotFoundException` auftauchen (z.B. für `net.sf.jasperreports.engine.fill.JRTemplatePrintLine`) oder bei Verwendung von SVGs eine `java.io.NotSerializableException` für `net.sf.jasperreports.renderers.BatikRenderer$Container` und der Stacktrace sieht danach aus, als wäre der Virtualizer dafür verantwortlich, kann man den Parameter `PrintingServices.AVOID_TEMP_FILES` hinzufügen. Daraufhin wird der Virtualizer für diesen Report nicht mehr verwendet. Alternativ kann man den Virtualizer auch global via der Einstellungen-Variable `jasperReports.reportVirtualizer.disabled` deaktivieren.
- Report-Parameter dürfen kein Punkte in ihren Namen haben, da diese in den „Scripted Attributes“ als Variablen eingeblendet werden und somit den Anforderungen für gültige Variablennamen in Groovy entsprechen müssen. Diese Parameter werden vom System automatisch ausgefiltert und stehen in den „Scripted Attributes“ nicht zur Verfügung. Es gibt dann jedoch zumindest eine Fehlermeldung dazu im Log.

17.1.4. Versionsverwaltung

Tagging von selbstcompilierten Bibliotheken

Wenn eine jar für das `nrx/lib`-Verzeichnis neu gebaut wird, sollte der Sourcecode-Stand im CVS entsprechend getaggt werden, damit ersichtlich ist, welcher Codestand in die jar eincompiliert wurde. Als Konvention sollte man zuerst die neu gebaute jar nach `nrx/lib` committen und anschließend den Sourcecode mit einem Tag nach dem Muster `v_xx_yy` versehen, wobei xx die major und yy die minor Version der jar nach ihrem Commit ist.

CVS

> Es dürfen normalerweise nur Dinge ins CVS eingespielt werden, die vorher (ausreichend - was auch immer das nun heißen mag...) erfolgreich getestet wurden. Das gilt insb. für Änderungen die mehrere oder gar alle Projekte betreffen, also im Modulsystem (/com/oashi/modules) und noch mehr im Core (/de/ipcon) und bei den Bibliotheken (/lib).
> Dies gilt in der Regel nur für den HEAD, in Branches, sofern sie keine laufenden Systeme betreffen, kann auch work in progress Code committet werden.

Nach jedem Commit gibt es eine E-Mail, die die Commit-Message und das entsprechende diff enthält, damit jeder sehen kann, was geändert wurde. Nicht nur bietet das bei E-Mail-Programmen mit Suchfunktion eine gewisse Schnellsuche, wer wann was geändert hat; die Kollegen weisen auch ab und zu auf Dinge hin, die man hätte besser machen können. Kein falscher Stolz — es entstehen hierdurch auch wertvolle Erkenntnisse.

Gute Commits sind…

- zeitnah nach der Änderung.
- kleinschrittig (alles, was zu einer Änderung gehört, aber nicht mehr).
- nicht alle am Tagesende auf einmal.

Gute Commit-Nachrichten…

- … sind in englischer Sprache (MyTISM-Konvention).
  Dies gilt auf für deutschsprachige Dokumente wie z.B. die Einarbeitungsdoku oder den Hitchhiker - man muss sich dann keine Gedanken um Ausnahmen machen.
- … fassen die Semantik(!) der Änderungen so kurz wie möglich, aber so umfassend wie nötig, zusammen.
- … beginnen mit einer Betreffzeile, die mit einem „vorangestellten Schlüsselwort“ kategorisiert wird, und der eine kurze Beschreibung der Änderung in Imperativform und ca. max. 50 Zeichen folgt, um einem Leser schnell zu vermitteln, worum es geht.
- … haben optional einen darauf folgenden Block, der Details des Commits beschreibt
- … referenzieren am Ende zusätzlich die Tickets, mit denen sie assoziiert werden sollen, mittels `#t[nr]` (z. B. `#t12345`).

Als Schlüsselwörter werden vorgeschlagen:

- change: Verhalten des Programmteils wurde geändert.
  change: in lookupPostenBackReferences return all Posten that indirectly reference startPosten if second parameter is null
- cleanup: Restrukturierung oder „Aufräumen“ des Quellcodes (hoffentlich ;-) ohne Verhaltensänderung.
  cleanup: remove nop-lines (after review by PB)
- refactoring: Größere Restrukturierungen oder Umbauten am Code (hoffentlich ;-) ohne Verhaltensänderung.
  refactoring: replace recursion by iteration in method countDown
- cosmetics: „Verschönerungen“ ohne Verhaltensänderung.
  cosmetics: Remove "upload" icon for Ablagen which cannot contain files anyway
- fix: Es wurden Probleme beseitigt.
  fix: add a missing description - otherwise calling describe() on that object returns an empty string
- fix regression: Es wurden bereits zuvor behobene Probleme, die sich wieder eingeschlichen haben, beseitigt.
  fix regression: re-add lost else-block, which was accidentally removed in v1.23

- l10n: Es wurden Änderungen vorgenommen, die die Lokalisierung betreffen. Dies hebt sich von „change“ dadurch ab, dass es sich hier nur um Benachrichtigungen des Benutzers handelt (wie etwa Strings, oder das Abholen/Lokalisieren von Strings durch l10n.msg() oder ähnlich) und das eigentliche Verhalten des Programms gleichbleibt.
  l10n: add missing french translations
- improvement: Die Datei wurde verändert um die Nutzerbarkeit oder das Arbeiten mit der Datei zu verbessern.
  improvement: Add table of contents to hitchhiker guide to improve finding one's way trough the document
- new: Es wurden neue Dateien oder neue Features hinzugefügt.
  new: add a script to initialize some demo data for the ReportingPlatform.

Dem Schlüsselwort folgt eine kurze Beschreibung der Änderungen des Commits. Die Kurzbeschreibung sollte in der Imperativform geschrieben sein, nicht mit einem Punkt enden und wenn möglich maximal 50 Zeichen umfassen (Ausnahmen bestätigen die Regel). Eine gute Kurzbeschreibung vervollständigt den Satz „This commit will…“, also z.B. „enhance the section about commit messages“. Eine schlechte Kurzbeschreibung ist „fix bug“, denn diese schließt den Satz nicht sehr schön ab („This commit will fix bug“ - *srsly?!*).

Der auf die Kurzbeschreibung folgende Block enthält das Wesentliche der Commit-Nachricht und ist der Ort, an dem man auf Einzelheiten der vorgenommenen Änderungen und den größeren Kontext eingehen kann. In diesem Block kann man erklären, warum man diese Änderungen vorgenommen hat, warum man die Änderungen genau auf diese Art implementiert hat, und alles andere, was Leuten helfen würde, den Denkprozess hinter dem Commit zu verstehen. Dinge, die aus den Code-Änderungen im Diff offensichtlich sind, sollten nicht noch einmal wiederholt werden. Es ist nicht nötig, die Änderungen Zeile für Zeile zu erklären; es ist wichtiger, Details auf höherer Ebene zu erklären, die beim Lesen des Codes vielleicht nicht offensichtlich sind. Das Ziel der Nachricht besteht letztlich darin, einen Kontext für den Entwicklungsprozess rund um diese Änderung zu schaffen, der sich in erster Linie auf die Beweggründe und Ziele des Commits bezieht.

Beispiel

```
change: enhance the section about commit messages

 - added key word "fix regression" and explanation / example for
that
 - elaborated and refined rules for short message
 - added part about the main body of the message and the
conventions for that
```

```
#t47110815
```

Für sehr kleine Commits, wie das Korrigieren eines Tippfehlers, wird natürlich kein weiterer Block benötigt, da die Kurzbeschreibung informativ genug sein sollte.

In der abschließenden Zeile der Commit-Nachricht kann man noch nützliche Metadaten zum Commit unterbringen, wie z.B. die Ticketnummer(n), Namen von Co-Autoren und zusätzliche Links. Dies kann helfen, wichtige Informationen, die sich auf diese Änderung beziehen, miteinander zu verknüpfen (so wird die Email zum Commit etwa von einem Service automatisch als TicketCommit-Objekt an das hier referenzierte Ticket gehängt).

Empfohlene Einstellungen: In SmartCVS: `Edit→Preferences` und dann bei `Actions→Commit` ein Häkchen bei „Remind me to enter a log message“ setzen.

Umgang mit Entwicklungszweigen / Branches im CVS

Für Entwicklungen im Core, Modulsystem oder in Projekten, die nicht im HEAD erfolgen sollen, muss in der Regel ein neuer Entwicklungszweig („Branch“) im CVS eröffnet werden.

Um den Aufwand für das Zusammenführen („Mergen“) des Codes vom bzw. in den HEAD so gering wie möglich zu halten, dürfen nur die Dateien verzweigt werden, die wirklich verändert wurden, nie ein ganzes Verzeichnis „auf Vorrat“.

Änderungen an L10n-Bundles in resources/l10n sollten nicht auf den Branch, sondern direkt im HEAD committed werden, auch wenn sie dort noch nicht benötigt werden oder dort temporär nicht korrekt sind, da das Zurückmergen in den HEAD viel zu aufwändig ist.

Erstellen eines Entwicklungszweiges / Branching

Zur Erstellung eines Entwicklungszweiges markiert man die Dateien, die in einen Zweig gebracht werden sollen und wählt im Menü `Tag / Branch` den Punkt `Create Branch` aus, oder drückt direkt `Strg+B`, nachdem man die Dateien markiert hat.

Im sich öffnenden Dialog `Create Branch` gibt man den Namen des Entwicklungszweiges ein und deaktiviert die Option `Check that not modified` bzw. via `Alt+C`.

Auschecken / Check-out

Um den aktuellen Stand des Zweigs und den aktuellen HEAD-Stand der nicht verzweigten Dateien zu bekommen, muss man beim Auschecken des Quellcodes in SmartCVS ein `Switch (Special Update)` vornehmen und kein normales Update der Datei(en). Diese Option befindet sich im Menü `Modify` oder als Tastenkombination standardmäßig unter

Strg+Shift+U.

Im Dialog des `Switch (Special Update)` muss man den Punkt `Retrieve Tag/Branch (new sticky)` anklicken oder mit `Alt+T` anwählen und dort den Entwicklungszweig-Namen eingeben; außerdem muss man weiter unten die Option `If no matching revision is found, use the most recent one` anklicken oder mit `Alt+I` anwählen. Nach dem Update werden alle Dateien im CVS mit sticky `branch-name` angezeigt (auch solche, die nicht gebranched sind).

