Ressourcen: Die Basis der Arbeit

Konstantin Grupp

1 Einleitung

1.1 Motivation und Zielsetzung

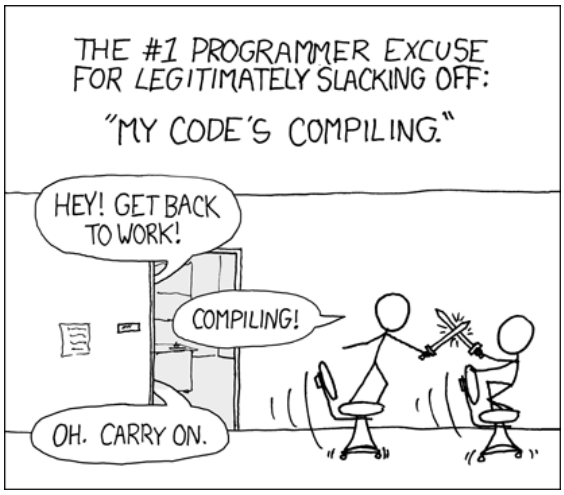

Abbildung 1: Compiling [8]

Eine der ältesten Ausreden von Programmierern ist es, dass sie nicht arbeiten können, weil ihr PC gerade Quellcode kompiliert. Diese Ausrede kommt nicht von ungefähr, da der Compile-Prozess früher stets manuell angestoßen werden musste und auch einiges an Zeit verschlang. Mit der Ressourcenverwaltung die Eclipse einführt, verändert sich in dieser Richtung vieles. Dies ist allerdings nur einer von vielen Gründen, warum es sich lohnt das Eclipse-Plugin `org.eclipse.core.resources` anzuschauen. Nahezu jedes Eclipse-Plugin verwendet es an der ein oder anderen Ecke. Es spielt dabei nicht nur eine wichtige Rolle beim Compile-Prozess, sondern bietet auch eine gute Möglichkeit Änderungen im Workspace zu überwachen. Mittels Abstraktion über Datei und Ordnerpfade, lässt sich das Plugin betriebssystemübergreifend einsetzten und kann externe Dateisysteme per Remote ansprechen. Dafür verwendet es verschiedene Softwaredesign Pattern wie das Visitor- und Proxypattern.

Anhand eines kleinen Beispielplugins wollen wir demonstrieren, wie einfach und nützlich sich das Ressourcen-Plugin im Praxistest verhält. Dazu schreiben wir einen Builder, der XML-Dateien (wie in Abbildung 2a) mittels XSLT formatieren kann und die Ergebnis in HTML-Dateien (wie in Abbildung 2b) abspeichert. Dabei wollen wir erreichen, dass direkt nach der Änderung einer Datei Eclipse unseren Builder benachrichtigt, der die Datei neu kompiliert.

1.2 Gliederung

Dieses Tutorial gliedert sich in vier Bereiche: Zunächst erfolgt eine Erläuterung der grundlegenden Interfaces und Klassen aus `org.eclipse.core.resources` sowie die Idee dahinter. Es erläutert hier wie Eclipse mittels des Ressourcen-Plugins Dateien verwaltet

```
<MODULE>
  <VORLESUNG>[...]</VORLESUNG>
  <SEMINAR>
    <ID>INF9999</ID>
      <TITEL>Technik und Konzepte der
           Eclipse Plattform</TITEL>
      <BESCHREIBUNG>Die Eclipse
           Plattform [...]</BESCHREIBUNG
           >
      <THEMEN>
        <THEMA>Ressourcen: Die Basis der
             Arbeit</THEMA>[...]
        <LINK url="https://ovidius.uni-
             tuebingen.de/ilias3">Quelle
             </LINK>
      </THEMEN>
  </SEMINAR>
</MODULE>
```

(a) Gekürzte XML-Quelldatei

Inhaltsangabe

1. Kryptologie
2. Technik und Konzepte der Eclipse Plattform
3. The MonetDB Column Store Database System

Seminare und Vorlesungen

Vorlesung: Kryptologie

Seminar: Technik und Konzepte der Eclipse Plattform

Beschreibung

Die Eclipse Plattform bietet ein reichhaltiges Ökosystem von >> vorhandenen Komponenten. Dies ermöglicht es, neue Plugins für die Eclipse IDE, aber auch eigenständige Rich-Client Applikationen auf der Basis der Eclipse Infrastruktur effektiv zu erstellen. Ziel des Seminars ist es, dass alle Teilnehmer/innen einen praxisnahen Überblick über die Entwicklung mit der Eclipse Plattform erhalten. Die Themen sind dabei so ausgewählt, dass sie jeweils spezifische Teilaspekte abdecken, die zusammen ein Gesamtbild der Eclipse-Entwicklung zeichnen. Übergreifend werden die software-technischen Voraussetzungen und Prinzipien für erweiterbare und modulare Software behandelt.

Themen

1. Ressourcen: Die Basis der Arbeit
2. Plugins: Die Basis für Erweiterbarkeit
3. Editoren: Änderung von Ressourcen
4. Views: Zusatzinformationen für Arbeitsprozesse

Quelle

(b) HTML-Ergebnis

Abbildung 2: XML-Datei mit entsprechend formatierter HTML-Datei

und welche Details bei der Verwendung zu beachten sind. Abschnitt 3 begeht einen kurzen Exkurs zu XSL-Transformationen. Mit diesem Hintergrundwissen gerüstet, erarbeitet Abschnitt 4, wie wir einen eigenen Builder in Eclipse integrieren können und welche Fehlerquellen wir dort beachten müssen. Abschließend gibt Abschnitt 5 einen Ausblick auf weitere interessante Themenpunkte.

2 Von IWorkspace bis zu IFile

Die Dokumentation des `org.eclipse.core.resources`-Plugins [4] wirft uns 42 Interfaces und 7 Klassen an den Kopf. Dieser Teil beschäftigt sich deshalb erst mit den größeren Zusammenhängen in den ersten drei Abschnitten. Zuallererst gibt Abschnitt 2.1 einen Überblick über die Ressourcen und erklärt was diese sind. Abschnitt 2.2 gibt einen Abriss darüber wie Workspace, Projekte, Ordner und Dateien miteinander verknüpft sind. Der letzte Abschnitt erläutert genauere Details einzelner Komponenten, die entweder unser Tutorial berücksichtigen muss oder für die Verwendung oft relevant sind.

