TU Clausthal
Institut für Informatik

Bachelorarbeit

Unterstützungsverfahren für Tutoren bei der Online-Abgabe von Übungsaufgaben

Sven Strickroth

26. Juli 2009

Bachelorarbeit

Unterstützungsverfahren für Tutoren bei der Online-Abgabe von Übungsaufgaben

eingereicht bei

Betreuender Prüfer: Prof. Dr. Niels Pinkwart
Zweitgutachter: Prof. Dr. Jörg P. Müller
Institut für Informatik
Abteilung für Wirtschaftsinformatik
Fakultät für Mathematik/Informatik und Maschinenbau
Technische Universität Clausthal

von
Sven Strickroth
Studienrichtung: Informatik, B. Sc.
Matrikelnummer: 352752

Datum: 26. Juli 2009

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange (Zeichenkodierung)
CA	Certification Authority
CGI	Common Gateway Interface
CSS	Cascading Style Sheets
DAO	Data Access Object
DBMS	Database Management System
GPL	GNU General Public License (Open Source Lizenz)
GUI	Graphical User Interface
HQL	Hibernate Query Language
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
JDBC	Java Database Connectivity (siehe [1])
JNDI	Java Naming and Directory Interface (siehe [2])
JSP	JavaServer Pages (Spezifikation in [3])
LDAP	Lightweight Directory Access Protocol
LZMA	Lempel-Ziv-Markow-Algorithmus
MITM	Man-In-The-Middle

MVC	Model-View-Controller
NCD	Normalized Compression Distance
NID	Normalized Information Distance
ORM	Object Relational Mapping
PHP	PHP Hypertext Preprocessor (rekursives Akronym)
POJO	Plain Old Java Object
RMI	Remote Method Invocation
SQL	Structured Query Language
SSL	Secure Sockets Layer
STDIN	Standard Input
STDOUT	Standard Output
TLS	Transport Layer Security
UML	Unified Modeling Language
URL	Uniform Resource Locator
VM	Virtual Machine
WSDL	Web Services Description Language
XSS	Cross-Site-Scripting

1 Einleitung

1.1 Motivation

Das selbstständige Bearbeiten von Hausaufgaben ist ein essentieller Teil der Lehre. Ohne praktische Übungen lässt sich keine Kompetenz in einem Fachgebiet wie z. B. dem Programmieren erwerben. Insbesondere bei Bachelor- und Masterstudiengängen sind diese fest in die Vorlesungen integriert und dienen meist als Vorleistung für eine Modulprüfung der jeweiligen Vorlesung.

Für die Bewertung der studentischen Lösungen werden momentan vornehmlich zwei Verfahren eingesetzt:

- Bearbeitete Hausaufgaben werden per E-Mail an die Betreuer der Vorlesungen gesandt, welche die Einsendungen dann kurz sichten und schließlich zur Korrektur sowie Bewertung an die jeweiligen Tutoren weiterleiten oder
- Studenten stellen ihre Lösungen in einer Übung persönlich einem Tutor vor.

Insbesondere bei großen Veranstaltungen ergibt sich bei diesen Abgabeverfahren eine Reihe von Schwierigkeiten:

Bei der E-Mail-Abgabe müssen die Lösungen sortiert, gespeichert und entpackt bzw. weitergeleitet werden, bis sie schließlich in verwertbarer Form beim zuständigen Tutor für die Bewertung vorliegen. Soll neben der Sichtung noch ein Funktionstest erfolgen, muss jede Abgabe einzeln kompiliert und getestet werden.

Bei beiden Verfahren ist die Erkennung von Duplikaten sehr aufwändig bis unmöglich, da ein Tutor im Normalfall lediglich eine (kleine) Auswahl der Einsendungen sieht. Duplikate können hierbei nur innerhalb dieser Teilmenge erkannt werden, nicht aber bei Abgaben, die von anderen Tutoren bearbeitet werden.

Nur bei der persönlichen Abgabe erhalten die Studenten direkt Rückmeldung über erhaltene Punkte. Bei der E-Mail-Abgabe sind die Punktestände ohne größeren Aufwand nur über einen anderen Kanal wie z. B. eine Webseite oder einen Aushang ersichtlich. Um eine Gesamtübersicht zu erhalten, müssen verschiedene Punktelisten zusammengeführt werden. Ein Gesamtsystem könnte alle drei Vorgänge sowohl für die Betreuer einer Vorlesung als auch für die Studenten vereinfachen.

Ziel dieser Arbeit ist es, den Abgabeprozess durch ein neu zu entwickelndes System zu unterstützen.

1.2 Aufgabenstellung

Gegenstand dieser Arbeit ist die Konzeption und Entwicklung eines internetbasierten Unterstützungssystems für Betreuer und Tutoren zur Kontrolle sowie Bewertung von Programmieraufgaben, welche in der Programmiersprache Java verfasst sind.

Hierzu gehört die Modellierung einer Datenstruktur bzw. eines Domänenmodells, die Entwicklung einer zu den Anforderungen passenden Architektur und die Auswahl von unterstützenden Technologien. Dazu zählt insbesondere die Auswahl, Implementierung/Integration und Evaluation möglicher Algorithmen für die Duplikat-Erkennung.

2 Analyse

2.1 Funktionale Anforderungen

In diesem Kapitel werden die funktionalen Anforderungen an das zu entwickelnde System vorgestellt. Funktionale Anforderungen beschreiben die Fähigkeiten eines Systems, die ein Anwender erwartet, um mit Hilfe des Systems die in der Motivation genannten Probleme zu lösen. Die Anforderungen werden aus den zu unterstützenden Geschäftsprozessen und den Ablaufbeschreibungen zur Nutzung des Systems abgeleitet (vgl. [4, Kapitel 5.3.7.5.2]).

Abbildung 2.1 (auf der folgenden Seite) veranschaulicht die Anforderungen mittels Unified Modeling Language (UML) Use-Case-Diagramm. Die Begriffe/Anwendungsfälle des Diagramms sind im Folgenden hervorgehoben und werden nun in Auftrittsreihenfolge erläutert.

Alle Benutzer müssen sich zunächst am System *registrieren/anmelden*. Dies ist notwendig, damit die Anwender sich später eindeutig authentifizieren können. Damit wird eine Zuordnung der Aktionen zu einem Benutzer möglich und abhängig von ihrer Autorisation können weitere Use-Cases ausgeführt werden. Angestrebt wird die Integration der bestehenden Authentifikations-Infrastruktur der Universität, um Single-Sign-On zu ermöglichen.

Ist ein Benutzer *SuperUser*, so kann er *Veranstaltungen anlegen* und Betreuer für diese festlegen.

Sobald eine Veranstaltung angelegt ist, so sollen hierfür Anmeldungen möglich sein. Diese Eintragung ermöglicht ihnen, die für die jeweilige Veranstaltung eingestellten Aufgaben und, sofern eine Berechtigung besteht, zusätzlich auch die Liste der angemeldeten Benutzer einzusehen.

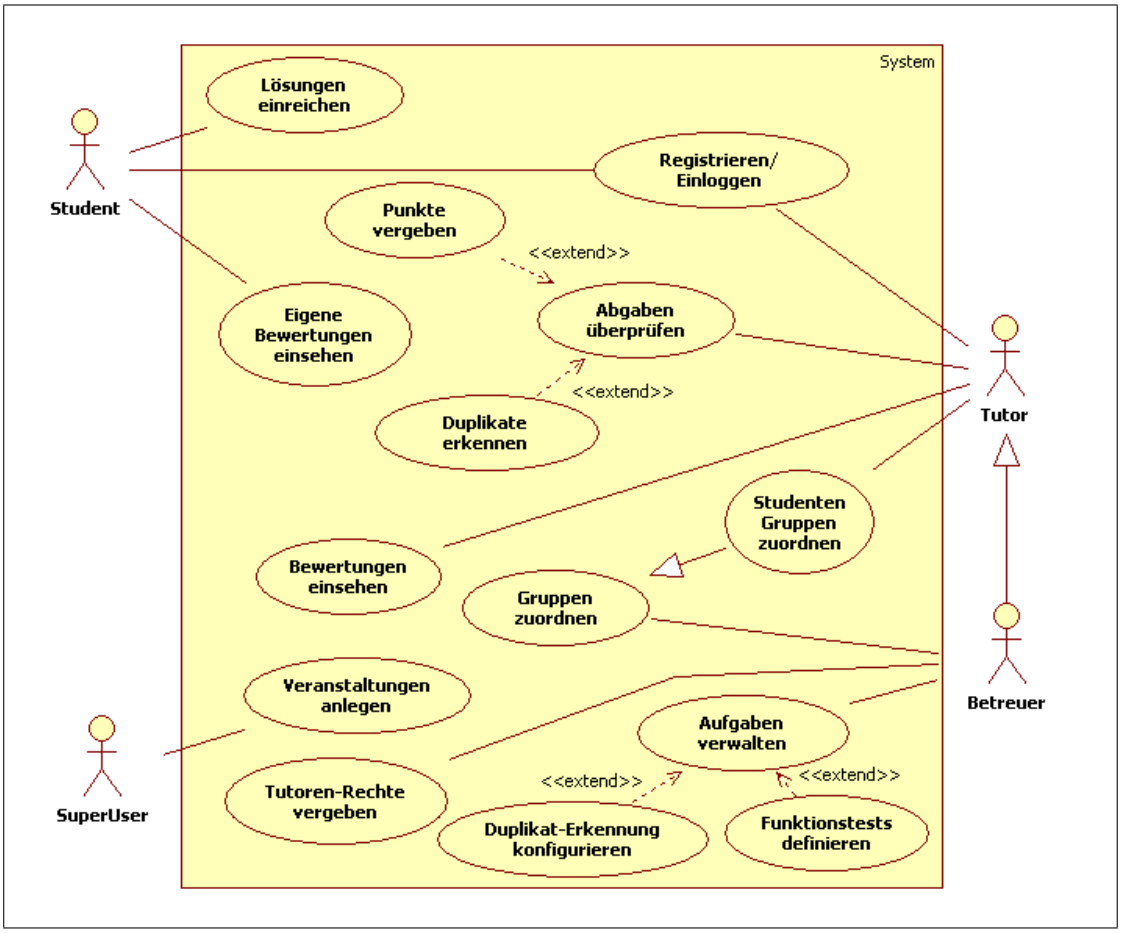


Abbildung 2.1: Anforderungen als Use-Case-Diagramm

Betreuer können aus den angemeldeten Benutzern ausgewählte Studenten zu Tutoren befördern (*Tutoren-Rechte vergeben*). Darüber hinaus können Betreuer Gruppen anlegen und diesen Gruppen Tutoren zuordnen (*Gruppen verwalten*). Diese Gruppen dienen der Festlegung, welche Tutoren vornehmlich für eingeteilte Studenten verantwortlich sind. Die Einteilung der Studenten soll durch die Tutoren vorgenommen werden (*Studenten Gruppen zuordnen*).

Betreuer sollen in der Lage sein, für ihre Veranstaltungen Aufgaben zu verwalten und insbesondere auch einstellen zu können (*Aufgaben verwalten*). Dies beinhaltet die folgenden Einstellungen:

- Titel der Aufgabe
- Aufgabenstellung
- Anzahl der maximalen Punkte, die für diese Aufgabe vorgesehen sind
- Zeitpunkt, ab wann eine Aufgabe für Studenten sichtbar sein soll
- Zeitpunkt, bis zu dem eine Abgabe bzw. eine Korrektur von Lösungen möglich sein soll
- Zeitpunkt, ab wann die vergebenen Punkte für Studenten sichtbar werden sollen

Optional soll es nach dem Anlegen einer Aufgabe möglich sein, speziell für jede Aufgabe einen automatischen Funktionstest einzurichten (*Funktionstests definieren*) und die *Duplikat-Erkennung zu konfigurieren*.

Nachdem Aufgaben angelegt sind, können angemeldete Studenten innerhalb der Abgabefrist ihre *Lösungen einreichen*. Direkt nach dem Upload der Lösung erhält der Student eine Rückmeldung, ob seine Abgabe kompiliert (dies inkludiert eine Syntaxprüfung und Abhängigkeitsprüfung; fehlt eine benötigte Datei, so ist eine Kompilierung nicht möglich). Sofern der Student es möchte, kann er korrigierte Lösungen einreichen ([5] beschreibt die Vorteile dieses Vorgehens).

Bereits vor Ablauf der Abgabefrist können Tutoren und Betreuer die eingereichten aber noch veränderbaren Lösungen einsehen (inkl. der Kompilierungs- und optionalen Testergebnisse). Direkt nach Ablauf der Frist soll das System Veränderungen an den Einsendungen nicht mehr zulassen und die konfigurierten Duplikat-Erkennungen starten.

Nach Abschluss der Duplikat-Erkennung sollen Betreuer als auch Tutoren die Ergebnisse übersichtlich im System präsentiert bekommen. Mit Hilfe dieser Daten können Tutoren die *Abgaben überprüfen*, *Duplikate erkennen* und schließlich direkt *Punkte vergeben*.

Zu jeder Bewertung durch die Tutoren soll gespeichert werden, wer die Punkte-Vergabe vorgenommen hat. Für jede Veranstaltung soll den Tutoren und dem Betreuer eine detaillierte Auswertung präsentiert werden (*Bewertungen einsehen*). Die Studenten sollen nach der Freigabe ihre erzielten Punkte für die einzelnen Aufgaben und auch aufsummiert für die besuchten Veranstaltungen einsehen können.

2.2 Domänenmodell/Datenmodell

Abbildung 2.2 (auf der nächsten Seite) zeigt das vollständige UML Datenmodell des Systems als Klassendiagramm.

Auf der linken Seite des Modells (als Paket dargestellt) befinden sich alle Daten, die sich direkt auf die Aufgaben (*Task*s) beziehen. Das mittlere Paket beinhaltet alle Klassen, die mit einer Abgabe (*Submission*) zusammenhängen. Das Paket unten auf der rechten Seite kapselt alle Klassen, die sich direkt auf eine Vorlesung (*Lecture*) beziehen. Auf der rechten Seite weiter oben befindet sich das Paket, welches die Benutzerdaten enthält. In diesem Abschnitt werden die Zusammenhänge genauer beschrieben:

Die *User* sind die registrierten Benutzer des Systems. Für jeden Anwender werden zur Identifikation die persönlichen Daten (Vorname, Nachname und E-Mail-Adresse) gespeichert. Ist ein Benutzer *Student*, so wird er innerhalb des Systems auch als solcher angesehen (im objektorientierten Sinne als Spezialisierung eines *User*s). Als Erkennungsmerkmal kann zusätzlich die Matrikelnummer hinterlegt werden. Unabhängig davon besteht die Möglichkeit, einem Benutzer global den *SuperUser*-Status zu verleihen. Dies ermöglicht ihm, administrative Aufgaben durchzuführen (z. B. Anlegen einer Veranstaltung).