Alle weiteren Checkouts müssen ebenfalls mit dem `Switch (Special Update)` vorgenommen werden, da u.U. die Datei sonst bei einem „normalen“ Update gelöscht wird.

Einchecken / Committen / Check-in

Wenn eine **neue Datei** erstellt wurde, kann diese - sofern das gesamte Verzeichnis wie oben beschrieben mit dem Entwicklungszweig-Tag ausgecheckt wurde - ganz normal neu eingecheckt werden und landet sofort nur im Zweig.

Wenn ein **neu erstelltes Verzeichnis** (auch mit neuen Dateien) in den Branch eingecheckt werden muss - das übergeordnete Verzeichnis mit dem Entwicklungszweig-Tag auschecken. Danach das neue Verzeichnis ganz normal einchecken.

Falls das Verzeichnis nicht komplett im Entwicklungszweig ist, muss man ggfs. etwas tricksen, um eine neue Datei direkt in den Branch committen zu können, indem man eine `Tag`-Datei im `CVS`-Unterverzeichnis des entsprechenden Verzeichnisses erstellt und dort als Text `Tbranchname` hinein schreibt, also z.B. `Tstrom-va-rework-kw49`; nach dem Commit der Datei muss diese Tag-Datei wieder aus dem entsprechenden CVS-Verzeichnis gelöscht werden.

Wenn eine Datei modifiziert wurde und diese sich bereits im Zweig befindet (**erkennbar an der Versionsnummer mit 4 Teilen, z.B. 1.1.2.1**), kann man die Datei ganz normal wie immer committen.

War eine existierende Datei bisher noch nicht im Branch, kommt die folgende Fehlermeldung, wenn man diese Datei committen möchte:

```
cvs server: Up-to-date check failed for
`de/ipcon/tools/TextTools.nrx'
cvs [server aborted]: correct above errors first!

Command Aborted.
```

In diesem Fall muss man die Datei zunächst in SmartCVS wieder vom sticky-tag befreien mittels `Switch (Special Update)` und dort die Option `Main trunks head (reset sticky)` anklicken oder mit `Strg+M` anwählen. Anschließend muss an der Datei wie oben beschrieben der Entwicklungszweig erstellt werden. Danach kann man die Datei ganz normal committen und diese gelangt in den Zweig. Die bereits durchgeführten Änderungen gehen dabei nicht verloren. Der Vorteil dabei ist, dass man nicht etwas aus Versehen in den HEAD committen kann, wenn man eigentlich im Zweig arbeitet.

Möchte man etwas willentlich in den HEAD committen, so lässt man den Schritt des Verzweigens einfach weg und committet nach HEAD. Anschließend sollte man wieder mittels `Switch (Special Update)` auf den sticky-Zustand wechseln, damit weitere Commits der Datei nicht mehr nach HEAD gelangen.

Zum Löschen von Dateien nur im Branch muss die entsprechende Datei erst gebranched werden und erst anschließend darf man das remove committen, damit das remove nicht im HEAD passiert.

Beim Löschen steht in der zugehörigen CVS-Mail immer, dass die Datei vom HEAD gelöscht wurde, auch wenn sie ausschließlich vom Branch entfernt wurde.
(Stand 01/2022)

Zusammenführen von Code / Merging

Entwicklungszweig updaten

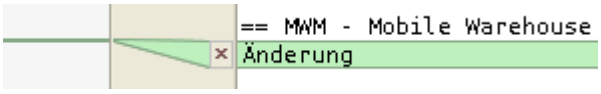
Um den Code von Dateien, die sich im Entwicklungszweig befinden, mit dem Code aus dem HEAD zusammen zu führen, muss ein Merge durchgeführt werden.

Vor dem Merge sollte man den Branch ***ohne*** Fallback auf HEAD auschecken (`If no matching revision is found, use the most recent one` ist abgewählt) und nur die Dateien, die tatsächlich im Branch sind, mergen. Damit erspart man sich, dass SmartCVS alle Dateien, die sich nicht im Entwicklungszweig befinden, als *Neu* markiert. Nach dem erfolgreichen Merge kann man dann den Branch wieder *mit* Fallback auf HEAD auschecken, damit man den Stand wieder normal compilieren und damit weiterarbeiten kann.

Dazu markiert man die gebranchten Dateien oder das gebranchte Verzeichnis, das man mergen möchte und wählt im Menü `Modify` den Punkt `Merge` aus. Im sich öffnenden Dialog `Merge` gibt man unter `Merge from` einfach `HEAD` ein.

Es ist nicht unüblich, dass beim Zusammenfügen Konflikte zwischen den Zweigen bestehen. Oftmals sind das Stellen in den Dateien, welche in den beiden Zweigen seit dem

„Branchen“ verändert wurden. Um die Konflikte zu beheben muss man manuell die Stellen korrigieren, welche die Konflikte verursachen. Diese Korrektur kann auch direkt im SmartCVS durchgeführt werden, indem man die Datei, welche den Konflikt verursacht, mit Hilfe der Aktion „Compare“ öffnet. In diesem Vergleichsbildschirm können dann einzelne Änderungen mit einem Klick auf das kleine *x* an der geänderten Stelle entfernt werden.



Einen solchen Konflikt wird im folgenden Beispiel erklärt:

Gegeben sei folgende Situation:

1. Das Skript `UnserTest.groovy` hatte in der Version 1.0 die Zeile `Map<String, Integer> intMap = [:]`
2. Nachdem wir die Datei ausgecheckt haben, haben wir lokal die Zeile in `HashMap<String, Integer> intMap = [:]` geändert.
3. Währenddessen hat allerdings ein anderer Benutzer die Zeile in `TreeMap<String, Integer> intMap = [:]` geändert und bereits committet.
4. Wenn wir unsere lokale Version nun committen wollen, kommt es zu einem Konflikt.

Der entsprechende Konflikt sieht wie folgt aus:

```
<<<<<<< UnserTest.groovy
   HashMap<String, Integer> intMap = [:]
=======
   TreeMap<String, Integer> intMap = [:]
>>>>>>> 1.1
```

Wie zu sehen ist, wird unsere Änderungen im obigen Teil angezeigt. Die bereits committete Version in der unteren Zeile. Weitere Konflikte sind, wie auch dieser hier, mit der Zeichenfolge `>>>>>>> Revision XXX` gekennzeichnet.

Im schlimmsten Fall muss man die Konflikte manuell per Hand in einem beliebigen Editor korrigieren.

Auftretende Konflikte können leider oftmals fälschlich durch SmartCVS gemeldet werden. Hier bietet es sich an, zunächst im „Log“ der Datei nachzuschauen, ob es im Branch bereits einen `merge from HEAD` commit gab, der jünger ist, als die neueste Version im HEAD - in diesem Falle kann man den Konflikt einfach ignorieren und den lokalen (Konflikt-)change

> mittels `Revert` zurücksetzen.

Nachdem alle Konflikte gelöst wurden, committed man die Änderungen. Wenn man beim Lösen eines Konflikts bemerkt, dass im Branch keine Änderungen nötig sind (weil z.B. der Code, der in HEAD geändert wurde, gar nicht mehr im Branch existiert), sollte man trotzdem einen „leeren“ Commit mit entsprechender Nachricht im Branch vornehmen, damit man bei späteren Konflikten anhand des Logs sehen kann, dass dieser Konflikt bereits kontrolliert wurde (`Commit` → `Advanced Options` → `Force commit even if there are no changes`) - danach das Committen von „no changes“ am Besten wieder deaktivieren. Als commit message für einen Merge hat sich die Nachricht „Merge from HEAD“ eingebürgert.

> Diese Vorgehensweise ist nötig, da man in SmartCVS nicht auf nur die Dateien filtern kann, die sich *wirklich* im Zweig befinden, und diese daher nur mit der oben angeführten Vorgehensweise einigermaßen komfortabel mergen kann.

Rebase eines Entwicklungszweigs

Zusätzlich bietet es sich an, ab und zu ein Rebase des Branches zu machen, um falsche Konflikte zu vermeiden, d.h. der Branch wird verworfen und die Änderungen aus dem Branch, basierend auf dem dann aktuellen HEAD-Stand, in einen neuen Branch committet.

Dazu nimmt man zunächst einen merge from HEAD vor, benennt dann das Projektverzeichnis um und checkt den HEAD frisch in das normale Verzeichnis aus. Anschließend kopiert man die Änderungen aus dem Verzeichnis, das den Branch enthält, mittels `rsync` und den Optionen `-rC` in das Projektverzeichnis, z.B. `rsync -rC myproject_branch/* myproject/`.

Nachdem man das Projektverzeichnis bzw. insbesondere die rot markierten Dateien nochmals mit `Update` aktualisiert hat, verbleiben die Dateien, die Änderungen enthalten, rot markiert und man kann diese neu branchen und dann in den neuen Branch committen.

> Neue Dateien (mit einem Blatt mit Sternchen markiert) bekommt man in den neuen Branch, indem man einen Trick beim Check-In anwendet.

Entwicklungszweig zurück in den HEAD mergen

Für den Merge nach HEAD ergibt sich dieses Problem nicht, da man ja nur die Änderungen aus dem Entwicklungszweig nach HEAD übernimmt. Im Entwicklungszweig entfernte oder neu hinzu gekommene Dateien werden während des Merge korrekt übernommen.

Vor dem Mergen sollte sicherheitshalber der HEAD-Stand getaggt werden, so dass sich

unerwartete, durch das Mergen verursachte Fehler leicht zurückdrehen lassen.

Als Vorlage für den Tagnamen bietet sich folgendes Muster an: `pre-merge-[tagname]`. Als commit message für einen Merge hat sich die Nachricht „Merge to HEAD“ bzw. „Merge from branch …“ etabliert.

Falls nur eine einzelne **neue** Datei in HEAD übernommen werden soll kann folgendermaßen vorgegangen werden:

- Branch komplett auschecken.
- CVS Log der Datei öffnen (`STRG+L`, rechte Maustaste → `Log` oder Button `Log` in der Hauptleiste).
- Dieses Log Fenster geöffnet lassen und HEAD auschecken.
- Rechte Maustaste über der gewünschten Branch Version → `Check Out As…`, Ordner für die Datei wählen.
- Die neue Datei per add/commit in den HEAD Zweig einchecken.

