Entwicklerdokumentation

PUML

Inhaltsverzeichnis

I. Prozess- und Implementationsvorgaben

I.1 Definition of Done

- Es können mindestens für Java-Quellcode Klassen- und Sequenzdiagramme erzeugt werden
- Sämtliche Codefragmente sind versioniert
- Der Code ist (soweit möglich) modular aufgebaut und damit gut wartbar
- Der Code ist gut strukturiert und ausreichend kommentiert, Coding Guidelines und Standards wurden eingehalten
- Die Testabdeckung beträgt mindestens 50 Prozent
- Es sind keine kritischen Bugs offen
- Die Entwicklerdokumentation enthält alle notwendigen Informationen, die ein potentielles späteres Team zum Weiterentwickeln des Codes benötigen würde
- Das Benutzerhandbuch enthält alle notwendigen Informationen, die ein Anwender braucht, um das Produkt bedienen zu können

I.2 Coding Style

Bitte die Datei javaCodeStyle.xml im specification-Verzeichniss in Eclipse importieren und verwenden. Hierfür in Eclipse unter „Window->Preferences->Java->Code Style->Formatter“ auf Import klicken und die XML-Datei auswählen.
Ist der passende Coding Style eingestellt kann der Quellcode mit „STRG+SHIFT+F“ automatisch formatiert werden. Wird dies vor jedem Commit gemacht, ensteht ein einheitlicher Code-Style und die Änderungen können gut mit GIT überprüft werden.
Des weiteren empfiehlt es sich bei größeren oder stark geschachtelten Code-Abschnitten die Zugehörigkeit der Schließenden Klammer mit einem Kommentar zu Kennzeichnen.

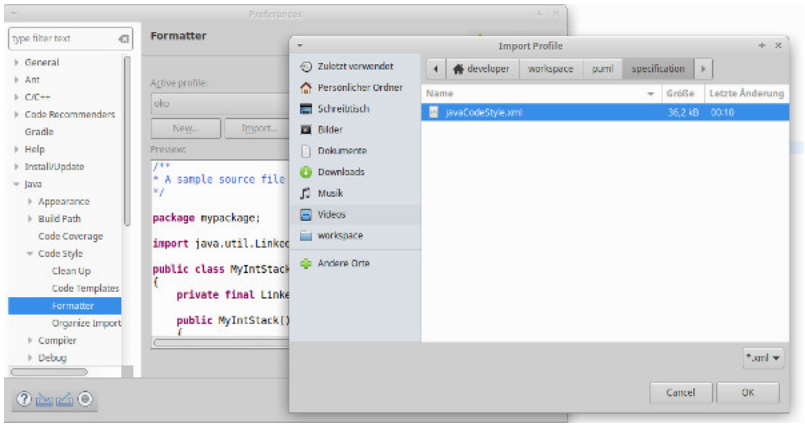


Abbildung 1: Code-Style in Eclipse importieren

Sonstige Konventionen:

- Variablen und Instanzen beginnen kleingeschrieben
- Klassen und Interfaces beginnen mit Großbuchstaben
- Besteht ein Namen aus mehreren zusammengesetzten Wörtern, beginnen alle weiteren Wörter mit Großbuchstaben (keine Unterstriche in Namen verwenden)
- Aussagekräftige Namen verwenden
- Alle Namen auf Englisch
- Die Kommentare auf Deutsch
- Lange Kommentare immer vor den Codeabschnitt
- Alle Methoden im Javadoc-Stiel dokumentieren

I.3 Zu nutzende Werkzeuge

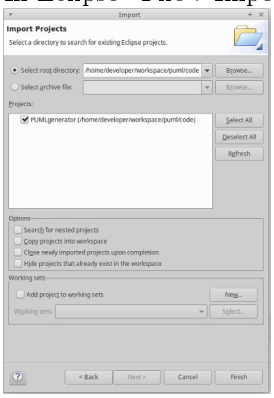
- Eclipse - Entwicklungsumgebung
- GIT - Dateiversionierung
- Meld - Unterschiede zwischen Dateien anzeigen
- Texmaker - Latex-Editor
- GIMP - Bildbearbeitung für das Editieren von Screenshots

II. Was wird wie gemacht?

II.1 Eclipse

II.1.1 Projekt in Eclipse importieren

In den workspace wecheln:
cd /workspace
Projekt Klonen:
git clone https://gitlab.imn.htwk-leipzig.de/weicker/puml.git
Benutzername und Passwort eingeben.
In Eclipse "File->Import->Existing Projects into Workspace"



Dann auf “Finish“ klicken.