2.1 Überblick

Jedes Betriebssystem besitzt seine eigene Form, wie der Anwender auf Dateien zugreifen kann. In aller Regel liegen Dateien innerhalb von Ordnern, die diese strukturieren. Diese Struktur und seine Dateien abstrahiert Eclipse durch Ressourcen. Diese ermöglichen dem Anwender den Zugriff auf das Dateisystem. Das Tutorial befasst sich nur mit dem Zugriff auf Dateien die im Betriebsystem verfügbar sind[1]. Eclipse arbeitet nur über Dateien des Workspaces. Dieser bildet ein eigenes Dateisystem. Lokale Ordner und Dateien, im folgenden Ressourcen genannt, sind dabei innerhalb des Workspaceordners gespeichert. Externe Ressourcen sind entsprechend außerhalb dieses Ordners gespeichert, im Eclipse Dateisystem sind diese allerdings verlinkt. Dies kommt zum Beispiel zur Anwendung, wenn man ein Projekt in Eclipse hinzugefügt bei dem nicht die Standardlocation gewählt wird.

[1] Mittels des Remote System Explorers kann Eclipse auch auf Betriebssystem externe Dateisysteme, wie ein Webserver, zugreifen [3].

Die Einführung eines eigenen Dateisystems hat mehrere Vorteile. Zum einen sind Anwendungen nicht abhängig von einem bestimmten Betriebssystem. Damit bleiben diese plattformunabhängig, was die Entwicklung, stark vereinfacht. Weiter beschränkt sich Eclipse durch die Möglichkeit der externen Ressourcen nicht auf eine vorgegebene Struktur die eingehalten werden muss und kann Details in der Entwicklung berücksichtigen indem teile des Projekts einfach nur verlinkt werden. So müssen eingebundene Bibliotheken nicht im Projekt direkt vorliegen, sondern es reicht die Verknüpfung. Damit bleibt der Stand stets aktuell und gleichzeitig ist keine Sonderbehandlung nötig, da das Plugin darüber abstrahiert. Wie wir im Folgenden noch sehen, lassen sich durch die Abstraktion auch zusätzliche Funktionalitäten integrieren. So können wir zum Beispiel Dateiänderungen überwachen, was für unsere automatische Kompilierung von Bedeutung ist.

2.2 Hierarchischer Aufbau

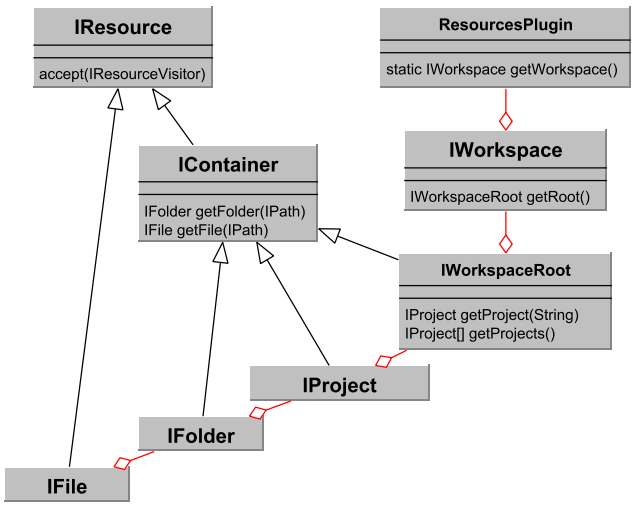

Abbildung 3: Gekürztes Klassendiagramm von `org.eclipse.core.resources`

Das Plugin bildet eine hierarchische Struktur über dem Workspace. Das gekürzte Klassendiagramm in Abbildung 3 gibt dabei einen schnellen Überblick welche Klassen dabei hauptsächlich benötigt werden. Egal ob Datei, Ordner, Projekt oder der Workspace, alle implementieren das Interface `IResource`. Sie sind also Ressourcen. Das Plugin verwendet Decoupling, um die Klassenstruktur zu verbergen und bietet uns nur Interfaces für Projekte, Ordner und Dateien an. Der Anwender erstellt also keine Instanzen von Projekt-Objekten sondern holt sich diese durch angebotene Methoden. Der Einstiegspunkt dafür bietet die Singletonklasse `ResourcesPlugin`. Das Plugin gibt vor, dass nur ein Workspace existieren kann. Dieser Workspace kann mittels der Methode `getWorkspace()` geholt werden. Dementsprechend bietet diese Klasse den globalen Einstiegspunkt für jeden Zugriff auf Ressourcen. Alle Ressourcen existieren nur im Kontext des Workspaces. Mit der Methode `getRoot()` erhält der Anwender die Ressource des Workspaces, welche die höchste Hierarchieebene einnimmt. Vergleichbar mit dem Wurzelpfad '/' in Linux. Diese Ebene enthält alle Projekte und so lässt sich auch mit der Methode `getProjects()` auf alle Projekte beziehungsweise mit der Methode `getProject(String)` auf ein bestimmtes Projekt zugreifen. `IWorkspaceRoot`, `IProject`, sowie die darunterliegende `IFolder` stellen für das Plugin alle eine Art Ordner dar und implementieren daher das `IContainer`

Interface. Dieses verfügt über die Methoden, mit denen der Anwender auf Unterressourcen, wie zum Beispiel Dateien oder weitere Ordner, zugreifen kann. Dazu benötigt man Pfade vom Typ `IPath`. Pfade drückt man in der Regel relativ zur Wurzel des Workspace aus. Weitere Varianten und Spezialfälle sprengen diesen Abschnitt und das Tutorial behandelt diese später im Detail. Zusammenfassend lässt sich festhalten, dass wir mittels mehrerer `get...`-Methoden auf konkrete Ressourcen zugreifen. So erhalten wir die nötigen Objekte, um diese Ressourcen verändern zu können.