Die *Lecture*s repräsentieren die durch das System zu unterstützenden bzw. am System teilnehmenden Veranstaltungen/Vorlesungen. Zu jeder Veranstaltung können benannte Gruppen (*group*s) zur Strukturierung der Teilnehmer hinzugefügt werden.

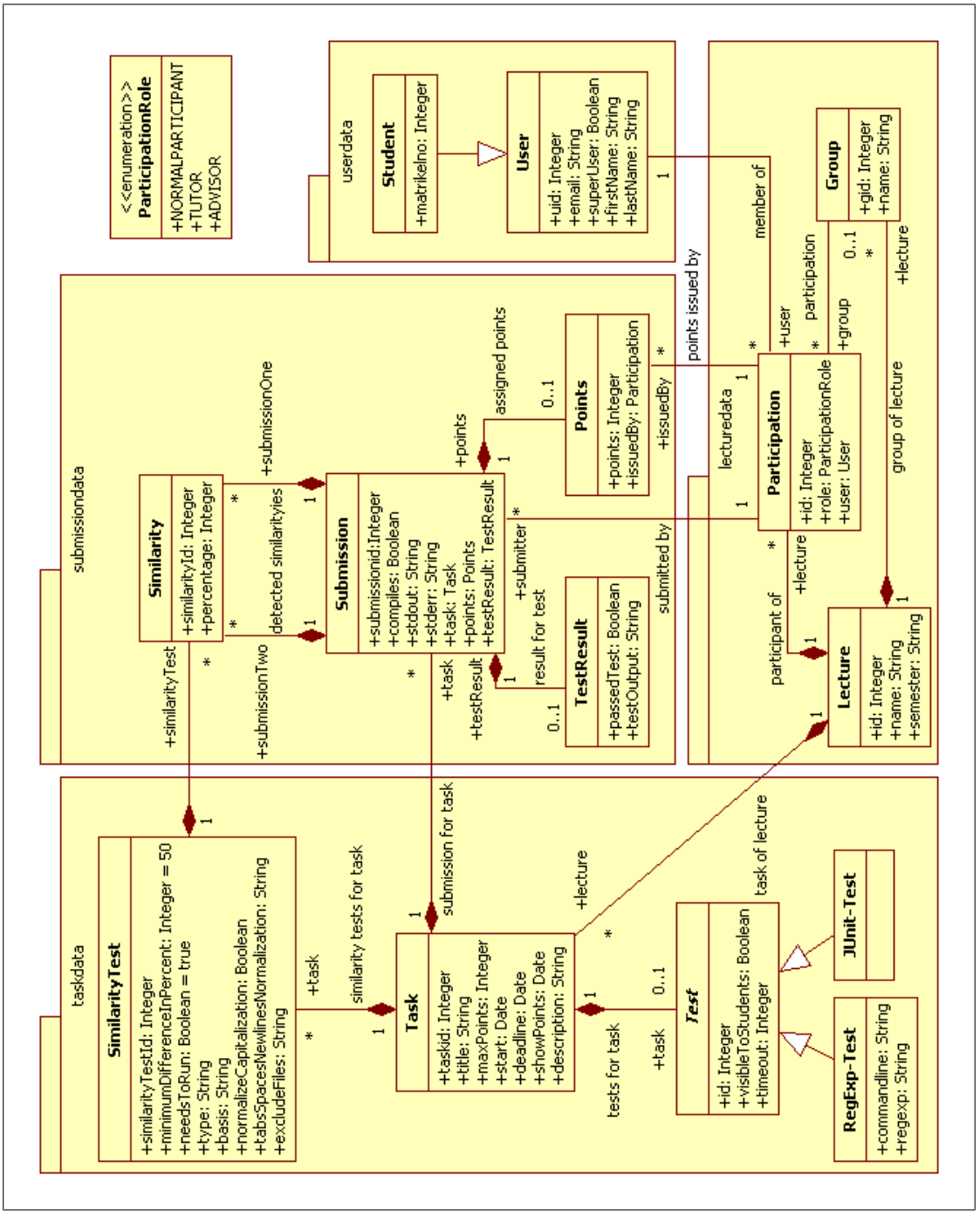


Abbildung 2.2: Das Datenmodell

Die Teilnahme eines Benutzers an einer Veranstaltung wird durch die Klasse *Participation* nach dem Koordinator-Analysemuster[1]modelliert. Dies ermöglicht eine Speicherung der teilnahmespezifischen Daten ohne Redundanzen. Insbesondere werden dort die Teilnahme-Rolle und die Gruppenzugehörigkeit vorgehalten.

Es wird zwischen drei Teilnahme-Rollen (*ParticipationRole*) unterschieden:

NormalParticipation
: Der Benutzer ist normaler Student und nimmt an der Veranstaltung ausschließlich als Teilnehmer teil.

Tutor
: Ein Tutor ist kein normaler Teilnehmer einer Veranstaltung. Er übernimmt vielmehr besondere Aufgaben wie z. B. die Gruppeneinteilung und die Bewertung von Abgaben.

Advisor
: Dieser Status ist eine Spezialisierung (im Sinne der objektorientierten Vererbung) des Tutor-Status'. Er ermöglicht dem Benutzer, zusätzlich Aufgaben einzustellen und Tutoren zu bestimmen.

Zu jeder Veranstaltung können beliebig viele Aufgaben (*Task*s) angelegt werden. Wie bei den funktionalen Anforderungen (siehe Kapitel 2.1) bereits beschrieben, werden hier die Eigenschaften der Aufgaben gespeichert. Optional kann für eine Aufgabe auch (maximal[2]) ein *Test* definiert werden. Für den abstrakten *Test* sind derzeit die beiden Ausprägungen *JUnit-Test* und *RegExp-Test* als Spezialisierungen vorgesehen.

Sollen für eine Aufgabe Ähnlichkeitsprüfungen der Einsendungen durchgeführt werden, so lassen sich mit Hilfe der *SimilarityTest*-Klasse die Randbedingungen einer solchen Prüfung festlegen: Welche Prüfung soll vorgenommen werden (*Type*, Algorithmusauswahl), auf welcher *Basis* (Quellcode, Kommentare oder beidem) soll die Plagiat-Prüfung stattfinden und inwiefern soll die *Basis* normalisiert werden. Ferner kann festgelegt werden, welche Dateien von der Überprüfung ausgeschlossen werden sollen (*ExcludeFiles*) und wie hoch die errechnete Ähnlichkeit mindestens sein muss, damit diese erkannt bzw. vermerkt wird.

[1] nach [6], Muster ähnlich einer UML Assoziationsklasse

[2] Keine Einschränkung, da man die einzige Ausgabe mit genau einem regulären Ausdruck prüfen und ein JUnit-Test auch eine TestSuite sein kann (vgl. Design JUnit [7], Komposite Pattern [8])

Meist intuitiv bei der Erstellung eines Datenmodells benutzt, aber dennoch erwähnenswert ist das Exemplartyp-Muster[3], welches sich exemplarisch an den Klassen *SimilarityTest* und *Similarity* verdeutlichen lässt. Dieses Muster dient der Datenkonsolidierung. Statt hier bei jeder *Similarity* Instanz die Konfiguration des zugrunde liegenden *SimilarityTest*s zu speichern, wird auf die Mehrfachspeicherung von gleichen Daten verzichtet und statt dessen eine Referenz verwendet.

Das letzte Paket enthält die abgabespezifischen Daten. Jeder Abgabe (*Submission*) ist genau eine Aufgabe zugeordnet; Sie beinhaltet Informationen über den eingesandten Quellcode: Ergebnis der Kompilierung, Fehlermeldungen des Compilers und Programm-Ausgaben (sofern ein Funktionstest definiert wurde). Zur Verbesserung der Übersichtlichkeit wurden das Funktionstest-Ergebnis (*TestResult*) und die Bepunktung (*points*) in eigene Klassen ausgelagert. Einsendungen wie auch die „Punkte“ sind mit der Teilnahme assoziiert, um deren Herkunft bestimmen und eine konsistente Datenhaltung[4] bereits auf der Modellebene garantieren zu können.

Resultate der *SimilarityTest*s werden *Similarity* genannt, speichern die prozentuale Ähnlichkeit und sind mit jeweils dem zugrunde liegenden *SimilarityTest* als auch mit den betreffenden beiden Einsendungen *SubmissionOne* und *SubmissionTwo* assoziiert.

2.3 Lösungsansatz

2.3.1 Allgemeiner Ansatz

Als grundlegender Ansatz wird eine Client-Server-Architektur angestrebt. Die Daten befinden sich dabei an einer zentralen Stelle, da sie so am besten zu verwalten sind. Sie sind über den Server verfügbar, der für die Einhaltung der Zugriffsrechte sorgt.

Ein Online-System, welches von vielen verschiedenen Personen verteilt benutzt wird, sollte einfach und mittels Standard-Software zugänglich sein: bestenfalls mit einer Software, die bereits bei der Mehrzahl der internetfähigen Computer vorhanden ist und nicht zusätzlich installiert werden muss. Anhand dieser Überlegung kommt momentan nur die HyperText Transfer Protocol- (HTTP) gemeinsam mit der HyperText Markup

[3] Analysemuster nach [9], oftmals auch Item-Item-Description-Muster genannt
[4] Einsendungen oder Bepunktungen von „Nichtteilnehmern“ sind nicht möglich

Language-Technologie (HTML) des World-Wide-Web in Frage. Web-Browser als Schnittstelle zum World-Wide-Web sind für nahezu jedes System vorhanden und sehr häufig auch Bestandteil aktueller Betriebssysteme.

Zur Nutzung dieser Technologie-Kombination ist eine Web- bzw. HTTP-Server-Software auf der Server-Seite erforderlich. Zusätzlich muss der Webserver in der Lage sein, Benutzereingaben entgegen zu nehmen und dynamische Antworten generieren zu können.

Da innerhalb des Systems größere Datenmengen anfallen und die Daten effizient abrufbar und verknüpfbar sein müssen, ist weiterhin eine relationale Datenbank notwendig.

Für die funktionale Anforderung *Duplikate erkennen* ist eine Duplikat-Erkennung mit Ergebnisnormalisierung erforderlich. Der nächste Abschnitt gibt einen Überblick über bestehende Methoden und Systeme.

2.3.2 Duplikat-Erkennung

Es gibt verschiedene Ansätze und Systeme für die Duplikat-Erkennung.

Unterschieden wird hauptsächlich zwischen dem „Merkmal“- oder auch „Attribut-counting“-Vergleich und dem Struktur-Vergleich. Laut [10] benutz(t)en die ersten automatischen Plagiat-Erkennungssysteme den Merkmal-Vergleich. Hierbei werden verschiedene Attribute der Software herangezogen und daraus eine Metrik gebildet. Als Attribute wurden z. B. folgende Werte benutzt (hier für Fortran-Programme): Anzahl von eindeutigen Operatoren, Anzahl von eindeutigen Operanden, Anzahl von Kommentaren oder Anzahl von Leerzeilen. Manche dieser Systeme benutz(t)en bis zu 24 verschiedene Software-Metriken, um Abstände in einem n-dimensionalen kartesischen Koordinatensystem zu berechnen (vgl. [11, Kapitel 1.2]). Jedoch haben solche Systeme nur mäßigen Erfolg ([10], [12] sowie [13]). Neuere Entwicklungen basieren alle auf dem Struktur-Vergleich. Aus diesem Grund und aus Mangel an Implementierungen für Java-Quellcode wurde diese Erkennungsmethode nicht weiter verfolgt.

Beim Struktur-Vergleich wird, wie der Name bereits impliziert, die Struktur von zu vergleichenden Quellcodes untersucht. Gefundene (approximative) Übereinstimmungen werden in eine Art Ähnlichkeitsmaß umgewandelt.

In der Literatur finden SIM (von Grune et al.), SIM (von Gitchell und Tran), YAP3 (Yet Another Plague Version 3), JPlag, MOSS (Measure Of Software Similarity) und Sherlock häufig Erwähnung. Alle genannten Systeme (bis auf MOSS) parsen den Quellcode, wandeln ihn in eine Token-Folge um (Sherlock bietet noch weitere Normalisierungen, [14]) und versuchen damit, auf teilweise verschiedenen Wegen Duplikate oder Teilduplikate zu erkennen.

SIM (von Grune et al., 1989, [15]) zählt sicherlich zu den ersten Struktur-basierten Duplikatstestern. Er versucht, abhängig von einer minimalen Länge maximale gemeinsame Substrings zu finden.

SIM (von Gitchell und Tran, 1999, [16]) wandelt die Token-Sequenz in einen String um, teilt ihn in funktionen-repräsentierende Sektionen und erzeugt String-Alignments durch Einfügen von Leerzeichen (inkl. Gap-Bewertung). Dadurch wird ein „Abstand“ zweier Programme ermittelt.

YAP3 ([12], 1996) versucht die maximalen nicht-überlappenden Token-Sequenzen (mittels Running-Karb-Rabin Greedy-String-Tiling) zu finden. Dadurch können auch modifizierte Duplikate gefunden werden, in denen Reihenfolgen von Methoden getauscht und/oder zusätzliche Aufrufe eingefügt wurden.

JPlag ([11], 2000) benutzt den „gleichen“ Algorithmus wie YAP3, verwendet aber noch weitere spezielle Token, welche die Semantik widerspiegeln (z. B. BEGINMETHOD statt nur OPEN_BRACE). In [11] befindet sich eine genauere Beschreibung und zudem eine empirische Evaluation.

MOSS ([17], 2003) bildet alle k-Gramme (alle Substrings der Länge k) des normalisierten Textes bzw. Quellcodes, hasht sie und wählt mit einem „Winnowing“ genannten lokalen Algorithmus innerhalb eines Fensters, das über alle Hashes geschoben wird, bestimmte Hashes (Fingerprints) aus. Durch die k-Gramme enthalten die Fingerprints auch positionale/strukturelle Informationen. Alle Fingerprints werden in einer Datenbank gespeichert. In einem zweiten Schritt erfolgt eine Prüfung aller Fingerprints der verschiedenen Dateien gegen die Datenbank. Erst danach werden Paare von vermeintlich ähnlichen Dokumenten, die einen Grenzwert übersteigen, überprüft.