Tagging

Spezialfälle

Manchmal ist es nötig, Dateien „nachzutaggen“ - etwa wenn Code noch nachträglich in ein Release einfliessen soll, obwohl dies zum Zeitpunkt des Taggens des Release-Codestands noch nicht bekannt war. Neben einfachen „move Tag“ oder „add Tag“-Operationen, um die gewünschte Version einer geänderten oder neuen Datei zu markieren, gibt es den Fall, dass eine Datei gelöscht wurde und diese NICHT mehr in das Release aufgenommen werden soll. Hier genügt es, das Tag zu entfernen („remove Tag“).

> Nur allzu leicht passiert es, wenn man mit Hilfe der Smart-CVS UI taggt, dass der Fokus versehentlich falsch gesetzt ist: beispielsweise auf dem ganzen Verzeichnis statt auf der gewünschten Datei. Insbesondere nach einer „remove Tag“-Operation kann es dann schwierig werden, den vorigen Tagstand zu rekonstruieren. Daher sollte man, bevor man ein potenziell „gefährliches“ Tag-Kommando absetzt, immer einen Backup-Tag auf der nächsthöheren Ebene erzeugen.

Skripte im CVS ablegen

- Skripte zum Datenimport liegen oft in einem Verzeichnis namens `import`. Es kommt aber auch vor, dass dieses Verzeichnis `datenquellen` heißt.
- Skripte für wiederkehrende Aufgaben (MyTISM-interne „Cronjobs“) sollten im CVS mitgepflegt werden, entweder per initialdata oder wenigstens per Textdatei (wird dann

in die GUI einkopiert). Der Ort für diese Dateien wird oft `services` genannt. Holt der Dienst Daten aus einem anderen System ab (zum Beispiel das periodische Lesen von Dateien auf einem Samba-Share des Fremdsystems, um damit Daten im MyTISM-System zu aktualisieren), kann das Verzeichnis auch `dx` heißen („Data Exchange“).

- Für Skripte, die durch den `run-bs` genannten Mechanismus händisch (einmal oder im Zuge einer Wartung wiederkehrend) ausgeführt werden müssen, wie beispielsweise ein CSV-Export von Tabellen auf Anfrage, kommen Verzeichnisnamen wie `bs` (Business Service), oder `tools` in Frage.

Natürlich kann man bestehende Skripte in passendere Verzeichnisse im CVS umziehen. Man sollte aber berücksichtigen, dass hierbei die Historie der CVS-Datei verloren geht und das entsprechend behandeln.

Wenn die Historie bewahrt werden soll, dann bitte an Patric wenden - er kennt den Kniff, um das zu bewerkstelligen. Da man dabei einiges kaputt machen kann, wird die Vorgehensweise bewusst nicht „an die grosse Glocke“ gehangen.

17.2. MyTISM Coding Conventions

Die gemeinsame Arbeit an der MyTISM-Codebasis erfordert absolute syntaktische und stilistische Disziplin. Da Quelltext weitaus häufiger gelesen als geschrieben wird, ist ein einheitlicher Programmierstil über alle Module hinweg unerlässlich.

Neuer Quelltext muss ausnahmslos gemäß den nachfolgenden Regeln erstellt werden. Ausnahmen gelten ausschließlich für Abschnitte, die explizit als „optional“ markiert sind; hier liegt die Anwendung im fachlichen Ermessen der Entwicklerin oder des Entwicklers.

Wird bestehender Quelltext bearbeitet, muss dessen Stil zwingend an diese Konventionen angepasst werden, sofern dies im Rahmen des aktuellen Refactorings vertretbar ist (siehe „Pfadfinder-Regel“). Bis zur vollständigen Umstellung der Codebasis ist es unvermeidlich, dass noch inhomogene „Inseln“ verschiedener Programmierstile existieren, die jedoch bei jedem Kontakt sukzessive eliminiert werden sollen.

17.2.1. Charset/Encoding

Für die Erstellung und Bearbeitung von Quelltexten sollte stets das Zeichensatz-Kodierungsschema UTF-8 verwendet werden.

17.2.2. Namenskonventionen

Die Benennung von Entitäten und Attributen sollte grundsätzlich in der Projektsprache erfolgen, was in aller Regel Deutsch ist.

Interfaces, Parameter sowie Hilfsvariablen, -methoden und -klassen sollten jedoch immer in Englisch benannt werden. Es können Ausnahmen hiervon in Absprache mit der Projektleitung getroffen werden, sofern dies sinnvoll erscheint.

Wenn Bezug auf einen deutschen Entitäts- oder Attributnamen genommen wird, sollte dieser Name als Zitat aus dem Schema betrachtet werden und wird unverändert in den ansonsten englischen Namen übernommen, zum Beispiel `recalcArtikel()`.

Die unterschiedliche Verwendung von Deutsch und Englisch hat den Zweck, dass schema-bezogene Getter und Setter leicht von allen anderen Methoden im Code unterschieden werden können. Außerdem erleichtert es nicht-deutschsprachigen Teammitgliedern die Arbeit, wenn nur die schema-bezogenen Methoden auf Deutsch sind, während alles andere auf Englisch bleibt.

Es ist zu beachten, dass die Konventionen möglicherweise noch nicht vollständig einheitlich umgesetzt sind, aber es dürfen keine neuen Altlasten in Form von deutschen oder „denglischen“ Methodennamen im Code erzeugt werden.

Packages

- Packages dürfen Unterstriche im Namen enthalten. Ansonsten gelten die normalen Regeln für Java-Packages.

Klassen

- Klassen sollten in der Regel keine Unterstriche im Namen enthalten. Dies gilt auch für Entitätsnamen. Ausnahmen bilden technische oder externe Erfordernisse. Für solche Entitäten muss ein abweichender Datenbankname ohne Unterstriche angegeben werden.

Unter keinen Umständen darf ein Unterstrich verwendet werden, um lediglich eine Art von Präfix zur „virtuellen“ Package-Unterteilung zu simulieren.

- Die Sprache für Klassennamen, die nicht zu einer Entität gehören, sollte in der Regel Englisch sein, dies gilt auch für Interfaces.
- Wenn Entitätsklassen aufgrund ihrer Länge in mehrere Klassen aufgeteilt werden, sollten die ableitenden Klassen mit einem aussagekräftigen englischen Suffix versehen werden, das den ausgelagerten Aspekt beschreibt. Als letztes Suffix sollte immer noch „Aspects“ angehängt werden, zum Beispiel „KundeContractAspects“.
- Klassennamen von Interfaces müssen immer mit einem „I“ enden und sollten wie in Java üblich Adjektive (beispielsweise „Comparable“, „Iterable“) oder Nomen sein, die eine Familie von Klassen beschreiben (zum Beispiel „Map“, „Collection“). Beispiele bei

uns sind „ServiceI“ und „CPoxBuilderI“.

- Abstrakte Klassen sollten mit dem Präfix „Abstract“ beginnen, sofern sie eine skelettartige Implementierung eines Interfaces darstellen. Ein Beispiel wäre `class AbstractBackendSession implements BackendSessionI abstract`. Diese Konvention entspricht dem Ratschlag von Joshua Bloch in "Effective Java": „By convention, skeletal implementations are called AbstractInterface, where Interface is the name of the interface they implement. [...] the Abstract convention is now firmly established.“
- Abstrakte *Entitäten* sollten dagegen *nie* mit dem Präfix „Abstrakt/e“ beginnen, da dies für Entitäten wenig(er) Sinn ergibt und redundant ist. Die Eigenschaft, ob eine Entität abstrakt ist oder nicht, kann leicht über die Methode `EntityI#isUserAbstract()` ermittelt werden. Das Beispiel „AbstraktePerson“ war eine Fehlentscheidung, die Entität wäre besser „JuristischePerson“ genannt worden.
- Klassennamen von Enums sollten niemals das Suffix „Enum“ erhalten, wie es generell für Java empfohlen wird und dem auch alle offiziellen Beispiele folgen.

Imports

- Imports von Klassen sollen in der Regel einzeln und nicht als *- oder rekursiver *-Import geschehen. Die Motivation dafür ist eine bessere Klarheit des Codes in Bezug darauf, welche Klassen aus welchen Packages exakt verwendet werden.
- Eine Ausnahme stellen eng begrenzte Packages dar, z.B. ein Package, das nur Exceptions enthält, die in der Klasse häufig verwendet werden und daher sonst zu sehr vielen, unzweideutigen Einzelimports führen würden.

Methoden

Folgende Namenskonventionen sollten für Methoden eingehalten werden:

- Die Sprache für (Hilfs-)Methodennamen ist in der Regel Englisch.
- Referenzen auf Entitäts- oder Attributnamen innerhalb von Methodennamen bleiben auf Deutsch und werden nicht auf Englisch übersetzt. Dies geschieht, da sie als Zitat aus dem Schema betrachtet werden und eine Übersetzung zu Verwirrung und Inkonsistenzen führen könnte.
- Boolsche Instanz-Methoden, die ein Objekt *auf Gleichheit mit einer Standard-Instanz überprüfen* (in der Regel eine automatisch aus den Initialdaten angelegte Instanz), sollten immer `ist` mit der angehängten deutschen Bezeichnung des gesuchten Objekts heißen, zum Beispiel `istJaehrlich` oder `istDeutschland`.

Statische Methoden

Statische Methoden sollten gemäß den folgenden Regeln benannt werden:

- `of` sollte als Präfix oder Name verwendet werden, wenn ***neue persistente Instanzen*** erstellt werden (Factory-Methoden). Diese Methoden erhalten typischerweise eine `Transaction` als ersten Parameter.
- `tempOf` sollte als Präfix oder Name verwendet werden, wenn ***neue temporäre Instanzen*** erstellt werden (Factory-Methoden). Diese Methoden erhalten typischerweise einen `BOLoaderI` als ersten Parameter.
- `by` sollte als Präfix oder Name verwendet werden, wenn ***existierende Instanzen*** eines Objekts zurückgegeben werden. Wenn die Instanz anhand eines oder weniger Primärschlüssel ermittelt wird, werden die Attributnamen dem `by` angehängt, zum Beispiel `byTid` oder `byMIMEAndEndung`. Diese Methoden erhalten typischerweise einen `BOLoaderI` als ersten Parameter.
- `for` sollte als Präfix gefolgt von der deutschen Bezeichnung des gewünschten Objekts verwendet werden, wenn ***genau eine existierende Instanz*** eines Objekts zurückgegeben wird, zum Beispiel `forJaehrlich` oder `forDeutschland`. Diese Methoden erhalten typischerweise einen `BOLoaderI` als einzigen Parameter.