Wir wissen bereits, dass das Interface `IResource` von allen Strukturelementen (wie `IFolder` und `IFile`) implementiert wird. Dieses Interface bietet eine große Bandbreite an Methoden die uns einen schnellen Zugriff ermöglichen. Dieses Interface implementiert das Composite-Pattern. Das heißt in ihm werden sowohl primitive Objekte (Dateien) als auch Behälter, wie Ordner und Projekte, repräsentiert. Diese intelligente Abstraktion erlaubt es uns, unabhängig vom genauen Typ, verschiedene Methoden anzubieten. Dazu gehören Methoden die Ressourcen löschen, kopieren, verschieben sowie verschiedene Methoden die Eigenschaften der Ressourcen abfragen. Zum Beispiel kann mit der Methode `exists()` geprüft werden, ob diese auch tatsächlich existiert. Die Klasse implementiert das Proxy-Pattern, denn Ressourcen-Objekte müssen nicht tatsächlich existieren [5]. Sie stellen lediglich eine Art Zeiger für den zugehörigen Ordner oder die zugehörige Datei dar. Als Anwender muss man stets in Betracht ziehen, dass Ressourcen *out of sync* sein können, falls diese nicht über die Methoden des Plugins verändert werden. Dies kommt daher, dass das Plugin nur einen Zustand hält und Veränderungen zurückschreibt. Dateimanipulationen über Klassen wie `java.io.File` sind daher zu vermeiden. Unter anderem bietet das Interface noch das Visitor Pattern an. Mit diesem lassen sich verschiedene Operationen einfach und gekapselt in einer Klasse hinzufügen. Eine solche Klasse muss dazu den `IResourceVisitor` implementieren. Um die Operation auszuführen muss ein Objekt dieser Klasse der Methode `accept()` übergeben werden. Der Visitor durchläuft dann alle Unterordner und Dateien und ruft jedes mal die implementierte Methode `visit()` auf. Dieses Pattern verwendet das Tutorial später, um alle Dateien die kompiliert werden müssen, zu finden.

2.3 Wichtige Interfaces und ihre Verwendung

In Abschnitt 2.2 haben wir gelernt, wie die grobe Struktur aussieht und weiterhin auch noch wie man auf Projekte und Dateien zugreift. In diesem Abschnitt können wir uns daher mit den Details der einzelnen Komponenten befassen. Dazu schauen wir im ersten Abschnitt zuerst die Klasse `IWorkspace` an. Daraufhin gibt es in Abschnitt 2.3.2 Infos zu den Details von Projekten. In Abschnitt 2.3.3 geht das Tutorial auf Ordner, Dateien und vor allem auf die Stolpersteine bezüglich Pfaden ein.

2.3.1 Der Workspace und das Überwachen von Änderungen Der Workspace enthält alle Projekte und verschiedene Zusatzinformationen. Wie bereits erwähnt gibt es exakt einen Workspace pro Eclipse Instanz. Von diesem kann der Anwender über den Singleton `ResourcesPlugin` die Workspace Instanz erhalten. Der Workspace bietet bereits vielfältige Funktionen. So bietet das Plugin hier die Schnittstelle zum Überwachen von Änderungen an und der Anwender kann damit den aktuellen Workspacezustand speichern. Weiter bietet die Klasse verschiedene Validierungsmethoden an. Wie bereits in Abschnitt 2.2 beschrieben, bietet die Klasse die Funktionalität, um auf Projekte zuzugreifen. Zusätzlich ermöglicht sie es dem Anwender Projektbeschreibungen zu laden. Dies benötigt man, um ein bestehendes Projekt zum Workspace hinzuzufügen.

Der Benutzer überwacht mit dem Observer Pattern Änderungen im Workspace. Wie

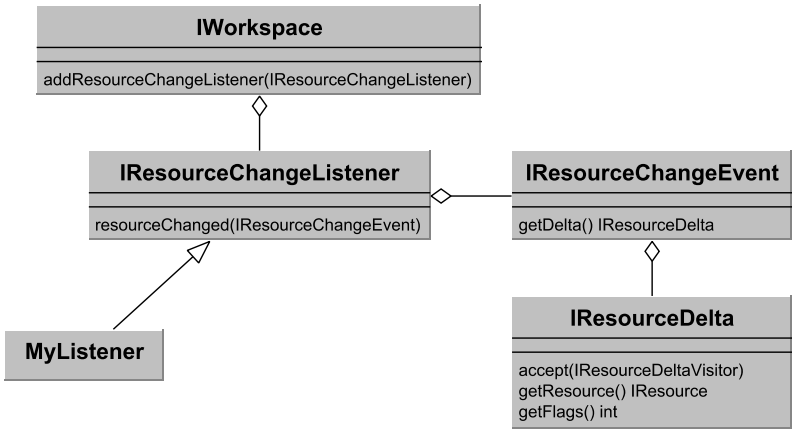

Abbildung 4: Zusammenhängende Klassen die für das Observer Pattern der Anwender für das ObserverPattern benötigt

bereits erwähnt ist diese Klasse auch die Schnittstelle für das Überwachen von Änderungen. Dies benötigt das Tutorial, um die automatische Kompilierung anwerfen zu können. Das Plugin kennt unsere Anwendung nicht, aber trotzdem benötigen wir eine Rückmeldung bei Änderungen. Deshalb müssen wir einen *Listener* anmelden. Sobald sich intern etwas ändert, ruft dann das Plugin alle angemeldeten *Listener* auf. Um den Zusammenhang der folgend genannten Klassen und Methoden zu verstehen, vergleiche die Beschreibung mit Abbildung 4. Einen solchen `MyListener` können wir mit der Methode `addResourceChangeListener()` anmelden. Ähnlich zum bereits beschriebenen Visitor Pattern implementiert `MyListener` das Interface `IResourceChangeListener`. Ein Objekt dieser Klasse melden wir dabei mit obiger Methode an. Die Methode `resourceChanged()` führt das Plugin für jeden angemeldeten *Listener* aus, falls sich etwas im Workspace ändert. Dies ist zum Beispiel der Fall, wenn der Nutzer eine Datei speichert. Die Methode liefert uns als Parameter noch ein `IResourceChangeEvent`, welches verschiedene Informationen zu den Änderungen enthält. Die wohl wichtigste Methode davon ist `getDelta()`, welche ein Objekt zurückgibt, das alle geänderten Ressourcen enthält. Es ist also nicht nur bekannt, dass sich etwas geändert hat, sondern auch was sich geändert hat. Das ganze bereitet uns das Plugin auch noch ausführlicher auf, indem für jede Ressource nachgefragt werden kann, was für eine Änderung vorgenommen wurde. So lässt sich zum Beispiel feststellen, ob eine Datei nur verändert, neu erstellt oder gar gelöscht wurde. Diese Datenstruktur implementiert das Visitor Pattern (`IResourceDeltaVisitor`) womit wir systematisch alle geänderten Dateien durchlaufen. Genau durch diese Funktionalität startet das Plugin den Kompilierungsvorgang für unseren Builder.

2.3.2 Projekte und ihre Eigenschaften Das Plugin repräsentiert ein Projekt mit der Klasse `IProject`. Ein Projekt definiert sich durch seine Eigenschaften. Zuallererst liegt der Fokus auf der Erstellung von Projekten. Das Handle für ein neues Projekt lässt sich vom Workspace, wie bereits in Abschnitt 2.2 erwähnt, holen. Der Aufruf der Methode `create()` erstellt das Projekt, das aber zu dem Zeitpunkt noch geschlossen ist. Öffnen lässt sich dieses entsprechend mit der Methode `open()`. Zu beachten ist, dass nur offene Projekte bearbeitet werden können.

