

2.1 Umsetzung von Architekturmustern

2.1.1

(i) Client-Server: Web-Anwendungen, wie z.B. unser moodle

Im moodle übernehmen Studenten die Client- und das System auf dem das moodle gehostet wird die Serverrolle. Die Anwendungslogik wird auf eine Menge von Studenten und Subsysteme von moodle verteilt.

(ii) Peer-to-Peer: Bittorrent

Bittorrent ist eine Umsetzung der Peer-to-Peer Architektur.

Peers werden durch einen Server/ Broadcast gefunden. Diese werden dann zu seeders und leachers, also Kommunizieren untereinander und nicht mit dem Server.

(iii) Pipe-and-Filter: Unix-Filter

Unix Pipes sind reguläre Ausdrücke, die compiliert werden. Dabei ist jeweils ein Filter/ eine Funktion durch ein Pipe-Symbol '|' getrennt und dessen Aussage wird weitergereicht.

(iv) Sharding: Data-Bases, wie z.B. oracle

Datenbanksysteme benutzen unter anderem Sharding (wenn gut implementiert in Kombination mit Load-Balancing) um Daten aufzuteilen und zu speichern.

2.1.2

Peer-to-Peer und Pipe-and-Filter sind ungeeignet.

Grund dafür sind die in der Vorlesung genannten Nachteile. So kann bei Peer-to-Peer der Server, durch den die Peers gefunden werden zu einem Engpass werden. Bezahlungen sollten schnell erfolgen können. Ferner sollten Konten eingerichtet und verwaltet werden können. Dies ist mit einer Peer-to-Peer Architektur nur erschwert möglich.

Pipe-and-Filter ist eine Architektur, die rein Funktional über das durchlaufen von Filtern funktioniert. Hierbei sind unter anderem Deadlocks möglich und das Verwalten, sowie I/O operationen durch Filter sind ebenfalls nur erschwert möglich.

2.2 Model-View-Whatever

Wie grenzen sich die Muster untereinander ab?

Wie wird jeweils der Fluss von Daten zwischen den Komponenten realisiert?

Wie reagiert das System jeweils auf Benutzereingaben?

Bei einem Model-View-Controller Modell korreliert jede Aktion in einem View mit dem zu korrespondierenden Controller. Der Controller entscheidet welche View angezeigt wird.

Es gibt im Model-View-Controller mehrere Views, zu denen je ein Controller existieren sollte. Der Controller nimmt jeweils die Änderungen am Model vor, die für den ihm zugehörigen View relevant sind.

Beim Model-View-Controller Modell ist eine klare Top-Down Hierarchie vorhanden.

Der Benutzer bedient die Gui, bzw. interagiert mit der View, diese Teilt das dem zuständigen Controller mit, der das Model editiert/ Updates vornimmt. Das Modell füllt dann den View.

Bei einem Model-View-Presenter Modell funktioniert der Presenter ähnlich zum Controller, jedoch verbindet der Presenter das Modell mit der View in einer 'dynamischeren' Weise. Jede Aktion im View geht direkt zum Presenter, der durch ein Interface mit dem View kommuniziert. Im Presenter ist also sämtliche Logik implementiert. Model und View werden so 'dumm' wie möglich gehalten. Benutzereingaben werden also über die Logik des Presenters verarbeitet, der dann View und Modell während der Laufzeit verbindet und Änderungen vornimmt.

Das Model-View-ViewModel Modell überlässt der View mehr Eigenständigkeit in ihrer Darstellung. Das Model benachrichtigt das ViewModel, dieses benachrichtigt die View lediglich, dass eine Änderung passiert ist, die View übernimmt dann die Darstellungsänderungen, statt direkt durch Propagation des Controllers. Über das ViewModel werden Komponenten der View mit Daten des Model verbunden, dementsprechend muss das ViewModel auch das Model modifizieren können und das Model die View über Veränderungen benachrichtigen.

Ähnlich zum Model-View-Controller werden also Benutzer eingaben bis zum Model durchgereicht. Jedoch direkter als über den Controller, da Daten nahezu direkt in Verbindung stehen. Das Model füllt also indirekt die View durch die Zwischeninstanz eines ViewModel, das die Verbindungen herstellt.

2.3 git

Mit # gekennzeichnete Lines bezeichnen Kommentare.

Das Editieren der Datei, musste nicht angegeben werden, deshalb ist es auskommentiert.

2.3.1

(i)

```
# S1:
$ git init
$ touch data.txt
$ git add data.txt
$ git commit -m "first commit"
# S2:
$ git checkout -b fork
# S3:
$ echo f > data.txt
$ git commit data.txt -m "fork commit"
$ git checkout master
$ echo m > data.txt
$ git commit data.txt -m "master commit"
# S4:
$ git merge fork
# An dieser Stelle werden wir auf einen merge-conflict aufmerksam gemacht
```

```
# Konflikt loesen z.B. mit git mergetool
$ git commit -m "resolved merge-conflict"
```

(ii)

```
* 9e58136 (HEAD -> master) resolved merge-conflict
|\
| * bd51b50 (fork) fork commit
* | 34835a4 master commit
|/
* d1cdcc3 first commit
```

2.3.2

(i)

```
# S1:
$ git init
$ touch data.txt
$ git add data.txt
$ git commit -m "first commit"
# S2:
$ git checkout -b fork
# S3:
$ echo f > data.txt
$ git commit data.txt -m "fork commit"
$ git checkout master
$ echo m > data.txt
$ git commit data.txt -m "master commit"
# S4:
$ git rebase master
# An dieser Stelle werden wir auf den rebase-conflict aufmerksam gemacht
# Konflikt loesen
$ git add data.txt
$ git rebase --continue
# S5:
$ git checkout master
$ git merge fork
```

(ii)

```
* c8ef802 (HEAD -> master, fork) fork commit
* 59b30b1 master commit
* 69762a1 first commit
```

2.3.3

Eine Rebase-Operation ermöglicht es Commits, die nicht Zielführend oder sogar schädlich für die Applikation/ das System waren aus der Historie zu löschen, damit also unzugänglich zu machen und den Stand des Branches auf den der Produktionsreifen Version zu setzten. Der Commit-Verlauf ist also sauber und sogar frei von Merge-Commits.

Dies ist bei einer Merge-Operation nicht der Fall, jedoch muss man hierbei die Branches nicht gleich zerstören und hat mehr Freiheiten bei der Weiterentwicklung. Man erhält durch das Aufgeben eines gewissen Grades an Übersichtlichkeit ein stabileres Projekt, denn bei Rebase-Operationen kann schnell ein Fehler unterlaufen und eine Menge an Kontext verloren gehen. Ferner kann leicht durch eine Rebase-Operation der Workflow ins stocken geraten.