Bundesverwaltungsamt

IsyFact-Standard

Konzept Fehlerbehandlung

Version 2.14
31.01.2017

IsyFact Standards

Ansprechpartner:

Referat Z II 2
Bundesverwaltungsamt
E-Mail: isyfact@bva.bund.de
Internet: www.isyfact.de

Dokumentinformationen

Dokumenten-ID:	Konzept_Fehlerbehandlung.docx

Java Bibliothek / IT-System

Name	Art	Version
isy-exception-core (Teil isyfact-base)	Bibliothek	siehe isyfact-bom v1.3.6
isy-exception-sst (Teil isyfact-base)	Bibliothek	siehe isyfact-bom v1.3.6

Inhaltsverzeichnis

1. Einleitung

Dieses Dokument enthält das technische Feinkonzept zur Fehlerbehandlung für alle Anwendungen gemäß Referenzarchitektur der IsyFact.

2. Aufbau und Zweck des Dokuments

Dieses Dokument umfasst das Feinkonzept zur Fehlerbehandlung für alle Anwendungen nach Referenzarchitektur der IsyFact. Es beschreibt die Anforderungen an die Nutzung und Verarbeitung von Exceptions und gibt konkrete Beispiele hierfür.

Ziel dieses Dokumentes ist es, einen Standard für die einheitliche Implementierung von Exceptions in Anwendungen festzulegen und eine Anleitung für die korrekte Verwendung von Exceptions zu geben. Dadurch wird eine einheitliche und korrekte Reaktion auf Fehler- und Ausnahmesituationen innerhalb der Anwendungen ermöglicht. Es werden hierbei die Anforderungen von Betrieb, Nutzern und Softwarelieferanten berücksichtigt.

Dieses Feinkonzept beschäftigt sich ausschließlich mit den übergreifenden Anforderungen an die Behandlung von Exceptions und macht Vorgaben zu deren Umsetzung und Anwendung. Die von den Anwendungen benötigten Exception-Klassen sind von der konkreten Anwendung abhängig und müssen im entsprechenden Projekt, unter Berücksichtigung der in diesem Feinkonzept gemachten Vorgaben, erstellt werden. Zur Umsetzung werden ausführliche Beispiele gegeben.

In *Kapitel 3* werden zunächst Anforderungen an die Fehlerbehandlung aufgestellt. Im folgenden *Kapitel 4* werden Vorgaben definiert, wie Exceptions zu erstellen sind und wie sie behandelt werden müssen. Konkrete Beispiele werden hierbei anhand der Vorlageanwendung dargestellt [VorlageAnwendung]. Die Vorlageanwendung ist eine Beispiel-Implementierung einer Anwendung gemäß der IsyFact-Standards und beinhaltet eine entsprechende Implementierung für die Vorgaben aus diesem Konzept. Abschließend werden in *Kapitel 5* typische DO's und DON'Ts erläutert, die bei der Fehlerbehandlung beachtet werden müssen.

3. Anforderungen an die Fehlerbehandlung

Die generelle Anforderung an die Fehlerbehandlung ist das Erkennen, Signalisieren und Verarbeiten von Fehlern, die im System auftreten.

Konkreter bedeutet dies, dass Fehler möglichst effizient behandelbar sein sollen, die Wartbarkeit von Fehlern sichergestellt ist und der Einsatz von Exceptions in einer standardisierten Form geschieht, um die Implementierung zu vereinheitlichen und für den Anwender angemessene Informationen zur Verfügung zu stellen.

Aus den genannten Anforderungen ergeben sich entsprechende Detailvorgaben, die zur Umsetzung der generellen Anforderungen an die Fehlerbehandlung notwendig sind:

Nutzen stiftende Fehlermeldungen: Eine Fehlermeldung muss klar verständlich sein, vor allem, wenn sie dem Benutzer angezeigt wird. Außerdem müssen die Fehlertexte so aufgebaut sein, dass sie den Betrieb und die Entwickler bei der Ursachenforschung und der schnellen Bearbeitung unterstützen.

Zuordnung von Fehlern: Die IsyFact-Architektur beschreibt eine serviceorientierte Anwendungslandschaft. Das Ergebnis, das einem Nutzer angezeigt wird, entsteht durch das Zusammenspiel verschiedener Anwendungen, die jeweils Services anbieten. Ein gemeldeter Fehler muss in dieser Landschaft einer Anwendung zuzuordnen sein.

Eigene Fehlertexte für Drittanbieter-Komponenten: Für unchecked Exceptions von Drittanbieter-Komponenten, wie z. B. Hibernate, müssen eigene Fehlertexte für die Komponente (nicht für die einzelnen möglichen Exceptions) hinterlegt sein.

Exceptions sind Teil der Komponenten- / Anwendungsschnittstelle: Es muss dokumentiert sein, welche Exceptions eine Komponente bzw. Anwendung an den Aufrufer weiterreicht.

Einfache Nutzung der Fehlerbehandlung: Die Nutzung der Fehlerbehandlung muss einfach sein. Das Vorgehen, wie und wann Fehler gemeldet werden, soll möglichst einfach sein.

Übersichtlichkeit: Die Fehlerbehandlung soll übersichtlich bleiben, d. h. auch das Melden von Fehlern soll sinnvoll und überschaubar sein. Der Code zur Fehlerbehandlung soll nicht über die ganze Anwendung verstreut werden.

Wartbarkeit von Fehlern: Die Anzahl der möglichen Fehler in einem System steigt mit der Größe des Systems. Es muss also sichergestellt werden, dass ein Überblick über die möglichen Exceptions vorhanden ist und hier keine Redundanzen entstehen.

Einfache Pflege von Fehlertexten: Fehlertexte eines Systems müssen einfach pflegbar und standardisiert sein.

Unterscheidung nach Art: Die verwendeten Exceptions müssen in fachliche / technische und checked / unchecked Exceptions unterschieden werden können.

Konsistenter Systemzustand: Die Konsistenz des Systems, insbesondere der Daten, muss in (unerwarteten) Fehlersituationen gewährleistet sein. Außerdem darf dies nicht zu einem unkoordinierten Absturz des Systems führen.

4. Implementierung der Fehlerbehandlung

Ziel der in den folgenden Kapiteln beschriebenen Fehlerbehandlung ist:

- das Erreichen eines effizienten und einheitlichen Umgangs mit Fehlern
- die Gewährleistung der Wartbarkeit von Anwendungen durch aussagekräftige Fehler
- die einfachere Wartung durch einen standardisierten Einsatz von Fehlerbehandlung und Fehlern

4.1. Prinzipielles Vorgehen

Exceptions werden grundsätzlich nur zur Signalisierung abnormer Ergebnisse bzw. Situationen eingesetzt.

Bei Eintritt einer solchen Situation muss der Aufrufer die Exception behandeln, sofern er diese behandeln kann. Dies gilt allgemein für die gesamte Fehlerbehandlung.

Das Konzept der in den folgenden Kapiteln beschriebenen Fehlerbehandlung zeigt die Abbildung 1.