Der Großteil der Eigenschaften verwaltet die Klasse `IProjectDescription`. Da-

zu gehören Projektname, die Projekt *Nature*, Buildkonfigurationen sowie Referenzen auf andere Projekte. Eine Kopie der Projektbeschreibung liefert die Methode `getProjectDescription()`. Nimmt man Änderungen vor, so müssen diese mit der Methode `setProjectDescription()` gespeichert werden. Das Ziel des Tutorials ist es einen Builder zu schreiben. Dementsprechend spielt es für uns eine große Rolle, wie sich dieser integrieren lässt. Der eigene Builder benötigt einen `ICommand`, den man über die Methode `newCommand()` erstellen und danach konfigurieren kann. Zu beachten ist, dass ein Projekt mehrere Builder haben kann und man bestehende Builder beim hinzufügen nicht löschen sollte. Die Build-Befehle sind in einem Array gespeichert, das man mit der Methode `getBuildSpec()` erhält. Umständlich kopieren wir daher das komplette Array und fügen den erstellten `ICommand` hinzu. Die Methode `setBuildSpec()` überschreibt die alte Konfiguration und setzt die neu definierte.

Die Projektbeschreibung ist im Projekt als `.project` Datei gespeichert. Diese lässt sich über die Methode `loadProjectDescription()` in der Klasse `IWorkspace` laden. Übergibt man der `create()` Methode auch noch eine Projektbeschreibung, so lässt sich auch ein existierendes Projekt hinzufügen das bisher nicht im Workspace gespeichert ist.

Mit der Projekt *Nature* kann man angeben, um welche Art Projekt es sich handelt. Der Anwender bestimmt diese durch eine ID in Form eines Strings. Handelt es sich um ein Java Projekt, so ist dies `org.eclipse.jdt.core.javanature`. Diese ID verlinkt mittels Extensionpoints auf eine Klasse die das Projekt entsprechend bearbeitet. Auf diese Art kann man auch seinen eigenen Builder aktivieren[2] Eine Projekt kann mehrere Natures besitzen.

2.3.3 Ordner, Dateien und ihre Pfade Dieser Abschnitt befasst sich damit wie Ordner und Dateien erstellt, verändert und gelöscht werden und damit was in Bezug auf die Angabe von Pfaden zu beachten ist. Wie auch bei Projekten stellen die Klassen `IFile` und `IFolder` nur einen Zeiger dar. Diesen Zeiger erhält man, wie in Abschnitt 2.2 beschrieben durch die Methoden `getFolder(IPath)` und `getFile(IPath)`. Zusätzlich bietet die Klasse `IProject` auch noch selbige Methoden an, die auch mit Strings als Pfade funktionieren als Pfad. Dabei muss man lediglich beachten, dass die Methode den Pfad relativ zum Projekt benötigt. Wie bei Projekten, wird eine Datei oder ein Ordner erst durch die Methode `create()` tatsächlich erstellt. Die Methode `setContents` ändert den Inhalt einer Datei.

Pfade im Dateisystem stellen eine große Fehlerquelle dar. Deshalb bietet das Eclipse-Plugin eine Abstraktion `IPath` an. Diese funktioniert unabhängig von dem gerade verwendeten Betriebssystem und ist nicht abhängig vom Trennzeichen. Den Pfad zu Dateien und Ordnern können wir mit drei Methoden aus `IResource` auslesen. Zum Einen gibt es `getProjectRelativePath()`, das einen `IPath` zurückgibt, der relativ zum Projekt ist. Die Methode `getFullPath()` liefert uns einen `IPath` relativ zum Workspace und `getLocation()` liefert uns den systemweiten absoluten Pfad. Dies sollte man immer im Hinterkopf behalten. Im folgenden Absatz besprechen wir noch eine häufige Stolperfalle die im Zusammenhang von `IFile` und `java.io.File` auftritt.

Viele Java-Librarys, auf die wir im Programmieralltag treffen, fordern von uns, dass wir Dateien als `java.io.File` übergeben. Das Ressourcen-Plugin liefert allerdings nur ein `IFile`. Eine `IFile` lässt sich natürlich zu einer `File` umwandeln. Nach Dokumentation der API können wir jeden `IPath` mit der Methode `toFile()` in eine `File` umwandeln. Die Praxis zeigt, dass diese Methode nur funktioniert, wenn der `IPath` ein systemweit

[2]Weitere Informationen hierzu finden sich in [2].

absoluter Pfad ist. Wir wissen, nur die Methode `getLocation()` liefert uns diesen. Dazu müssen wir beachten, dass die Methode `getLocation()` nicht auf extern gespeicherte Dateien angewendet werden kann, da darauf kein lokaler Systempfad existiert[3]. Wie in Abschnitt 2.2 bereits erwähnt, sollten wir Dateien stets über die Plugin-Methoden ändern und nicht über `File`, damit unsere Repräsentation nicht *out-of-sync* wird. Falls sich das absolut nicht vermeiden lässt, so ist danach ein Aufruf der Methode `refreshLocal()` nötig, die die entsprechende Datei auf Änderungen überprüft.

3 Exkurs: XSL Transformation

Bevor wir zur Implementation unseres Builders übergehen, wollen wir in diesem Abschnitt noch einen kurzen Überblick über XSL Transformationen geben. EXtensible Stylesheet Language Transformation (kurz XSLT) verwenden wir zur Transformation von XML-Dokumenten in andere XML-Dokumente [1]. XSL ist dabei in etwa so ein Stylesheet für XML wie es CSS für HTML ist. Wollen wir die XML-Datei in Abbildung 5a übersichtlich darstellen, wie in Abbildung 5b so verwenden wir ein XSL-Stylesheet. Diese XSL-Datei erläutern wir im nächsten Absatz.

```
1  <MODULE>
2    <VORLESUNG>[...]</VORLESUNG>
3    <SEMINAR>
4      <ID>INF9999</ID>
5        <TITEL>Technik und Konzepte der
               Eclipse Plattform</TITEL>
6        <BESCHREIBUNG>Die Eclipse
               Plattform [...]</BESCHREIBUNG
               >
7        <THEMEN>
8          <THEMA>Ressourcen: Die Basis der
                 Arbeit</THEMA>[...]
9          <LINK url="https://ovidius.uni-
                 tuebingen.de/ilias3">Quelle
                 </LINK>
10       </THEMEN>
11   </SEMINAR>
12 </MODULE>
```

