Empirische Untersuchungen des Side-by-Side Programming

Ulrich Stärk
Freie Universität Berlin

Mai 2008

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben. Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

_____

Ulrich Stärk

Inhaltsverzeichnis

1 Einleitung **5**

2 Datenerhebung **7**
2.1 Workshop . . . 7
2.1.1 Hibernate . . . 8
2.1.2 Spring . . . 11
2.1.3 Tapestry . . . 14
2.1.4 Ablauf . . . 16
2.2 Experiment . . . 17

3 Auswertung **20**
3.1 Verarbeitung der Daten . . . 20
3.2 Probleme bei der Verarbeitung . . . 21
3.2.1 Codec-Probleme . . . 21
3.2.2 Synchronität der Videos . . . 22
3.2.3 Unterschiedliche Zeitbasis . . . 23
3.2.4 Zu große Dateien . . . 23
3.3 Erste Analysen . . . 24
3.3.1 Auswertung der Fragebögen . . . 25
3.4 Analyse der Videos . . . 27
3.4.1 Grounded Theory . . . 27
3.4.2 Kodierblickwinkel . . . 28
3.4.3 Kodierschema . . . 29
3.4.4 Die Analysesoftware . . . 31
3.4.5 Das Vorgehen . . . 32

4 Ergebnis **33**
4.1 Arten der Zusammenkunft . . . 33
4.1.1 Projektwissen . . . 33
4.1.2 Wissen . . . 37
4.1.3 Strategie . . . 39
4.1.4 Planung . . . 41
4.1.5 Integration . . . 42
4.1.6 Fehlersuche . . . 46
4.1.7 Bemerkung . . . 47
4.2 Kenntnisstand und Verhältnis der Probanden zueinander . . . 49
4.3 Zahlen . . . 49

5 Fazit **51**

6 Weiterführende Arbeiten **53**

Anhänge **56**

Anhang A Aufgabe des Experiments **56**

Anhang B Fragebogen 1 **58**

Anhang C Fragebogen 2 **63**

Anhang D E-Mail an Alistair Cockburn **67**

Anhang E Antwort von Alistair Cockburn **67**

Anhang F Schulungsmaterialien **68**
F.1 Einführung in Hibernate 68
F.2 Vererbung und Assoziationen mit Hibernate 73
F.3 Spring und Hibernate 76
F.4 Tapestry 81
F.5 Tapestry - Formulare 85
F.6 Tapestry - ASOs 89

1 Einleitung

Extreme Programming (XP) [1] ist eine sogenannte leichtgewichtige oder agile Softwareentwicklungsmethode, bei der es um den Einsatz bestimmter Techniken geht, unter anderem um qualitativ hochwertigeren Code zu produzieren. Eine wichtige dieser Techniken ist das Pair Programming (PP). Hierbei arbeiten zwei Programmierer gemeinsam an einem Computer mit nur einer Maus und einer Tastatur. Sie analysieren die Anforderungen, Entwerfen ein Softwareartefakt und schreiben Testfälle und das Softwareartefakt an sich [1]. Dabei helfen sie sich gegenseitig bei der Aufgabe zu bleiben und greifen auf das Wissen das anderen zurück. Beim Pair Programming gibt es idealerweise zwei Rollen. Der Driver ist derjenige, der die Maus und Tastatur bedient. Er schreibt den Code und wird vom Observer oder Navigator dabei unterstützt.

Diverse Studien haben Vorteile beim Einsatz von Pair Programming gegenüber der Programmierung durch einen einzelnen Programmierer ermittelt. Nosek [2] fand heraus, dass Paarprogrammierer für die Erledigung einer Aufgabe nur 70 % der Zeit benötigten, die Soloprogrammierer brauchten und dabei qualitativ hochwertigeren Code ablieferten. Cockburn und Williams [3] berichten, dass zwei Paarprogrammierer zusammengerechnet gerade einmal 15 % mehr Zeit als ein Soloprogrammierer für die Lösung einer Aufgabe benötigten. Diesem Mehraufwand standen 15 % weniger Defekte gegenüber. Somit macht der Einsatz von Pair Programming auch aus ökonomischer Sicht Sinn. Williams et al. [4] berichten sogar von einer Zeitersparnis von 40-50 % bei gleichzeitiger Reduzierung der Fehlerrate. Williams und Kessler [5] schreiben, dass Paarprogrammierer weniger Defekte produzieren, weniger Zeit benötigen und glücklicher sind. Paarprogrammierung verbessert auch das Teamwork, fördert den Wissenstransfer, so dass sich die Paarprogrammierer besser mit dem Gesamtsystem auskennen, und Paarprogrammierer können von der Arbeitsweise und der Herangehensweise ihres Partners an Probleme lernen [5].

Allerdings empfinden manche Menschen die Arbeit im Paar beim Pair Programming als zu intensiv [6]. Auch Williams et al. [4] berichten von Paarprogrammierern, die bestimmte Aufgaben wie das stumpfe Programmieren an sich lieber alleine und komplexere Aufgaben wie Design und Analyse lieber im Paar erledigen. Selbst Beck führt bestimmte Faktoren wie sozialen Abstand — der Mindesabstand zwischen zwei Personen, so dass sich beide wohl fühlen —, Hygiene und persönliche Gefühle an, die zu einem Gefühl des Unwohlseins führen können und somit die Arbeit im Paar behindern [1]. Außerdem besteht die Gefahr, dass die Rollenaufteilung starr bleibt, also ein Entwickler ständig in die Rolle des Drivers, der andere ständig in die des Observers verfällt [7].

Um einige der positiven Effekte des Pair Programming mitzunehmen, gleichzeitig aber den Entwicklern Freiraum für ihre eigenen Aufgaben zu lassen, schlägt Cockburn das Side-by-Side Programming vor, bei dem den Programmierern zwei Computer zur Verfügung stehen. Die zwei Entwickler arbeiten nah genug beieinander, so dass sie den Bildschirm des jeweils anderen leicht einsehen können, aber jeder arbeitet an seiner eigenen Aufgabe. Trotzdem kann jederzeit der eine den anderen bitten, sich ein Stück Code anzusehen, bei der Fehlersuche zu

helfen, usw. Sobald diese gemeinsame Arbeit erledigt ist, kehrt jeder wieder an seine eigene Aufgabe, beispielsweise der Programmierung einer bestimmten Komponente, am eigenen Rechner zurück.

In [7] haben die Autoren neben dem Vergleich von Paarprogrammierung mit der Programmierung durch einen einzelnen Entwickler zusätzlich das Side-by-Side Programming untersucht. So berichten sie, dass Side-by-Side Programming schneller als die Paarprogrammierung ist, dabei aber weniger Mehraufwand (s.o.) als bei der Paarprogrammierung mit sich bringt. Allerdings scheint ein Effekt der Paarprogrammierung, nämlich dass sich beide Entwickler gleich gut mit dem Code auskennen, beim Side-by-Side Programming nicht so stark ausgeprägt zu sein. Für einen kleinen Change Request nach der Programmierung benötigten die Side-by-Side Programmierer mehr Zeit als die Paarprogrammierer.

[7] ist leider die einzige Quelle, die das Side-by-Side Programming untersucht. Ansonsten ist das Side-by-Side Programming noch gänzlich unerforscht. Deshalb versuche ich mit dieser Arbeit zum Wissen über das Side-by-Side Programming beizutragen. Am Lehrstuhl für Software Engineering am Institut für Informatik der Freien Universität Berlin werden derzeit die Mikroprozesse beim Pair Programming anhand von Audio- und Videoaufzeichnungen von Paarprogrammierern bei der Arbeit untersucht. Es gibt zwar viele Studien, die von Vorteilen des Pair Programming gegenüber der Einzelprogrammierung berichten, die dafür verantwortlichen Prozesse sind aber noch kaum erforscht. In diesem Kontext soll auch meine Arbeit stehen. In dieser Arbeit geht es nicht um eine quantitative Evaluation bestimmter Kennzahlen beim Einsatz von Side-by-Side Programming. Vielmehr möchte ich die Prozesse und Abläufe beim Side-by-Side Programming erforschen. Dazu führe ich ein Experiment durch, bei dem die Teilnehmer beim Side-by-Side Programming aufgezeichnet werden. Anhand dieser Video- und Audioaufzeichnungen möchte ich die Vorgänge beim Side-by-Side Programming näher beleuchten und Theorien über sie aufstellen.

2 Datenerhebung

Um an Audio- und Videodaten von Programmierern, die Side-by-Side Programming praktizieren, zu gelangen, habe ich ein Experiment durchgeführt. Bei diesem Experiment saßen jeweils zwei Programmierer nah beieinander, so dass jeder den Bildschirm des anderen leicht einsehen konnte. Jedem Paar standen also zwei Computer zur Verfügung. Mittels einer speziellen Software wurde der Bildschirminhalt aufgezeichnet. Über eine Webcam, die auf dem Bildschirm stand, wurde zusätzlich ein Bild des Programmierers und der Ton aufgezeichnet und in das Video des Bildschirminhalts eingeblendet. Anders als bei klassischen Experimenten gab es bei diesem Experiment keine Kontrollgruppe. Dies ist auch nicht nötig, da es nicht um den Vergleich von bestimmten Kenngrößen geht. Außerdem übte ich keinerlei Kontrolle über die Zusammensetzung der Paare aus.

Um Probanden zur Teilnahme an diesem Experiment zu gewinnen, habe ich einen viertägigen Workshop über die Entwicklung von Webanwendungen mit Java angeboten. Die Teilnehmer erklärten sich im Gegenzug bereit, am letzten Tag an einem Experiment teilzunehmen.

2.1 Workshop

Der Workshop fand vom 13.08.2007 bis zum 16.08.2007 [1] während der vorlesungsfreien Zeit in den Räumen des Fachbereichs Informatik der FU Berlin statt. Da er auf Studenten höherer Semester abzielte, wurde er auf Mailinglisten von Lehrveranstaltungen der höheren Semester sowie auf der Webseite der AG Software Engineering angekündigt. Ziel war es Studenten zu gewinnen, die alle schon mindestens drei Jahre Erfahrung in der Programmierung mit Java hatten und somit am ehesten mit professionellen Softwareentwicklern vergleichbar sind. Das Thema Web-Entwicklung wurde deshalb gewählt, weil die Entwicklung von Web-Anwendungen gerade zu Zeiten des *Web 2.0* eine sehr wichtige Rolle spielt und deshalb für die Praxis relevant ist. Außerdem lässt sich eine Aufgabe im Bereich einer mehrschichtigen Webanwendung leicht in kleinere Arbeitsschritte zerlegen und bietet somit den Programmierern die Möglichkeit parallel zu arbeiten. Genau das ist beim Side-by-Side Programming gewünscht und genau das will ich untersuchen.

Die im Workshop behandelten Komponenten für eine Webanwendung waren die populären Frameworks Hibernate, Spring und Tapestry. Im Workshop wurde eine dreischichtige Webanwendung zur Verwaltung einer kleinen Bibliothek entwickelt. Ziel war das Erlernen des Umgangs mit den drei Frameworks um die Teilnehmer auf das Experiment vorzubereiten. Am Workshop nahmen insgesamt 11 Studenten teil, die sich in fünf Paare aufteilten (vergleiche Abschnitt 3.3.1).

[1] Der Workshop musste vor der eigentlich Bearbeitungszeit der Masterarbeit stattfinden, da er in der vorlesungsfreien Zeit liegen sollte. Ansonsten wäre es schwierig geworden Räume und Teilnehmer zu finden, da er ganztätig statt fand.

2.1.1 Hibernate

Hibernate dient zur Abbildung von Objekten auf das relationale Modell. Dieses Vorgehen wird Object/Relation Mapping (ORM) genannt. Die Abbildung geschieht dabei transparent und automatisch unter Zuhilfenahme von Metadaten, welche die Abbildungen beschreiben. Sind diese Abbildungen erst einmal eingerichtet, muss der Programmierer im besten Fall keinen weiteren Code für das Laden aus und das Speichern in die Datenbank schreiben und kann sofort über die API des ORM Tools mit den in der Datenbank gespeicherten Daten arbeiten. Die Notwendigkeit für die Benutzung eines fertigen ORM Tools liegt in den Kosten für selbstgeschriebenen JDBC Code und seine Unflexibilität. In [8] argumentieren Christian Bauer und Gavin King — die Gründer und Kernentwickler des Hibernate Projekts — dass Code, der sich um die Persistenz von Objekten kümmert, zu den lästigsten Teilen einer Anwendung gehören kann. ORM Werkzeuge nehmen dem Entwickler einen Großteil dieser Arbeit ab. Außerdem entkoppelt der Einsatz eines ORM Werkzeugs den Persistenzcode von der Abbildung der Daten in der Datenbank. Bei handgeschriebenem Code muss bei einer Änderung der Datenabbildung oft auch der Persistenzcode angepasst werden. Der Einsatz eines ORM Werkzeugs macht das überflüssig. Lediglich die Metadaten, welche die Abbildung beschreiben, müssen angepasst werden. Ein weiterer wichtiger Punkt ist die Unabhängigkeit von einem Datenbankhersteller. So ist es mit den meisten ORM Werkzeugen problemlos möglich, den Datenbankhersteller zu wechseln ohne den vorhandenen Quelltext anzupassen. Es muss dem ORM Werkzeug lediglich mitgeteilt werden, mit was für einer Datenbank es sich gerade unterhält.

Die für die Interoperabilität von ORM-Werkzeugen wichtigen APIs sind im EJB 3.0 [9], [10] Standard geregelt. Er spezifiziert Enterprise Java Beans (EJB) und gliedert sich in zwei Teile: Teil 1 [9] handelt von Session Beans, Message Driven Beans, dem Deployment, dem Programmiermodell usw. Der zweite Teil, die Java Persistence API (JPA) [10], behandelt die Persistenz von Objekten. Die JPA spezifiziert die ORM Metadaten, die Interfaces für den Zugriff sowie eine eigene Anfragesprache, die JPA Query Language (JPA QL). Bei der Entwicklung der JPA flossen viele Konzepte von Hibernate ein [8]. So entspricht bspw. die JPA QL mit einigen Ausnahmen der Hibernate Query Language (HQL), der Hibernate eigenen Anfragesprache. Letztere besitzt im Vergleich zur JPA QL einige zusätzliche Funktionen. Das Ziel der JPA war die Schaffung eines einheitlichen Standards, dessen Umsetzungen untereinander austauschbar sein sollen und auch außerhalb eines EJB 3.0 Applikationsservers laufen.

Hibernate besteht aus verschiedenen Komponenten. Hibernate Core bietet die Basisfunktionen eines ORM Werkzeugs und ist für sich genommen voll funktionsfähig. Es besitzt eine eigene, Hibernate spezifische API, versteht Anfragen in der HQL und wird mittels XML Dateien konfiguriert. Auch die Abbildung von Objekten auf das relationale Modell geschieht mit Hilfe von XML Dateien. Mit der Komponente Hibernate Annotations kann der Entwickler diese Abbildung mittels Java Annotationen auch direkt in den Klassen vornehmen. Ihm stehen dabei entweder die JPA oder die Hibernate Annotationen zur Verfügung. Letztere bieten wie auch bei der HQL mehr Möglichkeiten, bei ihrem Einsatz ist der

Austausch des ORM Werkzeugs gegen ein anderes, das die JPA beherrscht, dann aber ausgeschlossen. Der Hibernate EntityManager ist die Umsetzung der Programmier API der JPA. Aber auch hier ist der Rückgriff auf die Hibernate eigene API wieder möglich. Alle drei Komponenten zusammen liefern eine komplette Umsetzung der JPA. Daneben gibt es noch viele weitere Komponenten, welche die Funktionen von Hibernate erweitern. Exemplarisch sei hier Hibernate Search erwähnt. Hibernate Search bietet eine Integration mit Lucene, einem Tool zur Erstellung und Durchsuchung von Volltextindizes. Mit Hibernate Search ist es einfach möglich, Volltextsuche in einer Datenbank bereitzustellen. Abbildung 1 zeigt die einzelnen Komponenten von Hibernate und wie sie aufeinander aufbauen.

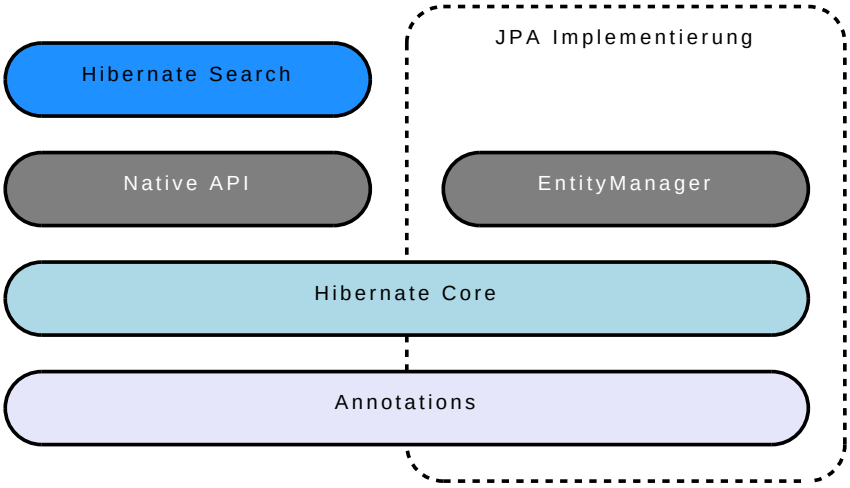


Abbildung 1: Übersicht über die Komponenten von Hibernate

Während des Workshops haben die Teilnehmer gelernt, was Hibernate ist, wie es in den Kontext der JPA einzuordnen ist und wozu es eingesetzt wird. Außerdem haben sie gelernt, Klassen und Assoziationen zwischen den Klassen sowie Vererbung mittels JPA Annotationen auf das relationale Modell abzubilden und mit der JPA API und dem Hibernate EntityManager zu arbeiten. Ein Beispiel für eine solche Abbildung geben die Abbildung 2, bei der ein Objekt vom Typ Cat auf das relationale Modell abgebildet wird. Die Annotation *@Entity* teilt dem Persistence Provider, in unserem Fall Hibernate, mit, dass das vorliegende Objekt eine zu persistierende Entität ist. Mit *@Table* kann der Entwickler bestimmte Eigenschaften der für diese Entität zu erstellenden Tabelle festlegen, in unserem Fall wird ein anderer Name für die Tabelle festgelegt. Beim Fehlen dieser Einstellung würde die Tabelle wie die Klasse, nämlich *Cat* benannt werden. Grundsätlich werden alle Felder einer Klasse in Attribute der Entität gewandelt. Durch bestimmte Annotationen an den Gettern und Settern lässt sich der Persistence Provider steuern. So können mit der Annotation *@Column* bestimmte Eigenschaften wie ein Standardwert festgelegt werden oder es kann festgelegt werden, ob dieses Feld leer sein darf (*NOT NULL*). Mit der Annotation *@Id* wird ein Attribut als Primärschlüssel markiert. Mit der Annotationen

```java
package workshop;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="KATZE")
public class Cat {

    private int id;
    private String name;
    private int age;

    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="CAT_ID")
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    @Column(nullable=false)
    public int getAge() { return age; }
    public void setAge(int age) {
        this.age = age;
    }

    @Column(nullable=false)
    public String getName() { return name; }
    public void setName(String name) {
        this.name = name;
    }
}
```

Abbildung 2: Einfaches Mapping mit Hibernate

@GeneratedValue lässt sich das Verhalten der Datenbank beim automatischen Generieren von Primärschlüsseln beeinflussen.

Mit den Annotationen *@OneToOne*, *@ManyToOne* und *@OneToMany* lassen sich Beziehungen zwischen Objekten analog zu den Beziehungstypen beim relationalen Modell abbilden. Dies ist in Abbildung 3 am Beispiel der Relation *Cat* dargestellt. Diese und andere Beispiele (vergleiche Anhang F) dienten dazu, den Teilnehmern die Funktionen von Hibernate zu verdeutlichen.

```
@Entity
public class Cat {

    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
    private int age;

    @OneToOne
    private Cat mate;

    @ManyToOne
    private Cat mother;

    @OneToMany(mappedBy="mother")
    private Set<Cat> kittens;
}
```

Abbildung 3: Mapping von Beziehungen mit Hibernate

2.1.2 Spring

Um die ansonsten entkoppelten Schichten und Teile eines komplexen Softwaresystems miteinander zu verbinden ohne Abhängigkeiten zwischen den einzelnen Teilen und Schichten einzuführen, benötigt es spezieller Software. In der Java Welt spielt hier die *Java Platform, Enterprise Edition (JEE)* eine wichtige Rolle. JEE spezifiziert diverse Dienste und Schnittstellen, die genutzt werden können, um die Schichten und Teile eines Softwaresystems miteinander zu koppeln. Dazu stehen diverse APIs bereit, gegen die programmiert werden kann. Das so erstellte Softwaresystem ist dann in einem JEE Container, der die in JEE spezifierten Dienste und Schnittstellen bereitstellt, lauffähig. Grundsätzlich unterscheidet man zwei Arten von JEE Container. Ein EJB Container stellt die Laufzeitumgebung für Enterprise Java Beans und somit die Geschäftslogik bereit, ein Web Container für Servlet und Java Server Pages (JSP). Komplette JEE Server, die sowohl einen Web als auch einen EJB Container enthalten, nennt man auch Application Server. Beispiele für solche Server sind JBoss Application Server, IBM WebSphere, BEA WebLogic und SAP NetWeaver. Einen reinen Web Container findet man beispielsweise in Apache Tomcat oder Jetty.

Der Einsatz von JEE und eines Application Servers für die Koppelung der ein-

zelnen Schichten eines Softwaresystems geht aber mit einer hohen Komplexität einher, gerade wenn es sich um kleinere Systeme handelt. In den letzten Jahren sind daher sogenannte leichtgewichtige Container entstanden, welche die Koppelung der Schichten und Teile eines Softwaresystems übernehmen, ohne dabei die Komplexität eines kompletten JEE Containers mitzubringen. Diese leichtgewichtigen Container setzen das Prinzip der Inversion of Control (siehe nächster Absatz) um. Der Unterschied zu einem vollwertigen JEE Container besteht darin, dass sich diese leichtgewichtigen Container fast ausschließlich um die Koppelung von Objekten miteinander kümmern. Viele Funktionen eines JEE Containers wie beispielsweise Transaktionsmanagement und Remoteaufrufe sind in einigen dieser leichtgewichtigen Containern nicht vorhanden. Andere wiederum (z.B. Spring) können einige Features eines JEE Containers dazuschalten, so dass das Beste aus beiden Welten genutzt werden kann.

Für die Koppelung der Schichten und Teile eines Softwaresystems gibt es grundsätzlich zwei Möglichkeiten. Bei der Benutzung einer Bibliothek ruft der Programmierer in seinem Programm eine Funktion in der Bibliothek auf, wartet auf die Rückgabe und fährt in seinem Code fort. Er hat die Kontrolle über den Aufruf. Die Koppelung der einzelnen Komponenten ist somit seine Aufgabe. Bei Inversion of Control hingegen besitzt das Framework die Kontrolle. Der Programmierer teilt dem Framework mit, an welchen Stellen es seinen Code ausführen soll. Dies funktioniert bspw. durch die Registrierung von Event Handlern beim Framework (z.B. .NET) oder die Implementierung spezieller Interfaces wie es bei einem EJB Container üblich ist. Die Koppelung der Komponenten übernimmt damit das Framework.

Das Spring Framework - kurz Spring - ist ein leichtgewichtiger Container, der eine spezielle Art von Inversion of Control, nämlich die Dependency Injection, einsetzt. Abhängigkeiten zwischen den einzelnen Objekten und Komponenten werden durch Injektion der benötigten Objekte aufgelöst. Die Injektion kann dabei entweder über einen Konstruktor oder eine Setter Methode geschehen. In einer Konfigurationsdatei werden Objekte, die in der Regel Plain Old Java Objects (POJOs) ohne komplexe Abhängigkeiten sind, registriert und dann von Spring verwaltet. Diese Objekte können dann gegenseitig oder in weitere Objekte injiziert werden. Dazu muss Spring lediglich mitgeteilt werden, welche Abhängigkeiten existieren und ob die Injektion über den Konstruktor einer Klasse oder eine Setter Methode geschehen soll. Spring koppelt also auf einfache Art und Weise die verschiedenen Teile und Objekte einer Anwendung, ohne dabei die Komplexität eines kompletten JEE Containers mit sich zu bringen und ohne dass der Entwickler Entwurfsmuster wie Factory oder Service Locator umsetzen muss.

```
<bean id="catDao" class="inf.fu.berlin.de.model.dao.jpa.CatDaoJpa" />

<bean id="catService"
  class="inf.fu.berlin.de.service.impl.CatServiceImpl">
  <property name="catDao" ref="catDao" />
</bean>
```

Abbildung 4: Konfiguration eines Objektes in Spring

```java
@Transactional(readOnly=true)
public class CatServiceImpl implements CatService {

    private CatDao catDao;

    public Cat getCatById(int catId) throws DataAccessException {
        return catDao.getCatById(catId);
    }

    public Cat getCatByNameCriteria(String catName) throws DataAccessException {
        return catDao.getCatByNameCriteria(catName);
    }

    public Cat getCatByNameJpaQl(String catName) throws DataAccessException {
        return catDao.getCatByNameJpaQl(catName);
    }

    @Transactional(readOnly=false)
    public void saveCat(Cat cat) throws DataAccessException {
        catDao.saveCat(cat);
    }

    @Transactional(readOnly=false)
    public void reattachCat(Cat cat) throws DataAccessException {
        catDao.reattachCat(cat);
    }

    public void setCatDao(CatDao catDao) {
        this.catDao = catDao;
    }
}
```

Abbildung 5: Darstellung eines von Spring verwalteten Services

Ein Beispiel dafür, wie die Verwaltung von Objekten in Spring konfiguriert wird, zeigt Abbildung 4. Hier wird Spring angewiesen ein Objekt vom Typ *CatDaoJpa* zu erstellen und unter dem Namen *catDao* verfügbar zu machen. Anschließend soll ein Objekt vom Typ *CatServiceImpl* unter dem Namen *catService* verfügbar gemacht werden, und das zuvor erstellte Objekt mit dem Namen *catDao* soll dem Feld *catDao* des Objekts vom Typ *CatServiceImpl* zugewiesen werden. Dies geschieht über den Aufruf eines Setters in der Klasse *CatServiceImpl*. In Abbildung 5 ist die Klasse *CatServiceImpl* dargestellt. Beim Start von Spring wurde der Setter *setDao* aufgerufen, mit dem diesem Service eine DAO Klasse zugeordnet wurde. Spring hält dann eine Instanz des Service vor, der mit der DAO Klasse verknüpft ist und bei Bedarf beispielsweise in eine Tapestry Seitenklasse injeziert werden kann.

Zusätzlich zu dieser grundlegenden Funktionalität unterstützt das Spring Framework den Entwickler noch mit einer Menge zusätzlicher Funktionen. So bietet Spring Transaktionsmanagement — wobei die Abgrenzung der Transaktionen entweder per Annotationen oder per XML Konfigurationsdateien erfolgen kann —, Integration mit Hibernate und anderen Persistenzwerkzeugen, Unterstützung für Web Services, ein eigenes Web Framework und vieles mehr.

Den Teilnehmern am Workshop wurden die grundlegenden Funktionen von Spring sowie die Integration mit Hibernate und die Transaktionsverwaltung durch Spring vermittelt. Sie lernten ihre Komponenten vom Spring Framework verwalten zu lassen und Abhängigkeiten zwischen ihren Objekten von Spring auflösen zu lassen. So waren sie anschließend in der Lage ihre Geschäftslogik entkoppelt von der Persistenzschicht zu entwickeln und die Verbindung der beiden Schichten sowie Transaktionen von Spring verwalten zu lassen (Vergleiche dazu auch Anhang F).