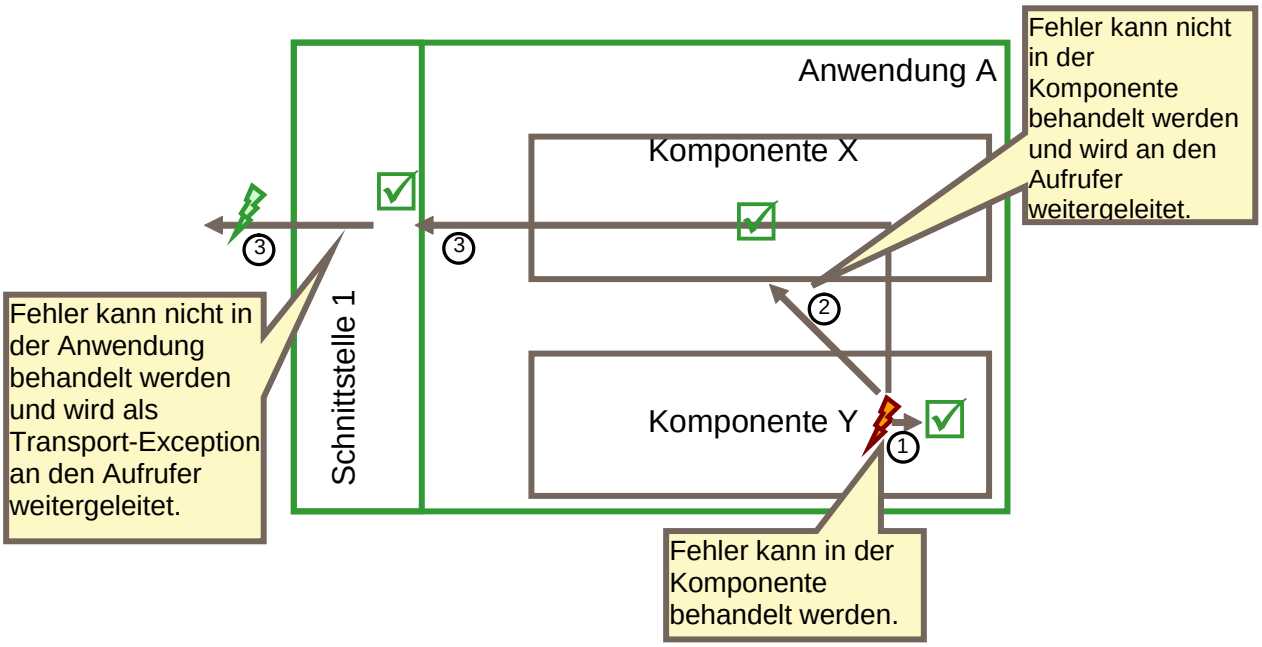

Abbildung 1: Prinzip der Fehlerbehandlung

In einer Komponente gelten die Schnittstellenbeschreibungen, die durch die Signaturen der aufgerufenen Methoden festgelegt sind. Hier auftretende Fehler sind zu behandeln, sofern möglich (siehe: ①). Ist dies nicht möglich, so reicht der Aufgerufene die Exception in der Aufruf-Hierarchie nach oben weiter, d.h. zur aufrufenden Komponente (siehe: ②) oder direkt zur Schnittstelle, sofern die Komponente von dort aufgerufen wurde.

Komponenten bilden Anwendungen. Auch auf Anwendungsebene gilt analog, dass die hier auftretenden Exceptions innerhalb der Anwendung behandelt werden müssen (siehe ②). Ist es auf Anwendungsebene nicht

möglich die Exception zu behandeln, so greift die oberste Schicht der Anwendung, die sog. Exception-Fassade. Diese ist Teil der Schnittstelle. Sie fängt alle Exceptions, die nicht durch die Anwendung selbst behandelt werden konnten. Die Exception-Fassade bereitet diese Exceptions für den Aufrufer gemäß der Schnittstellenbeschreibung auf und reicht sie an den Aufrufer weiter (siehe ③). Die hier weitergereichten Exceptions werden als Transport-Exceptions bezeichnet.

Anmerkung zur Verknüpfung von Fehlern in Anwendungs- und Datenbank-Logs

Wie in der Abbildung 1 zu sehen, werden Fehler spätestens auf der Schnittstellenebene behandelt und geloggt. Bei aufgetretenen Datenbank-Fehlern liefert die Datenbank-Schicht die Datenbank-Fehlermeldung als Teil der Fehlermeldung zurück. Spätestens an dieser Stelle wird also der Fehler in das Anwendungslog geschrieben, inklusive eines Zeitstempels und des Stack-Traces. Nähere Details zum Logging finden sich im Technischen Feinkonzept Logging [LoggingKonzept]. Mit Hilfe des Zeitstempels, der Datenbank-Fehlermeldung und des Stack-Traces kann dann die Verbindung zur Fehlermeldung im Datenbank-Log hergestellt werden.

Die Verknüpfung von Anwendungsfehlern über mehrere Anwendungen bzw. deren Log-Dateien hinweg geschieht über zwei IDs, die in *Kapitel 4.3* beschrieben sind.

4.2. Ausnahmen und Rückgabewerte

Für die Fehlerbehandlung ist es wichtig, den Unterschied zwischen Ausnahmen und Rückgabewerten zu verstehen und zu wissen wann sie eingesetzt werden.

Rückgabewerte sind das Ergebnis von Methodenaufrufen. Über definierte Werte (Rückgabewerte, engl. Return-Codes) kann der Erfolg eines Aufrufes signalisiert werden. Validierungen, also die (fachliche) Prüfung von Eingabewerten erfolgt in der Regel über Rückgabewerte (Fehlerlisten o.Ä.).

Ausnahmen (englisch *exception*) sind ein Konzept moderner Programmiersprachen, um die Bearbeitung von Fehlersituationen in gewissen Grenzen zu erzwingen und vom Compiler überprüfen zu lassen. Sie dienen dazu, Fehler an den Aufrufer zur Behandlung weiterzureichen. Ausnahmen sind zum Signalisieren von abnormen Ergebnissen gedacht. Sie werden eingesetzt, wenn ein Fehler vorliegt.

Eine Java-Methode kann Ergebnisse auf zwei Wegen zurückgeben. Im einfachsten Fall wird ein Ergebnistyp zurückgegeben. Daneben kann eine Methode auch eine Ausnahme (Exception) auslösen. Ausnahmen sollten nur benutzt werden, um ein abnormales Verhalten zu signalisieren. Für die Fehlerbehandlung sind in der Regel Ausnahmen zu verwenden.

Bei der Betrachtung von Komponentengrenzen spielen Ausnahmen eine besondere Rolle. Komponenten werden durch ihre Schnittstellen beschrieben, wozu auch die verwendeten Exceptions gehören. Eine

Komponente darf nur die in ihrer Schnittstelle aufgeführten Exceptions (oder Unterklassen hiervon) an den Aufrufer weiterreichen. Alle übrigen Exceptions müssen von der Komponente behandelt werden oder in eine Exception der Schnittstellendefinition transformiert werden.

Ausnahmen können als Nachricht an einen menschlichen Nutzer dienen, z. B. in einer Log-Datei, oder bereits von aufrufenden Komponenten sinnvoll behandelt werden. Daher ist es erforderlich, die Fehlersituation so genau wie möglich zu beschreiben. Dazu ist eine Klassifikation in fachliche und technische Fehler sinnvoll. Fachliche Fehler treten bei der korrekten Nutzung einer Komponenten-Schnittstelle auf. Technische Fehler werden von der unterliegenden Technik durch eine Ausnahme signalisiert.

Daher benötigt jede Komponente drei Exception-Hierarchien:

- Komponenten-Ausnahmen (technisch/fachlich erwartet),
- Komponenten-Ausnahmen (technisch unerwartet) und
- Schnittstellen-Ausnahmen (technisch/fachlich erwartet).

Diese werden im folgenden Kapitel 4.3 beschrieben.

4.3. Design von Fehlerklassen

Exceptions werden in erwartete und unerwartete Exceptions unterteilt. Technisch werden diese beiden Arten in Java als checked (erwartete) bzw. unchecked (unerwartete) Exceptions abgebildet. Aus Sicht der Anwendung werden Exceptions in fachliche und technische Exceptions unterteilt.

Aus der Tatsache, dass fachliche Fehler nie unerwartet sein können und behandelt werden müssen, ergibt sich, dass es keine fachlichen unerwarteten Exceptions geben darf. Technische Fehler sind dagegen nur manchmal sinnvoll behandelbar. Sie sind somit in der Regel unerwartet.

Technische erwartete Exceptions sind einzusetzen, sofern mit einem technischen Fehler zu rechnen ist, welcher sinnvoll behandelt werden kann.

Dadurch ergibt sich folgende Exception-Hierarchie:

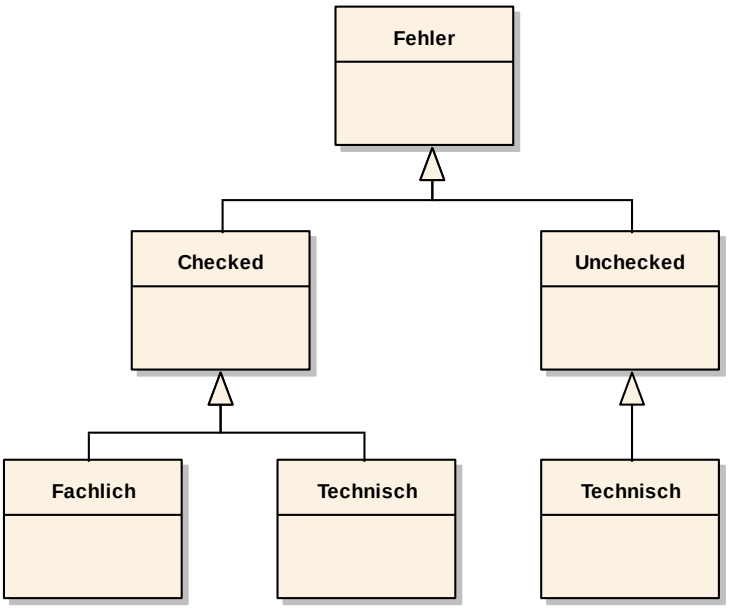

Abbildung 2: Abstrakte Exception Hierarchie

Grundsätzlich lassen sich also folgende Regeln für die Verwendung festhalten:

- Fachliche Exceptions sind immer checked.
- Behandelbare technische Exceptions sind checked.
- Nicht behandelbare technische Exceptions sind unchecked.

Neben der oben aufgeführten Hierarchie, in die sich alle Exceptions einteilen lassen, haben alle Exceptions eine gemeinsame Menge an Attributen, siehe Abbildung 3.