(a) Gekürzte XML-Quelldatei

Inhaltsangabe

1. Kryptologie
2. Technik und Konzepte der Eclipse Plattform
3. The MonetDB Column Store Database System

Seminare und Vorlesungen

Vorlesung: Kryptologie

Seminar: Technik und Konzepte der Eclipse Plattform

Beschreibung

Die Eclipse Plattform bietet ein reichhaltiges Ökosystem von >> vorhandenen Komponenten. Dies ermöglicht es, neue Plugins für die Eclipse IDE, aber auch eigenständige Rich-Client Applikationen auf der Basis der Eclipse Infrastruktur effektiv zu erstellen. Ziel des Seminars ist es, dass alle Teilnehmer/innen einen praxisnahen Überblick über die Entwicklung mit der Eclipse Plattform erhalten. Die Themen sind dabei so ausgewählt, dass sie jeweils spezifische Teilaspekte abdecken, die zusammen ein Gesamtbild der Eclipse-Entwicklung zeichnen. Übergreifend werden die software-technischen Voraussetzungen und Prinzipien für erweiterbare und modulare Software behandelt.

Themen

1. Ressourcen: Die Basis der Arbeit
2. Plugins: Die Basis für Erweiterbarkeit
3. Editoren: Änderung von Ressourcen
4. Views: Zusatzinformationen für Arbeitsprozesse

Quelle

(b) HTML-Ergebnis

Abbildung 5

Wir definieren XSL-Stylesheets im XML-Format. XSL-Tags starten mit `xsl` gefolgt von einem Doppelpunkt. Wir definieren Templates und wählen mit regulären Ausdrücken entsprechende Knoten aus, um das originale XML-Dokument zu formatieren. Die gekürzte XSL-Datei, die Abbildung 6 zeigt, soll die Grundsätze verdeutlichen. Eine XSL-Datei hat als Wurzel einen `xsl:stylesheet`-Knoten und üblicherweise hat jede XSL-Datei ein Template, dass auf die Wurzel matched. Darin definieren wir die Wurzelknoten der Ergebnisdatei. In diesem Beispiel ist das `<html>`. `xsl:apply-templates` gibt an, dass dort zutreffende Templates auf die Kindknoten eingefügt werden sollen. Das zweite Template formatiert die Knoten `SEMINAR` und `VORLESUNG` neu. Ihre definierten Kinder `ID`, `TITEL`, `BESCHREIBUNG`, `THEMEN` und `LINK` werden dabei mit ausgegeben. Andere Kinder ignoriert der Transformer. Die ersten beiden stehen dabei immer am Anfang. Der Transformer gibt

[3] Wir können externe Files lokal cachen. Dazu muss man die Methode `IFileStore.toLocalFile()` anwenden.[3]

die anderen Kinder in der Reihenfolge aus, wie sie in der Quelldatei stehen. Der | steht dabei für ein logisches Oder. Im Ergebnis erstellen wir so aus einer simplen Quelldatei (Abbildung 5a) eine übersichtliche dynamisch generierte HTML-Datei (Abbildung 5b). Ein ausführliches Tutorial zu XSLT bietet Oracle an [7], eine weitere schnelle Einführung gibt es von Matthias Hirzel [6].

```
<xsl:stylesheet version="1.0">
  <xsl:template match="/">
    <html><body>
      <xsl:apply-templates/>
    </body></html>
  </xsl:template>
  [...]
  <xsl:template match="SEMINAR|VORLESUNG">
      <div>
        <xsl:apply-templates select="ID"/>
        <xsl:apply-templates select="TITEL"/>
        <xsl:apply-templates select="BESCHREIBUNG|THEMEN|LINK"/>
      </div>
  </xsl:template>
  [...]
  <xsl:template match="SEMINAR/TITEL">
      <h2 align="center">Seminar: <xsl:apply-templates/> </h2>
    </xsl:template>
  [...]
</xsl:stylesheet>
```

Abbildung 6: Gekürztes XSL-Beispiel

4 Builder – Der stille Helfer im Hintergrund

Dieser Abschnitt kombiniert nun die Bausteine der Ressourcen mit der XSL Tranformation und baut daraus einen Builder. Was sie für die Integration des Builders benötigen erläutert Abschnitt 4.2. Der darauffolgenden Abschnitt 4.3 erläutert das grobe Zusammenspiel, geht auf Implementierungsdetails ein und erläutert die verschiedenen Buildarten. Im letzten Abschnitt erläutert das Tutorial zusätzliche Details, die für das Grundverständnis nicht relveant sind, für eine saubere Implementierung eine Rolle spielen.

4.1 Was ist ein Builder?

Der Builder führt in Java-Projekten die Kompilierung des Quellcodes durch. Er ruft dazu den Java-Compiler auf. Definieren wir uns einen eigenen Builder, können wir damit unsere eigenen Compiler aufrufen. In diesem Tutorial rufen wir beispielsweise unseren XSL Transformator auf. Der Build-Vorgang lässt sich über mehrere Wege starten. Zum Beispiel kann der Benutzer den Build-Vorgang manuell über das Menü `Project` → `Build Project` aufrufen. Ändert der Benutzer eine Datei seines Projekts, so stößt Eclipse den Build-Prozess aber automatisch an. Hier kommt das Observerpattern für die überwachten Ressourcen ins Spiel, das wir bereits in Abschnitt 2.3.1 angesprochen haben. Eclipse verwaltet den Build-Vorgang parallel zu anderen Aufgaben. Automatische Builds stellen in Eclipse den Standardfall dar. Ist es nicht der erste Build für das Projekt, so bieten sich inkrementelle Builds an. Diese berücksichtigen nur die veränderten Dateien. Das spart Zeit und Rechenleistung.

4.2 Builder integrieren

Eclipse fordert, dass wir einen eigenen Builder über sogenannte Extensions registrieren. Die Extension `org.eclipse.core.resources.builders` ermöglichen uns einen Builder

zu konfigurieren. Wichtig ist, dass die angegebene ID dabei eindeutig bleiben muss. Die ID erhält als Namen nach der üblichen Konvention den Pluginnamen[4] kombiniert mit der Builderklasse. Optional können wir an dieser Extension einen Namen für unseren Builder angegeben. Der Benutzer bekommt diesen Namen in den Projekteigenschaften unter Builder angezeigt.[5] Wir erstellen jetzt noch ein neues Run-Element und wählen dort unseren Builder aus.

