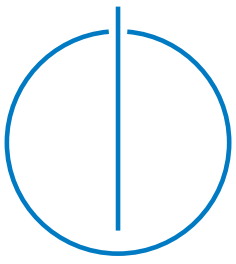
TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Bachelorarbeit
in
Wirtschaftsinformatik

Analyse der Verwendung des Eventlistener Konzepts einer objektrelationalen Persistenzschicht

Julian Lebherz



TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Bachelorarbeit
in
Wirtschaftsinformatik

Analyse der Verwendung des Eventlistener Konzepts einer objektrelationalen Persistenzschicht

Usage Analysis of the Event Listener Concept Provided by an Object-relational Persistence Layer

Bearbeiter:	Julian Lebherz
Betreuer:	Dr. Thomas Büchner
Aufgabensteller:	Prof. Dr. Florian Matthes
Abgabedatum:	15. Januar 2011

Ich versichere hiermit, dass ich diese Bachelorarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Datum	Unterschrift

Zusammenfassung

Das Konzept der Eventlistener findet mittlerweile in diversen objektrelationalen Persistenzschichten Anwendung. Hierbei machen sich diese Systeme den Grundgedanken der Eventlistener zu Nutze. Ursprünglich wurden Ereignisse aus der Nutzerinteraktion erfasst und automatisch entsprechend verarbeitet. In modernen, kollaborativen Portalsystemen mit Wikis und Blogs wird die Datenbasis von den Nutzern nicht nur selbst generiert, sondern auch strukturiert. In diesem Fall können Ereignisse, wie die Speicherung, Modifikation oder das Löschen von Datenobjekten, automatisch spezifischen Code ausführen, um dem Nutzer unliebsame, wiederkehrende Arbeit zu ersparen und das System in einem konsistenten Zustand zu erhalten.

Im Rahmen dieser Arbeit werden die Eventlistener-Konzepte in den Systemen Hibernate, Java Content Repository und Tricia analysiert. Der Fokus wird hierbei auf die quelloffene Plattform Tricia gelegt, in welcher das Eventlistener Konzept als ein essenziell wichtiger Baustein im gesamten System integriert ist. Die Implementierung dieses Konzeptes in Tricia wird detailliert beleuchtet und die verschiedenen Arten von Eventlistenern analysiert. Da in einem integrierten System wie Tricia außerordentlich viele Eventlistener ausgeführt werden, steigt die Komplexität deren Verflechtung parallel zum Umfang der Funktionalität des Systems. Die ausufernde Komplexität entwickelt sich zu einem Problem für die Entwickler selbst. Darum wird mit einem neu entwickelten, in die Eclipse IDE integrierten, Analysetool die chronologische Abfolge der Eventlistener protokolliert und übersichtlich dargestellt. Das Tool soll dabei den Entwicklungsprozess unterstützen und eine einfache Analyse der Abhängigkeiten zwischen den Eventlistenern ermöglichen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

Abkürzungsverzeichnis

ADO	ActiveX Data Objects
Ajax	Asynchronous JavaScript and XML
API	Application Programming Interface
ASP	Active Server Pages
CRX	Content Repository Extreme
CSS	Cascading Style Sheets
GUI	Graphical User Interface
HQL	Hibernate Query Language
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
JCR	Java Content Repository
JDBC	Java Database Connectivity
JPA	Java Persistence API
JSP	Java Server Pages
ORM	Object-Relational Mapping
RMI	Remote Method Invocation
SQL	Structured Query Language
SWT	Standard Widget Toolkit
UI	User Interface
UML	Unified Modeling Language
URL	Uniform Resource Locator
XML	Extensible Markup Language

1 Einleitung

Vor noch nicht allzu langer Zeit war die Aufgabenverteilung im Internet klar geregelt. Die Betreiber der Internetseiten waren für die Informationsbeschaffung, sowie deren Aufbereitung und Interpretation zuständig. Der normale Internetnutzer konsumierte diese Informationen. Diese Passivität war normal - im Radio, in der Zeitung und im Fernsehen. Die Verbreitung der Web 2.0 Technologien ermöglichte es jedoch, den Nutzer mehr und mehr aktiv an der Gestaltung der dargebotenen Inhalte zu beteiligen. Weltweit entwickelten sich zu jedem erdenklichen Thema Blogs, deren Autoren nicht nur den Inhalt, sondern auch dessen Repräsentation bestimmen.

Auf die Spitze getrieben wurde die Nutzerintegration schließlich von Portalen wie Wikipedia und Facebook. Hierbei stellen die Betreiber der Seiten lediglich die Infrastruktur und ein Softwareskelett bereit, welches von den Nutzern mit Daten und Informationen bestückt wird. Anfangs noch ein wenig belächelt, übertreffen diese Portale mittlerweile ihre passiven Vertreter hinsichtlich Datenvolumen und Nutzerzahlen um Längen. Interaktive Konzepte wie Wikis oder Blogs erhöhen jedoch die Komplexität der Serversysteme enorm. Die Nutzer erstellen hierbei nicht nur eigene Inhalte, sie bestimmen auch deren Strukturierung und Referenzierung untereinander. Um die Datenbasis trotz der kontinuierlichen, tiefgreifenden und teils strukturellen Veränderungen durch Nutzer konsistent zu halten, wurden diverse Methoden entwickelt. Einige beschränken den Nutzer mittels Rechtesystemen darauf, keine strukturellen Änderungen durchführen zu können. Der flexiblere Ansatz ist jedoch die automatisierte Behandlung solcher Änderung. Anpassungen, welche implementierungsspezifisch die logische Konsequenz einer initialen Operation darstellen, werden automatisiert durchgeführt.

Wenn beispielsweise innerhalb eines Wikis eine Seite von mehreren anderen Wikiseiten referenziert wird, so zieht eine Namensänderung der Wikiseite einiges nach sich. Diese Änderung muss in jeder einzelnen Verlinkung wiederholt werden, um das System nach der initialen Namensänderung wieder in einem konsistenten Zustand zu hinterlassen. All diese Änderungen selbst durchzuführen, ist dem Nutzer natürlich nicht zumutbar. Ein Ansatz zur aktiven Lösung des Problems sind Eventlistener. Sie werden von der initialen Veränderung ausgelöst und führen alle davon abhängigen Veränderungen automatisch aus. Der Nutzer selbst braucht sich deshalb nicht um Referenzen auf das veränderte Objekt oder seine Stellung in der Objekthierarchie zu kümmern.

1.1 Das Eventlistener Konzept

Das Konzept der Eventlistener beschreibt ganz grundsätzlich ein Entwurfsmuster zur Ereignisbehandlung in der objektorientierten Softwareentwicklung. Hierbei handelt es sich um eine Abwandlung des Observer Entwurfsmusters, welches die Gang of Four bereits 1994 dokumentiert hat [Gamma et al., 2007]. Beide Muster sind Verhaltensmuster, welche Veränderungen eines Objektes an davon abhängige Programmteile weiterleiten oder diese lediglich benachrichtigen. Im Folgenden wird die Funktionsweise des Observer Entwurfsmusters dargestellt und anschließend auf das Konzept der Eventlistener übertragen.

Das Observer Entwurfsmuster, auch Beobachter genannt, ist in Abbildung 1 dargestellt. Hierbei können sich Observer über die Methode register (Observer obs) an einem Subjekt anmelden und per unregister (Observer obs) wieder abmelden. Das Subjekt verwaltet eine Liste an registrierten Obervern und benachrichtigt diese via notifyObservers(), sodass auf ein Ereignis am Subjekt sofort reagiert werden kann. Die Klasse Observer dient hierbei lediglich als Schnittstelle. Konkrete Observer wie der ObserverX im Abbildung 1 erben von dieser Schnittstelle.

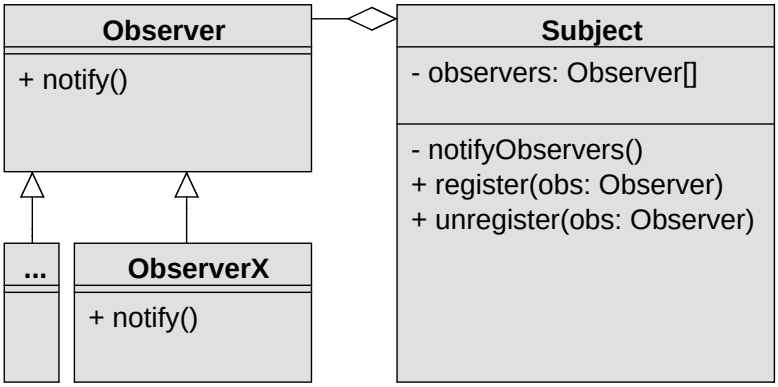


Abbildung 1: Das Observer Entwurfsmuster

Das Konzept der Eventlistener ähnelt dem Observer Entwurfsmuster stark, wobei Eventlistener die Observer substituieren können. Hierbei werden jedoch nicht alle Eventlistener gleichermaßen behandelt und vom Subjekt in einer einzelnen Liste verwaltet. Die Listener werden in diskreten Klassen definiert und auch als solche am Subjekt registriert. Jedes Subjekt verwaltet dadurch für jede Art von Ereignissen einen separaten Pool an aktiven Listenern. Ein Eventlistener registriert sich beim Subjekt also nur für eine bestimmte Art von Ereignissen.

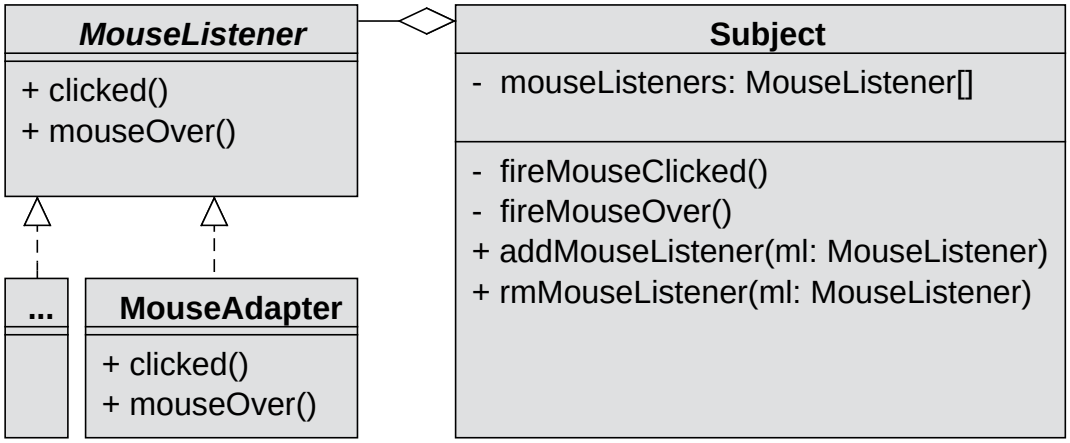


Abbildung 2: Das Eventlistener Entwurfsmuster

In Abbildung 2 kann sich der MouseListener per addMouseListener(MouseListener ml) als Eventlistener an einem Subjekt registrieren. Via rmMouseListener(MouseListener ml) wird der Listener wieder deaktiviert. Das Subjekt kann hierbei beispielsweise ein Applikationsfenster sein. Wenn nun der Nutzer innerhalb des Fensters auf etwas klickt, wird das Subjekt alle registrierten MouseListener benachrichtigen und damit ihre Methode clicked() ausführen.

Diese Aufgliederung erlaubt nun, verschiedene Eventlistener für verschiedene Ereignisse an ein und demselben Subjekt zu registrieren. Damit kann beispielsweise, wie in Abbildung 3 ersichtlich, zusätzlich ein KeyListener registriert werden. Eine solche Kategorisierung der Ereignisse lenkt den Ereignisfluss und reduziert somit die Komplexität der Ereignisbearbeitung. Das Eventlistener Konzept hat seinen Ursprung in der Behandlung von Nutzerinteraktionen, da hierbei durch viele verschiedene Ereignistypen ein echter Mehrwert gegenüber dem Observer Muster erzielt werden kann. Das Verhalten des Nutzers kann fast beliebig granular erfasst werden, weshalb in der UI-Programmierung unzählige Ereignisse denkbar sind.

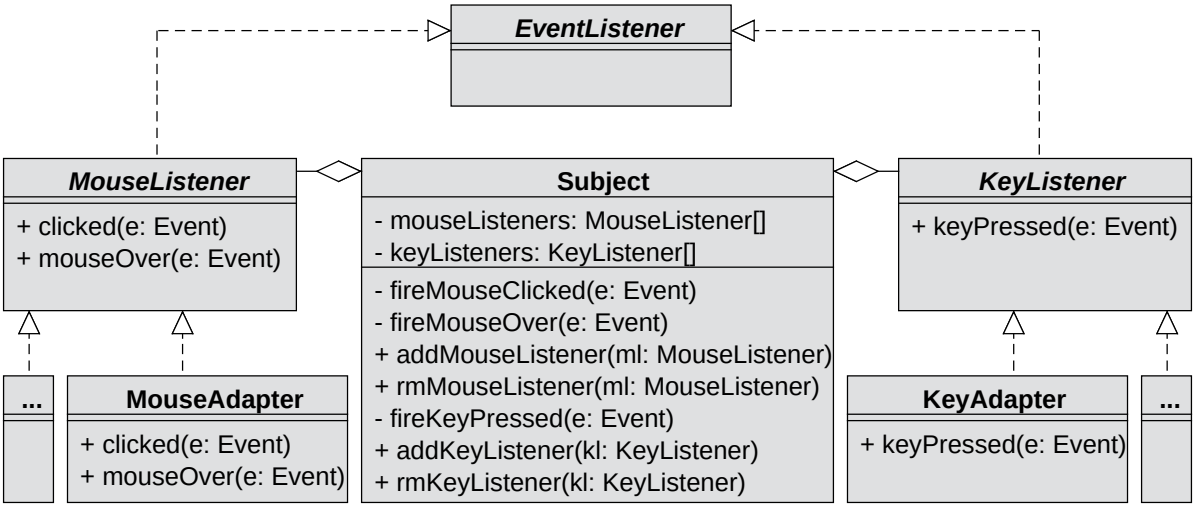


Abbildung 3: Ein erweitertes Eventlistener System

Wenn das Subjekt die registrierten Eventlistener triggert, wird zumeist ein Ereignisobjekt (Event) als Parameter mit übergeben. Dieses Objekt kapselt genauere Informationen über das Ereignis, beispielsweise die genaue Position der Maus beim Klick, die genaue Zeit des Klicks oder die Zeitspanne, in der die Maustaste gedrückt war.

Die Eventlistener Systeme wurden nach und nach auch auf Anwendungsgebiete übertragen, die mit Nutzerinteraktion nicht viel gemein haben. In verteilten Anwendungen ist es beispielsweise wichtig, über Änderungen und Aktualisierungen der Datenbasis informiert zu werden. Diese Funktionalität ist meist direkt in den objektrelationalen Persistenzschichten integriert. Im Folgenden wird zuerst die Begrifflichkeit „Objektrelationale Persistenzschicht“ genauer beleuchtet und anschließend in Kapitel 2.3.2 auf Systeme in der Praxis eingegangen.

1.2 Objektrelationale Persistenzschichten

Zur Persistierung von Daten werden unzählige Lösungen angeboten. Doch trotz interessanter Eigenschaften der Alternativen ist die relationale Datenbank vor objektorientierten, dokumentenorientierten oder XML-Datenbanken die bei weitem am häufigsten eingesetzte Variante. Etwa drei viertel der Erlöse in diesem Segment werden durch relationale Datenbanksysteme erwirtschaftet [Olofson & Shirer, 2010]. Da Anwendungssysteme jedoch zumeist objektorientiert entwickelt werden, muss zwischen System und Datenspeicher ein Paradigmenbruch überwunden werden.

Objekte in relationale Schemata zu überführen und zeitlich versetzt wieder zum gleichen Objekt zusammensetzen zu können, ist jedoch keine triviale Funktionalität. Dieser Vorgang wird durch das objektrelationale Mapping (ORM) durchgeführt. Es gibt diverse Ausführungen dieser objektrelationalen Adapterschicht, wobei teils die Logik in ein eigenes System verpackt wird (z.B. Hibernate). Teils ist sie auch als Komponente im Datenbanksystem verankert und gilt dann als Zugriffsschicht über der eigentlichen, relationalen Persistenzschicht.

Auf diese objektrelationalen Persistenzschichten wurde nun das Eventlistener Konzept übertragen. Dabei stellt sich zu aller erst einmal die Frage, welche Ereignisse in einer Persistenzschicht auftreten können. Objekte können gespeichert, gelöscht und modifiziert werden. Diese drei grundlegenden Ereignisse stellen die Basis für jedes Eventlistener Konzept in solch einer Persistenzschicht dar. Einige der spezifischen Implementierungen erlauben jedoch eine feinere Unterscheidung von Ereignissen. Darüber hinaus sind verschiedene Konzepte zur Registrierung der Listener (zentral vs. dezentral) anzutreffen. Im folgenden Kapitel wird anhand dreier Systeme die Funktionsweise der Eventlistener und die Unterschiede in deren Implementierungen diskutiert.

2 Eventlistener-Systeme in der Praxis

In der Praxis werden Eventlistener nicht nur in Java, C# und C++ eingesetzt, sondern vor allem auch in Skriptsprachen wie Javascript [Pixley, 2000] oder Actionscript [McCauley, 2008]. In diesen Fällen ist die Ereignisbehandlung von Nutzerinteraktionen wie zum Beispiel ein Mausklick oder eine Drag&Drop-Bewegung vorrangig. Adaptionen des Eventlistener Konzeptes zur Anwendung innerhalb bzw. an der Schnittstelle zu einer objektrelationalen Persistenzschicht hingegen finden sich für Java und C#/C++ basierte Lösungen (bespielsweise ADO.net [Sceppa, 2002]).

Für Java existieren diverse Frameworks, um Objekte in ein relationales Schema zu überführen und somit in üblichen relationalen Datenbanken abspeichern zu können. Der bekannteste Vertreter unter ihnen ist Hibernate [King et al., 2010]. Hibernate integriert ein umfassendes Ereignissystem und ist zusammen mit seiner Portierung NHibernate für .net Software das wohl am weitesten verbreitete Persistenzframework.

Einen alternativen Ansatz verfolgt das Java Content Repository. JCR spezifiziert lediglich eine Schnittstelle und nimmt Hersteller von Produkten zur Objektpersistierung in die Pflicht, diese JCR konform zu entwickeln [Nuescheler et al., 2005]. Vorgegeben ist hierbei ebenfalls ein Eventlistener System, welches via JCR API angesprochen werden kann. Diese Methodik erlaubt die Entkopplung von Applikationssoftware und eingesetzten Persistenzlösungen.

Das von der InfoAsset AG enwickelte Informationssystem Tricia hingegen implementiert ein Eventlistenersystem als zentrales Konzept im Umgang mit persistenten und transienten Daten [Büchner et al., 2010]. Im Folgenden werden die Ansätze von Hibernate, Java Content Repository sowie Tricia aufgegriffen und deren Eventlistener Systeme detailiert untersucht.

2.1 Hibernate

Hibernate ist ein quelloffenes Java Framework, dessen hauptsächliche Aufgabe das objektrelationale Mapping (ORM) darstellt. Hibernate ermöglicht es somit Objekte inklusive aller Attribute und Methoden in ein relationales Schema zu überführen und bei Bedarf wieder in die Laufzeitumgebung zu laden. Dabei werden die Java Datentypen automatisch in SQL Datentypen umgewandelt.[King et al., 2010]

Für den Anwendungsentwickler erfolgt die Persistierung relativ unabhängig von der eingesetzten Datenbank, da Anfragen zum Laden und Speichern von Objekten via SQL, der SQL-ähnlichen Hibernate Query Language (HQL), aber auch objektorientiert über die Hibernate Criteria-API gestellt werden können. Dabei unterstützt Hibernate nahezu alle aktuellen Datenbanktypen [Patricio, 2010] und wird mittlerweile täglich tausendfach heruntergeladen [Slavic, 2010].

Hibernate nimmt quasi eine Vermittlerrolle zwischen der Applikationsschicht und der Persistenzschicht ein. Der Anwendungsentwickler muss nicht zwischen dem objektorientierten Paradigma und der relationalen Datenrepräsentation innerhalb der Datenbanken

hin- und herwechseln. Er kann Objekte zur Speicherung an Hibernate weitergeben oder Hibernate zum Laden von Objekten aus der Datenbank auffordern. In beiden Fällen wird lediglich mit den Objekten selbst gearbeitet, es ist also keinerlei Kenntnis über den eigentlichen ORM Vorgang erforderlich. Hierdurch kann die Komplexität von datenbankbasierten, objektorientierten Applikationsentwicklungen enorm verringert werden. [Bauer & King, 2004, S.7f]

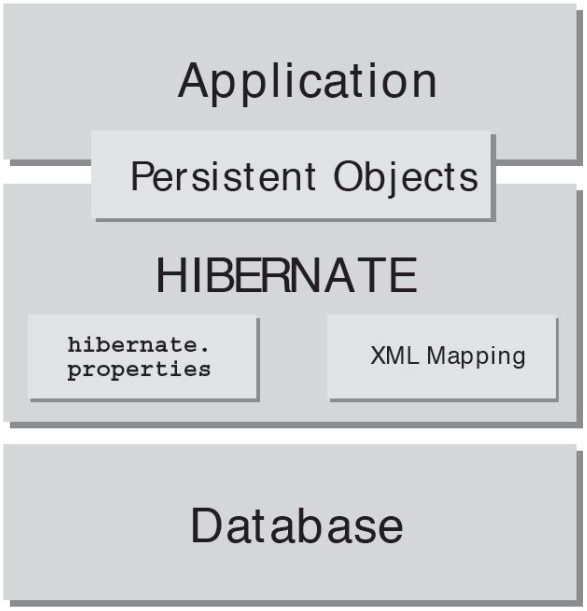


Abbildung 4: Hibernate an der Schnittstelle zwischen Applikation und Datenbank [King et al., 2010, S.27]

Abbildung 4 zeigt Hibernate als die Schnittstelle zwischen der Applikation (Application) und der Datenbank (Database). Dies verdeutlicht erneut, dass Hibernate eine Abstraktion von der verwendeten Datenbank ermöglicht und der Paradigmenbruch nun innerhalb von Hibernate standardisiert und strukturiert überbrückt wird. Die Datei hibernate.properties definiert Einstellungen in der Hibernateschicht und wird beispielsweise an die verwendete Datenbank angepasst. Im XML Mapping ist die Logik zur Überführung von Objekten und deren Assoziationen in ein relationales Schema definiert.[King et al., 2010, S.27f]

Hibernate bietet gleich zwei Eventlistener-Systeme, das „Interceptors System“ (ein Abfängersystem) und das „Eventsystem“. Beide können unabhängig und damit auch parallel eingesetzt werden, wobei teils die gleiche Funktionalität mit beiden Systemen bereitgestellt werden kann.[King et al., 2010, S.257-259]

2.1.1 Interceptor System

Das System der Abfänger bzw. Interceptors erlaubt die Inspektion und Manipulation von persistenten Objekten direkt bevor sie gespeichert, aktualisiert, gelöscht oder geladen werden. Dazu werden Interceptor Klassen geschrieben, die entweder das Interface org.hibernate.Interceptor direkt implementieren oder aber von der Klasse org.hibernate

.EmptyInterceptor erben. Die zweite Möglichkeit verringert den Entwicklungsaufwand erheblich, da hierbei nicht alle Methoden überschrieben werden müssen. Das Interceptor System kann unter anderem zur Behandlung der in Tabelle 1 angeführten Ereignisse eingesetzt werden.[King, 2010]

Transaktion	Nach Beginn der Transaktion (afterTransactionBegin()), als auch vor und nach Beendigung der Transaktion (beforeTransactionCompletion(); afterTransactionCompletion()); Eine Transaktion bezeichnet einen Datenaustausch zwischen Hibernate und der Datenbank.
Speichern	Bevor ein Objekt gespeichert wird (onSave())
Laden	Bevor ein Objekt geladen wird (onLoad())
Aktualisieren	Vor (preFlush()) und nach (postFlush()) einer Aktualisierung
Löschen	Bevor ein Objekt gelöscht wird (onDelete())

Tabelle 1: Ereignisse und zugehörige Methoden der Hibernate Interceptors [King, 2010]

Es existieren zwei verschiedene Arten von Interceptors. Sie können zum einen in einer Session definiert werden und sind in diesem Fall auch ausschließlich für diese Session gültig. Für eine zweite Session wird der Interceptor nicht automatisch geladen. Dabei stellt eine Session die Schnittstelle zwischen der Java Applikation und Hibernate dar. Sie wird vor einer Transaktion geöffnet und danach wieder geschlossen.[King et al., 2010, S.258f]

```
Session session = sessionFactory.openSession( new XyInterceptor());
```

Die zweite Möglichkeit besteht darin, einen Interceptor für eine Configuration und damit indirekt für eine SessionFactory zu definieren. Der Interceptor wird in alle Sessions, die von dieser SessionFactory aus initiiert werden, eingebunden.[King et al., 2010, S.258f]

```
new Configuration().setInterceptor( new XyInterceptor());
```

In diesem Fall muss jedoch darauf geachtet werden, dass der Interceptor Threadsicherheit gewährleistet, da sich mehrere Ereignisse und deren Behandlung innerhalb des Interceptors überschneiden können.[King et al., 2010, S.259]

Es können somit diverse Ereignisse abgefangen und spezifischer Code ausgeführt werden. Dafür müssen Interceptors zentral an der Session bzw. an der Configuration registriert werden. Jedoch existiert keine Möglichkeit die Ereignisse strukturiert zu filtern. Ein onDelete Event wird ohne Beachtung des Eventtyps oder des eigentlichen Quellobjektes an alle Interceptors weitergeleitet. Die betroffene Methode wird auch bei allen Interceptors ausgeführt. Ob diese jedoch auf das Ereignis reagieren, hängt von ihrer spezifischen Implementierung ab. Hierbei wird bereits ersichtlich, dass viele Ereignisse unnötig weitergeleitet werden, da bei weitem nicht alle Interceptors auf alle Events reagieren. Deshalb eignet sich das Interceptor System vor allem für Aufgaben, die nicht spezifisch für einzelne Objekttypen ausgeführt werden sollen (z.B. Protokollierung der Zugriffe auf Objekte/Ressourcen zur Erstellung von Nutzungsstatistiken).