- Fehlertext, mit der Information was passiert ist.
- Ausnahme-ID: referenziert den Fehler(-text) und dient als Referenz für die Art des Fehlers.
- Unique-ID: eineindeutige Nummer in der Anwendungslandschaft und dient als Referenz für die Instanz des Fehlers. Sie ist eine Referenz auf den aufgetretenen Fehler.

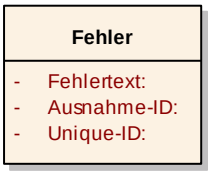

Abbildung 3: Attribute von Fehlern

4.4. Erstellen von Exceptions

4.4.1 Exceptions des Anwendungskerns

Aus den Vorgaben zum Design der Fehlerklassen in *Kapitel 4.3*, resultiert die folgende Exception-Hierarchie, die beispielhaft Exceptions der Beispiel-Anwendung definiert:

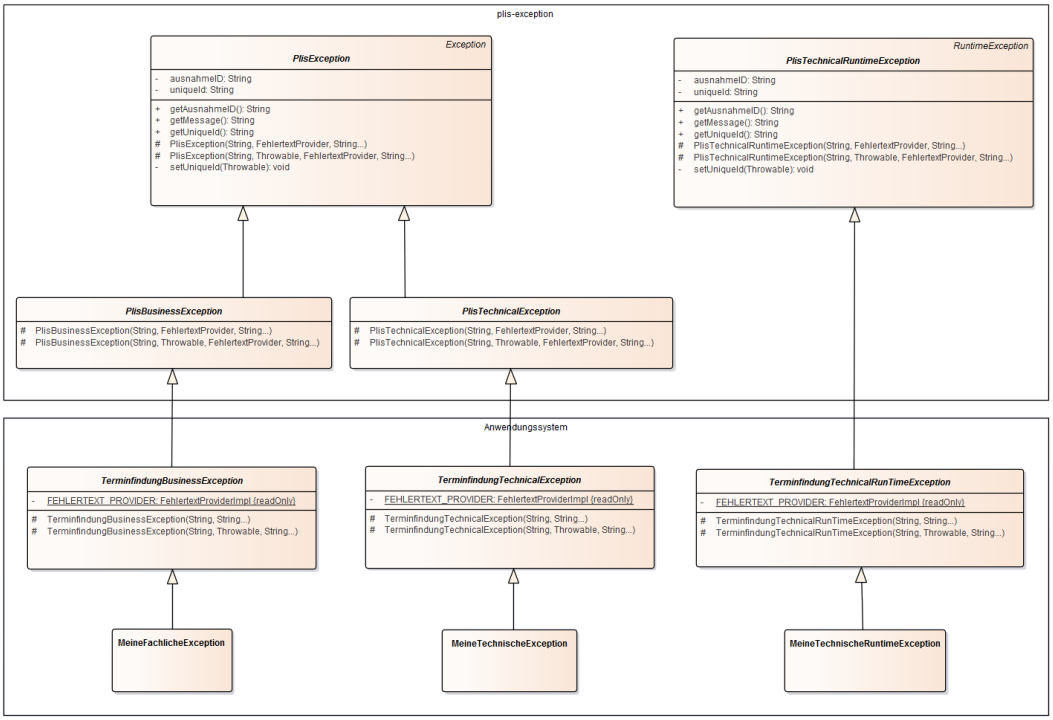

Abbildung 4: Exception-Hierarchie innerhalb einer Anwendung

Abbildung 4 zeigt die verschiedenen Hierarchiestufen von Fehlern. Auf oberster Ebene befinden sich die abstrakten Implementierungen für checked (`PlisException`) und unchecked (`PlisTechnical-RuntimeException`) Exceptions. Diese Oberklassen sind für alle Exceptions innerhalb einer Anwendung zu verwenden. Diese werden als eigenständige Bibliothek (`isy-exception-core`) angeboten und befinden sich im Paket `de.bund.bva.pliscommon.exception`. Sie verwalten die Ausnahme-ID, generieren eine UUID und laden den Fehlertext. Die Ausnahme-ID referenziert den Fehler(-text) und unterstützt den Nutzer bzw. den Betrieb beim Erkennen der Fehlerart, da ein bestimmter Fehler immer die gleiche Ausnahme-ID besitzt. Die generierte UUID ist eine im System eineindeutige Nummer, die beim Erstellen der Exception vergeben wird. Sie ist, wie die Ausnahme-ID, Teil der Fehlernachricht und dient dazu, einen aufgetretenen Fehler im System eindeutig zu referenzieren. Tritt nun ein Fehler bei mehreren Nutzern des Systems auf, kann mit Hilfe dieser UUID der Fehler, der bei einem bestimmten Nutzer auftrat, in den Log-Dateien der Anwendung identifiziert werden.

Werden in einer Anwendung Exceptions benötigt, so müssen zuerst vier eigene abstrakte Oberklassen für die Anwendungs-Exceptions abgeleitet werden (hier: `TerminfindungException`, `TerminfindungBusinessException`, `TerminfindungTechnicalException` und `TerminfindungTechnicalRuntimeException`).

Die Klassen `TerminfindungException` und `TerminfindungTechnicalRuntimeException` sind die abstrakten Oberklassen innerhalb einer Anwendung für checked und unchecked Exception. `TerminfindungException` wird über die Kindklassen

`TerminfindungBusinessException` und `TerminfindungTechnicalException` letztlich noch in fachliche und technische Exceptions unterschieden. Die Anwendungsoberklassen besitzen jeweils eine Referenz auf einen anwendungsspezifischen `FehlertextProvider`. Dieser wird benötigt, um die Fehlertexte zu laden. Diese vier Exceptions sind ebenfalls abstrakt, da auch diese Exceptions rein zur Unterscheidung der Art der Exception innerhalb der Anwendung dienen.

Die letztlich in einer Anwendung eingesetzten Exceptions werden dann von den genannten Klassen `TerminfindungBusinessException`, `TerminfindungTechnicalException` und `TerminfindungTechnicalRuntimeException` abgeleitet.

Die gezeigten Basis-Exceptions der Vorlageanwendung sind im Paket `de.msg.terminfindung.common.exception` abgelegt.

Eine Anwendung besitzt Exceptions auf zwei Ebenen. Auf der Anwendungsebene liegen alle Exceptions die querschnittlich, also von mehreren Komponenten, genutzt werden. Diese Exceptions gehören in das Paket:

`<organisation>`[1]`.<domäne>.<anwendung>.common.exception`

Die zweite Ebene der Exceptions ist die Komponentenebene. Hier liegen alle Exceptions die komponentenspezifisch sind, also nur von einer einzigen Komponente genutzt werden. Diese Exceptions gehören in das Paket:

`<organisation>.<domäne>.<anwendung>.core.<komponente>`

Konstruktoren

Die abstrakten Exceptions einer Anwendung müssen alle vier Konstruktoren implementieren. Die letztlich eingesetzten Exceptions implementieren nur die Konstruktoren, die benötigt werden. Eine Beispiel-Implementierung hierfür befindet sich in der Vorlageanwendung. Dies ist sinnvoll, um Aufwände bei der Erstellung von Exceptions zu sparen, da in diesem Fall lediglich der Konstruktor der Oberklasse aufgerufen werden muss.

Beispiel für eine fachliche Exception Hierarchie:

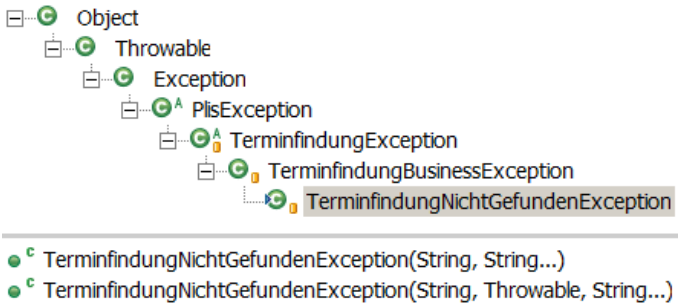

Abbildung 5: Beispiel fachliche Exception Hierarchie

[1] z.B. `de.bund.bva`

Das Beispiel in *Abbildung 5* zeigt eine fachliche Exception der Vorlageanwendung. Die fachliche Exception `TerminfindungNichtGefundenException` besitzt in diesem Beispiel nicht alle möglichen Konstruktoren. Dies dient lediglich der Veranschaulichung. Wie oben erwähnt ist es nicht notwendig, immer alle Konstruktoren zu implementieren. Voraussetzung für das Erstellen dieser Exception sind die Basis-Exceptions der Anwendung (hier `TerminfindungException` und `TerminfindungBusinessException`).

Die *Tabelle 1* erläutert die Bedeutung der Argumente der Konstruktoren.