Spezielle „Prüf"-Methoden

Methoden, die Prüfungen durchführen, sollten gemäß den folgenden Regeln benannt werden:

- `verify` sollte als Präfix verwendet werden, wenn möglicherweise eine `SaveException` ausgelöst wird.
- `check` sollte als Präfix verwendet werden, wenn es keinen Rückgabewert gibt, aber möglicherweise eine `RuntimeException` ausgelöst wird.
- `is`/`has` oder `are`/`have` sollten als Präfix verwendet werden, wenn die Methode einen `boolean` zurückgibt.

Variablen und Methodenparameter

Die Namenskonventionen für Variablen und Methodenparameter sind wie folgt:

- Je kürzer die „Lebenszeit" einer Variable ist, desto kürzer sollte der Name sein. Zum Beispiel kann eine Laufvariable in einer Methode kurz `i` heißen.
- Klassenvariablen und Methodenparameter sollten hingegen einen aussagekräftigen Namen haben und nicht nur aus zwei oder drei Buchstaben bestehen.
- Die Sprache für Variablennamen und Methodenparameter sollte in der Regel Englisch sein.
- In Enums sollten Konstruktoren für die dort definierten Konstanten wie üblich aufgerufen werden, ohne Leerzeichen zwischen Namen und öffnender Klammer.
- Eigene Instanzvariablen oder Fields sollten entweder mit zwei Kleinbuchstaben oder

mit zwei Großbuchstaben beginnen.
Ein Kleinbuchstabe, gefolgt von einem Großbuchstaben am Anfang (zum Beispiel `aProperty`), sollte vermieden werden, da dies zu Verwechslungen führen kann. Es ist empfehlenswert, stattdessen Namen wie `myProperty`, `property1`, `someProperty`, usw. zu verwenden, um Missverständnisse zu vermeiden.
Ein Großbuchstabe, gefolgt von einem Kleinbuchstaben am Anfang, sollte ebenfalls vermieden werden, da dies leicht mit Klassennamen verwechselt werden kann.
Dies folgt der Groovy Property Konvention.

17.2.3. Verwendung von Leerzeichen

Die Verwendung von Leerzeichen sollte in NetRexx, Java und Groovy den folgenden Konventionen entsprechen:

Nie Leerzeichen

- Auf der Innenseite von runden und eckigen Klammern.
- Zwischen einem unären Operator und seinem Argument (in Java und Groovy).
- Bei einem leeren Klammernpaar.
- Vor den Klammern mit den Parametern in Konstruktoren- oder Methodenaufrufen.

Immer Leerzeichen:

- Vor öffnenden geschweiften Klammern (in Java und Groovy).
- Bei Inline-Closures nach der öffnenden und vor der schließenden geschweiften Klammer sowie vor und nach dem Pfeil (in Groovy).
- Bei Schlüsselwörtern mit einer folgenden Klammer, wie `if`, `while`, `for`, `switch`, und `catch` (in Java und Groovy).
- Zwischen einem Operator und seinen Argumenten. Dies gilt auch für den Groovy Ternary Operator (Groovy).
- Bei der Inline-Definition von Maps nach dem Doppelpunkt hinter dem Key (Groovy).
- Nach Kommata, die der Aufzählung von Elementen dienen. Dies gilt auch für die Inline-Definition von Listen. (Groovy)

Beispiel 1

```
Map<Integer, Trick> tricks = [0: new Sparkle(), 1: new Fire(), 2:
new Fireball(), 7: new Cards()]
if (magicRequested()) {
   tricks.each { type, trick ->
      if (trick instanceof Magic) {
```

```
         Magic magic = trick as Magic
         magic.isReady() ? magic.doMagic() : trick.fakeMagic()
      } else {
         println "$value is no kind of magic :-("
      }
   }
}
```

Beispiel 2

```
method doMagic(type = int)
   type = type + 1
   select case type
      when 1 then
         println "Let nose sparkle"
      when 2, 3 then
         println "Start a fire"
      otherwise
         println "Perform card trick"
      end
```

17.2.4. Einrückung / Indentation

Es dürfen **keine** Tabulatoren genutzt werden, sondern nur „Soft-Tabs“, d.h. Leerzeichen.

NetRexx / Java / Groovy

Die Einrückung von Programmcode erfolgt **immer** mit 3 Leerzeichen, egal wo dieser Code auftaucht (Klasse, Skript, Dienst etc.).

In Bezug auf `do-end`-Blöcke gelten die folgenden Regeln:

- Das `do` eines `do-end`-Blocks steht immer in derselben Zeile wie der Code, der den Block einleitet. Erst in der nächsten Zeile beginnt der eingerückte Block.
- Das `end` eines `do-end`-Blocks schließt immer einen Block und eine Einrückungsebene ab. Das bedeutet, dass nach einem `end` der Code eine Stufe weiter links fortgesetzt wird, damit der Block optisch abgesetzt erscheint.

Beispiel

```
if 42 = 42 then do
```

```
   doMagic()
   doFurtherMagic()
   end
doSomethingDifferent()
```

Das `otherwise` eines `select-case-when-otherwise-end`-Blocks wird auf die gleiche Einrückungsebene wie die `when`-Anweisungen gerückt.

Beispiel

```
select case getMonthName()
   when 'January' then
      return 1
   when 'February' then
      return 2
   ...
   otherwise
      return 0
   end
```

Markup (xml, gsp, ...)

In Markup gelten folgende Einrückungsregeln:

- **Markup-Elemente**: Immer 2 Leerzeichen pro Einrückungsebene verwenden. Dies ist der etablierte Standard für Markup-Sprachen und sorgt für eine übersichtliche Darstellung.
- **Eingebetteter Groovy-Code**: Wenn Groovy-Code innerhalb von Markup-Elementen steht (z. B. in Skripten als CDATA), wird dieser Code um **eine** Einrückungsebene weiter eingerückt als der umgebende Node und **ebenfalls** mit 2 Leerzeichen pro Ebene eingerückt. Editoren können in der Regel die Einrückung nicht automatisch kontextbasiert für eingebetteten Programmcode anpassen. Durch die einheitliche Einrückungstiefe mit 2 Leerzeichen ist die Handhabung einfacher und der Code wird dennoch übersichtlich dargestellt. Für neuen Code ist dies verbindlich; bestehender Code kann seine Einrückungstiefe beibehalten, sollte jedoch bei Änderungen oder Ergänzungen an das oben beschriebene Format angepasst werden, um langfristig Konsistenz sicherzustellen.
- **Groovy in BS-Service-Skripten**: Groovy-Code in BS-Service-Skripten wird immer linksbündig ohne zusätzliche Einrückung formatiert.

bash- / Shell-Skripte (sh, ...)

Die Einrückung für bash- oder Shell-Skripte (z. B. sh-Skripte) erfolgt **immer** mit 3 Leerzeichen.

17.2.5. Zeilenlänge

optional

Die Zeilenlänge in unserem Code hat kein festes Hard Limit, aber wir streben an, ein Soft Limit von 120 Zeichen einzuhalten.

Wenn ein Zeilenumbruch in Programmcode eingefügt wird, sollte die folgende Zeile mit einer zweistufigen Einrückung versehen werden, was zwei Soft-Tabs entspricht. Dadurch wird sichergestellt, dass eine umgebrochene Zeile optisch schnell als Fortsetzung erkannt und von normalen Einrückungen unterschieden werden kann, ohne zu viel Leerraum am Anfang der Zeile zu haben.

Im Fall eines einzeiligen if-Blocks gelten ebenfalls die Regeln zur Klammerung von if-Blöcken, wie zuvor festgelegt.

Beispiel

```
signal SaveVetoException(l10n(L10N_KEY_ERR_WRONG_CATEGORY_TYPE,
[Object -
      taxCategory.describe(),
taxCategory.getEntity().getL10nName(), requiredtaxCatEntName]))
```

optional

Die Empfehlung zur Zeilenlänge von möglichst nicht mehr als 120 Zeichen gilt auch für Methodensignaturen. Hierbei ist zu beachten, dass sehr lange Parameterlisten ein Hinweis auf eine mögliche Code-Qualitätsproblematik sein können. In solchen Fällen sollte man über das Codedesign nachdenken und prüfen, ob es sinnvoll ist, die Methode zu refaktorieren, um die Anzahl der Parameter zu reduzieren, um den Code verständlicher und wartbarer zu gestalten.

Beispiel

```
   method filterCachedDefsByClassTimespanAndSteuerkategorie(bol =
BOLoaderI, -
         c = Class, from = Date, upto = Date, category =
AbstrakteGasSteuerkategorie) -
         returns Map<Long, AbstrakteGasSteuerDefinition> private
```

```
static
```

Bedingungen und Queries

Für umfangreiche Bedingungen und lange Query-Strings gilt die folgende Konvention:

- Bei überlangen Zeilen erfolgt in der Regel ein Zeilenumbruch *vor* einem `and` oder `or`, sodass der Verknüpfungsoperator in der Folgezeile steht. Dadurch wird leichter erkennbar, dass ein Ausdruck fortgeführt wird.
- Logisch zusammenhängende Teilausdrücke sollten möglichst nicht auseinandergerissen werden.
- Verschachtelte Ausdrücke können gegebenenfalls noch tiefer eingerückt werden, um eine Bedingung oder eine Query leichter erfassbar zu machen.

optional

Im Falle von Queries gibt es keine *festen* Regeln, und es obliegt dem Programmierer, die Ausdrücke sinnvoll zu gruppieren und einzurücken.

Es dürfen keine vorgefertigten Konstanten wie `ENT_` oder `ATT_` in Queries verwendet werden, obwohl dies die Query teilweise compile-fest machen würde, da dies die Lesbarkeit beeinträchtigt und zukünftige Refactorings für verbesserte Query-Konzepte behindert.