II.1.2 WindowBuilder installieren

In Eclipse "Help->Install New Software..."
Unter work with “2018-09 - http://download.eclipse.org/releases/2018-09“ auswählen.
In der Section “General Purpose Tools“ die im Bild stehenden Häckchen anklicken
Dann auf “Finish“ und sich durch die Installation klicken.

II.1.3 GUI editieren

Es muss der WindowBuilder installiert sein. Dann auf die Datei die die Grafische Oberfläche implementiert (GUI.java) mit der rechten Maustaste klicken. Dann “Open With->WindowBuilder Editor“ auswählen.

II.2 LaTeX

II.2.1 Geschachtelte Überschriften

Durch die Makros:

- \nsecbegin{MeineÜberschrift}
- \nsecend

können geschachtelte Überschriften verwendet werden. Die Kapitel einfach in diese Makros einschließen. Somit muss nicht darauf geachtet werden auf welcher Ebene man sich im Moment befindet. Dies vereinfacht insbesondere das Auslagern von Text in andere Dateien.

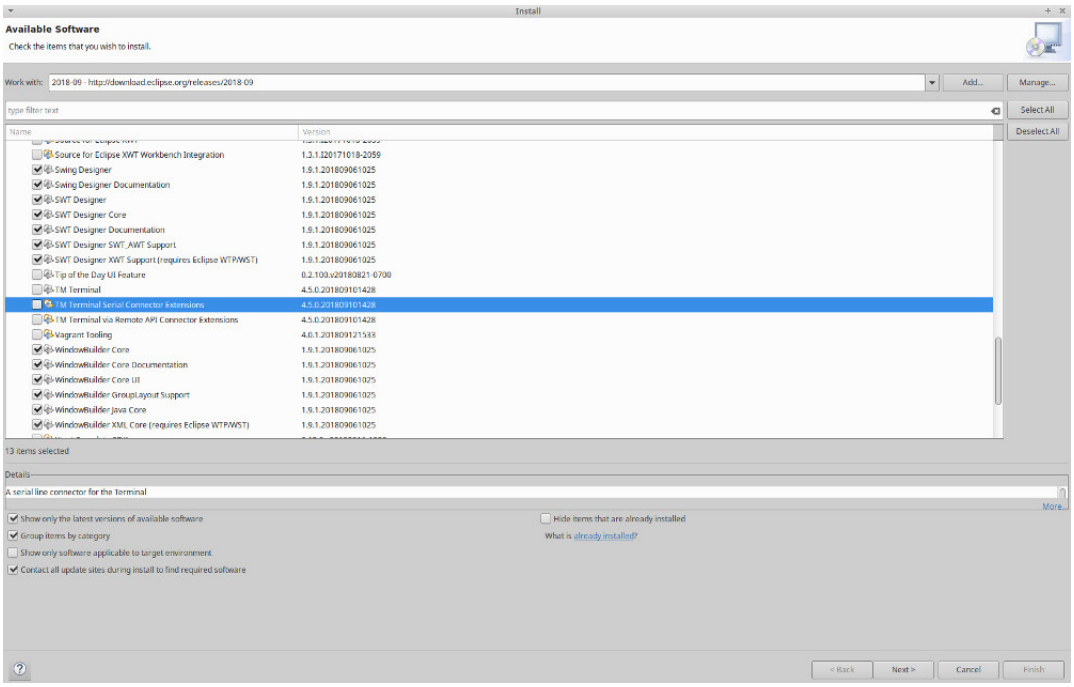


Abbildung 2: WindowBuilder installieren

Um die Makros in die Autovervollständigung des Texmakers aufzunehmen "Benutzer/in->Wortvervollständigung anpassen" wählen und dort die Makros hinzufügen.

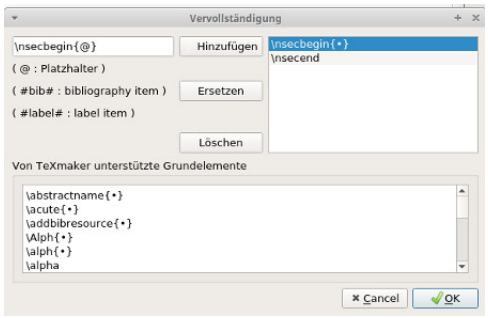


Abbildung 3: Autovervollständigung anpassen

II.2.2 Build-Dateien aufräumen

Beim erstellen des LaTeX-Dokuments werden jede Menge zusätzliche Dateien erstellt. Dank der entsprechenden ".gitignore-Datei" werden diese nicht in GIT hinzugefügt. Für den Fall dass man das Verzeichniss bei sich selbst bereinigen möchte, kann das "clean.sh"-Script ausgeführt werden.