TerminfindungNichtGefundenException	String	Throwable (optional)	String... (optional)
	Ausnahme-ID	Original-Exception, die gefangen wurde.	String oder String-Array mit Variablenwerten, für Platzhalter in parametrisierten Fehlertexten.

Tabelle 1: Argumente der Konstruktoren von Exceptions des Anwendungskerns

Beispiel für eine technische Runtime-Exception Hierarchie:

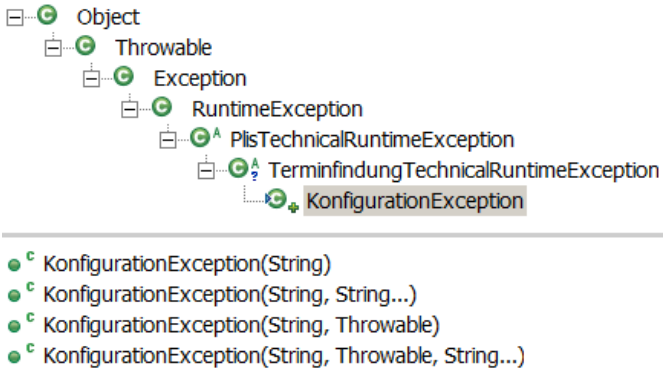

Abbildung 6: Beispiel technische Runtime-Exception Hierarchie

Die Abbildung 6 zeigt die technische Runtime-Exception `KonfigurationException`. Diese Exception könnte dafür verwendet werden, um bei einem Konfigurationsfehler z.B. "Konfigurationsparameter nicht gesetzt" geworfen zu werden. Die Exception ist eine `RuntimeException`, da ein solcher Fehler nicht behandelbar wäre. Um nun eine solche Klasse zu erstellen, muss zuvor nach obigem Schema (siehe *Abbildung 4*) die entsprechende Oberklasse erstellt worden sein.

Das Beispiel enthält wiederum alle möglichen Konstruktoren. Dies dient jedoch auch hier nur der Veranschaulichung. Es ist für Exceptions im Anwendungskern nicht notwendig, alle Konstruktoren zur Verfügung zu stellen. Eine Beschreibung der Argumente der Konstruktoren befindet sich *Tabelle 1*.

Die unter *Abbildung 5* und *Abbildung 6* dargestellten Konstruktoren sind notwendig, um zu gewährleisten, dass alle Exceptions immer eine Ausnahme-ID besitzen, die den Fehlertext identifiziert, d. h. andere Konstruktoren sind nicht gestattet.

Dokumentation

Checked Exceptions sind in Methoden-Signaturen zu deklarieren und im JavaDoc-Kommentar mittels `@throws` zu dokumentieren. Unchecked Exceptions sind nicht in den Methoden-Signaturen zu deklarieren, aber mittels `@throws` im JavaDoc-Kommentar zu dokumentieren.

4.4.2 Werfen einer Exception

Der folgende Abschnitt beschreibt das Werfen einer technischen checked Exception. Das Vorgehen wird nur für technische checked Exceptions beschrieben, da das Vorgehen für alle Arten von Exceptions gleich ist.

Gemäß der Anforderungen aus *Kapitel 3* sollte die Fehlerbehandlung übersichtlich sein. Zur Sicherstellung der Übersichtlichkeit darf die Anzahl der verwendeten Exceptions die Anzahl möglicher Behandlungen nicht überschreiten. Es sollte also für jede mögliche Fehlerbehandlung auch nur eine Exception geworfen werden. Sofern sie nicht behandelbar sind, sind hierfür technische unchecked Exceptions zu verwenden. Wenn mehrere Exceptions zur gleichen Fehlerbehandlung führen, macht es keinen Sinn, mehr als eine Exception hierfür zu deklarieren.

In einer Anwendung gibt es nun unter Umständen aber eine größere Anzahl an technischen Fehlern, die die Anwendung nie verlassen. Dies würde zu einer entsprechenden großen Anzahl an Fehlertexten führen, die nicht mehr verwaltbar wäre. Daher muss es in jeder Anwendung eine Ausnahme-ID geben mit einem generischen Fehlertext, der einen Platzhalter besitzt. Als feste Nummer wird für alle Anwendungen die `0001` festgelegt. Ein Aufruf einer solchen Exception mit einem generischen Fehlertext sieht dann wie folgt aus:

```
new MeineTechnischeException(FehlerSchluessel.MSG_ALLGEMEINER_FEHLER, "XYZ");
```

Die Konstante `FehlerSchluessel.MSG_ALLGEMEINER_FEHLER` referenziert einen generischen Fehlerstring, welcher einen Platzhalter besitzt:

Konstante:

```
/** Generische Exception für alle unbekannten Fehler. */
public static final String MSG_ALLGEMEINER_FEHLER = "TRMIN90001";
```

Fehlertext:

```
TRMIN90001 = Es ist ein allgemeiner Fehler im Modul Terminfindung aufgetreten.
```

Beim Einsatz von Exceptions muss immer eine Konstante zur Referenzierung von Fehlern verwendet werden. Die Fehlertexte dürfen nicht direkt mit dem String referenziert werden (z. B. hier `TRMIN90001`).

Beim Aufruf einer Exception wird im einfachsten Fall lediglich eine Ausnahme-ID übergeben, welche den Fehlertext identifiziert:

```
new TerminfindungNichtGefundenException
                (FehlerSchluessel.MSG_TERMINFINDUNG_NICHT_GEFUNDEN)
```

Der Konstruktor der Exception ruft den Konstruktor der abstrakten Eltern-Klasse auf (hier `TerminfindungBusinessException`):

```
public TerminfindungNichtGefundenException(String ausnahmeID) {
    super(ausnahmeID);
}

protected TerminfindungBusinessException(String ausnahmeID) {
    super(ausnahmeID);
}
```

Dieser Konstruktor wiederum ruft den Konstruktor seiner Eltern-Klasse auf (hier `TerminfindungException`), welcher die oberste Exception-Hierarchie-Stufe einer Anwendung darstellt:

```
protected TerminfindungException(String ausnahmeID) {
    super(ausnahmeID, FEHLERTEXT_PROVIDER);
}
```

Die weitere Kommunikation bis zur Erstellung des eigentlichen Fehlertextes ist in der *Abbildung 7* skizziert.

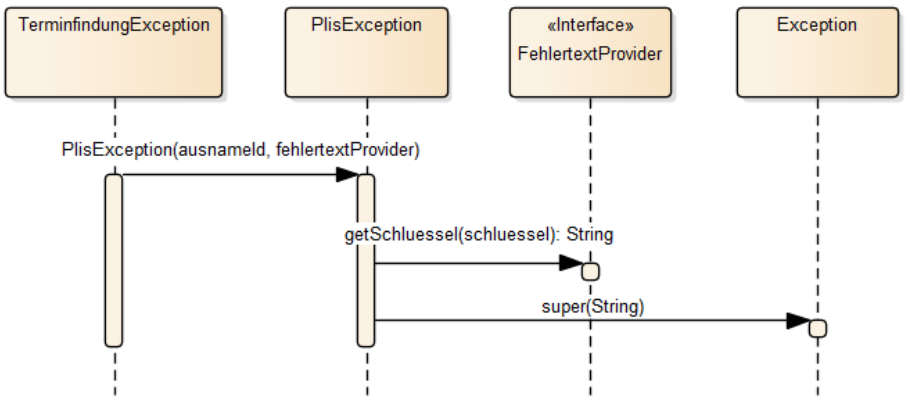

Abbildung 7: Abstrakter Ablauf der Erstellung einer Exception

Die `TerminfindungException` hält eine Referenz zu einem `FehlertextProvider` (siehe *Kapitel 4.6.1*), welcher die Möglichkeit bietet Fehlertexte auszulesen. Diese Referenz und die übergebene `Ausnahme-ID` werden an den Konstruktor der `PlisException` übergeben, welcher nun den Fehlertext lädt. Hierzu ruft er auf dem `FehlertextProvider` die `getMessage()`-Methode auf und bekommt den Fehlertext zurückgeliefert. Durch einen Aufruf des Konstruktors der Oberklasse `Exception` wird der Fehlertext gesetzt.

Bis dato hat der Text den Aufbau:

Fehlertext

Die `Plis-Exception`-Klassen überschreiben aber die `getMessage()`-Methode von `Exception` und erweitern den Fehlertext bei einem lesenden Zugriff. Der Fehlertext wird um die Ausnahme-ID und die UUID erweitert. Dies geschieht über die Klasse `FehlertextUtil`, damit die Formatierung der Fehlertexte an einer zentralen Stelle gekapselt ist.