Beispiel

```
   QUERY_GET_PVZS_TO_INVOICE = 'p WHERE NOT p.Ldel' -
         'AND NOT coalesce(p.Endabgerechnet, FALSE)' -
         'AND POD != NULL AND NOT POD.Ldel' -
         'AND Von <= $1' -
         'AND (Bis = NULL OR Bis > $2)' -
         'AND EXISTS (WITHIN AbrechnungsmodusGueltigkeiten a
WHERE NOT a.Ldel' -
            'AND a.Abrechnungsmodus.Tid = $3' -
            'AND a.Von <= $1' -
            'AND (a.Bis = NULL OR a.Bis > $1)' -
         ')' -
         'AND Vertrag != NULL AND NOT Vertrag.Ldel' -
         'AND NOT coalesce(Vertrag.WurdeObsolesziert, FALSE)' -
         'AND NOT coalesce(Vertrag.WurdeNichtBeauftragt, FALSE)'
```

17.2.6. Kommentare

Es gelten die folgenden Regeln für Kommentare:

- Kommentare sind immer in englischer Sprache anzufertigen. Dies fördert die Klarheit und Einheitlichkeit des Codes und erleichtert die Kommunikation und Zusammenarbeit in internationalen Teams.
- Kommentare beziehen sich in der Regel auf den unmittelbar folgenden Code (Comment-Before) und nicht auf Code, der dem Kommentar vorausgeht.

optional

- Mehrzeilige Kommentare sollten mittels Einrückung für folgende Zeilen von einzeiligen Kommentaren unterschieden werden. Dies bedeutet, dass ab der zweiten Zeile mindestens ein zusätzliches Leerzeichen zwischen dem Kommentarsymbol (`//`, `--`) und dem Text eingefügt wird.
  Ab spätestens 3 Zeilen sollten diese Kommentare als Blockkommentare angefertigt werden, indem sie mit einem einleitenden `/*` und einem abschließenden `\*/` versehen werden.
  Ob ein wie die anderen Sternchen ausgerichtetes `*` am Anfang jeder Zeile eingefügt wird (wie in den Oracle Coding Conventions empfohlen), bleibt der individuellen Präferenz überlassen. Dies ist nicht zwingend erforderlich, da moderne Editoren Syntax-Highlighting bieten und der Kommentarcharakter daher auch über mehrere Zeilen hinweg erkennbar ist.
- Um Teile des Codes auszukommentieren, kann jede Zeile einzeln am Anfang der Zeile auskommentiert werden, um einen auskommentierten Codeblock zu bilden. Dies ist hilfreich, wenn der Code bereits mehrzeilige Kommentare enthält, die nicht verschachtelt werden können.
- Wenn eine Methode mit einem Kommentar versehen werden soll, der jedoch nicht in Javadocs aufgenommen werden soll, sollte der Präfix `no-doc` vorangestellt werden. Dies signalisiert anderen Teammitgliedern, dass dieser Kommentar nicht in Javadocs umgeschrieben oder dort eingearbeitet werden soll.

Beispiel

```
//Another comment that is a bit longer and uses normal
indentation, but
// extends to a second line, which is indented slightly.
//Followed by the third separate comment
```

Trailing bzw. End-of-Line Kommentare

Eine Ausnahme von der Regel zu "Comment Before" sind kurze Zeilenkommentare, die in derselben Zeile wie der Code stehen. Diese werden als "Trailing" oder "End-Of-Line" Kommentare bezeichnet und stehen ihrer Natur nach immer *hinter* dem Code, auf den sie sich beziehen.

optional

Mehrere "End-Of-Line" Kommentare in einer Methode untereinander zu "alignen", ist eher ungünstig, da dies zwar in einigen Editoren übersichtlicher sein kann, aber zu längeren Zeilen führt und bei Änderungen am Code ggfs. den Bedarf für Anpassungen an vielen Zeilen nach sich zieht, um die Kommentare weiterhin optisch vom Code getrennt zu halten.

Generell entsprechen "End-Of-Line" Kommentare meist nicht der Erwartung der Java Lesenden. Google untersuchte in einer Studie ca. 100.000 Java-Projekte mit insgesamt ca. 10 Milliarden Codezeilen. Die Studie von Google zeigte, dass in Java-Code zwar durchschnittlich 55 % der Kommentare vor dem Code und 45 % hinter dem Code platziert werden, sie kommt jedoch zu dem Schluss, dass Kommentare in Java-Code in der Regel *vor* dem Code platziert werden, um *den Zweck und die Funktionsweise* des Codes zu erklären, und *hinter* dem Code, um Beispiele für die Verwendung des Codes oder Tests des Codes bereitzustellen.

Wenn viele aufeinanderfolgende Zeilen "End-Of-Line" Kommentare benötigen bzw. diese Kommentare sehr lang sind, so ist dies ein starker Hinweis dafür, dass das Verständnis des Codes eher von einem einleitenden erklärenden Kommentar oder Kommentarblock profitieren könnte.

Eine Ausname stellt das NetRexx Logging und die UI-Progressanzeigensteuerung im Kommentar-Stil dar, das meist besser hinter dem Code aufgehoben ist, es sei denn, der Logtext hat zusätzlich auch den Code dokumentierenden Charakter und kann somit gleichzeitig als Comment-Before Kommentar dienen.
Auch hier sollten in der Regel überlange Zeilen vermieden werden, sie werden aber hier eher toleriert als bei anderem Code.

17.2.7. Vergleiche mit `null` in NetRexx

In NetRexx sollten Vergleiche mit `null` mit einem einfachen Gleichheitszeichen (`=`) als Operator durchgeführt werden, anstelle des doppelten Gleichheitszeichens (`==`) (Java). Der Grund hierfür ist zum einen die Einheitlichkeit zu Vergleichen mit Skalaren und Objekten, zum anderen die etwas kürzere Schreibweise.

Beispiel

```
if a = null then
```

```
   return
```

17.2.8. Strings

Netrexx

In NetRexx und Groovy werden Strings im Allgemeinen mit einfachen Anführungszeichen (`'`) angeführt, wie in `'Ich bin ein String'`. Dies ist die Standardkonvention für die Darstellung von Zeichenketten in diesen Programmiersprachen.

In Sonderfällen, beispielsweise wenn ein Apostroph innerhalb des Strings verwendet werden muss, kann von dieser Konvention abgewichen werden, wie in `"Ich bin ein String mit einem Apostroph: Don’t worry."`.

Groovy

In Groovy müssen immer doppelte Anführungszeichen verwendet werden, sobald Interpolation verwendet wird. Interpolation ermöglicht das Einbetten von Variablen oder Ausdrücken in Zeichenketten. Verwendet man einfache Anführungszeichen, so wird keine Interpolation durchgeführt, sondern der String als reine Zeichenkette behandelt.

17.2.9. Klammerung von if-Blöcken, u.a.

NetRexx

Bei einzeiligen Anweisungen innerhalb des `if`-Blocks kann in der Regel das `do-end` entfallen.

Beispiel, wann man das do-end weglassen kann:

```
if 42 = 42 then
   return -- simply return
```

Das Hinzufügen eines `do-end` in Fällen, in denen innerhalb eines einzeiligen `if`-Statements ein (ein- oder mehrzeiliger) Kommentar steht, verhindert mögliche Missverständnisse oder unerwünschte Änderungen.

Im folgenden Beispiel wird das do-end verwendet, um klar anzuzeigen, dass es sich um einen Block handelt, der potenziell mehrere Anweisungen oder Kommentare enthalten kann.

Beispiel

```
if 42 = 42 then do
   -- if this is true we can simply return
   return
   end
```

Im folgenden Beispiel gehört das `return` *ohne* das `do-end` nicht mehr zum `if`-statement und wird somit unbeabsichtigt immer ausgeführt, da es sich bei `---<lw` tatsächlich um eine Anweisung handelt, die vor das return gezogen wird:

Beispiel

```
if 42 = 42 then
   return ---<lw 'Da stimmt doch was nicht!'
```

Java / Groovy

Wir setzen **immer** geschweifte Klammern für Blöcke, **auch** für einzeilige. Dies wird oft als „always use braces“ oder „always use curly braces“ Regel bezeichnet.

Es wurde entschieden, diese Regel abweichend von unserer Konvention in NetRexx zu verwenden. Gründe dafür sind u.a.: - Ohne geschweifte Klammern kann es unklar sein, wo der bedingte Block beginnt und endet, insbesondere bei verschachtelten Bedingungen. - Das Hinzufügen von geschweiften Klammern macht den Code robuster, indem sichergestellt wird, dass zukünftige Ergänzungen oder Änderungen am bedingten Block innerhalb der Klammern geschlossen sind. - Die Regel vermeidet Änderungen an nicht direkt betroffenem Code bei Erweiterungen des Blocks um weitere Zeilen, was ggfs. Konflikte im Versionsmanagementsystem nach sich ziehen könnte. - Das Auskommentieren des Blocks ist einfacher, da man nur die Zeile auskommentieren muss und nicht die Bedingung und die darin enthaltene Zeile.

Obwohl die o.g. Argumente auch für NetRexx gelten, haben wir uns entschieden, dort u.a. aufgrund der höheren Verbosität der Syntax (in NetRexx repräsentieren die Keywords „do“ bzw. „end“ die geschweiften Klammern) bei einzeiligen Blöcken trotzdem darauf zu verzichten.

Beispiel

```
if (cause != 23) {
   return
}
```

```
if (reason != 42) {
   universe.close()
   return
}
```

17.2.10. „return“-Angabe in Groovy

In Groovy ist die letzte Zeile einer Methode automatisch der Rückgabewert, das Schlüsselwort `return` ist eigentlich überflüssig. Wir lassen es jedoch höchstens bei einzeiligen Methoden in Skripten weg.

Beispiel

```
<onAction language="groovy">bo.Anzahl = 0 && !bo.Erledigt &&
!bo.isNew()</onAction>
```

Selbst wenn obige Zeile noch etwas komplexer wäre und man sie lediglich zur besseren Lesbarkeit umbrechen würde, lassen wir das `return` dort weg.

Sobald die Methode jedoch mehr Zeilen hat und damit komplexer wird, schreiben wir **immer** ein `return` vor den Rückgabewert.

```
if (coll = null) {
  throw new IllegalArgumentException("Given Collection of
BigDecimals must not be null!")
}
return coll.sum()
```

Allerdings ist es eine gute Praxis, `return` weiterhin bei einzeiligen Methoden in Klassen zu verwenden, um die Absicht des Teammitglieders klarer zu machen und zu zeigen, dass die Methode explizit einen Wert zurückgibt.

17.2.11. Logging in NetRexx

optional

Der NetRexx-Präprozessor innerhalb des MyTISM-Frameworks verfügt über spezifische Erweiterungen, die über den Standard-Funktionsumfang hinausgehen. Eine zentrale Rolle spielt dabei der „LogProgressWeaver“. Dieses Werkzeug durchläuft den Quellcode vor der Kompilierung und transformiert speziell formatierte „Log-Kommentare“ in

funktionsfähigen NetRexx-Code.