Sherlock ([14], 2004) unterstützt nur eine einzige Datei (kann auch ein unkomprimiertes tar-Archiv sein) und benutzt verschiedene Normalisierungen (die Umwandlung in eine Token-Folge ist eine davon). Danach wird für jeden Vergleich zweier Dateien über alle

Zeilen iteriert und die Zeilen werden in einer Hash-Map gespeichert. Tritt dabei eine Kollision auf und sind die Zeilen identisch, so wird versucht, eine Sequenz von gleichen Zeilen (als Run bezeichnet, darf auch durch eine geringe Zahl von Zeilen unterbrochen sein) zu finden, die in beiden Dateien auftritt. Es werden die längsten Runs gesucht und mit der Länge der Ursprungsprogramme in Prozent umgerechnet.

Plaggie ([18], 2006) basiert auf den gleichen Algorithmen wie JPlag (ist aber eine freie Implementation).

Die beiden Systeme JPlag und MOSS sind nur als Web-Applikationen nutzbar. Die Schnittstellenbeschreibungen beider Systeme sind verfügbar. Für die komplexe Vorgehensweise von MOSS existiert nur eine grobe Beschreibung des Algorithmus.

Die Verfügbarkeit des Programms ist ein wichtiger Faktor für eine einfache Integration. Speziell bei verfügbarem Quellcode ist die Lizenz zu beachten, die angibt, wie man das Programm nutzen bzw. verändern darf: Liegt der Quellcode des Systems vor und kann das System z. B. keine Java-Programme analysieren, so muss es erlaubt sein, diese Funktion hinzuzufügen.

Die folgende Tabelle stellt die wichtigsten technischen Daten der einzelnen Systeme übersichtlich dar:

System	**Java Analyse**	**Lokale Benutzung**	**Verfügbarkeit**	**Lizenz**
SIM (Grune et al.)	ja	ja	Quellcode (C)	unbekannt
SIM (Gitchell u. Tran)	nein	ja	nicht verfügbar[a]	–
YAP3	nein	ja	Quellcode (Perl)	unbekannt
JPlag	ja	nein	WSDL[b] Beschreibung	–
MOSS	ja	nein	Abgabe Format	–
Sherlock	ja	ja	Quellcode (Java)	GPL[c]
Plaggie	ja	ja	Quellcode (Java)	GPL

[a] Es ist nur eine textuelle Beschreibung des Algorithmus in [16] verfügbar.
[b] Web Services Description Language (WSDL)
[c] GNU General Public License (GPL)

Da die bisher genannten Methoden auf Quellcode optimiert sind, wird weiterhin eine sehr elementare Metrik in Betracht gezogen, die den Abstand zweier Zeichenketten

widerspiegelt (Anzahl von elementaren Änderungsschritten wie Einfügen und Löschen eines Zeichens). Diese Metrik wird auch Levenshtein- oder Editier/Edit-Distanz genannt und basiert auf dynamischer Programmierung.

Unabhängig davon gibt es noch einen Ansatz, der u. a. im Data-Mining eingesetzt wird, ein universelles Ähnlichkeitsmaß zu verwenden: Die Kolmogorow-Komplexität. [13] beschreibt ein System namens SID (Shared Information Detection), das genau hierauf basiert. Die Kolmogorow-Komplexität eines Strings x, $K(x)$, misst den Grad der Information den x enthält. $K(x)$ ist die Länge des kürzesten Algorithmus (ohne Eingaben), der x ausgibt. Nimmt man eine weitere Zeichenkette y hinzu, so gibt $K(x|y)$ die Länge des kürzesten Algorithmus an, der unter der Eingabe y x erzeugt. Die gemeinsame Information zweier Strings kann nach [13] und [19] definiert werden als

$$d_s(x,y) = \frac{K(x|y) + K(y|x)}{K(xy)},$$

wobei xy die Konkatenation von x und y darstellt. Dies entspricht dem Aufwand der wechselseitigen Erzeugung sowie einer Normalisierung auf $[0,1]$. Besser für ein Clustering bzw. eine Duplikat-Erkennung eignet sich laut [20] bzw. [19] die Normalized Information Distance (NID), welche den Abstand (innerhalb von $[0,1]$) angibt, wie ähnlich sich zwei Strings sind:

$$d(x,y) = \frac{\max\{K(x|y), K(y|x)\}}{\max\{K(x), K(y)\}}$$

$d_s(x,y)$ und $d(x,y)$ sind Metriken. Die Funktion K selbst ist keine Metrik, da es die Dreiecksungleichung nicht erfüllt. Es wurde ferner in [19, Kapitel 5] die Universalität gezeigt. Hieraus folgt, dass die NID alle anderen berechenbaren Metriken minorisiert.

3 Design/Entwurf

3.1 System-Architektur

Um den groben Ansatz aus Kapitel 2.3 zu konkretisieren, soll das System nicht nur aus zwei (Client-Server), sondern aus mehreren Schichten bestehen. Abbildung 3.1 zeigt die Zusammensetzung der gewählten Vier-Schichten-Architektur (eine leicht veränderte Drei-Schichten-Architektur). Es wird zwischen *GUI* (Graphical User Interface), *Applikation*, *Persistenz* und *Datenbank* unterschieden. Durch diese Aufteilung sollen die (möglichen) Verteilungen und die einzelnen Schnittstellen besser ersichtlich werden.

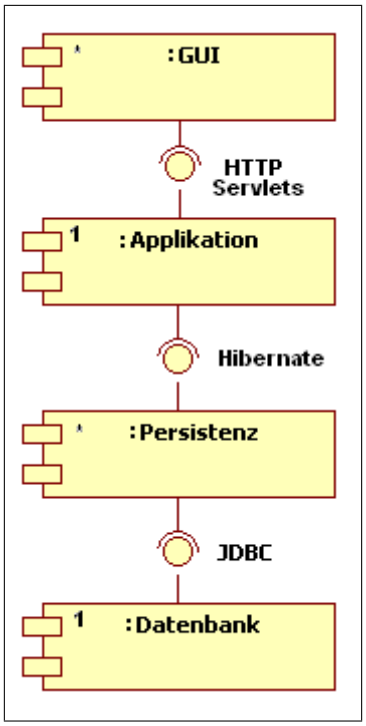


Abbildung 3.1: Schichten-Architekturmodell

Die Präsentationsschicht (*GUI*) dient als Benutzerschnittstelle und befindet sich auf den Clients (ähnlich einem Ultrathin-Client). Ein Webbrowser sorgt dort für die Darstellung der Inhalte (über HTML und Cascading Style Sheets (CSS)) sowie für die Entgegennahme der Eingaben und kommuniziert mittels HTTP-basierender Protokolle mit der *Applikation*sschicht.

Bei der *Datenbank* handelt es sich, wie in Kapitel 2.3 angedeutet, um ein relationales Database Management System (DBMS). Die *Persistenz*schicht enthält das Datenmodell. Es handelt sich dabei um die Datenhaltungsschicht klassischer Drei-Schichten-Architekturen; Sie verbirgt die Details konkreten *Datenbank* und ermöglicht der *Applikation*sschicht den Datenzugriff. Die Kommunikation zwischen der *Persistenz*schicht und der *Datenbank* geschieht über Java Database Connectivity (JDBC) in Verbindung mit dem Hibernate-Framework ([21], für das Object Relational Mapping (ORM)), einer Implementierung der Java Persistenz API ([22]). Die *Datenbank* muss sich nicht auf dem gleichen Rechner befinden wie die darüber liegenden Schichten, sondern kann sich auch auf einem weiteren Rechner befinden (ähnlich dem FAT-Client).

Die *Applikation*sschicht kapselt die Anwendungslogik. Sie nimmt also Anfragen der *GUI* entgegen, verändert Daten der *Persistenz*schicht und bereitet Daten für die *GUI* auf. Innerhalb der *Applikation*sschicht wurde die Variante MVC Model 2 (MVC-2) des Model-View-Controller-Musters (MVC) angewendet (siehe [23, Kapitel 8.2]). Das klassische MVC-Muster ist für webbasierte Anwendungen ungeeignet, da das Modell den View, der sich (im Browser) auf dem Client befindet, nicht über das Observer-Muster bei Änderungen informieren kann.

Das Modell enthält die Daten, die vom View angezeigt werden sollen. Es ist vom Controller und den Views unabhängig.

Der View sorgt im klassischen MVC für die Darstellung der Daten aus dem Modell. Bei Web-Applikationen geschieht dies nur indirekt, da dem View nur eine Beschreibungssprache (hier HTML) zur Verfügung steht. Der „echte“ View auf dem Client im Browser interpretiert die Beschreibung und sorgt für die Darstellung und Weiterleitung der Änderungswünsche des Benutzers an den Controller.

Der Controller nimmt die Anfragen und Eingaben der Benutzer entgegen, führt Operationen auf dem Modell aus (Geschäftslogik) und sorgt für die Auswahl des Views. Er besteht genau genommen aus zwei Teilen: der Webserver Virtual Machine (VM) und

speziellen Controller-Servlets (vgl. Kapitel 4.2.1 über die Auswahl der Programmiersprache). Die Webserver VM nimmt die Benutzereingaben im CGI POST- bzw. GET-Format entgegen, stellt diese den Servlets über ein Application Programming Interface (API) zur Verfügung und übergibt den Kontrollfluss an ein (konfiguriertes) Servlet. Das ausgewählte Servlet kann entweder ein View oder ein Controller sein, der weitere Aktionen durchführt und schließlich zu einem View weiterleitet.

Diese Aufteilung hat den Vorteil, dass Änderungen an den Views unabhängig durchgeführt werden können. Innerhalb dieser Arbeit generiert der View der *Applikation*sschicht ausschließlich striktes HTML 4.0 ([24, S. 40]). Dies bedeutet, dass der HTML-Code keine Informationen über Seitendarstellung sondern nur über die Seitenstruktur enthält. Dadurch liegt eine weitere Trennung von Inhalt und Layout vor und ermöglicht die Konfiguration der Darstellung über CSS, die erst im Browser auf dem Client angewendet werden.

Abbildung 3.2 (auf der nächsten Seite) zeigt ein UML Sequenzdiagramm mit dem Ablauf einer „lesenden" Anfrage wie z. B. das Auflisten aller Aufgaben einer Vorlesung. Der *Browser* stellt eine HTTP-Anfrage an den *Webserver*. Die Anfrage durchläuft dort mehrere Filter[1] und gelangt bei korrekter Authentifikation zu einem *Controller-Servlet*, wo Daten aus der Datenbank geladen und die Anfragen intern zu einem *View-Servlet* weiterleitet werden. Die dort erzeugte *HTML-Ausgabe* wird schließlich zum *Browser* übertragen.

Da der Zugriff auf das System nur authentifiziert erfolgen soll und damit nicht in jedes Servlet Authentifikations-Aufrufe eincodiert werden müssen, wird die Authentifikation über einen Filter (*AuthenticationFilter*) erreicht. Alle Aufrufe müssen diesen Filter passieren. Liegt keine valide Authentifikation vor, so unterbricht dieser Filter den Aufruf und fragt den Benutzer nach seinen Daten.

Der *HibernateFilter* sorgt dafür, dass wartende Änderungen an der Datenbank durchgeführt werden und der Zwischenspeicher der Hibernate-Sitzung geleert wird.

Für die Bearbeitung von HTML-Formularen bzw. Änderungsanfragen wird das Post-Redirect-Get-Muster (siehe [25]) angewendet. Abbildung 3.3 zeigt einen solchen Aufruf: Der Controller leitet nicht (wie in Abbildung 3.2) intern an einen View weiter, sondern

[1] Filter sind Teil der Servlet Spezifikation 2.5 ([3]) und können transparent vor oder nach Servlets geschaltet werden, um z. B. Authentifikation, Komprimierung oder Bildkonvertierung durchzuführen.