Der Text hat dann folgenden Aufbau:

```
#AusnahmeId Fehlertext #UUID
```

Der Fehlertext wird in dieser Form aufbereitet, um sicherzustellen, dass sowohl die Ausnahme-ID als auch die UUID

- beim Loggen der Exception immer in die Log-Datei der Anwendung geschrieben werden, ohne dass eine spezielle Implementierung des Loggings notwendig ist,
- beim Loggen der Exception durch den Aufrufer einer Schnittstelle immer in die Log-Datei der aufrufenden Anwendung geschrieben werden, ohne dass eine spezielle Implementierung des Loggings notwendig ist und
- der Anwender, sofern er den Fehlertext angezeigt bekommt, auch immer die Ausnahme-ID und die UUID sieht, um diese gegebenenfalls direkt weitergeben zu können.

4.4.3 Exceptions für Anwendungsschnittstellen

In den vorhergehenden Kapiteln wurde das Werfen von Fehlern in der Anwendung beschrieben. In diesem Kapitel geht es um Exceptions, die zur Schnittstelle einer Anwendung gehören und vom Aufrufer verarbeitet werden. Diese werden in IsyFact als Transport-Exceptions bezeichnet.

Neben den Vorgaben zum Design der Fehlerklassen in *Kapitel 4.3* gelten für Transport-Exceptions noch weitere Vorgaben, da diese an die Aufrufer weitergereicht werden.

Für Exceptions an den Anwendungsschnittstellen gelten weitere Vorgaben:

- Sie erben immer von `PlisBusinessToException` oder `PlisTechnicalToException` und implementieren somit immer `Serializable`,
- stellen die Felder Ausnahme-ID, UUID und Fehlernachricht zur Verfügung und
- erben nicht von internen Anwendungsexceptions.

Daraus ergibt sich für Transport-Exceptions folgende Hierarchie:

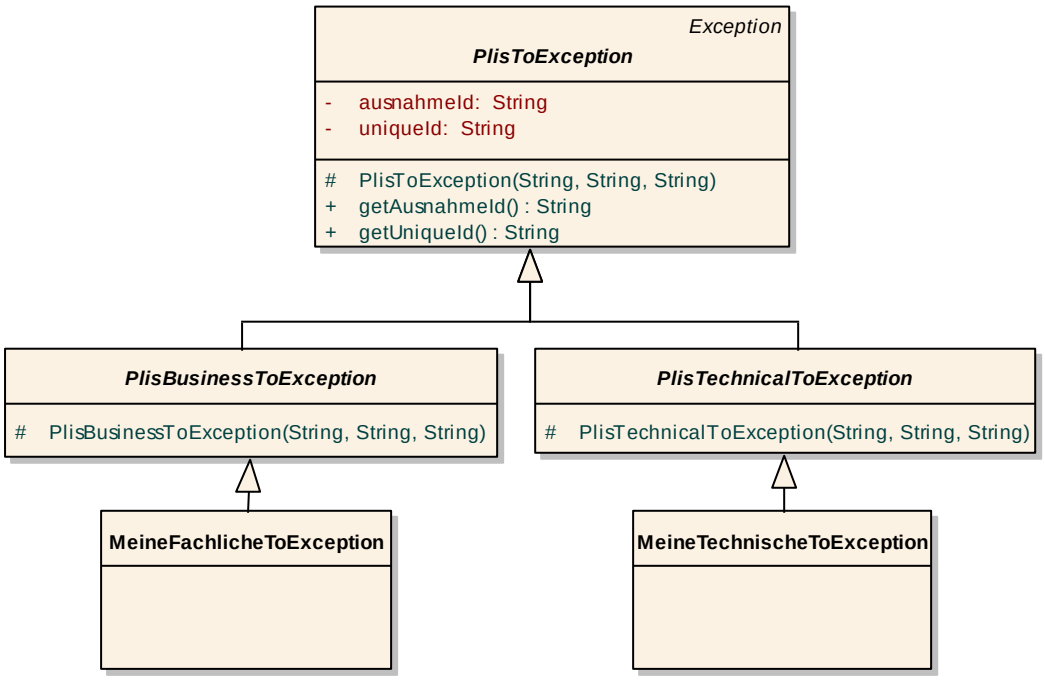

Abbildung 8: Exception Hierarchie für Transport-Exceptions

Weiterhin werden für die genannten Technologien, welche für die Anwendungsschnittstellen verwendet werden, folgende Vorgaben gemacht:

- **SOAP** (pro Operation)
  - Definition von 0..1 technischen Exceptions (gleich für alle Operationen einer Schnittstelle)
  - Definition von 0..n fachlichen Exceptions
  - Übermittlung der Ausnahme-ID
  - Übermittlung der UUID
  - Übermittlung des Fehler-Typs („Name“ der Exception)
  - Übermittlung der Fehler-Nachricht (kein Stack-Trace)
- **REST** (keine Exceptions)
  - Übermittlung der Ausnahme-ID
  - Übermittlung der UUID
  - Übermittlung von Fehler-Kategorie (technisch/Art des fachlichen Fehlers)
  - Übermittlung von Fehler-Nachricht (kein Stack-Trace!)

- **Spring HttpInvoker** (pro Methode)
  - Definition von 0..1 technische Exceptions (gleich für alle Methoden einer Schnittstelle)
  - Definition von 0..n fachliche Exceptions
  - Übermittlung der Ausnahme-ID
  - Übermittlung der UUID
  - Übermittlung von Fehler-Nachricht (kein Stack-Trace!)

4.4.4 IsyFact-Bibliotheken für Fehlerbehandlung

Die in den vorigen Abschnitten beschriebenen abstrakten Oberklassen, die zur Umsetzung der Exception-Hierachie notwendig sind, werden über die IsyFact-Bibliotheken `isy-exception-core` und `isy-exception-sst` in neue Anwendungen integriert.

Dabei enthält die Bibliothek `isy-exception-core` die notwendigen Klassen für den Anwendungskern, die Bibliothek `isy-exception-sst` die Klassen für die Schnittstellen (Transport-Exceptions).

4.5. Behandlung von Exceptions

Die in *Kapitel 4.4.1* aufgeführten Fehlerarten müssen (irgendwann) behandelt werden. Der Zeitpunkt hängt von den Möglichkeiten der Fehlerbehandlung ab, die zum Zeitpunkt des Auftretens des Fehlers existieren.

Grundsätzlich gilt, dass der Aufrufer alle Fehler behandelt, die er behandeln kann, und alle übrigen weiterreicht.

Die Fehlerbehandlung besitzt folgende Ausprägungen:

- Protokollieren und Ignorieren
- Protokollieren und Schaden begrenzen, z.B. DB-Verbindung freigeben
- Protokollieren, Warten und erneut Versuchen
- Original-Exception weiterwerfen
- Protokollieren und endgültige Exception erzeugen

Wann bzw. ob ein Fehler behandelt werden kann, ist im Einzelfall zu entscheiden. Die ersten vier Ausprägungen sind Möglichkeiten innerhalb einer Komponente oder einer Anwendung. Die Fehlerbehandlung entspricht den gängigen `try-catch`-Blöcken mit entsprechender Verarbeitung der Exception, z. B. Weiterreichen oder Behandeln und Loggen. Das folgende Code-Beispiel zeigt das Weiterwerfen der Original-Exception:

```
try {
   verwaltung.leseTerminfindung(terminfindungsRefNr);
} catch (TerminfindungNichtGefundenException ex) {
   // Exception kann nicht behandelt werden, also wird sie weitergereicht
  throw ex;
}
```

Die letzte Variante ist die endgültige Fehlerbehandlung, die in einer Exception-Fassade (siehe *Abbildung 9*) stattfindet.

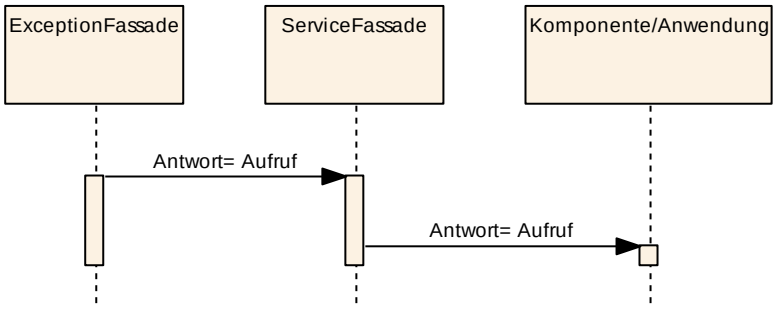

Abbildung 9: Aufruf von der Schnittstelle zum Anwendungskern

Die `ExceptionFassade` bildet die Klammer um einen Aufruf an die Anwendung und ist für die Top-Level Fehlerbehandlung zuständig. Sie leitet den Aufruf an die `ServiceFassade` (Details, siehe [AnwendungskernDetailkonzept]) weiter, welche die Transaktionsklammer um den Aufruf bildet und ruft die Anwendung bzw. die Komponente auf. Dieser zweistufige Prozess ist notwendig, da es unerwartete Exceptions geben kann, wenn die Transaktion geschlossen wird, also ein Commit durchgeführt wird. Diese Exceptions treten außerhalb der eigentlichen Anwendung auf. Daher muss die Exception-Fassade als Schicht über der Transaktionsklammer liegen, um auch diese Fehler abzufangen, zu loggen, in Transport-Exceptions umzuwandeln und an den Aufrufer weiterzureichen.