II.2.3 Entwicklerdokumentation und Handbuch erstellen

Wenn etwas an der Entwicklerdokumentation oder am Handbuch geändert wurde, müssen diese Dokumente neu erstellt werden. Hierfür zunächst wie gewohnt die LaTeX-Projektdokumentation

erstellen. Anschließend kann das “buildAllDocuments.sh“-Script ausgeführt werden. Dieses erstellt dann die entsprechenden Dokumente.
Weitere Informationen zum “multiaudience-Paket“ unter https://www.uweziegenhagen.de/?p=3252.

II.3 GIT

II.3.1 Benutzername und eMail ins GIT eintragen

In Linux kann durch den Aufruf:
gedit ~/.gitconfig
die “.gitconfig-Datei“ editiert werden. In dieser werden unter anderem auch Benutzername und eMail-Addressse des Benutzers gespeichert.

II.3.2 Basics

```
#Aktueller Zustand ausgeben
#Auf Welchem Branch bin ich?
#Gibt es Dateien die gendert sind?
#Wurden Dateien gelscht?
#Wurden neue Dateien hinzugefgt?
#Sind Aenderungen bereits fr den Commit vorgemerkt?
#Bin ich vor oder hinter dem Remote-branch?
git status

#Alle Aenderungen fr den Commit vormerken.
# Fr  den Punkt kann auch ein Pfad angegeben werden um bestimmte nderungen vorzumerken.
#Wenn die .gitignore-Datei richtig gepflegt wird, sollte immer die Variante mit dem Punkt
    verwendet werden knnen.
git add .

#Vorgemerkte nderungen Commiten
#Anschlieend muss die Commit-Nachricht im Editor eingetragen werden
git commit

#Alle commits auflisten
git log

#Unterschiede zwischen der aktuellen Version und einem lteren Commit anzeigen
git difftool hashDesCommits

#Unterschiede zwischen zwei aelteren commits anzeigen
git difftool hashDesErstenCommits hashDesZweitenCommits

#Zu einem lteren Commit wechseln
git checkout hashDesCommits

#Neuen Branch vom aktuellen Stand aus erstellen
git branch nameDesNeuenBranches

#Zu einem Branch wechseln
git checkout nameDesBraches
```

II.3.3 Mergen

```
#Einen anderen Branch in meinen aktuellen mergen
git merge nameDesBranches

#Bei Merge-Konflikt
git mergetool

#Fuer eine Datei direkt meine Version verwenden
git checkout --ours -- nameDerKonfliktdatei

#Fuer eine Datei die Remote-version verwenden
git checkout --theirs -- nameDerKonfliktdatei

#Nach dem mergen
git commit
```

II.3.4 Arbeiten mit dem Server

```
#Remote anzeigen
git remote -v

#Aktuelle Version eines Branches holen
git pull origin branchName

#Meine nderungen auf einen Branch hochladen
git push origin branchName
```

II.3.5 Eigenen Branch mit dem Master Synchronisieren

Wenn sich der Master wärend der Entwicklung am eigenen Branch weiter entwickelt hat, können die Änderungen des Master auf folgende weise in den eigenen Branch übernommen werden.

```
git status #Pruefen ob auf meinem Branch
#wenn nicht
git checkout myBranch
#Lokalen Master aktuallisieren
git pull origin master #sollte auch gleich in myBranch mergen
#wenn nicht
git merge master
#Wenn merge-konflikt
git mergetool
#Jetzt noch den Merge commiten
git commit
```

II.3.6 Lokale Branches aufräumen

ACHTUNG: Sollte nur gemacht werden, wenn alle Änderungen in den Master übernommen wurden und somit sicher sind!!

```
#Alle branches loeschen die es nicht mehr auf dem Server gibt
git remote prune origin
#Lokale branches die mit dem master gemerged wurden loeschen
git branch --merged master | grep -v '^[ *]*master$' | xargs git branch -d
```

II.4 Code

II.4.1 Logger

Ein Logger hat verschiedene Level in denen geloggt werden kann:

- severe - Schwerwiegende Fehler
- warning - Warnungen
- info - Informationen
- config - Konfigurationshinsweise
- fine - Fein
- finer - Feiner
- finest - Am Feinsten