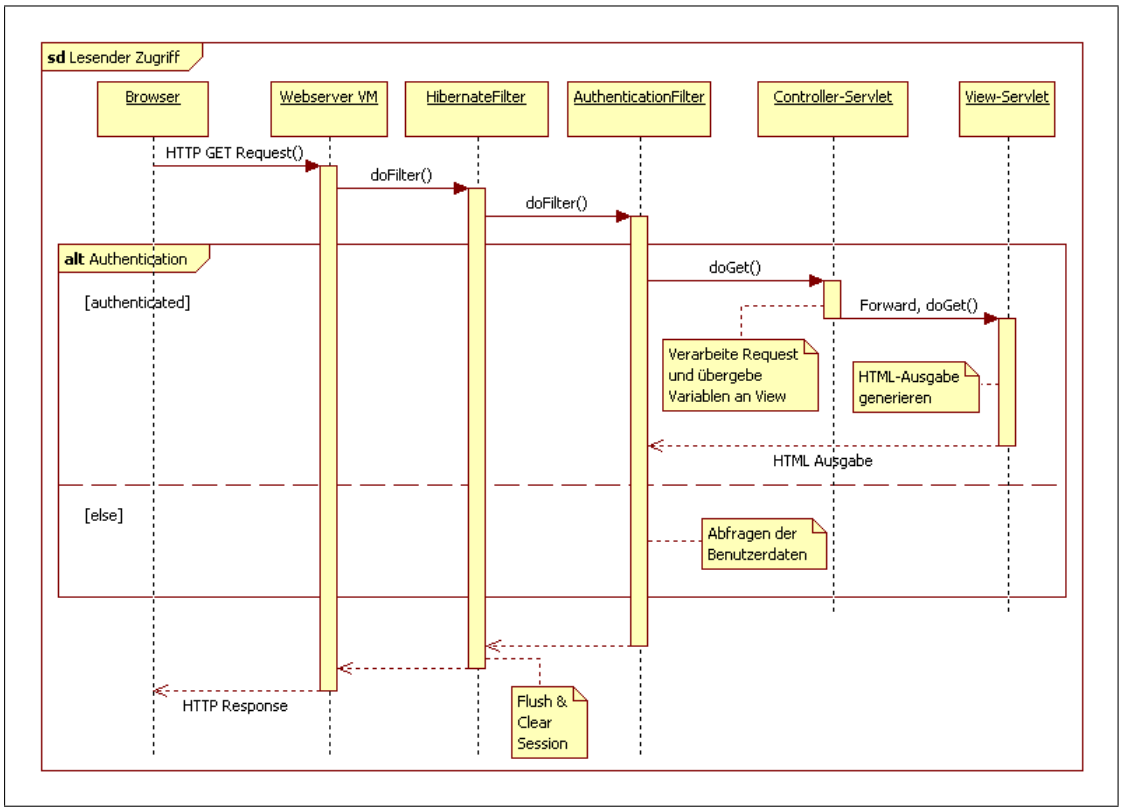


Abbildung 3.2: Verhalten bei einem HTTP-GET-Request

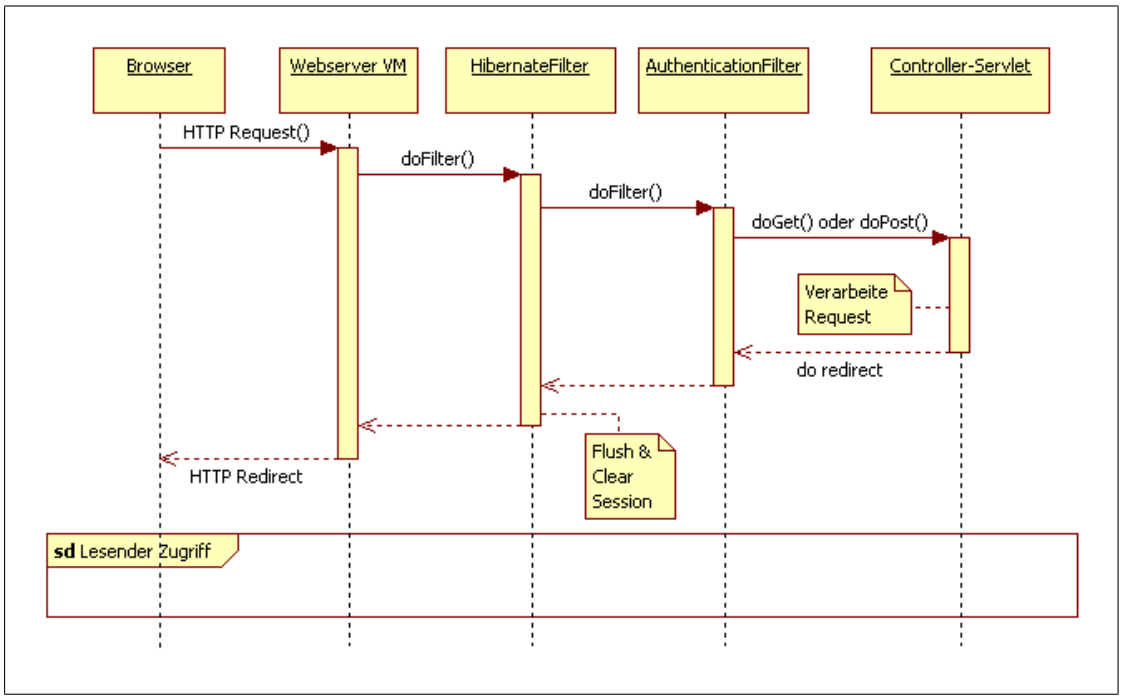


Abbildung 3.3: Verhalten bei einem modellmodifizierenden HTTP-Request

sorgt dafür, dass der *Browser* eine neue Anfrage stellt. Dies hat den Vorteil, dass der erzeugte View nach einer Änderung keine Aufrufparameter mehr enthält, welche bei Benutzung des Zurück-Buttons oder beim Bookmarken der Seite eine erneute Änderungsanfrage zur Folge hätte.

Der Zugriff auf die Datenbank erfolgt über Data Access Object-Klassen (DAO). Diese Klassen ermöglichen es, Daten aus der Datenbank (über Hibernate) abzufragen, zu löschen oder zu speichern. Sollte zukünftig das Hibernate-Framework evtl. durch ein anderes ersetzt werden, so ist dies relativ einfach möglich: Es müssen lediglich neue DAO-Implementierungen entwickelt und die Fabrik (*DAOFactory*) muss angepasst werden, damit die neuen DAOs benutzt werden.

Die Paketstruktur ist in Abbildung 3.4 dargestellt.

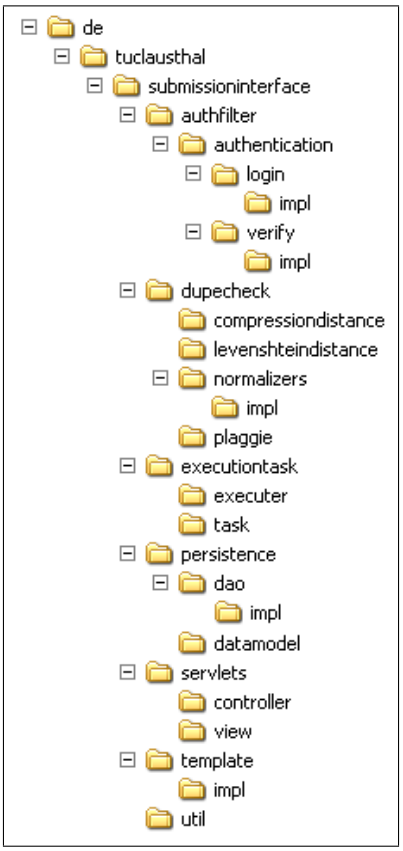


Abbildung 3.4: Paket-Struktur

In den Paketen *login*, *verify*, *normalizers*, *dao* und *template* befinden sich die Interfaces, in den Paketen *impl* direkt darunter dann die Referenz-Implementierungen. Bei *dao*, *executiontask*, *authfilter* und *template* befinden sich zudem Fabriken, um hier eine größere Anpassbarkeit zu erreichen.

3.1.1 Architektur des Authentifikationsfilters

Im Zentrum des Entwurfs des Authentifikationsfilters stand eine große Anpassbarkeit an verschiedenste (Benutzername-/Passwort-basierender) Authentifikationssysteme. Zu diesem Zweck wurde die Authentifikation in zwei Teile aufgeteilt: *Login* und *Verify*, die vom *AuthenticationFilter* miteinander verknüpft werden (vgl. Abbildung 3.5).

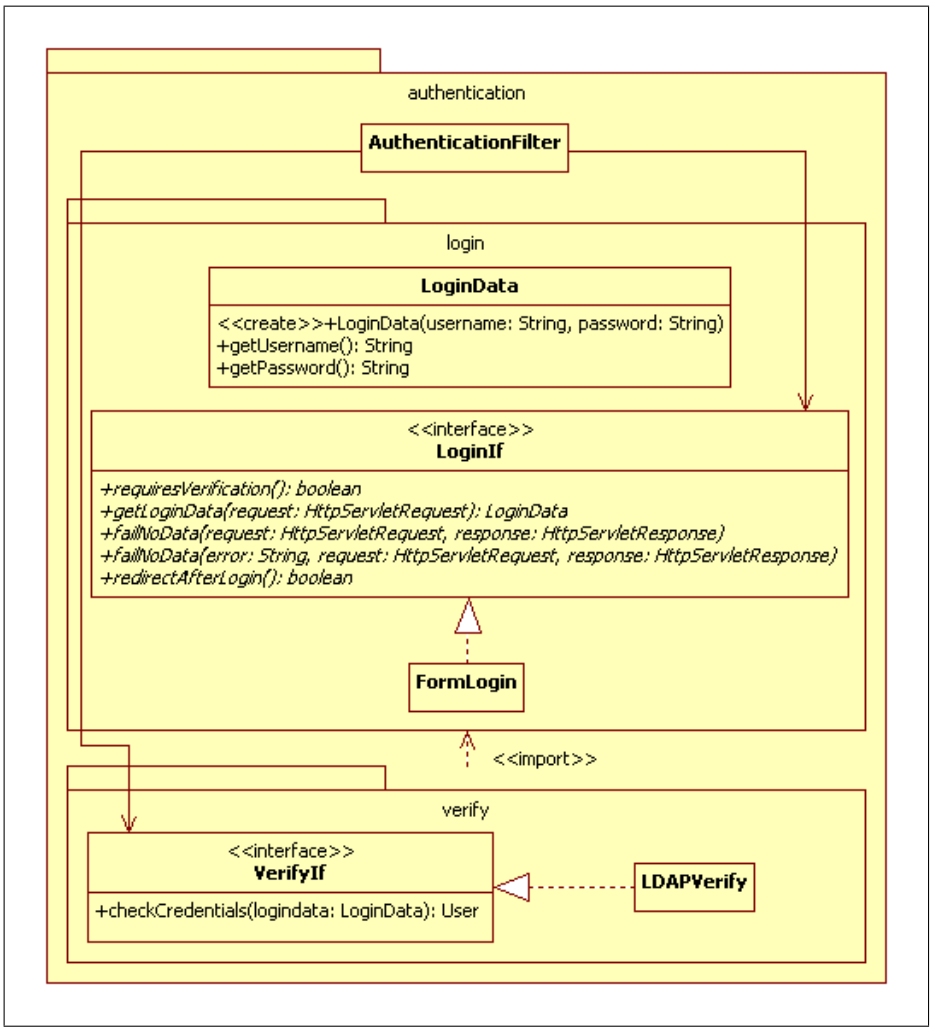


Abbildung 3.5: Aufbau des Authentifikationsfilters

Der *Login*-Part ist verantwortlich für die Abfrage der Benutzerdaten. Der *Verify*-Part übernimmt die Verifikation der eingegeben Daten. Dadurch können beide unabhängig voneinander ausgetauscht werden.

Das *LoginIf* kapselt alle Eigenschaften und Methoden für eine konkrete Login-Methode:

requiresVerification(): Boolean
: Diese Methode gibt zurück, ob eine (weitere) Verifikation mittels *VerifyIf* der Daten notwendig ist.

getLoginData(HttpServletRequest request): LoginData
: Diese Methode ruft die Login-Daten ab und gibt sie zurück.

failNoData(HttpServletRequest request, HttpServletResponse response)
: Diese Methode wird aufgerufen, wenn keine Login-Daten vorliegen und diese vom Benutzer noch abgefragt werden müssen.

redirectAfterLogin(): Boolean
: Der Rückgabewert dieser Methode legt fest, ob ein HTTP-Redirect nach einer erfolgreichen Authentifikation nötig ist.

Im Rahmen dieser Bachelorarbeit wurde eine Formular-basierte Passwortabfrage (*FormLogin*) mit Verifikation durch einen Lightweight Directory Access Protocol-Server (LDAP, *LDAPVerify*) zur Authentifikation umgesetzt. Mit Hilfe dieser Struktur sind auch andere Methoden realisierbar, wie z. B. HTTP-Basic-Authentification.

Das *VerifyIf* besitzt lediglich eine Methode *checkCredentials*, welche mit den abgefragten Benutzerdaten (*LoginData*) aufgerufen wird und den *User* oder im Fehlerfall *null* zurück liefert.

Eine Schwierigkeit des HTTP-Protokolls besteht darin, dass es komplett zustandslos ist. Dies bedeutet, dass jeder Request-Zyklus für sich allein steht. Folglich muss eine Sitzung (Login, authentifizierter Zugang, Logout) „künstlich“ nachgebildet werden, sofern die Überprüfung nicht bei jedem Zugriff erfolgen soll. Die Servlet-API bietet hierfür spezielle Funktionen, die vom Authentifikationsfilter zu diesem Zweck genutzt werden.

3.1.2 Architektur der Kompilierungs- und Funktionstests

Da die Kompilierungs- und Funktionstests eine gewisse Zeit benötigen und zum Ende der Abgabefrist wahrscheinlich gehäuft auftreten werden, sollten diese nicht synchron nach der Abgabe vom Controller-Servlet ausgeführt werden, sondern asynchron (teilweise serialisiert) im Hintergrund. Dadurch werden die Wartezeiten für den Benutzer auf die HTTP-Response von der Dauer der Tests entkoppelt und gleichzeitig eine Überlastung des Servers durch sehr viele parallel ablaufende Tests vermieden.

Abbildung 3.6 zeigt die Architektur des Frameworks, das für die Tests entworfen wurde: Dabei wird zwischen den Aufgaben (den Tests, hier *ExecutionTask* genannt) und den Ausführern der Aufgaben (*Executer*) unterschieden.

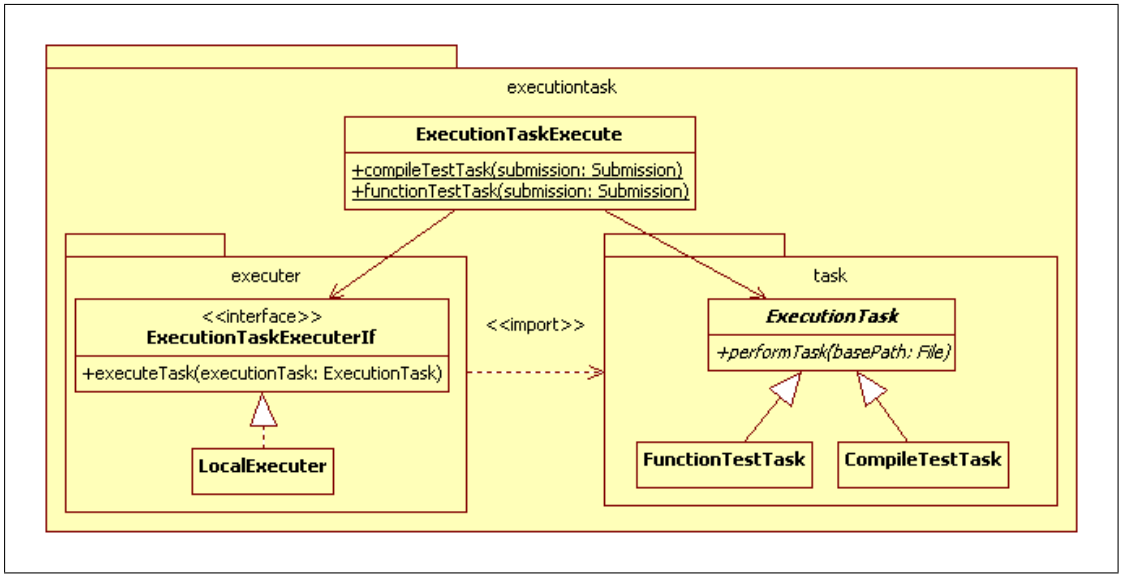


Abbildung 3.6: Architektur der Kompilierungs- und Funktionstests

Die abstrakte Klasse *ExecutionTask*, von der alle Aufgaben erben, ist sehr allgemein gehalten, so dass nahezu beliebige Aufgaben implementiert und mit diesem Framework ausgeführt werden können. In der aktuellen Version sind hier nur der Kompilierungs- (*CompileTestTask*) und die Funktionstests (*FunctionTestTask*) vorgesehen.

Ein *Executer* kapselt die Logik, wie die *ExecutionTask*s ausgeführt werden. Dies beinhaltet z. B. das Eintragen in eine Warteschlage und die Ausführung der einzelnen Aufgaben durch Aufrufen der *performTask*-Methode.