Das Vorgehen für die Fehlerbehandlung nach Abbildung 9 gilt für alle Arten von Außenschnittstellen. Die HttpInvoker-Schnittstelle ist die am häufigsten angebotene Schnittstelle in der Anwendungslandschaft. Aus diesem Grund wurde diese Schnittstelle für das Code-Beispiel gewählt, zur Veranschaulichung der Top-Level Fehlerbehandlung.

Das Code-Fragment zeigt die Fehlerbehandlung in der Exception-Fassade der Meldung im Vorlage-Register für die HttpInvoker-Schnittstelle.

```
public int cdErworben(AufrufKontextTo kontext, CdAblageDatenTo cdAblageDaten)
          throws MeldungVerarbeitungException, MeldungTechnikException {
  // Logging-Kontext setzen.
  NDC.push(kontext.getLoggingKontext());

  try {
      return meldungServiceFassade.cdErworben(kontext, cdAblageDaten);
  } catch (MaxBestandUeberschrittenException ex) {
      LOG.debug("Methode cdErworben fehlgeschlagen.", ex);
      // Exceptions in Schnittstellen-Exceptions transformieren.
      throw (MeldungVerarbeitungException) PlisExceptionMapper.mapException(ex,
              MeldungVerarbeitungException.class);
  } catch (CdRegisterTechnicalRunTimeException ex) {
      LOG.error("Methode cdErworben fehlgeschlagen.", ex);
```

```
      throw (MeldungTechnikException) PlisExceptionMapper.mapException(ex,
              MeldungTechnikException.class);
  } catch (Throwable t) {
      LOG.error("Methode cdErworben fehlgeschlagen.", t);
      // unbekannte Exceptions in Schnittstellen-Exceptions transformieren.
      MeldungTechnikException ex =
            PlisExceptionMapper.createToException(AusnahmeIdUtil.getAusnahmeId(t),
                    new FehlertextProviderImpl(), MeldungTechnikException.class);
      LOG.error("Methode cdErworben fehlgeschlagen: übergebener Fehler: "
              + ex.getMessage());
      throw ex;
  } finally {
      // Auf jeden Fall am Ende den Logging-Kontext entfernen.
      NDC.pop();
  }
}
```

Das obige Code-Beispiel fängt alle Exceptions und wandelt diese in entsprechende Transport-Exceptions um. Als erwartete Exceptions gibt es hier die Exception `MaxBestandUeberschrittenException`. Diese wird, sofern sie auftritt, in eine `MeldungVerarbeitungException` umgewandelt und weitergereicht. Zu beachten ist, dass in das Error-Log nur betrieblich relevante Fehler geschrieben werden sollen. Fachlich Fehler sind in der Regel irrelevant für den Betrieb. Daher wird die `MaxBestandUeberschrittenException` ins Debug-Log geschrieben.

Weitere erwartete Fehler gibt es nicht, somit wird nun eine Fehlerbehandlung für unerwartete Fehler der Anwendung durchgeführt (alle Exceptions vom Typ `CdRegisterTechnicalRunTimeException`). Als letzte mögliche Fehlerbehandlung werden alle unerwarteten Exceptions vom Typ `Throwable` gefangen.

Der erste Block in diesem Beispiel behandelt eine fachliche Exception. Die restlichen Blöcke behandeln unerwartete Exceptions. Fachliche Exceptions müssen immer in fachliche Transport-Exceptions umgewandelt werden, alle anderen Exceptions sind in technische Transport-Exceptions umzuwandeln.

Alle Blöcke einer solchen Fassade auf der Anwendungsgrenze verwenden die Klasse `PlisExceptionMapper` (siehe *Kapitel 4.5.1*) zur Umwandlung der Anwendungs-Exceptions in Transport-Exceptions und zur Erstellung von Transport-Exceptions. Letzteres wird im letzten `catch`-Block des obigen Code-Beispiels genutzt, da in diesem Fall keine Exception vom Typ `PlisException` bzw. `PlisRuntimeException` vorhanden ist und somit keine Ausnahme-ID, UUID und kein Fehlertext zu übernehmen sind. In diesem Fall ist die benötigte Ausnahme-ID zu berechnen, mit Hilfe der Klasse `AusnahmeIdUtil` (siehe *Kapitel 4.5.1*).

Die `catch`-Blöcke für anwendungsinterne Runtime-Exceptions (vom Typ `<anwendung>TechnicalRunTimeException`) und alle übrigen unerwarteten Exceptions (`Throwable`) müssen immer implementiert

werden. Hierdurch wird verhindert, dass die Schnittstelle nicht spezifizierte Exceptions weiterreicht.

4.5.1 Mapping von Exceptions

Auf der Schnittstelle einer Anwendung müssen interne Anwendungs-Exceptions in Transport-Exceptions umgewandelt werden, bzw. es müssen neue Transport-Exceptions erstellt werden. Hierfür stellt das Paket `plis-exception` eine eigene Klasse zur Verfügung: `PlisExceptionMapper` (siehe *Abbildung 10*).

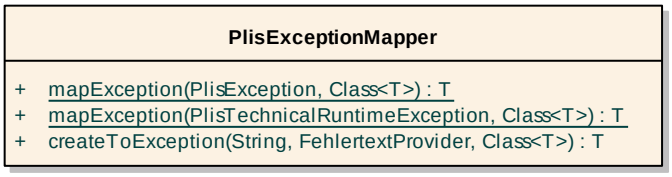

Abbildung 10: PLIS Exception Mapper

Die Klasse `PlisExceptionMapper` bietet zwei statische Methoden an, um aus `PlisException` und `PlisRuntimeException` entsprechende Transport-Exceptions zu erstellen. Hierfür muss lediglich die umzuwandelnde Exception und die Klasse der gewünschten Transport-Exception mitgegeben werden. Hier ein Beispiel für das Mappen einer technischen `PlisException ex` in eine technische Transport-Exception:

```
PlisExceptionMapper.mapException(ex, MeldungTechnikException.class)
```

Neben den `Plis-(Runtime-)Exceptions` können weitere Exceptions auftreten. Diese besitzen jedoch keine Ausnahme-ID oder eine UUID, z. B. Runtime-Exceptions von eingesetzten Frameworks wie Hibernate. Auch diese Exceptions müssen in Transport-Exceptions umgewandelt werden. Diese Transport-Exceptions werden mittels der `createToException()`-Methode erstellt:

```
MeldungTechnikException ex =
            PlisExceptionMapper.createToException(AusnahmeIdUtil.getAusnahmeId(t),
                        new FehlertextProviderImpl(),
MeldungTechnikException.class);
```

In diesem Beispiel wird für ein `Throwable "t"` eine technische Transport-Exception erzeugt. Aus dem Code-Beispiel ist außerdem ersichtlich, dass zur Erstellung einer Transport-Exception aus einer unbekannten Exception noch eine weitere Klasse benötigt wird, die Klasse `AusnahmeIdUtil`. Dies ist, wie schon in *Kapitel 4.4.4* erwähnt, notwendig, da keine Ausnahme-ID bekannt ist.

Die Klasse `AusnahmeIdUtil` bietet eine Methode zur Analyse einer übergebenen Exception. Ihr Rückgabewert ist die zur Exception passende Ausnahme-ID, siehe *Abbildung 11*.

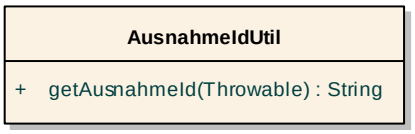

Abbildung 11: Ausnahme-ID Util

Diese Klasse ist anwendungsspezifisch und für jede Anwendung zu implementieren. Die Klasse ist als Teil des Paketes

`<organisation>.<domäne>.<anwendung>.common.exception`

zu erstellen und muss immer als Klassennamen `AusnahmeIdUtil` besitzen und die statische Methode `getAusnahmeId(Throwable)` zur Verfügung stellen.