2.1.2 Eventsystem

Das Eventsystem kann nun zusätzlich oder als Ersatz für die Verwendung von Interceptors eingesetzt werden. Jede Aktion die innerhalb einer Session ausgeführt werden kann, generiert ein spezifisches Ereignis, welches durch eigene Eventlistener behandelt werden kann. Zu jeder Art von Ereignis wiederum existiert sowohl ein Interface, als auch eine Default-Implementierung. Eventlistener können das Interface implementieren, sollten aber besser von der Default-Implementierung erben, um den Eventlistener so einfach wie möglich zu halten. Diese Aufsplittung in eventspezifische Listener stellt einen großen Unterschied zu den Interceptors dar. In Tabelle 2 werden einige der möglichen Ereignisse aufgelistet. Im Vergleich zu den Methoden eines Interceptors sind die Arten der Eventlistener im Eventsystem noch feiner aufgegliedert.

Speichern	Beim Speichern eines Objektes (PersistEventListener)
Laden	Vor und nach dem Laden eines Objektes oder zeitpunktunabhängig (PreLoadEventListener / PostLoadEventListener / LoadEventListener)
Aktualisieren	Vor und nach einer Aktualisierung (PreUpdateEventListener / PostUpdateEventListener)
Synchronisieren	Beim Synchronisieren des Objektzustandes im Speicher mit der Datenbank (FlushEventListener / FlushEntityEventListener)
Sperren	Beim Sperren eines Objektes (LockEventListener)
Löschen	Vor und nach dem Löschen eines Objektes oder unabhängig (PreDeleteEventListener / PostDeleteEventListener / DeleteEventListener)
...	...

Tabelle 2: Ereignisse und Listenertypen im Hibernate Eventsystem [King, 2010]

Als Beispiel wird in Quelltext 1 ein LoadEventListener vorgestellt. MyLoadListener implementiert die LoadEventListener Schnittstelle und kann somit auf LoadEvents reagieren. Der Eventlistener MyLoadListener prüft hierbei, ob die Ladeoperation authorisiert ist oder nicht.

```
public class MyLoadListener implements LoadEventListener {

  public void onLoad(LoadEvent event, LoadEventListener.LoadType loadType)
      throws HibernateException {
    if ( !MySecurity.isAuthorized( event.getEntityClassName(), event.
        getEntityId() ) ) {
      throw MySecurityException("Unauthorized access");
    }
  }

}
```

Quelltext 1: Beispiel für einen LoadEventListener [King et al., 2010, S.259]

Damit der MyLoadListener im Falle von LoadEvents auch wirklich benachrichtigt wird, muss er registriert werden. Dies kann im Eventsystem sowohl programmatisch, als auch deklarativ geschehen. Quelltext 2 zeigt die programmatische Variante, in der sowohl der MyLoadListener, als auch ein DefaultLoadListener an der Configuration registriert werden.

```
Configuration cfg = new Configuration();
LoadEventListener[] stack = { new MyLoadListener(), new
    DefaultLoadEventListener() };
cfg.EventListeners().setLoadEventListeners(stack);
```

Quelltext 2: Programmatische Registrierung von Eventlistenern [King et al., 2010, S.260]

Um Eventlistener deklarativ zu registrieren, wird für sie ein Konfigurationseintrag in der XML Hibernate Konfiguration erstellt. Diese XML Datei ist in Quelltext 3 ausschnittsweise dargestellt. Sie enthält innerhalb der SessionFactory ein Event-Objekt mit dem tag type="load". Dieses Objekt enthält zwei Klassenpfade, unter denen die Eventlistener-Klassen zu finden sind. Natürlich können parallel weitere Event-Typen mit zusätzlichen Eventlistenern definiert sein.

```
<hibernate-configuration>
  <session-factory>
    ...
    <event type="load">
      <listener class="com.eg.MyLoadListener"/>
      <listener class="org.hibernate.event.def.DefaultLoadEventListener"/>
       ...
    </event>
    ...
  </session-factory>
</hibernate-configuration>
```

Quelltext 3: Deklarative Registrierung von Eventlistenern [King et al., 2010, S.260]

Das Eventsystem von Hibernate erlaubt also durch ausdifferenzierte Arten von Eventlistenern, Ereignisse nur an diejenigen Listener weiterzuleiten, welche sich auch für die spezifischen Events registriert haben. Dies reduziert die Komplexität der Ereignisabwicklung im Vergleich zum Konzept der Interceptors enorm. Allerdings werden die Eventlistener wie auch die Interceptors an einer zentralen Instanz registriert. Somit ist es nicht direkt möglich, dass ein Listener ausschließlich Ereignisse von bestimmten (persistenten) Objekten übermittelt bekommt. Die Selektion der relevanten Ereignisse muss deshalb innerhalb der Listenerimplementierung erfolgen, was bei komplexeren Ereignissystemen nicht mehr praktikabel ist.

2.2 Java Content Repository

Das Java Content Repository (JCR) spezifiziert eine API um auf Inhalte aus verschiedenartigen Datenquellen einheitlich zugreifen zu können. JCR ermöglicht somit ein komfortables Arbeiten mit Persistenzschichten. Hierzu werden nicht für jede einzelne Persis-

tenzlösung Adapter geschaffen, vielmehr nimmt JCR die Hersteller von Content Repositories in die Pflicht, ihre Produkte JCR konform zu entwickeln. Deshalb können mittels JCR Datenquellen ausgetauscht oder mehrere parallel angesprochen werden, ohne die Applikation strukturell zu verändern. Die JCR Spezifikation wurde so definiert, dass sowohl hierarchisch organisierte, als auch nicht-hierarchisch organisierte Datenquellen (beispielsweise relationale Datenbanken) genutzt werden können. Dies erhöht die Flexibilität von JCR enorm. Beispiele für Java Content Repository konforme Lösungen sind:

- Jackrabbit JCR - Referenzimplementierung (Apache Software Foundation)
- Content Repository Extreme - CRX (Day Software AG)
- Workplace Web Content Management (IBM)
- SAPERION Content Repository (SAPERION)
- eXo Platform (ObjectWeb)
- Beehive (Oracle)
- WebCenter Suite (Oracle)

Das grundlegende Architekturkonzept hinter JCR teilt das Repository in verschiedene Workspaces auf. Ein Repository bezeichnet eine einzelne Datenquelle (z.B. eine relationale Datenbank). Abbildung 5 zeigt diese Struktur. Jedem Workspace ist eine Baum von Objekten zugeordnet, der jeweils über den Wurzelknoten (root) verfügbar gemacht wird.

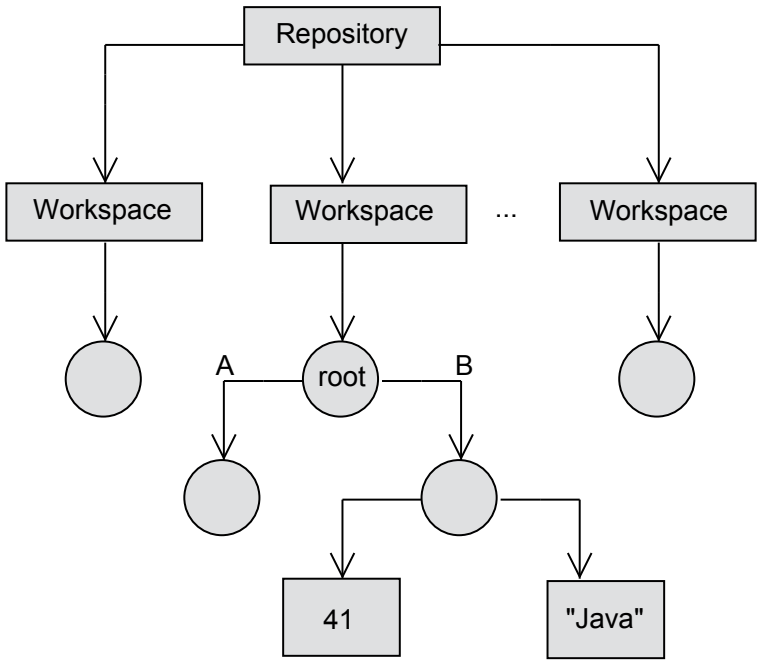


Abbildung 5: Repository Modell mit mehreren Workspaces; in Anlehnung an [Barik, 2006]

Jeder Wurzelknoten wiederum spannt einen eigenen Baum aus Knoten (Node) und Blättern (Properties) auf. Bei diesem Datenmodell handelt es sich um das Kompositum Entwurfsmuster, welches Objekte in Baumstrukturen überführt und somit deren hierarchische Beziehungen abbildet [Gamma et al., 2007, S.163-173]. Das Entwurfsmuster ist in Abbildung 6 dargestellt. Im Kompositum existiert ein abstrakter Datentyp Item (auch Komponente genannt). Dieser Datentyp kann sowohl einen Knoten, als auch ein Blatt darstellen.

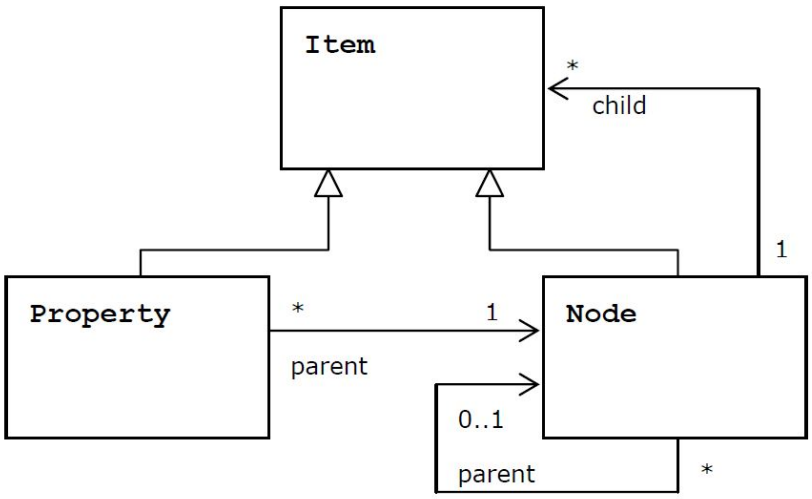


Abbildung 6: JCR Datenmodell innerhalb eines Workspaces [Nuescheler et al., 2005, S.22]

Wenn eine Komponente ein Blatt repräsentiert, enthält sie zwangsläufig einen Verweis auf den Ordner, der die Komponente direkt enthält. Beispiele für Blätter sind einzelne Zahlenwerte, Zeichenketten, aber auch Binärdateien wie Bilder oder Filme. Abbildung 5 zeigt Blätter mit dem numerischen Wert 41 und der Zeichenkette „Java“. Sie gehören zu Knoten B, der selbst im Wurzelknoten root enthalten ist.

Fungiert eine Komponente als Knoten, so hat sie ebenfalls einen Verweis auf ihren direkten Vaterknoten. Zusätzlich allerdings enthält jeder Knoten auch eine Liste an Verweisen auf Komponenten. Diese zeigen auf Objekte, die dem Knoten hierarchisch untergeordnet sind. Das Beispiel in Abbildung 5 zeigt den Wurzelknoten root des mittleren Workspaces mit Verweisen auf die Komponenten A und B.

2.2.1 Observation

Das in JCR integrierte Eventlistener System heißt Observation (dt. Überwachung). Dieses System ist zwar innerhalb der JCR Spezifikation definiert, die Funktionalität ist allerdings für JCR konforme Datenquellen optional [Nuescheler et al., 2005, S.269]. Es warten jedoch fast alle Lösungen mit dem Eventlistener System auf. Ob dies der Fall ist, kann zur Erhaltung der Flexibilität programmatisch geprüft werden.

Innerhalb des Java Content Repository Eventlistener Systems werden Ereignisse vom Typ javax.jcr.observation.Event generiert. Von jedem Event kann der Pfad im Datenmodell

über getPath(), sowie der Nutzer der das Ereignis ausgelöst hat per getUserID() ermittelt werden [Nuescheler et al., 2005, S.270f]. In Tabelle 3 sind die einzelnen Ereignistypen aufgelistet. Der Methodenaufruf getType() liefert diese Ereignistypen als Rückgabewert. Seit Version 2.0 existieren zusätzlich die Methoden getIdentifier () und getInfo(). Sie liefern einen eindeutigen Identifikator der Komponente respektive die Parameter der Methode, die den Event ausgelöst hat. Darüber hinaus wird ab 2.0 jedem Ereignis ein Zeitstempel zugewiesen, der via getDate() ausgelesen werden kann.

NODE_ADDED	Ein Knoten wurde hinzugefügt
NODE_REMOVED	Ein Knoten wurde aus dem Baum gelöscht
PROPERTY_ADDED	Ein Blatt wurde hinzugefügt
PROPERTY_REMOVED	Ein Blatt wurde aus dem Baum gelöscht
PROPERTY_CHANGED	Ein Blatt wurde verändert

Tabelle 3: Ereignisse im JCR Eventsystem [Nuescheler et al., 2005, S.270f]

Eventlistener werden pro Workspace registriert, gelten also durch ihre Registrierung nicht für andere Workspaces im selben Repository. In JCR sind sie vom Typ javax.jcr.observation.EventListener. Wenn ein registrierter Eventlistener ein Ereignis empfängt, so wird die Methode onEvent(EventIterator event) aufgerufen. Der EventIterator stellt hierbei ein Sammelobjekt für potenziell mehrere Ereignisse dar. Dies kann zum Beispiel durch das Löschen eines Knotens geschehen. Es wird sowohl ein NODE_REMOVED für den Knoten selbst, als auch NODE_REMOVED- und PROPERTY_REMOVED-Ereignisse für dessen abhängige, untergeordnete Komponenten generiert.

Da die Ereignisse nicht von JCR selbst erzeugt werden, können diese auch erst bei Operationen generiert werden, welche mit der darunterliegenden Datenspeicherschicht kommunizieren. Dies stellt einen essentiellen Unterschied zu den Interceptors und dem Eventsystem von Hibernate dar. In Hibernate werden verschiedene Ereignisse vor der eigentlichen Transaktion mit der Speicherschicht generiert. Als Beispiele seien hierfür preFlush() bei den Interceptors und PreDeleteEventListener im Eventsystem genannt.

Um einen Eventlistener zu registrieren muss zunächst der sogenannte Observation Manager ausgewählt werden. Jeder Workspace hat genau einen Observation Manager vom Typ javax.jcr.observation.ObservationManager. An diesem Observation Manager können die Eventlistener zentral registriert werden. Quelltext 4 zeigt die für die Registrierung zuständige Methode.

```
addEventlistener (EventListener listener,
                  int eventTypes,
                  String absPath,
                  boolean isDeep,
                  String[] uuid,
                  String[] nodeTypeName,
                  boolean noLocal)
```

Quelltext 4: Registrierung von Eventlistenern [Nuescheler et al., 2005, S.272]

Bei der Registrierung von Eventlistener müssen also diverse Parameter definiert werden. Diese können dazu genutzt werden einzelne Ereignistypen auszuwählen, für die der Eventlistener registriert werden soll. Zusätzlich können jedoch auch die Komponenten beschränkt werden, deren Ereignisse an den Eventlistener weitergeleitet werden. Dies ist bei Hibernate so nicht möglich bzw. muss in der Implementierung des Eventlisteners behandelt (gefiltert) werden. Um die einzelnen Filtermöglichkeiten genauer zu beleuchten, werden diese in Tabelle 4 einzeln erläutert. Alle Filtermechanismen können kombiniert angewandt werden. Ein Ereignis muss also allen Filtern gerecht werden, um zum Eventlistener durchgereicht zu werden.

eventTypes	Hier werden diejenigen Ereignistypen angegeben, für welche sich der Eventlistener registrieren soll. Eine Mehrfachauswahl ist durch Bitmasken möglich.
absPath	Es werden lediglich Ereignisse an den Eventlistener weitergeleitet, deren zugeordneter Vaterknoten am angegebenen Pfad ist.
isDeep	Wenn **true** mitgegeben wird, so darf der Pfad des Vaterknotens auch in einem Teilbaum des absPath liegen.
uuid	Nur Ereignisse, deren zugeordneter Vaterknoten einem Identifikator aus der uuid-Liste enspricht, werden übertragen.
nodeTypeName	Nur Ereignisse, deren zugeordneter Vaterknoten einem Knotentyp aus der nodeTypeName-Liste entspricht, werden übertragen.
noLocal	Wenn **true** mitgegeben wird, so werden Ereignisse aus der eigenen Session ignoriert.

Tabelle 4: Filtermöglichkeiten per Parameter [Nuescheler et al., 2005, S.272f]

Diese kombinierten Filtermöglichkeiten sowohl auf Basis der Ereignistypen, als auch auf Basis der Komponenten stellen gegenüber Hibernate einen enormen Zugewinn an Flexibilität dar. Zusätzlich fällt positiv auf, dass sich die Komponenten hierarchisch nach dem Pfad der Vaterknoten (für hierarchisch organisierte Speicherschichten) und auch nach Identifikatoren (z.B. für relationale oder objektorientierte Datenbanken) filtern lassen. Es werden darum potenziell weniger Ereignisse verschickt, die unbehandelt bleiben. Fraglich ist auch ob die große Anzahl an Ereignistypen in Hibernate gegenüber JCR von Nutzen ist.

2.2.2 Journaled Observation

Seit JCR 2.0 existiert zusätzlich zu den Eventlistenern die Möglichkeit der Journaled Observation. Hierbei geht es grundsätzlich um eine Funktion zur Protokollierung aller aufgetretenen Ereignisse. Über ObservationManager.getEventJournal gelangt man an dieses Protokoll. Die Methode kann alternativ auch mit allen Filtermöglichkeiten aus Tabelle 4 ausgeführt werden. In diesem Fall enthält das Journal entsprechend nur die ausgefilterten Ereignisse. Erwähnenswert ist auch die Funktion EventJournal.skipTo(Calendar date), welche

das Journal quasi vorspult, um einen komfortablen Umgang zu ermöglichen. Zusätzlich wurde ein neuer Ereignistyp eingeführt. Ein PERSIST-Ereignis fasst Ereignisse zusammen, die innerhalb einer persistenten Veränderung des Workspaces enstanden sind. Da solch ein Journal recht schnell wachsen kann, ist auch die Möglichkeit gegeben, es hinsichtlich Zeitspanne oder Speicherbedarf zu beschränken. Die Journaled Observation ist eine sehr interessante Funktion, die vor allem eine nachträgliche Analyse des Systembetriebs ermöglicht. [Nuescheler et al., 2009]

2.3 Tricia

Tricia ist eine quelloffene Plattform zur Entwicklung von dynamischen Webanwendungen und betrieblichen Informationssystemen. Derartige Entwicklungen sind meist mit einem komplexen Geflecht aus verschiedensten Technologien verbunden. Während die Applikationslogik im Webserver steckt, muss beispielsweise per Hibernate, JCR oder JPA der Paradigmenwechsel zur Persistenzschicht überbrückt werden. Um die Inhalte dynamisch für den Benutzer aufzubereiten, werden wiederum Technologien wie Servlets, Java Server Pages (JSP) oder Active Server Pages (ASP) eingesetzt. Dies wird durch den Paradigmenbruch zwischen Anwendungslogik und der Präsentation (HTML und XML) notwendig. Tricia bildet den gesamten Informationsfluss von der Datenbank bis zur Webschnittstelle in einer Java Lösung ab. Dabei basiert Tricia auf dem quelloffenen, leichtgewichtigen Webserver Jetty, der wiederum selbst in Java geschrieben wurde. Die Verbindung zur Datenbasis wird per JDBC hergestellt, wobei ein Tricia-spezifisches objektrelationales Mapping eingesetzt wird. Die Generierung von Webseiten übernimmt ein eigens implementiertes Templatesystem, welches durch den Einsatz von Ajax dynamische Seiten erstellen kann.[Büchner et al., 2010]

Tricia ist ein nahezu vollständiges Informationssystem. Es benötigt ausschließlich Java und für viele Datenbanken sind passende JDBC Konnektoren verfügbar. Es können beispielsweise MySQL, Oracle, Microsoft SQL oder IBM DB2 Datenbanken genutzt werden. Um Tricia flexibel erweitern und anpassen zu können, ist die Plattform mit einer modularen Architektur ausgestattet. Die Kernfunktionalität ist in einem Paket namens toro implementiert. Zusätzlich exisitieren diverse, teilweise aufeinander aufbauende Plugins, welche die Funktionalität von toro erweitern. In Tabelle 5 sind einige dieser Plugins beschrieben, wobei zwischen Code- und Styleplugins unterschieden wird.[InfoAsset, 2010]

file	ist ein Code-Plugin, welches eine hierarchische Datenstruktur realisiert
wiki	ist ein Code-Plugin, welches die Funktionalität der wikis integriert
blog	ist ein Code-Plugin, welches blogging ermöglicht
jakob	ist ein Style-Plugin, welches HTML Templates und CSS Dateien bereitstellt

Tabelle 5: Plugins der Tricia Platform [Büchner et al., 2010, S.3][InfoAsset, 2010]

2.3.1 Datenmodell

Datenobjekte werden in Tricia generell als Assets behandelt. Assets haben gewisse Kenndaten die als Features bezeichnet werden. Diese Features können entweder als Roles Beziehungen definieren oder als Properties Attributwerte speichern. Abbildung 7 zeigt dieses Datenmodell. Jeder persistente Asset-Typ ist darüber hinaus einem Schema zugeordnet, welches ein spezifisches objektrelationales Mapping ermöglicht.