Um nun an einer bestimmten Stelle etwas zu loggen, macht man folgenden Aufruf:

PUMLgenerator.logger.getLog().[Level]("[Ausgabe]")

Hierbei wird die in der Hauptklasse (PUMLgenerator) definierte Instanz "loggerïn Verbindung mit dem Getter getLog() aufgerufen, danach der Level des Logs definiert und am Ende der Ausgabestring eingegeben.
Beispiel:
PUMLgenerator.logger.getLog().info("Dies ist eine Information");

Um die Logs nun auch auf die Konsole auszugeben oder in eine Datei zu schreiben müssen die jeweiligen Handler aktiviert werden. Mit den Funktionen *startLoggingFile(String path)* und *startLoggingConsole(Boolean console)* lassen sich die Handler aktivieren.
Beispiel:
PUMLgenerator.logger.startLoggingFile("testfolder/tempData/PUMLlog/");
Aktiviert das Schreiben einer Logdatei
PUMLgenerator.logger.startLoggingFile(true);
Aktiviert das Ausgeben auf die Konsole

Zum Start des FileHandlers wird eine Logdatei mit exakter Uhrzeit/Datum als Präfix erstellt. Jeder Logeintrag wird im jeweils angegebenen Ordner gespeichert. Zu jedem Logeintrag gibt es das genaue Datum inkl. Uhrzeit sowie die Informationen in welcher Klasse und in welcher Methode es geloggt wurde. Des Weiteren werden auch Logausgaben mit den gleichen Eigenschaften (Uhrzeit, Klassenname, ...) über die Console realisiert.

LogMX
Die Logdatei wird in einem XML-Format gespeichert. Um dieses komfortabel auslesen zu können, kann man sich mit Hilfe von LogMX, einem plattformunabhängigem LogViewer, die Logdatei anschauen und Filtern (nach Klassen, Methoden, Uhrzeit, ...).
LogMX kann auf diverse Arten implementiert werden.

Linux
Für Linux Distributionen gibt es ein entpackbares Archiv, welches nach dem Download entweder

direkt als Anwendung gestartet werden kann. Hierzu muss einfach die logmx.sh ausgeführt werden. Beim Start muss als erstes angegeben werden, wo die Datei zu finden ist und danach ob nur eine Logdatei oder mehrere Logdateien in einem Verzeichnis geöffnet werden sollen. Als nächstes muss das Log Format angegeben werden. Das Log Format für alle PUML-Logs ist Java logging. Final muss nun nur noch Java XML format ausgewählt und der Pfad zur Datei eingegeben werden. Danach kann die Datei gelesen werden.

Um die Dateien direkt über Eclipse zu öffnen muss man einmalig ein paar Einstellungen vornehmen. Unter

$\rightarrow Window \rightarrow Preferences \rightarrow General \rightarrow ContentTypes$
kann man in der Liste Text auswählen und ausklappen. Wenn man jetzt XML markiert (kein ausklappen nötig; gemeint ist nicht XML(Illformed)) kann man in untersten Fenster unter Associated Editors über Äddëinen neuen Editor hinzufügen. Dazu wählt man die logmx.sh Datei aus dem entpackten Ordner aus und speichert die Einstellungen. Wenn man nun via Rechtsklick eine Logdatei in Eclipse auswählt kann man unter Öpen With"nun logmx auswählen.

Windows
Für Windows gibt es drei Download-Varianten. Man kann zwei .exe-Installer (mit und ohne Java) herunterladen oder ein gepacktes Archiv, welches ebenfalls eine .exe-Datei enthält. Wie in der Linux-Variante kann man auch hier das Programm allein starten oder auch direkt aus Eclipse heraus. Die Schritte sind bei der Implementierung exakt die selben, wie unter *"Linux"* beschrieben, nur das hier nicht die logmx.sh, sondern die LogMX.exe/LogMX-64.exe ausgewählt werden muss (entweder im Archivordner oder im Installationspfad). Da LogMX einen Java Path benötigt, muss dieser noch in der Datei startup.conf ergänzt werden.

II.5 Profiler

II.5.1 Installation

Help > Eclipse Marketplace
Search: Profiler > Java Mission Control > Install
Nach der Installation sollte in der Symbolleiste ein Icon zum Starten des Profilers erscheinen.

II.5.2 Verwendung

Nach dem Starten wird in dem auftauchenden Fenster unter Local 'Eclipse' und anschließend 'Start JMX Console' ausgewählt.