Hier eine mögliche Implementierung für das Mapping von Exceptions auf Ausnahme-IDs des Vorlage-Registers.

```
/**
  * Liefert eine passende AusnahmeID zu einer übergebenen Ausnahme.
  * @param throwable
  *            Throwable, welches analysiert werden soll
  * @return String Ausnahme-ID
  */
 public static String getAusnahmeId(Throwable throwable) {
     if (throwable instanceof DataAccessException) {
         // generische Datenbank-Fehlermeldung
         return FehlerSchluessel.MSG_GENERISCHER_DB_FEHLER;
     } else if (throwable instanceof TransactionException) {
         // generische Datenbank-Fehlermeldung
         return FehlerSchluessel.MSG_GENERISCHER_DB_FEHLER;
     } else if (throwable instanceof JmxException) {
         // generische JMX-Fehlermeldung
         return FehlerSchluessel.MSG_GENERISCHER_JMX_FEHLER;
     } else if (throwable instanceof CdRegisterBusinessException) {
         LOG.warn("Es wurde eine CdRegisterBusinessException analysiert. "
        + "Eigentlich sollte diese Verarbeitung über catch-Blöcke geschehen.");
         // Fehler-Code auslesen
         return ((CdRegisterException) throwable).getAusnahmeID();
     } else if (throwable instanceof CdRegisterTechnicalException) {
         LOG.warn("Es wurde eine CdRegisterTechnicalException analysiert. "
        + "Eigentlich sollte diese Verarbeitung über catch-Blöcke geschehen.");
         // Fehler-Code auslesen
         return ((CdRegisterException) throwable).getAusnahmeID();
     } else if (throwable instanceof CdRegisterTechnicalRunTimeException) {
         // Fehler-Code auslesen
         return ((CdRegisterTechnicalRunTimeException)
                 throwable).getAusnahmeID();
     }
     // Kein Mapping Möglich: generische Fehlermeldung
```

```
        LOG.debug("Die Exception der Klasse " + throwable.getClass()
            + "wurde keiner Kategorie zugeordnet: "
            + "Ausgabe einer generischen Fehlermeldung.");
        return FehlerSchluessel.MSG_GENERISCHER_FEHLER;
    }
```

Die Ermittlung der Ausnahme-ID (`AusnahmeIdUtil.getAusnahmeId (Throwable throwable)`) prüft auch auf die internen Exceptions der Anwendung. Grundsätzlich sollte es aber nie zu einer positiven Prüfung dieser Bedingungen kommen, da diese Funktionalität nur auf der Schnittstelle verwendet wird. Sollte hier also ein Treffer für interne Exceptions auftreten, so wurden die `catch`-Blöcke nicht sauber implementiert (z. B. wurde einfach nur `catch Throwable` verwendet). Dies würde dazu führen, dass die Original-Nachricht überschrieben würde, was bei der Verwendung von Fehlertexten mit Platzhaltern zu einem Informationsverlust für den Aufrufer führt.

Das obige Code-Beispiel bzw. die gesamte Klasse `AusnahmeIdUtil` aus dem Vorlage-Register kann als Template für andere Anwendungen genutzt werden. Hierfür sind lediglich kleine Anpassungen notwendig: die Prüfung auf anwendungsinterne Exceptions ist anzupassen und die verwendeten Fehlerschlüssel sind auf die Anwendung anzupassen.

Neben der oben gezeigten Fehlerbehandlung für HttpInvoker-Schnittstellen gibt es auch eine Fehlerbehandlung für Axis-basierte Schnittstellen, also SOAP und REST. Im Unterschied zu HttpInvoker-Schnittstellen werden die Transport-Exceptions nicht in Ausnahmen von Typ `Exception`, sondern im Fall von SOAP in `AxisFaults` umgewandelt. Auch hier gilt das generelle Prinzip auf oberster Ebene der Anwendung eine Exception-Fassade zu verwenden, siehe *Abbildung 9*.

4.6. Fehlertexte und deren Einsatz

Fehlertexte müssen in `ResourceBundles` abgelegt werden. Die Ablage der Fehlertexte wird durch das *Technische Feinkonzept: Überwachung und Konfiguration* [ÜberwachungKonfigKonzept] vorgegeben, das Laden der Dateien wird in Spring durch Holder-Klassen realisiert und ist im *Detailkonzept_Komponente_Anwendungskern* [AnwendungskernDetailkonzept] erläutert.

Als Schlüssel werden die Ausnahme-IDs verwendet. Diese setzen sich aus fünf Buchstaben und fünf numerischen Zeichen zusammen:

[A-Z]{5}[0-9]{5}

Ausnahme-IDs der Fachanwendung „ABCDE“ könnten dann z.B. wie folgt aussehen: `ABCDE10034`

Die Ausnahme-IDs sind in Nummernkreise für die einzelnen Komponenten unterteilt. Ein Nummernkreis umfasst immer 1000 Nummern, d. h. es gibt die Kreise 00xxx bis 99xxx. Bei der Erstellung einer neuen Anwendung ist

in der Spezifikations- bzw. Konstruktionsphase festzulegen, welche Komponente welchen Nummernkreis verwenden muss.

Die Ausnahme-IDs referenzieren immer einen Fehlertext. Die referenzierten Fehlertexte können mit Platzhaltern versehen werden ({1}, {2} etc.), um den Text um kontextbezogene Daten zu erweitern, z. B.

Der Konfigurationsparameter "**{0}**" enthält ungültigen Wert "**{1}**".

Hierzu wird dem Konstruktor der zugehörigen Exception ein String oder String-Array mit den Werten für die Platzhalter übergeben:

```
new TerminfindungNichtGefundenException(
        VerwaltungFehlerSchluessel.MSG_TERMINFINDUNG_NICHT-GEFUNDEN,
        terminfindungRef.toString());
```

Die Verwendung der Fehlertexte geschieht über Konstanten der Klassen. Jede Komponente besitzt eine eigene Schlüsselklasse, welche die komponentenspezifischen Ausnahme-IDs beinhaltet. Diese Klasse ist abstrakt, muss dem Namensschema `<Komponente>FehlerSchluessel` entsprechen und im Paket

`<organisation>.<domäne>.<anwendung>.core.<komponente>.konstanten`

abgelegt werden. Die Klasse erbt außerdem noch von der Schlüsselklasse für die gesamte Anwendung, um Zugriff auf allgemeine Ausnahme-IDs, wie z. B. Datenbank-Fehler zu haben, da diese in der Anwendungsklasse spezifiziert sind und für alle Komponenten gleich sind. Die Anwendungsklasse ist im Paket

`<organisation>.<domäne>.<anwendung>.common.konstanten`

abzulegen und muss in jeder Anwendung `FehlerSchluessel` heißen.

Kommen neue Fehlertexte hinzu, so müssen die Schlüssel in einer der oben genannten Klassen als Konstanten hinzugefügt werden. Ausnahme-IDs für allgemeine Fehler müssen in die Anwendungsklasse, komponentenspezifische in die Komponentenklasse. Die Konstanten müssen einen sprechenden Namen tragen und immer mit `MSG_` beginnen, z.B.

```
/** Die im Parameter {0} übergebene Liste ist leer. */
public static final String MSG_LISTE_LEER = "TRMIN90004";
```

4.6.1 Fehlertextprovider

Das Auslesen von Fehlertexten wird durch einen `FehlertextProvider` implementiert. Dieser `FehlertextProvider` ist pro Anwendung zu implementieren und befindet sich im Paket:

`<organisation>.<domäne>.<anwendung>.common.exception`

Zu implementieren sind die zwei `getMessage`()-Methoden des `FehlertextProvider`-Interfaces von PLIS-Exception, siehe *Abbildung 12*.

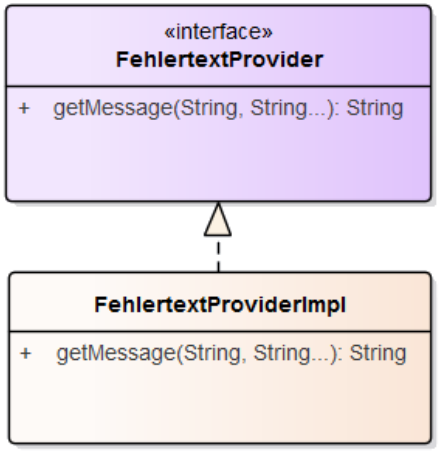

Abbildung 12: Fehlertextprovider

Die Implementierung muss Spring-Mechanismen verwenden, um die Fehlertexte aus einem `ResourceBundle` auszulesen. Dies führt zu einer Vereinheitlichung der Fehlerbehandlung, da sich das Laden von Fehlertexten in den einzelnen Anwendungen nicht unterscheidet.

Eine Beispiel-Implementierung hierfür befindet sich in der Vorlageanwendung.

5. DO's und DON'Ts[2]

Im Folgenden werden Vorgaben gemacht, wie Fehler behandelt werden müssen und wie Fehler nicht behandelt werden dürfen.

5.1. DO's

Log it or throw it
Exceptions sind in der Regel zu behandeln und zu loggen. Ist es nicht möglich die Exception zu behandeln, muss sie an den Aufrufer weitergegeben werden. Die Exception wird im Fall eines Weiterwerfens nicht geloggt. Details zum Logging befinden sich im *Technischen Feinkonzept Logging* [LoggingKonzept].

Nur vorkommende Exceptions verwenden
Nur Exceptions in Methodensignaturen verwenden, die auch vorkommen können. Dies führt sonst zu leeren `catch`-Blöcken oder der Behandlung aller Fehler über das Fangen einer globalen Exception.