2.1.3 Tapestry

Tapestry ist ein Framework für die Erstellung komplexer Webanwendungen mit Java. Bei klassischen Servlet Anwendungen muss sich der Programmierer um viele Dinge kümmern. Formularparameter müssen aus einer Anfrage ausgelesen und verarbeitet werden, eine Java Server Page (JSP) muss aktiviert werden, diese wiederum muss auf vom Servlet in einem HttpServletRequest gespeicherte Eigenschaften zurückgreifen, URLs müssen auf Servlets abgebildet werden, Sitzungsdaten müssen verwaltet werden, usw. All dies erfordert eine genaue Kenntnis über die Namen von Formularparametern und URLs. Schon ein kleiner Tippfehler in einer JSP und ein Link läuft ins Leere oder das Servlet verweigert seine Arbeit aufgrund eines falsch benannten Formularparameters. Anfrageorientierte Web Frameworks wie Struts oder Spring MVC nehmen dem Entwickler einen Teil der Aufgaben ab. Sie füllen POJOs automatisch mit Werten aus einem Formular oder unterstützen ihn bei der Verwaltung von Sitzungsdaten und helfen bei der Zuordnung von URLs auf Teile der Programmlogik - im Falle von Struts von URLs auf sog. Actions und von diesen auf JSPs je nach Ausgang der Aktion. Der Entwickler muss dennoch genau wissen wie seine Parameter und URLs heißen und wie der Ablauf von Anfrage, Verarbeitung und Antwort vonstatten gehen soll. Außerdem ist in größeren Projekten eine Arbeitsteilung zwischen Designern und Entwicklern schwierig, da die Vorlagen für die Generierung von HTML von JSP Tags durchzogen sind.

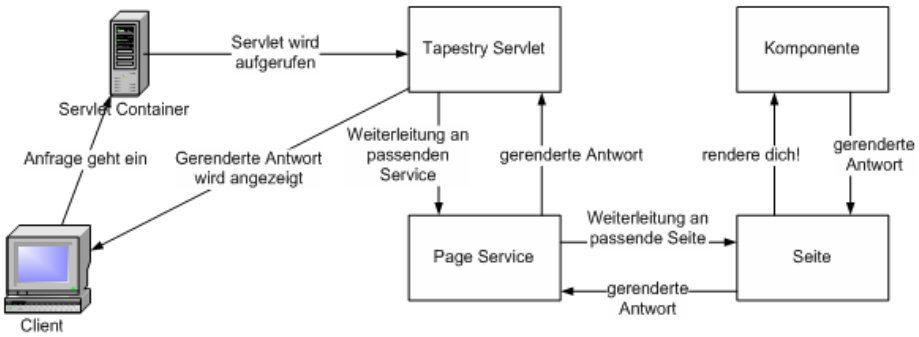


Abbildung 6: Ablauf einer Anfrage an eine Tapestry Seite

Tapestry bezeichnet sich selbst als komponentenorientiertes Framework. Damit soll zum Ausdruck gebracht werden, dass sich die Entwicklung von Webanwendungen nicht um den Zyklus von Anfrage, Verarbeitung und Antwort dreht. Vielmehr steht dem Entwickler eine Vielzahl von Komponenten zur Verfügung, welche für bestimmte Aufgaben zuständig sind. So gibt es Komponenten für die Darstellung eines Eingabefeldes oder die Iteration über eine Liste oder einfach eine Komponente für die Darstellung eines Textes auf der generierten Webseite. Wie bei einer Desktopanwendung platziert der Entwickler diese Komponenten auf einer Seite oder gruppiert sie zu einer eigenen, wieder verwendbaren Komponente, die dann auf einer Seite platziert wird. Diese Platzierung passiert in gewöhnlichen (X)HTML Seiten. Dazu werden bestehende HTML Elemente um ein zusätzliches Attribut erweitert, das nur von Tapestry interpretiert wird und bei der Ausgabe nicht mehr vorhanden ist. Somit können HTML Vorlagen für

Tapestry auch ohne Probleme von Designern in ihren HTML Editoren bearbeitet werden, ohne Kenntnis von der Seitenlogik oder speziellen Anweisungen wie JSP Tags haben zu müssen. Zu jeder Seite und zu jeder Komponente gehören außerdem eine Seiten- bzw. eine Komponentenklasse, welche die Eigenschaften einer Seite bzw. Komponente speichert und sog. Listener Methoden zur Verfügung stellt. Diese Methoden werden bei bestimmten Ereignissen wie bspw. einer Formularübermittlung oder dem Klick auf einen Button von Tapestry aufgerufen. Wird nun eine Anfrage an eine Seite gestellt, so durchläuft eine Seite alle ihre Komponenten und weist sie an, ihren HTML Code zu generieren. Am Ende fasst die Seite den gesamten HTML Code zusammen und übermittelt ihn an den Anfragesteller. Diese Ablauf ist in Abbildung 6 dargestellt. Hierbei muss sich der Entwickler nun überhaupt keine Gedanken mehr darüber machen, wie seine Formularparameter oder Seiten benannt werden. Tapestry übernimmt bei der Übermittlung eines Formulars diese Zuordnung automatisch. Auch Kenntnisse über URLs sind nicht mehr von Bedeutung. Der Programmierer muss Tapestry nur noch anweisen „baue mir hier einen Link zur Seite xyz“ und Tapestry kümmert sich um die Generierung eines Links und die Zuordnung einer URL zu dieser Seite.

```
<html jwcid="@Shell" title="Formular">
<body jwcid="@Body">
<p>
  <font color="red">
    <span jwcid="@Delegator" delegate="ognl:delegate.firstError">Fehler</span>
  </font>
</p>
<form
  jwcid="myForm@Form"
  delegate="bean:delegate"
  success="listener:onSuccess"
  clientValidationEnabled="true"
>
<p>
  <label jwcid="@FieldLabel" field="component:name">Name</label>
  <input
    jwcid="name@TextField"
    value="ognl:name"
    validators="validators:required"
    displayName="Name"
  />
</p>
<p>
  <label jwcid="@FieldLabel" field="component:alter">Alter</label>
  <input
    jwcid="alter@TextField"
    value="ognl:age"
    validators="validators:required,min=0"
    translator="translator:number,omitZero=true"
    displayName="Alter"
  />
</p>
<p><input jwcid="@Submit" /></p>
</form></body>
</html>
```

Abbildung 7: Template einer Tapestry Seite

In Abbildung 7 ist ein Beispiel für ein Template einer Tapestry Seite dargestellt. Diese Seite dient dem Anlegen neuer Objekte vom Typ *Cat* in der Datenbank. So wurde neben einigen Standardkomponenten wie *@Shell* und *@Body* auch noch die Komponente *@Form* auf der Seite platziert. Innerhalb dieser Kom-

ponente, die ein Formular darstellt, gibt es Komponenten für die Eingabe von Daten (*@TextField*) und für die Darstellung von Namen für die Eingabefelder (*@FieldLabel*). Außerdem gibt es eine Komponente für die Darstellung von Fehlern, beispielsweise bei der Eingabe von nicht erlaubten Werten (*@Delegator*). Das Formular ist über den Parameter *success* mit einer Listener Methode in der Seitenklasse verbunden, die aufgerufen wird, wenn das Formular abgeschickt wird und alle Eingaben korrekt sind.

Im Beispiel ist dies die Methode *onSuccess* (Vergleiche Abbildung 8). Diese Methode ruft die von Tapestry automatisch gefüllten Werte für Alter und Name der Katze ab und erstellt ein neues Objekt vom Typ *Cat*. In die Seitenklasse (Abbildung 8) wird außerdem ein von Spring verwaltetes Objekt vom Typ *CatService* (Vergleiche Abschnitt 2.1.2) injiziert, dass es uns nun ermöglicht, die gerade erstellte Katze zu speichern.

```
public abstract class Home extends BasePage {

    @Bean
    public abstract ValidationDelegate getDelegate();

    @InjectObject("spring:catService")
    public abstract CatService getCatService();

    public void onSuccess() {
        System.out.println("success");
        Cat cat = new Cat();
        cat.setName(getName());
        cat.setAge(getAge());
        getCatService().saveCat(cat);
    }

    public abstract String getName();
    public abstract int getAge();
}
```

Abbildung 8: Seitenklasse einer Tapestry Seite

Ein Großteil des Workshops war der Entwicklung von Webanwendungen mit Tapestry bedacht. Die Teilnehmer haben gelernt, eigene Seiten aus den in Tapestry enthaltenen Komponenten zu erstellen und ihre Seiten mit eigener Logik in den Listener Methoden zu versehen. Des Weiteren waren sie nach dem Workshop in der Lage, eigene Komponenten zu erstellen und in ihre Seiten einzubinden. Auch die Integration von Spring mit Tapestry wurde ihnen beigebracht, so dass sie aus der Präsentationsschicht auf ihre Geschäftslogik zugreifen konnten.

2.1.4 Ablauf

Während des Workshops bekamen die Teilnehmer immer eine Einführung in einige Aspekte der jeweiligen Technologie, die sie dann in anschließenden Übungen

anwenden sollten. An den ersten beiden Tagen bekamen die Teilnehmer Einführungen in Form von Präsentationen in die einzelnen Techniken und erlernten sie anschließend in praktischen Übungen (siehe Anhang F). Dabei gab es pro Tag drei Präsentationen á ca. 30 Minuten, der Rest der eingeplanten 8 Stunden stand für Übungen zur Verfügung.

Am ersten Tag bekamen die Teilnehmer eine Einführung in Hibernate sowie die Middleware Spring (siehe Anhänge F.1, F.2 und F.3). In den Übungen wurde der Grundstein der Webanwendung gelegt. Die Teilnehmer haben die für die Bibliotheksanwendung notwendigen Geschäftsobjekte identifiziert und mittels JPA Annotationen abgebildet. Im ersten Schritt erfolgte der Zugriff auf die Datenbank mit einer kleinen Konsolenanwendung und dem Hibernate EntityManager. Im Laufe des Tages programmierten die Teilnehmer die für die Bibliotheksanwendung benötigte Geschäftslogik dann in Form von Java Objekten (Plain Old Java Objects, POJO) und integrierten diese Logik und den Zugriff auf die Datenbank mittels des Spring Frameworks. Somit waren nach dem ersten Tag der Zugriff auf die Datenbank und Teile der Geschäftslogik realisiert.

Am zweiten Tag stand die Einführung in Tapestry und die Kombination mit Hibernate und Spring auf dem Lehrplan (siehe Anhänge F.4, F.5 und F.6). In der ersten Übung zu Tapestry sollten die Teilnehmer ihre erste dynamische Webseite mit Tapestry erstellen sowie den Umgang mit grundlegenden Komponenten von Tapestry üben. Anschließend erlernten sie den Umgang mit Formularen und verbanden die ersten erstellten Webseiten mit dem Spring Framework, um aus der Präsentationsschicht auf die Geschäftslogik zugreifen zu können. In der letzten Übung wurde der Umgang mit zustandsbehafteten Objekten, Lokalisierung sowie eigenen Komponenten vertieft.

Am dritten Tag wurde dann der Einsatz aller drei Frameworks geübt. Aufbauend auf die Aufgaben der Vortage erstellten die Teilnehmer eine Bibliotheksanwendung mit Möglichkeiten zur Benutzerverwaltung, Medienverwaltung, Ausleihe und Rückgabe von Medien.

Während des Workshops waren die Teilnehmer angehalten im Paar zu arbeiten. Für die Erfüllung der Geschäftsanforderungen stand den Paaren nur ein begrenzter Zeitrahmen zur Verfügung, so dass sie gezwungen waren, sich auf mehrere Rechner zu verteilen. Durch die Anordnung der Rechner — jeweils zwei nebeneinander stehende Rechner standen einem Paar zur Verfügung — praktizierten die Paare damit ohne es zu wissen Side-by-Side Programming.

2.2 Experiment

Am letzten Tag mussten die Probanden die Bibliotheksanwendung um zusätzliche Geschäftsanforderungen erweitern. Dabei wurde ihre Programmiertätigkeit im Rahmen eines Experiments aufgezeichnet. Für die Aufnahme des Bildschirminhaltes kam die Software Camtasia Studio zum Einsatz. Zusätzlich zum Bildschirminhalt zeichnete Camtasia Studio auch noch Bild und Ton des jeweils vor dem Computer sitzenden Programmierers mittels einer auf dem Bildschirm ste-

henden Webcam auf.

Wie an den Vortagen auch arbeiteten die Teilnehmer des Workshops in den Paarungen, in denen sie sich zu Beginn des Workshops zusammengefunden hatten. Je einem Programmiererpaar standen wieder zwei Rechner zur Verfügung. Bei dem Experiment handelt es sich nicht um ein Experiment im klassischen Sinne. So gibt es weder eine Kontrollgruppe noch wird Kontrolle über die Zusammensetzung der Probanden ausgeübt. Das ist aber auch nicht notwendig, da ich keine Kennzahlen vergleichen möchte, sondern die Vorgänge beim Side-by-Side Programming beleuchten will.

Die Aufgabe der Programmierer (siehe Anhang A) bestand darin, die Bibliotheksanwendung um die Angabe eines Standorts für ein Exemplar eines Mediums zu erweitern. Es sollte möglich sein neue Standorte anzulegen, zu editieren und zu löschen. Bei der Löschung eines Standorts, der noch einem Exemplar zugewiesen ist, sollte dem Benutzer eine Fehlermeldung angezeigt werden. Außerdem sollten zwei Seiten zur Darstellung von Standorten angelegt werden. Auf einer Seite sollte zu einem gewählten Standort alle dort eingestellten Exemplare und Details zu diesen angezeigt werden, auf einer weiteren Seite sollten alle Standorte mit ihren jeweils dort eingestellten Exemplaren dargestellt werden. Für die Bearbeitung der Aufgabe standen den Teilnehmern 2 $^1/_2$ Stunden Zeit zur Verfügung. Nach Ablauf dieser Zeit wurden die Probanden gebeten ihre Arbeit einzustellen.

Die Aufgabe baute auf der an den Vortagen entwickelten Bibliotheksverwaltung auf. Für die Paare, die es an den Vortagen nicht geschafft hatten die für die Aufgabe nötigen Entities, DAOs und Service-Klassen zu entwickeln, stand eine Musterlösung bereit, die allerdings von niemanden in Anspruch genommen wurde.

Für die Lösung der Aufgabe mussten die Teilnehmer folgende Teilschritte durchführen:

- Erstellung einer Entity Standort, Versehung derselben mit JPA Annotationen und Verknüpfung mit der Entity Exemplar
- Erstellung einer DAO-Klasse zum Laden von Standorten aus der Datenbank, zur Speicherung modifizierter Standorte und zum Löschen von Standorten
- Erstellung einer Service-Klasse für den Zugriff auf die DAO-Klasse und zur Behandlung von Fehlern etc.
- Erstellung einer Seite zum Anlegen neuer Standorte
- (optional) Erstellung einer Seite zum Ändern eines Standortes (oder Integration in die Seite zum Anlegen neuer Standorte)
- Erstellung einer Webseite zum Auflisten aller Standorte
- Erstellung einer Webseite zum Auflisten aller Exemplare und ihrer Details zu einem bestimmten Standort

Diese Teilschritte wurden ihnen allerdings nicht aufgezeigt, die Paare waren völlig frei in der Organisation ihrer Arbeit.

Nach dem Experiment füllten die Teilnehmer zwei Fragebögen mit insgesamt 27 Fragen aus. Der erste Fragebogen (Anhang B) bezog sich auf die gerade bearbeitete Aufgabe, im zweiten Fragebogen (Anhang C) sollten die Teilnehmer Auskunft über sich selbst geben.

3 Auswertung

3.1 Verarbeitung der Daten

Nach dem Experiment mussten die aufgezeichneten Daten in ein für die weitere Analyse geeignetes Format gebracht werden. In einem ersten Schritt wurden die von Camtasia Studio aufgezeichneten Video- und Audiodaten konvertiert. Dabei wurde von Camtasia Studio ein AVI-Video erzeugt, das den Bildschirminhalt des jeweiligen Programmierers zeigt und unten rechts das aufgezeichnete Kamerabild einblendet. Hinterlegt ist das Video mit einer Mono Tonspur des aufgezeichneten Tons. Das Video wurde mit einem Codec des Camtasia Herstellers Techsmith, dem Tachsmith Screen Capture Codec (TSCC), codiert. Der Ton wurde unkomprimiert übernommen. Da Camtasia bei der Aufzeichnung der Programmierer angwiesen war das Video mit 4 Bildern pro Sekunde (frames per second, FPS) aufzuzeichnen, hatte das resultierende Gesamtvideo mit dem Kamerabild auch 4 FPS. Abbildung 9 zeigt den Screenshot eines Videos, wie es von Camtasia Studio exportiert wurde. Im unteren rechten Bereich ist das Kamerabild des Probanden eingeblendet.

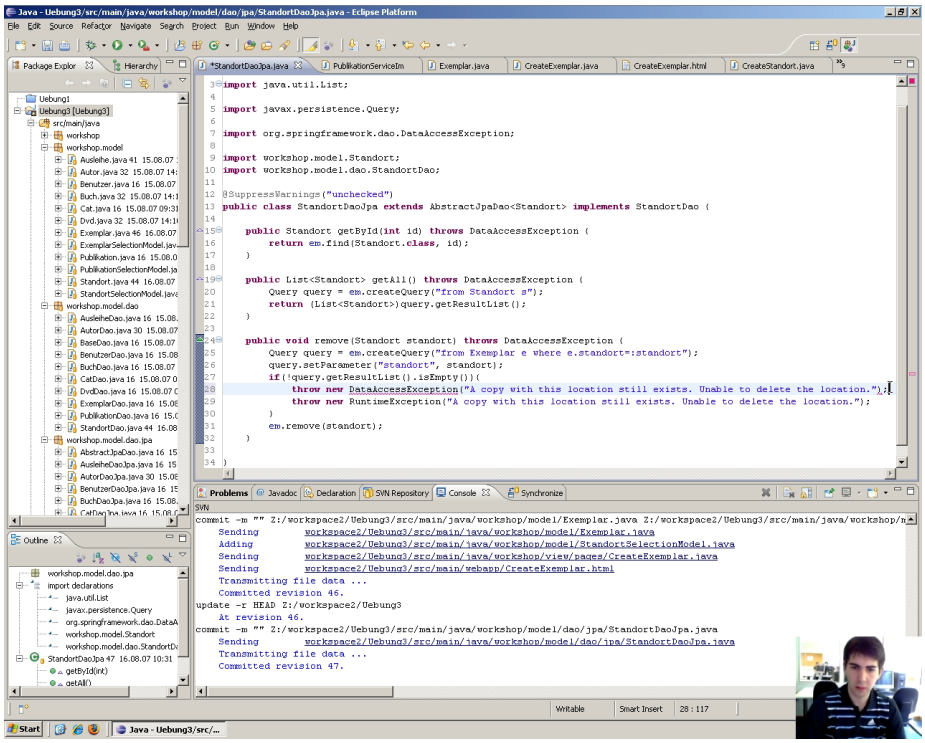


Abbildung 9: Video nach dem Export aus Camtasia Studio

Im nächsten Schritt wurden die zwei Videos je eines Paares zu einem großen Video für die spätere Analyse kombiniert. Dazu kam die Software Adobe Premiere zum Einsatz. Die beiden Einzelvideos eines Paares wurden dabei nebeneinander

angeordnet, so dass ein großes, 2560 Pixel breites und 1024 Pixel hohes Video mit Stereoton und 10 FPS entstand. Die Tonspur des links angeordneten Videos wurde auf den linken Kanal des Videos gelegt, die des rechts angeordneten Videos auf den rechten Kanal.

Nach der Vorbereitung des Videos habe ich es mittels des FrameServer Plugins für Adobe Premiere von Satish Kumar. S. (http://www.debugmode.com/frameserver/) an VirtualDub (http://www.virtualdub.org/) umgeleitet und dort zu einem AVI Video mit 4 FPS zusammengesetzt. Für die Komprimierung der Videodaten kam der Xvid Codec (http://www.xvid.org/) zum Einsatz. Die resultierenden Videodateien hatten eine Größe von 2 bis 2,5 Gigabyte. Abbildung 10 zeigt einen Screenshot eines solchen Videos.

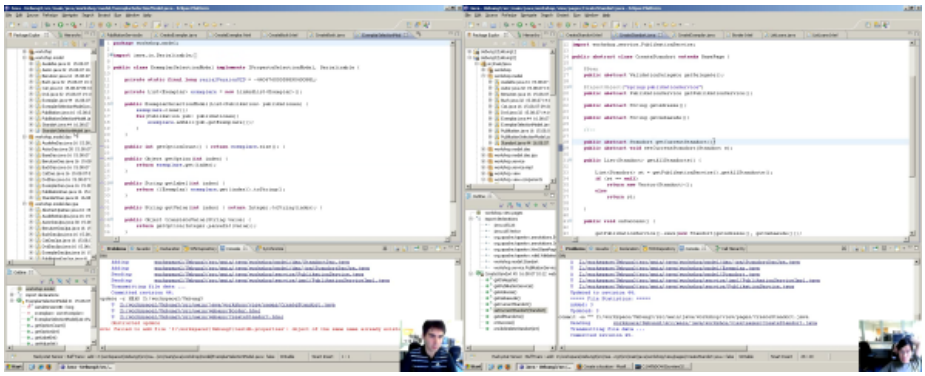


Abbildung 10: Zusammengesetztes Video nach dem Export aus Adobe Premiere

3.2 Probleme bei der Verarbeitung

3.2.1 Codec-Probleme

In einem ersten Versuch habe ich probiert, die aus Camtasia kommenden Videos direkt mit dem Xvid Codec zu komprimieren, um die Dateigröße möglichst klein zu halten und da mir Stephan Salinger berichtete, dass er mit dem TSCC keine besonders guten Erfahrungen gemacht hatte und deshalb auf den DivX Codec, wie Xvid auch ein Codec aus der MPEG-4 Familie, zurückgegriffen hat. Dazu installierte ich mir das K-Lite Codec Pack von http://www.codecguide.com/, das den Xvid Codec beinhaltet. Ich exportierte nun die Videos aus Camtasia Studio und importiertie die Xvid-codierten Videos in Adobe Premiere. Dort ordnete ich die Videos entsprechend an und versuchte sie wieder mit dem Xvid Codec komprimiert zu exportieren. Das Video wurde angeblich korrekt gerendert, die Wiedergabe des Gesamtvideos schlug aber fehl, der Bildschirm blieb schwarz.

Daraufhin deinstallierte ich das K-Lite Codec Pack wieder und installierte eine andere Version des Xvid Codecs. Mit dieser Version funktionierte das Rendern des Videos aus Adobe Premiere dann, die anderen Videos, die mit der Version des Xvid Codecs aus dem K-Lite Codec Pack codiert waren ließen sich aber

nicht mehr in Adobe Premiere importieren. Also ließ ich Camtasia Studio die Videos nun noch einmal mit der neuen Version des Xvid Codecs erzeugen und versuchte diese in Adobe Premiere zu importieren, was fehl schlug. Auf der Suche nach einer Lösung für dieses Problem fand ich heraus, dass Adobe Premiere nicht für die Verarbeitung von MPEG-codierten Videos geeignet ist und ich unkomprimierte Videos verwenden soll [11]. Aufgrund der hohen Auflösung der Videos von 1280 mal 1024 Pixeln schied dies allerdings aus — die unkomprimierte Videodatei wäre einfach viel zu groß geworden.

Ich habe die Videos dann von Camtasia Studio mit dem TSCC komprimiert erzeugen lassen und konnte diese dann problemlos in Adobe Premiere importieren und bearbeiten. Anschließend renderte ich die Videos mit dem Xvid Codec. Bei der Wiedergabe blieb allerdings der Bildschirm wieder schwarz. Ein Hinweis auf einer Hilfeseite für Adobe Premiere [11] empfahl, die Videos Bild für Bild mittels des FrameServer Plugins an VirtualDub zu übergeben und dort das Komprimieren mit dem Xvid Codec vorzunehmen. Diese Methode erzeugte dann endlich die gewünschten Resultate.

3.2.2 Synchronität der Videos

Camtasia Studio hat Probleme mit der Synchronität der Videos. Im Benutzerforum des Herstellers berichten einige Benutzer von Problemen mit der Synchronität zwischen der Bildschirmaufzeichnung und dem eingeblendeten Kamerabild und dem Ton. Auch ich habe dieses Phänomen erlebt. Diese Abweichung wird immer größer, je länger die Aufzeichnung läuft. Zusätzlich zu dieser Abweichung von Kamerabild und Bildschirmaufzeichnung habe ich auch noch eine Abweichung zwischen den Einzelvideos eines Paares festgestellt. Passiert am Rechner des einen Partners nichts, weil dieser beispielsweise gerade am Rechner des anderen sitzt oder etwas liest, läuft dessen Videoaufzeichnung schneller ab als die des anderen. Dies hat sehr unschöne Effekte. Durch die räumliche Nähe zeichnen die beiden Kameras den gleichen Ton auf, wobei die Kamera, die näher an der Quelle ist, den Ton lauter empfängt. Sind die beiden Videos nicht mehr synchron, kommt es zu Echo-Effekten, die bei der Analyse sehr störend sind. Je weiter das Video fortschreitet, desto größer wird der Abstand zwischen den beiden Aufzeichnungen, teilweise habe ich Unterschiede von 30 Sekunden erlebt.

Um die Synchronität wieder herzustellen, musste ich die Videos in Adobe Premiere weiter bearbeiten. Ich habe die beiden Einzelvideos in jeweils 5 Minuten lange Segmente unterteilt und an den Segmentgrenzen die Synchronität hergestellt, indem ich die Videogeschwindigkeit der Segmente jeweils erhöhte oder senkte. Somit kommt es in einem der jeweils 5 Minuten langen Segmente nur noch zu minimalen Abweichungen, die kaum noch stören.

3.2.3 Unterschiedliche Zeitbasis

Die von Camtasio Studio erzeugten Videos arbeiteten mit 4 FPS, da mehr Bilder pro Sekunde die Größe der Videos zu stark erhöht und die Aufzeichnung mehr Rechenleistung erfordert hätte. Adobe Premiere ist darauf ausgelegt, Videos für die Wiedergabe auf Fernsehgeräten zu erzeugen und arbeitet deshalb standardmäßig mit 24 oder 25 FPS. Der niedrigste Wert, den Adobe Premiere verarbeiten kann ist 10 FPS. Beim Import der Videos werden also die mit 4 FPS codierten Videos intern auf 10 FPS hochgerechnet. Da aber nur 4 FPS im Ausgangsvideo zur Verfügung stehen, werden die fehlenden Bilder durch Zwischenzeilenbilder der ursprünglichen Bilder ersetzt. So soll ein flüssiges Bild entstehen. Bei diesen Zwischenzeilenbildern handelt es sich aber um Bilder, die nur jede zweite Zeile eines gesamten Bildes enthalten. Zwei hintereinander folgende Zwischenzeilenbilder ergeben erst ein Gesamtbild. Das führt zu deutlich sichtbaren Weglassungen im resultierenden Videos und kann nur verhindert werden, indem Adobe Premiere explizit gesagt wird, Vollbilder zu rendern.

In der Version von Atlas.TI, die ich anfangs einsetzte, gab es keine Zeitleiste mit der man von Stelle zu Stelle im Video springen konnte. Stattdessen wurde nur die Nummer des aktuellen Einzelbildes angezeigt, und man konnte von Einzelbild zu Einzelbild im Video springen. Deshalb versuchte ich die aktuelle Wiedergabezeit von Adobe Premiere im Video einblenden zu lassen. Hierzu musste das Video aber auf 25 FPS hochgerechnet werden, was wieder zu unschönen Halbbildern führte, die abgeschaltet werden mussten und außerdem die Größe des Videos enorm erhöhte. In einer neueren Version der Analysesoftware gab es dann die Möglichkeit die Darstellung der Zeitleiste von Einzelbildern auf echte Zeiteinheiten zu ändern, so dass dieser Schritt nicht mehr notwendig war.