Die Klasse *ExecutionTaskExecute* kann als eine Art Fabrik für die konkreten *Executer* angesehen werden. Die Methoden dieser Klasse erstellen eine konkrete *ExecutionTask*-Instanz und rufen einen *Executer* auf, der dann für die weitere Ausführung sorgt.

In der Referenz-Implementation wird der *Executer* (*LocalExecuter*) innerhalb der VM des Webservers ausgeführt, wobei durchaus mehrere (aber nur eine feste Anzahl, z. B. Anzahl Prozessor-Kerne minus eins) *ExecutionTask*s parallel ausgeführt werden. Durch die gewählte Architektur könnten sie aber auch z. B. über Remote Method Invocation (RMI) in eine oder auch mehrere (verteilte) VMs ausgelagert werden.

3.1.3 Architektur der Duplikat-Erkennung

Alle Duplikat-Erkennungsmethoden basieren auf der abstrakten Klasse *DupeCheck* (vgl. Abbildung 3.7). Sie besitzt zwei Methoden und wurde auch in Form eines Frameworks entworfen:

calculateSimilarity(StringBuffer fileOne, StringBuffer fileTwo, int maximumDifferenceInPercent)
: CalculateSimilarity berechnet die Ähnlichkeit von *fileOne* und *fileTwo* mit unterer Schranke von *maximumDifferenceInPercent* (z. B. als Abbruchbedingung für einen Algorithmus, wenn die berechnete Ähnlichkeit unter diesen Wert fällt) und gibt das Ergebnis in Prozent zurück.

performDupeCheck(SimilarityTest similarityTest)
: Diese Methode wird mit einem *SimilarityTest* aufgerufen und führt anschließend die konfigurierte Duplikat-Erkennung durch. D. h. sie iteriert über alle eingesandten Dateien (lädt und normalisiert sie), ruft jeweils für einen Vergleich zweier Dateien bzw. Zeichenketten die *calculateSimilarity*-Methode auf und speichert die berechneten Werte in der Datenbank.

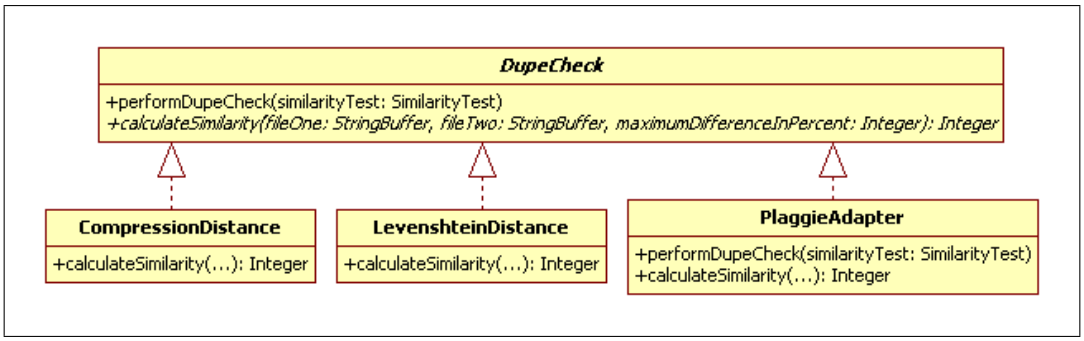


Abbildung 3.7: Architektur der Duplikat-Erkennung

Mittels Subclassing werden die konkreten Duplikat-Erkennungsmethoden umgesetzt. Bei allen auf Zeichenketten basierenden Algorithmen (*LevenshteinDistance* und *CompressionDistance*) muss ausschließlich die *calculateSimilarity*-Methode implementiert werden. Für andere bzw. externe Erkennungssysteme wie Plaggie wird die *performDupeCheck*-Methode überschrieben und damit ein Adapter (siehe [8], *PlaggieAdapter*) zum Verbinden der inkompatiblen Schnittstellen und zum Speichern der Ergebnisse in der Datenbank erstellt.

Die Normalizer für Java-Quellcode sind nach der Art des Composite-Patterns (nach [8]) organisiert (vgl. Abbildung 3.8). Hierdurch ist es möglich, aus einzelnen, einfachen NormalizerIf-Implementationen komplexe Normalizer zu konstruieren und es erleichtert das Hinzufügen neuer Normalizer. Zudem sorgt dies dafür, dass der Client (hier die Klasse *DupeCheck*) vereinfacht wird, da die Logik für die Schachtelung oder auch z. B. Caching vor ihm verborgen bleibt und durch das Composite gekapselt wird: Der Client muss nur mit dem NormalizerIf umgehen können.

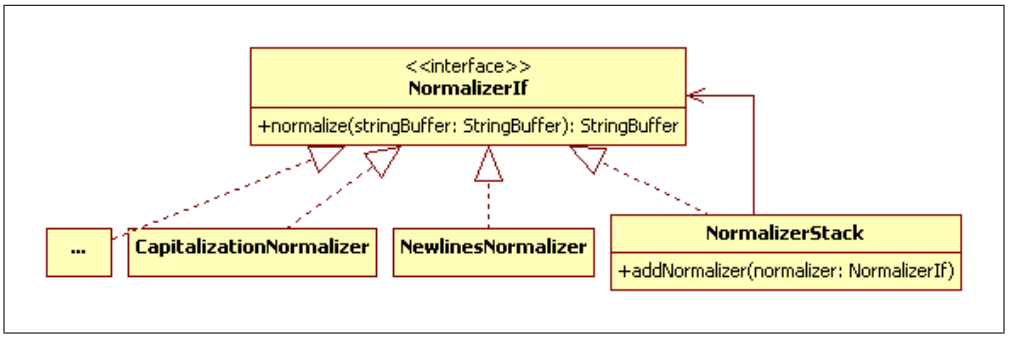


Abbildung 3.8: Aufbau der Normalizer

Die Duplikat-Erkennung soll nach Ablauf der Abgabefrist automatisch in einer separaten VM durchgeführt werden, um die Webserver VM nicht zu belasten. Dafür ist ein Taskplanner erforderlich (z. B. cron), wie er in allen gängigen Betriebssystemen vorhanden ist.

3.2 Anforderungen an die Duplikat-Erkennung

Im Abschnitt 2.3.2 wurden verschiedene Systeme bzw. Duplikat-Erkennungsmethoden vorgestellt. Innerhalb dieses Systems sollen unterschiedliche Duplikat-Erkennungsmethoden (gleichzeitig) zur Auswahl stehen, um die Erkennungsrate weiter zu verbessern.

[14] und [26] beschreiben die Motivation der Studenten, Plagiate als eigene Lösungen abzugeben. Auch werden hier unterschiedliche Vorgehensweisen der Studenten beleuchtet. Demnach kopieren Studenten Programmieraufgaben und ändern dabei nur Namen (oder auch Variablen und Einrückungen), ohne das Programm vollständig zu verstehen. Insbesondere bei sehr stark festgelegten, kleineren Aufgaben sind die auf Programmcode optimierten Algorithmen (mit starker Normalisierung) ungeeignet. Bei einer Aufgabe der Art

> Zählen Sie mit einer While-Schleife von 10 herunter, geben Sie jeden 0,5er Schritt und am Ende *Zero!* auf der Konsole aus.

errechnen sie fast ausschließlich 100 % Ähnlichkeit zwischen den Einsendungen.

Aus diesem Grund muss eine Auswahl getroffen werden, so dass bei komplexeren sowie bei festgelegteren, einfacheren Aufgaben eine gute Erkennung möglich und sichergestellt wird.

4 Implementierung

4.1 Auswahl der Duplikat-Erkennung

Nachdem im Abschnitt 2.3.2 einige Algorithmen vorgestellt und im Abschnitt 3.2 die Anforderungen an die Algorithmen, die innerhalb des Systems eingesetzt werden sollen, vorgestellt wurden, wird hier die Auswahl beschrieben und es werden weitere Informationen zur Realisierung gegeben.

Nach [27] und eigenen Tests sind MOSS und JPlag für komplexere Aufgaben (besonders) zu empfehlen. Beide sind jedoch nur als Webdienste verfügbar, was zur Folge hätte, dass studentische Ausarbeitungen zur Überprüfung auf externe Server geladen werden müssten (Datenschutz).

Plaggie kann als eine frei verfügbare Variante von JPlag gesehen werden, die nach dem Test von [27] erstellt wurde. Daher findet Plaggie in dem genannten Test keine Erwähnung und wird als Erkennungsmethode für komplexere Aufgaben ausgewählt und in das System integriert. Der in Plaggie enthaltene Greedy-String-Tiling Algorithmus weist eine Worst-Case Komplexität von $\mathcal{O}(n^3)$ (n ist die Länge der längsten Token-Folge eines Vergleichs) auf. Es ist jedoch möglich, die Durchschnitts-Zeitkomplexität durch weitere Optimierungen auf unter $\mathcal{O}(n^2)$ zu verbessern ([11, Kapitel 3.4]). Insbesondere werden durch die Tokenisierung (ein ASCII-Zeichen[1] pro Token) sehr lange Programme auf kurze Token-Folgen (n in der Größe von wenigen hundert Zeichen) normalisiert, so dass sich die quadratische Laufzeit nicht so stark auswirkt.

Für kleinere Aufgaben wird auf die elementare Editier-Distanz und die universelle Kolmogorow-Komplexität gesetzt. Je nach Grad der Normalisierung (doppelte Leerzeichen entfernen, in Kleinschreibung konvertieren, ...) und Grundlage der Überprüfung (nur Quellcode, nur Kommentare oder gemischt) sollen mit beiden Metriken unter-

[1] American Standard Code for Information Interchange (ASCII)

schiedliche Aspekte verglichen werden können, die mit Plaggie nicht differenzierbar sind.

Der Levenshtein-Algorithmus hat eine Speicher- und Zeitkomplexität von $\mathcal{O}(n \cdot m)$ (n und m sind die Längen der Zeichenketten), kann jedoch im Speicherverbrauch auf $\mathcal{O}(\max\{m, n\})$ optimiert werden. Bricht man den Algorithmus bereits ab, sobald eine minimale Übereinstimmung unterschritten wird, so kann auch die (praktische) Zeitkomplexität reduziert werden (k sei die maximale Distanz): $\mathcal{O}(k \cdot \min\{m, n\})$. Der Algorithmus eignet sich daher vor allem für kürzere Zeichenketten/Lösungen, welche u. a. durch Normalisierung erreicht werden können. Normalisierung ist im Allgemeinen notwendig, damit einfache Änderungen (z. B. das Einfügen von Leerzeichen) nicht zu einer niedrigeren prozentualen Ähnlichkeit führen. Bei komplexeren Lösungen gibt es bei diesem Algorithmus zudem das Problem, dass er, anders als Plaggie, keine Methodenpermutationen (oder andere quellcodespezifischen Änderungen) erkennt und folglich eine höhere Distanz errechnet.

Die Kolmogorow-Komplexität soll nach [13] nicht so anfällig für Methodenpermutationen oder andere Veränderungen sein und ist nach Aussagen der Autoren in der Theorie nicht betrügbar, jedoch ist hier trotzdem eine Normalisierung aus dem o. g. Grund anzuraten. Für die Praxis gibt es das Problem, dass die Kolmogorow-Komplexität nicht berechenbar ist, durch eine gute Komprimierung kann sie aber angenähert werden (in der Literatur daher auch oft „Kompressions-Distanz“ genannt). Formt man die NID aus Kapitel 2.3.2 mit Hilfe von

$$\max\{K(x|y), K(y|x)\} = \max\{K(xy) - K(y), K(xy) - K(x)\} = K(xy) - \min\{K(x), K(y)\},$$

basierend auf dem grundlegenden Theorem $K(x|y) = K(xy) - K(y)$ der Kolmogorow-Komplexität um und substituiert $K(x)$ durch $Comp(x)$, erhält man die Normalized Compression Distance (NCD)

$$d(x, y) \approx \frac{Comp(xy) - \min\{Comp(x), Comp(y)\}}{\max\{Comp(x), Comp(y)\}},$$

wobei $Comp(x)$ die Länge des komprimierten Strings x darstellt. Ein Abstand von 1 bedeutet, dass die beiden Strings unähnlich sind, 0 das Gegenteil. Für eine prozentuale Angabe definiert man: $\tilde{d}(x, y) = (1 - d(x, y)) \cdot 100$. Es ist entscheidend, dass ein sehr

guter Kompressionsalgorithmus gewählt wird. Nach einem Test des Linux Journals ([28]) komprimiert der Lempel-Ziv-Markow-Algorithmus (LZMA) sehr gut bei gleichzeitig akzeptabler Geschwindigkeit. Auf Grundlage dieses Ergebnisses wurde LZMA für eine Implementierung ausgewählt.

Das System verfügt damit über eine elementare (Editier-Distanz), eine universelle (Kompressions-Distanz) und eine auf Programm-Quellcode optimierte Erkennungsmethode (Plaggie) sowie eine für jede Plagiat-Prüfung konfigurierbare Normalisierung.

4.2 Technologien

Bei der Auswahl der Technologien bzw. deren Implementierungen wurde besonders auf Flexibilität und Portierbarkeit/Plattformunabhängigkeit geachtet. Insbesondere wurde auf die Verwendung von Open Source Produkten Wert gelegt.

Als Datenbanksystem für die Realisierung des Systems wurde MySQL ([29]) gewählt, da es sich hier um eine relativ kleine und sehr portierbare Software handelt. Wie in Abschnitt 4.2.2 beschrieben, ist man auf diese Wahl aber nicht festgelegt.

Nachdem die Entscheidung für Java als Programmiersprache getroffen war (Abschnitt 4.2.1), wurde Tomcat ([30]) als Webserver bzw. Servlet-Container ausgewählt. Tomcat 6.0 implementiert die Servlet Spezifikation 2.5 ([3]) und ist unter einer freien Lizenz (Apache-Lizenz) verfügbar. Tomcat wurde in Java verfasst und ist damit, wie auch die Servlets, auf allen Java-Plattformen lauffähig.

4.2.1 Auswahl der Programmiersprache

Für die Web-Architektur stehen eine Reihe von bewährten Programmiersprachen bereit. Ursprünglich wurde für Webserver das Common Gateway Interface (CGI) entworfen, um dynamische Inhalte erzeugen zu können. Mit Hilfe dieser Schnittstelle kann theoretisch jede Programmiersprache verwendet werden, deren Kompilierung bzw. Laufzeitumgebung auf der Plattform des Webservers ausführbar ist. CGI legt dabei das Parameter-Übergabeformat bei HTTP-Anfragen (für GET und POST), diverse Umgebungsvariablen (mit Informationen über den Webserver und die Anfrage)

sowie den Datenaustausch zwischen Webserver und CGI-Anwendung (über Standard Input (STDIN) und Standard Output (STDOUT)) fest.