Wir wollen dem Benutzer ermöglichen unseren Builder zu aktivieren. Dies erfolgt mittels eines Kontextmenüs[6] bei dem der Benutzer den umgesetzten Builder mittels des Kontextmenüs (`Configure` → `Add XML to HTML Builder`) hinzufügen und entsprechend auch entfernen kann. Abschnitt 2.3.2 erwähnte bereits, dass die Projektbeschreibung die verwendeten Builder für das Projekt speichert. Um den Builder zu verwenden, müssen wir also beim Aufruf des Kontextmenüs den Builder dort hinzufügen. Wie beschrieben, erstellen wir uns einen neuen `ICommand`. Dieser erhält von uns die oben definierte ID als Builder Name. Diese ID verwendet Eclipse, um mittels der Reflection-API die Builderklasse zu instanzieren. Die Builderklasse wird dabei nur einmal instanziert. Kraft dieser ID lässt sich der Builder auch wieder deaktivieren, indem wir den BuildCommand aus dem Array entfernen.

4.3 Builder schreiben

Wir wollen jetzt unseren eigenen Builder implementieren[7]. Dazu erstellen wir eine Buildklasse `MyBuilder`. Diese Klasse hat man entsprechend als Extension angegeben, wie im letzten Abschnitt beschrieben. Die Klasse `MyBuilder` muss von der abstrakten Klasse IncrementalProjectBuilder erben und die Methode `build(int kind, Map<String,String> args, IProgressMonitor monitor)` implementieren (siehe Abbildung 7). Der `kind` gibt dabei die Buildart an. Die Variable `args` die gewählte Konfiguration an, in unserem Fall ist die Map leer, da wir keine Konfigurationsmöglichkeiten angeben. Die letzte Variable `monitor` ignorieren wir für diesen Abschnitt. Diese benötigen wir erst später, wenn wir den Fortschrittsbalken hinzufügen. Eclipse ruft die Methode auf, um den Buildprozess zu starten. Als Rückgabewert können wir eine Liste an Projekte angegeben, für die der Builder beim nächsten Aufruf die Deltas benötigt. Dies benötigen wir allerdings nicht, da die XSL Transformation nicht von anderen Projekten abhängt.

4.3.1 Die Buildarten Im Buildprozess ist es nicht jedes mal notwendig, das komplette Projekt von Grund auf neu zu betrachten. Deshalb gibt es vier verschiedenen Buildarten. Diese sind `FULL_BUILD`, `CLEAN_BUILD`, `AUTO_BUILD` und `INCREMENTAL_BUILD`. Unsere Methode in Abbildung 7 wählt dementsprechend aus, welcher Buildvorgang nun konkret durchgeführt werden soll. Ein `AUTO_BUILD` oder `INCREMENTAL_BUILD` ist abhängig von vorherigen Buildvorgängen. Ist kein vorheriger Zustand vorhanden, so ist es immer ein `FULL_BUILD`. Die Methode `getDelta()` gibt dabei den Unterschied des Projekts zum letzten Build an. Erkennt Eclipse Änderungen im Projekt, so stößt es einen `AUTO_BUILD`

[4] Zusätzlich sollte man sich im Hinterkopf behalten, dass ein Plugin immer als Root-Package seinen eigenen Namen hat.

[5] Eclipse generiert dabei auch automatisch mehrere Optionen die wir für unseren Builder festlegen können. Diese sind für den vorgestellten Beispielbuilder allerdings nicht relevant.

[6] Dieses Tutorial befasst sich mit Ressourcen und Buildern. Es verzichtet dabei darauf zu erläutern, wie der Entwickler das Kontextmenü um eigene Befehle erweitern kann. Schauen sie sich hierzu das Tutorial Menüs & Toolbars an, um deren Funktionsweise zu verstehen.

[7] Den Quellcode für den hier vorgestellten Builder können sie unter `http://konstantin-grupp.de/eclipse.php` herunterladen.

```
1  protected IProject[] build(int kind, Map<String,String> args, IProgressMonitor
       monitor)
2      throws CoreException {
3    IProject project = this.getProject();
4    if (kind == IncrementalProjectBuilder.FULL_BUILD) {
5      fullBuild(monitor);
6    } else {
7      IResourceDelta delta = getDelta(project);
8      if (delta == null) {
9        fullBuild(monitor);
10     } else {
11       incrementalBuild(delta, monitor);
12     }
13   }
14   return null;
15 }
```

Abbildung 7: Die `build()`-Methode [3]

an. Vom Benutzer angestoßene Builds sind dementsprechend inkrementelle Builds. Bei einem `CLEAN_BUILD` ruft Eclipse erst `clean()` auf und startet dann einen vollständigen Buildvorgang. Ein `FULL_BUILD` soll dabei unabhängig von vorherigen Buildvorgängen das komplette Projekt neu bearbeiten. Deshalb löscht die Methode `fullBuild()` den kompletten Ordner mit eventuell alten Kompilierungen, wie der Ausschnitt in Abbildung 8 zeigt. Danach durchsucht die Methdoe das komplette Projekt mit dem Visitor Pattern. Wie das funktioniert behandelt der folgende Abschnitt.

```
1 project.getFolder(COMPILED_FOLDER).delete(false, monitor);
2 XMLtoHTMLResourceVisitor visitor = new XMLtoHTMLResourceVisitor(project,monitor);
3 project.accept(visitor);
4 compileFoundFiles(visitor.getResult(), monitor);
```

Abbildung 8: Ausschnitt aus fullBuild-Methode

4.3.2 Visitoren Ressourcen bilden umfangreiche Baumstrukturen. Dadurch werden häufig Rekursionen notwendig. Um die Programmierung zu vereinfachen, bietet `IResource` und `IResourceDelta` das Visitor-Pattern an, wie bereits in Abschnitt 2.2 und 2.3.2 angesprochen. Für die Implementierung macht es kaum einen Unterschied, ob man über die `IResource` oder die `IResourceDelta` Struktur visitiert. Bei Letzterem kann man mit der Methode `getResource()` die zugehörige Ressource erhalten und fast gleich verfahren wie sonst auch. Daher fassen wir diese zusammen. Wir implementieren beide Visitoren in einer Klasse `XMLtoHTMLResourceVisitor` und erstellen eine zusätzliche Methode `visitHelper()` welche fast die komplette Funktionalität beider Visitoren abdeckt. Diese ist in Abbildung 9 abgebildet. Für alle Dateien die eine Kompilierung nötig haben legen wir eine Liste an, die wir dann später abarbeiten (Zeile 11). Dafür müssen wir prüfen ob die aktuelle Ressource eine Datei ist, was Zeile 2 erledigt. Entsprechend wollen wir auch nur XML-Dateien kompilieren, deshalb prüfen wir das in Zeile 10. Für den Delta Visitor ist lediglich noch zu beachten, dass der Builder für jede gelöschte Quelldatei auch deren kompilierte Datei löscht. Im Visitor legen wir dafür einfach eine Liste an zu löschenden Dateien an. Weiter müssen wir bei inkrementellen Builds auch darauf achten, dass sich bei geänderter XSL-Datei auch alle kompilierten Dateien ändern. Diesem kommen wir in Zeile 6-9 (in Abbildung 9) nach.