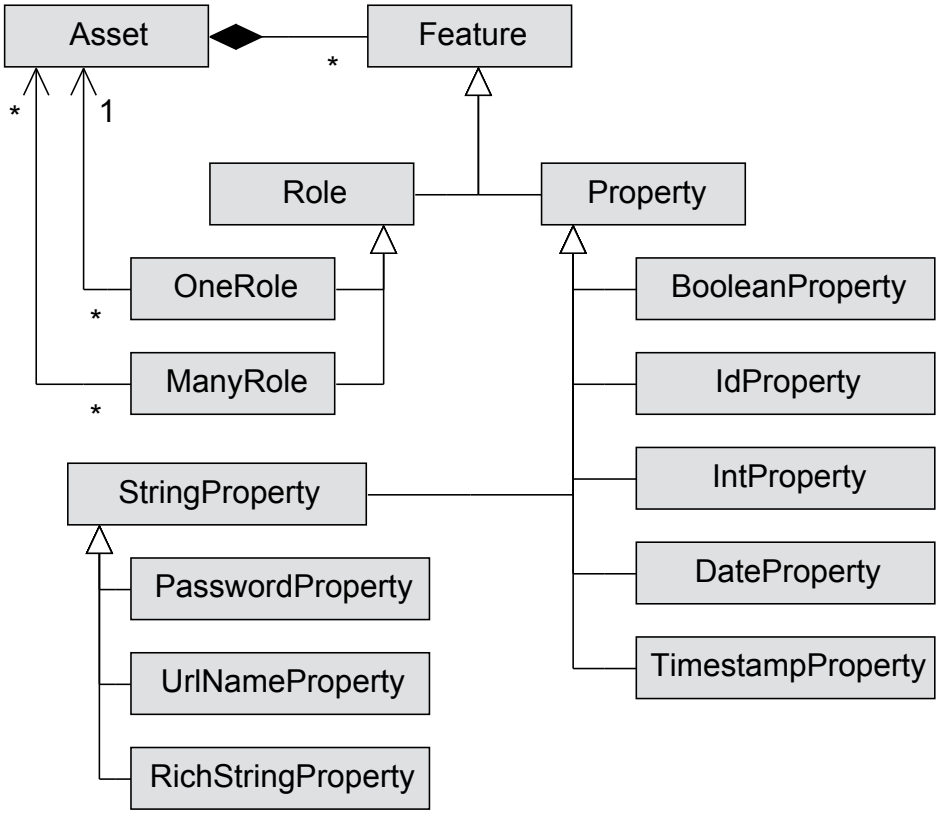


Abbildung 7: Ein Auszug aus dem Datenmodell der Tricia Plattform; in Anlehnung an [InfoAsset, 2010]

Um das Datenmodell näher zu erläutern, ist es hilfreich ein Beispiel aus dem wiki Plugin heranzuziehen. In Abbildung 8 stellen die Klassen Wiki und auch WikiPage Assets dar. Die einzelnen Attribute der Klassen sind als Features, genauer Properties, definiert. Das Attribut name beispielsweise ist eine StringProperty, der urlName ist eine UrlNameProperty und der content der WikiPage ist eine RichStringProperty.

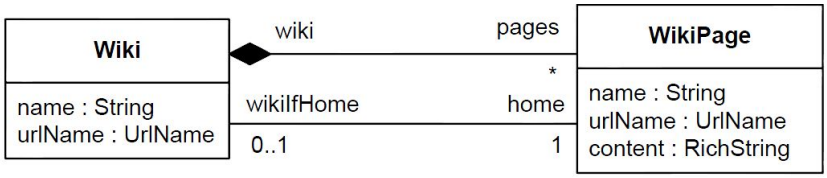


Abbildung 8: Wiki-Beispiel zum Tricia Datenmodell [Büchner et al., 2010, S.5]

Die Assoziationen in Abbildung 8 sind in Tricia ebenfalls als Feature modelliert. Diese entsprechen jedoch der Unterklasse Role. Ein Wiki kann beliebig viele WikiPages beinhalten. Deshalb ist diese Aggregation eine ManyRole. Umgedreht gehört jede WikiPage genau zu einem Wiki, was einer OneRole entspricht. Die Aggregation wird also in zwei einzelne Roles umgesetzt. Jedes Wiki definiert darüber hinaus eine WikiPage als seine Startseite. Diese Assoziation ist in beide Richtungen durch eine OneRole definiert.

Wie aus Abbildung 7 hervorgeht, gibt es noch einige andere Property-Typen. Die meisten sind durch ihren Namen selbsterklärend. Die RichStringProperty ist jedoch besonders interessant. Hierbei handelt es sich um einen String der durch HTML Tags und Attribute bereichert werden kann. Der Inhalt einer WikiPage beispielsweise ist als RichStringProperty deklariert.

2.3.2 Changelistener

Tricia bietet ein Eventlistener System, dass sich deutlich von den Systemen in Hibernate und JCR absetzt. Eventlistener werden in Tricia ChangeListener genannt, da sie immer dann benachrichtigt werden, wenn sich ein Datenobjekt ändert.

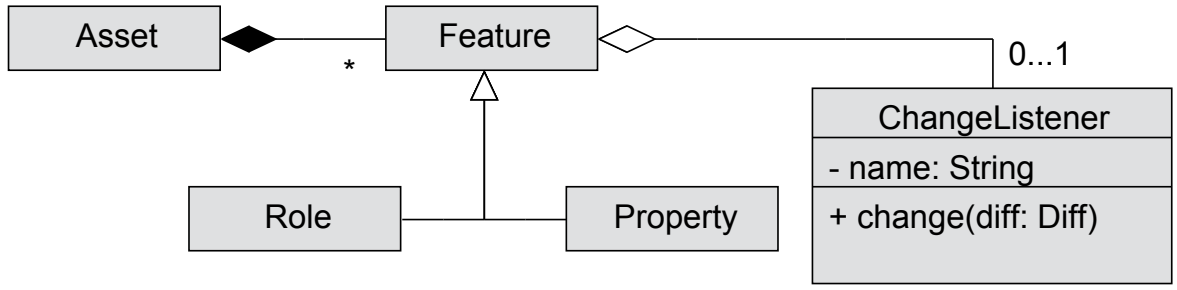


Abbildung 9: ChangeListener im Datenmodell; in Anlehnung an [InfoAsset, 2010]

Abbildung 9 zeigt die Anbindung der ChangeListener an den Kern des Tricia Datenmodells. Jeder Changelistener hat einen Namen und implementiert die Methode change(Diff diff), welche bei der Aktivierung des ChangeListeners getriggert wird. Der Parameter diff gibt dabei den Anfangszustand und den Endzustand des veränderten Features an. Das Interessante hierbei ist, dass die ChangeListener direkt an einem Feature und somit indirekt an einem Asset definiert werden. Sie haben also eine direkte Referenz auf ein bestimmtes Feature und werden auch nur bei dessen Veränderung informiert. Dieser implizite Filter macht eine komplexe Registrierung mit vielen Filterparametern wie beim Java Content Repository überflüssig. In Tricia entfällt eine zentrale Registrierung durch diesen Aufbau völlig.

In Tricia gibt es verschiedene Arten von ChangeListenern. Wie aus Abbildung 10 hervorgeht, gliedern sie sich ganz grundsätzlich in InstantChangeListener und DeferredChangeListener, korrespondierend zu den zwei Arten von Ereignissen, die in Tricia ChangeListener triggern können. Wenn sich ein Feature ändert, so werden sofort alle InstantChangeListener des Features angestoßen. Dieses Ereignis (FeatureChanged) wird auch ohne Persistenzoperation generiert. Bei der Speicherung eines Assets, werden alle DeferredChangeListener getriggert,

welche über ein Feature indirekt mit dem Asset verbunden sind. Die beiden ChangeListenern -Arten sind in Tricia noch feiner unterteilt. Zu einer genauen Analyse der verschiedenen ChangeListener sei auf Kapitel 3 verwiesen.

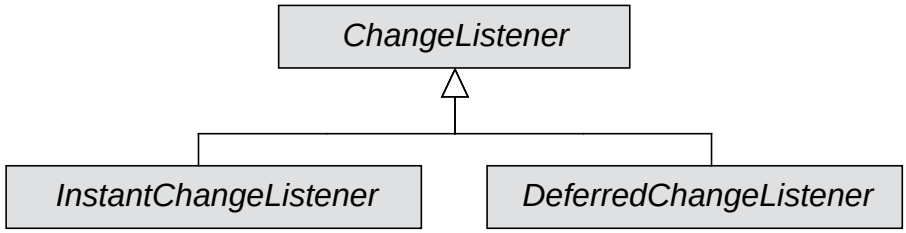


Abbildung 10: Arten von ChangeListener; in Anlehnung an [InfoAsset, 2010]

Um ChangeListener an einem Feature zu registrieren, werden sie in Tricia ganz einfach in eben diesem Feature definiert. Dies geschieht durch anonyme Klassen und ist beispielhaft in Quelltext 5 dargestellt. Das Codebeispiel zeigt eine Klasse WikiPage, welche das Feature name beinhaltet. Dabei handelt es sich um ein StringProperty-Objekt, welches als anonyme Klasse innerhalb von WikiPage definiert ist. Innerhalb dieses Features wird nun ebenfalls per anonymer Klasse ein ChangeListener (in diesem Fall ein InstantChangeListener) spezifiziert. Der Kern liegt hierbei im Überschreiben der change(Diff diff)-Methode des ChangeListeners. Diese wird sofort ausgeführt, sobald sich das Feature name ändert.

```
public class WikiPage {
    ...
    public final StringProperty name = new StringProperty() {
        final ChangeListener updateUrlName = new InstantChangeListener() {
            @Override
            public void change(Diff diff) {
                doSomething();
            }
        };
                                ...
    };
}
```

Quelltext 5: Registrierung von Eventlistenern [InfoAsset, 2010]

Das Eventlistener System in Tricia unterscheidet sich somit ganz grundlegend von den Konzepten in Hibernate oder JCR. Anstatt bei einer zentralen Instanz werden in Tricia die Eventlistener direkt an den Datenobjekten registriert. Deren Veränderung löst schließlich ein Ereignis aus und triggert die zugehörigen Eventlistener. Es findet also eine Art implizite Filterung statt. Darüber hinaus existiert in Tricia ein zusätzlicher Eventlistener-Typ. InstantChangeListener erlauben es, das Konzept auf nichtpersistente Operationen auszudehnen. Somit ermöglicht Tricia einen sehr flexiblen Umgang mit Eventlistenern.

3 Eventlistener in Tricia

Das Konzept der Eventlistener bzw. ChangeListener in Tricia unterscheidet sich maßgeblich von den verfügbaren Implementierungen in Hibernate, JCR oder ähnlichen Systemen. InstantChangeListener beispielsweise werden innerhalb von nicht persistenten Vorgängen angestoßen und können deshalb lediglich in einem solch integrierten System realisiert werden. Darüber hinaus ist auch die implizite Registrierung der Eventlistener direkt im Code des relevanten Subjektes in der Form ein Alleinstellungsmerkmal. Im Folgenden wird nun speziell auf das Eventlistener System in Tricia eingegangen und die einzelnen Arten der ChangeListenern analysiert. Wie in Abbildung 11 ersichtlich ist, erben alle Eventlistener direkt oder indirekt von der Klasse ChangeListener.

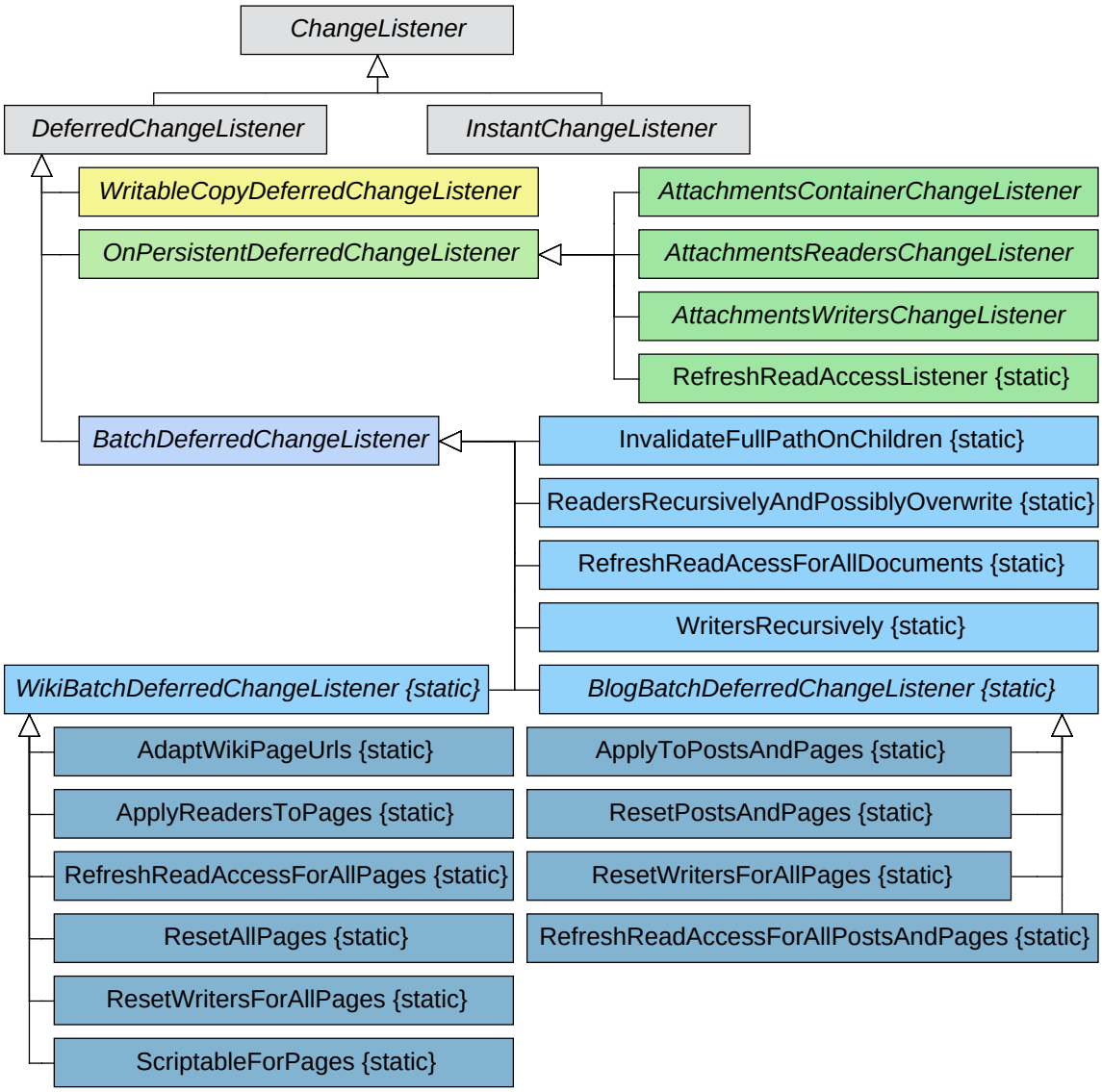


Abbildung 11: Die ChangeListener-Typen als Übersicht; in Anlehnung an [InfoAsset, 2010]

3.1 Analyse der Eventlistener-Typen

Jedem ChangeListener ist ein sogenanntes Diff-Objekt zugeordnet. Es kapselt unter anderem den alten und den neuen Wert des Features, durch dessen Modifikation der ChangeListener getriggert wurde. Wie Abbildung 12 verdeutlicht, gibt es für verschiedene Features auch verschiedene Arten dieser Diff-Objekte. Ein RichStringDiff enthält beispielsweise den alten und den neuen Inhalt einer RichStringProperty.

Für alle Features, die keinerlei gesonderte Behandlung erfordern, existiert zusätzlich ein SimpleValueDiff. Dieses Diff-Objekt enthält als alten und neuen Wert ganz einfach ein Objekt, ganz gleich welchen Typs. Eine Besonderheit stellt der NoChangeDiff dar. Dieses Konstrukt wird immer dann eingesetzt, wenn sich das zugehörige Feature nicht verändert hat, die Auswirkungen der ChangeListener aber erwünscht sind. Hierfür stellen einige Features die Methode triggerChangeListener() bereit. Diese Methode ruft intern featureChanged(**new** NoChangeDiff()) auf und stößt damit die InstantChangeListener direkt und die DeferredChangListener zeitlich versetzt an. Die Gruppe der RoleDiffs ist weiter unterteilt in Diffs zum Löschen, Hinzufügen, Leeren und Setzen von Beziehungen. Die Beziehungen zwischen Assets sind in Tricia, wie die Attribute der Assets, ebenfalls als Features definiert.

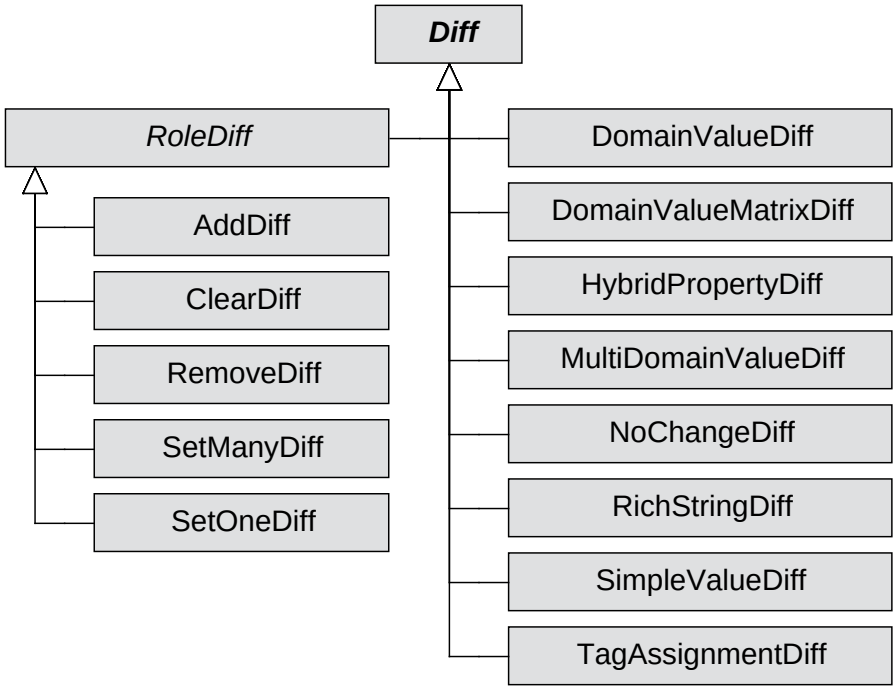


Abbildung 12: Übersicht der Diff-Objekte in Tricia

Zwischen InstantChangeListenern und DeferredChangeListenern wurde bereits in Kapitel 2.3.2 unterschieden. Die DeferredChangeListener sind jedoch noch um einiges feiner ausdifferenziert. In der folgenden Auflistung werden die verschiedenen ChangeListener einzeln erläutert und deren Verwendung in den Tricia Modulen toro, file, blog und wiki analysiert.

InstantChangeListener werden ausgeführt, sobald sich das Feature, an dem sie definiert wurden, verändert. Das heißt sie können auch ohne Persistenzoperation getriggert werden und entfalten ihre Wirkung quasi sofort. Diese Funktionalität ermöglicht es, dass Eventlistener ein Kernkonzept von Tricia darstellen und im Vergleich zu den Möglichkeiten in Hibernate oder JCR ein weitaus größeres Einsatzgebiet abdecken.

Eingesetzt werden die InstantChangeListener beispielsweise als updateUrlName-ChangeListener zur Weitergabe einer Namensänderung an die zugehörige URL. Diese Funktion ist in den Klassen Wiki, WikiPage, Blog, BlogPage, BlogPost, Principal, Search und DocDocument integriert. Allerdings verändert sie die URL nur dann, wenn es sich um ein neues Objekt handelt. Eine weitere Anwendung stellt die Weitergabe von Eigenschaften an abhängige Strukturen dar. Neu erstellte WikiPages übernehmen per InstantChangeListener die Eigenschaften des zugehörigen Wikis bezüglich Kommentier- und Skriptfähigkeit. Es sind außerdem diverse Listener in Paths definiert. Sie erhalten beispielsweise serialisierte Tags im Pfad, speichern alte Pfade automatisch und löschen gleichzeitig alte Pfade aus dem Cache. Darüber hinaus ist das Setzen von Zeitstempeln (Document) und der automatische Mailversand bei Aktivierung einer Membership ebenfalls via InstantChangeListener realisiert. Die InstantChangeListener realisieren somit eine ganze Reihe unterschiedlicher Funktionen im System und sorgen mit ihrer sofortigen Ausführung für das gewünschte Verhalten.

DeferredChangeListener sind Listener, welche zeitlich versetzt ausgeführt werden. Wenn sich Features an einem Asset ändern, so werden die zugehörigen DeferredChangeListener am Asset registriert. All diese Listener werden aktiviert, sobald das zugehörige Objekt persistiert wird. Unter DeferredChangeListenern wird zwischen WritableCopyDeferredChangeListenern, OnPersistentDeferredChangeListenern und BatchDeferredChangeListenern unterschieden.

WritableCopyDeferredChangeListener werden nicht auf dem persistenten Objekt ausgeführt. Von dem Persistenzobjekt wird eine WritableCopy erstellt und auf dieser Kopie der Eventlistener ausgeführt. Dieses Vorgehen ist nur notwendig, wenn innerhalb des Listeners das persistente Ojekt selbst wieder verändert wird. Um Persistenzobjekte zu verändern, muss grundsätzlich zuerst eine WritableCopy erzeugt und diese anschließend wieder persistiert werden. WritableCopyDeferredChangeListener werden in Tricia kaum verwendet.

WikiPages können per Vaterverweis auf eine WikiPage hierarchisch organisiert werden. Die Kinder unterliegen einer gewissen Ordnung, welche natürlich auch dann erhalten bleiben soll, wenn eine zusätzliche WikiPage auf dieselbe Vater-WikiPage verweist. Diese Ordnung wird durch einen WritableCopyDeferredChangeListener erhalten. Außerdem können WikiPages Skripte enthalten, zu deren Ausführung ein Funktionsindex verwaltet wird. Bei einer Veränderung des WikiPage-Inhaltes erstellt ein WritableCopyDeferredChangeListener den Funktionsindex neu. Innerhalb der betrachteten Tricia Module findet dieser Eventlistener-Typ keinerlei weitere Anwendung.