Für CGI-Anwendungen haben sich im Laufe der Zeit Programmiersprachen (zumeist Skriptsprachen[2]) mit dynamischer Typisierung und vor allem guten Zeichenkettenoperationen bewährt. Insbesondere sei hier Perl erwähnt, da diese Sprache ursprünglich für die Verarbeitung von Strings auf Kommandozeilenebene entwickelt wurde und es hier eine große Anzahl von Erweiterungen (Modulen) gibt, welche eigene Entwicklungen vereinfachen. Bei Perl wird die HTML-Ausgabe mit in das Skript hinein kodiert.

Speziell für dynamische Webseitenerstellung wurde PHP Hypertext Preprocessor (PHP) entwickelt und beinhaltet bereits eine große Anzahl von (speziellen) Funktionen, die die Webseitenentwicklung vereinfachen (Sessions, Datenbankanbindung, ...). Als HTML Präprozessor wird hier nicht HTML in PHP eingebettet, sondern umgekehrt PHP-Code mit Hilfe spezieller Tags („<?php“, „<?“ sowie „?>“) in HTML.

Abbildung 4.1 zeigt ein „Hello World“-Beispiel implementiert in Perl und PHP.

Perl:

```
#!/usr/bin/perl
print "<HTML><BODY>";
print "<H1>Hallo Welt</H1>";
print "</BODY></HTML>";
```

PHP:

```
<HTML><BODY>
<? echo "<H1>Hallo Welt</H1>"; ?>
</BODY></HTML>
```

Abbildung 4.1: Perl und PHP CGI-Beispiele

Hauptproblem von CGI-Anwendungen ist, dass für jede Anfrage ein eigener Prozess gestartet werden muss. Daher gibt es für viele Webserver Erweiterungen für Perl und PHP, so dass der Interpreter genau einmal innerhalb des Webservers existiert und über mehrere Anfragen wiederverwendet wird (je nach Erweiterung wird auch der Bytecode zwischengespeichert).

Der letzte Vertreter, der in diesem Rahmen betrachtet werden soll, ist Java. Auch wenn die Sun Java-Implementation nicht Open Source ist, gibt es inzwischen eine kompatible Open Source Implementation namens OpenJDK. Sun spezifizierte so genannte Java-Servlets ([3]). Dabei handelt es sich um spezielle Klassen, deren Instanzen innerhalb der Webserver VM HTTP-Anfragen bearbeiten. Darüber hinaus muss der Webserver

[2]Skriptsprachen werden interpretiert, die Kompilierung nach Änderungen entfällt

weitere spezifizierte Funktionen erbringen (z. B. Sessions, Zugriff auf Details der HTTP-Anfragen sowie auf die CGI-Parameter). Die Servlet-Schnittstelle verhält sich ähnlich wie das CGI, aber es werden hier keine Prozesse gestartet. HTML-Ausgaben werden wie bei Perl in den Programmcode eingebettet (siehe „Hello World“-Beispiel in Abbildung 4.2). Kurze Zeit später spezifizierte Sun die so genannten JavaServer Pages (JSP), die weniger fundierte Java-Kenntnisse erfordern und somit den Fokus wieder in Richtung HTML-Design lenken: Ähnlich wie bei PHP werden Java-Code-Fragmente bzw. spezielle JSP-Kommandos in HTML-Seiten eingebettet. Die Webseiten werden dann beim Zugriff (automatisch) transparent in Servlets umgewandelt. Bei der Entwicklung von Servlets bzw. JSP kann auf sämtliche verfügbaren Java-Ressourcen (JDBC, Java Naming and Directory Interface (JNDI), ...) zurückgegriffen werden. Diese Technologie setzt einen speziellen Java-fähigen Webserver voraus, der die Servlet-Spezifikation implementiert.

```
public class TestServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
                  throws IOException, ServletException {
    response.setContentType("text/html");
    PrintWriter out = servletResponse.getWriter();
    out.println(<HTML><BODY>");
    out.println(<H1>Hallo Welt</H1>");
    out.println(</BODY></HTML>");
  }
}
```

Abbildung 4.2: Servlet-Beispiel

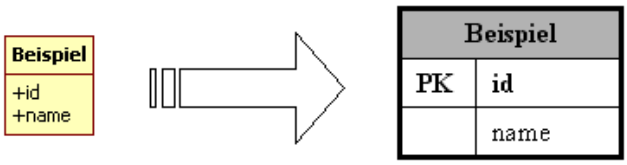
Für den Browser auf der Clientseite hat die Auswahl der Sprache keine Auswirkungen, da er mit dem Webserver über HTTP kommuniziert und der Webserver die Anfragebehandlung (auf der Serverseite) kapselt.

Obwohl die Technologien vom Funktionsumfang für dynamische Web-Anwendungen vergleichbar sind, wird Java bevorzugt: Für die funktionalen Anforderungen (Kompilierungsprüfung und Funktionstests) muss auf dem Webserver ohnehin Java vorhanden sein. Java ist im Gegensatz zu Perl und PHP strikt objektorientiert. Zudem existiert für Java (ohne Berücksichtigung externer Tools) ein Dokumentations-Generator (JavaDoc). Ferner ist die Datenhaltungs-Abstraktion Hibernate für Java verfügbar, die Duplikat-Erkennung Plaggie ist in Java verfasst und beide lassen sich so direkt integrieren.

4.2.2 Datenbankanbindung und Abstraktion mit Hibernate

Durch die Verwendung des Hibernate Persistenz-Frameworks kann eine Abstraktion der Datenhaltung geschaffen werden. Man modelliert ein Klassenmodell, welches durch Hibernate auf ein relationales Datenbankmodell abgebildet wird (ORM). Das System kann objektorientiert entwickelt werden, obwohl es auf einer relationalen Datenbank basiert.

Mit Hibernate ist es damit möglich, den Zustand von Plain Old Java Objects (POJO) sowie deren Beziehungen in einer relationalen Datenbank abzulegen und aus den Daten der Datenbank zu rekonstruieren, ohne dass man als Programmierer explizit die Structured Query Language (SQL) benutzen muss (siehe [31]).




```
@Entity
@Table(name = "beispiel") // Tabellenname
public class Beispiel {
    private int id;
    private String name;

    @Id // Primary Key
    @GeneratedValue // Autoincrement
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }

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

Abbildung 4.3: Funktionsweise des objektrelationalen Mappings

Klassen werden dabei auf (meist gleichnamige) Tabellen mit den Attributsnamen als Spalten abgebildet (Abbildung 4.3). Referenzen auf andere Klassen über Attribute werden (automatisch) durch Verknüpfungstabellen realisiert. Einstellungen für das Mapping bzw. die Abbildung, insbesondere für 1:1-, n:m- oder 1:n-Beziehungen, werden über Annotationen vorgenommen.

Abfragen an die Datenbank werden mit einer speziellen API oder in einer eigenen Sprache Hibernate Query Language (HQL) formuliert. Dies hat den Vorteil, dass man nicht an einen speziellen SQL-Dialekt einer Datenbank gebunden ist, sondern die tatsächlich verwendete Datenbank (im Rahmen der durch Hibernate unterstützten Datenbanken) austauschen kann.

4.3 Sicherheitsaspekte

4.3.1 HTTP

Bei HTTP handelt es sich um ein sehr einfaches Protokoll auf ASCII-Basis. Sämtliche Daten inklusive der Benutzernamen, Passwörter und Sitzungskennungen (Cookies) werden unverschlüsselt zwischen dem Server und dem Client ausgetauscht. Abhilfe kann hier durch die Verwendung von HyperText Transfer Protocol Secure (HTTPS) geschaffen werden. Dabei wird HTTP über eine symmetrisch verschlüsselte Verbindung auf Basis von Secure Sockets Layer (SSL)/Transport Layer Security (TLS) von Ende-zu-Ende getunnelt.

Der alleinige Einsatz einer (sicheren) Verschlüsselung, wie sie nach heutigen Maßstäben durch HTTPS erreicht werden kann, garantiert jedoch noch keine Abhörsicherheit und Integrität der Daten. SSL bzw. TLS sind anfällig für sog. Man-In-The-Middle-Angriffe (MITM): Ein Angreifer schaltet sich zwischen den Client und den echten Server. Folglich werden die Daten zwischen Server und Angreifer sowie zwischen Angreifer und Client verschlüsselt (vgl. Abbildung 4.4).

Abbildung 4.4: Veranschaulichung eines Man-In-The-Middle-Angriffs

Die Daten liegen dem Angreifer im Klartext vor. Als Gegenmaßnahme muss der Server ein durch eine vertrauensvolle Certification Authority (CA) unterschriebenes Zertifikat verwenden. Erst in dieser Kombination kann der Client eine sichere verschlüsselte Datenverbindung aufbauen und sicherstellen, dass er mit dem richtigen Server kommuniziert.

Hierzu muss in der Tomcat-Konfiguration „server.xml“ die SSL-Fähigkeit mit dem folgenden Eintrag aktiviert werden:

```
<Connector port="443" SSLEnabled="true" maxThreads="150" scheme="https"
           secure="true" keystoreFile="filename.keystore"
           sslProtocol="TLS" clientAuth="false" />
```

Eine vergleichbare Technik (auch auf SSL/TLS basierend) existiert auch für LDAP sowie viele Datenbankprotokolle und sollte bei Datenaustausch in unsicheren Netzen verwendet werden.

4.3.2 Dynamische Seitengenerierung

Dynamische Webseiten nehmen oftmals Eingaben (z. B. in Form von HTTP-GET/POST-Parametern) entgegen, um damit z. B. Befehle zu konstruieren, welche die Seitengenerierung bzw. den Seiteninhalt beeinflussen. Hier kann es bei Manipulation und unsachgemäßer Handhabung der Parameter zu SQL-Injection oder Cross-Site-Scripting (XSS) kommen (vgl. [32, Kapitel 4]).

Für Datenbankabfragen wird meist SQL benutzt, wobei die Anfragen dynamisch konstruiert werden und i. d. R. auch Eingaben vom Benutzer enthalten, der z. B. eine bestimmte Seiten-ID abrufen möchte. Werden die Benutzereingaben hier ohne Prüfung direkt in die SQL-Anfrage eingebaut, besteht die Gefahr, dass ein Angreifer eigenen SQL-Code an die Datenbank übermitteln kann. Verwundbar ist beispielsweise folgende Anfrage:

```
"SELECT * FROM users WHERE username = '" + username + "'
 AND passwort = '" + passwort + "';"
```

Ist es einem Angreifer möglich, das Passwort „' OR 1=1 --“ zu übermitteln, kann er sich quasi als beliebiger Benutzer einloggen (das Apostroph schließt den String, 1=1

ist immer wahr und „--“ leitet einen Kommentar in SQL ein, so dass der Rest der Original SQL-Abfrage ignoriert wird). Schutz bietet nur das Maskieren („Escaping“ genannt) von Metazeichen (im Beispiel „’“) in Benutzereingaben bzw. das Verwenden von Prepared Statements (dabei werden die Variablen getrennt von der SQL-Anfrage an die Datenbank gereicht). Auch wenn nicht direkt SQL eingesetzt wird (sondern z. B. HQL) ist die Problematik dort gleich.

Bei XSS wird versucht, eigene Inhalte in eine dynamisch generierte Webseite einzubringen. Dabei kann es sich z. B. um JavaScript handeln, wodurch auf einer vermeintlich vertrauenswürdigen Webseite (Zugangs-)Daten (oder Cookies) ausspioniert werden können. Auftreten kann dieses Problem, wenn Benutzereingaben (persistent oder nicht-persistent) ungefiltert in Ausgaben eingebaut werden (meist bei Suchanfragen oder Fehlermeldungen):

```
<p>Sie suchten nach {Suchbegriff q}</p>
```

Im nicht-persistenten Fall wird ein HTTP-Parameter beispielsweise wie folgt gesetzt:

```
http://domain.tld/suche?q=<script>alert('XSS');</script>
```

Die Belegung des Parameters *q* wird ungefiltert in der Antwort ausgegeben und das Skript im Browser ausgeführt:

```
<p>Sie suchten nach <script>alert('XSS');</script></p>
```

Im persistenten Fall werden die Eingaben erst (korrekt und unverändert) in einer Datenbank gespeichert und werden bei späteren Abfragen (auch ohne HTTP-Parameter) unverändert in die entsprechende Seite eingebaut.

Benutzer können den Uniform Resource Locator (URL) bzw. Parameter allgemein beliebig verändern und somit auch unerwartete Anfragen an die Web-Applikation stellen. Als Faustregel gilt: Man darf Benutzereingaben niemals vertrauen und muss diese immer erst überprüfen und bereinigen. Dieses Vorgehen wird Sanitizing genannt.

4.3.3 Automatische Tests

Allgemein problematisch ist das Ausführen von fremdem Programmcode in einer produktiven Umgebung, da das ausgeführte Programm mit den Rechten des Aufrufers (im Fall des Abgabesystems mit den Rechten des Webservers) ausgestattet wird und in diesem Rahmen auch Schadcode zur Ausführung bringen kann. Vorstellbar wäre hier das Löschen von Dateien, das Verändern von Daten (z. B. Punktestände) oder im ungünstigsten Fall die Übernahme des ganzen Servers (vgl. [5]).

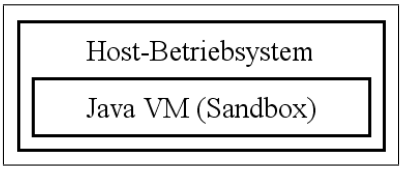


Abbildung 4.5: Die Java Virtual Machine als Sandbox

Begegnen kann man diesem Problem, indem das Programm in einer abgeschirmten Umgebung (Sandbox) auf dem Hostsystem ausgeführt wird (vgl. Abbildung 4.5). Java läuft designbedingt in einer virtuellen Maschine und nicht direkt auf der „Hardware“, um u. a. plattformunabhängig zu sein. Bereits in den ersten Versionen wurde viel Wert auf Sicherheit gelegt, aber erst seit Version 1.2 (Java 2) ist es möglich, sehr detailliert Rechte (Permissions[3]) für Java-Applikationen zu vergeben (vgl. [33]), die durch den Security-Manager durchgesetzt werden.

Solange keine nativen Methoden verwendet werden, ist der Zugriff auf Systemressourcen (Dateisystem, Threads, Netzwerk, Prozesse, ...) nur über die Java-API möglich. Vor der Ausführung als „gefährlich“ eingeschätzter Operationen wird der Security-Manager befragt. Im Fall fehlender Rechte wird eine Exception geworfen und der Zugriff unterbunden.

Konfigurieren kann man die Permissions über sog. Policy-Dateien (speziell formatierte ASCII-Dateien). Im Rahmen dieser Arbeit wird eine sehr restriktive Policy verwendet, die jegliche „gefährlichen“ Operationen unterbindet, aber dennoch, ähnlich wie in [5] beschrieben, in einem temporären Testverzeichnis auch das Schreiben und Löschen erlaubt:

[3] Liste der Std. Permissions: http://java.sun.com/j2se/1.4.2/docs/guide/security/permissions.html