- Overview
  Hier sind allgemeine Informationen über die Systembelastungen zu finden.
- MBean Browser (Managed Beans)
  MBeans sind Java Objekte, die verwaltet werden können. Das können Geräte, Anwendungen oder andere Ressourcen, die verwaltet werden, sein.
  Diese können unter diesem Reiter abgerufen werden.
- Triggers
  Um verschiedene Events zu testen, können Trigger gesetzt werden. Mittels 'Add' ruft man eine Liste der möglichen Trigger auf und kann diese nach dem Hinzufügen über die Reiter 'Conditions', 'Actions' und Constraints modifizieren.
  Außerdem lassen sich Trigger importieren sowie exportieren.

- System
  Eingie Systeminformationen werden hier angezeigt.
- Memory
  Dieser Punkt gibt einen Überblick über Heap und Garbage Collection.
- Threads
  Hier findet man eine Liste der laufenden Threads.
- Diagnostic Commands
  Die hier aufgelisteten Befehle können hilfreich sein, um spezielle Diagnosen zu erfragen. Dafür wird der entsprechende Befehl ausgewählt, falls nötig unter 'Description - Value' mit Werten versehen und dann kann er ausgeführt werden.

Allgemein können die einzelnen Felder durch die '+'-Symbole weiter angepasst werden.

II.6 Generell

II.6.1 Graphviz

Um PlantUML richtig und im vollem Umfang anzeigen zu können muss Graphviz installiert werden:

1. Download unter graphviz.org/download/ für jeweiliges System
2. Folgt den Installationsanweisungen
3. Graphviz sollte nun erfollgreich installiert sein.

Graphviz soll für den Endkunden in den Installer implementiert werden.

II.7 XML / XPath

II.7.1 XPath-Ausdrücke ausführen

Für das Verarbeiten von XPath-Ausdrücken wurde die Methode "getList" in der Klasse "XmlHelperMethods" erstellt. Von dieser Klasse existiert eine Instanz in der Klasse "PUMLGenerator" mit dem Namen "xmlHelper". Ein XPath-Ausdruck kann also folgendermaßen ausgeführt werden:

```
NodeList myNodeList = PUMLGenerator.xmlHelper.getList(myNode, "myXPathExpression");
```

Wobei eine List mit allen gefunden Knoten zurückgeliefert wird. Es kann relativ zum übergebenen Knoten (myNode) gesucht werden. Weitere Informationen zu XPath-Ausdrücken unter:
https://www.w3schools.com/xml/xpath_syntax.asp

II.7.2 Nächsten Nachbarknoten suchen

Weil sich die "nextSibling"-Methode der Java-XML-Implementierung etwas seltsam verhält und ggf. die Leerzeichen der Zwischenräume zwischen den Knoten als text-Knoten erkennt, sollte auch hier ein XPath-Ausdruck verwendet werden. (https://stackoverflow.com/questions/17641496/how-can-i-get-the-si
Der Ausdruck:

```
Node nextNode = PUMLGenerator.xmlHelper.getList(currentNode,
    "following-sibling::*").item(0);
```

wählt den nächsten Nachbarknoten aus. Das selbe gilt für die Auswahl des ersten Kindknotens. Der hierfür benötigte Ausdruck:

```
Node childNode = PUMLgenerator.xmlHelper.getList(methodefNode, "child::*").item(0);
```

III. Best practice

III.1 GIT

III.1.1 Keine nicht benötigten Dateien adden

Vor dem “git add .“ immer mit “git status“ prüfen welche Dateien hinzugefügt werden. Sollten nicht für das Projekt benötigte Dateien (z.B. übersetzte Binärdateien oder Dokumentation von Librarys) dabei sein, bitte die entsprechende “.gitignore-Datei“ vervollständigen. Danach sollten die Dateien beim “git status“ nicht mehr angezeigt und somit nicht mehr geadded werden.
HINWEIS: Die “.gitignore-Datei“ ist (wie an dem führenden Punkt zu sehen ist) versteckt und wird nur nach dem setzen des entsprechenden Häckchens im Dateimanager oder beim “ls -a“ angezeigt.

III.1.2 Branches nach aktuell zu bearbeitendem Thema benennen

Damit direkt aus dem Branchname ersichtbar ist was innerhalb des Branches bearbeitet wird, ist es sinnvoll den Name entsprechend zu wählen (Z.B. GUIBranch). Da die Branches mit dem Gitlab-Server synchronisiert werden können, ist es auch ohne weiteres möglich dass mehrere Personen an einem Branch arbeiten.