Diese Methode der instrumentierten Protokollierung („comment-style logging“) erlaubt es, Logging-Anweisungen platzsparend und übersichtlich direkt rechts vom betroffenen Quellcode zu platzieren. Damit diese automatisierte Generierung fehlerfrei greift, muss das für das Logging verwendete Objekt zwingend den Namen `log` tragen.

Die Syntax dieser Kommentare folgt dem festen Schema `---[Reihenfolge][Typ] "[NetRexx-String]"`. Über die spitze Klammer wird dabei die Ausführungsreihenfolge im Verhältnis zur aktuellen Codezeile gesteuert: Während `>` das Logging unmittelbar nach der Zeile ausführt, wird es bei `<` davor eingefügt – eine Option, die sich besonders bei der Protokollierung vor einem `return` bewährt hat.

Die verfügbaren Typenkürzel bilden die klassischen Schweregrade der Protokollierung ab: `ld` steht für Debug, `li` für Info, `lw` für Warning und `le` für Error. Als Sonderfall existiert zudem das Kürzel `p`, mit dem sich die Progressanzeige der grafischen Benutzeroberfläche (GUI) direkt steuern lässt.

Da der LogProgressWeaver die Kommentare lediglich durch reale Codezeilen ersetzt, ergibt sich eine kritische strukturelle Anforderung für Bedingungsprüfungen. Innerhalb von `if … then`-Konstrukten muss das Logging zwingend in einen `do … end`-Block eingebettet werden. Ohne diese Kapselung würde der generierte Logging-Aufruf die logische Integrität der `if`-Anweisung aufbrechen, was dazu führt, dass Log-Ausgaben fälschlicherweise unabhängig von der Bedingung erscheinen oder nachfolgende Codezeilen fälschlich der Bedingung untergeordnet werden.

Dies verdeutlicht das folgende Beispiel der generierten Logik:

```
if 0 then
   say 'Fail 1' ---<le "Das solltest du nicht sehen"
if 0 then
   say 'Fail 2' --->le "Das auch nicht"
if 0 then do
   say 'No Fail' ---<le "Siehst du nicht, auch nicht mit >"
   end
```

Nach der Verarbeitung durch den Weaver resultiert daraus dieser (fehlerhafte) Ablauf, bei dem die ersten beiden Nachrichten trotz der `if 0`-Bedingung ausgegeben würden:

```
if 0 then
   LoggingAusgabe: "Das solltest du nicht sehen"
```

```
say 'Fail 1'
if 0 then
   say 'Fail 2'
LoggingAusgabe: "Das auch nicht"
if 0 then do
   LoggingAusgabe: "Siehst du nicht, auch nicht mit >"
   say 'No Fail'
   end
```

Für den reibungslosen Einsatz sind zudem formale Aspekte der Quelltextgestaltung zu beachten:

1. **Abstand:** Log-Kommentare müssen zwingend mit mindestens einem Leerzeichen vom vorangehenden Code getrennt werden, da sie andernfalls nicht als Steuerbefehl erkannt, sondern als gewöhnliche einzeilige NetRexx-Kommentare behandelt werden.
2. **Zeilenumbruch:** Um die Suchbarkeit innerhalb der Projektdateien zu gewährleisten, sollten Logging-Strings nicht in der Mitte umgebrochen werden. Ein Umbruch unmittelbar nach der Einbindung einer Variablen oder eines Parameters ist jedoch zulässig.
3. **Deaktivierung:** Da der Weaver gezielt nach dem beschriebenen Muster sucht, lässt sich ein Logging-Befehl nicht durch einfaches Auskommentieren via `--` oder `/* … */` unterdrücken. Stattdessen muss das Suchmuster aktiv gebrochen werden. Es empfiehlt sich die Verwendung eines expliziten Hinweises wie `-- disabled log -<ld`, um die Absicht klar zu kennzeichnen und spätere Verwechslungen mit Tippfehlern auszuschließen.

17.2.12. Klassen, Methoden und Variablen

Klassen

Reihenfolge der Klassen-Modifier

Das Einhalten einer festen Reihenfolge für Modifier erleichtert die Lesbarkeit des Codes und trägt zur Konsistenz bei.

Die von Oracle vorgegebene Reihenfolge der Klassen-Modifier ist: `@Annotation public protected private abstract static final strictfp`

Für NetRexx ist, basierend darauf, die Reihenfolge so: `@Annotation extends implements public inheritable shared private abstract interface dependent static final uses`

Zwischen alle Modifier darf nur *ein einzelnes* Space gesetzt werden.

Statische Importe in Java bzw. "uses" in NetRexx

Best Practices für statische Importe in Java

Statische Importe in Java erlauben den Zugriff auf statische Member einer Klasse, ohne den Klassennamen zu qualifizieren. Dies kann den Code kompakter und lesbarer machen, birgt aber auch Risiken.

- **Sparsam einsetzen:** Statische Importe sollten nur verwendet werden, wenn sie die Lesbarkeit tatsächlich verbessern.
- **Einzelne Member importieren:** Anstatt alle statischen Member einer Klasse zu importieren (`import static java.util.Collections.*;`), sollten nur die benötigten Member einzeln importiert werden (`import static java.util.Collections.sort;`).
- **Klare Namensgebung:** Aussagekräftige Namen für Konstanten und Enums verwenden, um Konflikte zu vermeiden und die Lesbarkeit zu erhöhen.
- **Konsistenz:** Konsistent sein in der Verwendung von statischen Importen innerhalb eines Projekts.

Wann statische Importe sinnvoll sein können

- **Konstanten:** Der Import von Konstanten wie `Math.PI` oder selbst definierten Konstanten kann die Lesbarkeit verbessern, insbesondere wenn diese häufig verwendet werden.

```
import static java.lang.Math.PI;
double circumference = 2 * PI * radius;
```

- **Enums:** Ähnlich wie bei Konstanten kann der Import von Enum-Werten den Code vereinfachen.

```
import static MeinEnum.WERT_A;
if (variable == WERT_A) { ... }
```

- **Utility-Methoden:** Statische Methoden aus Utility-Klassen, die häufig verwendet werden, können importiert werden, um den Code zu verkürzen. Beispiel: `Collections.sort()`.

```
import static java.util.Collections.sort;
sort(meineListe);
```

Wann statische Importe vermieden werden sollten

- **Übermäßiger Gebrauch:** Zu viele statische Importe können den Code unübersichtlich machen und die Herkunft der Member verschleiern.
- **Namenskonflikte:** Wenn mehrere Klassen gleichnamige statische Member haben, können Konflikte auftreten. In solchen Fällen ist es besser, den Klassennamen zu qualifizieren.
- **Methoden mit Seiteneffekten:** Statische Methoden, die Seiteneffekte haben, sollten nicht importiert werden, da dies die Lesbarkeit und Wartbarkeit beeinträchtigen kann.

Best Practices für die Verwendung von `uses` in NetRexx

Die `uses`-Anweisung in NetRexx ermöglicht den Zugriff auf statische Member (Eigenschaften und Methoden) einer Klasse, ohne den Klassennamen explizit angeben zu müssen. Dies ähnelt den statischen Importen in Java, jedoch mit dem Unterschied, dass in NetRexx immer die gesamte Klasse importiert wird und nicht einzelne Member.

- **Sparsam einsetzen:** `uses` sollte nur verwendet werden, wenn es die Lesbarkeit tatsächlich verbessert.
- **Reihenfolge beachten:** Die Reihenfolge der Klassen in der `uses`-Liste kann wichtig sein, da bei Namenskonflikten die erste passende Klasse verwendet wird.
- **Klare Namensgebung:** Aussagekräftige Namen für Konstanten und Methoden in den Klassen verwenden, die mit `uses` eingebunden werden.
- **Konsistenz:** Konsistent sein in der Verwendung von `uses` innerhalb eines Projekts.

Wann `uses` sinnvoll sein kann:

- **Häufig verwendete Konstanten:** Wenn eine Klasse viele Konstanten enthält, die im Code oft verwendet werden, kann `uses` die Lesbarkeit verbessern.
- **Utility-Klassen:** Klassen mit vielen statischen Hilfsmethoden, die im Code häufig benötigt werden, können mit `uses` effizienter genutzt werden.

Wann `uses` vermieden werden sollte:

- **Zu viele Klassen:** Die Verwendung von `uses` für zu viele Klassen kann den Code unübersichtlich machen und die Herkunft der Member verschleiern.
- **Namenskonflikte:** Wenn mehrere Klassen mit `uses` eingebunden werden und diese

gleichnamige statische Member haben, können Konflikte auftreten. In solchen Fällen ist es besser, den Klassennamen zu qualifizieren.

- **Große Klassen:** Bei sehr großen Klassen mit vielen statischen Membern kann `uses` die Lesbarkeit beeinträchtigen, da unklar ist, welche Member tatsächlich verwendet werden.

Ein häufiger Fehler ist die Angabe von "uses BinaryBoolTools" in Verbindung mit dem Versuch, die Methode "xor" aufzurufen. Dies funktioniert nicht, da "xor" in NetRexx jetzt ein Operatorname ist, sodass die Kurzschreibweise nicht mehr funktioniert. Die entsprechende Fehlermeldung des Compilers lautet: "Error: Unexpected comma ',' in expression".

Methoden

Formatierung und Zeilenabstände

Zwischen zwei Methoden sollten immer zwei Leerzeilen eingefügt werden, um die Lesbarkeit und Übersichtlichkeit des Codes zu verbessern. Diese visuelle Trennung erleichtert es Teammitgliedern, den Code schnell zu scannen und zwischen den verschiedenen Methoden zu navigieren.

Einige Editoren und Entwicklungsumgebungen bieten zudem die Möglichkeit, virtuelle Trennlinien zwischen den Methoden einzublenden.

Beispiel

```
method doMagic()
   if not isWizard() then
      return
   performMiracle()


method performMiracle()
   wiggleMagicWand()
   startFireworks()
```

Eine Ausnahme von dieser Regel sind Methoden, die aus nur einer Zeile bestehen, wie sie beispielsweise in abstrakten Klassen oder Interfaces vorkommen können. In solchen Fällen ist auch nur eine Leerzeile oder sogar garkeine erlaubt. Allerdings ist es in Interfaces und abstrakten Klassen ratsam, JavaDoc zu verwenden, so dass der Abschnitt für die Methode normalerweise sowieso mehr als eine Zeile umfasst.