```
grant {
permission java.util.PropertyPermission "*", "read";
permission java.io.FilePermission "file:TEMPDIR", "read, write, delete";
permission java.lang.RuntimePermission "accessDeclaredMembers";
};
```

Der Security-Manager wird über eine spezielle Option des Java-Interpreters auf der Kommandozeile aktiviert und auf eine bestimmte Policy-Datei festgelegt:

```
-Djava.security.manager -Djava.security.policy=myPolicy.policy
```

Versucht ein Java-Programm z. B. auf eine Datei zuzugreifen, für die keine Permissions vorliegen, so bricht die VM die Ausführung mit einer Exception ab:

```
public class Beispiel {
  public static void main(String args[]) {
    new File("/test").delete();
  }
}
```

```
Exception in thread "main" java.security.AccessControlException: access denied
                           (java.io.FilePermission test.file delete)
        at java.security.AccessControlContext.checkPermission(Unknown Source)
        at java.security.AccessController.checkPermission(Unknown Source)
        at java.lang.SecurityManager.checkPermission(Unknown Source)
        at java.lang.SecurityManager.checkWrite(Unknown Source)
        at java.io.File.deleteFile(Unknown Source)
        at Test.main(Test.java:4)
```

5 Ergebnis

5.1 Bewertung der Duplikat-Erkennung

Nach Implementierung bzw. Integration der im Abschnitt 4.1 ausgewählten Algorithmen wurden sie zur Überprüfung der Annahmen mit Einsendungen eines Programmierkurses für Informatiker getestet. Grundlage waren vier Aufgaben mit jeweils ca. 25 Lösungen. Dabei handelte es sich um drei sehr einfache Aufgaben (while-Schleifen und switch-case-Blöcke; vgl. Abbildungen 5.1 bis 5.5) und um eine anspruchsvollere Aufgabe, bei der nur die Signatur bzw. das Ergebnis einer Methode, nicht aber deren genauer Algorithmus, vorgeschrieben war.

Die Annahme aus Abschnitt 3.2, dass die auf Java-Quellcode optimierte Plaggie-Erkennung hauptsächlich 100 % Ähnlichkeiten zwischen verschiedenen Lösungen bei den einfachen Aufgaben errechnet, wurde bis auf wenige Ausnahmen bestätigt. Als Beispiel für eine 100 % Ähnlichkeit seien hier die Algorithmen aus den Abbildungen 5.3 und 5.4 hervorzuheben. Diese beiden Algorithmen haben keine 100 % Ähnlichkeit mit den Algorithmen aus den Abbildungen 5.1, 5.2 oder 5.5. Jedoch wurden Lösungen, die eine if-Bedingung innerhalb der While-Schleife wie 5.2 beinhalteten, als 100 % ähnlich zu dieser erkannt; ebenso bei Einsendungen, die wie 5.1 den Wert $0,5$ in eine weitere Variable ausgelagert haben. Einsendungen ohne automatisch ermittelte Ähnlichkeit waren Lösungen mit einzigartigen Lösungswegen (es gab nur eine Abgabe, welche die Zeile „`System.out.println("Zero!");`“ nach der While-Schleife in eine unnötige if-Abfrage schachtelte) oder fehlerhafte Implementierungen (jedoch gab es auch dort 100 % Ähnlichkeiten). Eine besondere Ausnahme ist in Abbildung 5.5 dargestellt. Dabei handelt es sich um eine Abgabe, welche zu keiner weiteren Einsendung eine Ähnlichkeit aufweist. Dies ist sicherlich darauf zurückzuführen, dass ein komplett falscher Algorithmus implementiert wurde. Die linke obere Tabelle in Abbildung 5.6 zeigt die genannten Ergebnisse.

```
public class SimpleWhileLoop {
  /**
    * @param args
    */
  public static void main(String[] args) {
    // TODO Auto-generated method stub
    int result=0;
    float sub= (float)0.5;
    while(result!=0){
      result=result--;
      System.out.println(result---sub);
    }
    System.out.println("Zero!");
  }
}
```

Abbildung 5.1: Abgabe-Beispiel 1

```
public class SimpleWhileLoop {
  /**
   * @param args
   */
  public static void main(String[] args) {
    // TODO Auto-generated method stub
    double i = 10;
    System.out.println("Aufgabe 4.a)\nDie Zahlen von 10 (...)");
    while (i > 0) {
      System.out.println(i);
      i -= 0.5;
      if (i == 0)
        System.out.println("\"ZERO!\"");
    }
  }
}
```

Abbildung 5.2: Abgabe-Beispiel 2

```
public class SimpleWhileLoop {
  /**
   * @param args
   */
  public static void main(String[] args) {
    // TODO Auto-generated method stub
    double i = 10.0;
    while(i>=0.5) {
    System.out.println(i);
    i-=0.5;
    }
    System.out.println("Zero");
  }

}
```

Abbildung 5.3: Abgabe-Beispiel 3

```
public class SimpleWhileLoop {
  public static void main(String[] args) {
    float i = 10f;
    while (i>= 0.5){
      System.out.println(i);
      i=i-0.5f;
    }
    System.out.println("Zero!");
  }
}
```

Abbildung 5.4: Abgabe-Beispiel 4

```
public class SimpleWhileLoop {
  /**
   */
  public static void main(String[] args) {
    // TODO Auto-generated method stub
    int zahl =0 , summe=10 , i = 10;
    while (i<=zahl){
      summe +=i;
    }
System.out.println("summe 10 bis" + zahl+":"+summe);
  }