OnPersistentDeferredChangeListener werden direkt auf dem persistenten Objekt ausgeführt. Dies impliziert, dass innerhalb des Listeners das Persistenzobjekt selbst un-

verändert bleibt. Zumeist wird diese Art von ChangeListenern zur Erhaltung von Referenzen eingesetzt. Ein OnPersistentDeferredChangeListener sorgt bei einer URL-Änderung eines Blogs dafür, dass sich alle abhängigen Komponenten (BlogPages und BlogPosts) daran anpassen. Des Weiteren kann die Option showComments im Blog automatisch alle korrespondierenden Optionen in den BlogPages und BlogPosts umstellen. Ein anderes Beispiel stellt die RichStringProperty dar, in der standardmäßig ein OnPersistentDeferredChangeListener definiert ist. Dieser Listener überprüft alle Referenzen auf andere Datenobjekte die innerhalb des Textes deklariert werden. Schlussendlich integriert jede UrlNameProperty einen Listener, der im Falle einer Veränderung alle Assets benachrichtigt, von denen aus auf dieses Objekt verwiesen wird.

Unter den OnPersistentDeferredChangeListenern sind zudem die folgenden, spezialisierten Eventlistener definiert:

- AttachmentsContainerChangeListener
- AttachmentsReadersChangeListener
- AttachmentsWritersChangeListener
- RefreshReadAccessListener

Während die AttachmentsContainerChangeListener in Wikis und Blogs spezifiziert sind, werden AttachmentsReadersChangeListener und AttachmentsWritersChangeListener and den WikiPages und BlogPosts definiert. Funktion der Listener ist hierbei, dass die Anhänge der WikiPages oder BlogPosts automatisch Änderungen an Schreib- bzw. Leserechten ihrer Vaterobjekte weitergereicht bekommen und diese somit übernehmen können. Wenn man einem Nutzer beispielsweise verbietet, auf einer bestimmten WikiPage zu editieren, so sollte dieser User auch keinen Schreibzugriff auf dessen angehängte Dateien mehr haben. Es gehts also wieder einmal darum, das System als Ganzes konsistent zu halten. Der RefreshReadAccessListener aktualisiert, wie der Name bereits nahelegt, die Leserechte auf dem persistenten Objekt. Dies wird oft durch Veränderungen von RightRoles nötig, da nach dieser Operation möglicherweise kein Recht zum Lesezugriff mehr besteht.

BatchDeferredChangeListener sind Eventlistener, welche grundsätzlich asynchron ausgeführt werden und allesamt statisch definiert sind. Wenn die angestoßene Operation besonders aufwändig ist, werden sogenannte Batchjobs generiert. In diesen Batchjobs werden die BatchdeferredChangeListener ausgeführt ohne dass der Nutzer auf deren Fertigstellung warten muss. Wird das System beendet, so werden die noch nicht ausgeführten Batchjobs persistiert und nach erneutem Systemstart automatisch ausgeführt. Als Beispiel sei InvalidateFullPathOnChildren angeführt. Dieser BatchdeferredChangeListener aktualisiert die Pfadangaben von Directory-Objekten und rekursiv aller hierarchisch untergeordneten Path -Objekte. Dies können wiederum Ordner sein, aber einzelne Dokumente. Da das Ende der Ausführung dieses Listeners nicht absehbar ist, wird er asynchron angestoßen. Die bereitgestellten Funktionen der BatchDeferredChangeListener korrespondieren im Wesentlichen mit denen der bereits vorgestellten Eventlistener.

PersistentEvents ergänzen die dargestellte Hierarchie der ChangeListener um weitere Ereignisse und deren Behandlung. Ein PeristentEvent wird nicht wie ein ChangeListener innerhalb eines Features, sondern direkt im Asset, also im zugehörigen Persistenzobjekt definiert. Wie bereits bei den ChangeListenern realisiert, folgen also auch die PersistentEvents dem Grundsatz, dass die Ereignisbehandlung in demselben Objekt definiert werden soll, aus dem später das Ereignis hervorgeht. Die PersistentEvents erweitern somit das Einsatzspektrum der Eventlistener in Tricia enorm. Modifikationen an Objekten können durch dieses Konzept auch auf der Ebene von Persistenzobjekten behandelt werden. Die ChangeListener hingegen beschränken sich auf die Ebene der Features.

Die Ausführung eines solchen PersistentEvents ist dem Ablauf bei den ChangeListenern sehr ähnlich. Nach der Persistierung des Objektes wird per getAfterPersistEvent() der zugehörige Code ausgeführt. Dabei spielt es keine Rolle welche Features im Objekt verändert wurden. Dieser AfterPersistEvent sorgt zum Beispiel dafür, dass nach der Modifikation und Persistierung eines Objektes die Suchfunktion benachrichtigt und aktualisiert wird.

Bei der Erstellung eines gänzlich neuen Objektes wird zusätzlich ein PersistentNewEvent ausgelöst und korrespondierender Code ausgeführt. Darüber hinaus existiert noch eine dritte Art an PersistentEvents, welche jedoch im vorliegenden Kontext weniger wichtig erscheint. Die AfterRemoveEvents behandeln beispielsweise das zweistufige Löschen von Objekten. Dabei werden gewisse Objekte zuerst in den Papierkorb verschoben, nicht direkt gelöscht und können somit später wiederhergestellt werden.

Während der Eventlisteneranalyse sind folgende Punkte aufgefallen:

- Der Listener RefreshReadAccessForAllDocuments wird nicht instanziiert.
- Die beiden Eventlistener namens updateFunctionsIndex an den Features content und scriptable der Klasse WikiPage sind DeferredChangeListener, sollten eigentlich jedoch als WritableCopyDeferredChangeListener deklariert sein.
- Der OnPersistentDeferredChangeListener createLinks in der Klasse RichStringProperty generiert eine writableCopy von dem Entity zu welchem die RichStringProperty gehört und sollte deshalb als WritableCopyDeferredChangeListener deklariert sein.

3.2 Komplexität der Eventlistener

Das Eventlistenerkonzept stellt eines der grundlegenden Architekturkonzepte in Tricia dar. Durch die Registrierung von Listenern direkt am relevanten Objekt wird dem Entwickler viel Arbeit abgenommen, da die Ereignisse weder innerhalb der Listener, noch bei dessen Registrierung gefiltert werden müssen. Zusätzlich erweitert die Variante der InstantChangeListener das Einsatzspektrum der ChangeListener und hebt damit die Implementierung in Tricia deutlich von Hibernate, JCR und vergleichbaren Systemen ab.

Parallel zur Funktionalität erhöht sich auch die Komplexität des Systems, jedoch wird die hohe Komplexität nicht maßgeblich durch die zusätzlichen Funktionen verursacht.

Grundsätzlich ist die Möglichkeit, innerhalb eines ChangeListeners andere Persistenzobjekte zu verändern, eine wichtige und grundlegende Funktion von Eventlistenersystemen in objektrelationalen Persistenzschichten. Nur so kann die Veränderung der URL eines Wikis an dessen WikiPages weitergereicht werden. All diese WikiPages müssen innerhalb des URL-ChangeListeners geladen, das URL-Attribut verändert und wieder persistiert werden. Nun ist es aber möglich, dass an den URL-Attributen der WikiPages wiederum weiter ChangeListener definiert sind und diese durch den ChangeListener getriggert werden. Prinzipiell kann also jeder ChangeListener mehrere weitere ChangeListener auslösen und somit die simple Veränderung eines Attributs zur komplexen und aufwändigen Operation machen.

Als Beispiel sei hierfür ein Wiki angeführt. Die Ausgangssituation stellt sich wie folgt dar: das Wiki beinhaltet diverse WikiPages, während zu einigen dieser WikiPages wiederum Anhänge (Attachments) hochgeladen wurden. Sowohl die WikiPages, als auch deren Attachments, sind von der URL des Wikis direkt oder indirekt abhängig. Eine solche URL wird nach dem Schema in Tabelle 6 generiert.

WikiPage	[hostname]/wikis/[wikiName]/[pageName]
Attachment	[hostname]/file/Attachments/wikis/[wikiName]/[pageName]/[attachmentName]

Tabelle 6: URL Schemata für WikiPages und Attachments

Dies hat zur Folge, dass die Änderung der URL eines Wikis zur umfangreichen Operation wird. Abbildung 13 zeigt den Verlauf dieses Beispiels auf. Der Nutzer ändert initial die URL zusammen mit dem Namen eines Wikis, da er Name und URL konsistent halten möchte. Im Fall eines neuen Wikis wird die URL automatisch aus dem Namen erzeugt und wenn nötig URL-konform anpasst. Nun ist zwar das Wiki unter der neuen URL erreichbar, die enthaltenen WikiPages wissen jedoch noch nichts von der Änderung und besitzen deshalb noch die alte URL. Es müssen also alle URLs der WikiPages einzeln

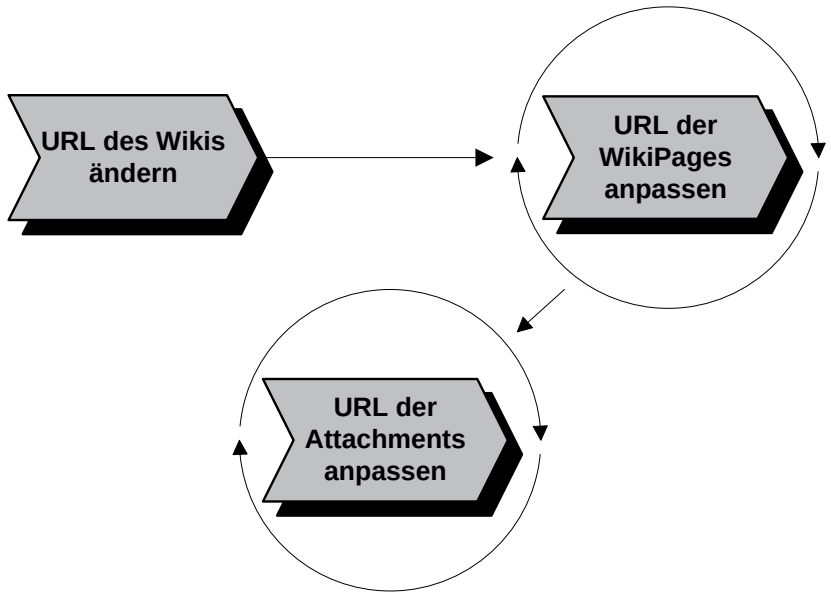


Abbildung 13: Verzweigte Ausführung von ChangeListenern

angepasst werden. Innerhalb dieser Prozesse werden enthaltene Attachments ebenfalls automatisch mittels ChangeListenern angepasst. Hierbei wird deutlich, dass eine simple Änderung durch das Konzept der ChangeListener einen ungeahnt umfangreichen Prozess initiieren kann.

Grundsätzlich sind selbst zyklisch abhängige ChangeListener möglich. Aus einem solchen Zyklus würde eine Endlosschleife in der Ausführung von Tricia resultieren. Bisher ist in Tricias ChangeListener-System kein Mechanismus zur automatischen Detektion solcher Abhängigkeiten integriert. Eine Endlosschleife würde beim Testen des Systems auch schnell auffallen, allerdings ist fraglich, ob ein solches Fehlverhalten sofort auf das System der ChangeListener schließen ließe.

Durch die geschachtelte Ausführung der ChangeListener entsteht demnach zwangsläufig ein dichtes Geflecht von Listenern. Das Geflecht ist zwar Sinnbild für die Flexibilität des Konzeptes, jedoch wird eben dieses Geflecht schnell undurchsichtig und somit zum Problem. Der Nutzer des Systems steht hierbei außen vor, denn er bekommt von den ChangeListenern und deren verzweigter Ausführung nichts mit. Viel mehr entwickelt sich das Konzept für den Entwickler zum Problem. Es existiert für ihn keinerlei Möglichkeit, eine Übersicht über den Programmablauf hinsichtlich ChangeListenern zu bekommen. Solch eine Übersicht sollte verdeutlichen, welche ChangeListener durch welche Veränderung von Attributen ausgelöst werden. Innerhalb einer IDE kann zwar im Debugging Modus Schritt für Schritt der Programmausführung gefolgt werden, diese Methode ist jedoch äußerst aufwändig und bietet keine Übersicht der interessanten Parameter der ChangeListener.

Um ein besseres Verständnis von Tricia und speziell dessen Eventlistener Systems zu ermöglichen, wird ein Tool benötigt, welches die ausgeführten ChangeListener dokumentiert. Das Tool soll außerdem Tricias Entwickler beim Debugging und bei der Erweiterung der Systemfunktionalität unterstützen. Im Rahmen dieser Arbeit wurde ein solches Tool als Eclipse Plugin entwickelt. Im Kapitel 4 wird sowohl der Entwicklungsprozess, als auch das Tool selbst, vorgestellt und erläutert.

4 Change Listener Logger

Wie in Kapitel 3 bereits erläutert wurde, verursacht das in Tricia integrierte System der ChangeListener gerade für Entwickler eine enorme Komplexität. Der genaue Programmablauf ist nur schwer nachzuvollziehen, da jedes Asset über dessen Features Listener enthalten kann, in denen wiederum andere Assets modifiziert werden. Deren Features können selbst wieder ChangeListener triggern und so weiter. Wenn Features verändert werden, kann dies also ungeahnt viele und möglicherweise ungewollte Folgeoperationen nach sich ziehen.

Um die Entwicklung neuer Funktionalität zu beschleunigen und das Debugging zu vereinfachen, wurde im Rahmen dieser Arbeit das Change Listener Logger Tool „CLLogger“entwickelt. Dabei handelt es sich um ein Plugin für die intergrierte Entwicklungsumgebung Eclipse, welches alle getriggerten ChangeListener während der Systemausführung protokolliert. Da der CLLogger die Listener live in einem Eclipse Fenster darstellt, kann die verflochtene Ausführung der ChangeListener beim Debugging Schritt für Schritt nachvollzogen werden.

Ein weiterer Zweck des Tools ist es, den Entwicklern ein umfassendes Verständnis des ChangeListener-Konzeptes zu vermitteln. Nicht alle Entwickler haben die Evolution des Eventlistener-Systems mit begleitet. Da es sich jedoch um eines der grundlegenden Konzepte Tricias handelt, wird die Funktionalität der ChangeListener in jedem Tricia-Plugin genutzt. Deshalb sollten auch jene Entwickler, die sich nicht besonders detailiert mit den ChangeListenern befasst haben, ohne allzu großen Aufwand ein grundlegendes Verständnis der geschachtelten ChangeListener-Ausführung bekommen. Bei Bedarf bietet der CLLogger außerdem tiefe Einblicke in das Abhängigkeitsgeflecht der ChangeListener.

Grundlegende Anforderungen an das CLLogger Tool wurden in Kooperation mit erfahrenen Tricia Entwicklern ausgearbeitet und folgende Randpunkte fixiert:

- Das Tool soll in Java implementiert werden.
- Das Tool soll als eigene Applikation oder als Eclipse Plugin realisiert werden.
- Der zusätzliche Code zur Anbindung der CLLoggers an Tricia soll die Performance des Systems so wenig wie möglich beeinträchtigen.
- Vorerst soll der Fokus auf die DeferredChangeListener gelegt werden. Die zusätzliche Behandlung der InstantChangeListener soll nur integriert werden, wenn sie keine allzu großen Umstände bereitet. In diesem Fall soll jedoch ein Schalter integriert werden, mit dem man das Protokollieren von InstantChangeListenern ein- und ausschalten kann.
- Die getriggerten ChangeListener und PersistEvents sollen live im Tool dargestellt werden.
- Die Chronologie der Eventlistener soll ebenfalls dokumentiert werden.

- Zusätzlich zu den ChangeListenern sollen PersistentEvents berücksichtigt werden. Dabei sind sowohl PersistentNewEvents, als auch AfterPersistEvents von Bedeutung und sollen analog zu ChangeListenern behandelt werden. Die AfterRemoveEvents sind im Kontext des CLLogger Tools nicht relevant und müssen nicht mit einbezogen werden.

Wie bei jedem Logging Tool stellt sich auch für den CLLogger die Frage, welche Informationen und Daten protokolliert werden sollen. Die verschiedenen Arten von ChangeListenern und PersistentEvents wurden dahingehend analysiert. Es stellte sich heraus, dass bezüglich der InstantChangeListener und der DeferredChangeListener dieselben Datenfelder relevant sind. PersistentNewEvents und AfterPersistEvents können ebenfalls gruppiert betrachtet werden, da auch hier dieselben Informationen von Bedeutung sind. Eine Auflistung der relevanten Daten findet sich in Tabelle 7.

InstantChangeListener & DeferredChangeListener	Der Name des ChangeListeners und der Typ des zuständigen Handlers werden protokolliert. Ein Handler wird in Tricia für jede Anfrage des Nutzers instanziiert und führt anschließend die notwendigen Operationen im System aus. Somit kapselt ein Handler diejenigen ChangeListener, die innerhalb dieser Anfrage getriggert werden. Des Weiteren werden sowohl der Name des Features, an dem der ChangeListener definiert wurde, als auch dessen Typ protokolliert. Von Interesse ist darüber hinaus auch der Name und Typ des Assets, welches eben dieses Feature enthält. Hierbei kann es vorkommen, dass dieses Asset kein direktes Persistenzobjekt darstellt, sondern als Mixin (eine Art Eigenschaft eines Persistenzobjektes) einem anderen Asset zugeordnet ist. In diesem Fall sind Name und Typ beider Assets relevant. Außerordentlich wichtig sind die Informationen über die Veränderung des Features an sich. Jeder ChangeListener beinhaltet ein Diff-Objekt. Da dieses Objekt die Modifikation inklusive altem und neuem Wert enthält, ist eine Textdarstellung des Diff-Objektes im Protokoll obligatorisch.
PersistentNewEvents & AfterPersistEvents	Die relevanten Informationen der PersistentEvents sind denen der ChangeListener ähnlich. PersistentEvents besitzen jedoch kein Diff-Objekt. Protokolliert werden neben dem Ereignis-Typ, der zuständige Handler, sowie Name und Typ des spezifischen Assets. Auch in diesem Fall kann der PersistentEvent durch ein Mixin mit dem Basis-Asset verbunden sein. Dann werden, wie bei den ChangeListenern, die Daten von beiden Assets aufgenommen. Da die PersistentEvents direkt mit einem Asset verbunden sind, wird kein Feature mitprotokolliert.

Tabelle 7: Relevante Daten und Informationen der Eventlistener in Tricia

Rahmenbedingungen für die Repräsentation dieser Daten wurden nur spärlich definiert. Es stand lediglich die Idee im Raum, dass die ChangeListener und PersistentEvents in einer Baumdarstellung angezeigt werden. Konkretisiert wurden diese Vorstellungen erst als fest stand, mit welchem Umfang an Daten gerechnet werden muss. Wie in Kapitel 3.2 bereits erläutert wurde, kann die geschachtelte Ausführung von Eventlistenern relativ schnell unübersichtlich, umfangreich und in sich komplex werden. Aufgrund dessen war das Vorgehen, die Nutzerschnittstelle erst später im Entwicklungsprozess festzulegen, plausibel und konnte im Nachhinein auch als richtige Entscheidung bestätigt werden.

Während die Entwicklung des CLLoggers in Kapitel 4.1 beschrieben ist, wird das Tool in Kapitel 4.3 dokumentiert. Hierbei wird die Integration des CLLoggers in den Entwicklungsworkflow anhand des in Kapitel 3.2 angeführten Wiki-Beispiels erläutert.

4.1 Analyse des Persistierungsprozesses

Angesichts des Ziels, ein kompaktes CLLogger Tool zu entwickeln, wurde bewusst auf ein schwergewichtiges Vorgehensmodell verzichtet. Grundlegend lässt sich die Entwicklung des Tool jedoch in zwei Prozesse aufteilen. Zum einen müssen die Daten, welche in Tabelle 7 beschrieben wurden, im System während der Laufzeit abgegriffen werden. Zum anderen wurde das Tool an sich entwickelt, was sich wiederum in die Implementierung des Konnektors, das GUI Design, sowie die interne Verarbeitung der Eventlistener aufteilen lässt.

Den ersten Schritt stellte die Entwicklung der in Tricia zu integrierenden Protokollfunktion dar. Dieses Modul konnte relativ isoliert entwickelt werden, da die Anforderungen in Tabelle 7 klar definiert wurden. Die relevanten Informationen sind allerdings, bedingt durch die Architektur des ChangeListener-Konzeptes, oft nur über Umwege zu erreichen. Dieses Problem ist vor allem der Registrierung von Eventlistener via anonymen Klassen geschuldet. Zu Beginn stand deshalb eine umfassende Einarbeitung in das System als Ganzes, sowie in den Quelltext der Komponenten toro und wiki an.

Der CLLogger soll für jeden ausgeführten ChangeListener oder PersistentEvent Daten protokollieren. Dazu war es wichtig, die passenden Stellen im Quellcode von Tricia zu identifizieren. Schnell wurde klar, dass es sich um zwei verschiedene Prozesse im Modul toro handelt. Zum einen werden die InstantChangeListener sofort nach der Modifikation eines Features ausgeführt, zum anderen werden die übrigen Eventlistener allesamt erst nach der nächsten Persistierung des Objektes getriggert. Der CLLogger muss also an zwei Stellen Informationen abgreifen.

Die relevante Stelle Für InstantChangeListener konnte in der Klasse Feature identifiziert werden. Jeder einzelne InstantChangeListener ist in einer Unterklasse von Feature definiert. In der Klasse Feature existiert nun eine Methode mit der Signatur featureChanged(**final** Diff diff). Sie wird direkt aufgerufen, sobald ein Feature verändert wird. Innerhalb der Methode werden alle an diesem speziellen Feature registrierten InstantChangeListener mittels changeListener.change(diff) angestoßen. Dieser Aufruf wird synchron verarbeitet, das heißt die ChangeListener werden komplett ausgeführt ehe die Methode featureChanged(...) beendet wird.

Sowohl die zeitliche Abfolge der InstantChangeListener, als auch deren verschachtelte Ausführung, können erhalten bleiben, indem man den CLLogger direkt vor und nach der Methode changeListener.change(diff) benachrichtigt. Dabei ist der erste Aufruf eine Art Startereignis und der zweite das korrespondierende Endereignis. Wenn sich die beiden Ereignisse einander zuordnen lassen, so kann man sie eindeutig von all dem Code unterscheiden, der innerhalb dieses InstantChangeListeners ausgeführt wurde. Ein Listener kann durchaus selbst wiederum andere ChangeListener auslösen. Um die Ausführungshierarchie richtig zu protokollieren, muss der CLLogger im Tool selbst eine Art Baum verwalten. Wenn ein Startereignis aus Tricia übermittelt wird, so kann ein Knoten hinzugefügt werden. Steht in dem Moment jedoch noch ein Endereignis eines anderen Listeners aus, so wird der neue Knoten unterhalb des noch aktiven Knotens eingefügt.

Die InstantChangeListener können somit protokolliert werden. Nicht ganz so einfach sieht der Prozess für die restlichen Eventlistener aus. Die Analyse hat gezeigt, dass diese Listener in der Klasse PersistentEntity innerhalb der Persistierungsmethode angestoßen werden. Der erste Versuch, den Prozess der Objektpersistierung zum besseren Verständnis und zur Dokumentation in ein UML Sequenzdiagramm zu überführen, ist in Abbildung 14 zu sehen. Das Modell repräsentiert den Speichervorgang des Systems [Stand: 17.11.2010], ist jedoch noch viel zu umfangreich und komplex. Da UML bezüglich Sequenzdiagrammen diverse Richtlinien definiert, wurde für den zweiten Anlauf die Syntax eines einfachen Flussdiagramms herangezogen. Abbildung 15 zeigt das weitaus abstraktere Ergebnis [Stand: 17.11.2010]. Die stark verringerte Komplexität dieses Modells wurde zusätzlich durch eine umfassende Optimierung des Quelltextes ermöglicht.

