Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 8

Verteiltes Modellrepository für TOSCA

Kai Mindermann

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Frank Leymann
Betreuer/in:	Dipl.-Inf. Oliver Kopp
Beginn am:	1. Juni 2012
Beendet am:	1. Dezember 2012
CR-Nummer:	H.3.2, H.3.4, H.3.5, H.5.2

Verteiltes Modellrepository für TOSCA

Kai Mindermann

Zusammenfassung—In dieser Bachelorarbeit untersuche ich die Möglichkeit, existierende Back-End Systeme, wie zum Beispiel Datenbank Systeme (DBS), durch verteilte Versionsverwaltungssysteme (VVS) zu ersetzen. Dabei gehe ich auf die Anforderungen ein, die bei verteilter Arbeit wichtig sind und vergleiche verschiedene VVS miteinander. Im Weiteren stelle ich 3 unterschiedliche Ansätze, wie ein Back-End so mit einem VVS implementiert werden könnte, vor. Als konkrete Implementierung zeige ich dabei eine auf git aufbauende Zwischenschicht *GitWorkingTreeWatcher*, die einfach in vorhandene Anwendungen integriert werden kann. Diese ermöglicht es die in einem Ordner gespeicherten Dateien, automatisch mit git zu versionieren. Zusätzlich entwickle ich ein beispielhaftes Konzept wie ein versioniertes Repository im Visual Editor for TOSCA (VALESCA) dargestellt werden könnte und gehe darauf ein, wie die Architektur zur Integration eines versionierten Repositories, geändert werden sollte.

1 Einführung

Aufgrund der aktuell zu beobachtenden Entwicklungen ist es ist leicht vorherzusagen, dass auch in Zukunft immer mehr IT-Projekte auf Cloud-Technologien basieren werden. Um den kompletten Prozess der Entwicklung und besonders der Auslieferung zu vereinfachen und zu standardisieren, wurde Topology and Orchestration Specification for Cloud Applications (TOSCA) [1] entwickelt. Mit TOSCA lassen sich Cloud-Anwendungen modellieren und mithilfe eines TOSCA-Containers automatisch ausliefern und installieren. Zur Modellierung kann zur Zeit eine Weboberfläche, VALESCA [2], benutzt werden. Da es hierfür oft wichtig ist, dass mehrere Personen, auch gleichzeitig, modellieren können, muss das zugrunde liegende Datenmodell, auf dem die Weboberfläche basiert, dies unterstützen. Zur Zeit kommt zur Datenhaltung ein dateibasiertes Repository zum Einsatz. Dies erfordert, dass die Personen eine Verbindung zu diesem Repository haben, um arbeiten zu können. Wünschenswert ist ein Programm, bei dem man lokal Änderungen, auch offline, vornehmen kann, und sobald man zufrieden mit diesen Änderungen ist, sie erst dann zentral speichert. Um so ein Vorgehen zu Unterstützen gibt es verschiedene Möglichkeiten.

In dieser Arbeit wird dazu im Folgenden untersucht, wie man durch Nutzen verteilter Versionsverwaltungssysteme (VVS) die Datenhaltung so umsetzen kann, dass sowohl lokal, als auch auf einem zentralen Objekt gearbeitet werden kann. Als Erstes stelle ich relevante verwandte Arbeiten in Abschnitt 2 vor. Darauf folgend gehe ich in Abschnitt 3 zunächst auf verschiedene Grundlagen ein und erläutere die notwendigen Begrifflichkeiten die später benutzt werden. In Abschnitt 4 behandle ich VVS als Back-Ends. Hiernach gehe ich in Abschnitt 5 auf die Anforderungen an ein Back-End für VALESCA ein. Die vom Repository zu behandelnden Konflikte werden in Abschnitt 6 behandelt. In Abschnitt 7 untersuche ich verschiedene VVS. Nach der Evaluation folgt in Abschnitt 8 der Entwurf für die im darauffolgenden Abschnitt 9 beschriebene Implementierung mit git [3]. Im Abschnitt 10 werden die Ergebnisse zusammengefasst.

2 Verwandte Arbeiten

Tammo van Lessen [4] hat ein Repository für Geschäftsprozesse entwickelt. Er hatte ähnliche Anforderungen an sein Repository wie ich (siehe Abschnitt 5), verwendet als Back-End jedoch eine relationale Datenbank.