3.2.4 Zu große Dateien

Für die Analyse der Daten habe ich die Software Atlas.TI 5.2 der Firma Scientific Research eingesetzt. Als sich meine Analysen dem Ende der Videos näherten merkte ich, dass diese nicht komplett von Atlas.TI wiedergegeben wurden. Bei einem Video fehlte nur ein sehr kleiner Teil, bei einem anderen fehlten fast 30 Minuten. Bei wieder einem anderen fehlte nichts. Ich verglich daraufhin die Größe der Dateien und stellt fest, dass die Datei, bei der nichts fehlte knapp kleiner als 2 Gigabyte war. Die Datei, bei der nur ein bißchen fehlte, war knapp größer als 2 Gigabyte. Die Datei, bei der sehr viel fehlte, war deutlich größer als 2 Gigabyte. Ich vermute, dass Atlas.TI Probleme mit Dateien größer 2 Gigabyte hat. Da der Ton der Videodateien bisher unkomprimiert eingebunden war, komprimierte ich ihn nachträglich mit VirtualDub. Durch den Einsatz eines MP3-Codecs konnte ich die Dateigröße auf 780 bis 930 Megabyte verringern und konnte anschließend auch die bisher weggelassenen Teile mit Atlas.TI analysieren.

3.3 Erste Analysen

Nachdem die Daten gewonnen und in ein für die Analyse geeignetes Format gebracht waren, begann ich mit der Analyse der Daten. Als erstes musste ich feststellen, dass es bei der Benennung der Videos eines Paares zu einem Namenskonflikt gekommen war und die Aufzeichnungssoftware eins der Teilvideos überschrieben hatte. Somit schied eins der fünf Paare für die Analyse aus.

Während der Programmiersitzung machte ich mir im Abstand von 15 Minuten Aufzeichnungen darüber, ob das Paar gemeinsam vor einem Rechner arbeitet oder jeder getrennt an seinem Computer sitzt. Bei einem Paar fiel auf, dass es die ganze Zeit gemeinsam an einem Rechner arbeitete. Nach dem Experiment fragte ich das Paar, ob mein Eindruck stimme, dass sie die gesamte Zeit Pair Programming praktiziert hätten. Dies wurde bejaht. Somit schied ein weiteres Paar aus.

Durch den Wegfall von zwei Paaren für die Analyse bestand die Befürchtung, dass das vorhandene Videomaterial nicht ausreichen würde. Allerdings war die Zeit auch sehr knapp um ein weiteres Experiment zu planen und durchzuführen. Deshalb entschied ich mich zuerst einmal alle Videos auf Episoden der Zusammenkunft und Trennung zu untersuchen und zu schauen, ob das vorhandene Material evtl. ausreicht. Dazu musste ich als erstes festlegen, was ich als Zusammenkunft betrachten würde.

In einer E-Mail an Alistair Cockburn beschrieb ich ihm mein Forschungsvorhaben und fragte ihn, was seiner Meinung nach zu Zusammenkünften bzw. Trennungen führt und was er noch für Beobachtungswert hält (siehe Anhang D). Er riet mir unter anderem auf folgende Dinge acht zu geben (Anhang E):

- Partner 1 arbeitet an seiner Aufgabe und bemerkt etwas, das Partner 2 gerade macht und kommentiert das.
- Die beiden Partner sprechen das Vorgehen zur Erledigung der Aufgabe ab, z.B. „Ich mache die Service-Klasse, schreib du doch schon einmal die HTML-Seite für die Anzeige“.

Somit habe ich für meine Analyse eine Zusammenkunft wie folgt definiert: Eine Zusammenkunft ist das gemeinsame Arbeiten an einem Rechner, zum Beispiel zur Behebung von Fehlern oder zum Testen des Zusammenspiels einzelner Komponenten, oder eine Kommunikation zwischen den Partnern, beispielsweise zum Transfer von Wissen oder zur Absprache des Vorgehens. Eine Zusammenkunft ist beendet, wenn das in der Zusammenkunft behandelte Thema abgearbeitet ist oder einer der Partner zu einer Aktion übergeht, die nicht im Zusammenhang mit der Zusammenkunft steht.

Mit dieser Definition habe ich die Videos auf Episoden von Zusammenkünften untersucht und insgesamt 91 solcher Zusammenkünfte gefunden. Aufgrund dieser hohen Zahl von Zusammenkünften konnte ich davon ausgehen genug Mate-

rial für meine Analysen zu besitzen. Somit war ein weiteres Experiment nicht notwendig.

3.3.1 Auswertung der Fragebögen

Im ersten Fragebogen befragte ich die Probanden zur gerade bearbeiten Aufgabe. Zuerst wollte ich wissen, ob die Aufgabe vollständig gelöst wurde und wie schwierig sie die Aufgabe empfunden haben. Dies sollte mir später helfen, die Fähigkeiten der Probanden richtig einschätzen zu können. Anschließend fragte ich, ob die Probanden die Möglichkeit sich zu trennen, um an zwei Rechnern zu arbeiten, genutzt haben. Falls sie das taten, wollte ich wissen aus welchen Gründen und ob sie danach wieder zusammen kamen, wie oft sie wieder zusammen kamen und ob sie bei der Zusammenkunft an einem Rechner gearbeitet haben. So gaben die Probanden an, dass Zusammenkünfte statt fanden um die Aufgabe untereinander aufzuteilen, die einzelnen Teile miteinander zu integrieren, Fehler zu beheben und Wissen zu transferieren. Zum Schluss stellte ich fünf Fragen zum persönlichen Erleben der Zusammenarbeit mit dem Partner. Der erste Fragebogen sollte mir einen ersten Eindruck vor der genaueren Durchsicht der Videos verschaffen und mich das im Video Beobachtete durch die persönlichen Eindrücke der Probanden besser verstehen lassen.

Im zweiten Fragebogen versuchte ich mir einen Eindruck über die Kenntnisse und Fähigkeiten der Probanden zu verschaffen. So fragte ich nach deren Fach- und Hochschulsemestern, ihrer Programmiererfahrung und ihrer Selbsteinschätzung zu ihren Programmierkenntnissen, um einschätzen zu können, wie groß die Kenntnisse der Probanden in etwa waren. Im zweiten Teil des Fragebogens wollte ich erfahren, ob die Probanden schon zuvor zusammen gearbeitet haben und wie sie die Zusammenarbeit während des Workshops bewerten. Der letzte Teil sollte mir ein Feedback zum Workshop geben.

Nach der Auswertung der Fragebögen konnte ich folgendes Bild von der Zusammensetzung der Paare zeichnen. Ein Paar bestand aus zwei Programmierern sehr hoher Semester. Der eine Partner war Doktorand im 12. Fachsemester, der andere Diplomand im 14. Beide hatten etwa gleich viel Erfahrung in der Java-Programmierung, 6 und 7 Jahre, und kaum Erfahrung in der Programmierung von Web-Anwendungen mit Java. Beide hatten vorher noch nie zusammen gearbeitet. Im Vergleich zu den Programmierkenntnissen ihrer Kommilitonen gaben der eine an, in etwa zum Mittelmaß zu gehören, der andere gab an, zu den besten 5 Prozent zu gehören. Somit hatte dieses Paar den größten Unterschied was den Kenntnisstand anging. Deshalb ist es auch nicht überraschend, dass der eine die Aufgabe als leicht, der andere als seinem Kenntnisstand angemessen beurteilte. Dieses Paar war auch das einzige, das die Zusammenarbeit nur als harmonisch und nicht als sehr harmonisch bezeichnete und ihr eine Schulnote von 2 bzw. 3 gab. Bei der Auswertung merkte ich, dass der eine Proband unterschwellig einen gewissen Unmut über die Fragen und die Arbeitsweise seines Kollegen durchblicken ließ und der andere eine gewisse Gehemmtheit an den Tag legte. Dieses Paar war auch das einzige, wo beide Wissenstransfer als einen Grund für Zusammenkünfte angab. Bei den anderen Paaren standen Gründe wie Integra-

tionstests, Aufgabenaufteilung und Fehlersuche im Vordergrund. Außerdem gab keiner der beiden an, für Integrationstests zusammengekommen zu sein. Insgesamt gaben die beiden an, nur selten zusammengekommen zu sein, und wenn sie es denn sind, haben die beiden ihren eigenen Angaben nach nie gemeinsam an einem Computer gearbeitet.

Dieses Paar stellte eine zusätzliche Besonderheit dar. Am Tag des Experiments erschien der Partner des einen Probanden nicht zum Experiment. Deshalb sprang nach ca. 30 Minuten Warten der 11. Teilnehmer des Workshops für ihn ein und die Zusammenkünfte beginnen erst gegen 10:06:00 (Vergleiche dazu Grafik über alle Zusammenkünfte aller Paare in Abschnitt 4).

Das zweite Paar bestand aus zwei Studenten des 6. und 8. Fachsemesters, die sich gemeinsam am Workshop angemeldet hatten und zuvor auch schon zusammen gearbeitet hatten. Beide gaben an zu den besten 40 Prozent ihres Semesters zu gehören, was die Programmierkenntnisse angeht. Die Erfahrung in der Programmierung mit Java entsprach den Fachsemestern, nämlich 3 und 4 Jahre. Allerdings konnte der Programmierer mit 3 Jahren Programmiererfahrung schon 2 Jahre Erfahrung in der Webentwicklung mit Java vorweisen, der andere nur ein Jahr. Die Schwierigkeit der Aufgabe bewerteten sie als ihren Kenntnissen angemessen. Die Zusammenarbeit empfanden die beiden als sehr harmonisch und gaben ihr Schulnoten von 1 und 2. Als Gründe für Zusammenkünfte gaben die beiden Fehlerbehebung, Integrationstests und Aufteilung der Aufgabe an. Einer der beiden führte auch noch Wissenstransfer an. Die beiden gaben an, regelmäßig zusammen gekommen zu sein und während dieser Zusammenkünfte oft bzw. immer an einem Computer gemeinsam gearbeitet zu haben.

Das dritte Paar war das einzige Paar, wo einer der Programmierer eine Frau war. Die beiden waren im 7. und 8. Fachsemester und gaben an ein bzw. 4 Jahre Programmiererfahrung in Java und ein bzw. 2 Jahre Erfahrung mit Web Entwicklung mit Java zu haben. Dabei schätzte der eine im Vergleich zu seinen Kommilitonen zum Mittelmaß zu gehören, der andere zählte sich zu den besten 40 Prozent. Der sich zum Mittelmaß zählende Programmierer empfand die Aufgabe als ziemlich schwer, der andere als seinen Kenntnissen angemessen. Auch diese beiden empfanden die Zusammenarbeit als sehr harmonisch und gaben ihr Schulnoten von 1 und 2. Wie das erste Paar hatten diese beiden vorher noch nie zusammen gearbeitet. Bei ihren regelmäßigen Zusammenkünften gaben die beiden an, oft an einem Computer gemeinsam gearbeitet zu haben um die Integration der Aufgabenteile durchzuführen.

Zusammengefasst konnte ich also drei Paare mit folgenden Eigenschaften beobachten. Paar 1 besteht aus Probanden höherer Semester, die sehr unterschiedliche Kenntnisstände aufweisen, vorher nie zusammengearbeitet haben und die Zusammenarbeit als hilfreich aber nicht so harmonisch wie die anderen beurteilen. Paar 2 ist ein Team von zwei Programmierern, die schon zuvor zusammen gearbeitet haben, in etwa den gleichen Kenntnisstand besitzen und die Zusammenarbeit als hilfreich und sehr harmonisch betrachten. Paar 3 ähnelt Paar 2 außer der Tatsache, dass beide vorher noch nie zusammen gearbeitet haben und es einen kleinen, aber nicht so großen Unterschied wie Paar 2 beim Kenntnisstand gibt.

3.4 Analyse der Videos

3.4.1 Grounded Theory

Grounded Theory (GT) ist eine von Anselm Strauss und Barney Glaser in den 1960er Jahren entwickelte Methode um Daten systematisch qualitativ zu analysieren. Am besten lässt sich Grounded Theory mit „Gegenstandsverankerte Theoriebildung“ [12] übersetzen. Bei GT handelt es sich also um eine Methode, systematisch Theorien aus gesammelten Daten herauszuarbeiten.

Zu Beginn einer Studie mit GT sollte nach Strauss und Corbin die Erarbeitung einer geeigneten Fragestellung stehen. Diese soll so formuliert sein, dass sie einerseits die Forschung auf einen bestimmten Aspekt fokussiert, andererseits aber auch offen genug ist um neue Erkenntnisse nicht auszuschließen. „Die Fragestellung in einer Untersuchung mit der Grounded Theory ist eine Festlegung, die das Phänomen bestimmt, welches untersucht werden soll.“ [12] Diese Festlegung erfolgt in Abschnitt 3.4.2.

Die zentrale Praxis bei der Arbeit mit der GT ist das Kodieren von Daten. Um überhaupt über die gesammelten Daten reden zu können, muss es einen Mechanismus geben, sie zu benennen. Beim Kodieren der Daten werden diese deshalb in kleinste Teile zerlegt, die jeweils ein bestimmtes Phänomen, beispielsweise ein bestimmtes Ereignis oder eine Aussage, darstellen. Jedes dieser Phänomene bekommt einen Kode zugewiesen und wird somit konzeptualisiert. Im Laufe des Kodierprozesses werden die Phänomene kontinuierlich miteinander verglichen, ob es vielleicht etwas Vergleichbares schon einmal gab, so dass vergleichbare Phänomene dem gleichen Konzept zugeordnet werden können. Damit ist der Forscher nun in der Lage mit einer Menge von abstrakten Konzepten zu arbeiten anstatt mit einer Flut von ungeordneten Daten.

Um mit der möglicherweise großen Menge an Konzepten umgehen zu können, werden diese zu Kategorien zusammengefasst und miteinander in Verbindung gesetzt. Anschließend werden diese Kategorien in ihrem Kontext betrachtet und in Beziehung zueinander gesetzt. Was führt zu ihrem Auftreten, was sind die dafür notwendigen Vorbedingungen, in welchem Zusammenhang treten sie auf, was sind die daraus resultierenden Konsequenzen — solche und weitere Fragen könnten dabei gestellt werden. Zum Schluss steht das Berichten über diese Kategorieren und ihre Beziehungen, Ursachen und Eigenschaften in einer Geschichte, der Theorie. Da diese komplett aus den Daten abgeleitet wurde, handelt es sich um eine „gegenstandsverankerte“ Theorie. Dieser Ablauf ist aber keineswegs starr. Vielmehr wird zwischen den Schritten hin und her gesprungen. So kann es beispielsweise schon während der Benennung von Phänomenen zu einer Kategorisierung kommen oder während des Schreibens der Geschichte werden Beziehungen zwischen den Kategorien untersucht.

3.4.2 Kodierblickwinkel

Beim Einsatz der „reinen“ Grounded Theory Methoden zur qualitativen Analyse von reichhaltigen Video- und Audiodaten sind die Autoren von [13] auf das Problem gestoßen, in einer Flut von Daten zu ertrinken und nicht zu wissen, auf welchen Aspekt des Themas sie sich beschränken sollen. Die gesammelten Video- und Audiodaten stellen eine enorme Menge an Daten dar. So ist in ihnen nicht nur die Kommunikation zwischen den Partnern zu hören, sie beinhalten auch Editiervorgänge, Mausbewegungen, Eingabe von Daten in den Computer und entsprechende Ausgaben, Ausdruck und Körpersprache der Probanden, nervöse Ticks usw., die alle kodiert und betrachtet werden könnten. Zur Vermeidung dieses Problems schlagen sie vier Praktiken vor. Eine dieser Praktiken ist die Entwicklung eines geeigneten Blickwinkels auf die zu kodierenden Daten. Bei Strauss und Corbin heißt dieser Schritt Formulierung einer geeigneten Fragestellung. Um mich nicht in unnötigen Details zu verlieren sondern auf die wichtigsten Aspekte konzentrieren zu können, mache ich mir vorher Gedanken darüber, was ich mit meiner Arbeit erreichen möchte und welche Phänomene dazu notwendigerweise in meinen Daten kodiert werden müssen.

In meiner Arbeit möchte ich herausfinden, wann sich die Paare zum Arbeiten an ihrem jeweils eigenem Rechner trennen und wann sie wieder zusammenfinden, um gemeinsam an einem Rechner zu arbeiten. Meine Aufmerksamkeit konzentriert sich daher auf die Zeitpunkte der Trennung und der Zusammenkunft. Ich werde genau beobachten, was kurz vor und während einer Zusammenkunft passiert. Dabei liegt mein Augenmerk auf den Aktivitäten, die zu einer Trennung bzw. einem Zusammenkommen führen. Dies ist eine notwendige Einschränkung zu dem in der Einleitung postulierten Anspruch meiner Arbeit, die Vorgänge beim Side-by-Side Programming zu beleuchten. Eine solch allgemein gehaltene Forschungsfrage hätte mich vor die gleichen Probleme wie in [13] beschrieben gestellt.

Die Aufgabe der Probanden läßt sich in bestimmte zu erledigende Schritte einteilen, die zur Lösung notwendigerweise getan werden müssen. Diese Schritte werde ich sofern in den Videodaten vorhanden ebenfalls kodieren. Ebenso beeinflussen wahrscheinlich auch die Kenntnisse der Probanden und deren Verhältnis zueinander, wann sie sich trennen und wann sie sich wieder zusammenfinden. Auch darauf wird ein Augenmerk bei der Kodierung der Videos liegen. Kodiert wird grundsätzlich nur das, was auch wirklich aus den Daten hervor geht. Also werde ich keinerlei subjektive Annamen über Denkprozesse oder Vorraussagen kodieren, die nicht direkt beobachtbar sind.

Am Ende meiner Arbeit sollen dann schließlich zwei Dinge stehen: Ein Kodierschema, mit dem Aktivitäten, Schritte und Eigenschaften von Personen, die zu einer Trennung bzw. zu einem Zusammenfinden führen, und ihre Zusammenhänge kodiert werden. Zum anderen soll eine Theorie über die Einflüsse entstehen, die zur Trennung bzw. Zusammenkunft der Probanden beim Side-By-Side Programming führen. Falls es mir möglich ist, werde ich auch erste Empfehlungen an Praktizierer des Sidy-by-Side Programming abgeben.

3.4.3 Kodierschema

Zu Beginn der Auswertung habe ich für die beobachteten Phänomene eigene Kodes vergeben. Das „Erfinden“ von Kodes für ein Phänomen, die Abgrenzung einzelner Kodes untereinander und das in Beziehung setzen von Kodes ist aber ein sehr aufwendiger und langwieriger Prozess. So ist es bei jedem Phänomen erforderlich zu prüfen, ob es für dieses Phänomen bereits einen passenden Kode gibt. Falls nicht müsste entweder ein neuer eingeführt werden oder ein existierender angepasst werden. Daraufhin müssen alle anderen Phänomene überprüft werden, ob der vergebene Kode noch passt. Dieses ständige Überprüfen und neu Bewerten der Konzepte ist ein sehr langwierigier Prozess. Um ein Gefühl für die dahinter steckende Arbeit zu bekommen habe ich diesen Prozess mit ca. 20 Minuten Videomaterial durchgeführt. Anschließend habe ich ein bereits vorhandenes Kodierschema von Stephan Salinger benutzt und um eigene Kodes erweitert.

Stephan Salinger untersucht bei seiner Arbeit das Pair Programming. Beim Pair Programming sitzen zwei Programmierer gemeinsam an einem Computer und entwickeln ein Softwareartefakt. Das beinhaltet das gemeinsame Schreiben von Code, die Fehlersuche, das Lesen von Dokumentation, den Wissenstransfer, usw. Die Vorgänge beim Pair Programming sind also ähnlich denen, die während einer Zusammenkunft beim Side-by-Side Programming auftreten. Da mein Augenmerk auf den Vorgängen während und kurz vor einer Zusammenkunft liegt, passt das Kodierschema gut zu meinem Problem. Es musste lediglich um wenige Kodes erweitert werden, um bestimmte Phänomene feingranularer benennen zu können.

Stark abstrahiert lässt sich das übernommene Kodierschema in zwei Teile aufteilen: Kodes für Phänomene, die eine Kommunikation zwischen den Partnern darstellen (HHI, Human-Human Interaction) und Kodes, die eine Interaktion eines Programmierers mit dem Computer darstellen (HCI, Human-Computer Interaction).

Die HHI-Kodes wiederum sind in verschiedene Kategorien unterteilt: *Hypothesis*, *Knowledge*, *Finding*, *Strategy*, *Action* und *Degree of Freedom*. Kodes der Kategorie *Hypothesis* beschreiben Phänomene, die mit Hypothesen oder Vermutungen der Programmierer zu tun haben. So markiert der Kode *propose_hypothesis* die Äußerung einer Hypothese oder Vermutung — meist eingeleitet durch Äußerungen wie „ich glaube“ oder „ich vermute mal“ — und der Kode *agree_hypothesis* markiert die Zustimmung zu einer zuvor geäußerten Vermutung. Entsprechend markiert *disagree_hypothesis* die Ablehnung einer Vermutung.

Die Kategorie *Knowledge* umfasst Kodes zur Beschreibung von Phänomenen aus dem Bereich Wissenstransfer. Der Kode *ask_knowledge* wird beispielsweise verwendet, wenn einer der Partner den anderen bittet, eine Wissenslücke zu schließen. Die Wissenserklärung an sich wird dann mit *explain_knowledge* kodiert, dem ein *ask_knowledge* vorangehen kann, aber nicht muss. Wird das übermittelte Wissen in Frage gestellt und ein eigener Vorschlag gemacht, wird der Kode

challenge_knowledge verwendet. Mit *agree_knowledge* und *disagree_knowledge* schließlich wird die Zustimmung oder Ablehnung von übermitteltem Wissen kodiert.

Ein Phänomen, das mit einem Kode aus der Kategorie *Finding* versehen wurde, bezieht sich auf eine Erkenntnis, die ein Programmierer äußert, z.B. nach dem Lesen von Dokumentation. So wird die Äußerung dieser Erkenntnis mit *think aloud_finding* kodiert, die Zustimmung zu einer solchen Erkenntnis mit *agree_finding*.

Kodes aus den Kategorien *Strategy*, *Action* und *Degree of Freedom* beschreiben Phänomene, die von der Kommunikation über die zu erledigende Arbeit handeln. So handeln Phänomene, die mit Kodes aus der Kategorie *Strategy* versehen wurden, von Abstimmungen über die zu verfolgende Strategie, also die Planung der nächsten Arbeitsschritte. *ask_strategy* kodiert die Frage zur Strategie. Einem *propose_strategy*, das kodiert wird, wenn ein konkreter Vorschlag zum Vorgehen gemacht wird, kann ein *ask_strategy* vorangehen, muss es aber nicht. Stehen mehrere Vorschläge zur Auswahl, wird die Festlegung auf einen mit *decide_strategy* kodiert. Das Zustimmen zu einem Vorschlag aus einem *propose_strategy* wird mit *agree_strategy* kodiert. Wird die vorgeschlagene Strategie abgelehnt und ein Alternativvorschlag gemacht, wird *challenge_strategy* kodiert.

Die Planung des konkreten, nächsten Arbeitsschrittes wird mit Kodes aus der Kategorie *Action* versehen. Mit Kodes aus der Kategorie *Degree of Freedom* werden Phänomene beschrieben, die eine Abstimmung über Designentscheidungen darstellen, z.B. die Abstimmung über die Benennung einer bestimmten Methode oder den Rückgabetyp einer Funktion. Zusätzlich zu Kodes mit den gleichen Verben wie bei *Strategy*, also *ask_*, *propose_*, *agree_*, *decide_* und *challenge_* gibt es bei diesen beiden Kategorien noch weitere Verben. *explain_* kodiert eine Erklärung eines *Action* oder *Degree of Freedom* Phänomens, *disagree_* seine Ablehnung und *amend_* seine Erweiterung um einen weiteren Aspekt.

Die HCI-Kodes beschreiben Phänomene, die eine Interaktion der Probanden mit dem Computer darstellen. So gibt es Kodes für das Schreiben von Code (*write_sth*), für das Suchen von Objekten (*search_sth*), das Erkunden ohne bekanntes Zielobjekt (*explore_sth*), für die Arbeit mit der Versionsverwaltung (*commit_changes*, *update_codebase*,*revert_changes*), für das Überprüfen von Artefakten(*verify_sth*), für das laute Lesen von Programmausgabenoder Dokumentation (*read_sth*), für das Lesen der Aufgabenstellung(*read_requirements*) und für das explizite Hinweisen auf bestimmteStellen auf dem Bildschirm, beispielsweise ein Stück Code oder Dokumentation(*show_sth*). Alle anderen, nicht explizit kodierten HCI-Aktivitätenwerden mit dem Kode *do_sth* zusammengefasst. Darunter fällt beispielsweise das Starten von Eclipse oder der Applikation oder das Aufräumen des Desktops.

Zwischen den HCI-Kodes und den HHI-Kodes gibt es noch Kodes, die eine Kommunikation zwischen den Programmierern darstellen, sich aber auf eine Aktivität am Computer beziehen. Diese fallen in die Kategorie *Activity*. Hierzu gehören beispielsweise Äußerungen eines Partners auf die gerade erfolgende HCI-Aktivität des anderen (*disagree_activity* bei Ablehnung, *agree_activity*

bei Zustimmung). Eine wertfreie Äußerung über die gerade erfolgende HCI-Aktivität wird mit *think aloud_activity* kodiert. Außerdem gibt es noch ein paar Kodes, die für sich alleine stehen und keiner Kategorie zugeordnet sind, wie z.B. *ask_comprehension* für das Fragen einer Verständnisfrage oder *remember_requirements* für das Erinnern an die Aufgabenstellung.

Ich habe dieses Kodierschema um Kodes für das Beschreiben der Zusammenkünfte an sich und bestimmte HCI-Kodes erweitert, wie z.B. explizite Kodes für die Arbeit mit der Versionsverwaltung.

3.4.4 Die Analysesoftware

Für die Analyse der Videos setzte ich die Software Atlas.TI der Firma Scientific Research in der Version 5.2.18 ein. Atlas.TI wurde speziell dafür entwickelt, den Forscher bei der qualitativen Analyse von Daten zu unterstützen. Ausgelegt ist Atlas.TI auf die Bearbeitung von Text-Dokumenten, es ist aber auch eines der wenigen Werkzeuge mit Unterstützung für die Arbeit mit Multimedia-Dokumenten wie Audio-Dateien und Videos.

Mit Atlas.TI kann ich die Videos in kleine Segmente, sogenannte *Quotations*, unterteilen und diese so isolierten Konzepte mit Kodes versehen. Atlas.TI bietet weiterhin verschiedene Funktionen zum Organisieren und Verwalten von Kodes, *Quotations* und Memos. Auch das Bearbeiten mehrerer Videos innerhalb der gleichen *Hermeneutic Unit* (HU), Atlas.TIs Bezeichnung für einen Container, der Kodes, *Quotations*, Memos und Verweise auf zu untersuchende Dateien enthält, ist möglich. Außerdem gibt es vielfältige Möglichkeiten, Kodes, *Quotations* und andere Objekte miteinander in Beziehung zu setzen. Diese sind aber sehr umständlich zu bedienen, so dass man leicht der Versuchung erliegt, diese Beziehungen textuell innerhalb von Memos oder Kommentaren zu Kodes und *Quotations* zu erfassen – so ist es auch mir ergangen.

Man merkt Atlas.TI deutlich an, dass es ursprünglich für die Bearbeitung von Textdokumenten konzipiert wurde. Die Unterstützung für Video-Daten ist zwar vorhanden, aber umständlich. So konnte bis zu einem Update Atlas.TI die Zeitleiste nicht mit Zeitangaben versehen sondern arbeitete auf der Ebene von Einzelbildern. Das Navigieren im Video wurde somit ziemlich umständlich. Auch die Angabe der Position einer *Quotation* innerhalb eines Videos erfolgt nur auf Basis von Einzelbildern. Hier kann auch mit keinem Update Abhilfe geschaffen werden. Als letzten Punkt möchte ich die Trägheit von Atlas.TI beim Navigieren innerhalb eines Videos erwähnen. Es dauert manchmal mehrere Sekunden bis Atlas.TI innerhalb eines Videos 10 Einzelbilder vor oder zurück navigiert. Das direkte Springen an bestimmte Positionen im Video ist überhaupt nicht möglich.