Zu Beginn des Persistierungsprozesses in Abbildung 15 wird geprüft, ob das Objekt überhaupt beschreibbar ist (isWritable). Wenn dies zutrifft, werden alle weiteren Schritte in einem ServerMode(Systemzustand) ausgeführt. Dieser Modus NoReadAccessCheck gilt danach für den Rest der Persistenzoperation. Allgemein beschreiben die in Graustufen hinterlegten Rechtecke allesamt solche ServerModes. Die nächste Verzweigung prüft, ob die Option CheckWriteAccess aktiviert ist und ob das Objekt modifiziert werden darf. Wenn die Option aktiv ist und das Objekt nicht verändert werden darf, so wird der gesamte Prozess abgebrochen. Andernfalls kann der Prozess weitergeführt und in der folgenden Verzweigung die Validität geprüft werden. Auch hierbei existiert eine Option CheckValidity . Falls diese Option aktiv und das Objekt selbst nicht valide ist, so wird auch an dieser Stelle der gesamte Persistierungsprozess abgerochen.

Die weiteren Operationen werden im ServerMode PermitPersistentOperations ausgeführt. Lediglich innerhalb dieses ServerModes können Objekte persistiert werden. Der eigentlichen Persistierung ist ein Hook vorgelagert. Der AfterIsValidAndBeforeMakePersistentHook kann per Option (die vorgelagerte Verzweigung) aktiviert werden und bietet die Möglichkeit, spezifischen Code direkt vor der Persistierung auszuführen. Anschließend werden alle DeferredChangeListener des aktiven Objektes in einer Liste gespeichert und die Persistierung angestoßen. Die Methode applyPersist() kapselt in Tricia die Kommunikation mit der ORM-Schicht und kann somit in dieser Analyse als Blackbox betrachtet werden.

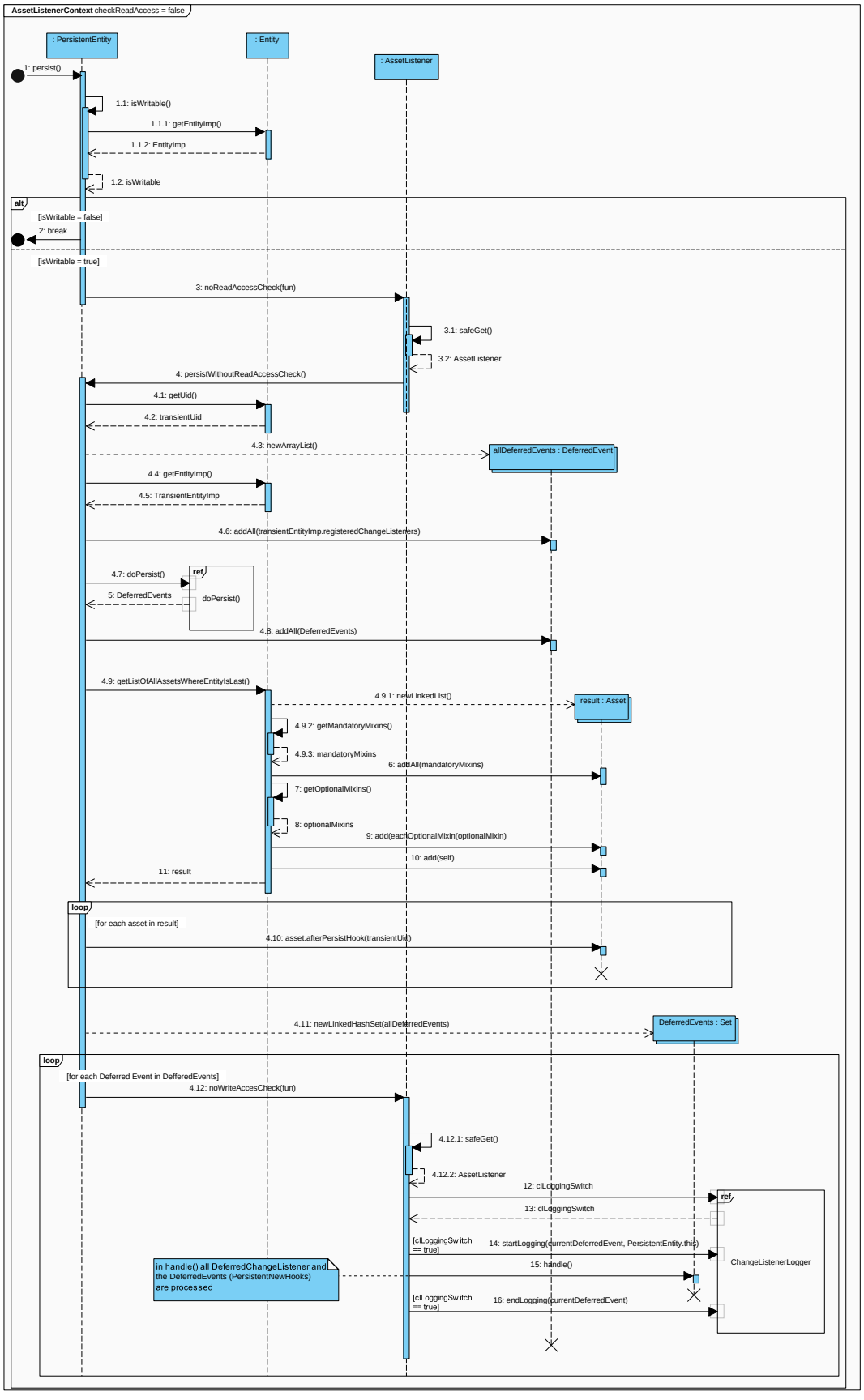


Abbildung 14: Granulare Dokumentation des Persistierungsprozesses in Tricia. [Stand: 17.11.2010]

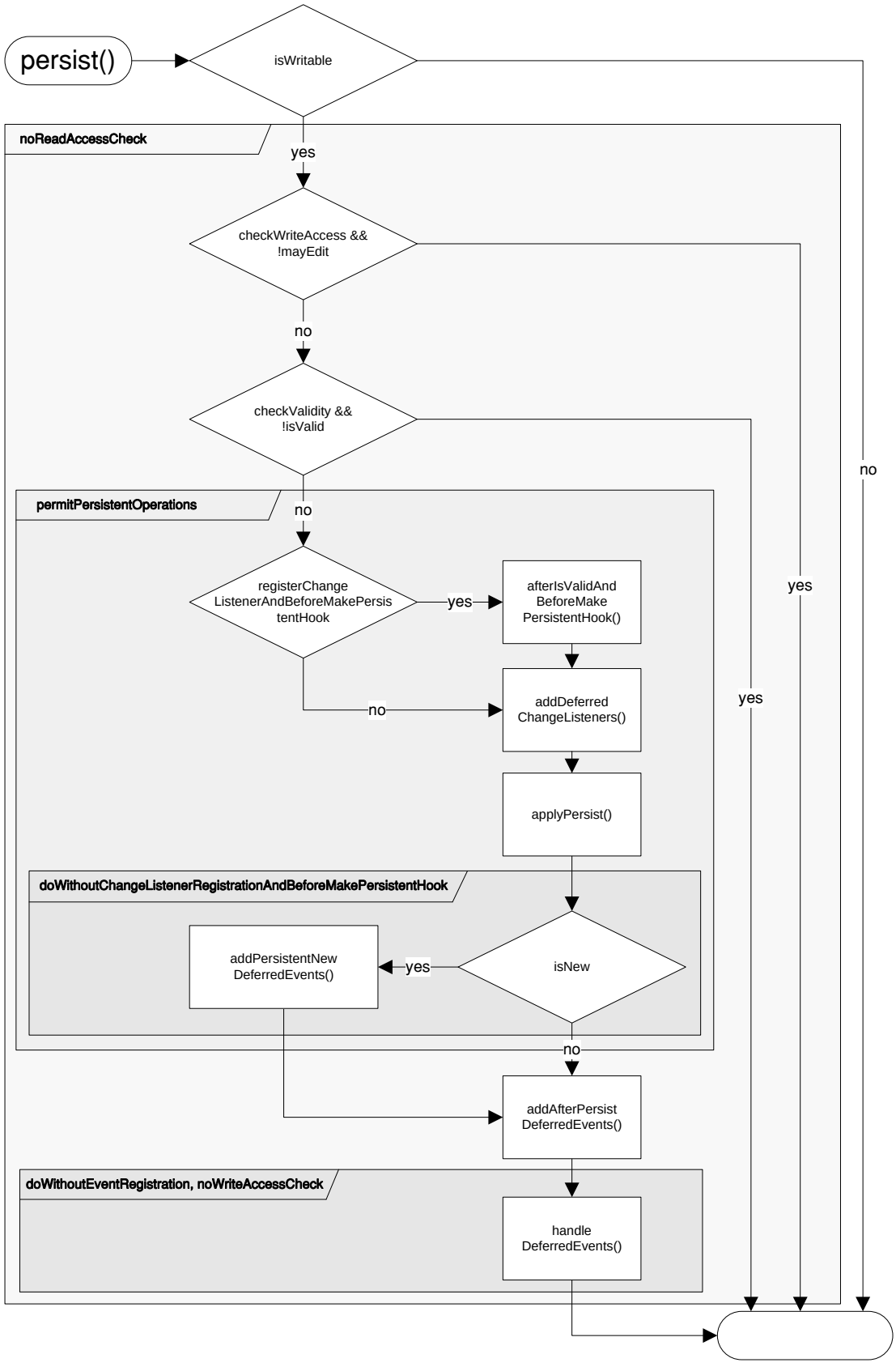


Abbildung 15: Eine abstrakte Modellierung des Persistierungsprozesses in Tricia. [Stand: 17.11.2010]

Zusätzlich zum bereits aktiven ServerMode PermitPersistentOperations wird der ServerMode doWithoutChangeListenerRegistrationAndBeforeMakePersistentHook eingeleitet. Innerhalb dieser ServerModes werden, sofern es sich bei dem Persistenzobjekt um ein neues Objekt handelt, PersistentNewEvents zur Liste der DeferredChangeListenern hinzugefügt. Nun werden alle AfterPersistEvents des persistierten Objektes an die Liste der DeferredEvents angehängt. Dazu sind jedoch die ServerModes doWithoutChangeListenerRegistrationAndBeforeMakePersistentHook und auch die PermitPersistentOperations aufgehoben. Die resultierende Liste enthält nun sowohl DeferredChangeListener, als auch PersistentNewEvents und AfterPersistEvents.

Daraufhin werden die Eventlistener aus der Liste nacheinander angestoßen. Hierfür geht das System in die ServerModes DoWithoutEventRegistration und NoWriteAccessCheck über. Dadurch wird zum einen verhindert, dass während der Ausführung der Listener wiederum neue Listener registriert werden und zum anderen, dass die Ausführung der Listener am Zugriffsschutz scheitert. Die Methode handleDeferredEvents() startet schlussendlich die einzelnen Eventlistener.

Es werden somit während des Persistierungsprozesses alle registrierten Eventlistener gesammelt und nach erfolgreicher Objektspeicherung ausgeführt. Analog zum Prozess bei den InstantChangeListenern handelt es sich auch hier um lediglich eine Zeile, die die Listener anstößt. Somit kann das Vorgehen von den InstantChangeListenern eins zu eins übertragen werden. Es soll also direkt vor und nach handleDeferredEvents() eine Nachricht generiert und zum CLLogger Tool geschickt werden. Abbildung 16 zeigt einen angepassten Ausschnitt aus Abbildung 15. Hierbei wird per LoggingSwitch-Option das Protokollieren an- bzw. ausgeschalten. Wenn es aktiv ist, so wird der CLLogger per startLogging() benachrichtigt und danach durch handleDeferredEvents() der jeweilige Eventlistener angetoßen. Nach Be-

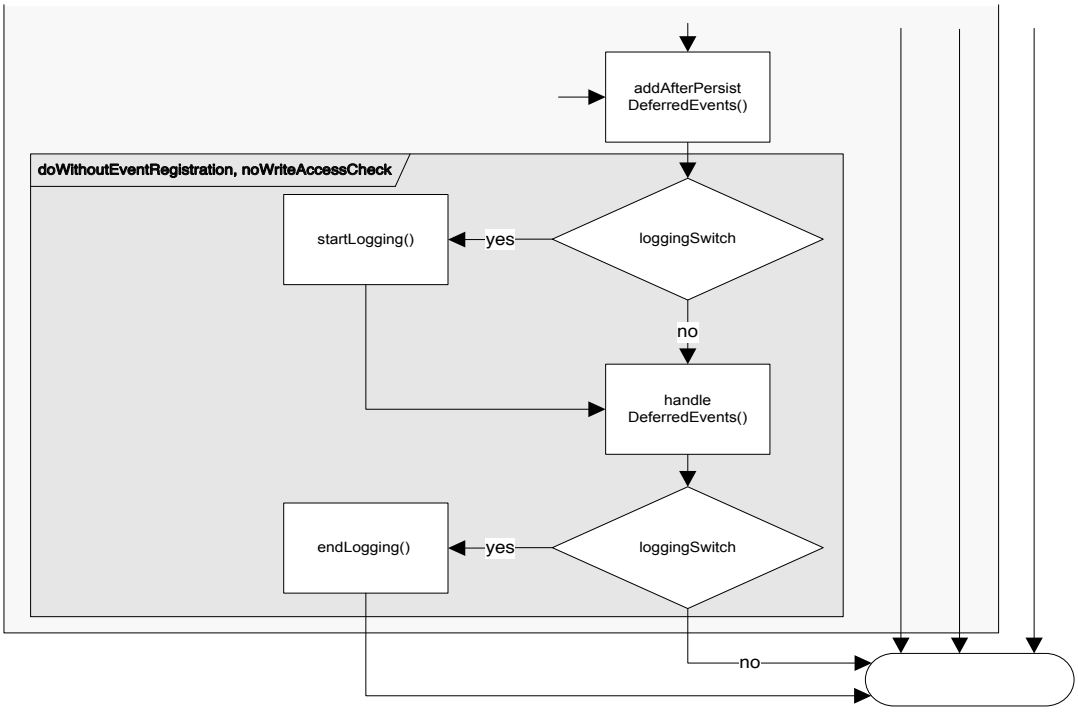


Abbildung 16: Modifikation des Persistierungsprozesses

endigung der Listenerausführung wird der CLLogger schließlich via endLogging() über diesen Fortschritt informiert. Die Schleife, welche für jeden einzelnen Eventlistener aus der DeferredEvent-Liste einmal durchlaufen wird, enthält auch die beiden Aufrufe startLogging() und endLogging(). So kann gewährleistet werden, dass für jeden Listener ein Start- und ein Endereignis generiert wird.

Die relevanten Passagen im Quelltext von toro sind somit identifiziert. Außerdem wurde bereits eine Technik ins Auge gefasst, mit der die Chronologie und die Hierarchie der ausgeführten ChangeListener und PersistentEvents protokolliert werden kann. Im folgenden Kapitel 4.2 wird die Realisierung des CLLogger Tools dokumentiert.

4.2 Die Entwicklung des Tools

In Tabelle 7 sind all jene Daten aufgelistet, welche vom CLLogger über die ChangeListener, PersistentNewEvents und AfterPersistEvents gesammelt und protokolliert werden sollen. An den in Kapitel 4.1 identifizierten Stellen im Code kann auf all diese Informationen zugegriffen werden. Zur vorübergehenden Darstellung dieser Daten wurde eine Klasse implementiert, die sämtliche Informationen über die getriggerten Eventlistener auf der Console ausgibt. Abbildung 17 zeigt einen Screenshot der Konsole.

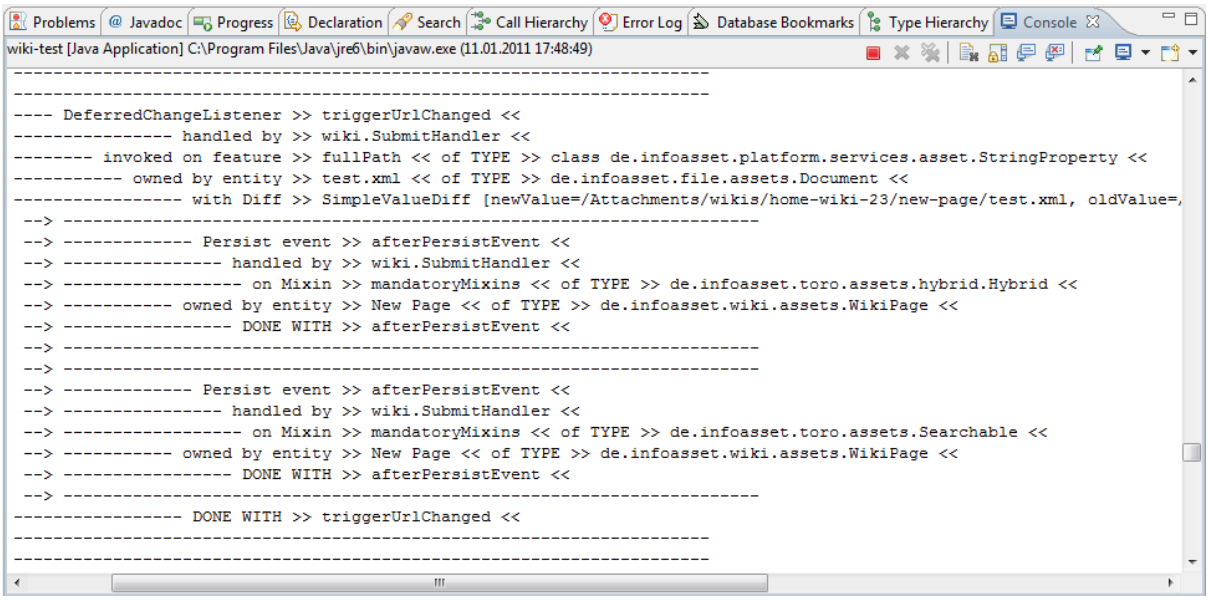


Abbildung 17: Konsolenausgabe zur Darstellung von ChangeListenern

Dieses Beispiel zeigt in der Konsole einen DeferredChangeListener und zwei PersistentEvents. Die Einrückung der beiden AfterPersistEvents repräsentiert dabei die hierarchische Struktur der Eventlistener-Ausführung. Für jede Hierarchiestufe wird der Ausgabe ein zusätzlicher Pfeil vorangestellt. Um die verzweigte Ausführung der Listener vollständig zu erfassen, wird zu jedem Listener eine weitere Zeile ausgegeben, sobald sein Rumpf vollständig verarbeitet wurde. Die Zeile wird nach dem Schema DONE WITH >> [ListenerName] << generiert. Die beiden AfterPersistEvents sind vom DeferredChangeListeners triggerUrlChanged, sowie dessen Endzeile am unteren Rand des Screenshots, umschlossen.

In der Ausgabe sind der Name des ChangeListeners und dessen zuständigen Handlers angegeben. Im Beispiel handelt es sich hierbei um triggerUrlChanged der von einem wiki.SubmitHandler ausgeführt wird. Wenn beispielsweise einer Wikiseite eine neue URL zugewiesen wird, so muss diese Änderung unter anderem an die angehängten Dateien der Seite propagiert werden. Der vorliegende DeferredChangeListener triggerUrlChanged repräsentiert einen Teil dieser logischen Folgemaßnahmen. Die Ausgabe zeigt darüber hinaus den Namen und den Typ des veränderten Features. Im Beispiel wurde also der Pfad (fullPath; eine StringProperty) verändert. Dieses Feature ist in einem Document-Objekt namens test.xml verankert. Dieses Objekt repräsentiert einen XML-Anhang zur relevanten WikiPage und auch von ihm werden Name und Typ protokolliert (owned by entity...).

Zusätzlich werden für ChangeListener die zugehörigen Diff-Objekte ausgegeben. Verantwortlich für die resultierende Darstellung der verschiedenen Diff-Typen ist deren Methode toString(), welche die strukturierten Daten in eine Zeichenkette überführt. In diesem speziellen Fall handelt es sich um ein SimpleValueDiff, welches Informationen über die Modifikation des Pfadattributes von test.xml beinhaltet. Die beiden eingeschlossenen AfterPersistEvents werden innerhalb des DeferredChangeListeners triggerUrlChanged angestoßen, da dieser die Verlinkungen auf den Anhang test.xml aktualisiert. Diese Aktualisierung modifiziert die WikiPage New Page, welche den Anhang enthält, und persistiert diese wieder. Für WikiPages sind wiederum unter anderem die Mixins Hybrid und Searchable definiert. Diese beiden Mixins enthalten jeweils einen AfterPersistEvent. Im AfterPersistEvent des Mixins Searchable wird beispielsweise das Suchmodul (Lucene) von der Veränderung benachrichtigt. Alle relevanten, in Tabelle 7 aufgeführten, Daten können somit protokolliert werden.

Im nächsten Schritt wurden zu allererst einige Randpunkte für die GUI definiert. Grundsätzlich legt die hierarchische Ausführungsstruktur der Eventlistener eine baumbasierte Darstellung nahe. In einem Baum bleibt die Hierarchie bzw. Schachtelung der getriggerten Listener bereits implizit erhalten. Ebenfalls kann die chronologische Abfolge der Listener einfach als Reihenfolge der einzelnen Knoten mit in die Datenstruktur aufgenommen werden. Basierend auf dieser Erkenntnis wurden folgende Anforderungen dokumentiert:

- Als Knoten der höchsten Hierarchieebene sollen die Handler eingesetzt werden. Sie fungieren in der Ansicht quasi als Sammelordner für die ChangeListener und PersistentEvents, welche durch sie angestoßen werden.
- Die Reihenfolge dieser Handler-Knoten soll deren chronologischer Abfolge entsprechen.
- Jeder Eventlistener wird als ein Knoten im Baum dargestellt. Wenn während der Ausführung weitere Listener getriggert werden, so erscheinen diese durch das Expandieren des umschließenden Listeners.
- Zusätzlich zur Baumdarstellung soll ein Feld integriert werden, in dem alle protokollierten Daten der Eventlistener angezeigt werden. Dabei sollen jeweils die Informationen über den momentan selektierten Listener dargestellt werden. Ist ein Handler-Knoten selektiert, so wird nichts angezeigt, da dieser keine spezifischen Daten beinhaltet.

- Es soll eine Suchfunktion integriert werden, die den Baum direkt an das Suchwort anpasst und alle Knoten ausblendet, die nicht zum Suchwort passen. Hiermit kann der Baum komfortabel gefiltert werden. Es sollen alle Daten über die ChangeListener bzw. PersistentEvents mit in die Suche einbezogen werden.
- Für jede Art der Eventlistener soll ein passendes Icon gefunden und in die Baumdarstellung integriert werden

Diese Vorgaben wurden in das Mockup in Abbildung 18 übertragen. Die Suchleiste am oberen Rand des Fensters wird als Filtermechanismus für den die Baumdarstellung verwendet. Der Baum selbst enthält einige Handler-Knoten. Diese Handler beinhalten diverse, teils hierarchisch ausgeführte, ChangeListener und PersistentEvents. Einem Knoten untergeordnete Listener können auf- und zugeklappt werden. Am unteren Fensterrand werden die Eigenschaften (Properties) der einzelnen Eventlistener angezeigt.