Beispiel

```
class Magic interface

   method isWizard() returns boolean

   method doMagic()

   method performMiracle()

   method wiggleMagicWand()
   method startFireworks()
```

Reihenfolge der Methoden

optional

Um die Lesbarkeit und Wartbarkeit des Codes zu verbessern, kann das Etablieren einer klaren Ordnung und Struktur für BO-Klassen hilfreich sein.

Die öffentlichen statischen Methoden sollten an erster Stelle stehen, gefolgt von den privaten statischen Helfermethoden, die idealerweise in der Nähe ihrer Verwendung definiert werden sollten.

Die Häufung von öffentlichen statischen Methoden kann oft darauf hinweisen, dass keine objektorientierte Herangehensweise verwendet wurde.

Die Konstruktoren sollten unmittelbar nach den öffentlichen statischen Methoden platziert werden.

Daraufhin folgen die standardmäßigen, MyTISM-spezifischen Methoden wie `initDefaults`, `delete`, `verifyOnClient` und `verifyOnServer`, jeweils mit den entsprechenden `before`- und `after`-Varianten. Anschließend kommen die client- und serverseitigen `before`- und `afterSave`-Methoden, gefolgt von `isReadOnly` und `isMandatory`.

Es ist empfehlenswert, die Implementierung von virtuellen Attributen sowie gegebenenfalls überschriebene Getter-, Setter-, Adder- und Remover-Methoden danach zu platzieren.

Reihenfolge der Methoden-Modifier

Eine festgelegte Reihenfolge der Modifier in der Methodendefinition kann die Lesbarkeit

des Codes verbessern.

Die Modifier sollten nach einem eventuell vorhandenen `returns X`, aber vor den `signals` (die am Ende der Definition stehen sollten), angegeben werden.

Die von Oracle vorgegebene Reihenfolge der Methoden-Modifier ist: `@Annotation public protected private abstract static final synchronized native strictfp`.

Für NetRexx ist, basierend darauf, die Reihenfolge so:

- `@Annotation` wird vor der Methodendefinition platziert.
- Hinter `@Annotation` können, falls vorhanden, die folgenden Modifier in dieser Reihenfolge angegeben werden: `public inheritable shared private abstract static final constant protect`.

Zwischen alle Modifier darf nur *ein einzelnes* Space gesetzt werden.

Beispiel

```
@Override
method doMagic() returns String inheritable
   return "so macht man's"


method doMagic2() returns String public static
   return "so geht's natürlich auch"
```

Methodenparameter

Anzahl der Parameter

optional

Es ist ratsam, nicht mehr als drei Methodenparameter zu verwenden. Eine Methode mit zu vielen Parametern kann unhandlich sein, da die Typen und Verwendungszwecke der Parameter wiederholt überprüft werden müssen, um sicherzustellen, dass die richtigen Werte an den richtigen Stellen übergeben werden. Dies kann zu erhöhtem Aufwand bei der Fehlersuche und Wartung des Codes führen.

Wenn eine Methode viele Parameter erfordert, sollten alternative Ansätze zur Datenübergabe in Betracht gezogen werden. Dies kann durch das Kapseln der Parameter in ein Datenobjekt oder durch die Verwendung von Variablen mit größerem Gültigkeitsbereich geschehen.

Typisierung von Parametern und Rückgabewerten

Es wird empfohlen, für Collections und Maps wenn möglich den sogenannten „Diamond Operator“ zur Typisierung zu verwenden.

Obwohl der NetRexx-Parser den Diamond Operator erkennt und ihn für den Java-Compiler unverändert lässt, gibt es bisher keine vollständige Unterstützung für den Diamond Operator (oder Generics) in NetRexx. Dennoch ist es empfehlenswert ihn zu verwenden, ähnlich wie bei der Verwendung der @Override-Annotation. Dies hilft, den Code auf eine zukunftssichere Weise zu schreiben und erleichtert anderen Teammitgliedern das Verständnis des Codes.

Typisierung mit allein stehender Wildcard, wie z.Bsp. im Fall von `Map<String, ?>` wird von NetRexx (noch) nicht unterstützt.

Beispiel

```
   method toMap(c = Collection<BOI>) returns Map<Long, ? extends
BOI> public static

   method nullSafeContainsBO(m = Map<Long, BOI>, bo = BOI)
returns boolean public static

   method getUpdateableEntities() returns Set<EntityI>
```

Leerzeilen innerhalb von Methoden

Leerzeilen, ebenso wie Kommentare, können innerhalb von Methoden dazu beitragen, den Code visuell zu gliedern, logische Abschnitte voneinander zu trennen, und somit die Lesbarkeit und Struktur des Codes weiter zu verbessern. Logische Abschnitte können z.Bsp. sein: *Initialisierung mehrerer Variablen zu Beginn einer Methode oder vor Ausführung einer Schleife, Bestimmen von Defaults, Checks von einem oder mehrerer Methoden-Parameter, Fehlerbehandlung (catch-Block), kurze Schleifen, uvm.*

Beispiel (ohne Leerzeilen / Kommentare)

```
if endDate = null then
   endDate = getDefaultOf(Date()), 3)
startDate = calculateStart(startDate)
endDate = calculateEnd(endDate)
return buildCSVContent(startDate, endDate,
this::getFormattedPrice)
```

Beispiel (mit Leerzeilen / Kommentare)

```
-- customer explicitly requested the default to be "now + x days"
if endDate = null then
   endDate = getDefaultOf(Date()), 3)

startDate = calculateStart(startDate)
endDate = calculateEnd(endDate)

-- handles the creation of the file content which can then be
saved or displayed by the caller
return buildCSVContent(startDate, endDate,
this::getFormattedPrice)
```

Manchmal bietet es sich an, einen Codeblock, der bereits als zusammenhängende logische Einheit identifiziert wurde, mit einem erklärenden Kommentar zu versehen. Kommentarzeilen sollten zusätzliche Informationen zur Funktionsweise der Methode bieten und dem Leser helfen, den Code besser zu verstehen. Siehe zur Platzierung und Formatierung von Kommentaren auch den Abschnitt Kommentare.

Auch hier gilt - wie bei vielen Dingen - nach Möglichkeit ein gesundes Maß an Leerzeilen und Kommentaren zu wählen. Zu viele Leerzeilen oder überflüssige Kommentare wie z.Bsp. solche, die 1:1 eine Zeile Code wiederspiegeln, können dazu führen, dass die eigentliche Logik aus dem Fokus des Lesers rückt und das Verständnis sogar erschweren.

Letztendlich liegt die sinnvolle Anwendung von Leerzeilen und Kommentaren im eigenen Ermessen.

Variablen

Reihenfolge der Modifier in Variablen-Deklarationen

Die von Oracle vorgegebene Reihenfolge der Variablen-Modifier ist: `@Annotation public protected private static final transient volatile`.

Für NetRexx ist, basierend darauf, die Reihenfolge so: `@Annotation public inheritable shared private static final constant transient volatile`.

Zwischen alle Modifier darf nur *ein einzelnes* Space gesetzt werden.

Typendeklaration in Groovy

Es empfiehlt sich, Variablen in Groovy immer mit dem konkreten Typ zu deklarieren, um die

Lesbarkeit des Codes zu erhöhen und mögliche Probleme während der Kompilierung zu vermeiden. Sofern die Verwendung eines abstrakteren Typs oder einer Schnittstelle ausreicht, kann auch dieser verwendet werden, um die Flexibilität des Codes durch eine lockerere Kopplung zu erhöhen.

Beispiel

```
String datum = bo.getUserDate() // Methodennamen besagt nicht
klar ob String oder Date
```

In manchen Fällen, insbesondere für Laufvariablen, ist jedoch ein einfaches `def i = 0` ausreichend.

Propertyschreibweise für Groovy

Früher haben wir den ersten Buchstaben von Properties in unserem Groovy Code häufig groß geschrieben, selbst wenn es sich dabei nicht um ein Akronym handelte.

Wir halten uns jetzt an die offizielle Namenskonvention von Groovy und schreiben den ersten Buchstaben von Properties klein, es sei denn die Property beginnt mit 2 Großbuchstaben.

Ein Klein- direkt gefolgt von einem Großbuchstaben sollte bereits bei der Namenswahl vermieden werden, siehe Link für Details.

Wenn bestehender Code diese Konventionen nicht einhält, sollte der Code bei Gelegenheit aktualisiert werden, zumindest methodenweise. IDEs unterstützen diese „unübliche“ Schreibweise nicht und können daher keine semantisch orientierten Unterstützungen für diesen Code anbieten.

Bei der Aktualisierung des Codes muss darauf geachtet werden, dass es sich bei der geänderten Variable wirklich um eine Property handelt und nicht um eine Map mit „Ort“ oder „uRL“ als extern definierten Text-Key.

Property beginnt mit 2 Kleinbuchstaben

Beginnt eine Property mit 2 Kleinbuchstaben, z.B. `String ort`, so sind folgende Schreibweisen erlaubt bzw. nicht erlaubt:

```
   // erlaubt
   bo.ort // ruft bo.getOrt()
   bo.getOrt()
   bo.setOrt(city)
```

```
// nicht erlaubt
bo.Ort
```

Dynamisches Groovy

Die nicht erlaubte Schreibweise

- läuft intern auf eine MissingMethodException und landet erst via MOP (Meta-Object-Protocol) bei `bo.getOrt()`. Dies wird normalerweise jedoch beim ersten Auftreten gecached und ist daher kein Performance-Killer. In Grails gibt es, zumindest vor Grails 3, jedoch kein Caching und diese Schreibweise hat daher sehr wohl einen Performance-Impact.
- führt zudem `isReadonly`-Checks aus und prüft bei Skalaren eventuelle Typ-Constraints aus der types.xml ab, wie Länge, Regex, Min/Max, etc. Das kann zu unerwarteten Laufzeitfehlern führen und reduziert die Nachvollziehbarkeit des Codes. Code sollte bei Bedarf diese Eigenschaften daher besser explizit prüfen und behandeln. Siehe

  BO#propertyMissing → CBOAttribute#setValue(Object, Object).

Statisch kompiliertes Groovy

- hat diesen Seiteneffekt nicht und ruft den Setter/Getter direkt auf.

Property beginnt mit 2 Großbuchstaben

Beginnt ein Attributsname mit 2 Großbuchstaben, dann **muss** der erste Buchstabe in der Propertyschreibweise ebenfalls groß geschrieben werden, ansonsten werden Änderungen an dem Objekt eventuell **nicht** von der Transaction aufgezeichnet!