3.4.5 Das Vorgehen

Wie zuvor erwähnt habe ich die Videos zuerst auf Zusammenkünfte untersucht und diese entsprechend kodiert. Anschließend habe ich mir die Phänomene innerhalb der Zusammenkünfte sehr genau angeschaut und sehr feingranular mit Kodes versehen. Dabei habe ich mit Atlas.TI sogenannte Quotations erstellt. Diese Quotations sind nichts anderes als ein Ausschnitt auf der Zeitleiste des Videos, stellen also einen kleinen Teil (manchmal nur eine oder zwei Sekunden) des Videos dar, in dem ein Phänomen auftritt. Da ich immer zwei Probanden gleichzeitig aufgezeichnet habe, können in einem solchen Ausschnitt zwei Phänomene auftreten, die kodiert werden müssen. Beispielsweise kann es sein, dass innerhalb einer extrem kurzen Zeitspanne auf die Frage eines Probanden die Antwort des anderen kommt, oder einer der beiden schlägt etwas vor und lässt sich dabei ein wenig aus, während der andere konstant zustimmt, beispielsweise durch ein Nicken oder ein „Hm.“ oder ein „Ja genau.“. In solchen Fällen wurde der Quotation dann zwei Kodes zugewiesen, einer für das Phänomen des einen, der ander für das Phänomen des anderen Partners.

Da ich zusätzlich wissen wollte, was denn nun genau zu einer Zusammenkunft führt, habe ich mir auch einen Zeitraum kurz vor der Zusammenkunft angeschaut. Willkürlich habe ich einen Zeitraum von 30 Sekunden vor einer Zusammenkunft festgelegt, den ich untersucht habe. In Fällen, wo dies nicht ausreichte, habe ich mir einen noch längeren Zeitraum vor der Zusammenkunft angeschaut.

Nach dieser sehr feingranularen Analyse habe ich mir die Kategorien, in welche die Kodes fallen, angeschaut und basierend darauf die Zusammenkünfte in verschiedene Arten von Zusammenkünften eingeteilt. An Stellen, wo durch die Verwendung der Kodes nicht klar war, welche Art von Zusammenkunft vorliegt, habe ich eine Handlungszusammenfassung der Zusammenkunft herangezogen. Während der Kodierung der Videos habe ich für jede Zusammenkunft eine Zusammenfassung des Inhalts angefertigt, die Auskunft darüber gibt, was während dieser Zusammenkunft passiert. Mit Hilfe dieser Zusammenfassung, der verwendeten Kodes und den zuvor ermittelten Zusammensetzungen der Paare und ihrer Eigenschaften, konnte ich die Zusammenkünfte dann analysieren.

4 Ergebnis

4.1 Arten der Zusammenkunft

Grundsätzlich habe ich zwei Typen der Zusammenkunft beobachtet. Es gibt physische und nicht-physische Zusammenkünfte. Bei den physischen Zusammenkünften bewegt sich einer der beiden Partner zum Rechner des anderen und arbeitet dort mit ihm zusammen weiter, lässt sich etwas zeigen, schaut dem anderen zu oder kommuniziert mit ihm. Bei den nicht-physischen Zusammenkünften fehlt das Weggehen vom eigenen Rechner.

Zusätzlich zu dieser Unterscheidung auf physischer Ebene kann man Zusammenkünfte nach ihrem Inhalt kategorisieren. Bei der Analyse der Daten konnte ich sechs Arten der Zusammenkunft identifizieren. Dabei kann es passieren, dass der gleichen Zusammenkunft mehrere Arten der Zusammenkunft zugeordnet wurden. Das war der Fall, wenn es inhaltlich während der Zusammenkunft eine Zäsur gab. Ging es bei einer Zusammenkunft beispielsweise um die Mitteilung der Fertigstellung einer Komponente (siehe Abschnitt 4.1.1) und folgte danach die Planung des weiteren Vorgehens (vergleiche Abschnitte 4.1.3 und 4.1.4), bekam diese Zusammenkunft zwei Kategorien zugeordnet.

In der Abbildung über alle Zusammenkünfte der drei Paare (siehe Abbildung 21 auf Seite 50) sind die Lage und Dauer der Zusammenkünfte dargestellt. Zusammenkünfte, denen mehrere Kategorien zugeordnet waren sind dort mit der dominierenden Kategorie verzeichnet, also mit der Kategorie um die es hauptsächlich während dieser Zusammenkunft ging.

4.1.1 Projektwissen

Zusammenkünfte vom Typ Projektwissen sind solche, die vom Austausch von Wissen über das Projekt handeln. Diese sind wichtig für die Koordination der Arbeit. So ging es bei solchen Zusammenkünften beispielsweise um den Austausch von Informationen über die Namensgebung von in geteilter Arbeit entwickelter Komponenten. Sätze wie „wie hast du diese Seite benannt“ oder „wie heißt die Funktion zum Speichern der Entity“ fielen in diesem Zusammenhang, weil der andere diese Informationen brauchte, um mit seiner Aufgabe weiter zu kommen. Auch der Austausch von Informationen über den Speicherort von Komponenten („wo hast du das hingetan“), den Bearbeitungsstand („ich bin jetzt mit X fertig“) oder allgemeiner Informationen wie „wieso hast du das so und so gemacht“ oder „hast du X schon eingecheckt“ fand in Zusammenkünften des Typs Projektwissen statt. Erkennen konnte ich Zusammenkünfte, bei denen es um Wissenstransfer ging, durch die Vergabe von Kodes der Kategorien *Knowledge* und *Hypothesis* (Vergleiche Abschnitt 3.4.3). Hatte ich für Phänomene innerhalb einer Zusammenkunft oft Kodes dieser beiden Kategorien benutzt, musste es sich um eine Zusammenkunft zwecks Wissenstranfers handeln. Durch die Art des transferierten Wissens konnte ich die Zusammenkunft dann der Ka-

tegorie Projektwissen zuordnen.

Zusammenkünfte vom Typ Projektwissen sind die am häufigsten auftretenden Zusammenkünfte und überwiegend (27 von 43) nicht-physisch, d.h. die Partner bewegen sich nicht von ihren Rechnern weg sondern bleiben an ihrem Platz. Bei trivialen Fragen kann es sogar passieren, dass der Gefragte gar nicht erst seine Arbeit unterbricht um dem Partner zu helfen, sondern mit seiner Tätigkeit einfach weitermacht, während er die Frage des Partners beantwortet. Bis auf bei Paar 1, wo sich einer der beiden erst einmal einarbeiten musste, traten diese Zusammenkünfte überall während der Programmiersitzung auf. Grund für solche Zusammenkünfte ist hauptsächlich der Austausch über den Stand der Bearbeitung. So teilt beispielsweise der eine Partner dem anderen mit, dass er mit dem Schreiben der Service-Klasse fertig ist, oder einer der beiden fragt, ob die HTML-Seite schon fertig ist. Des Weiteren werden Zusammenkünfte vom Typ Projektwissen dazu genutzt, Detailfragen zur Implementierung oder zur Namensgebung zu klären. Es wird dann beispielsweise geklärt, in welcher Klasse eine bestimmte Funktion geschrieben wurde, oder wie diese heißt. Dies geschieht, wenn einer der beiden eine Funktionalität, die der andere entwickelt hat, benötigt, ihm aber nicht klar ist, wo er diese Funktionalität findet oder wie sie genau heißt. Zuletzt dienen sie auch dazu, den anderen darüber zu informieren, was man gerade macht. Dies stellt dann keine Planung dar sondern hat rein informativen Charakter. Der Grund hierfür ist nicht klar geworden und scheint mit in der Persönlichkeit und Arbeitsweise der Person zu liegen. So bekam ein Proband häufig solche Statusmeldungen von seinem Partner, bei einem anderen Team habe ich solche Meldungen wiederum überhaupt nicht beobachten können.

Überhaupt beeinflussen die Eigenschaften und Fähigkeiten der Probanden die Häufigkeit und Art der Zusammenkünfte vom Typ Projektwissen. Das Paar, das schon einmal zuvor zusammen gearbeitet hat und wo beide in etwa den gleichen Kenntnisstand besaßen (Paar 2) traf sich zehn mal um Projektwissen auszutauschen, davon nur einmal gemeinsam vor einem Rechner. Dagegen trafen sich das Paar mit nur geringen Unterschieden beim Kenntnisstand, die aber noch nie zuvor zusammen gearbeitet hatten (Paar 3), 14 mal, davon acht mal vor einem Rechner. Der eine Partner bei diesem Paar beherrschte Deutsch nicht als Muttersprache. Es könnte daran gelegen haben, dass sich der andere immer am Rechner des Nichtmuttersprachlers hat zeigen lassen, was genau er meint. Das Paar mit dem großen Unterschied beim Kenntnisstand (Paar 1) traf sich mit 19 mal am häufigsten um Wissen über das Projekt auszutauschen. Davon fanden 7 Treffen gemeinsam am Rechner eines Partners statt, um sich genau zeigen zu lassen, was gemeint ist, genau wie bei Paar 3. Die Tatsache, ob die Partner vorher schon einmal zusammengearbeitet haben, scheint also die Häufigkeit der Zusammenkünfte, um Projektwissen auszutauschen und ob diese Zusammenkünfte gemeinsam an einem Rechner stattfinden, zu beeinflussen.

Bei Paar 1 konnte ich beobachten, dass der Partner mit dem geringeren Kenntnisstand oft Zusammenkünfte vom Typ Projektwissen genutzt hat, um sich von dem anderen bei der Fehlersuche helfen zu lassen. Vorher hatte er sich schon länger mit dem Problem beschäftigt. Anstatt aber direkt um Hilfe zu bitten wenn er nicht weiter kam, hat er die bestehende Zusammenkunft genutzt. Er

schien gehemmt zu sein, den Partner mit dem sehr viel höheren Kenntnisstand mit seinem Problem zu behelligen. Leider ist dieses Gehemmtsein nicht direkt aus der Kodierung ablesbar. Lediglich die Tatsache, dass fünf Fehlersuchen aus anderen Zusammenkünften hervorgingen und nur drei für sich alleine standen ist ein Indiz dafür. Ansonsten beruht diese Aussage einzig auf meinen Beobachtungen seines Verhaltens.

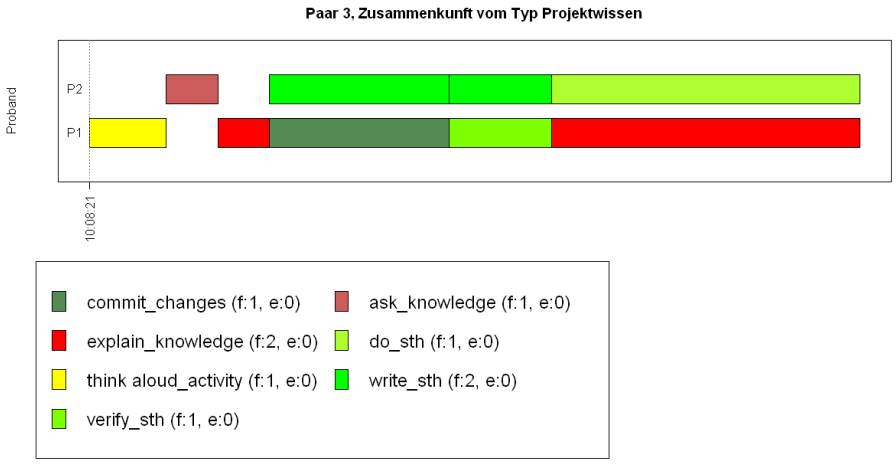


Abbildung 11: Zusammenkunft vom Typ Projektwissen bei Paar 3

Abbildung 11 zeigt die Kodierung während einer Zusammenkunft vom Typ Projektwissen, die 30 Sekunden dauert. Die Handlungszusammenfassung für diese Episode lautet wie folgt: „Mit einem langgezogenen „OK“ leitet P1 diese Zusammenkunft ein, P2 fragt daraufhin, ob er schon fertig sei. P1 informiert P2, dass er noch kurz bräuchte. Nachdem er fertig ist informiert er P2, dass er mit dem Service jetzt fertig ist und P2 ihn benutzen kann. Allerdings sei aber noch irgend etwas kaputt, das er jetzt herausfinden wird.“

Bei der Analyse sah ich, dass diese Zusammenkunft von Kodes der Kategorie *Knowledge* dominiert wird (rot dargestellt). Es handelte sich also wahrscheinlich um einen Wissenstransfer. Die Handlungszusammenfassung bestätigte das und lieferte auch die Art des Wissenstransfers, nämlich Projektwissen. Auf das langgezogene „OK“ von P1, das eine Verbalisierung der gerade durchgeführten Tätigkeit darstellt (*think aloud_activity*, gelb dargestellt), folgt die Frage von P2, ob P1 fertig sei (*ask_knowledge*). P1 erklärt, dass er noch kurz bräuchte (*explain_knowledge*). Daraufhin commitet er den Code (*commit_changes*) und überprüft, ob alles soweit stimmt (*verify_sth*), während P2 weiter an seinem Code schreibt (*write_sth*). Diese HCI-Aktivitäten sind in der Grafik grün gehalten. Abschließend informiert P1, dass er nun fertig sei, sich darin aber noch ein Fehler versteckt, den er noch finden muss (*explain_knowledge*). Währenddessen führt P2 eine nicht weiter spezifizierte HCI-Aktivität aus (in diesem Fall handelt es sich um das Schließen von Editor-Fenstern in Eclipse).

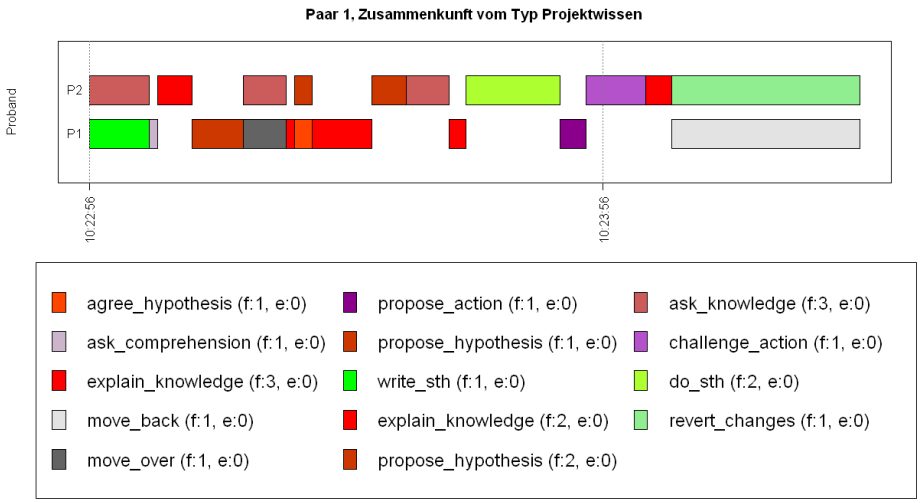


Abbildung 12: Zusammenkunft vom Typ Projektwissen bei Paar 1

Bei der 00:01:30 langen physischen Zusammenkunft, die in Abbildung 12 dargestellt ist, war die Sache nicht ganz so eindeutig. Die Handlungszusammenfassung lautete „P2 hat die Datenbank gelöscht und nun Probleme sich anzumelden und fragt P1 wie er einen neuen Benutzer anlegen kann bzw. ob es einen Default-User gibt. P1 geht zu P2 um zu schauen, was er genau meint und macht Vorschläge. Sie versuchen einen neuen Benutzer anzulegen, was nicht klappt. Anschließend Reverted P2 die Datenbank um den alten Inhalt wieder zu bekommen.“. Aus dieser Handlungszusammenfassung könnte man schließen, dass es sich um eine Fehlersuche handelt. Auch die lange HCI-Aktivität *revert_changes* passt nicht sofort ersichtlich zu einem Wissenstransfer. Diese ist aber nur deshalb so lang, weil ich den gesamten Vorgang, vom Markieren der Dateien über den Zugriff auf das richtige Kontextmenü bis zum Abschluss des Vorgangs, kodiert habe. Inhaltlich macht dieser Vorgang nur einen sehr kleinen Teil dieser Zusammenkunft aus. Auch ist die rege Kommunikation zu Beginn der Zusammenkunft ein Indiz dafür, dass es eher um Wissensaustausch ging, erkennbar anhand der vielen kleineren Blöcke. Da hier also überwiegend Wissen und Hypothesen über das Projekt ausgetauscht wurden (in der Grafik in Rottönen gehalten), ordnete ich diese Zusammenkunft der Kategorie Projektwissen zu.

Kam es während einer Zusammenkunft vom Typ Projektwissen zu einer Mitteilung über den Stand der Bearbeitung, also teilte beispielsweise einer dem anderen mit, dass er mit seiner zuvor aufgeteilten Arbeit fertig ist, folgte oft entweder eine Zusammenkunft vom Typ Integration, um die fertig gestellten Komponenten zu testen, oder eine Zusammenkunft vom Typ Strategie oder Planung, um das weitere Vorgehen zu besprechen. Manchmal gingen solche Zusammenkünfte direkt ineinander über, d.h. eine Zusammenkunft war anfangs vom Typ Projektwissen und ging dann über in Integration, Strategie oder Planung.

Die Durschnittsdauer der 34 „reinen“ Zusammenkünfte vom Typ Projektwissen, also derer, die nicht zusätzlich noch einer anderen Art zugeordnet sind, beträgt 41 Sekunden. Dabei ist ein Ausreißer mit einer Dauer von 00:03:45. Ohne diesen Ausreißer haben die dann verbleibenden 33 Zusammenkünfte eine Durchschnittsdauer von 36 Sekunden.

4.1.2 Wissen

Die andere Art des Wissenstranfers, die ich unterscheide, ist der Transfer von „echtem“ Wissen. Zusammenkünfte, bei denen es darum ging, wurden der Kategorie Wissen zugeordnet. „Echtes“ Wissen bezeichnet dabei Informationen, von denen die Probanden annehmen, dass sie wahr sind und die kein Wissen über das Projekt wie beim Projektwissen darstellen. Auch wenn die Zusammenkunft dazu dient „echtes“ Wissen zu bilden, also sich beide nicht über den Wahrheitsgehalt der ausgetauschten Informationen sicher sind — in diesem Fall kamen dann Kodes der Kategorie *Hypothesis* zum Einsatz und es fielen Sätze wie „ich glaube das geht so und so“ — ordnete ich die Zusammenkunft der Kategorie Wissen zu. Bei den beobachteten Zusammenkünften vom Typ Wissen ging es um den Austausch von Wissen über bestimmte Technologien wie beispielsweise den korrekten Einsatz der Versionsverwaltung oder von JPA Annotationen. Kam es also beispielsweise zu einer Zusammenkunft, weil einer der beiden Partner eine Wissenslücke entdeckt hat und den anderen um Schließung derselben bat, wurde diese Zusammenkunft der Kategorie Wissen zugeordnet. Allerdings muss der Wissenstransfer nicht immer explizit verlangt sein. So kann es auch passieren, dass einer der beiden aus anderen Gründen, z.B. aus dem in seinen Augen ineffizienten Einsatz einer Technologie durch den anderen, diesem sein Wissen mitteilt.

Zusammenkünfte vom Typ Wissen sind die am seltensten auftretenden Zusammenkünfte und waren allesamt physischer Natur, d.h. die Partner trafen sich vor einem Rechner. Von den sieben Zusammenkünften dieser Art entfielen sechs auf nur ein Paar. Dies war Paar 1, das Paar mit dem größten Unterschied beim Kenntnisstand. Dabei floss das Wissen ausschließlich vom Partner mit dem höheren Kenntnisstand zu dem mit dem geringeren. Ein Grund dafür liegt am Typ des einen Partners. Der gleiche Proband hat seinen Partner auch regelmäßig darüber unterrichtet, was er gerade macht. Ausschlaggebend ist hier aber der Unterschied beim Kenntnisstand der beiden. In den Fragebögen gab der Partner mit dem geringeren Kenntnisstand an, öfter Wissen von seinem Partner benötigt zu haben. So erklärte der kenntnisreichere der beiden dem anderen öfter bestimmte Funktionen von SVN oder JPA Annotationen. Am krassesten trat dieser Unterschied beim Kenntnisstand in einer Situation zu Tage, als der eine Partner eine Möglichkeit für SQL Injection im Code des anderen fand und ihm das erklären musste. Trotz der angegebenen sieben Jahre Erfahrung in der Programmierung mit Java und einem Jahr Erfahrung in der Web-Programmierung mit Java, wusste er nicht genau was das ist und welche Implikationen eine solche Schwachstelle mit sich bringt. In einem späteren Gespräch, das ich mit dem Partner mit dem größeren Kenntnisstand hatte, stellte sich dann auch heraus, dass er zeitweise sehr entnervt war, da er nicht auf einem gleichen Niveau mit

seinem Partner arbeiten konnte.

Die verbleibende Zusammenkunft entfiel auf Paar 2, das Paar mit den beiden ausgeglichenen Programmierern, die zuvor schon einmal zusammen gearbeitet haben. Hier ging es einfach um eine Funktion von Tapestry, an deren richtige Anwendung der eine der beiden sich nicht mehr erinnern konnte (siehe Abbildung 13).

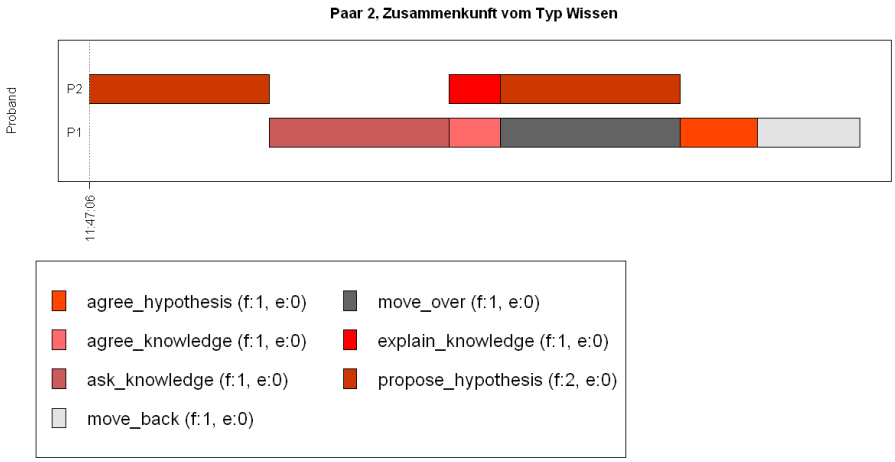


Abbildung 13: Zusammenkunft vom Typ Wissen bei Paar 2

Abbildung 13 zeigt beispielhaft die Kodierung einer halbminütigen Zusammenkunft vom Typ Wissen von Paar 2. Es ist klar ersichtlich, dass es bei dieser Zusammenkunft um den Austausch von Wissen und Hypothesen geht (in Rottönen dargestellt). Ohne die Handlungszusammenfassung aber ist nicht erkennbar, ob es sich um Projektwissen oder „echtes“ Wissen handelt. Die Handlungszusammenfassung für diese Zusammenkunft lautet: „P2 ist sich nicht sicher, wie mehrere Listener-Parameter in Tapestry gehen und teilt P1 seine Vermutung mit. P1 fragt daraufhin P2 nach Funktionsweise von Hibernate (merge/reattach), P2 antwortet ihm. Dann bewegt sich P2 zu P1 und beide gleichen ihre Hypothesen zu mehreren Listener-Parametern bei Tapestry ab.“. Da sich beide bei ihren Aussagen zu den Listener-Parametern nicht sicher sind, wurden diese Passagen mit Kodes der Kategorie *Hypothesis* kodiert. Bei dieser Zusammenkunft war nicht ganz klar, ob sie nun physisch oder nicht-physisch ist, da es sowohl eine nicht-physische Komponente gibt, nämlich die Gegenfrage von P1 zu einer Funktion von Hibernate, als auch eine physische (der Abgleich der Hypothesen zu Tapestry). Da sich der Großteil der Zusammenkunft dem Abgleich der Hypothesen zu Tapestry widmet und davon wiederum der Großteil zusammen verbracht wird, behandelte ich diese Zusammenkunft als physische Zusammenkunft.

Die einminütige Zusammenkunft vom Typ Wissen von Paar 1, die in Abbildung 14 visualisiert ist, war nach der Handlungszusammenfassung schwieriger einzuordnen. In dieser hatte ich zusammengefasst: „P1 hat einen Fehler begründet

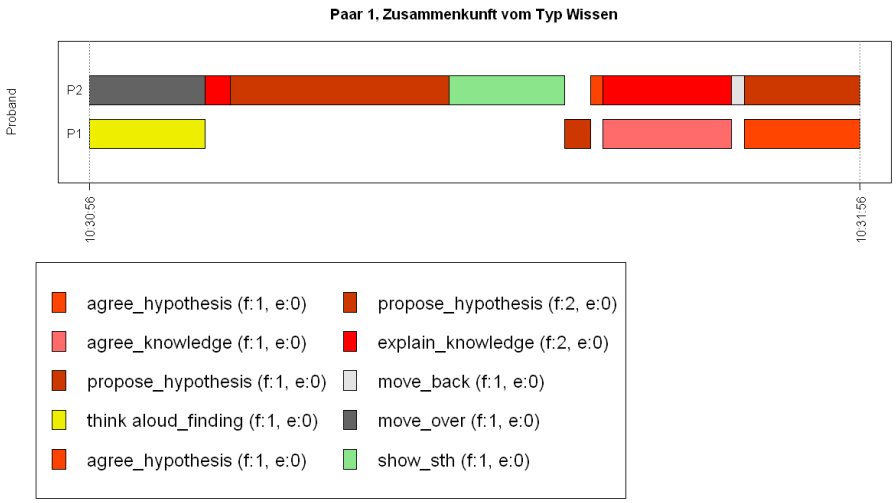


Abbildung 14: Zusammenkunft vom Typ Wissen bei Paar 1

in einem Problem mit den Annotationen bei den Entities und P2 erklärt ihm, wie er es richtig machen muss.“. Dies für sich alleine genommen könnte eine Zusammenkunft vom Typ Fehlersuche bedeuten. P1 hat gerade die Erkenntnis gewonnen, dass etwas bei ihm nicht stimmt und äußert das in einem *think aloud_finding*. Anstatt aber jetzt den Fehler zu analysieren, präsentiert P2 gleich eine Erklärung und lässt sich noch weiter über Details dazu aus, ersichtlich durch die Verwendung von Kodes der Kategorien *Hypothesis* und *Knowledge* (in Rottönen dargestellt). Daher stellt diese Zusammenkunft eher einen Wissenstransfer dar, auch wenn damit ein Fehler behoben werden konnte.

4.1.3 Strategie

Die Kategorie Strategie bezeichnet Zusammenkünfte, bei denen es um die grobe Aufteilung von Aufgaben oder die Planung des allgemeinen Vorgehens geht. Während Zusammenkünften dieser Art fallen beispielsweise Sätze wie „Schreibe du die HTML-Seiten, ich mache die Datenschicht“ oder „Ich mache Standorte anlegen, Du verändern“. Charakteristisch für eine Zusammenkunft vom Typ Strategie ist, dass das besprochene Vorgehen mehrere Einzelschritte umfasst und noch nicht ganz klar ist, was genau der konkrete, nächste Schritt ist, geschweige denn, wie der zu schreibende Code aussehen soll. Hinweis für eine solche Zusammenkunft ist die Verwendung von Kodes aus der Kategorie *Strategy* zur Beschreibung der beobachteten Phänomene.

Zusammenkünfte vom Typ Strategie konnte ich bei allen Paaren am Anfang der Programmiersitzung beobachten. Hier wurde meist die grobe Aufteilung besprochen, also welcher Partner welchen Teil erst einmal macht. Dabei verließen die Partner ihre Rechner nicht, die Zusammenkünfte waren also nicht-physisch.