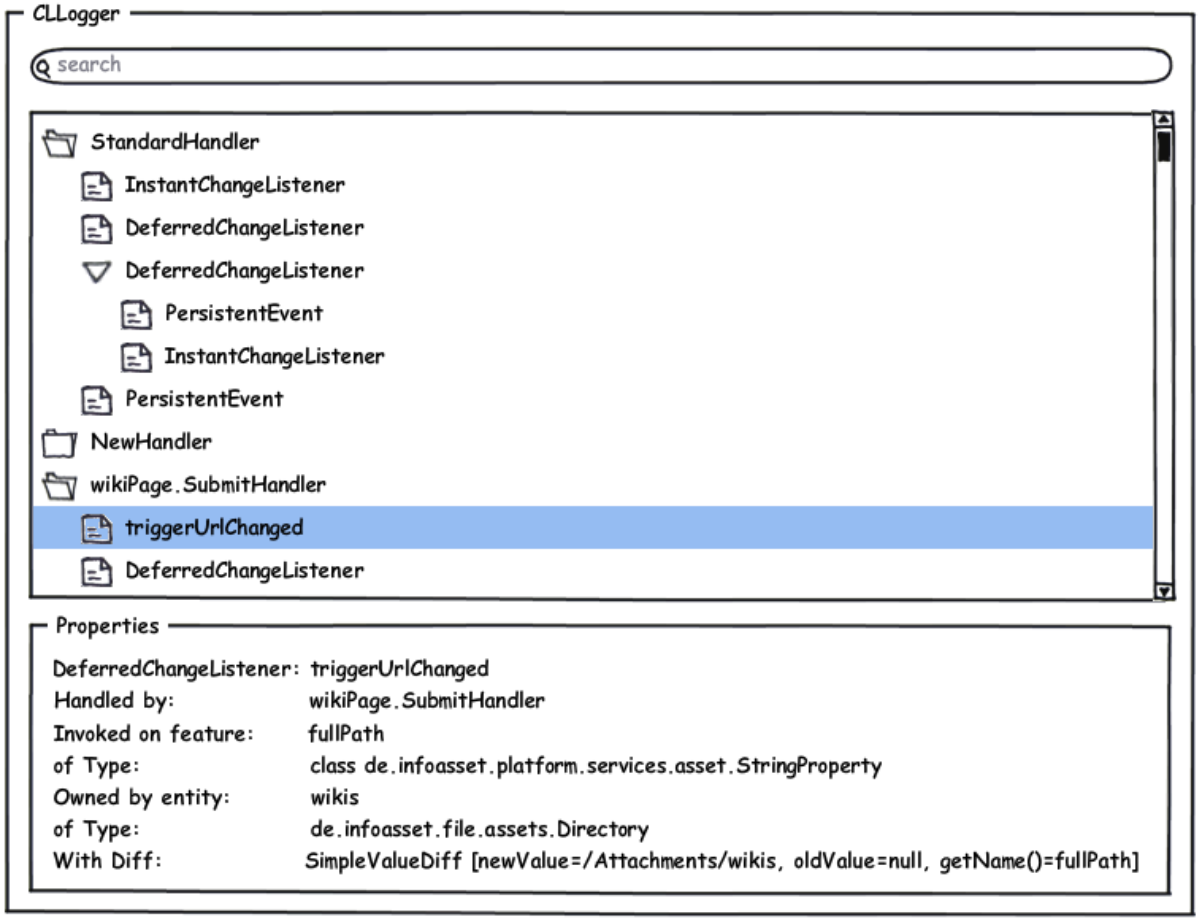


Abbildung 18: Mockup der CLLogger GUI

Im angegebenen Beispiel handelt es sich um den DeferredChangeListener triggerUrlChanged. Dieser Listener wird angestoßen, wenn das Feature fullPath an einem Verzeichnis namens wikis modifiziert wird. Dabei wird als Diff ein SimpleValueDiff mit dem alten und neuen Pfad übergeben.

Im nächsten Schritt wurde diese graphische Benutzeroberfläche auf das CLLogger Tool übertragen. Dazu haben sich zwei Alternativen ergeben:

- Die Funktionalität wird rein mit SWT implementiert. Dieses Vorgehen lässt offen, ob das Tool als Eclipse Plugin oder eigenes Programm realisiert wird. SWT stellt dabei Java Bausteine zur GUI Programmierung bereit. Unter anderem findet sich dabei auch eine Komponente zur Darstellung von Bäumen [Eclipse, 2010].
- Bei der Implementierung wird JFace eingesetzt. Dieses UI-Toolkit setzt aus SWT Komponenten komplexere Artefakte zusammen. Mit JFace lässt sich beispielsweise ein FilteredTree erstellen [Eclipse, 2008]. Das Besondere an der Baumdarstellung ist hierbei, dass ein Filtermechanismus mit umfassender Funktionalität bereits integriert ist. Über dem Baum wird standardmäßig ein Textfeld zur Eingabe des Filters angezeigt. Bei Eingabe einer Zeichenkette wird automatisch der gesamte Baum gefiltert und alle unpassenden Knoten ausgeblendet. Allerdings hängt die Komponente FilteredTree von einigen Eclipse-Bibliotheken ab, sodass sie nicht ohne größeren Aufwand in einem eigenen Programm außerhalb von Eclipse genutzt werden kann.

Die Entscheidung fiel diesbezüglich auf JFace und den FilteredTree, da in ihm ein beträchtlicher Teil der Programmlogik bereits implementiert ist. Zuerst wurde das Ziel verfolgt, den FilteredTree außerhalb eines Eclipse-Plugins funktionstüchtig in das CLLogger Tool zu integrieren. Dieses Vorgehen stellte sich jedoch bald als sehr komplex heraus, da der FilteredTree auf diverse weitere Ressourcen zugreift, die in einer normalen Java Applikation nicht vorhanden sind. Von da an wurde die Entwicklung eines CLLogger Plugins vorangetrieben. Innerhalb eines Eclipse-Plugins funktioniert der FilteredTree einwandfrei. Auch kann somit eine nahtlosere Integration des CLLogger Tools in den Entwicklungsprozess von Tricia erreicht werden, da das Analysetool direkt in die IDE Eclipse eingebettet wird.

Die Entscheidung für ein Eclipse-Plugin birgt jedoch ein zusätzliches Problem. Wie können die protokollierten Daten während der Ausführung von Tricia an das Eclipse Plugin übermittelt werden? Die beiden Komponenten werden in verschiedenen Prozessen ausgeführt und können somit nicht direkt auf Ressourcen des jeweilig anderen zugreifen. Nach ausführlicher Recherche relevanter Quellen hat sich ergeben, dass zur Lösung dieses Problems das RMI Konzept zweckentfremdet werden kann. Da in diesem Anwendungsfall jedoch beide Prozesse auf dem selben Rechner ausgeführt werden, handelt es sich hierbei um keinen wirklich „entfernten“ Methodenaufruf.

Abbildung 19 zeigt das Prinzip von Java RMI. Grundsätzlich fungiert immer eine Komponente als Server und eine andere als Client. Die RMI Registry wird dabei vom Server verwaltet und leitet dem Client auf Anfrage die Adresse der relevanten Serverklasse weiter. Um die Klasse dann auch wirklich zu laden, kommuniziert der Client dann direkt mit dem Directory. Um ein genaueres Bild vom Ablauf bei der Java RMI Kommunikation zu bekommen, werden die Prozesse in der folgenden Auflistung einzeln erläutert.

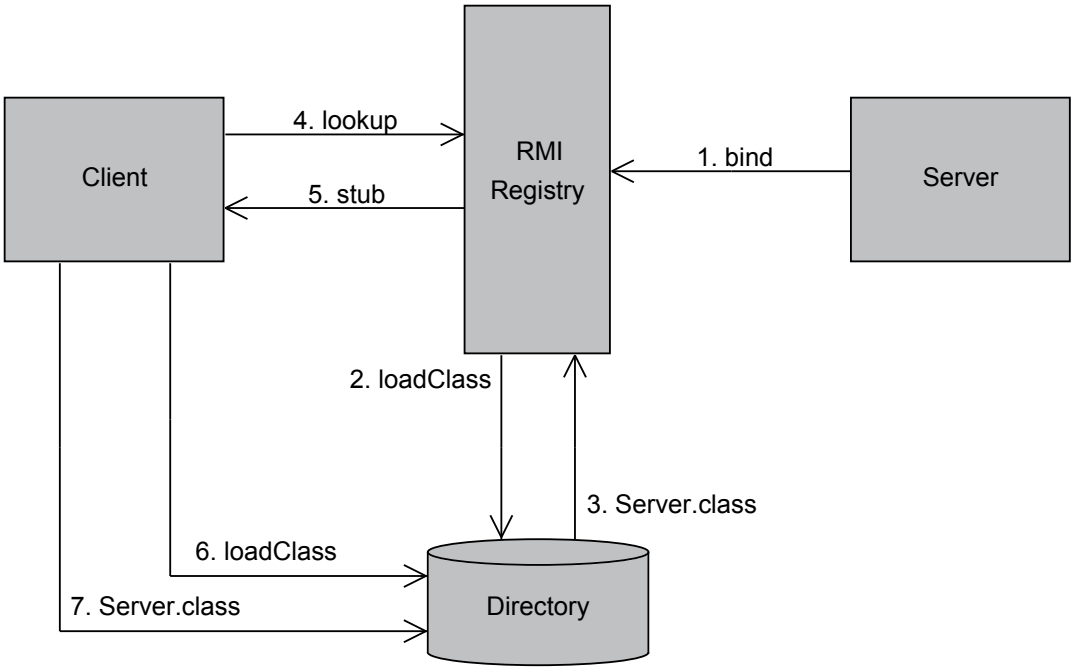


Abbildung 19: Funktionsweise von Java RMI; in Anlehnung an [Heinisch et al., 2007, S.1034]

1. Anfangs registriert der Server die Klasse Server.**class** unter einem bestimmten Servicenamen bei der RMI Registry. Dabei wird unter anderem die Pfadangabe zu der Serverklasse übermittelt. Diesen Prozess nennt man „Binding“, denn er verbindet eine Klasse mit dem spezifischen Namen des Services.

2. Die RMI Registry lädt daraufhin die identifizierte Datei aus dem spezifizierten Datenverzeichnis.

3. Die RMI Registry empfängt die Klasse Server.**class**.

4. Nun greift der Client das erste mal aktiv ein. Er fragt per lookup bei der RMI Registry einen bestimmten Service an.

5. Die RMI Registry schickt dem Client nicht die wirkliche Klasse, sondern ein Stub-Objekt (Quasi ein Ersatzobjekt für die lokal nicht verfügbare, wirkliche Serverklasse).

6. Im Stub-Objekt ist die Information enthalten, wo sich der Code der Klasse Server.**class** befindet. Basierend darauf kann der Client nun diesen Code anfordern.

7. Schlussendlich erhält der Client den Code der Serverklasse. Er kann somit das Stub-Objekt ansprechen, wie wenn dies lokal verfügbar wäre und nicht nur eine Referenz auf das reale, entfernte Server-Objekt.

Per Remote Method Invocation können somit alle protokollierten Daten der ChangeListener und PersistentEvents an das CLLogger Plugin weitergeleitet werden. Für eine Beschreibung der RMI Funktionalität im CLLogger Tool sei auf Kapitel 4.3 verwiesen.

Nachdem alle Probleme mit der Kommunikation zwischen Plugin und dem laufenden Tricia System beseitigt waren, konnte die GUI (Mockup in Abbildung 18) implementiert werden. Abbildung 20 zeigt die entstandene, in die Eclipse IDE integrierte Benutzeroberfläche. Die getriggerten Eventlistener wurden von einem wikiPage.SubmitHandler angestoßen, der beispielsweise nach der Änderung des Namens einer WikiPage ausgeführt wird. Abbildung 21 zeigt zum Vergleich die korrespondierende Konsolenausgabe.

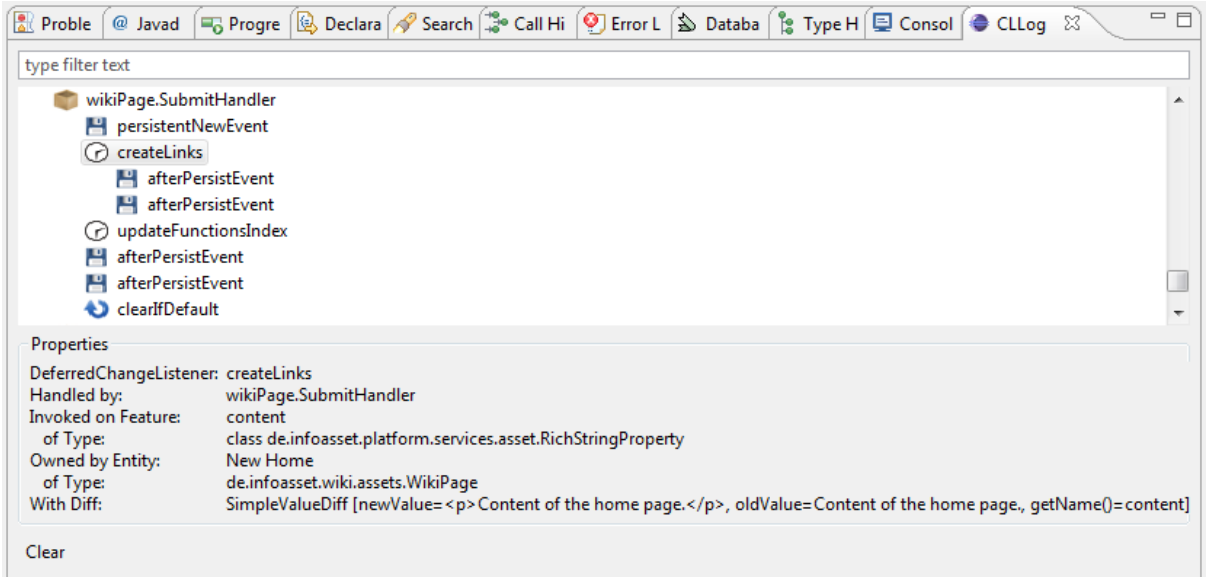


Abbildung 20: Die GUI des CLLoggerPlugins

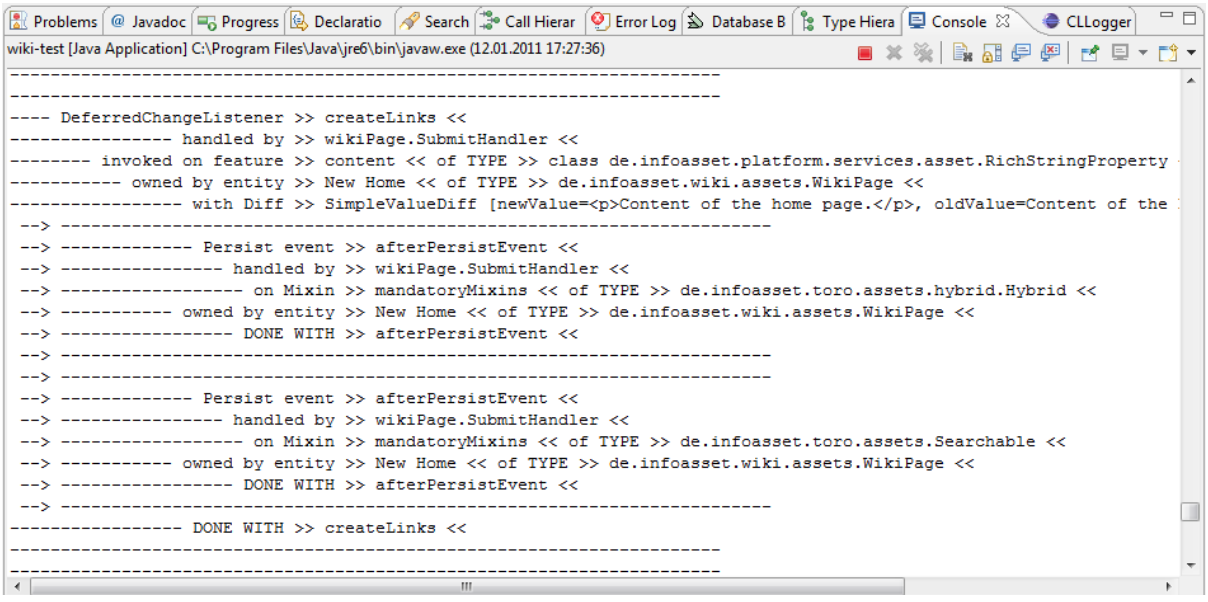


Abbildung 21: Zum Vergleich mit Abbildung 20 die korrespondierende Konsolenausgabe

In der GUI aus Abbildung 20 sind die einzelnen Listener mit Icons gekennzeichnet. Die Handler werden durch ein Kistensymbol repräsentiert, stellvertretend für deren Kategorisierung der Eventlistener. Die PersistentEvents sind mit einem Diskettensymbol gekennzeichnet, welches den Speichervorgang darstellt. Die Uhr vor jedem DeferredChangeListener

symbolisiert den zeitlichen Verzug zwischen Aktivierung und Ausführung der Listener. Der letzte auf dem Screenshot sichtbare Listener ist ein InstantChangeListener und wird aufgrund der verzögerungsfreien Ausführung mit einem zyklischen Pfeil hervorgehoben.

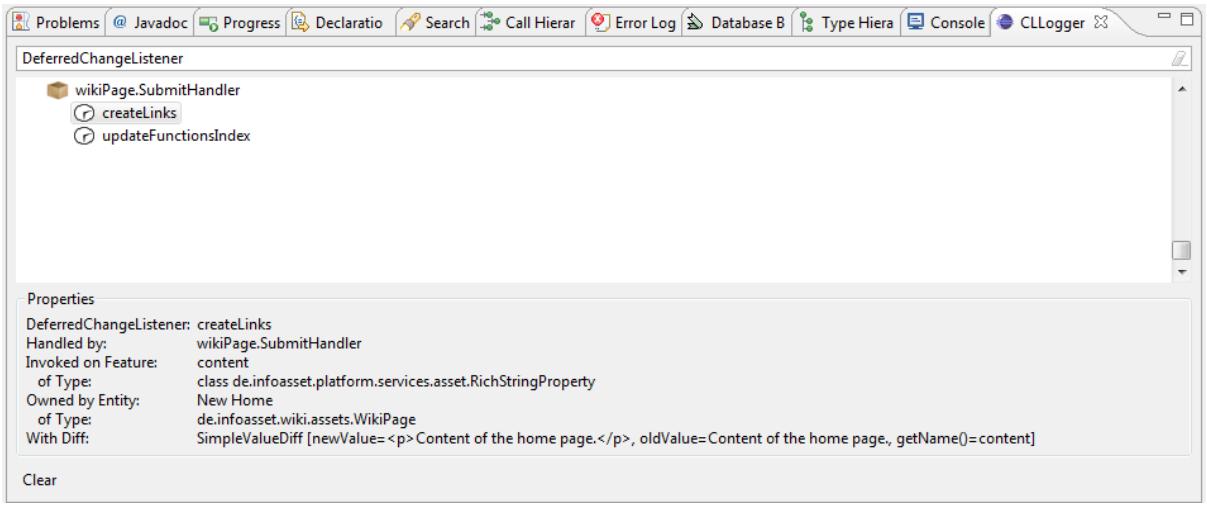


Abbildung 22: CLLoger GUI mit gefilterter Baumdarstellung

Abbildung 22 zeigt das Fenster des CLLoggers im gefiltertern Zustand. Der angezeigte Baum wird hierbei automatisch und live an den eingegebenen Filter angepasst. Es werden alle Knoten angezeigt, die entweder direkt dem Filter entsprechen oder mindestens einer der untergeordneten Eventlistener diesen Filter erfüllt. Um die einzelnen Listener zu filtern wird nicht nur der angezeigte Name verwendet, sondern alle verfügbaren Informationen mit einbezogen. Der Filtermechanismus sucht also auch in den Daten, die als Eigenschaften unterhalb des Baumes angezeigt werden können.

Um die Anwendungsmöglichkeiten des Filters zu erweitern, unterstützt er mit dem Sternsymbol * Wildcards, die für beliebig lange Zeichenketten stehen können. Vor der Aktualisierung des Baumes wird der Filtereingabe standardmäßig eine Wildcard vorangestellt.

Im Beispiel aus Abbildung 22 wurde dieselbe Ausgabe aus Abbildung 20 mit dem Stichwort „DeferredChangeListener“ gefiltert. Sichtbar bleiben in dem Fall lediglich die beiden DeferredChangeListener createLinks und updateFunctionsIndex, sowie deren Handler wikiPage .SubmitHandler. Der Handler selbst genügt zwar nicht dem Filter, jedoch seine untergeordneten Listener und somit bleibt auch er zur Wahrung der hierarchischen Struktur sichtbar. Die Funktionalität der Wildcards ermöglich es beispielsweise, dass das Stichwort „eferredC∗eListener“ zumindest dieselben Listener anzeigt. Potenziell könnten durch diese Platzhalter zusätzliche Listener (z.B. „ReferredChangeListener“) angezeigt werden. Die restlichen Eventlistener werden nicht angezeigt, können jedoch jederzeit durch Verändern der Zeichenkette im Textfeld wieder im Baum sichtbar gemacht werden. Die Datenstruktur bleibt während des Filtervorgangs somit unverändert.

Eine weiter Anforderung an den CLLogger war es, die Belastung des Tricia Systems durch das Tool so gering wie möglich zu halten. Um dies zu gewährleisten, wird sowohl

die Versendung der Listenerdaten, als auch die initíale Kontaktaufnahme via RMI asynchron ausgeführt. Außerdem werden alle gesammelten Daten als einfache Zeichenketten protokolliert, um damit auf unnötige Komplexität zu verzichten. Des Weiteren wird die gesamte Funktionalität des Tools automatisch an- und abgeschalten. Wenn Tricia gestartet wird, ohne dass die Ansicht des CLLogger Plugins aktiviert ist, so wird für die restliche Ausführung des Systems der CLLogger komplett deaktiviert. Dies betrifft nicht nur die Kommunikation mit dem Plugin, sondern auch die Erstellung der Protokolle an sich, also den Code innerhalb von Tricia. In diesem Fall nutzt der CLLogger überhaupt keine Rechenleistung.

Das CLLogger Tool erfüllt somit alle zuvor definierten Anforderungen:

- Es stellt alle getriggerten ChangeListener, PersistentNewEvents und AfterPersistEvents live im Fenster des Eclipse-Plugins dar.
- Die hierarchischen und chronologischen Strukturen werden durch die Repräsentation in einer Baumstruktur erhalten.
- Die oberste Hierarchieebene im Baum gruppiert die angestoßenen Eventlistener nach den korrespondierenden Handlern und strukturiert somit die Masse der Listener.
- Per Eingabe eines Stichwortes kann der Baum live gefiltert werden.
- Sämtliche relevante Daten werden protokolliert und im Plugin angezeigt.
- Der CLLogger belastet die Performance von Tricia nur geringfügig. Wenn das Fenster des CLLoggers geschlossen ist, so wird die Verbindung per RMI gar nicht erst aufgebaut.

Durch die Integration des Tool in die Eclipse IDE, kann der CLLogger nahtlos in den Entwicklungsprozess des Triciakerns toro bzw. Tricias Plugins eingebunden werden. Aktiviert wird das Analysetool ganz einfach durch Öffnen des zugehörigen Fensters. Im folgenden Kapitel 4.3 wird die Arbeit mit dem CLLogger anhand des Komplexitätsbeispiels aus Abbildung 13 erläutert.