Die Variable `isIncre` gibt dazu an, ob unser aktueller Buildvorgang ein inkrementeller ist. Nach dem Visitieren prüft die Methode `incrementalBuild()` entsprechend, ob ein vollständiger Build notwendig ist (siehe Abbildung 12). Im Falle des inkrementellen Builds wenden wir den eben spezifizierten Visitor dann auf das Ressourcen Delta und im Falle des vollständigen Builds auf das gesamte Projekt an. Wie das beim vollständigen Build im Quellcode aussieht, zeigt Abbildung 8. Zusammenfassend erhalten wir also mit den Visitoren einen einfachen Zugriff auf die relevanten Dateien.

```
1  private boolean visitHelper(IResource resource, boolean isIncre) {
2    if (resource.getType() != IResource.FILE) {
3      return true;
4    } else {
5      IFile sourceIFile = project.getFile(resource.getProjectRelativePath());
6      if (isIncre && sourceIFile.getName().equals(MyBuilder.STYLESHEET_FILE)) {
7        this.needRebuild = true;
8        this.files.clear();
9        return false;
10     } else if (sourceIFile.getFileExtension().equals("xml")) {
11       files.add(sourceIFile);
12     }
13   }
14   return true;
15 }
```

Abbildung 9: Die `visitHelper()`-Methode

4.3.3 Dateien compilieren Mit der Java-Bibliothek `javax.xml` transformieren wir die XML-Dokumente mit einem XSL-Stylesheet in eine HTML-Datei. Wir müssen uns also ein Methode `transform()` basteln, die aus dem `InputStream` von XML-Quelldatei und XSL-Stylesheet Datei einen `InputStream` für unsere kompilierte Datei bilden. Dabei ist es wichtig, dass dies für `InputStream` funktioniert, da die Klasse `IFile` den Inhalt nur als `InputStream` ausgeben und speichern kann. Zusätzlich wollen wir die Verwendung von `java.io.File` aus den in Abschnitt 2.3.3 genannten Gründen vermeiden. Die genauen Details wie die Tranformation funktioniert ist für dieses Tutorial nicht von Relevanz. Interessierte können sich in Abbildung 11 einen Überblick verschaffen. In Zeile 10-12 findet die tatsächliche Transformation statt. Die Methode `transform()` rufen wir für jede Datei auf, die kompiliert werden soll. Der genaue Aufruf erfolgt dann so wie es Abbildung 10 zeigt. Unser Builder generiert sich einen Zeiger auf die Zieldatei mit der Methode `getFile()`. Der Einfachheit halber ergänzen wir den Namen der Zieldatei um `.html` und speichern die Datei lediglich in den Zielpfad. Danach findet die Transformation statt. Beim Speichervorgang gibt es eine Unterscheidung ob die Datei schon existiert, da dementsprechend die Methode `create()` nicht bei existierenden Dateien funktioniert.

```
1 IFile result = project.getFile(destination.append("/"+name+".html"));
2 InputStream output = XSLTransformer.transform(sourceXML.getContents(), styleXSLT.
      getContents(), monitor);
3 if (result.exists()) {
4   result.setContents(output, false, true, monitor);
5 } else {
6   result.create(output, false, monitor);
7 }
```

Abbildung 10: Ausschnitt aus der Schleife in `compileFoundFiles()`-Methode

```
1  public static InputStream transform(InputStream sourceXML, InputStream stylesheetXSL,
       IProgressMonitor monitor) {
2    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
3    StreamSource stylesource = new StreamSource(stylesheetXSL);
4    ByteArrayOutputStream resultOS = new ByteArrayOutputStream();
5    StreamResult result = new StreamResult(resultOS);
6    try {
7      DocumentBuilder builder = factory.newDocumentBuilder();
8      Document document = builder.parse(sourceXML);
9      DOMSource source = new DOMSource(document);
10     TransformerFactory tFactory = TransformerFactory.newInstance();
11     Transformer transformer = tFactory.newTransformer(stylesource);
12     transformer.transform(source, result);
13   } catch (TransformerException | SAXException | ParserConfigurationException |
         IOException e) {
14     OutputHandler.getInstance().println(e.getMessage());
15     return new ByteArrayInputStream("compile exception".getBytes());
16   }
17   return new ByteArrayInputStream(resultOS.toByteArray());
18 }
```

Abbildung 11: Die `transform()`-Methode [7]

4.4 Erweiterung

Einen grundsätzlichen Builder haben wir nun erstellt. Dieser lässt sich benutzerfreundlicher gestalten, wenn wir einen Fortschrittsbalken verwenden. Wie wir diesen implementieren erläutert dieser Abschnitt. Ein Benutzer benötigt in weniger als einer Sekunde eine Rückmeldung, ob etwas passiert und was passiert. Eclipse bietet den `IProgressMonitor` an, um dem Benutzer eine Rückmeldung über den aktuellen Fortschritt zu geben. Das `IProgressMonitor` Interface bietet eine Unzahl an Fehlerquellen, mit dem der Fortschrittsbalken schnell kaputt geht oder nutzlos wird. Im Folgenden werden einzelne Punkte erläutert und Tipps gegeben. Will man die Arbeit auf mehrere Methoden verteilen, so muss man mit `SubMonitor.convert(monitor,ticks)` einen `Submonitor` generieren. Die Anzahl der Ticks gibt dabei die Gesamtanzahl der Restarbeit an. Jeder Methode übergibt man dabei ein Kind des Monitors mit `newChild(ticksworked)`. Im Detail schauen wir dazu die Methode `incrementalBuild()` in Abbildung 12 an. In Zeile 3 erstellen wir den Untermonitor mit dem wir die Arbeit aufteilen. Dabei geben wir 100 Ticks an, welche jeweils 1% der Arbeit repräsentieren. Wir schätzen ab, dass 5% der Arbeitszeit für das finden der veränderten Dateien benötigt werden. Dazu übergeben wir dem Visitor ein `newChild(5)`. Hat sich eine XSL-Datei geändert, so wird ein vollständiger Build benötigt[8], wie Abschnitt 4.3.2 erläutert. Dieser erhält dann die restliche Zeit. Entsprechend verhält es sich für die Methode `compileFoundFiles()`. Die entsprechenden Methoden teilen den Teil des Fortschrittsbalkens für sich wiederum so auf, wie sie es für nötig halten.