Im Verlauf der Sitzung traten solche Zusammenkünfte entweder nach einer Mitteilung über den Bearbeitungsstand (Typ Projektwissen) oder nach einer Integrationssitzung auf. Sie fielen also mit der Fertigstellung eines Arbeitsschrittes zusammen. In sieben von neun Fällen stand eine Zusammenkunft vom Typ Strategie auch nicht für sich alleine, sondern trat zusammen mit einer anderen auf, beispielsweise vom Typ Projektwissen (siehe Abbildung 16) oder Integration. Dann wurde diese Zusammenkunft gleich dazu genutzt, das weitere Vorgehen zu besprechen. Fiel die Zusammenkunft mit einer vom Typ Integration zusammen, so war sie meistens physisch. Daher ist es auch nicht möglich, Aussagen über die Dauer solcher Zusammenkünfte zu machen.

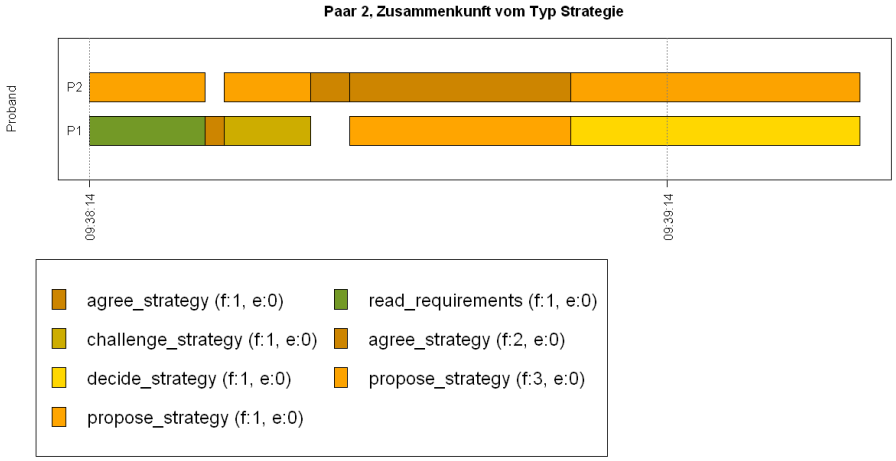


Abbildung 15: Zusammenkunft vom Typ Strategie bei Paar 2

Ein Beispiel für eine nicht-physische Zusammenkunft vom Typ Strategie liefert Abbildung 15. Bei dieser Zusammenkunft musste ich noch nicht einmal die Handlungszusammenfassung lesen („P2 macht Vorschläge zum Vorgehen, während P1 noch die Anforderungen liest. P1 macht dann weitere Vorschläge.“), da direkt aus den verwendeten Kodierungen hervorging, dass es sich um eine Strategiezusammenkunft handeln musste. Die Verwendung von Kodes der Kategorie *Strategy* ist in diesem Bild in Orangetönen gehalten.

Abbildung 16 zeigt eine Zusammenkunft, die mit einem kurzen Teil Projektwissen beginnt (P2 teilt P1 mit, dass seine Komponente jetzt funktioniert und er sie eincheckt) und dann in eine Strategiezusammenkunft mündet. Die Handlungszusammenfassung für diese Zusammenkunft lautet „P2 informiert P1, dass sein Teil erst mal jetzt läuft und er das ganze committed. Anschließend teilen sie die Aufgaben untereinander auf.“. Es gibt also erst einmal einen kleinen Teil Projektwissen, gefolgt von einer Art Unterbrechung der Zusammenkunft, bei der P2 seinen Code eincheckt. Anschließend treffen sich beide bei P1 und besprechen das Vorgehen. Obwohl in dieser Zusammenkunft ein Teil HCI-Aktivität vorkommt, habe ich diese Zusammenkunft als Strategiezusammenkunft deklariert, da der eigentliche Grund für diese Zusammenkunft und auch ein großer

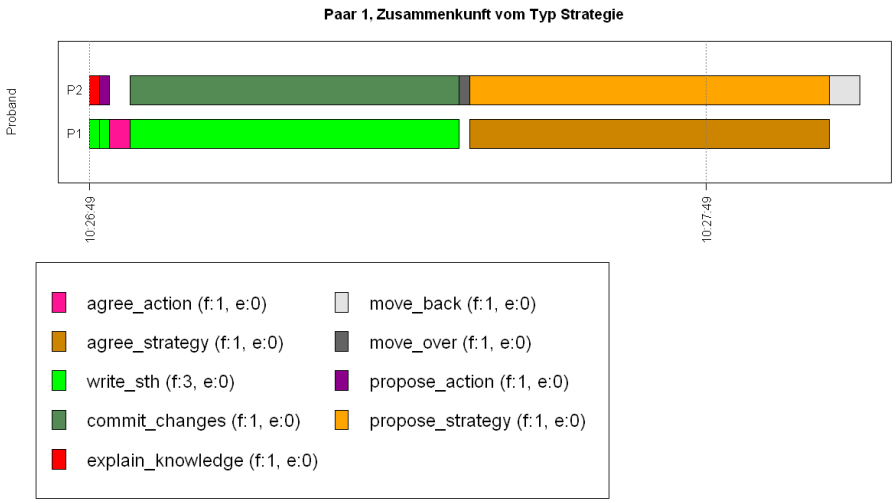


Abbildung 16: Zusammenkunft vom Typ Strategie bei Paar 1

Teil der Zeit auf die Abstimmung des Vorgehens entfällt. Ich hätte diese Zusammenkunft auch in eine sehr kurze vom Typ Projektwissen und eine vom Typ Strategie teilen können. Von der Handlung her gehören diese beiden aber zusammen.

Bei Paar 3 fiel auf, dass dieses Paar nur eine einzige Strategiezusammenkunft zu Beginn der Sitzung hatte. Im Verlauf der Arbeit gab es keine weitere grobe Aufteilung der Arbeit mehr. Das heißt nicht, dass es keine weitere Absprachen mehr gab. Zusammenkünfte vom Typ Planung (siehe weiter unten) kamen auch bei diesem Paar noch vor. Es gab aber keine Absprachen mehr zum generellen Vorgehen, nur noch solche zum nächsten konkreten Schritt.

4.1.4 Planung

Bei Zusammenkünften der Kategorie Planung geht es dann um den konkreten nächsten Arbeitsschritt oder die Ausprägung eines Freiheitsgrades. Ein Freiheitsgrad ist beispielsweise die Benennung einer Methode oder die Festlegung auf einen bestimmten Rückgabewert und stellt in der Planung den feinsten Teil dar. Während Zusammenkünften vom Typ Planung hörte ich oft Dinge wie „wir brauchen noch eine Methode zum Editieren von X“ oder „Wie nennen wir das jetzt?“ oder „du musst mal ein Update machen“ oder „was soll ich hier zurückgeben?“. Während solcher Zusammenkünfte habe ich für die Phänomene Kodes der Kategorien *Action* und *Degree of Freedom* verwendet. Traten diese gehäuft auf, musste es sich um eine Zusammenkunft vom Typ Planung handeln.

Zusammenkünfte vom Typ Planung treten an den verschiedensten Stellen auf. Auf eine Zusammenkunft vom Typ Integration kann eine vom Typ Planung fol-

gen, um das weitere Vorgehen abzustimmen. Auch auf die Mitteilung, dass man nun mit seiner Komponente fertig ist kann eine Planung folgen wenn nicht vorher schon klar war, was der nächste konkret zu erledigende Schritt ist. Mit Planung sind auch Abstimmungen über die Namensgebung von Methoden, Klassen, Paketen oder HTML-Seiten gemeint. So kann es mitten während der Programmierung zu einer kurzen Zusammenkunft vom Typ Planung kommen, wenn einer der beiden den Namen einer Methode, Klasse oder HTML-Seite festlegen will, weil er gerade an dieser Stelle angelangt ist. Dabei rangiert die Dauer solcher Zusammenkünfte zwischen wenigen Sekunden und wenigen Minuten (< 5).

Abbildung 17 zeigt eine physische Zusammenkunft vom Typ Planung. Aus der Handlungszusammenfassung („P2 fällt auf, dass noch ein Service für Standorte benötigt wird. P1 informiert sie, dass er die gleich macht und sagt ihr, dass sie erst einmal die Seitenklasse schreiben soll und was da alles rein muss und wie die einzelnen Methoden und Felder heißen sollen.“) geht hervor, dass es sich hier um die Planung eines konkreten nächsten Schrittes und die Absprache von Designentscheidungen handelt. Die Visualisierung bestätigt das. So stehen in Blautönen gehaltene Blöcke für Phänomene, die Kodes der Kategorie „Degree of Freedom“ erhalten haben und in verschiedenen Violettönen eingefärbte Blöcke für die Verwendung von Kodes der Kategorie „Action“. Beide Kategorien werden verwendet für Phänomene, die eine Abstimmung über den nächsten Arbeitsschritt oder eine Designentscheidung darstellen.

In Abbildung 18 ist sichtbar, dass P1 nicht gleich auf die Bestrebung seines Partners, eine Designentscheidung zu fällen, eingeht sondern erst einmal an seinem Code weiterschreibt. Erst nach erneuter Nachfrage besprechen sie sich zu Methoden in ihren Klassen. Die Handlungszusammenfassung für diese Zusammenkunft lautet „P2 macht sich immer noch Gedanken, welche Daten er wie darstellen will. Zuerst macht P2 einen Einwurf, auf den P1 nicht direkt reagiert sondern erst einmal weitermacht mit seiner Tätigkeit. Anschließend unterhalten sie sich im Detail über die Funktion von DAOs und Services und welche Methoden sie dort implementieren und aufrufen wollen.“. Durch die überwiegende Verwendung von Kodes der Kategorie *Degree of Freeedom* muss es sich hierbei tatsächlich um eine Zusammenkunft vom Typ Planung handeln.

Die Zusammensetzung der Paare scheint keinen Einfluss auf die Häufigkeit und Dauer oder die während einer solchen Zusammenkunft besprochenen Themen zu haben.

4.1.5 Integration

Die letzten beiden Arten der Zusammenkunft lassen sich nicht direkt aus der Verwendung von Kodes bestimmter Kategorien ablesen. Bei der Zusammenkunft vom Typ Integration geht es um das Testen des Zusammenspiels der einzelnen Komponenten, die von den beiden Partnern entwickelt wurden. Dabei können auch auftretende Fehler gemeinsam behoben werden. Es gibt für diese Art Zusammenkunft auch keine typischen Sätze, die zwischen den Probanden fallen, wie es bei Strategie oder Planung der Fall ist. Also muss die Kategorisierung

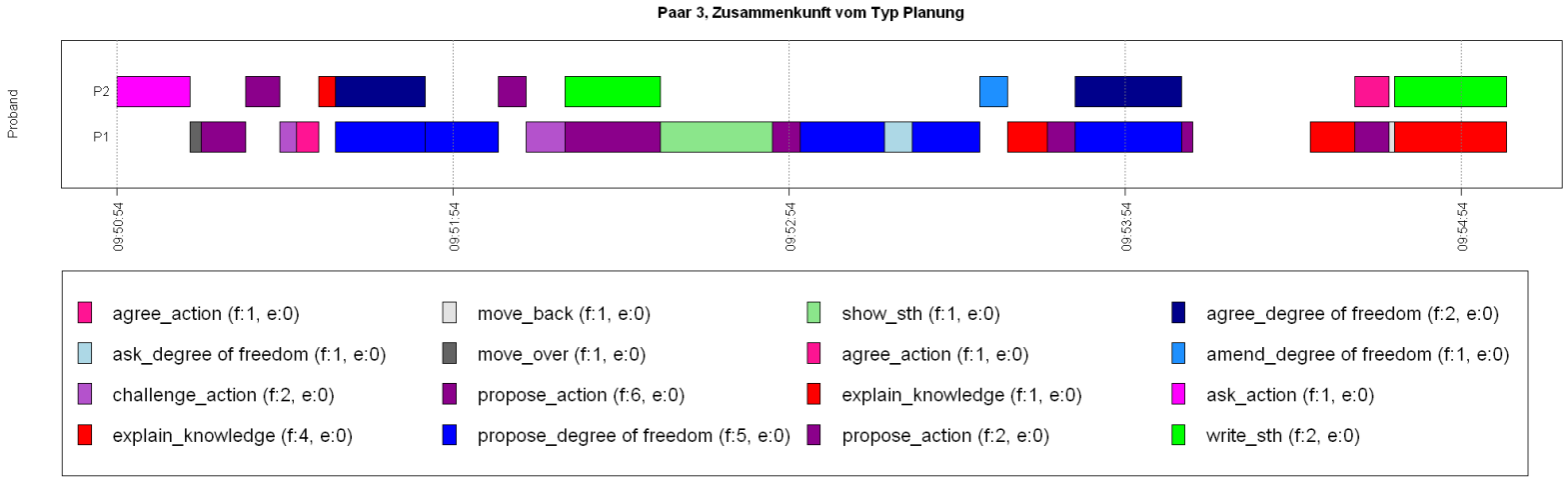


Abbildung 17: Zusammenkunft vom Typ Planung bei Paar 3

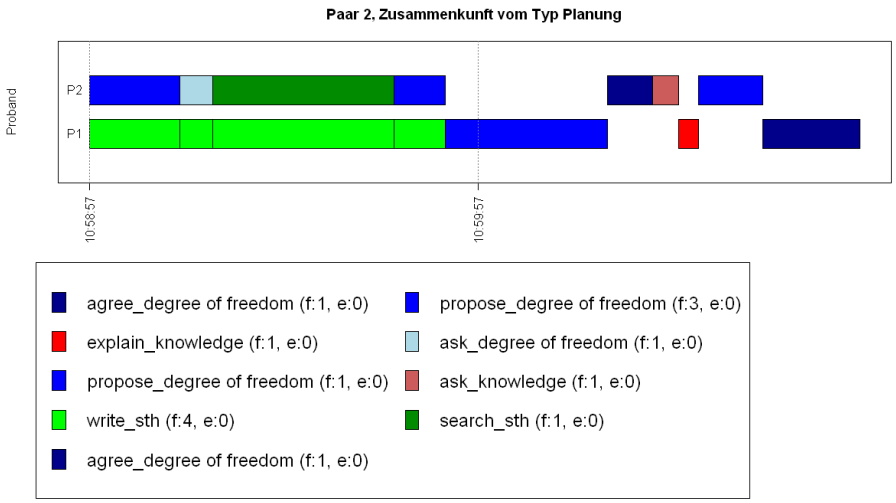


Abbildung 18: Zusammenkunft vom Typ Planung bei Paar 2

aufgrund der Beobachtung der Aktivitäten der Partner und dem was gesagt und in welchem Kontext es gesagt wurde, erfolgen. Oft kommt es jedoch zur Verwendung des Kodes *verify_sth*, den ich immer dort benutzt habe, wo es um die Überprüfung eines Artefakts (geschriebener Code, Konfigurationsdatei, etc.) geht, um sicherzustellen, dass es korrekt arbeitet. Dieser Kode, in Verbindung mit dem Kode *write_sth*, der verwendet wird um das Schreiben von Code zu markieren, und Kodes für die Beschreibung von Kommunikation können jedoch auf eine Zusammenkunft vom Typ Integration hindeuten.

Integrationszusammenkünfte sind immer physisch und finden wenn dann kurz nach der Fertigstellung einer Komponente statt. Oft findet dann davor eine Zusammenkunft vom Typ Projektwissen statt, wo die Partner sich über den Fortschritt der Arbeiten austauschen. In allen Fällen findet vorher eine Synchronisation der Codebasis statt, indem beide Partner ihren Code einchecken und anschließend ein Update durchführen. Manchmal geht ein solcher Informationsaustausch über den Stand der Arbeiten aber auch direkt über in eine Zusammenkunft vom Typ Integration. Integrationen können auch durch einen auftretenden Fehler unterbrochen werden. Dann trennen sich die beiden Partner wieder und der für die fehlerhafte Komponente Verantwortliche behebt den Fehler, während der andere an seiner Aufgabe weiterarbeitet. Anschließend kommt es wieder zu einer Integrationssitzung. Bei dem Paar mit dem großen Unterschied im Kenntnisstand (Paar 1) kam es nicht ein einziges mal zu einer Zusammenkunft vom Typ Integration. Stattdessen testete jeder für sich alleine die Funktionalität seiner Komponenten.

Abbildung 19 zeigt, dass es manchmal nicht einfach ist, anhand der Kodierung den Typ einer Zusammenkunft zu erkennen. Während dieser Zusammenkunft kamen Kodes aller Kategorien zum Einsatz. Die Abbildung zeigt eine Zusammenkunft vom Typ Integration, die zum Ende in eine vom Typ Strategie

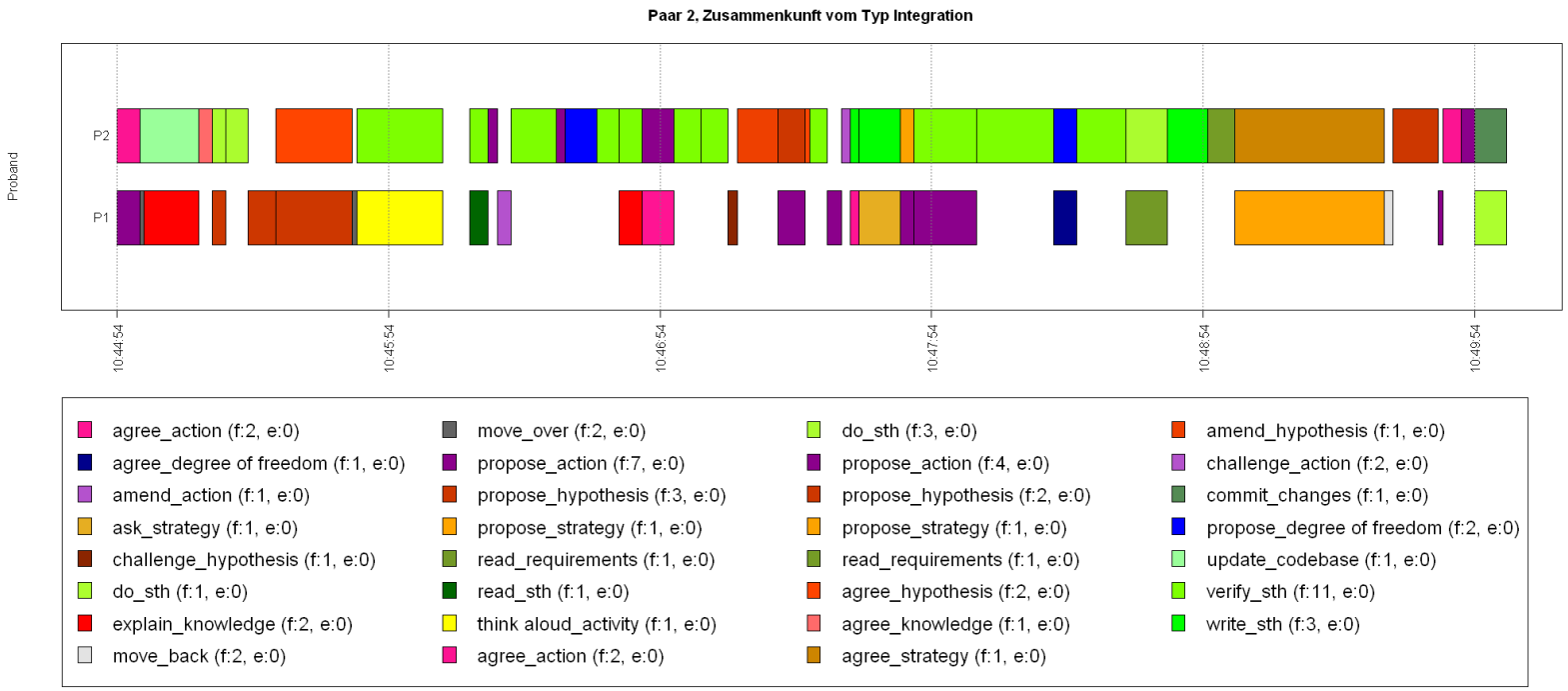


Abbildung 19: Zusammenkunft vom Typ Integration bei Paar 2

übergeht. Bei der Zuordnung waren zwei Dinge hilfreich. Es kam oft zur Verwendung von HCI-Kodes wie *verify_sth* und *write_sth* (in Grüntönen gehalten) und die Handlungszusammenfassung zeigt eine Integration an: „Nachdem die beiden sich in der vorangegangenen Zusammenkunft getrennt haben, damit P1 noch einen Fehler behebt, kommen die beiden nun wieder zusammen um den bisherigen Entwicklungsstand zu testen. Anschließend stimmen sie das weitere Vorgehen ab und trennen sich dann wieder.“. Das Abstimmen des weiteren Vorgehens konnte ich durch die Verwendung von Kodes der Kategorie *Strategy* ablesen. In der Abbildung sind diese orangefarben eingezeichnet.

Zusammenkünfte vom Typ Integration sind von längerer Zeitdauer als die anderen Arten der Zusammenkunft. So dauerte die kürzeste Zusammenkunft 00:07:08 und die längste 00:39:00 (beobachtet bei Paar 2). Die Durchschnittsdauer der insgesamt neun Zusammenkünfte vom Typ Integration beträgt 00:11:48 (Durchschnitt Paar 2: 00:14:43, Durchschnitt Paar 3: 00:09:28).

4.1.6 Fehlersuche

Bei der Zusammenkunft vom Typ Fehlersuche verhält es sich ähnlich. Bei solchen Zusammenkünften geht es um einen Fehler, der bei einem der beiden Partner auftritt und unter Mithilfe des anderen behoben werden soll. Hier treten die gleichen Kodes wie bei der Integration auf, so dass ich nicht direkt von den Kodes ableiten konnte, um welche Art der Zusammenkunft es sich handelt. Ich musste deshalb anhand der Handlungszusammenfassung entscheiden, ob es sich um eine Fehlersuche oder eine Integration handelt. Die Einführung neuer Kodes zur besseren Identifikation einer Fehlersuche wäre nicht möglich gewesen. Die beobachteten Phänomene sind nämlich hier wie überall anders die Gleichen. Ein Wissenstransfer innerhalb einer Fehlersuche ist der gleiche wie innerhalb einer Zusammenkunft vom Typ Projektwissen und wird immer mit Kodes aus der Kategorie Knowledge kodiert. Genau so verhält es sich mit allen anderen Phänomenen. Es ist das Zusammenspiel der einzelnen Phänomene, das eine Zusammenkunft ausmacht.

Fehlersuche wird in aller Regel gemeinsam vor einem Rechner betrieben, es sei denn es handelt sich um einen trivialen Fehler, den die Partner durch einfache Kommunikation ohne genauere Betrachtung des Problems beheben können. Ein Beispiel für solch triviale Fehler sind typische Fehler, die durch einen Neustart des Webservers behoben werden konnten. In solch einem Fall war nur der Vorschlag den Webserver neu zu starten ausreichend, um den Fehler zu beheben. Normalerweise muss aber einer der beiden Partner zum anderen gehen, um sich den Fehler zeigen zu lassen. So dauerte die kürzeste Fehlersuche gerade einmal 00:01:00, die längste 00:09:48.

Zusammenkünfte vom Typ Fehlersuche traten bei Paar 2 und Paar 3 immer zwischen zwei Zusammenkünften vom Typ Integration auf. Diese Integrationssitzungen wurden unterbrochen, weil beim Integrieren eine der Komponenten noch Fehler aufwies. Daraufhin trennten sich die Partner wieder, derjenige mit der fehlerhaften Komponente versucht den Fehler zu beheben, der andere arbei-

tete an seinen Aufgaben weiter. Dabei schaffte es derjenige mit der fehlerhaften Komponente dann nicht den Fehler alleine zu beheben, und es kam zu Zusammenkünften vom Typ Fehlersuche. Nach Behebung der Fehler kam es dann wieder zu einer Integrationszusammenkunft.

Bei Paar 1, dem Paar mit dem großen Unterschied beim Kenntnisstand, trat die Fehlersuche isoliert für sich alleine auf, also nicht zwischen zwei Zusammenkünften vom Typ Integration. Bei Paar 1 konnte ich zudem beobachten, dass die Fehlersuche nur in zwei von neun Fällen das alleinige Thema einer Zusammenkunft war. In allen anderen Fällen hat einer der beiden Partner eine bestehende Zusammenkunft genutzt, um seinen Fehler vorzutragen und um Hilfe beim Partner zu bitten (siehe Abschnitt 4.1.1).

Die letzte Zusammenkunft, die ich beispielhaft darstellen möchte, ist eine physische Zusammenkunft vom Typ Fehlersuche (Abbildung 20). Auf den ersten Blick zeigt sie Ähnlichkeiten mit Abbildung 19. Wie schon erwähnt sind die Kodes die gleichen, die auch typischerweise während einer Zusammenkunft vom Typ Integration eingesetzt werden, mit einer charakteristischen Häufung von HCI-Kodes (Grüntöne). Lediglich die Handlungszusammenfassung kann Aufschluss darüber geben, um welche Art der Zusammenkunft es sich tatsächlich handelt. In diesem Fall lautete diese „P2 hat immer noch Probleme mit einem Fehler und P1 geht rüber und beide suchen gemeinsam. Sie stellen fest, dass P1 vergessen hat, den Spring service in der applicationContext.xml zu registrieren. Außerdem hat er im Service einen Setter für das DAO Objekt vergessen. Nach Behebung der Fehler klappt auch das Hinzufügen von Standorten, allerdings funktioniert das Löschen noch nicht. Um das zu beheben trennen sie sich wieder. Während die beiden auf das Neustarten der Applikation warten, besprechen sie schon das weitere Vorgehen.“. Interessant ist der letzte Satz der Handlungszusammenfassung, der mich auch schon bei der Betrachtung der Kodes stutzig gemacht hat. Gegen Ende der Zusammenkunft gibt es einen kurzen Teil, in dem sich die beiden über das weitere Vorgehen abstimmen, während sie auf das Starten der Applikation warten, in der Annahme, dass der Fehler nun behoben ist und alles klappt. Dies ist in Orangetönen visualisiert und zeigt, dass eine klare Trennung von Zusammenkünften in ihre verschiedenen Arten nicht immer machbar ist.

4.1.7 Bemerkung

Bemerkung ist ein Spezialfall von Zusammenkünften, den ich nur zwei mal beobachtet habe. In seiner Antwort an mich (siehe Anhang E) beschreibt Alistair Cockburn in Punkt 5, dass er es als „gut“ bezeichnen würde, wenn einer der beiden Partner etwas auf dem Bildschirm des anderen entdecken und darauf eingehen würde. Dies ist bei Paar 2 zwei mal passiert. In einem Fall sah der eine Partner eine SVN Fehlermeldung auf dem Bildschirm des anderen und kommentierte das mit „Boah, lösch’ mal deine Properties“. Gemeint waren damit bestimmte Dateien, die von der Datenbank angelegt wurden. Dabei wartete der Kommentator nicht auf die Fertigstellung einer Aktion des anderen und schaute explizit auf seinen Bildschirm, sondern sah die Fehlermeldung scheinbar aus dem Augenwinkel und machte dann diesen Kommentar.