}
```

Abbildung 5.5: Abgabe-Beispiel 5

Plaggie

	5.1	5.2	5.3	5.4	5.5
5.1		0	0	0	0
5.2	0		0	0	0
5.3	0	0		100	0
5.4	0	0	100		0
5.5	0	0	0	0	

Levenshtein und Kompressions-Distanz ohne Normalisierung

	5.1	5.2	5.3	5.4	5.5
5.1		60 55	74 68	54 0	61 63
5.2	60 55		65 58	0 0	50 53
5.3	74 68	65 58		68 58	59 67
5.4	54 0	0 0	68 58		51 0
5.5	61 63	50 53	59 67	51 0	

Levenshtein und Kompressions-Distanz auf Code-Basis ohne Normalisierung

	5.1	5.2	5.3	5.4	5.5
5.1		53 0	68 62	67 61	59 60
5.2	53 0		58 50	55 50	0 50
5.3	68 62	58 50		87 76	58 62
5.4	67 61	55 50	87 76		57 59
5.5	59 60	0 50	58 62	57 59	

Levenshtein und Kompressions-Distanz auf Code-Basis komplette Normalisierung

	5.1	5.2	5.3	5.4	5.5
5.1		52 0	68 61	69 64	59 59
5.2	52 0		58 52	55 51	0 0
5.3	68 61	58 52		90 78	59 59
5.4	69 64	55 51	90 78		58 61
5.5	59 59	0 0	59 59	58 61	

Abbildung 5.6: Tabellarische Ergebnisübersicht der implementierten Duplikat-Erkennungsmethoden auf Basis der Beispiel-Abgaben

Bessere Ergebnisse erzielten hier die Kompressions- und die Levenshtein-Distanz, welche je nach Normalisierungsgrad und Datenbasis eine differenziertere Sicht boten: Ohne Normalisierung bewegten sich die maximalen Ähnlichkeiten für alle Dateien im Rahmen von 50 bis 90 %. Mit kompletter Normalisierung verschob sich die untere Grenze auf etwa 65 %. Insbesondere gab es bei den Testdaten zwei Einsendungen, bei denen die Ähnlichkeit ohne Normalisierung bei ca. 75 % lag und bei kompletter Normalisierung auf 100 % stieg (zurückzuführen auf Veränderungen der Einrückungen; bestätigtes Plagiat). Bei Beschränkung auf den Quellcode ohne Kommentare ergaben sich für die Analyse ohne Normalisierung maximale Ähnlichkeiten im Bereich von 55 bis 91 % bei der Kompressions- und 65 bis 96 % bei der Levenshtein-Distanz (ohne das bestätigte Plagiat). Bei kompletter Normalisierung erhöhte sich bei beiden Metriken die maximale Ähnlichkeit nur noch sehr geringfügig (um ca. 1 %), dagegen aber die untere Schranke um mehrere Prozentpunkte.

Die konkreten Werte für die in diesem Abschnitt dargestellten fünf Beispiel-Einsendungen sind den Tabellen aus Abbildung 5.6 zu entnehmen. Immer am unteren Rand der Ähnlichkeiten (ca. 50 bis 65 %) lag der Algorithmus aus Abbildung 5.5 (auch im Auswertungsergebnis aller 25 Einsendungen). Dieses Resultat basiert darauf, dass diese Abgabe trotz einer fehlerhaften Implementierung dennoch Übereinstimmungen mit den richtigen Lösungen beinhaltete (Klassendefinition und Name, Variablendefinition, While-Schleife und Ausgabe).

Alle anderen Einsendungen, die eine richtige Lösung implementierten, hatten bis auf eine Lösung mindestens eine maximale Ähnlichkeit von 69 % (der rechten unteren Tabelle in Abbildung 5.6 zu entnehmen). Bei der Ausnahme handelt es sich um die Abgabe aus Abbildung 5.2 mit einer maximalen Ähnlichkeit von 58 %, wobei die Ursache auf die Wiederholung der Aufgabenstellung innerhalb des Algorithmus zurückzuführen ist. Neben dieser Vorgehensweise könnte auch eine Einstreuung von unnötigen Befehlen zur Verschleierung eines Plagiats verwandt werden.

Unabhängig davon war mit diesen beiden Metriken, anders als mit Plaggie, eine Differenzierung zwischen den Einsendungen aus den Abbildungen 5.3 und 5.4 möglich: Abbildung 5.6 zeigt, dass die berechnete Ähnlichkeit ohne Normalisierung ca. 68 % und bei der kompletten Normalisierung 90 % beträgt (mit der Levenshtein-Distanz). Unterschiede sind insbesondere bei den verwandten Datentypen und der Subtraktion zu finden. Die entscheidene Fragestellung ist hier (besonders bei derartig kurzen Algorithmen):

Welche Änderungen sind legitim und welche deuten auf eine Verschleierung hin. Nach [15] gibt es immer Wege, Plagiate zu verschleiern, jedoch sind die Autoren der Meinung, dass Änderungen/Verschleierungen, die das Verständnis der Programmiersprache und des Algorithmus voraussetzen, akzeptiert werden sollten. Als Begründung führen sie an, dass der Student in diesem Fall in der Lage wäre, die Aufgabe auch eigenständig zu lösen. Im Zweifel sollten die Studenten aufgefordert werden, ihr Programm zu erklären (vgl. [26]).

Bei der letzten, fortgeschritteneren Aufgabe zeigte Plaggie seine Überlegenheit und entdeckte zwei sehr ähnliche Algorithmen (96 % Ähnlichkeit, die von den Tutoren auch als Plagiat bestätigt wurden). Sowohl die Kompressions- als auch die Levenshtein-Distanz sind hier ohne Normalisierung ungeeignet: Es wurden ausschließlich zwei vermeintliche Plagiate mit ca. 53 % Ähnlichkeit gefunden. Der Grund hierfür war, dass die Studenten die Aufgabe falsch interpretiert und sehr kurze Lösungen eingereicht hatten. Lediglich mit der kompletten Normalisierung auf Code-Basis gab es noch akzeptable Ergebnisse (91 bzw. 93 % Ähnlichkeiten der beiden 96 % Plaggie-Übereinstimmungen; sowie weitere Ähnlichkeiten von 55 bis 82 %). Es ist hier aber ersichtlich, dass die beiden Distanzen eher für größere Gesamt-Ähnlichkeiten und weniger für Teil-Ähnlichkeiten, wie sie Plaggie erkennt, sinnvoll sind. Gäbe es in den Einsendungen mehrere (Instanz-)Methoden und wären diese Methoden vertauscht worden, hätte zumindest die Levenshtein-Distanz eine deutlich niedrigere Ähnlichkeit berechnet.

Bei Kommentaren als Basis für die Duplikat-Erkennung wurden teilweise sehr große Ähnlichkeiten berechnet, aber größtenteils bewegten sich die Wahrscheinlichkeiten eines Plagiats (über die verschiedenen Aufgaben hinweg) im gleichen Rahmen wie der Vergleich bei den vollständigen Dateien. Sehr gut ersichtlich wird hier eine mögliche Problematik: Kommentare sind meist sehr kurz und/oder bestehen aus automatisch generierten Eintragungen der Entwicklungsumgebung (vgl. Kommentare in 5.1, 5.2, 5.3 und 5.5). Beides führt jeweils dazu, dass es zu großen Ähnlichkeitswerten kommen kann, auch wenn die Algorithmen sich ziemlich unterscheiden (z. B. 5.1 und 5.4). Es wurde bei der Aufgabenstellung kein besonderer Wert auf Kommentare gelegt, folglich waren nur sehr wenige, kurze davon vorhanden. Der Vergleich von Kommentaren sollte weiter analysiert werden, insbesondere bei Aufgabenstellungen, in denen besonderer Wert auf Kommentierungen gelegt wird (letzteres ist ein Tipp von [26] zur Plagiatsverhinderung im Vorfeld).

Zudem hat sich gezeigt, dass sich die Kompressions-Distanz innerhalb des Tests sehr ähnlich zur Levenshtein-Distanz (im Allgemeinen $\pm 10\,\%$) verhalten hat, aber die berechneten Ähnlichkeiten meist geringer ausfielen (dies entspricht der Minorisierung der Kolmogorow-Komplexität).

Weiterhin können mit dem universalen und dem elementaren Ansatz (wie prinzipiell auch mit Plaggie) Verschleierungsversuche erkannt werden. Insbesondere bei kürzeren Aufgaben gibt es eine maximale Ähnlichkeit von durchschnittlich 65 bis 90 %. Ergeben sich für eine Lösung deutliche Abweichungen (wie z. B. die Abgabe aus Abbildung 5.2 durch die Wiederholung der Ausgabenstellung, vgl. Zeilen von 5.2 mit niedrigen Werten und vielen Nullen in Abbildung 5.6), könnte dies auf eine Verschleierung hindeuten (dieser und weitere Aspekte in [15]). Selbst eine hohe errechnete Ähnlichkeit muss nicht zwangsläufig auf ein Plagiat hindeuten: Ursache kann auch eine zu stark festgelegte Aufgabenstellung oder eine zu hohe Normalisierung/Abstraktion der Quelldaten sein. In jedem Falle ist eine Bewertung mit viel Augenmaß durch einen Tutor bzw. Betreuer (evtl. nach Rücksprache mit dem einsendenden Studenten) notwendig ([26] gibt praktische Hinweise für diesen Fall).

Wie bereits erwähnt, konnte diese Überprüfung nur auf einer beschränkten Datenbasis durchgeführt werden, jedoch ist zu vermuten, dass die Ergebnisse auch bei einer umfangreicheren Evaluation bestätigt werden. Weitere Untersuchungen sind hier anzuraten.

5.2 Bewertung der Funktionstests

Die automatischen Funktionstests wurden mit den gleichen Daten des vorherigen Abschnitts evaluiert.

Bei der in Abschnitt 3.2 vorgestellten Aufgabe haben von 26 Einsendungen 12 den automatischen Funktionstest, basierend auf einem regulären Ausdruck, nicht bestanden. Bei näherer Betrachtung fiel auf, dass acht Einsendungen die Aufgabenstellung nicht erfüllt haben (z. B. ein falscher Algorithmus in Abbildung 5.5, die Zahlen wurden nicht ausgegeben oder die Ausgabe „Zero!“ fehlte) und korrekt negativ klassifiziert wurden. Die verbliebenen vier Einsendungen haben zwar die Aufgabenstellung erfüllt, erzeugten jedoch eine abweichende Ausgabe. Oftmals fehlte das Ausrufezeichen am Ende. Auffällig war hier die Abgabe in Abbildung 5.2: Zusätzlich zu der geforderten Ausgabe wurde die Aufgabenstellung und „ZERO“ in Anführungszeichen ausgegeben.

In einer weiteren Aufgabe wurde ein Algorithmus als UML Aktivitätsdiagramm vorgestellt, das in Java übersetzt werden sollte. Am Ende des Programms stand eine Unterscheidung einer Zahl *Zahl* zwischen „Die Zahl *Zahl* ist eine fröhliche Zahl!“ und „Die Zahl *Zahl* ist eine traurige Zahl!“. Ein Funktionstest auf Basis eines regulären Ausdrucks bewertete 14 von 25 Einsendungen als fehlerhaft. Eine Untersuchung der Ergebnisse zeigte, dass 10 Einsendungen tatsächlich fehlerhaft umgesetzt wurden. Die anderen vier wurden auf Grund von Tipp- bzw. Kodierungsfehlern („oe“ statt „ö“), fehlender Leerzeichen zwischen der Zahl und dem Text (`System.out.println("Die Zahl" + args[0] + " ist eine fröhliche Zahl");`) oder fehlendem Ausrufezeichen nicht korrekt positiv klassifiziert.

Beide Tests enthielten keine falsch positiven Klassifizierungen.

Auf jeden Fall sollten die Studenten auf die Einhaltung der Ausgaben hingewiesen werden. Ferner sollte der reguläre Ausdruck bei Textausgaben Groß- und Kleinschreibung ignorieren, Leerschritte bei Variablen-Einfügungen großzügig behandeln und insbesondere bei Umlauten Alternativen (d. h. den Umlaut, als auch Umschreibungen wie „o“ oder „oe“ für „ö“) zulassen, um mögliche Fehlklassifizierungen zu minimieren.

Für den Großteil der Einsendungen hat die Klassifizierung im Rahmen des Tests gut funktioniert: Positiv getestete Lösungen müssen dadurch nicht mehr ausgiebig manuell (nach)geprüft werden. Natürlich muss weiterhin überprüft werden, ob Programm-Ausgaben, die nicht von Eingabeparametern abhängen, neu berechnet und nicht nur statisch ausgegeben werden. Dies könnte jedoch schon anhand großer Abweichungen zu allen anderen Einsendungen bei der Duplikat-Erkennung auffallen. Insbesondere wenn die Überprüfung auf Basis eines regulären Ausdrucks fehlschlagen sollte, kann der bearbeitende Tutor sowohl die generierte Ausgabe des Programms als auch den Quellcode einsehen, ohne das Programm herunterladen und lokal kompilieren zu müssen. Mit Hilfe dieser Daten wird eine mögliche Teil-Punkt-Vergabe erleichtert.

JUnit-Tests sind sicherlich auf Grund der strengen Typisierung, sofern nicht ein String (ähnlich wie bei den regulären Ausdrücken) überprüft wird, besser geeignet. Eine Evaluation konnte auf Grund fehlender Daten leider nicht durchgeführt werden.

5.3 Bewertung/Fazit

Im Rahmen dieser Bachelorarbeit sollte ein Online-Unterstützungsverfahren für Tutoren erstellt werden, welches auf ein einheitliches, abgestimmtes System aufsetzt. Die Abgabe, Duplikat-Erkennung, Funktionstests und schließlich die Bewertung der Arbeiten soll in einem einzigen System vorgenommen werden.

Dazu wurden verschiedene Duplikat-Erkennungs-Algorithmen und -Methoden vorgestellt und deren unterschiedliche Charakteristika beleuchtet. Schließlich wurden die drei Erfolg versprechendsten Algorithmen (mit unterschiedlichen Ansätzen, teilweise verändert) in das System integriert und mit Erfolg an Einsendungen von verschiedenen Studenten getestet. Jedoch wurde hier deutlich, dass es kein perfektes Duplikat-Erkennungssystem gibt bzw. geben kann. Insbesondere bei stark festgelegten oder einfachen Aufgaben ist es sehr schwer, Duplikate auszumachen — für den Tutor, wie auch für ein Erkennungssystem. In diesem Bereich hat sich deutlich gezeigt, dass eine manuelle Kontrolle unumgänglich und eine Entscheidung nur mit viel Augenmaß und evtl. Rücksprache mit den Studenten möglich ist.

Ein ähnliches Fazit kann für die automatischen Funktionstests gezogen werden: Sie können nicht zur Automatisierung des Bewertungsprozesses benutzt werden, sondern können den Tutor unterstützen und damit die Korrektur beschleunigen bzw. vereinfachen.

Es wurde ein erweiterbares, modulares, plattformunabhängiges, freies System mit aktuellen Technologien erstellt, welches die Anforderungen (vgl. Kapitel 2.1) erfüllt und der Zielsetzung (siehe Kapitel 1.1) gerecht wird. Es wurde eine Basis geschaffen, die Lehre bzw. den Prozess der Bewertung von Aufgaben zu verbessern.

5.4 Ausblick

Das entwickelte System unterstützt die Duplikat-Erkennung; die genauen Einstellungen der Normalisierung und die Erkennungslimits müssen für die unterschiedlichen Aufgaben(typen) jedoch noch evaluiert werden, um eine möglichst optimale Übersicht mit einer akzeptablen Falsch-Positive- und Falsch-Negative-Quote zu erhalten.

In der aktuellen Version des Systems beruht die Duplikat-Erkennung ausschließlich auf Quellcode-Basis. In der Regel fallen Plagiate auch durch eindeutige auffällige Fehler auf. Dadurch ergibt sich eine weitere Duplikat-Erkennungsmethode, die auf der Ausgabe des Compilers bzw. der Ausgabe des Funktionstests basiert. Wie gut sich eine solche Methode eignet, müsste geprüft werden.

Aus persönlicher Tutor Erfahrung des Autors sind Abschreiber Wiederholungstäter. Vorstellbar wäre daher eine Kennzeichnung von Abgaben eines Studenten, der bereits auffällig geworden ist. Die konkrete Umsetzung und der Nutzen dieser Information muss in der Praxis erst noch erprobt werden.

Literaturverzeichnis

1 SUN MICROSYSTEMS: *The Java Database Connectivity (JDBC)*. – URL `http://java.sun.com/javase/technologies/database/`

2 SUN MICROSYSTEMS: *Java Naming and Directory Interface (JNDI)*. – URL `http://java.sun.com/products/jndi/`

3 MURRAY, Gregory ; YOSHIDA, Yutaka: *Java(tm) Servlet Specification Version 2.4*. 2003. – URL `http://jcp.org/aboutJava/communityprocess/final/jsr154/index.html`

4 O. A.: *V-Modell XT 1.3 Dokumentation*. 2006. – URL `http://ftp.tu-clausthal.de/pub/institute/informatik/v-modell-xt/Releases/1.3/V-Modell-XT-Gesamt.pdf`. – Zugriffsdatum: 27. Dezember 2008

5 REEK, Kenneth A.: The TRY system -or- how to avoid testing student programs. In: *SIGCSE Bull.* 21 (1989), Nr. 1, S. 112–116. – ISSN 0097-8418

6 BALZERT, Heide: *Lehrbuch der Objektmodellierung. Analyse und Entwurf*. Heidelberg : Spektrum, 1999. – ISBN 3-8274-0285-9

7 GAMMA, Erich: *JUnit - A Cook's Tour*. 2009. – URL `http://junit.sourceforge.net/junit3.8.1/doc/cookstour/cookstour.htm`. – Zugriffsdatum: 2. Juni 2009

8 GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994. – ISBN 978-0-20163-361-0

9 COAD, Peter ; NORTH, David ; MAYFIELD, Mark: *Object Models: Strategies, Patterns, and Applications*. Prentice Hall, 1996

10 VERCO, Kristina L. ; WISE, Michael J.: Software for Detecting Suspected Plagiarism: Comparing Structure and Attribute-Counting Systems. In: *Proc. of 1st Australian Conference on Computer Science Education*, ACM, 1996, S. 86–95

11 PRECHELT, L. ; MALPOHL, G. ; PHILIPPSEN, M.: Finding Plagiarisms among a Set of Programs with JPlag. In: *Journal of Universal Computer Science* 8 (2002), Nr. 11, S. 1016–1038. – URL `http://www.jucs.org/jucs_8_11/finding_plagiarisms_among_a`

12 WISE, Michael J.: YAP3: improved detection of similarities in computer program and other texts. In: *SIGCSE Bull.* 28 (1996), Nr. 1, S. 130–134. – ISSN 0097-8418

13 BRENT, Xin C. ; CHEN, Xin ; FRANCIA, Brent ; LI, Ming ; MCKINNON, Brian ; SEKER, Amit: Shared Information and Program Plagiarism Detection. In: *IEEE Trans. Inform. Th* 50 (2003), S. 1545–1551

14 JOY, Mike ; LUCK, Michael: Plagiarism in Programming Assignments. Coventry, UK, UK : University of Warwick, 1998. – Forschungsbericht

15 GRUNE, Dick ; HUNTJENS, Matty: Detecting copied submissions in computer science workshops. (1989). – URL `http://www.cs.vu.nl/~dick/sim.html`. – Zugriffsdatum: 8. Juni 2009

16 GITCHELL, David ; TRAN, Nicholas: Sim: a utility for detecting similarity in computer programs. In: *SIGCSE Bull.* 31 (1999), Nr. 1, S. 266–270. – ISSN 0097-8418

17 SCHLEIMER, Saul ; WILKERSON, Daniel S. ; AIKEN, Alex: Winnowing: local algorithms for document fingerprinting. In: *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. New York, NY, USA : ACM, 2003, S. 76–85. – ISBN 1-58113-634-X

18 AHTIAINEN, Aleksi ; SURAKKA, Sami ; RAHIKAINEN, Mikko: Plaggie: GNU-licensed source code plagiarism detection engine for Java exercises. In: *Baltic Sea '06: Proceedings of the 6th Baltic Sea conference on Computing education research*. New York, NY, USA : ACM, 2006, S. 141–142

19 LI, Ming ; CHEN, Xin ; LI, Xin ; MA, Bin ; VITÁNYI, Paul: The similarity metric. In: *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA : Society for Industrial and Applied Mathematics, 2003, S. 863–872. – ISBN 0-89871-538-5

20 CILIBRASI, Rudi ; VITANYI, Paul: Clustering by Compression. In: *IEEE Transactions on Information Theory* 51, S. 1523–1545

21 JBOSS: *Hibernate*. – URL `http://www.hibernate.org`

22 SUN MICROSYSTEMS: *Java Persistence API*. – URL `http://java.sun.com/javaee/technologies/persistence.jsp`

23 LAHRES, Bernhard ; RAYMAN, Gregor: *Praxisbuch Objektorientierung. Professionelle Entwurfsverfahren*. Galileo Computing, 2006. – URL `http://openbook.galileodesign.de/oo/`. – ISBN 978-3-89842-624-4

24 ERLENKÖTTER, Helmut: *X/HTML. Flexible Webseiten von Anfang an*. Rowohlt Taschenbuch Verlag, 2004. – ISBN 978-3-49961-248-0

25 JOURAVLEV, Michael: *Redirect After Post*. 2004. – URL `http://www.theserverside.com/tt/articles/article.tss?l=RedirectAfterPost`. – Zugriffsdatum: 1. Juni 2009

26 WAGNER, N.: *Plagiarism by Student Programmers*. – URL `http://www.cs.utsa.edu/~wagner/pubs/plagiarism0.html`. – Zugriffsdatum: 8. Juni 2009

27 LANCASTER, Thomas ; CULWIN, Fintan: A Comparison of Source Code Plagiarism Detection Engines. In: *Computer Science Education* 14 (2004), Nr. 2, S. 101–117

28 MORSE, Kingsley G.: Compression tools compared. In: *Linux J.* 2005 (2005), Nr. 137, S. 3. – URL `http://www.linuxjournal.com/article/8051`. – ISSN 1075-3583

29 SUN MICROSYSTEMS: *MySQL 5.1*. – URL `http://www.mysql.com`

30 APACHE SOFTWARE FOUNDATION: *Apache Tomcat 6.0*. – URL `http://tomcat.apache.org`

31 MINTER, Dave ; LINWOOD, Jeff: *Pro Hibernate 3*. Apress, 2005. – ISBN 978-1-59059-511-4

32 CHESWICK, William R. ; BELLOVIN, Steven M. ; RUBIN, Aviel D.: *Firewalls and Internet Security; Repelling the Wily Hacker. Second Edition*. Reading, MA : Addison-Wesley, 2003

33 GONG, Li ; MUELLER, Marianne ; PRAFULLCHANDRA, Hemma ; SCHEMERS, Roland: *Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2*. 1997

Anhang

Anhang 1: Ehrenwörtliche Erklärung

E R K L Ä R U N G

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäßaus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind, und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Ort, Datum Unterschrift der Kandidatin/des Kandidaten