Beginnt eine Property mit 2 Großbuchstaben, meist ein Akronym, z.B. `String URL`, so sind folgende Schreibweisen erlaubt bzw. nicht erlaubt:

```
// erlaubt
bo.URL // ruft bo.getURL()
bo.getURL()
bo.setURL(newURL)

// nicht erlaubt
bo.uRL
```

Die nicht erlaubte Schreibweise ignoriert bei einer Zuweisung den Setter und **verhindert, dass die Änderung aufgezeichnet wird**. Der Getter wird ebenfalls übersprungen, das fällt bei persistenten Attributen jedoch meist nicht auf. Hinter dieser Schreibweise kann sich häufig ein Bug verstecken.

Beispiel

```
①
class Test {

   private int hALLO = 0 ②

   public setHALLO(int i) {
      println("Setter: $hALLO -> $i:")
      hALLO = i
   }

   public int getHALLO() {
      print("Getter: ")
      return hALLO
   }

   def propertyMissing(String name, value) {
      println("Set propertyMissing: $name -> $value")
   }
   def propertyMissing(String name) {
      print("Get propertyMissing '$name':")
      return null
   }

}

def t = new Test()
println t.HALLO               // Getter: 0
println t.hALLO               // 0
println "Setting via hALLO"   // Setting via hALLO
t.hALLO = 2                   //
println t.HALLO               // Getter: 2
println "Setting via HALLO"   // Setting via HALLO
```

```
t.HALLO = 3                    // Setter: 2 -> 3
println t.HALLO                // Getter: 3
println t.hallo                // Get propertyMissing 'hallo':
null
```

① CompileStatic oder nicht ist an dieser Stelle egal.

② Groovy ignoriert die 'private' Sichtbarkeit!

Falls sowohl eine „is“ als auch eine „get“ Methode vorliegen, bevorzugt Groovy immer die „is“ Methode in der Property-Schreibweise. Das kann im Formularcode zu Verwirrung führen, da dort sowohl eine „getRoot()“ als auch eine „isRoot()“ Methode existieren. Daher ist dort immer die ausführliche Schreibweise zu empfehlen.

Javadoc

Die Verwendung einer Vorlage hilft dabei, konsistente und gut dokumentierte Javadoc-Kommentare im Sourcecode zu erstellen.

Beispiel für eine saubere und einheitliche Formatierung von Javadoc-Kommentaren

```
/*
 * [Short description of the class or method]
 *
 * [More detailed description of the class or method in HTML
format. This
 * description may have arbitrary length and may contain
structuring elements
 * like lists, tables and images.]
 *
 * [Within the description you might link with {@link CBO} to
another class.
 * The current class can be marked with {@code CurrentClass}]
 *
 * [If you want to include an image, place an image file in the
 * doc-files directory and reference it: <img src="doc-
files/myImage.bmp">]
 *
 * [You can also include a <a href="doc-
files/myExample.java">reference</a>
```

```
 * to a source code file.]
 *
 * [Optional: Description of how this class or method is used and
its intended
 * purpose, if it's not obvious from the name or signature.]
 *
 * @param paramName    [Optional: Description of the parameter,
including its
 *                      type and any restrictions on the value.
Repeat for each
 *                      parameter.]
 * @return             [Optional: Description of the return
value, including
 *                      its type and any restrictions on its
value.]
 * @throws Exception   [Optional: Description of any exceptions
that can be
 *                      thrown by this method, including the
conditions under
 *                      which they are thrown. Repeat for each
exception.]
 * @deprecated        [Optional: Description of when and why this
class or
 *                      method was deprecated, and what should be
used instead.]
 * @see               [Optional: Reference to another class or
method that is
 *                      related to this one.]
 * @since             [Optional: Version in which this class or
method was
 *                      introduced.]
 */
```

Die Dokumentation in Java sollte bestimmte Schlüsselkomponenten enthalten:

- **Kurze Beschreibung:** Eine prägnante Erklärung der Klasse oder Methode in einem einzigen Satz.
- **Ausführliche Beschreibung:** Eine detaillierte Erläuterung des Elements, die in Form

von HTML-Text verfasst ist und auch strukturierte Elemente wie Listen, Tabellen und Bilder enthalten kann.

- **@param-Tag:** Dieses Tag wird verwendet, um die Parameter einer Methode oder eines Konstruktors zu dokumentieren. Für jeden Parameter sollte es eine Beschreibung geben, die Informationen über den Typ und eventuelle Wertebeschränkungen enthält.
- **@return-Tag:** Mit diesem Tag wird der Rückgabewert einer Methode dokumentiert. Die Beschreibung sollte Angaben zum Typ des Rückgabewerts und eventuelle Wertebeschränkungen enthalten.
- **@throws-Tag:** Hiermit werden die Ausnahmen dokumentiert, die von einer Methode ausgelöst werden können. Jede Ausnahme sollte detailliert beschrieben werden, einschließlich der Bedingungen, unter denen sie auftritt.
- **@deprecated-Tag:** Wenn eine Klasse oder Methode nicht mehr verwendet werden sollte, wird dieses Tag verwendet. Es sollte erläutert werden, wann und warum die Klasse oder Methode als veraltet betrachtet wird und welche Alternative stattdessen verwendet werden sollte.
- **@see-Tag:** Dieses Tag dient dazu, auf andere Klassen oder Methoden zu verweisen, die in Zusammenhang mit der aktuellen stehen.
- **@since-Tag:** Hier wird die Version der Software angegeben, in der die Klasse oder Methode erstmals eingeführt wurde.

Die Verwendung dieser Dokumentationskomponenten erleichtert anderen Teammitgliedern das Verständnis der Klassen und Methoden und fördert die Wartbarkeit des Codes.

17.2.13. SQL und OQL

Für reservierte Schlüsselwörter wie `SELECT`, `WHERE`, `WITHIN`, `EXISTS` etc. sollten grundsätzlich Großbuchstaben verwendet werden. Dies gilt auch für Funktionsaufrufe wie `LOWER` oder `DATE_PART`.

optional

Lange Abfragen können nach eigenem Ermessen umgebrochen werden. Hier hat sich allerdings bewährt, die `WHERE` clause sowie logische Operatoren in einer neuen Zeile zu beginnen und entsprechend einzurücken. Einen „Goldstandard“ dafür gibt es aber leider nicht.

Beispiel

```
WebAccount w WHERE NOT Ldel
   AND AnmeldungVerweigern
   AND NOT EXISTS(SELECT Id FROM ERechnungKonfiguration k
```

```
      WHERE NOT Ldel
      AND NOT PODVertragZuordnung?.Ldel
      AND NOT PODVertragZuordnung?.Vertrag?.Ldel
      AND PODVertragZuordnung?.Vertrag?.WurdeUnterzeichnet
      AND PODVertragZuordnung?.Vertrag?.Bot?.Name IN $1
      AND LOWER(Adresse) = LOWER(w.Name)
      AND (
         PODVertragZuordnung?.Bis = NULL
         OR (
            PODVertragZuordnung?.Bis != NULL
            AND PODVertragZuordnung?.Bis > $2
         )
      )
   )
```

optional

Teilweise wird die recht übersichtliche „River“-Einrückung verwendet. Hierbei sollten Leerzeichen verwendet werden, um den Code so auszurichten, dass die Haupt-Schlüsselwörter alle an der gleichen Zeichengrenze enden. Dies bildet einen „Fluss“ in der Mitte, der es dem Auge des Lesers erleichtert, den Code zu überfliegen und die Schlüsselwörter von den Implementierungsdetails zu trennen. Allerdings wird das von den meisten Editoren nicht gut unterstützt und muss daher von Hand gepflegt werden.

Beispiel für die River-Einrückung

```
(SELECT f.species_name,
        AVG(f.height) AS average_height, AVG(f.diameter) AS
average_diameter
   FROM flora AS f
  WHERE f.species_name = 'Banksia'
     OR f.species_name = 'Sheoak'
     OR f.species_name = 'Wattle'
  GROUP BY f.species_name, f.observation_date)

  UNION ALL

(SELECT b.species_name,
        AVG(b.height) AS average_height, AVG(b.diameter) AS
average_diameter
```

```
   FROM botanic_garden_flora AS b
  WHERE b.species_name = 'Banksia'
     OR b.species_name = 'Sheoak'
     OR b.species_name = 'Wattle'
  GROUP BY b.species_name, b.observation_date);
```

Epilog: Vom Wissen zur Anwendung

Sie haben nun das Fundament von MyTISM kennengelernt. Der Weg führte Sie von der ersten Orientierung in der Benutzeroberfläche bis hin zu den tieferen Mechanismen, die im Hintergrund für reibungslose Abläufe sorgen.

Eine Plattform mit so vielen Möglichkeiten fordert anfangs unweigerlich etwas Einarbeitungszeit und Struktur. Doch genau diese klare Struktur ist der Schlüssel für ein entspanntes, sicheres und fehlerfreies Arbeiten.

Wenn die grundlegende Bedienung und die Logik hinter dem System einmal verinnerlicht sind, gehen Ihnen viele Handgriffe bald intuitiv von der Hand. Was bleibt, ist mehr Zeit und Raum für Ihre eigentlichen, wichtigen Kernaufgaben im Tagesgeschäft.

Denken Sie bei Ihrer zukünftigen Arbeit stets an die elementaren Grundprinzipien zurück, die wir am Beispiel unseres Krankenhauses erarbeitet haben:

- **Sorgfalt und Datenqualität:** Pflegen Sie Ihre Informationen strukturiert und nutzen Sie die Hilfen des Systems, um stets saubere Datenbestände zu gewährleisten.
- **Klare Berechtigungen und Sicherheit:** Gewähren Sie als Administrator Zugriffsrechte immer nur exakt dort, wo sie für die tägliche Arbeit zwingend benötigt werden.
- **Übersicht und Teamwork:** Hinterlassen Sie Ihre Arbeitsbereiche, Ordner und Masken stets aufgeräumt, um sich und Ihren Kollegen den Arbeitsalltag zu erleichtern.

Dieses Handbuch ist als lebendiges Nachschlagewerk konzipiert, das mit Ihren Aufgaben wächst. Kehren Sie jederzeit zu den spezifischen Kapiteln zurück, wenn Sie in der Praxis auf neue oder ungewohnte Herausforderungen stoßen.

MyTISM ist Ihr tägliches Werkzeug und Ihr verlässlicher Helfer. Wir wünschen Ihnen viel Erfolg und stets ein reibungsloses Arbeiten bei all Ihren zukünftigen Aufgaben!