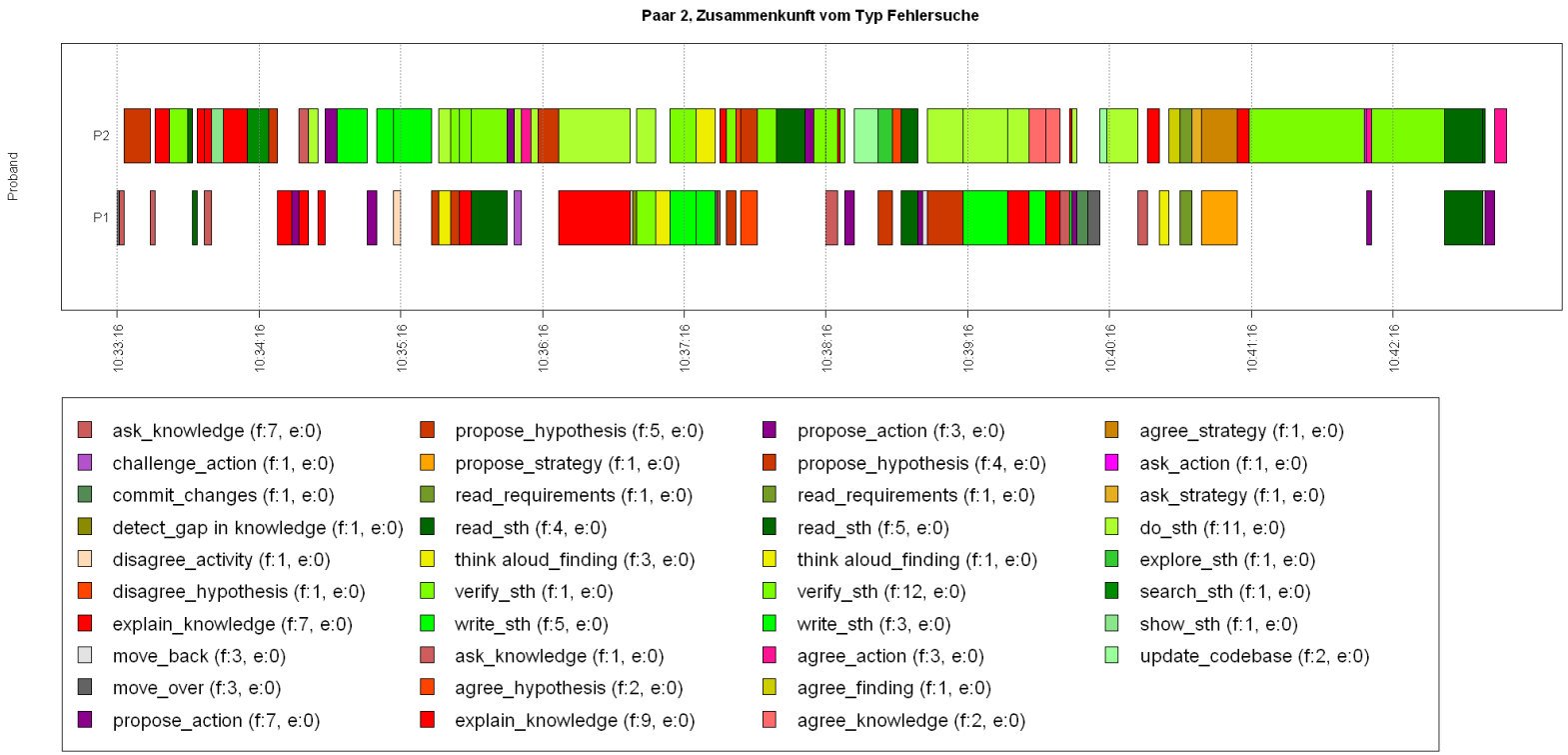


Abbildung 20: Zusammenkunft vom Typ Fehlersuche bei Paar 2

Im zweiten Fall war die Ursache für die Bemerkung sogar noch weniger auffällig, d.h. es gab keine Fehlermeldung mit einem leuchtend roten Warnschild o.ä. Stattdessen hatte der eine Partner Probleme, die Auto-Vervollständigung von Eclipse zu benutzen. Bei der Auto-Vervollständigung reicht es einen Teil eines Namens einzugeben. Drückt man danach eine bestimmte Tastenkombination, vervollständigt Eclipse automatisch den Namen oder bietet eine Liste von Möglichkeiten zur Auswahl an. Offensichtlich klappte das in dem Fall mit einer JPA Annotation nicht und der andere kommentierte das lachend mit „ha, jetzt hast du das auch mal“.

Dass dieser Typ der Zusammenkunft nur bei Paar 2 auftrat ist auffällig. Dieses Paar ist nämlich das einzige, bei dem die beiden Partner schon früher zusammen gearbeitet haben.

4.2 Kenntnisstand und Verhältnis der Probanden zueinander

Wie ich im vorherigen Abschnitt bereits herausgearbeitet habe, schlägt sich der Unterschied beim Kenntnisstand und beim Verhältnis der Probanden zueinander (haben sie vorher schon einmal zusammengearbeitet, ist das Verhältnis zueinander natürlich oder eher gehemmt) auf die Häufigkeit und Art der Zusammenkunft nieder. Es ist noch nicht klar, welche Arten und Häufigkeiten der Zusammenkünfte „gutes“ Side-by-Side Programming ausmachen. Nimmt man jedoch die verspührte Harmonie während der Zusammenarbeit als einen Hinweis, so scheint es ratsam, Partner von in etwa gleichem Kenntnisstand zusammen arbeiten zu lassen. Haben die beiden vorher sogar schon einmal zusammen gearbeitet, kommt es zu Situationen, die Alistair Cockburn als „eine gute Sache“ bezeichnet, nämlich das unverhoffte Kommentieren der Aktion des anderen.

Auch die Häufigkeit und Art der Zusammenkünfte wird scheinbar duch den Kenntnisstand und das Verhältnis der Probanden zueinander beeinflusst. So gibt es bei dem Paar mit dem großen Unterschied beim Kenntnisstand viele Zusammenkünfte vom Typ Fehlersuche, wo der weniger Kenntnisreiche Hilfe von seinem Partner benötigt. Auch wird hier öfter als bei allen anderen Paaren Wissen transferiert und zwar wieder vom Kenntnisreicheren zu seinem Partner. Außerdem gibt es nicht eine einzige Zusammenkunft vom Typ Integration, wo sich beide zusammensetzen und ihre Teile zusammenbringen und testen.

4.3 Zahlen

Als letztes möchte ich einige interessante Zahlen präsentieren:

Paar 2 verbrachte von allen Paaren die meiste Zeit mit Zusammenkünften. Während ihrer Bearbeitungszeit von 02:30:00 hatten sie Zusammenkünfte mit einer Gesamtdauer von 01:31:10, das entspricht 61 % der Zeit. Dieses Paar hatte mit 7 Sekunden die kürzeste Zusammenkunft (vom Typ Projektwissen) sowie

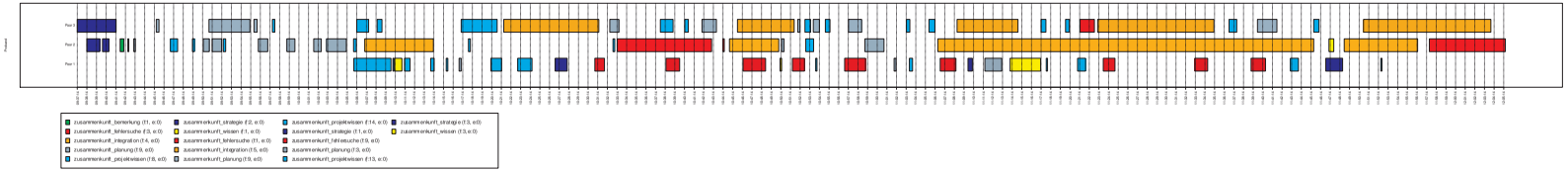


Abbildung 21: Zusammenkunft vom Typ Fehlersuche bei Paar 2

Art der Zusammenkunft	Gesamt	Paar 1	Paar 2	Paar 3
Projektwissen	43 (34)	19 (13)	10 (8)	14 (13)
Wissen	7 (2)	6 (1)	1 (1)	0 (0)
Strategie	9 (2)	4 (0)	4 (2)	1 (0)
Planung	27 (18)	6 (2)	10 (7)	11 (9)
Integration	9 (7)	0 (0)	4 (2)	5 (5)
Fehlersuche	13 (7)	9 (3)	3 (3)	1 (1)
Bemerkung	2 (1)	0 (0)	2 (1)	0 (0)

Tabelle 1: Häufigkeit der Zusammenkünfte

mit 39 Minuten auch die längste (hauptsächlich vom Typ Integration aber zu Beginn auch Planung und am Schluss Strategie). Diese Zusammenkunft ist der Grund, warum Paar 2 von allen Paaren die meiste Zeit mit Zusammenkünften verbrachte. Rechnet man diese Zusammenkunft heraus, sind es nur noch 35 % der Zeit. Das Ignorieren dieser Zusammenkunft scheint aber nicht ratsam, da es auffällig ist, dass gerade dieses Paar, das sich vorher schon kannte, eine so lange Zusammenkunft hat. Außerdem kommt Paar 3 in Summe auf eine in etwa gleiche Gesamtdauer von Integrationszusammenkünften, so dass diese 39 minütige Zusammenkunft nicht aus dem Rahmen fällt. Bei Paar 2 sind es in Summe 00:58:52 verteilt auf vier Zusammenkünfte, bei Paar 3 00:47:22, verteilt auf fünf.

Paar3 verbrachte von den aufgezeichneten 02:27:33 Arbeitszeit 01:16:20 mit Zusammenkünften, was einem Anteil von 51 % entspricht. Paar 1 verbrachte die wenigste Zeit mit Zusammenkünften. Von den 02:05:53 waren es gerade einmal 27 % oder 00:33:46.

Auch hier fällt wie bei den Zusammenkünften vom Typ Bemerkung auf, dass wieder das Paar heraussticht, das schon vorher zusammen gearbeitet hat. Das Paar, das vorher noch nie zusammen gearbeitet hat und wo der Unterschied beim Kenntnisstand am größten war, verbrachte die wenigste Zeit mit Zusammenkünften.

Tabelle 1 stellt die Häufigkeit der Zusammenkünfte Gesamt und nach Paar aufgeschlüsselt dar. Dabei wurden Zusammenkünfte, denen mehrere Typen zugeordnet wurden auch mehrfach gezählt. Eine Zusammenkunft der beispielsweise Projektwissen und Strategie zugeordnet wurde, wurde sowohl für Projektwissen als auch für Strategie gezählt. Die Zahl in Klammern gibt die Zahl der Zusammenkünfte an, die isoliert für sich, also nicht in Verbindung mit einer anderen Art der Zusammenkunft, auftraten.

5 Fazit

In meiner explorativen Arbeit wurde zum ersten Mal versucht, die Vorgänge beim Side-by-Side Programming zu beleuchten. Mein Schwerpunkt lag dabei

auf den Gründen für und den Vorgängen während einer Zusammenkunft. Im Folgenden sind meine Hypothesen noch einmal zusammengefasst.

Es gibt sechs Arten der Zusammenkunft und den Spezialfall Bemerkung. Projektwissen ist die am häufigsten auftretende Art der Zusammenkunft, bei der Wissen über das Projekt ausgetauscht wird und verläuft überwiegend nicht-physisch. Paare, die vorher schon einmal zusammengearbeitet haben und in etwa den gleichen Kenntnisstand besitzen, benötigen weniger solcher Zusammenkünfte als Paare, die vorher noch nie zusammengearbeitet haben und/oder einen großen Unterschied beim Kenntnisstand haben. Ist einer der beiden gegenüber dem anderen Partner gehemmt, beispielsweise weil der Unterschied beim Kenntnisstand zu groß ist, werden solche Zusammenkünfte genutzt, um andere Zusammenkünfte wie beispielsweise Fehlersuche einzuleiten. Kommt es bei einer Zusammenkunft vom Typ Projektwissen zu einer Mitteilung über die Fertigstellung einer Komponente, folgt darauf häufig eine Zusammenkunft vom Typ Integration, um diese zu testen, oder eine vom Typ Strategie oder Planung, um das weitere Vorgehen zu besprechen.

Zusammenkünfte vom Typ Wissen sind physisch und beinhalten den Transfer von Wissen, wie beispielsweise den korrekten Einsatz einer bestimmten Technologie. Ist der Unterschied beim Kenntnisstand der Partner groß, kommt es häufiger zu solchen Zusammenkünften, als wenn der Kenntnisstand in etwa gleich ist. Dabei fließt bei einem großen Unterschied beim Kenntnisstand das Wissen vom kenntnisreicheren zum schwächeren Partner.

Bei Zusammenkünften vom Typ Strategie, die nicht-physisch verlaufen, kommt es zur Aufteilung der Aufgaben oder zur Planung des allgemeinen Vorgehens. Diese Planung umfasst immer mehr als den konkreten, nächsten Arbeitsschritt. Diese Zusammenkünfte treten am Anfang der Programmiersitzung und nach einer Mitteilung über den Bearbeitungsstand (Projektwissen) oder nach einer Integrationssitzung auf. Strategiezusammenkünfte treten oft nicht alleine sondern gleichzeitig mit Zusammenkünften vom Typ Projektwissen oder vom Typ Integration auf. Im letzteren Fall sind solche Zusammenkünfte physisch.

Planungszusammenkünfte drehen sich um die Planung des nächsten konkreten Arbeitsschritts oder die Ausprägung eines bestimmten Freiheitsgrades. Sie treten an jeder Stelle der Programmiersitzung auf und dauern zwischen wenigen Sekunden und wenigen Minuten. Die Zusammensetzung der Paare hat keinen Einfluss auf die Häufigkeit und Dauer oder die während einer solchen Zusammenkunft besprochenen Themen.

Zusammenkünfte vom Typ Integration sind lang und physisch und dienen dem Testen des Zusammenspiels der in Einzelarbeit entwickelten Komponenten. Bei so einer Zusammenkunft können auch auftretende Fehler gemeinsam behoben werden. Ist der Unterschied beim Kenntnisstand eines Paares groß und wird die Zusammenarbeit dabei nicht als harmonisch empfunden, kommt es während der Programmiersitzung zu keiner Zusammenkunft vom Typ Integration. Integrationssitzungen sind beendet, wenn entweder das Testen erfolgreich war, dann kann es zu einer Strategie- oder Planungszusammenkunft zum Festlegen des weiteren Vorgehens kommen, oder wenn es zu einem Fehler in einer der Komponenten

kommt, der in Einzelarbeit vom verantwortlichen Partner behoben wird. Nach Behebung des Fehlers kommt es dann wieder zu einer Integrationssitzung.

Bei Zusammenkünften vom Typ Fehlersuche treffen sich die Partner meistens vor einem Rechner, es sei denn es handelt sich um triviale Fehler. Bei Paaren mit ausgeglichenem Kenntnisstand treten diese Zusammenkünfte immer zwischen zwei Zusammenkünften vom Typ Integration auf. Bei einem Paar mit großen Unterschieden beim Kenntnisstand ist Fehlersuche oft nicht das alleinige Thema einer Zusammenkunft sondern eine bestehende Zusammenkunft wird vom weniger kenntnisreichen Partner dazu genutzt seinen Fehler vorzutragen und den anderen um Hilfe zu bitten.

Einen Sonderfall nimmt Bemerkung ein. Dieser Typ Zusammenkunft tritt nur bei Paaren auf, die schon zusammen gearbeitet haben und in etwa einen gleichen Kenntnisstand besitzen. Bei solchen Zusammenkünften geht es um die Kommentierung der Tätigkeit oder des Bildschirminhalts des Partners — etwas, das Cockburn als „eine gute Sache“ bezeichnet (siehe Anhang E).

In einigen Punkten decken sich meine Erkenntnisse mit den Erwartungen von Cockburn. Wie bereits erwähnt, konnte ich Zusammenkünfte vom Typ Bemerkung beobachten. Auch die Zusammenarbeit an unterschiedlichen Teilen des gleichen Systems (Vergleiche Anhang E Punkt 6) und die damit verbundene notwendige Kommunikation, das Integrieren und die Fehlersuche konnte ich beobachten. Dies ist aber hauptsächlich der Aufgabenstellung geschuldet, die ein paralleles Arbeiten am gleichen System nahe legte. Wie sich das Ganze bei der Arbeit an unterschiedlichen Systemen verhält, kann ich nicht sagen.

Auch habe ich nach Anzeichen für ein Unwohlsein der Partner gesucht und sie bei dem Paar mit dem großen Unterschied beim Kenntnisstand auch gefunden. Nicht nur bewerten sie die Zusammenarbeit weniger gut als die anderen Paare, sie zeigten bzw. sagten auch, dass sie gehemmt bzw. genervt waren. Anders als Cockburn konnte ich also ein Unbehagen bei zumindest einem Paar feststellen. Cockburns andere Punkte beziehen sich auf die Beobachtung von Side-by-Side Programming im Arbeitsalltag. Dadurch, dass ich aber ein Experiment durchgeführt habe, sind sie für meinen Fall nicht relevant.

6 Weiterführende Arbeiten

Dadurch, dass ich letztlich nur drei Paare beim Side-by-Side Programming beobachten konnte, sind meine Hypothesen zum Side-by-Side Programming eher schwach. Durch den Einsatz von GT sind sie zwar in den Daten verankert, die Gründe für Zusammenkünfte, gerade wenn sie von den Eigenschaften der Paare abhängig sind, konnten aber nur aus der Beobachtung von nur drei Paaren abgeleitet werden.

In Folgestudien müssen daher zwei Dinge untersucht werden. Zum einen muss überprüft werden, ob auch dort die von mir herausgearbeiten Arten der Zusam-

menkunft mit den von mir beschriebenen Eigenschaften auftreten. Zum anderen müssen anhand einer größeren Anzahl von Paaren der Einfluss der Paare auf die Zusammenkünfte untersucht werden.

Sollte in einer Folgestudie wieder ein Experiment durchgeführt werden, scheint eine Aufgabe im Bereich einer mehrschichtigen Anwendung sinnvoll zu sein. Die Probanden haben damit die Möglichkeit, die Aufgabe untereinander aufzuteilen. Somit werden Zusammenkünfte und Trennungen begünstigt. Andererseits wäre es auch interessant Side-by-Side Programming mit einer Fallstudie im Arbeitsalltag zu beobachten.

Die Durchführung eines Workshops vor einem Experiment ist aus zweierlei Gründen ratsam. Zum einen werden die Teilnehmer durch einen Experten auf den gleichen Wissenstand über die einzusetzenden Technologien gebracht und lernen diese auch praktisch kennen. Zum anderen bietet ein Workshop dem Forscher schon vor dem Experiment die Möglichkeit, die Probanden und ihre Eigenschaften, Kenntnisse und Arbeitsweise kennenzulernen. Beim Entwurf und der Durchführung einer Folgestudie sollten aber unbedingt mehr als nur ein Forscher mitwirken. So war es mir nicht möglich, während des Workshops schon Beobachtungen anzustellen, da mich die Durchführung des Workshops ganz vereinnahmt hat.

Literatur

[1] Kent Beck and Cynthia Andres. Extreme Programming Explained: Embrace Change (2nd Edition). Addison-Wesley Professional, 2004.

[2] John T. Nosek. The case for collaborative programming. Commun. ACM, 41(3):105–108, 1998.

[3] Alistair Cockburn and Laurie Williams. The Costs and Benefits of Pair Programming. Extreme programming examined, pages 223–243, 2001.

[4] Laurie Williams, Robert R. Kessler, Ward Cunningham, and Ron Jeffries. Strengthening the Case for Pair Programming. IEEE Software, 17(4):19–25, 2000.

[5] L. Williams and R. Kessler. Pair Programming Illuminated. Addison-Wesley Professional, 2002.

[6] Alistair Cockburn. Crystal Clear. Addison-Wesley, 2004.

[7] Jerzy R. Nawrocki, Michal Jasinski, Lukasz Olek, and Barbara Lange. Pair Programming vs. Side-by-Side Programming. Lecture Notes in Computer Science, 3792:28–38, 2005.

[8] C. Bauer and G. King. Java Persistence with Hibernate. Manning Publications Co. Greenwich, CT, USA, 2006.

[9] Sun Microsystems. JSR 220: Enterprise JavaBeans, Version 3.0. EJB Core Contracts and Requirements. Technical report, URL: http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html, 2006.

[10] Sun Microsystems. JSR 220: Enterprise JavaBeans, Version 3.0. Java Persistence Api. Technical report, URL: http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html, 2006.

[11] Diverse. Premiere Pro Wiki: Xvid, http://ppro.wikia.com/wiki/Xvid. Zugriff am 09.04.2008.

[12] A.L. Strauss and J.M. Corbin. Grounded Theory: Grundlagen Qualitativer Sozialforschung. Beltz, Psychologie-Verl.-Union, 1996.

[13] Stephan Salinger, Laura Plonka, and Lutz Prechelt. A Coding Scheme Development Methodology Using Grounded Theory for Qualitative Analysis of Pair Programming. 2007.

Anhänge

Anhang A Aufgabe des Experiments

Workshop Java Webentwicklung

Aufgabe 7

In diesem Experiment sollt Ihr Eure bestehende Webanwendung um zusätzliche Funktionalität erweitern. Wie bisher sollt Ihr dabei im Paar arbeiten. Hierbei stehen Euch wie gewohnt zwei Rechner zur Verfügung.
Für die Bearbeitung der Aufgabe habt Ihr 2,5 Stunden Zeit. Organisiert Euch so, dass Ihr diese Vorgabe einhalten könnt und auch wirklich innerhalb dieser Zeit fertig werdet.
Während des Experiments steht Euch kein Betreuer zur Beantwortung von Fragen zur Verfügung, es sei denn Ihr habt Probleme mit dem Rechner oder dem Aufzeichnungsprogramm.
Solltet Ihr fachliche Fragen haben, müsst Ihr Euch an die Dokumentation der Softwareomponenten halten, die auch auf der Inhaltsseite des Workshops verlinkt ist.
Während der Arbeit werdet Ihr aufgezeichnet (Audio, Video, Programmiertätigkeit).

Bevor Ihr mit der Arbeit beginnt, befolgt bitte folgende Schritte:

- Meldet Euch an beiden Rechnern mit Eurer Kennung an.
- Stellt bitte sicher, dass der Bildschirmschoner abgeschaltet ist.
- Stellt bitte sicher, dass in den Energieoptionen der Bildschirm nie ausgeschaltet wird.
- Auf dem Desktop befindet sich ein Symbol „Camtasia Recorder“. Ruft damit den Camtasia Recorder auf beiden Rechnern auf. Startet die Aufzeichnung mit einem Klick auf den Aufzeichnen Button, möglichst auf beiden Rechnern gleichzeitig.

Aufgabenstellung:

Erweitert die Webanwendung um die Möglichkeit, Exemplaren einen Standort zuzuweisen. Dieser soll angeben, in welchem Gebäude mit welcher Adresse sich ein Exemplar befindet. Legt Seiten zum Erstellen und Verändern von Standorten an. Außerdem soll ein Standort auch gelöscht werden können. Falls dieser Standort noch bei einem Exemplar hinterlegt ist, soll dem Benutzer eine Fehlermeldung angezeigt werden. Legt außerdem eine Seite an, die für einen Standort alle dort eingestellten Exemplare und Details zu diesen auflistet sowie eine Seite, die alle Standorte mit allen dort jeweils eingestellten Exemplaren auflistet.
Ihr müsst Euch nicht speziell um das Design oder ein besonders schönes Aussehen Eurer Lösung kümmern. Wichtig ist, dass Ihr die Funktionen innerhalb der vorgegebenen Zeit umsetzt.

Anhang B Fragebogen 1

Datum: _____ Name:_____

Fragebogen nach der Ersten Programmierungssitzung im Workshop Java Webentwicklung

Der Fragebogen wird anonymisiert ausgewertet. Die Namensangabe dient nur zur Absicherung gegen eventuelle Fehler oder Lücken und wird entfernt, sobald die Zuordnung der Daten erreicht ist.
Lediglich die Experimentatoren der AG Software Engineering bekommen Deinen Namen zu sehen.

1. Habt Ihr die Aufgabe vollständig gelöst?

☐ JA ☐ NEIN

2. Wie schwierig/leicht fandest Du die zu lösende Aufgabe?
(sehr leicht, leicht, meinem Können/Wissen angemessen, ziemlich schwer, sehr schwer):

_____

3. Beschreibe kurz die größten Schwierigkeiten mit der Aufgabe:

_____

4. Habt Ihr die Möglichkeit Euch zu trennen und an zwei Rechnern zu arbeiten genutzt?

☐ JA ☐ NEIN

Falls Ihr Euch getrennt habt:

5. In welchen Situationen/aus welchen Gründen habt Ihr Euch getrennt?

_____

Falls Ihr Euch getrennt habt:

6. Seid Ihr zwischenzeitlich wieder zusammen gekommen?

☐ REGELMÄßIG ☐ SELTEN ☐ NIE

Falls Ihr regelmäßig oder selten wieder zusammen gekommen seid:

7. In welchen Situationen/aus welchen Gründen seid Ihr wieder zusammen gekommen?

_____

Falls Ihr regelmäßig oder selten wieder zusammen gekommen seid:

8. Habt Ihr bei Eurem Zusammenkommen wieder gemeinsam an einem Rechner gearbeitet?

☐ IMMER ☐ OFT ☐ SELTEN ☐ NIE

Falls Ihr **nicht** wieder zusammen gekommen seid:

9. Warum seid Ihr nicht wieder zusammen gekommen?

_____

Falls Ihr Euch getrennt habt:

10. Wie beurteilst Du die Auswirkungen Eurer Trennungen in Hinsicht auf das erfolgreiche (Vollständigkeit, Richtigkeit, in der vorgegebenen Zeit, etc.) Bearbeiten der Aufgabe?

_____

Falls Ihr Euch **nie** getrennt habt:

11. Warum habt Ihr Euch nicht getrennt?

_____

12. Hast Du die Zusammenarbeit mit Deinem Partner als hilfreich empfunden?

☐ JA ☐ NEIN

13. Welche speziellen Aspekte fandest Du an der Zusammenarbeit hilfreich?

_____

14. Welche speziellen Aspekte fandest Du an der Zusammenarbeit störend?

_____

15. Wie harmonisch hast Du die Zusammenarbeit mit Deinem Partner empfunden? (sehr harmonisch, harmonisch, neutral, weniger harmonisch, nicht harmonisch)

_____

16. Sonstige Anmerkungen (zur Aufgabe, zum Ablauf, zum Workshop, etc.):

_____

Anhang C Fragebogen 2

Datum: _____ Name: _____

Fragebogen zum Workshop Java Webentwicklung

Fragen zur Person

Der Fragebogen wird anonymisiert ausgewertet. Die Namensangabe dient nur zur Absicherung gegen eventuelle Fehler oder Lücken und wird entfernt, sobald die Zuordnung der Daten erreicht ist.
Lediglich die Experimentatoren der AG Software Engineering bekommen Deinen Namen zu sehen.

Was studierst Du (z.B. Bachelor Informatik) und in welchem Fach- und Gesamtsemester bist Du?

_____
Abschluss Fach Fachsemester Hochschulsemester

Wie viele Jahre Programmiererfahrung in Java hast Du etwa? _____

Und in Webentwicklung mit Java? _____

Als wie geübt im Umgang mit der Entwicklungsumgebung Eclipse würdest Du Dich selbst bezeichnen? (Schulnoten 1-6) _____

Wie gut schätzt Du Deine Programmierkenntnisse im Vergleich zu anderen aus Deinem Semester ein? Ich gehöre

☐ zu den fähigsten 5%
☐ zu den fähigsten 10%
☐ zu den fähigsten 20%
☐ zu den fähigsten 40%
☐ zum Durchschnitt
☐ zu den unfähigsten 40%
☐ zu den unfähigsten 20%
☐ zu den unfähigsten 10%
☐ zu den unfähigsten 5%

Welche Teilaufgaben des Programmierens macht Dir am meisten Spaß (mehrere, auch eigene Antworten sind möglich)?

- ☐ Anforderungsanalyse
- ☐ Spezifikation
- ☐ Design
- ☐ Coding
- ☐ Debugging
- ☐ Testen
- ☐ Eigene Ideen umsetzen
- ☐ Bibliotheken verwenden
- ☐ Algorithmen entwickeln
- ☐ _____

Fragen zur Zusammenarbeit:

Die folgenden Fragen beziehen sich auf die Zusammenarbeit mit Deinem Übungspartner während des Workshops:

Hattest Du schon vorher mit **diesem** Partner gemeinsam programmiert?

☐ JA ☐ NEIN

Wenn ja: Wie regelmäßig ist Eure Zusammenarbeit (in Stunden pro Woche über welchen Zeitraum, z.B. 2 Stunden pro Woche in den letzten 6 Monaten)?

_____

Bewerte die Zusammenarbeit mit **diesem** Partner während Eurer gemeinsamen Programmierungssitzungen während des Workshops. (Schulnoten 1-6)
Note:_____

Wenn nein: Arbeitest Du auf einem anderen Gebiet gelegentlich oder häufig zusammen (z.B. gemeinsam Übungsaufgaben lösen)?