Bei der Verwendung eines Fortschrittsbalkens ist zu beachten, dass man sich überlegen muss, wie viel Arbeit einzelne Teile ihrer Implementierung benötigen. Der in Abschnitt 4.3 eingeführte `XMLtoHTMLResourceVisitor` compiliert daher nicht direkt die Dateien, sondern speichert die zu bearbeitenden Dateien lediglich in einer Liste. Der Builder arbeitet die Liste später dann ab in der beschriebenen Methode `compileFoundFiles`.

[8] Eigentlich sollte ein vollständiger Build durch die Methoden des Builders `forgetLastBuildState()` und `needRebuild()` durchgeführt werden. Dadurch erhält die Methode `build()` einen Monitor der bereits die ganze Arbeit erledigt hat. Deshalb wird hier manuell der vollständige Buildvorgang aufgerufen.

```
1  private void incrementalBuild(IResourceDelta delta,
2      IProgressMonitor monitor) throws CoreException {
3    SubMonitor subMonitor = SubMonitor.convert(monitor, 100);
4    XMLtoHTMLResourceVisitor visitor = new XMLtoHTMLResourceVisitor(getProject(),
         subMonitor.newChild(5));
5    delta.accept(visitor);
6    if (visitor.needsRebuild()) {
7      fullBuild(subMonitor.newChild(95));
8    } else {
9      compileFoundFiles(visitor.getResult(), subMonitor.newChild(95));
10   }
11 }
```

Abbildung 12: Die `incrementalBuild()`-Methode als Beispiel für die Verwendung von `IProgressMonitor`

Zu diesem Zeitpunkt kennt er aber bereits die Anzahl der zu bearbeitenden Dateien, da der Visitor eine Liste dieser Dateien bereithält. Jede Datei benötigt dabei je ein Tick für die Kompilierung und das Speichern des Ergebnisses. Unser Submonitor für diese Methode benötigt also #Anzahl $\cdot$ 2 Ticks zum Verteilen der Arbeit. Der Builder erstellt mittels der Methode `newChild(1)` einen Untermonitor für die Methoden `transform()`, `create()` bzw. `setContents()`. So lässt sich der Fortschrittsbalken anwendungsfreundlich einsetzen.

Mit dem Interface `IProgressMonitor` bietet das Plugin nicht nur die Möglichkeit einen Fortschrittsbalken zu verwenden, sondern auch die Möglichkeit aktuelle Operationen abzubrechen. Dazu muss bei der Verwendung beachtet werden, dass in Schleifen oder Rekursionen die viel Zeit benötigen mit `isCanceled()` überprüft wird, ob der Nutzer die Operation abbrechen möchte. Wurde sie abgebrochen, so sollte man `OperationCanceled Exception` werfen.

Üblicherweiße erhält man einen Monitor von Eclipse direkt. Initiiert man eine langlaufende Operation durch die View so ist dies nicht der Fall. Will man dafür einen Monitor verwenden, so kann man sich eine Klasse erstellen, die von der abstrakten Klasse `Job` erbt. Dort implementiert man dann die Methode `run()` (vergleichbar zur Implementierung von Threads) in der man auch einen von Eclipse bereitgestellten Monitor erhält. Man erstellt von der Klasse ein neues Objekt und mit der Methode `schedule()` fügt man es der Warteschlange hinzu.

5 Das haben wir erreicht

Dieses Tutorial gab einen groben Überblick über die Möglichkeiten des Eclipse-Plugins `org.eclipse.core.resources`. Wir haben in Abschnitt 2 gelernt, wie wir Ressourcen als Abstraktionsebene über dem Dateisystem verwenden können und wie wir deren Eigenschaften auslesen. Weiter haben wir mehrere Pattern angeschaut, die das Plugin verwendet. Darunter das Visitor und Observer-Pattern. Mit diesem Basiswissen haben wir uns in Abschnitt 4 einen eigenen Builder implementiert. Zusätzlich haben wir diesen mit einem Fortschrittsbalken ausgestattet, um unseren Builder anwenderfreundlich zu gestalten. Das Ressourcen-Plugin enthält aber noch viele weiter interessante Funktionen auf die wir hier nicht eingehen können um den Rahmen nicht zu sprengen. Unter anderem ist das die Historyverwaltung, die eine Art Versionsverwaltung bietet. Weiter gibt es noch interessante Punkte, wie man als Anwender mit Dateien umgeht die extern (zum Beispiel auf einem Webserver) gespeichert sind. Auch Buildkonfigurationen sowie die

Buildordnung im Workspace haben wir der Einfachheit geschuldet komplett ignoriert. Alles in allem haben wir mit unserem Tutorial gezeigt, wie einfach es ist einen Builder zu schreiben, mit dem man die Effizienz am Arbeitsplatz steigern und die Anzahl der Schwertkämpfe am selbigen verkleinern kann.

Literatur

[1] Clark, James: *XSL Transfomations (XSLT).* `http://www.w3.org/TR/xslt`, 1999.

[2] Corporation, IBM: *Platform Plug-in Developer Guide - Advanced resource concepts - Project nature.* `http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2FresAdv_natures.htm`, 2012.

[3] Corporation, IBM: *Platform Plug-in Developer Guide - Advanced resource concepts - Working with resources in other file systems.* `http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2FresAdv_efs_resources.htm`, 2012.

[4] Corporation, IBM: *Eclipse Resources Documentation Release 4.4 API*, 2014.

[5] Gast, Holger: *Software-Architektur 1.* Technischer Bericht, Universität Tübingen, December 2011.

[6] Hirzel, Matthias: *Seminar Grundlagen von Programmiersprachen.* Technischer Bericht, Universität Tübingen, January 2012.

[7] Oracle: *Transforming XML Data with XSLT.* `https://docs.oracle.com/javase/tutorial/jaxp/xslt/transformingXML.html`.

[8] xkcd.com: *Compiling.* `http://xkcd.com/303/`.