4.3 CLLogger Eclipse Plugin - Dokumentation

Um den Nutzen des CLLogger Tools zu verdeutlichen, wird nun das Beispiel für komplexe Abhängigkeiten zwischen Eventlistenern aus Abbildung 13 aufgegriffen und Schritt für Schritt nachvollzogen. Hierfür wird sowohl das Tricia Webinterface, als auch die Eclipse IDE mit aktiviertem CLLogger herangezogen. Dieses Beispiel dient auch als Referenz zur Einbettung des Tools in den Entwicklungsprozess. Innerhalb des relevanten Beispiels wird einer WikiPage ein Anhang zugeordnet und anschließend der Name und dadurch indirekt die URL des übergeordneten Wikis verändert. Dadurch ergeben sich diverse Folgeoperationen, um das System wieder in einen konsistenten Zustand zu überführen. Die anschließende Erläuterung des Beispiels zeigt auf, wie das CLLogger Tool die komplexen Abhängigkeiten dieser Folgeoperationen aufzeigen kann.

Speziell um die Funktionalität der Wikis zu testen, kann Tricia in der wiki−test Konfiguration gestartet werden. Dabei wird zusätzlich das Plugin file aktiviert und grundlegende Benutzergruppen erstellt. Zusätzlich kann über den Testaccount Max Mustermann auf ein automatisch generiertes Wiki namens Home Wiki und dessen einzige WikiPage namens Home zugegriffen werden. Abbildung 23 zeigt das Webinterface, nachdem der Testnutzer eingeloggt ist und die verfügbaren Wikis anzeigen lässt.

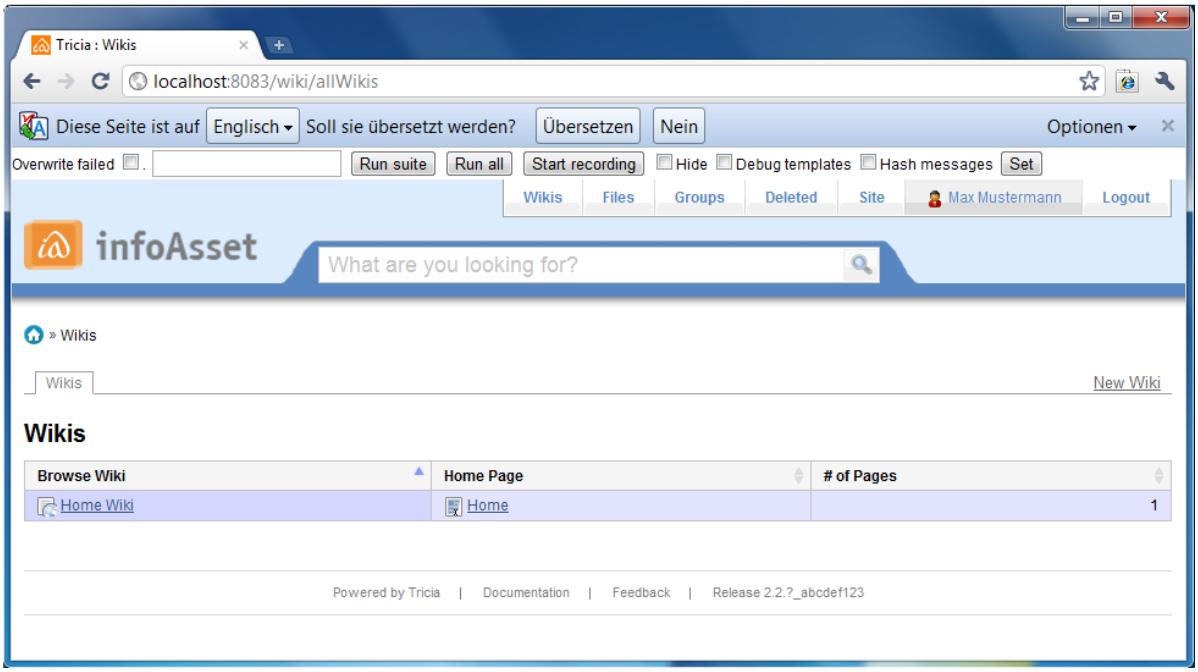


Abbildung 23: Eine Übersicht über die Einträge, welche durch die wiki−test-Konfiguration automatisch generierten werden.

Im Verlauf dieses Start- und Initialisierungsprozesses werden 56 InstantChangeListener, 64 DeferredChangeListener, 24 AfterPersistEvents und 15 PersistentNewEvents getriggert. Diese Zahlen lassen erahnen, dass augenscheinlich simple Aktionen sehr viele Eventlistener anstoßen können.

Im nächsten Schritt steht der Upload eines Anhangs an. Im vorliegenden Beispiel wird die Datei test.xml hochgeladen und in der bereits automatisch generierten WikiPage Home referenziert. Dieser Vorgang ist in Abbildung 24 als Screenshot dargestellt. Im Hinblick auf den simplen Vorgang, einen Anhang hochzuladen, ist auch hier wieder die Anzahl der getriggerten Listener beeindruckend. Es werden allein während diesem Prozess 46 InstantChangeListener, 37 DeferredChangeListener, 19 AfterPersistEvents und 6 PersistentNewEvents angestoßen. Da der Fokus der Analyse auf der Modifikation der Wiki-URL liegt, werden auch diese Listener mit der Schaltflächer Clear aus der Baumdarstellung des CLLogger Tools entfernt. Diese Schaltfläche kann die Komplexität der nachgelagerten Analyse wesentlich reduzieren. Indem der Entwickler direkt vor der relevanten Interaktion mit dem System durch Betätigung von Clear alle vorhandenen Knoten aus dem Baum löscht, wird die resultierende Anzahl der angezeigten Eventlistener deutlich verringert und damit deren Analyse vereinfacht.

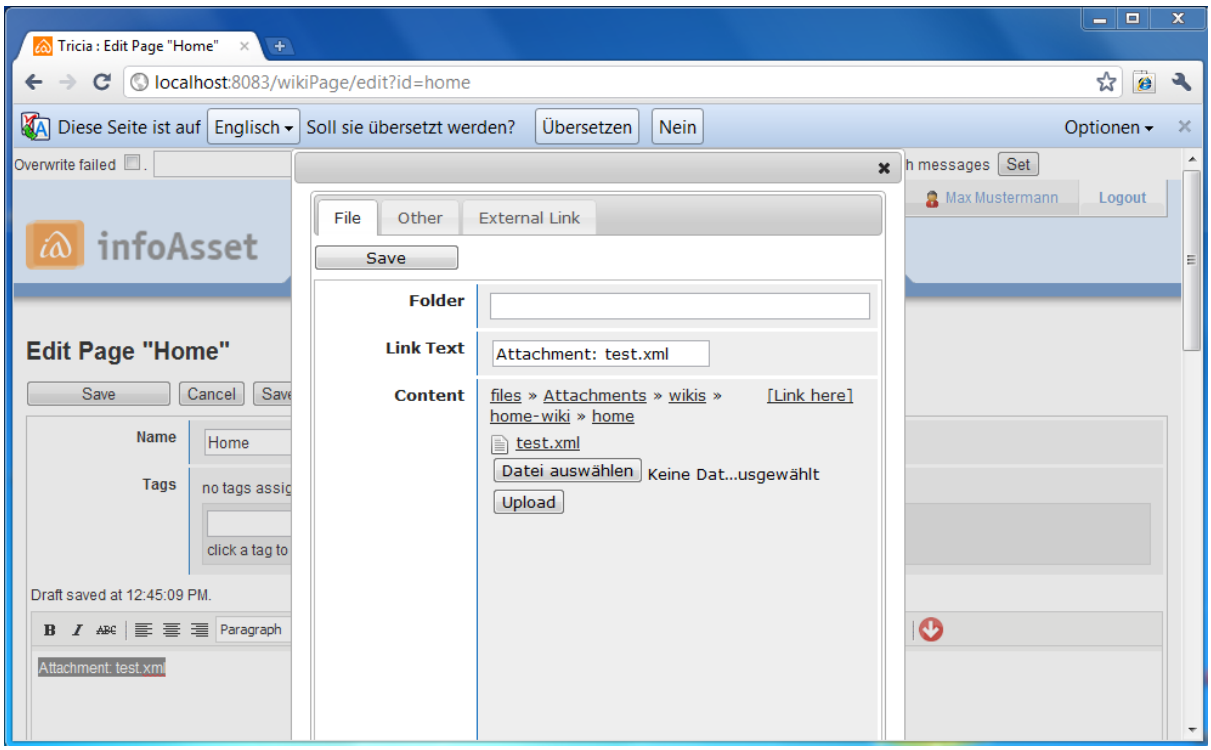


Abbildung 24: Upload eines Attachments im Webinterface.

Damit ist nun die Basis für das Komplexitätsbeispiel aus Abbildung 13 geschaffen. Es existiert ein Wiki, welches eine WikiPage enthält und in dieser WikiPage ist ein Anhang referenziert. Im Folgenden soll nun, wie in Abbildung 25 dargestellt, die URL des Wikis verändert werden. Dabei stellt sich die Frage, warum der Nutzer die URL eines Wikis von Hand ändern sollte. Wenn er jedoch den Namen des Wikis modifiziert, dann ist es naheliegend, dass er in der Adresszeile des Browsers den neuen Namen erwartet und außerdem voraussetzt, dass auf die URL mit dem neuen Namen verlinkt werden kann. Die automatische Anpassung der URL bei jeder Namensänderung ist in Tricia momentan deaktiviert, da sie unter Umständen extrem viele Eventlistener anstoßen und damit die Performance erheblich beeinträchtigen würde. Wie im Bezug auf Tabelle 6 bereits erläutert wurde, muss eine solche URL-Änderung sowohl an die in diesem Wiki enthaltenen WikiPages, als auch an deren Attachments weitergereicht werden. Diese Aufgabe ist je nach Anzahl der Wikipages und Attachments so umfangreich, dass der verantwortliche ChangeListener bereits standardmäßig als BatchDeferredChangeListener asynchron angestoßen wird.

In Abbildung 25 sind beide Attribute modifiziert. Das Feature name wurde von „Home Wiki“ zu „New Home Wiki“ abgeändert, dessen urlName von „home-wiki“ zu „new-home-wiki“. Nach Betätigung der Schaltfläche Save werden zahlreiche InstantChangeListener, DeferredChangeListener und PersistentEvents angestoßen. In Abbildung 26 sind die getriggerten Eventlistener im CLLogger Tool dargestellt. Insgesamt wurden nur duch diese eine Aktion 22 InstantChangeListener, 19 DeferredChangeListener, 10 AfterPersistEvents und 2 PersistentNewEvents ausgeführt. Dabei sollte jedoch noch einmal über den Umfang des Wikis reflektiert werden. Es handelt sich hierbei lediglich um eine einzige WikiPage mit einem einzigen Anhang und keinerlei weiteren Referenzen auf die Objekte. Dadurch wird

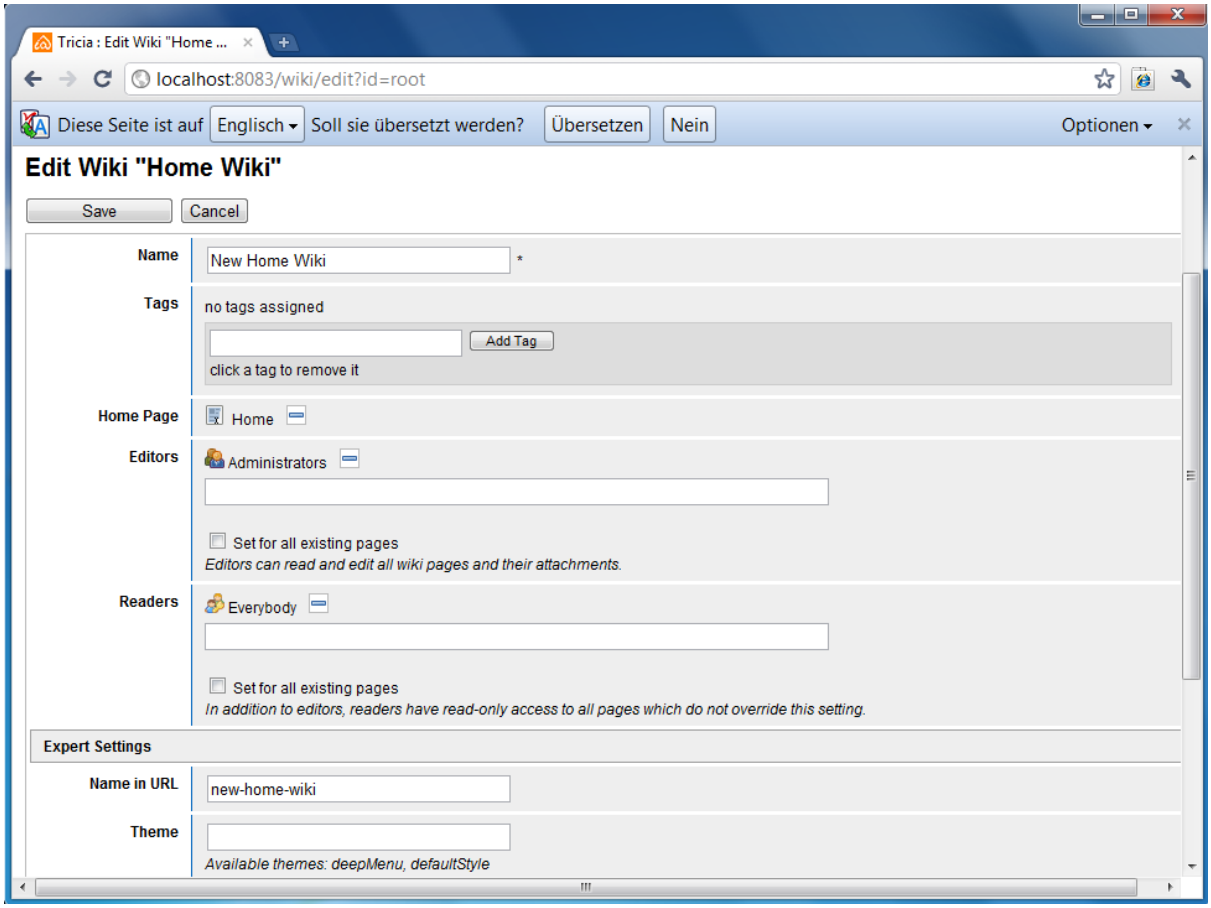


Abbildung 25: Modifikation der URL eines Wikis im Webinterface.

schnell klar, dass die Anzahl der Eventlistenern in Produktivsystemen um ein Vielfaches über der vorliegenden Anzahl im isolierten Experiment liegen. Die logische Konsequenz ist daher, dass die Abhängigkeiten unter den einzelnen Listenern genau analysiert werden sollten, um die Gesamtperformance des Systems nicht unnötig zu beeinträchtigen. Zusätzlich sollte speziell darauf geachtet werden, ob Listener mehrfach innerhalb einer Operation getriggert werden.

In der folgenden Auflistung werden einige der ausgeführten Eventlistener genauer analysiert. Dabei referenziert die Nummer jeweils den markierten Listener in Abbildung 26.

1. An dem Feature name ist ein InstantChangeListener mit dem Namen updateUrlName definiert. Dieser entfaltet im vorliegenden Beispiel jedoch keine Wikung, da nur bei neuen Objekten die URL sofort dem vergebenen Namen angepasst wird.

2. Die beiden DeferredChangeListener AdaptUrlForThisAsset sind nicht innerhalb der anonymen Klasse in Wiki oder WikiPage definiert. Dieser Eventlistener ist ein Standardlistener jeder UrlNameProperty. Im ersten Fall wird die URL des Wikis verändert und dazu auch das korrekte Diff-Objekt übergeben. Um diese Änderung der URL an die WikiPage weiterzureichen, wird jedoch nicht die UrlNameProperty der WikiPage verändert, sondern der Listener AdaptUrlForThisAsset am Feature urlName „manuell„

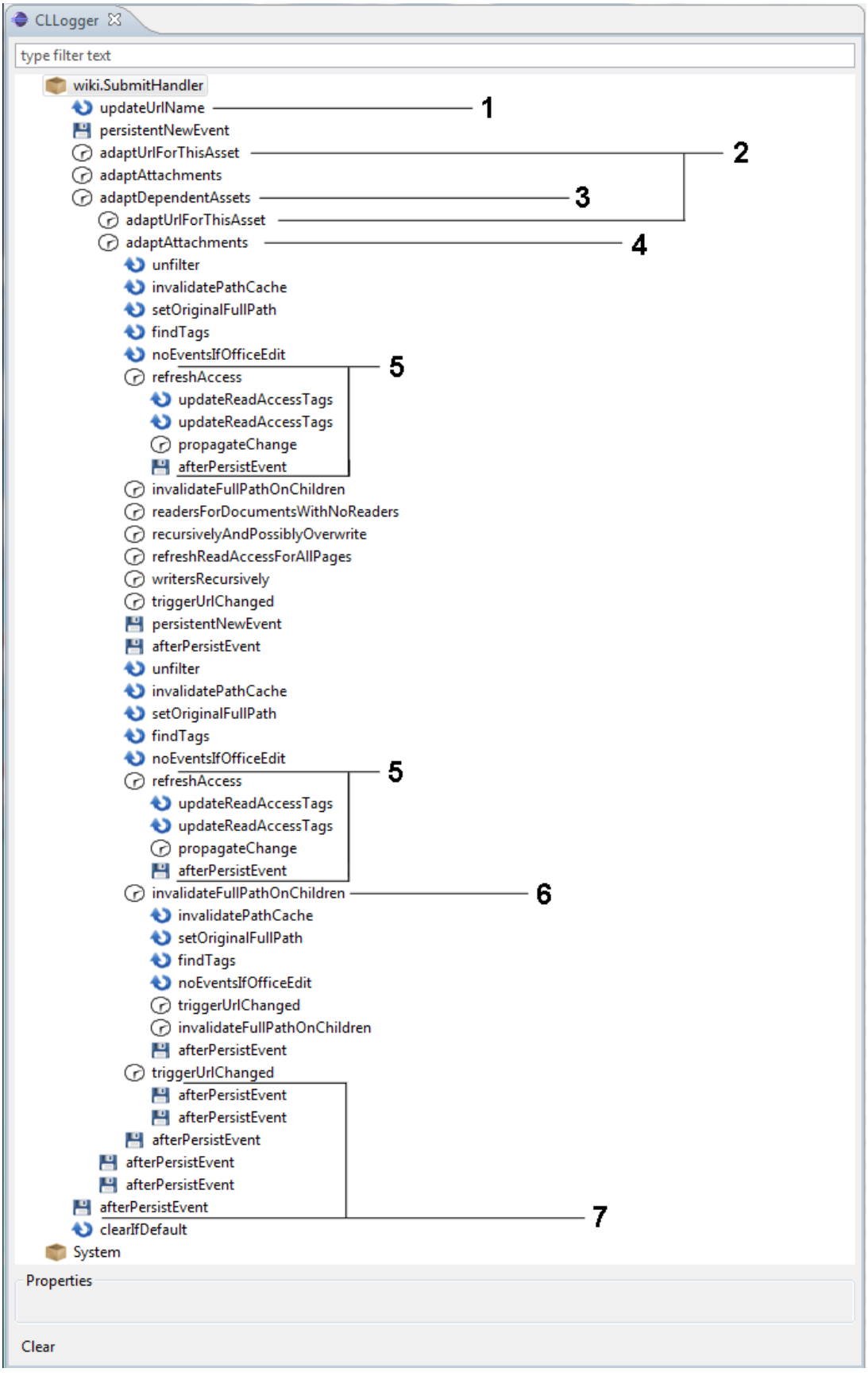


Abbildung 26: CLLogger Ansicht nach Modifikation der Wiki-URL im Webinterface.

per triggerChangeListener() mit einem NoChangeDiff[] angestoßen. Innerhalb dieses Listeners wird über einen Forwarder auf die neue, richtige URL zugegriffen und das Feature schlussendlich auch angepasst. Diese Vorgehensweise erscheint auf den ersten Blick sehr umständlich und nicht besonders intuitiv. Aufgrund dessen sollte sie noch einmal analysiert und überdacht werden.

3. Der DeferredChangeListener AdaptDependentAssets ist verantwortlich für die Vorgehensweise aus Punkt 2. Innerhalb des Listeners werden alle WikiPages aus dem aktiven Wiki aktualisiert. Allerdings geschieht dies über den manuellen Aufruf der Listener am UrlNameProperty. Deshalb sollte dieser Eventlistener in die Analyse von Punkt 2 einbezogen werden.

4. Der DeferredChangeListener AdaptAttachments wird in der selben Weise aufgerufen, wie AdaptDependentAssets. Auch er ist direkt in der Klasse UrlNameProperty definiert und wird manuell mit einem NoChangeDiff[] gestartet.

5. Die RefreshAccess-Blöcke werden auf Basis der Directory-Objekte angestoßen. Definiert sind die Listener jedoch in Path, der Vaterklasse von Directory. Die Listener sollen die Leserechte für den jeweiligen Ordner aktualisieren. Dabei wird jedoch der InstantChangeListener updateReadAccessTags in beiden Fällen doppelt aufgerufen. Im ersten Durchlauf werden alle Leserechte gelöscht und im zweiten wird versucht, passende Leserechte zuzuweisen. Auch dieses Vorgehen erscheint nicht direkt intuitiv. Es sollte analysiert werden, ob diese Aktualisierung nicht in einem Schritt durchführbar ist.

6. Der Eventlistener InvalidateFullPathOnChildren ist ein BatchDeferredChangeListener und am Feature fullPath der Klasse Path definiert. Er aktualisiert den fullPath aller Objekt, die diesem Path als Kinder zugeordnet sind. Innerhalb dieses Listeners wird unter anderem der DeferredChangeListener triggerUrlChanged aufgerufen. In diesem Fall reflektiert er durch die URL-Änderung an der Datei test.xml. Damit ist die Weitergabe der initialen Modifikation der Wiki-URL am letzten Objekt angekommen.

7. Schlussendlich werden alle AfterPersistEvents der zuvor aufgespannten Hierarchie abgearbeitet. Dabei handelt es sich um die AfterPersistEvents aus den Mixins Searchable und Hybrid. Innerhalb des AfterPersistEvents von Searchable beispielsweise wird das Suchmodul Lucene benachrichtigt. Die Listener werden für das Attachment, für ein Directory, für die WikiPage und für das Wiki aufgerufen. Allerdings ist auffällig, dass die Listener für die WikiPage doppelt aufgerufen werden. Auch dieser Prozess sollte somit analysiert und überdacht werden.

Diese Analyse zeigt wieder, wie komplex eine augenscheinlich einfache Änderung der URL eines einzigen Wikis werden kann. Dazu müssen sich noch nicht einmal viele WikiPages oder Attachments in diesem Wiki befinden. Die nähere Betrachtung einiger ausgewählter Stellen im Prozess zeigt auf, dass im Handling der ChangeListener und PersistentEvents durchaus noch Optimierungspotenzial vorhanden ist. Die Schwierigkeit besteht jedoch in der genauen Erfassung aller Einflussfaktoren und Abhängigkeiten. Das CLLogger Tool hilft, diese Abhängigkeiten aufzuzeigen und erleichtert es somit dem Entwickler, ein umfassendes

Verständnis der Funktionalität und Eigenheiten des in Tricia integrierten Eventlistenersystems zu entwickeln.

Technische Dokumentation: Im Folgenden wird zur technischen Dokumentation ein Komponentendiagramm des CLLogger Tools vorgestellt und erläutert. Dieses Modell kann als Verständnisgrundlage für eine mögliche Weiterentwicklung des Tools genutzt werden. Das Komponentendiagramm ist, korrespondierend zu der Komponente in toro und dem Plugin selbst, auf die Abbildungen 27 und 28 aufgeteilt.