☐ JA ☐ NEIN

Fragen zum Workshop:

Bezüglich des im Workshop erlernten: Wie würdest Du Deine Erfahrung mit Hibernate vor und nach dem Workshop einschätzen? (Schulnoten 1-6)

Theoretisches Verständnis vorher: _____ nachher: _____

Praktische Anwendung vorher: _____ nachher: _____

Und mit Tapestry? (Schulnoten 1-6)

Theoretisches Verständnis vorher: _____ nachher: _____

Praktische Anwendung vorher: _____ nachher: _____

Fühltest Du Dich während des Workshops überfordert?

☐ OFT ☐ AB UND ZU ☐ SELTEN ☐ NIE

Fühltest Du Dich während des Workshops unterfordert?

☐ OFT ☐ AB UND ZU ☐ SELTEN ☐ NIE

Wie fandest Du den Workshop? Was hat Dir gefallen? Was hat Dir nicht gefallen? Was könnte man besser machen? Hast Du den Eindruck etwas Sinnvolles gelernt zu haben?

Anhang D E-Mail an Alistair Cockburn

Dear Dr. Cockburn,

In my master's thesis I am researching into Side-by-Side Programming as described by you in Crystal Clear. I am paying particular attention on what activities lead to a get-together or a breakup of the two programmers and want to formulate a theory about it using Grounded Theory based on audio and video recordings of programmers.

What do you think leads to a get-together/breakup of the programmers? What other phenomena are important specifically in a side-by-side context? Comparing your expectations to my own findings would add more credibility to my work.

I look forward to hearing from you,

Ulrich Stärk

Anhang E Antwort von Alistair Cockburn

Hi, Ulrich,

Very nice, thank you ----

1. It takes effort for two people to move their screens to within 50 cm of each other ... so they have to concretely decide to do that, it won't be an accident.

2. Similarly, it will take effort for them to move their screens apart from each other again ... which means they have to concretely decide to stop doing side-by-side ... this is good, because laziness and inertia are likely to hold the practice in place, unlike pair programming, where laziness and inertia keep people apart.

3. My friend who tried it spent most of his time writing emails to bosses, and thought he wouldn't like having his junior colleague read his emails. He sort of got used to this ... but what he liked was that he could watch his junior colleague write code, and would periodically kibbitz on that code. I would expect this ''I'm not comfortable with X watching my screen'' to be fairly common.

4. The two programmers on the same team doing this expressed no

dismay or discomfort.

5. I would watch the videos for moments when person A is doing his/her own task, and then suddenly notices person B doing something, and then jumps over to comment on person B's task. This should be considered a ''good thing'', I think.

6. I would look for times when they are working actively and simultaneously on two communicating aspects of the same task or system;
- e.g. ''You compile that, I'll set up the build; let me know when the compile is done and I'll hit ''Build''' (or similar);
- or ''You fix the bug in that piece of code; I'll update the test case over here'' (thus parallelizing two tasks that one person would have to do sequentially).
- or ''You fix the server side, I'll fix the client side, let me know when you're ready''

7. I'd watch for indications of discomfort, e.g. ''too nosy'', ''too loud on the phone'', ''not enough privacy'', whatever else is related. ...

that's all for now --- will you send me the results, please?

Anhang F Schulungsmaterialien

F.1 Einführung in Hibernate

Freie Universität Berlin

Workshop Java Webentwicklung
Einführung in Hibernate

Ulrich Stärk

Ablauf

- Montag bis Donnerstag
  - 09:00 Uhr s.t. Beginn, bis ca. 17:00 Uhr
  - 1 Stunde Mittagspause
- Donnerstag Experiment
  - Aufzeichnung der Programmiertätigkeit
  - Anwesenheitspflicht
- Keine reguläre Lehrveranstaltung
  - Keine Leistungspunkte oder Scheine, Teilnahmebestätigung möglich
- Workshop
  - Keine Vorlesung
  - Kurze Einführung in die jeweiligen Techniken, dann direkt Programmierung
  - Bringt Euch ein, wenn Ihr das eine oder andere schon kennt!

Ablauf

- Montag:
  - Hibernate
  - Spring (sehr kurz)
- Dienstag:
  - Tapestry
- Mittwoch:
  - Weiterentwicklung unserer Webanwendung
  - Tapestry
- Donnerstag:
  - Weiterentwicklung unserer Webanwendung
  - Experiment
- Bitte an ALLEN Tagen da sein, die Übungen bauen aufeinander auf!

Hibernate

- Object/Relational Mapping (ORM)
  - automatische, transparente Abbildung von Objekten auf relationales Modell
  - unter Zuhilfenahme von Metadaten, die die Abbildung beschreiben
- Wieso ORM?
  - handgeschriebener JDBC Code teuer und unflexibel
  - viele Probleme wie Vererbung, Assoziationen und Navigierung des Objektgraphen gelöst
  - Wechsel des Datenbanksystems in der Regel unproblematisch

Einordnung

- Hibernate für sich erst einmal komplett benutzbar
  - mit eigener API
  - mit eigener Query Language (HQL)
- EJB3 Standard spezifiziert Enterprise Java Beans
  - Teil 1 handelt von Session Beans, Message Driven Beans, Deployment, Programmiermodell etc.
  - Teil 2 handelt von Persistenz
    - ORM Metadaten, Interfaces, Query Language (JPA QL)
    - Java Persistence API (JPA)
- Viele Konzepte von Hibernate flossen in die Entwicklung der JPA
  - z.B. HQL ≈ JPA QL (mit ein paar wenigen Erweiterungen in HQL)
- JPA Umsetzungen sollen gegeneinander austauschbar sein und auch außerhalb eines EJB3 Containers laufen können

Komponenten

- Hibernate Core
  - bietet die ORM Funktionalität
  - Konfiguration und Mapping mit XML Metadaten
- Hibernate Annotations
  - Mapping direkt mittels JPA oder Hibernate Annotationen in den einzelnen Klassen
    - JPA Annotationen garantieren Austauschbarkeit mit anderen ORM Tools, die die JPA beherrschen
    - Hibernate Annotationen bieten in einigen Fällen mehr Funktionalität
- Hibernate EntityManager
  - Umsetzung der Programmier-Interfaces der JPA
  - Aber auch Rückgriff auf Hibernate APIs möglich
- Alle zusammen sind eine komplette Implementierung der JPA

Konfiguration

- Ordner META-INF im Verzeichnis src, darin persistence.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
  version="1.0">
  <persistence-unit name="uebung1">
    <properties>
      <property name="hibernate.archive.autodetection" value="class" />
      <property name="hibernate.connection.driver_class"
        value="org.hsqldb.jdbcDriver" />
      <property name="hibernate.connection.url"
        value="jdbc:hsqldb:file:testdb;shutdown=true" />
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.HSQLDialect" />
      <property name="hibernate.connection.username" value="sa" />
      <property name="hibernate.connection.password" value="" />
      <property name="hibernate.hbm2ddl.auto" value="update" />
      <property name="hibernate.connection.shutdown" value="true" />
    </properties>
  </persistence-unit>
</persistence>
```


Einfaches Mapping

```java
package workshop;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Cat {

    private int id;
    private String name;
    private int age;

    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    public int getAge() { return age; }
    public void setAge(int age) {
        this.age = age;
    }

    public String getName() { return name; }
    public void setName(String name) {
        this.name = name;
    }
}
```

Einfaches Mapping

```java
package workshop;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name="KATZE")
public class Cat {

    private int id;
    private String name;
    private int age;

    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="CAT_ID")
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    @Column(nullable=false)
    public int getAge() { return age; }
    public void setAge(int age) {
        this.age = age;
    }

    @Column(nullable=false)
    public String getName() { return name; }
    public void setName(String name) {
        this.name = name;
    }
}
```


equals() und hashCode()

- Wichtig: equals() und hashCode() überschreiben

```java
@Override
public boolean equals(Object o) {
    if(o == null || o.getClass() != this.getClass())
        return false;
    return ((Cat)o).id == this.id;
}

@Override
public int hashCode() {
    return id;
}
```

- Besser: Natürliche Merkmale in equals() und hashCode() benutzen anstatt künstlichen Schlüssel (id)

Datenbankzugriff

```java
EntityManagerFactory emf = Persistence.createEntityManagerFactory("uebung1");
EntityManager em = emf.createEntityManager();
EntityTransaction tx = null;
try {
    tx = em.getTransaction();
    tx.begin();

    /* tu irgendwas */

    tx.commit();
} catch (RuntimeException e) {
    if(tx != null)
        tx.rollback();
    e.printStackTrace();
} finally {
    em.close();
}
```


Instanzen Laden

- Anhand der ID

```java
Cat cat = em.find(Cat.class, 1);
```

- Anhand des Namens mit JPA QL

```java
Query query = em.createQuery("from Cat c where c.name=:name");
query.setParameter("name", "Felix");
/*
alternativ:
Query query = em.createQuery("from Cat c where c.name=?");
query.setParameter(0, "Felix");
*/
Cat cat = (Cat) query.getSingleResult();
```

- Anhand des Namens mit Criteria API (native Hibernate API)

```java
Session session = (Session) em.getDelegate();
Criteria criteria = session.createCriteria(Cat.class);
criteria.add(Expression.eq("name", "Felix"));
Cat cat = (Cat) criteria.uniqueResult();
```

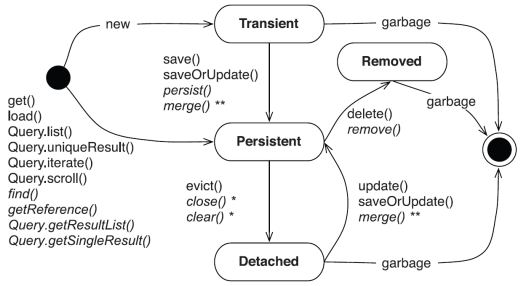

Instanzen speichern

- Neues Objekt persistent machen

```java
Cat cat = new Cat();
cat.setName("Felix");
cat.setAge(1);
em.persist(cat);
```

- Veränderungen an einem Objekt speichern
  - Innerhalb eines persistence context: automatisch beim commit der nächsten Transaktion
  - Außerhalb (detached instances): `em.merge()`

Lebenszyklus eines Objekts



* Hibernate & JPA, affects all instances in the persistence context
** Merging returns a persistent instance, original doesn't change state

Figure 9.1 Object states and their transitions as triggered by persistence manager operations

aus Christian Bauer, Gavin King: Java Persistence with Hibernate

Workshop Java Webentwicklung

Übung 1 – Einfache Hibernate Mappings

1)
   a) Öffnet einen Kommandozeilenprompt (Start -> Ausführen -> cmd) und wechselt in das Verzeichnis `Z:\Workspace`.
   b) Erstellt hier mit Maven ein neues Projekt. Benutzt dazu den Archetype uebung1 der groupId work.shop. Das Aufruf sieht dann wie folgt aus:

```
mvn archetype:create \
 -DarchetypeGroupId=work.shop     \
 -DarchetypeArtifactId=uebung1    \
 -DarchetypeVersion=1.0-SNAPSHOT  \
 -DgroupId=<Euer Paketname>       \
 -DartifactId=Uebung1
```

   c) Wechselt nun in den neuen Ordner Uebung1 und führt dort das Kommando `mvn eclipse:eclipse` aus.
   d) Importiert das Projekt nun in Euren Eclipse Workspace. Evtl. müsst ihr die Variable M2_REPO auf Euer Repository verweisen lassen.

2)
   a) Erstellt eine Benutzerklasse mit dem Namen `Benutzer`. Diese Klasse soll einen Benutzer in unserer Anwendung repräsentieren. Ein Benutzer braucht natürlich bestimmte Eigenschaften. Dazu gehören

      - eine ID, die den Benutzer eindeutig identifiziert
      - ein Benutzernamen, mit dem sich der Benutzer an unserer Anwendung anmeldet
      - ein Passwort, das der Benutzer benötigt, um sich anzumelden
      - den Vor- und Nachnamen des Benutzers
      - beliebige weitere Felder, die euch einfallen

   b) Annotiert diese Klasse wie vorgestellt mit EJB3 Persistenz Annotationen, damit Hibernate diese Klasse verarbeiten kann. Dabei sollen die ID ein automatisch generierter Wert und der Benutzername eindeutig im System sein.

   c) Überschreibt außerdem die Methode `equals()`. Zwei User-Objekte seien gleich, wenn sie die gleiche ID besitzen. Das erfordert auch das Überschreiben der `hashCode()` Methode. Es ist auch sinnvoll `toString()` zu überschreiben.

   d) Wiederholt das Ganze mit der Klasse `Buch`. Ein Buch besitzt

      - eine ID, die es eindeutig identifiziert
      - eine ISBN
      - einen Titel
      - einen Autor (als String)

3) Schreibt eine kleine Konsolenapplikation, die ein oder mehrere Benutzerobjekte erstellt und mittels Hibernate in der Datenbank abspeichert.

Dazu müsst Ihr eine `persistence.xml` schreiben und Sie im Ordner `META-INF` unterhalb Eures Quellordners ablegen. Anschließend besorgt Ihr Euch in Eurem Quellcode eine `EntityManagerFactory` mit dem Aufruf `Persistence.createEntityManagerFactory("euerUnitName")`. Von diesem wiederum bekommt Ihr ein `EntityManager` Objekt. Mit diesem könnt Ihr dann arbeiten. Wiederholt das Ganze für Buchobjekte oder integriert diese Funktionalität in die gleiche Anwendung.

4) Erweitert Eure Anwendung um Funktionen zum Auslesen eines Benutzers aus der Datenbank. Ein Benutzer soll dabei entweder anhand seiner ID oder anhand seines Benutzernamens aus der Datenbank geholt werden. Für die Auffindung anhand seines Benutzernamens schreibt Ihr eine Version, welche die JPA Query Language benutzt und eine Version basierend auf der Criteria API.
   Bücher sollen vorerst nur anhand ihrer ID aus der Datenbank geholt werden können. Benutzt dafür die Java Persistence API und die Criteria API.

F.2 Vererbung und Assoziationen mit Hibernate

Freie Universität Berlin

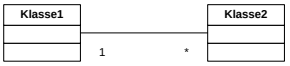
Workshop Java Webentwicklung
Vererbung und Assoziationen mit Hibernate

Ulrich Stärk

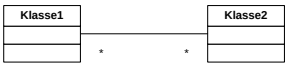
Assoziationen

- Beziehungen zwischen Objekten abbilden
  - OneToOne

  - ManyToOne aus Sicht von Klasse1, OneToMany aus Sicht von Klasse2



  - ManyToMany



Assoziationen

- Mapping mit den Annotationen
  - javax.persistence.OneToOne
    - keine weiteren Angaben notwendig
  - javax.persistence.ManyToOne
    - keine weiteren Angaben notwendig
  - javax.persistence.OneToMany
    - Angabe des mappedBy Attributs notwendig.
      - Gibt das Feld in der referenzierten Klasse an, das die Referenz hält.
        - impliziert bidirektionale Assoziation
      - Ansonsten sollte eine Assoziationstabelle erstellt werden (geschieht automatisch wenn nicht explizit angegeben).
  - javax.persistence.ManyToMany
    - Eine Seite muss das mappedBy Attribut angeben oder eine @JoinTable Annotation haben.
      - Die andere Seite darf keins der beiden haben.

Beispiele

- Mappings:

```
@Entity
public class Cat {

    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
    private int age;

    @OneToOne
    private Cat mate;

    @ManyToOne
    private Cat mother;

    @OneToMany(mappedBy="mother")
    private Set<Cat> kittens;
}
```

- Resultierendes SQL:

```
CREATE MEMORY TABLE CAT(
  ID INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH 1) NOT NULL PRIMARY KEY,
  AGE INTEGER NOT NULL,
  MATE_ID INTEGER,
  MOTHER_ID INTEGER,
  CONSTRAINT FK107B6AABAA2E6 FOREIGN KEY(MOTHER_ID) REFERENCES CAT(ID),
  CONSTRAINT FK107B6184D764 FOREIGN KEY(MATE_ID) REFERENCES CAT(ID)
)
```


Vererbung

- 3 Arten, Vererbung abzubilden:
  - Tabelle pro Klassenhierarchie
    - Alle Subklassen kommen in eine Tabelle
    - Unterscheidung der Subklassen über eine Unterscheiderspalte (enthält typischerweise den Namen der Subklasse als String)
    - großer Nachteil: keine NOT NULL Einschränkungen möglich!
  - Tabelle pro konkreter Klasse
    - jede nicht-abstrakte Klasse in der Klassenhierarchie wird durch eine eigene Tabelle repräsentiert
    - Schwierigkeiten beim Mappen auf den Obertyp
    - Schwierigkeiten bei Queries über alle Subtypen
  - Tabelle pro Unterklasse (aka Joined Subclass)
    - Eine Tabelle pro Klasse in der Klassenhierarchie
    - Tabelle der Oberklasse hält Attribute, die allen Unterklassen gemein sind
    - Tabellen der Unterklassen enthalten nur nicht-vererbte Attribute
    - Primärschlüssel der Unterklassen ist gleichzeitig Fremdschlüssel auf die Oberklasse
    - präferiert für Vererbung

Vererbung - Annotationen

- Tabelle pro Klassenhierarchie
  - „table per class hierarchy“
  - @Inheritance(strategy=InheritanceType.SINGLE_TABLE)
  - Kontrolle der Unterscheiderspalte und ihrer Werte mit
    - @DiscriminatorColumn
    - @DiscriminatorValue
- Tabelle pro konkreter Klasse
  - „table per concrete class“
  - @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
- Tabelle pro Unterklasse
  - „joined subclass“
  - @Inheritance(strategy=InheritanceType.JOINED)

Beispiel

- Mapping:

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
public class Cat {

    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;
    private int age;

    @OneToOne
    private Cat mate;

    @ManyToOne
    private Cat mother;

    @OneToMany(mappedBy="mother")
    private Set<Cat> kittens;
}
```

```
@Entity
public class DomesticCat extends Cat {

    private String name;

}
```

- Resultierendes SQL:

```
CREATE MEMORY TABLE CAT(
  ID INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH 1) NOT NULL PRIMARY KEY,
  AGE INTEGER NOT NULL,
  MATE_ID INTEGER,
  MOTHER_ID INTEGER,
  CONSTRAINT FK107B6AABAA2E6 FOREIGN KEY(MOTHER_ID) REFERENCES CAT(ID),
  CONSTRAINT FK107B6184D764 FOREIGN KEY(MATE_ID) REFERENCES CAT(ID)
)

CREATE MEMORY TABLE DOMESTICCAT(
  ID INTEGER NOT NULL PRIMARY KEY,
  NAME VARCHAR(255),
  CONSTRAINT FK24E5F1B8CF6593EA FOREIGN KEY(ID) REFERENCES CAT(ID)
)
```

Workshop Java Webentwicklung

Übung 2 – Erweiterte Hibernate Mappings

1) In dieser Übung wollen wir unser Datenmodell erweitern. Dazu schauen wir uns zuerst die Klasse `Buch` aus der letzten Übung an. Bisher speichern wir den Autor noch als String innerhalb des Buches. Wenn wir nun mehrere Bücher vom selben Autor haben, speichern wir Informationen redundant. Das wollen wir vermeiden.

   a) Erstellt eine Klasse `Autor`. Ein Autor besitzt

      - eine eindeutige automatisch generierte ID
      - einen Vornamen
      - einen Nachnamen

   b) Löscht das bisherige String-Feld `autor` der Klasse `Buch`. Erstellt in der Klasse `Buch` ein Feld `autor` und beschreibt mit der entsprechenden Annotation eine korrekte Abbildung auf die Klasse `Autor`.
   c) Erstellt in der Klasse `Autor` eine Abbildung auf die Klasse `Buch`. Die Methode `getBuecher()` soll dabei alle Bücher, die dieser Autor geschrieben hat, zurückliefern
   d) Bindet die neue Klasse `Autor` in eure Hibernate Konfiguration ein.

2) Wir wollen jetzt die Ausleihe eines Buches abbilden.

   a) Da ein Buch in der Bibliothek mehrmals vorhanden sein kann, kapseln wir ein Buch in dem Objekt `Exemplar`. Erstellt eine `Exemplar`-Klasse und verseht sie mit den nötigen Annotationen. Macht Euch selber Gedanken darüber, welche Felder ein Exemplar benötigt. Stellt auch sicher, dass ihr zu einem `Buch` Objekt alle Exemplare erhalten könnt (`getExemplare()` in der Klasse `Buch`).
   b) Erstellt eine Klasse `Ausleihe`, die Informationen darüber speichern soll, welcher Benutzer sich wann welches Exemplar ausgeliehen hat. Macht Euch auch hier wieder Gedanken darüber, welche Felder Ihr benötigt.

3) In unserer Bibliothek sollen ab sofort auch DVDs ausgeliehen werden können

   a) Erstellt eine übergeordnete Klasse `Publikation` und leitet von ihr die Klassen `Buch` und `Dvd` ab. Die übergeordnete Klasse soll dabei für alle Publikationen gültige Felder enthalten. Passt die Klassen `Exemplar` und `Ausleihe` entsprechend an.

4)
   a) Passt eure Konsolenanwendung an, um Objekte von Typ `Autor`, `Exemplar` und `Dvd` in der Datenbank zu speichern.
   b) Eure Konsolenanwendung soll eine Liste aller Bücher mit ihren Autoren und zu jedem Buch eine Liste aller Exemplare ausgeben können. Zu jedem Exemplar soll gesagt werden können, ob das Buch ausgeliehen ist oder nicht. Falls es ausgeliehen wurde soll angezeigt werden von wem und wann.

F.3 Spring und Hibernate

Freie Universität Berlin

Workshop Java Webentwicklung
Spring und Hibernate

Ulrich Stärk

Spring

- sog. leichtgewichtiger Container
  - im Gegensatz zu einem schwergewichtigen JEE Container
- bietet viel Funktionalität
  - DAO Support, Transaktionsmanagement, Webservices, Web Framework, u.v.m.
- Dependency Injection Framework
  - Eine Art Inversion of Control (IoC) Container
  - notwendige Abhängigkeiten werden in XML konfiguriert und vom Framework mit setter-Methoden aufgelöst
- Wir werden unsere DAOs und Geschäftslogik von Spring verwalten lassen
  - Transaktionsmanagement
  - Verwaltung des PersistenceContext

Konfiguration

- Schritt 1: Datenquelle anlegen

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
  <property name="driverClassName"><value>org.hsqldb.jdbcDriver</value></property>
  <property name="url">
    <value>jdbc:hsqldb:file:testdb;shutdown=true</value>
  </property>
  <property name="username"><value>sa</value></property>
  <property name="password"><value></value></property>
</bean>
```

- Schritt 2: PersistenceUnit definieren

```
<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="dataSource"/>
  <property name="jpaVendorAdapter">
    <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
      <property name="showSql" value="true" />
      <property name="generateDdl" value="true" />
      <property name="databasePlatform" value="org.hibernate.dialect.HSQLDialect" />
    </bean>
  </property>
</bean>
```

- Schritt 3: Transaktionsmanager definieren

```
<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
```


Konfiguration

- Schritt 4: Transaktionsmarkierung
  - mit Annotation @Transactional

```
<tx:annotation-driven />
```

- Schritt 5: Annotationen @PersistenceUnit und @PersistenceContext ermöglichen

```
<bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor" />
```

- Schritt 6: Übersetzung der RuntimeExceptions von JPA in Springs DataAccessExceptions

```
<bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"/>
```


Konfiguration

- Eigene Beans definieren

```
<bean id="catDao" class="inf.fu.berlin.de.model.dao.jpa.CatDaoJpa" />

<bean id="catService"
  class="inf.fu.berlin.de.service.impl.CatServiceImpl">
  <property name="catDao" ref="catDao" />
</bean>
```


Eigene DAO - Interface

- Interface CatDao

```
public interface CatDao {

    public Cat getCatById(int catId) throws DataAccessException;
    public Cat getCatByNameJpaQl(String catName) throws DataAccessException;
    public Cat getCatByNameCriteria(String catName) throws DataAccessException;
    public void saveCat(Cat cat) throws DataAccessException;
    public void reattachCat(Cat cat) throws DataAccessException;

}
```


Eigene DAO - Implementierung

```
@Repository
public class CatDaoJpa implements CatDao {

    @PersistenceContext
    private EntityManager entityManager;

    public Cat getCatById(int catId) throws DataAccessException {
        return entityManager.find(Cat.class, catId);
    }

    public Cat getCatByNameCriteria(String catName) throws DataAccessException {
        Session session = (Session)entityManager.getDelegate();
        Criteria criteria = session.createCriteria(Cat.class);
        criteria.add(Expression.eq("name", catName));
        return (Cat) criteria.uniqueResult();
    }

    public Cat getCatByNameJpaQl(String catName) throws DataAccessException {
        Query query = entityManager.createQuery("from Cat c where c.name=:name");
        query.setParameter("name", catName);
        return (Cat) query.getSingleResult();
    }

    public void saveCat(Cat cat) throws DataAccessException {
        entityManager.persist(cat);
    }

    public void reattachCat(Cat cat) throws DataAccessException {
        entityManager.merge(cat);
    }
}
```


Eigener Service - Interface

- Interface CatService

```
public interface CatService {

    public Cat getCatById(int catId) throws DataAccessException;
    public Cat getCatByNameJpaQl(String catName) throws DataAccessException;
    public Cat getCatByNameCriteria(String catName) throws DataAccessException;
    public void saveCat(Cat cat) throws DataAccessException;
    public void reattachCat(Cat cat) throws DataAccessException;

}
```

Eigener Service - Implementierung

```
@Transactional(readOnly=true)
public class CatServiceImpl implements CatService {

    private CatDao catDao;

    public Cat getCatById(int catId) throws DataAccessException {
        return catDao.getCatById(catId);
    }

    public Cat getCatByNameCriteria(String catName) throws DataAccessException {
        return catDao.getCatByNameCriteria(catName);
    }

    public Cat getCatByNameJpaQl(String catName) throws DataAccessException {
        return catDao.getCatByNameJpaQl(catName);
    }

    @Transactional(readOnly=false)
    public void saveCat(Cat cat) throws DataAccessException {
        catDao.saveCat(cat);
    }