Rollback durch Besitzer der Transaktionsklammer
Das Rollback geschieht durch die Schnittstelle, den Dialog oder den Batch, welcher die Transaktionsklammer bildet.

Aufräumen
Bei der Behandlung von Fehlern ist ein geordneter Systemzustand herzustellen, z. B. das Schließen der Datenbankverbindung über einen `finally`-Block.

Throw Early / Failing fast
Fehler sollten früh erkannt werden und zu entsprechenden Ausnahmen führen, bevor sich der Aufruf in tieferen Schichten befindet. Beispiel: Übergibt man `null` an `FileInputStream` wird eine `NullPointerException` in `java.io` geworfen. Passender wäre es aber gleich in der Methode, die `FileInputStream` verwendet auf `null` zu prüfen und eine `Exception` zu werfen.

5.2. DON'Ts

Neben den oben genannten Punkten, wie man Exceptions richtig verwendet, gibt es auch eine Liste von Anti-Patterns, die bei der Verwendung von Exceptions zu Problemen führen und daher vermieden werden sollten:

Interne Exceptions in der Schnittstelle
Interne Exceptions dürfen in der Schnittstelle nicht vorkommen, da diese ansonsten dem Aufrufer bekannt sein müssen. Dies würde zu einer engeren Kopplung von Aufrufer und Aufgerufenem führen und dem Komponentengeheimnis widersprechen.

[2] Siehe: http://www.onjava.com/pub/a/onjava/2003/11/19/exceptions.html
http://today.java.net/pub/a/today/2006/04/06/exception-handling-antipatterns.html

Fluss-Steuerung über Exceptions

Catch-Blöcke dienen der Fehlerbehandlung und dürfen nicht als `else`-Zweig genutzt werden, z. B.

```
try {
    obj = mgr.getObject(id);
} catch (NotFoundException e){
    obj = mgr.createObject(id);
}
```

Ebenso sind GoTos über catch/throw-Konstrukte zu vermeiden:

```
public void useExceptionsForFlowControl() {
    try {
       while (true) {
                    increaseCount();
               }
     } catch (MaximumCountReachedException ex) {}
  //weitere Verarbeitung
}

public void increaseCount() throws MaximumCountReachedException {
    if (count >= 5000) throw new MaximumCountReachedException();
}
```

Leere catch-Blöcke

Wenn dies der Fall ist, dann ist die Exception unnötig oder die Ausnahme muss behandelt werden. Siehe auch *Kapitel 4.4.4*:

```
try {
    myMethod();
} catch (MyException me) {}
//weitere Verarbeitung
```

In Ausnahmefällen, (z. B. `InterruptedException`) kann ein leerer `catch`-Block durchaus sinnvoll sein. In diesem Fall ist ein entsprechender Kommentar im `catch`-Block zu hinterlegen, warum die Exception nicht behandelt wird.

Destruktives Wrappen

Das destruktive Wrappen einer Exception zerstört den StackTrace und ist nur für Exceptions an den Außen-Schnittstellen sinnvoll. Destruktiv gewrappte Exceptions sind in jedem Fall vor dem Wrappen zu loggen.

```
catch (NoSuchMethodException e) {
    throw new MyServiceException("Blah: " + e.getMessage());
}
```

Catch Exception

Global die Elternklasse einer Exception zu fangen ist zu unspezifisch. Dadurch entfällt die Möglichkeit, auf verschiedene Ausnahmen unterschiedlich reagieren zu können:

```
try {
    foo();
} catch (Exception e) {LOG.error("Foo failed", e);}
```

→ Wenn so etwas sinnvoll ist, dann ist die Signatur der aufgerufenen Methode zu überdenken. Ist es nicht möglich die Exceptions der Methode (`foo()`) unterschiedlich zu behandeln, so ist die Methode auf sinnvoll behandelbare Exceptions einzuschränken.

Exception Flut

Nur Exceptions werfen, die auch sinnvoll zu behandeln sind:

```
public void zuViel() throws MeineException,
                            NeAndereException,
                            AuchNeAndereException,
                            NochNeAndereException {
```

Throw Exception

Es sollten aussagekräftige Exceptions verwendet werden, um dem Aufrufer eine adäquate Fehlerbehandlung zu ermöglichen. daher ist folgendes Konstrukt nicht erlaubt:

```
public void keineAussage() throws Exception {
```

Log and throw

Das Loggen und Weiterwerfen von Exceptions führt zu unbrauchbaren Log-Dateien. Tritt eine Exception in einer tiefen Aufrufhierarchie auf, wird ein und dieselbe Exception mehrmals in einer Log-Dateien gespeichert. Dies behindert bei der Fehlersuche. Daher gilt die Regel aus *Kapitel 5.1 Log it or throw it*, d. h. entweder man behandelt und loggt die Exception oder man reicht sie weiter. Die folgenden Code-Fragmente sind daher in Anwendungen gemäß Referenzarchitektur nicht erlaubt:

```
catch (NoSuchMethodException e) {
    LOG.error("Blah", e);   throw e;
}

catch (NoSuchMethodException e) {
    LOG.error("Blah", e);
    throw new MyServiceException("Blah", e);
}

catch (NoSuchMethodException e) {
    e.printStackTrace();
```

```
    throw new MyServiceException("Blah", e);
}
```

Log and return null / Catch and Ignore

Das Ignorieren von Fehlern ist zu vermeiden, da der Aufrufer sonst keinen Fehler bemerkt, den man unter Umständen in der weiteren Verarbeitung berücksichtigen müsste. Folgende Konstrukte sind somit nicht erlaubt:

```
catch (NoSuchMethodException e) {return null;}

catch (NoSuchMethodException e) {LOG.error("Blah", e); return null;}
```

→ Exceptions sollten weitergereicht werden, außer es handelt sich nicht um eine Ausnahme, z. B. `return null` für den Fall, dass nichts gefunden wurde.

throw im finally-Block

Exceptions in `finally`-Blöcken führen zu einem Verlust des Original-Fehlers:

```
try {myMethod();} finally {cleanUp();}
```

→ Wirft `cleanUp()` eine Exception, ist die original Exception von `myMethod()` verloren. Es ist somit nicht gestattet in `finally`-Blöcken Methoden aufzurufen, welche potentiell Exceptions werfen.

Nicht unterstützte Methode gibt null zurück

Null als Rückgabewert einer Methode, sofern sie nicht unterstützt wird, deckt sich mit dem oben aufgeführten Punkt "Catch and Ignore". Der Aufrufer hat in diesem Fall nicht mitbekommen, dass die Methode eigentlich gar nicht unterstützt wird. Im einfachsten Fall tritt eine `NullPointerException` auf, welche aber nicht den eigentlichen Fehlergrund widerspiegelt:

```
public String myMethod() {// Nicht unterstützt.
    return null;
}
```

In diesem Fall sollte eine entsprechende `UnsupportedOperation-Exception` geworfen werden:

```
public String myMethod() {
    throw new UnsupportedOperationException();
}
```

Sich auf getCause() verlassen

Dies führt zu Problemen bei gewrappten Exceptions (`getCause().getCause()` notwendig). Exceptions sollten zu einer eindeutigen Behandlung führen. Das folgende Code-Fragment unterscheidet die Fehlerbehandlung anhand des Grundes der gefangenen Exception:

```
catch (MyException e) {if (e.getCause() instanceof FooException) {
```

→ Dies funktioniert nur, sofern eine Exception nicht mehrmals gewrappt wurde. Es dürfen nur die für die Schnittstelle spezifizierten Exceptions behandelt werden. Ist auf der Aufruferseite eine Auswertung mittels `getCause()` notwendig, so ist die Schnittstelle zu überarbeiten. Der Grund hierfür ist die Anforderung des Aufrufers an die Schnittstelle, die Fehler genauer unterscheiden und unterschiedlich behandeln zu können.

Technische checked Exceptions zur Schnittstelle durchreichen

Technische checked Exceptions sind zu verwenden, um den Aufrufer zur Fehlerbehandlung zu zwingen. Der Aufrufer muss den Fehler behandeln und nicht in eine technische unchecked Exception wrappen. In Einzelfällen mag dies notwendig sein, muss dann aber mit dem Chefdesigner abgestimmt werden.

6. Quellenverzeichnis

[AnwendungskernDetailkonzept]

Detailkonzept der Komponente Anwendungskern
10_Blaupausen\technische_Architektur\Detailkonzept_Komponente_Anwendungskern.pdf
.

[LoggingKonzept]

Konzept Logging
20_Bausteine\Logging\Konzept_Logging.pdf.

[ÜberwachungKonfigKonzept]

Konzept Überwachung und Konfiguration
20_Bausteine\Ueberwachung_Konfiguration\Konzept_Ueberwachung-Konfiguration.pdf .

[VorlageAnwendung]

Beispielimplementierung „Vorlage-Anwendung“
Wird auf Anfrage bereitgestellt.

7. Abbildungsverzeichnis

8. Tabellenverzeichnis