Eine zentrale Funktion innerhalb der toro-Komponente (Abbildung 27) des CLLoggers erfüllt die Klasse ChangeListenerLogger. Sie enthält den Schalter um die Protokollfunktion ein- und auszuschalten (clLoggingSwitch), sowie eine getrennte Option, um die Protokollierung von InstantChangeListenern zu aktivieren und deaktivieren (instantChangeListenerSwitch). Diese Klasse ist auch das Bindeglied zum toro-Code. Die statischen Methoden startLogging (...) und endLogging generieren dabei neue Protokolleinträge.

Diese Einträge sind allesamt vom Typ ChangeListenerLoggerEventImpl. Je nach Eventlistener wird ein Objekt der Klasse ChangeListenerLoggerPersistEvent oder Objekt der Klasse ChangeListenerLoggerStartEventImpl erzeugt. Beides sind Unterklassen von der Klasse ChangeListenerLoggerEventImpl. Nachdem der Code eines Listeners ausgeführt wurde, wird ein Objekt der Klasse ChangeListenerLoggerEndEvent generiert. All diese Eventklassen werden über korrespondierende Interfaces zugänglich gemacht, da das Plugin schließlich eine Vorlage benötigt, um diese Objekte verarbeiten zu können. In diesen Interfaces sind alle Methoden zum Auslesen der relevanten Daten definiert. Deshalb wurde in den Implementierungsklassen auf eine zusätzliche Auflistung der Attribute und Methoden verzichtet.

Jeder ChangeListenerLoggerEvent enthält sowohl den Namen des getriggerten Eventlisteners, als auch den Namen des zuständigen Handlers. Die ChangeListenerLoggerStartEvents speichern zusätzlich den Namen des Features, dessen Typ und den Namen vom umschließenden Entity inklusive Typbezeichnung. Falls der Listener in einem Mixin ausgelöst wurde, so werden auch die Daten über dieses Mixin aufgenommen. Schlussendlich wird noch das Diff-Objekt des Listeners protokolliert und ob dieser deferred oder instant ist. Ein ChangeListenerLoggerPersistEvent hingegen enthält zusätlich zu den Standardinformationen eines ChangeListenerLoggerEvents lediglich die Daten über das Entity und das Mixin, in welchem es getriggert wurde. Ein ChangeListenerLoggerEndEvent enthält zwar keine zusätzlichen Daten, ist aber wichtig um die chronologische Struktur der Listenerausführung rekonstruieren zu können.

Ein per startLogging (...) erzeugter Event wird via inserLoggingEvent(...) in die queue eingefügt. Sollte noch kein Thread gestartet sein, um die queue wieder zu leeren, dann wird eine Instanz der Klasse ChangeListenerLogger erstellt und als eigener Thread loggerThread gestartet, um die Ausführung des laufenden Tricia-Systems so wenig wie möglich zu verzögern. Deshalb muss die Klasse das Interface Runnable implementieren. Der loggerThread entnimmt in der Methode run() die Elemente aus der queue und benachrichtigt alle observer davon. Jedes Element wird schließlich auch per RMI an das CLLogger Plugin übermittelt. Die Klasse für den Konsolenoutput aus vorangegangenen Screenshots wurde beispielsweise

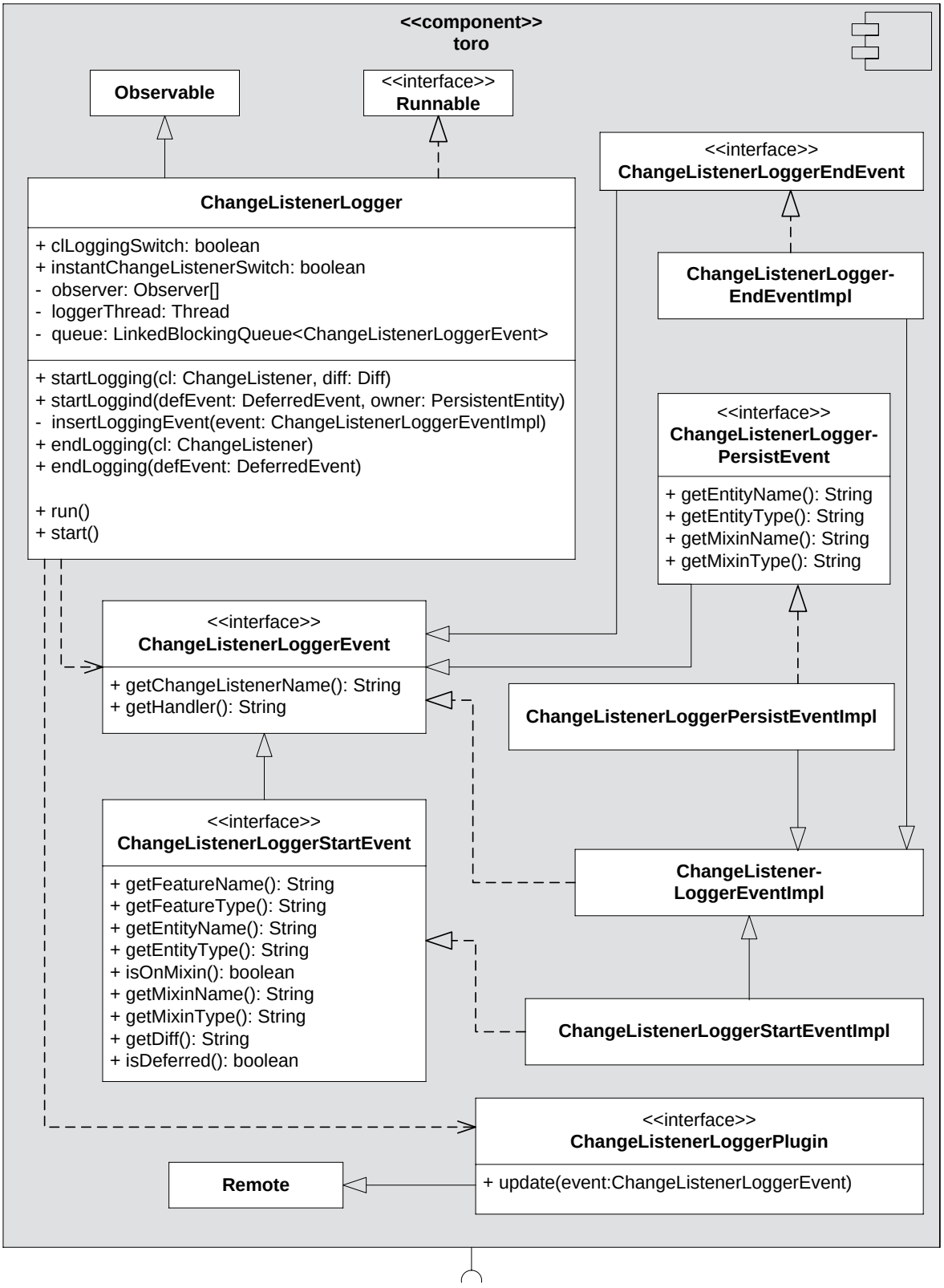


Abbildung 27: Die toro Komponente im Komponentenmodell mit Abbildung 28.

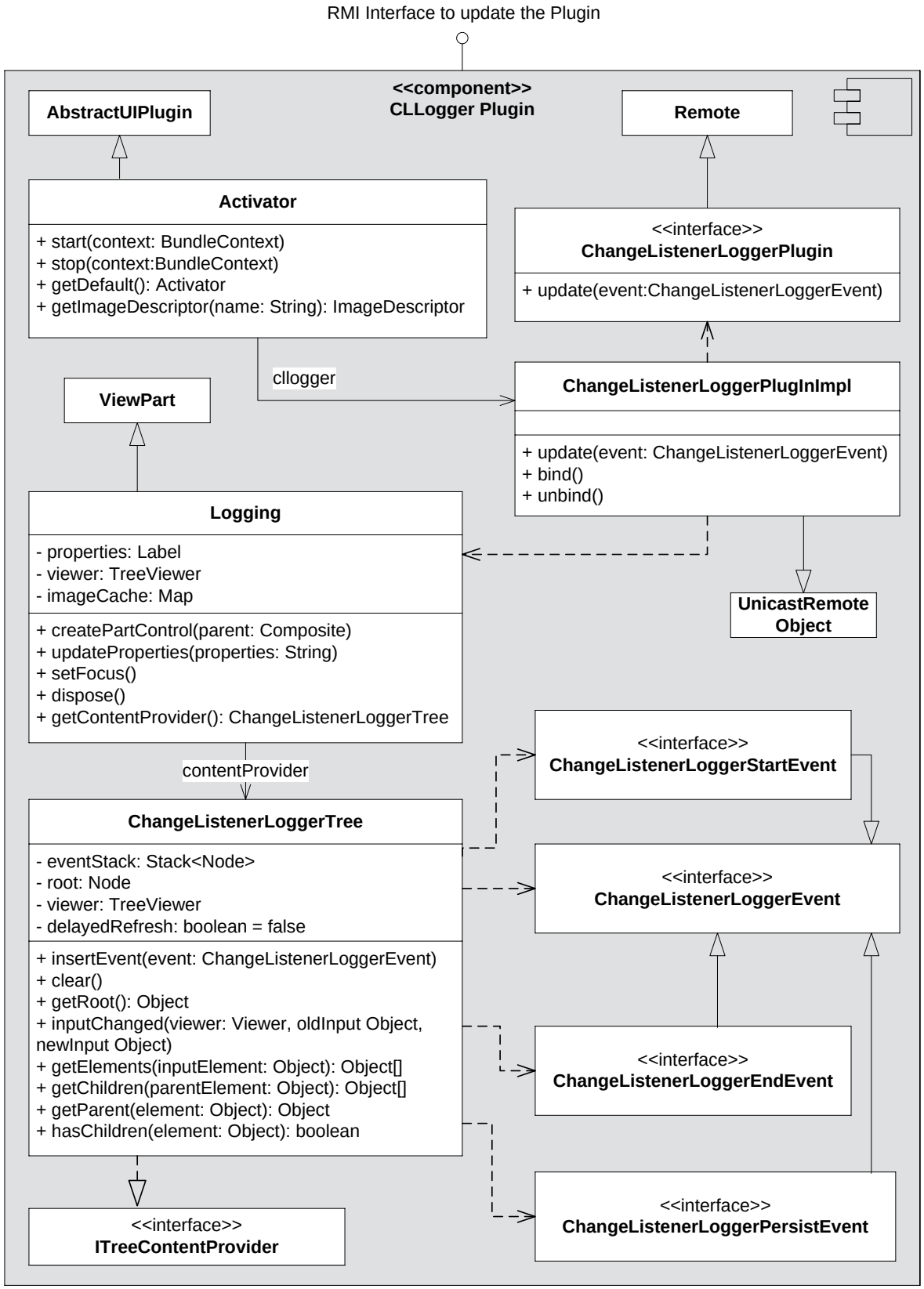


Abbildung 28: Die Plugin Komponente im Komponentenmodell mit Abbildung 27.

als Observer registriert. Wenn die queue leer ist, so wartet der loggerThread ganz einfach auf neue Elemente.

Zur Übermittlung des ChangeListenerLoggerEvents wird per RMI eine Remotereferenz auf das ChangeListenerLoggerPluginImpl-Objekt im Plugin erzeugt. Dafür ist dessen Interface ChangeListenerLoggerPlugin, welches von der Klasse Remote erbt, notwendig. Durch dieses Interface kann anschließend die Methode update(...) auf der Remotereferenz aufgerufen und somit die Baumdarstellung im Plugin aktualisiert werden.

In der Plugin-Komponente (28 ist nicht nur dieses Interface enthalten, sondern auch dessen Implementierung. Die Klasse ChangeListenerLoggerPlugInImpl stellt quasi den RMI Server dar, der den Service RMIUpdate über die RMI Registry zugänglich macht. Per bind() und unbind() kann dieser Service aktiviert oder deaktiviert werden. Generiert wird dieses UnicastRemoteObject vom Activator des Plugins während dem Startvorgang. Dieser Activator stellt zusätzlich zu den Standardmethoden die Funktion getImageDescriptor(...) bereit. Über sie können Bilder und Icons komfortabel geladen werden.

Das sichtbare Fenster des CLLogger Plugins ist in der Klasse Logging implementiert, da diese von der Klasse ViewPart erbt. In ihr sind das Label zur Anzeige von Listenereigenschaften, ein TreeViewer, sowie ein imageCache definiert. Der TreeViewer gehört zum FilteredTree und über den imageCache werden die Icons der Listener geladen. Die Methode createPartControl(...) wird automatisch zur Initialisierung des Plugin-Fensters ausgeführt. In ihr wird der FilteredTree, sowie ein ChangeListenerLoggerTree erstellt und als contentProvider eingesetzt. Zusätzlich wurden eigene LabelProvider, SelectionListener und PatternFilter definiert, um das Verhalten der Baumdarstellung an die Anforderungen anzupassen. Per updateProperties(...) kann die Eigenschaftsanzeige im Plugin-Fenster aktualisiert werden und innerhalb der dispose()-Methode wird der imageCache geleert und der Service per unbind() deaktiviert.

Die Datenstruktur, welche die Baumdarstellung des FilteredTree mit Daten versorgt, ist in der Klasse ChangeListenerLoggerTree implementiert. Sie implementiert dazu das Interface ITreeContentProvider und kann durch die Interfaces aller ChangeListenerLoggerEvents auch mit den per RMI übertragenen Objekten arbeiten. Um die Baumstruktur konsistent aufbauen zu können, verwaltet der ChangeListenerLoggerTree einen Stack. Auf diesen Stack werden per insertEvent(...) neue Knoten gelegt. Außer die Handler-Knoten enthält jeder dieser Knoten einen ChangeListenerLoggerEvent. Die Handler-Knoten werden dabei automatisch hinzugefügt. Die Methoden inputChanged(...), getElements(...), getChildren(...), getParent(...) und hasChildren(...) sind für die Versorgung der Baumdarstellung erforderlich und müssen aufgrund des ITreeContentProviders überschrieben werden. Die Schaltfläche Clear am unteren Rand des Plugin-Fensters kann quasi als Reset für die Datenstruktur der Baumdarstellung genutzt werden. Dies hat zur Folge, dass weder Tricia, noch das Plugin neu gestartet werden muss, falls die Darstellung der Changelistener und PersistentEvents zu umfangreich wird. Mit dieser Schaltfläche ist es möglich, dass einzig und allein die Listener der momentan relevanten Operation dargestellt werden.

5 Ausblick

Für die Zukunft ist natürlich interessant, in wie weit das CLLogger Tool in der alltäglichen Entwicklung von Tricia eingesetzt wird und auf welche Weise es sich in den Workflow der Entwicklung und des Debugging integrieren lässt. Um die Nutzung des Tools zu analysieren ist ein ausgiebiger Praxistest mit mehreren Tricia-Entwicklern unerlässlich. Erst nach einiger Zeit im Praxiseinsatz kann wirklich bestimmt werden, welche Informationen im Tool einen reellen Mehrwert bieten und welche Daten die Darstellung eher unübersichtlich werden lassen. In dieser Hinsicht ist noch einiges an Optimierungspotenzial vorhanden. Während der Einführungsphase können Optimierungen dieser Art jedoch mit äußerst geringem Aufwand zeitnah integriert werden.

Weitere Ideen zur Evolution des Tool beziehen sich auf die graphische Benutzeroberfläche. Da nur eine sehr begrentzte Anzahl an verschiedenen Eventlistener-Arten existiert, könnte die Selektion der dargestellten Listener auf per Dropdown-Liste realisiert werden. Diese Funktionalität kann momentan nur durch die Eingabe des Eventlistener-Typs als Filterstichwort genutzt werden. Einen echten Mehrwert würde die Dropdown-Liste jedoch dann bieten, wenn zusätzlich zur Selektion eines bestimmten Listener-Typs, mit einem spezifischen Stichwort gefiltert werden kann. Damit könnte die Auswahl an dargestellten Listenern weiter eingeschränkt und mit dem Umfang auch die Komplexität der Baumdarstellung reduziert werden. Diese Funktionalität kann mit relativ begrenztem Aufwand in den CustomPatternFilter integriert werden.

Eine grundlegende Idee zur Optimierung des Workflows stellt die Verknüpfung von im Baum dargestellten Eventlistenern mit den korrespondierenden Stellen im Code dar.

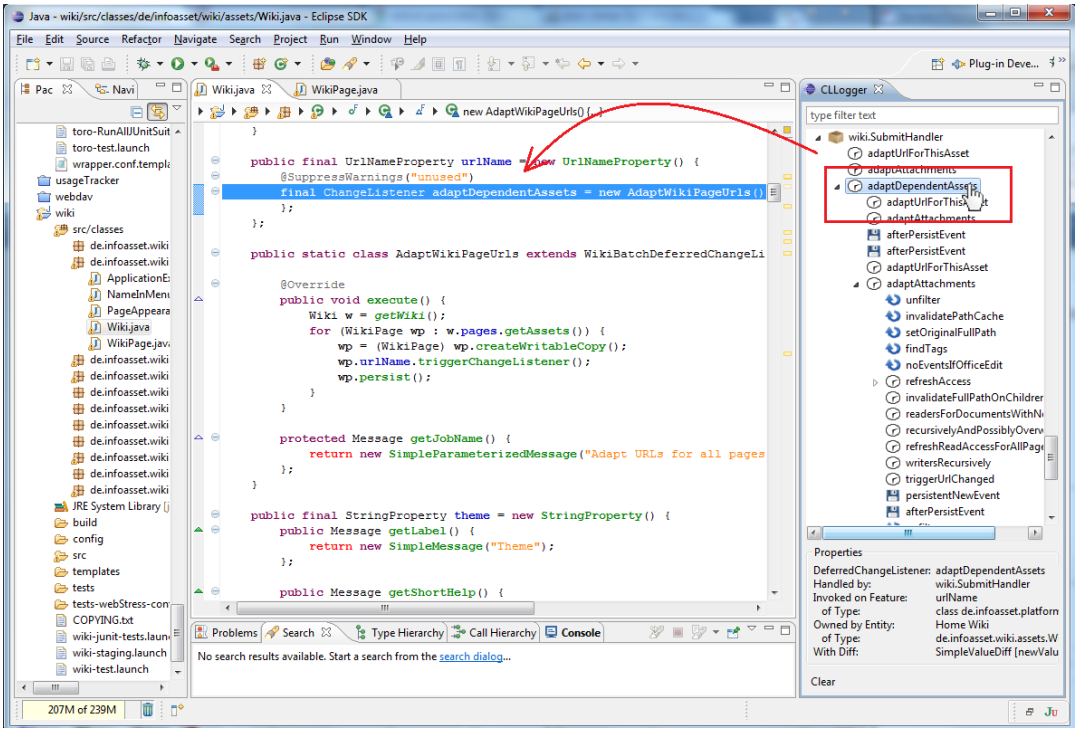


Abbildung 29: Feature Request: Selektion korrespondierenden Tricia Sourcecodes

Wie in Abbildung 29 dargestellt, könnte per Doppelklick auf einen Knoten im Baum die zugehörige Deklaration des Listeners in der Eclipse IDE aufgerufen werden. Im Beispiel wurde durch einen Doppelklick auf den DeferredChangeListener adaptDependentAssets die korrespondierende Klasse Wiki aufgerufen und an die relevante Stelle im Code gescrollt. Diese Funktionalität könnte zwar das CLLogger Tool weitaus besser im Entwicklungsworkflow integrieren, allerdings ist diese Aufgabe nicht trivial und mit einigem Entwicklungsaufwand verbunden. Daher stellt sich logischerweise die Frage, ob die Funktion den zusätzlichen Aufwand rechtfertigt. Auch zur Beantwortung dieser Frage kann die Analyse der Praxisphase aufschlussreich sein.

Zu guter Letzt birgt die asynchrone Ausführung der BatchDeferredChangeListener ein Risiko für die chronologische Integrität der ChangeListener-Darstellung im CLLogger Tool. Um die Eventlistener eindeutig zum triggernden Prozess zuordnen zu können, müssen möglicherweise zusätzliche Informationen gespeichert werden. Denkbar wäre eine automatische Identifikatorvergabe bei der Einleitung dieser asynchronen Listener. Eine ähnliche Frage stellt sich bei der Zuordnung von einzelnen Listenern zu verschiedenen Nutzern des Systems. Im Rahmen dieser Bachelorarbeit wurden einige Rahmenbedingungen bewusst vereinfacht, sodass auf Basis des nun vorhandenen CLLoggers in zukünftigen Entwicklungsstufen auch zusätzliche Komplexitätsfaktoren berücksichtigt werden können.

Literatur

[Barik, 2006] Barik, T. (2006). *Introducing the Java Content Repository API.* Solo.

[Bauer & King, 2004] Bauer, C. & King, G. (2004). *Hibernate in Action (In Action series).* Manning Publications Co.

[Büchner et al., 2010] Büchner, T., Matthes, F., & Neubert, C. (2010). *Data Model Driven Implementation of Web Cooperation Systems with Tricia.* Technical report, Technische Universität München, Institute for Informatics.

[Eclipse, 2008] Eclipse (2008). *Eclipse Documentation - The JFace UI framework.* Eclipse Foundation.

[Eclipse, 2010] Eclipse (2010). *Eclipse Documentation - SWT Widgets.* Eclipse Foundation.

[Gamma et al., 2007] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (2007). *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley.

[Heinisch et al., 2007] Heinisch, C., Müller-Hofmann, F., & Goll, J. (2007). *Java als erste Programmiersprache.* Teubner.

[InfoAsset, 2010] InfoAsset (2010). *Tricia Developer Documentation.* InfoAsset AG.

[King, 2010] King, G. (2010). *Hibernate API.* Red Hat, Inc.

[King et al., 2010] King, G., Bauer, C., Andersen, M. R., Bernard, E., Ebersole, S., & Ferentschik, H. (2010). *Hibernate Reference Documentation*, 3.6.0.cr2 edition.

[McCauley, 2008] McCauley, T. (2008). *Introduction to event handling in ActionScript 3.0.* Adobe.

[Nuescheler et al., 2009] Nuescheler, D., Piegaze, P., Abnous, R., Anderson, T., et al. (2009). *Content Repository for Java Technology API 2.0 Specification.* Day Management AG, 2.0 edition.

[Nuescheler et al., 2005] Nuescheler, D., Piegaze, P., Anderson, T., Bell, G., Clemm, G., Choy, D., Collins, J., Guggisberg, S., Mazzocchi, S., Myers, J., Owen, J., Pfeifroth, F., Pitfield, D., Reed, C., Spivak, V., & Victor, D. B. (2005). *Content Repository API for Java Technology Specification.* Day Management AG, 1.0 edition.

[Olofson & Shirer, 2010] Olofson, C. & Shirer, M. (2010). *Worldwide Database and Data Integration Software Tracker.* IDC, 1h 2010 edition.

[Patricio, 2010] Patricio, A. (2010). *Hibernate Supported Databases.* JBoss Enterprise, Red Hat.

[Pixley, 2000] Pixley, T. (2000). *W3C Recommendation - Document Object Model Events.* Netscape Communications Corp.

[Sceppa, 2002] Sceppa, D. (2002). *Microsoft Ado.Net (Core Reference)*. Microsoft Press.

[Slavic, 2010] Slavic, S. (2010). *Hibernate Product Evaluation FAQ*. JBoss Enterprise, Red Hat.