    @Transactional(readOnly=false)
    public void reattachCat(Cat cat) throws DataAccessException {
        catDao.reattachCat(cat);
    }

    public void setCatDao(CatDao catDao) {
        this.catDao = catDao;
    }
}
```

Workshop Java Webentwicklung

Übung 3 – Integration mit Spring

Um den nächsten Schritt auf dem Weg zu unserer Webapplikation zu machen, wollen wir nun die Business Logic implementieren. Dazu werden wir von Spring gemanagte Services und DAOs schreiben.

1) Erstellt mit dem Maven Archetype ein leeres Projekt, das Hibernate und Spring kombiniert. Dabei wird ein rudimentärer `CatService` mit seiner Implementierung `CServiceImpl` sowie ein Interface `CDao` mit seiner Implementierung `CatDaoJpa` sowie die Klasse `Cat` angelegt, die Ihr als Ausgangspunkt für Eure Entwicklung benutzen könnt.

   Ruft Maven dazu wie folgt auf:
   - Wechselt in ein Verzeichnis, in dem Ihr das Projekt erstellen wollt
   - Führt dort folgenden Befehl aus:

     ```
     mvn archetype:create \
      -DarchetypeGroupId=work.shop     \
      -DarchetypeArtifactId=uebung3    \
      -DarchetypeVersion=1.0-SNAPSHOT  \
      -DgroupId=<Euer Paketname>       \
      -DartifactId=<Euer Projektname>
     ```
   - Wechselt in das Verzeichnis `<Euer Projektname>` und führt dort den Befehl `mvn eclipse:eclipse` aus.
   - Startet Eclipse und importiert das soeben erstellt Projekt in Euren Workspace.

2) Kopiert Eure Klassen `Ausleihe`, `Autor`, `Benutzer`, `Buch`, `Dvd`, `Exemplar` und `Publikation` in das Paket `model` Eures Quellverzeichnisses.

3) Erstellt Services und DAOs für folgende Anforderungen. Verwendet dabei soweit möglich Euren Code aus den vorangegangenen Übungen. Macht Euch klar, dass Instanzen von Objekten, die Eure Services zurückliefern, *detached* sind und es somit zu Exceptions kommt, wenn Ihr versucht auf nichtgeladene Referenzen zuzugreifen (Lazy Loading klappt außerhalb eines PersistenceContext nicht). Stellt also sicher, dass Ihr alle nötigen Referenzen geladen habt, bevor Ihr Objekte zurückliefert. Wo nötig, teilt Spring mit, Euren Code innerhalb einer Transaktion auszuführen. Services gehören in das Paket `service`, DAOs in das Paket `dao`. Benutzt Interfaces!

   a) Login eines Benutzers mit Benutzername und Passwort
   b) Anlegen eines neuen Benutzers
   c) Liste aller Benutzer
   d) Liste aller ausgeliehener Exemplare
   e) Liste aller Publikationen inkl. ihrer Exemplare
   f) Anlegen einer neuen Publikation
   g) Anlegen eines neuen Exemplars
   h) Ausleihe eines Exemplars
   i) Rückgabe eines Exemplars

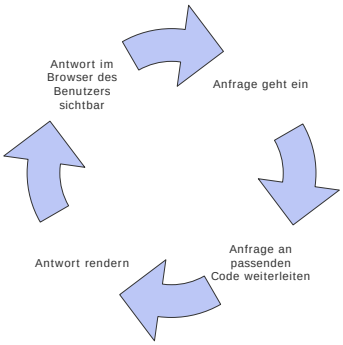
4) Testet Eure Services, indem Ihr in einer Konsolenapplikation eine Instanz von Spring erstellt, Euch den Service liefern lasst und darauf die entsprechenden Methoden aufruft. Füllt Eure Datenbank mit Daten und probiert Ausleihe, Rückgabe etc. aus. Schaut Euch dazu die Klasse `SpringTest` an, sie ist ein Beispiel dafür, wie man eine Instanz von Spring bekommt und auf ihr arbeitet.

F.4 Tapestry

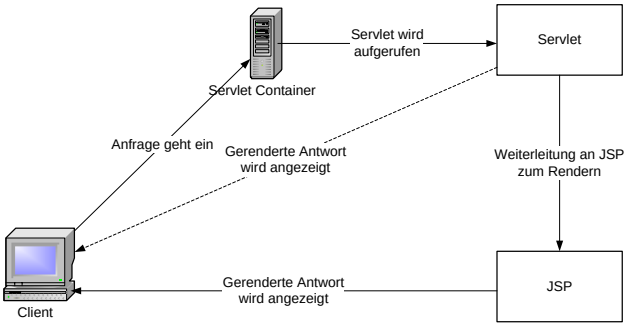
Workshop Java Webentwicklung
Tapestry

Ulrich Stärk

Webanwendungen



Servlets



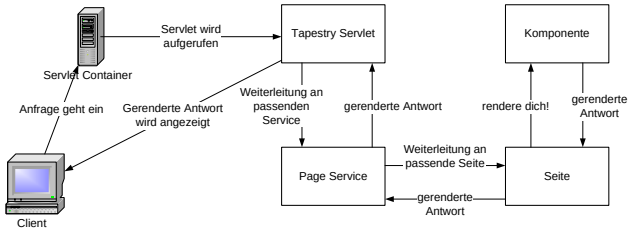
Nachteile

- Formularparameter etc. werden in HttpServletRequest gespeichert
  - und in evtl. vorhandener JSP ausgelesen
  - Entwickler braucht genaue Kenntnis von URLs und Parameternamen
    - schon kleiner Tippfehler kann gesamte Anwendung nutzlos machen
- HTTP zustandslos
  - Daten müssen in Session gespeichert werden
  - manuelle Verwaltung mühselig
- Mischung von HTML und JSP Tags
  - Arbeitsteilung zwischen Designern und Programmieren schwierig

Tapestry

- Framework für Webanwendungen
- Übernimmt mühselige Tasks
  - Session kommt nirgendwo in Tapestry vor
    - Falls benötigt, kümmert sich das Framework drum
  - Namen von URLs und Formularparameter unbedeutend
    - Framework übernimmt das
    - Programmierer muss nur sagen „baue mir hier ein Eingabefeld“ oder „hier ein Link zur Seite XYZ“
      - das geht sogar programmatisch, so dass es Compilerfehler geben würde bei falscher Benennung
- Komponentenorientiert
  - nicht prozessorientiert wie Servlet API
  - erinnert eher an die Entwicklung von Desktop Applikationen
- Aufteilung in Seiten und Komponenten
  - eine Seite besteht aus Komponenten
    - können wiederum aus Komponenten bestehen
  - eine Seite hat ein Template (HTML) und eine Seitenklasse

Anfrage bei Tapestry



Services

- page
  - Antwort durch eine Seite
- direct
  - Antwort direkt durch eine Komponente
    - meistens Links und Formulare
- asset
  - Bild, CSS, JavaScript oder anderes Objekt wird aus dem Classpath geladen
- restart
  - Session wird verworfen und Startseite angezeigt

Seiten

- eine Seite besteht aus 2 oder 3 Teilen
  - Template (z.B. Home.html)
    - enthält HTML und bindet mit speziellen Attributen (jwcid) Komponenten ein
    - kann implizite Komponentendefinitionen enthalten
  - Seitenklasse (z.B. Home.java)
    - leitet von org.apache.tapestry.html.BasePage ab
    - in aller Regel abstrakt
    - speichert Seitenspezifische Objekte und stellt listener-Methoden zur Verfügung
    - kann Komponentendefinitionen enthalten
  - Seitenspezifikation (z.B. Home.page, optional)
    - enthält Verweis auf Seitenklasse
    - enthält Komponentendefinitionen
    - falls nicht vorhanden wird die Seitenklasse im Paket, das mit org.apache.tapestry.page-class-packages angegeben wurde, gesucht

Komponenten

- wie Seiten
  - Komponentenklasse leitet aber von org.apache.tapestry.BaseComponent ab
  - Komponentenspezifikation endet auf .jwc, z.B. Login.jwc
    - kann wie bei Seiten auch weggelassen werden
    - Komponentenklassen werden im Paket, das mit org.apache.tapestry.component-class-packages angegeben wurde, gesucht
  - benötigen nicht unbedingt ein Template

Beispielkomponenten

- Insert
  - Wert einer Seiteneigenschaft einfügen
  - Zeit
    - mit Ognl wird beim Rendern in der Seitenklasse nach der Eigenschaft currentTime gesucht und diese in der Seite eingefügt
      - impliziert ein getCurrentTime() in der Seitenklasse
- PageLink
  - Link (<a href…>) zu einer anderen Seite erstellen
  - Foo
    - Erzeugt einen Link zum Page Service mit dem Parameter page=Home
- Shell
  - Rendert das <html> Element um die Seite
  - <html jwcid="@Shell" title="Titel der Webseite">…</html>
- Body
  - Rendert das <body> Element und wird häufig benötigt
  - <body jwcid="@Body">…</body>

Seitenklasse

- Quellcode

```
public abstract class Home extends BasePage {

    public Date getCurrentTime() {
        return new Date();
    }
}
```

- abstrakt, weil Tapestry die Seitenklasse automatisch erweitern kann
  - z.B. Definition der Komponenten mit Annotationen

```
public abstract class Home extends BasePage {

    @Component(bindings={"value=currentTime"})
    public abstract Insert getMyInsert();

    @Component(bindings={"page=literal:Home"})
    public abstract PageLink getMyPageLink();

    public Date getCurrentTime() {
        return new Date();
    }
}
```

  - und dann Foo bzw. Foo

Listener-Methoden

- sind der eigentliche Kern der Applikation
- werden aufgerufen um wirkliche Dinge zu tun
  - von Formularen oder Links
- können Parameter haben
  - keinen, IRequestCycle cycle, einen oder mehrere, einen oder mehrere und IRequestCycle cycle
- können verschiedene Rückgabetypen haben
  - keinen
    - aktuelle Seite wird wieder angezeigt
  - String
    - Rückgabe wird als Seitenname interpretiert und diese wird aktiviert
  - IPage
    - Diese Seite wird aktiviert
  - ILink
    - ein Redirect zu dieser URL wird durchgeführt

Beispiele

- Template

```
<p>
  <a jwcid="@DirectLink" listener="listener:onClick" href="#">
    Home aufrufen mit listener-Methode
  </a>
</p>
<p>
  <a jwcid="@DirectLink" listener="listener:onClickString" parameters="literal:Test test 1 2 3" href="#">
    Link mit String parameter
  </a>
</p>
<p>
  <a jwcid="@DirectLink" listener="listener:onClickDate" parameters="ognl:currentTime" href="#">
    Link mit Date parameter
  </a>
</p>
```

- Seitenklasse

```
public void onClickString(String parameter) {
    System.out.println(parameter);
}

public void onClickDate(Date date) {
    System.out.println(date);
}

public String onClick() {
    return "Home";
}
```


Life: Tapestry einrichten

- pom.xml anpassen
- web.xml anpassen

Workshop Java Webentwicklung

Übung 4 – Erste Schritte mit Tapestry

In dieser ersten Übung zu Tapestry wollen wir eine Willkommensseite erstellen und eine kleine zweite Seite, auf die wir verweisen.

1)
   a) Entfernt die Kommentare in Eurer `pom.xml`, die bisher dafür gesorgt haben, dass die Tapestry Abhängigkeiten nicht geladen werden. Führt anschließend in Eurem Projektverzeichnis ein `mvn compile` aus, um die Abhängigkeiten zu laden und das Projekt neu zu erstellen.

   b) Entfernt die Kommentare in Eurer web.xml, um das Tapestry Servlet im Servletcontainer zu laden.

   c) Erstellt im Paket `view.pages` die Klasse `Home`, die von der Klasse `org.apache.tapestry.BasePage` abgeleitet wird.

   d) Erstellt im Verzeichnis `webapp` eine Datei `Home.html`. Diese Seite soll einen statischen Willkommenstext sowie das aktuelle Datum und die aktuelle Uhrzeit anzeigen. Passt die Seitenklasse entsprechend an. Die Komponente zum Einfügen soll dabei implizit im HTML Quelltext definiert werden.

2)
   a) Erstellt eine weitere Seite, die auch dynamischen Inhalt anzeigen soll. Diesmal sollen die benötigten Komponenten aber mittels Annotationen in der Seitenklasse definiert werden und nur mit ihrer ID im HTML Quelltext referenziert werden.

   b) Erweitert die Startseite um einen einfachen Link, der auf diese Seite verweist.

3) Erweitert die zweite Seite um einen Link, der in der dazugehörigen Seitenklasse eine Methode ohne Parameter aufruft.

4) Erweitert die zweite Seite um einen Link, der in der dazugehörigen Seitenklasse eine Methode mit Parameter aufruft.

F.5 Tapestry - Formulare

Freie Universität Berlin

Workshop Java Webentwicklung
Tapestry - Formulare

Ulrich Stärk

Formulare

- übergeordnete Komponente @Form rendert <form> Element
  - darunter Komponenten für die verschiedensten Formularfelder
    - @Hidden, @Textfield, @Checkbox, @Select, @Upload, @Button, @Submit, @Radio, @RadioGroup, usw.
  - muss innerhalb einer @Body Komponente stehen
  - für jede Input-Komponente sollte ein @FieldLabel existieren
    - das Label markiert das Feld bei einem Fehler als fehlerhaft
    - außerdem stellt es den Namen des Eingabefeldes dar, der aus einem Message-Katalog stammen kann (Lokalisierung, kommt später)
  - fast alle Eingabe-Komponenten haben das Attribut validators und einige noch das Attribut translator
    - validators ist eine Liste von Validatoren, die auf die Eingabe angewandt werden sollen
    - translator übersetzt automatisch den Wert in ein anderes Format als String

@TextField

- Rendert ein Eingabefeld
  - Attribut value verweist auf eine Property in der Seitenklasse, in der der Wert gespeichert wird

```
<p>
  <label jwcid="@FieldLabel" field="component:name">Name</label>
  <input
    jwcid="name@TextField"
    value="ognl:name"
    validators="validators:required"
    displayName="Name"
  />
</p>
<p>
  <label jwcid="@FieldLabel" field="component:alter">Alter</label>
  <input
    jwcid="alter@TextField"
    value="ognl:age"
    validators="validators:required,min=0"
    translator="translator:number,omitZero=true"
    displayName="Alter"
  />
</p>
```


@Form

- <form jwcid="myForm@Form" delegate="bean:delegate" success="listener:onSuccess" clientValidationEnabled="true">
  - rendert Formular mit der ID myForm (sollte gesetzt werden)
  - Validierung wird an die mit dem Parameter delegate angegebene Bean „delegate“ abgegeben
  - bei erfolgreicher Validierung wird die im Parameter success angegebene Listener-Methode aufgerufen
  - clientValidationEnabled="true" führt clientseitige Validierung durch
- Seitenklasse

```
@Bean
public abstract ValidationDelegate getDelegate();

public void onSuccess() {
    System.out.println("success");
}
```


@Delegator

- @Delegator delegiert das Rendern einfach an ein Objekt, das das IRender Interface implementiert
  - ist also eine Art Wrapper-Komponente
- wird z.B. benutzt, um Fehlermeldungen der ValidationDelegate Bean anzuzeigen
  - dort gibt es eine Methode getFirstError(), die ein IRender Objekt zurückliefert, das den ersten Fehler rendern kann
  - die Methode getErrorRenderers() liefert eine Liste mit IRender Objekten, für jedes Fehlerhafte Feld eins
    - kann mit der @For Komponente durchlaufen werden
- Anzeige des ersten Fehlers des ValidationDelegate

```
<p>
<font color="red">
<span jwcid="@Delegator" delegate="ognl:delegate.firstError">Fehler</span>
</font>
</p>
```


Alles zusammen - Template

```
<html jwcid="@Shell" title="Formular">
<body jwcid="@Body">
<p>
  <font color="red">
    <span jwcid="@Delegator" delegate="ognl:delegate.firstError">Fehler</span>
  </font>
</p>
<form
  jwcid="myForm@Form"
  delegate="bean:delegate"
  success="listener:onSuccess"
  clientValidationEnabled="true"
>
<p>
  <label jwcid="@FieldLabel" field="component:name">Name</label>
  <input
    jwcid="name@TextField"
    value="ognl:name"
    validators="validators:required"
    displayName="Name"
  />
</p>
<p>
  <label jwcid="@FieldLabel" field="component:alter">Alter</label>
  <input
    jwcid="alter@TextField"
    value="ognl:age"
    validators="validators:required,min=0"
    translator="translator:number,omitZero=true"
    displayName="Alter"
  />
</p>
<p><input jwcid="@Submit" /></p>
</form></body>
</html>
```


Alles zusammen - Seitenklasse

```
public abstract class Home extends BasePage {

    @Bean
    public abstract ValidationDelegate getDelegate();

    public void onSuccess() {
        System.out.println("success");
        Cat cat = new Cat();
        cat.setName(getName());
        cat.setAge(getAge());
    }

    public abstract String getName();
    public abstract int getAge();
}
```


Ausflug: Lebenszyklus von Seiten

- Finde passende Seite und verkuppele sie mit dem Request
  - Persisten Properties wurden gesetzt
  - PageAttachListener
    - weitere Properties setzen
- Aktiviere Seite (Wähle Seite zum Rendern der Antwort)
  - PageValidateListener
    - Zugriff prüfen
    - PageRedirectException
- Beginne Rendern
  - PageBeginRenderListener
    - letzte Chance um Properties zu setzen
- Ende Rendern
  - PageEndRenderListener
- Entkoppele Seite vom Request und zurück in den Pool
  - PageDetachListener

Submit von Formularen

- Beim Submit eines Formulars wird dieser Zyklus zwei bzw. drei mal durchlaufen
  - erstes Rendern erzeugt Seite und darin Formular
    - Namen und IDs der Eingabefelder sind dynamisch und werden nicht zwischengespeichert
    - wie weiß das Formular also, welcher Feldname zu welcher Property gehört
  - Rewinding rendert erneut
    - aber nur das Formular
    - um Namen und IDs der Eingabefelder erneut zu erzeugen und damit die Zuordnung zu den Properties zu kennen
    - anschließend werden diese gesetzt
  - wenn als Ergebnis des Submits wieder die gleiche Seite angezeigt wird, erfolgt ein drittes Rendern

@For

- mit der @For Komponente kann eine Collection oder ein Array durchlaufen werden
- …
  - mit dem source Parameter wird die Quelle für die Iteration festgelegt
    - Iterator, Collection, Array, Object (wird als Collection mit 1 Element behandelt) oder null (Iteration wird nicht durchgeführt)
  - in der mit value angegebenen Eigenschaft wird der Wert der aktuellen Iteration gespeichert
    - damit könnte z.B. eine @Insert Komponente eine Liste von Objekten auf einer Webseite darstellen
      - innerhalb der @For Komponente

Spring integrieren

- geschieht automatisch durch die tapestry-spring Abhängigkeit in der pom.xml
- tapestry-spring-1.0.0.jar hat ein META-INF Verzeichnis, in dem es eine hivemodule.xml gibt
  - konfiguriert Apache Hivemind, das von Tapestry genutzt wird
  - zusätzliche hivemodule.xml erweitern einfach Hivemind
  - alle Konfiguration für Tapestry mit Spring ist in der hivemodule.xml schon drin
- mit der @InjectObject Annotation lassen sich Hivemind Objekte in der Seitenklasse einfügen
  - mit @InjectObject("spring:meinService") lässt sich die Spring Bean mit dem Namen meinService einfügen
    - Anfrage geht an einen Hivemind service, der mit dem spring: Prefix verbunden ist
    - dieser wertet das hinter spring: aus und liefert die entsprechende Spring Bean

Spring integrieren: Seitenklasse

```
public abstract class Home extends BasePage {

    @Bean
    public abstract ValidationDelegate getDelegate();

    @InjectObject("spring:catService")
    public abstract CatService getCatService();

    public void onSuccess() {
        System.out.println("success");
        Cat cat = new Cat();
        cat.setName(getName());
        cat.setAge(getAge());
        getCatService().saveCat(cat);
    }

    public abstract String getName();
    public abstract int getAge();
}
```


Life: Spring integrieren

- web.xml anpassen
- hivemodule.xml anschauen

Workshop Java Webentwicklung

Übung 5 – Formulare mit Tapestry

Wir wollen nun unsere Mini-Anwendung um eine Registrierungsseite für neue Benutzer erweitern.

1)

a) Bindet Spring in Eure `web.xml` ein. Entfernt dazu die Kommentare um die Spring Definitionen in der web.xml. Dies wird Spring beim Starten des Servletcontainers laden.

2)

a) Erstellt eine Formular-Seite, auf der sich ein Benutzer registrieren kann. Diese Seite soll alle Informationen abfragen, die zum Anlegen eines neuen `Benutzer`-Objekts notwendig sind.

b) Verseht das Formular mit sinnvollen Validierungsvorschriften und zeigt dem Benutzer bei Problemen mit der Eingabe die Fehler an.

c) Speichert das neue `Benutzer`-Objekt bei erfolgreicher Validierung mittels des entsprechenden Spring-Services in der Datenbank.

d) Erweitert die Startseite um einen Link, der auf diese Seite verweist.

3)

a) Erstellt eine weitere Seite, mit der ein Benutzer seine eigenen Daten verändern kann, oder erweitert die vorhandene Registrierungsseite um entsprechende Parameter um zu unterscheiden, ob es sich um eine Neuanlage oder eine Veränderung eines bereits vorhandenen Benutzers handelt.

b) Erweitert die Startseite um einen Link, der auf diese Seite verweist. Evtl. müsst Ihr in diesem Link der Seite die entsprechenden Parameter mitgeben. Oder Ihr verwendet einen DirectLink und liefert in der listener-Methode die entsprechende Seite zurück. Vorher könnt Ihr auf das Seitenobjekt Methoden zum Setzen von Parametern aufrufen.

F.6 Tapestry - ASOs

Freie Universität Berlin

Workshop Java Webentwicklung
Tapestry - ASOs

Ulrich Stärk

ASOs

- Application State Objects (ASOs) speichern Informationen über mehrere Seitenaufrufe hinweg
  - Informationen über den aktuellen Besucher während seiner Sitzung
  - Singleton für die gesamte Applikation
- zwei Arten von ASOs:
  - Session Scope
    - Lebenszyklus des ASO ist an die Session gebunden
    - Name des Default Session-Scoped ASO: visit
  - Application Scope
    - ASO existiert während der gesamten Laufzeit der Applikation
    - Name des Default Application-Scoped ASO: global
  - Default ASOs sind Instanzen von HashMap

Eigenes ASO

- Für die Erstellung eines ASO muss die hivemodule.xml angepasst werden

```
<contribution configuration-id="tapestry.state.ApplicationObjects">
    <state-object name="visit-object" scope="session">
        <create-instance class="workshop.view.VisitObject"/>
    </state-object>
</contribution>
```

- Eine Instanz von workshop.view.VisitObject wird in der Session gespeichert und ist unter dem Namen visit-object verfügbar
  - Parameterloser Konstruktor muss verfügbar sein
  - Einfügen in die Seitenklasse mit @InjectState Annotation
    - @InjectState public abstract VisitObject getVisitObject();
    - @InjectState("visit-object") ...
- Äquivalent für Application-Scoped ASOs
  - dann halt mit scope="application" statt session

Erzeugung eines ASOs

- Ein ASO wird erzeugt, sobald das erste Mal darauf zugegriffen wird
  - auch wenn gar nichts drin steht oder gar nichts reingeschrieben wird
  - typischerweise wird in einem ASO gespeichert, ob ein User eingeloggt ist oder nicht
    - Prüfung auf Seiten, die keinen eingeloggten User erfordern würde eine Session erzeugen
- erst einmal prüfen, ob Instanz des ASOs vorhanden
  - state: binding prefix in Verbindung mit @If Komponente
  - @InjectStateFlag Annotation in der Seitenklasse
    - gleicher Syntax wie @InjectState, allerdings boolean Rückgabewert

Persistent Properties

- Properties einer Seite können bei Bedarf auch in der Session oder auf der Clientseite gespeichert werden
  - die Annotation @Persist speichert den Wert einer Property in der Session
  - ist dann auch beim nächsten Seitenaufruf vorhanden

Komponenten

- Seiten in Tapestry bestehen aus Komponenten
- eigene Komponenten möglich
  - bestehen aus Komponentenklasse und optional einem Template
  - Konfigurationsparameter org.apache.tapestry.component-class-packages gibt an, in welchem Paket sich eigene Komponenten befinden
  - leiten von BaseComponent ab
- gängige Komponente: Border
  - Erstellung eines einheitlichen Layouts für alle Seiten
  - Seitennavigation, Anordnung etc. in Border-Komponente
  - wird in allen Seiten eingebunden
- Komponenten können wiederum andere Komponenten enthalten
  - bspw. Login-Komponente zur Verwendung auf mehreren Seiten
    - mit @Formular, mehreren @FieldLabel und @TextField, usw.

@Border

```
<html jwcid="@Shell" title="Startseite">
  <body jwcid="@Body">
    <div jwcid="@RenderBody">
      Hier wird der Inhalt erscheinen
    </div>
  </body>
</html>
```

- @Shell und @Body bekannt
- in @RenderBody wird später der Inhalt der Komponenten, die von @Border umschlossen werden, gerendert
- Komponentenklasse genau wie Seitenklasse
  - leitet allerdings von BaseComponent ab
- Einbindung z.B. in Home.html

```
<html>
  <head><title>Startseite</title></head>
  <body jwcid="$content$">
    <span jwcid="@Border">
      Seitenspezifischer Inhalt
    </span>
  </body>
</html>
```


Lokalisierung

- Mit Tapestry ist es einfach möglich, Inhalte Benutzern aus verschiedenen Sprachräumen zur Verfügung zu stellen
  - Seite übersetzen und unter z.B. Home_en.html oder Home_fr.html abspeichern genügt schon als erster Schritt
- Einzufügende Textschnipsel können in lokalisierte Component Message Catalogs abgelegt werden
  - z.B. Home.properties, Home_en.properties, Home_fr.properties
  - key=value Paare
  - werden mit dem message: binding prefix referenziert
    - z.B. displayName="message:name" für das Namensfeld aus dem Formularbeispiel
  -
    - Abkürzung

Lokalisierung

- mit getLocale() kann auf das eingestellte Gebietsschema zugegriffen werden
  - z.B. ein Datum kann dann mit dem entsprechenden DateFormat zurückgegeben werden
- mit getEngine().setLocale() kann das Gebietsschema eingestellt werden
  - gilt aber erst für den nächsten Seitenaufruf
- mit der Eigenschaft org.apache.tapestry.accepted-locales wird eine Liste der akzeptierten Gebietsschemen festgelegt
  - kommasepariert, z.B. en,de,fr
    - 1. Eintrag ist default
  - Performancegründe
  - kann mit der @InjectMeta Annotation ausgelesen werden

Workshop Java Webentwicklung

Übung 6 – ASOs, Komponenten und L10N mit Tapestry

Unsere Anwendung soll natürlich auch den Login eines Benutzers unterstützen. Damit wir uns auch merken können, mit welchem Benutzer wir uns gerade unterhalten, benötigen wir ein Objekt, das mit der Session assoziiert wird und Informationen über den aktuellen Benutzer speichert.

1)
   a) Erstellt ein Application State Object (ASO), das ein `Benutzer`-Objekt speichert, das einen eingeloggten Benutzer repräsentiert.
   b) Registriert das ASO in der Hivemind Registry `hivemodule.xml`.

Natürlich benötigen wir auch einen Login-Dialog, der es einem Benutzer ermöglicht, sich anzumelden.

2)
   a) Erstellt eine Login-Komponente, die es einem Benutzer ermöglicht, sich anzumelden. Diese Komponente besteht im Wesentlichen aus einem Formular. Bei erfolgreicher Anmeldung soll das `Benutzer`-Objekt in dem in Aufgabe 1 erstellten ASO gespeichert werden. Das ASO soll aber erst erstellt werden, wenn es wirklich gebraucht wird, also wenn die Anmeldung erfolgreich war.
   b) Bindet Eure Login-Komponente auf der Startseite ein.

3) Eure in Übung 5 erstellte Seite zur Veränderung eines Benutzerobjekts soll nun so modifiziert werden, dass sie auf dem `Benutzer`-Objekt eines angemeldeten Benutzers arbeitet. Stellt außerdem sicher, dass nur ein angemeldeter Benutzer auf diese Seite Zugriff hat (`PageValidateListener`). Überhaupt soll der Link zu dieser Seite nur angemeldeten Benutzern angezeigt werden (achtet aber darauf, dass Ihr nicht versehentlich eine Instanz des ASOs aus Teil 1 erstellt (`state` Binding Präfix)).

Wir wollen unsere Anwendung nun auch englischsprachigen Benutzern zur Verfügung stellen.

4)
   a) Speichert Eure Startseite in 3 Versionen ab: Home_de.html für die deutsche Übersetzung, Home_en.html für die englische Übersetzung und Home.html für die Standardversion. Übersetz Eure Startseite entsprechend.
   b) Teilt Tapestry mit, dass nur die Gebietsschemen de und en akzeptiert werden.
   c) Testet die Übersetzungen, indem Ihr in Eurem Browser die Standardsprache ändert.

5)
   a) Erstellt auf der Startseite ein Dropdown Feld (`PropertySelection`), mit dem der Benutzer aus den akzeptierten Gebietsschemen auswählen kann. Erstellt dazu ein `LocaleSelectionModel`, das die akzeptierten Gebietsschemen enthält und der `PropertySelection` übergeben wird. Das gewählte Gebietsschema soll anschließend für die gesamte Applikation gesetzt werden. Denkt daran, dass diese

Änderung für die aktuelle Anfrage noch nicht gilt und Ihr evtl. den Benutzer auf eine neue (oder die gleiche) Seite umleiten solltet.

6) Übersetzt alle anderen Seiten und Ressourcen entsprechend Teil 4.