Craig Roberts [5] beschreibt eine Implementierung eines mit git versionierten Dateisystems in ownCloud [6]. Es wird darauf eingegangen wie mit PHP ein Dateisystem auf Basis von git erstellt werden kann. In der Einleitung wird auch das Problem der Behandlung von großen binären Dateien in VVS beschrieben, aber nicht näher darauf eingegangen. Eine nähere Diskussion wie mit Konflikten umgegangen wird, wird auf eine Mailingliste ausgelagert und somit nicht definiert.

Reilly Grant [7] stellt eine Dateisystemschnittstelle *figfs* zu git vor. Diese verwendet das Kernel-Modul Filesystem in Userspace (FUSE) um Zugriffe auf ein git-Repository als konkretes Dateisystem nahe der Betriebssystemebene anzubieten. Während die Arbeit zu *figfs* noch nicht abgeschlossen ist, kann man damit schon grundlegende Dateisystemoperationen durchführen. Interessant ist eine Verbesserung im Vergleich zu herkömmlichen Dateisystemen: Bei häufigen Zugriffen wird dort eine einfache „string lookup hash table“ vorgeschlagen, da git Objekte mit einem Secure Hash Algorithm (SHA)-1 Hash identifiziert. Dies kann schneller sein als, wenn bei Pfadzugriffen alle Elternordner auf Zugriff überprüft werden müssen.

Eine auf konkreten Fragestellungen basierende Evaluierung von vielen VVS findet man unter [8]. Dort werden verschiedene Fragen gestellt und diese für jedes untersuchte VVS in Stichworten oder 1-2 Sätzen beantwortet.

In Bezug auf die Behandlung von Konflikten durch gleichzeitiges Bearbeiten stellt Tancred Lindholm [9] einen Algorithmus *3DM* inkl. Beispiel-Implementierung vor. In seiner Masterarbeit geht er auf verschiedene Strategien, eXtended Markup Language (XML) Bäume zu vereinen, ein. Besonders hilfreich ist die Logging

Funktion, bei nicht lösbaren Konflikten bei der anhand des Logs die Entstehung der Konflikte klar wird. In Abschnitt 6 gehe ich näher auf seine Ergebnisse und Untersuchungen ein.

3 GRUNDLAGEN

Dieser Abschnitt erläutert wichtige Grundlagen, die in den folgenden Kapiteln der Ausarbeitung benötigt werden. Diejenigen die sich in den entsprechenden Bereichen schon auskennen können diese überspringen. Dennoch gehe ich hier speziell auf die Teilbereiche ein, welche für die nachfolgenden Abschnitte wichtig sind.

3.1 Cloud-Computing

Unter Cloud-Computing, auch nur Cloud (engl. Wolke), versteht man eine grobe Struktur verteilter Rechner die für unterschiedliche Dienstleistungen benutzt werden können. Nach einem Standard des National Institute of Standards and Technology (NIST) ist Cloud-Computing so definiert:

> Cloud Computing ist ein Modell, das es erlaubt bei Bedarf, jederzeit und überall bequem über ein Netz auf einen geteilten Pool von konfigurierbaren Rechnerressourcen (z.B. Netze, Server, Speichersysteme, Anwendungen und Dienste) zuzugreifen, die schnell und mit minimalen Managementaufwand oder geringer Serviceprovider-Interaktion zur Verfügung gestellt werden können. [10]

Der Begriff der Wolke kommt daher, dass für den Kunden einer Cloud-Diensleistung die dahinter liegenden technischen Infrastrukturen in einer Wolke verschwinden. Die Dienstleistungen reichen von bereitgestellten Netzwerken über Infrastruktur as a Service (IaaS), wie Rechenleistung und Speicherplatz, bis hin zu Platform as a Service (PaaS) und Software as a Service (SaaS) die in der Cloud betrieben wird. Mehr dazu [11].

3.2 Topology and Orchestration Specification for Cloud Applications (TOSCA)

TOSCA ist ein Standard für automatisierte Verwaltung von Cloud-Anwendungen. In dieser Arbeit beziehe ich mich auf Version WD13 der TOSCA-Spezifikation [1]. Beschrieben wird der Standard dort wie folgt:

> IT services (or just services in what follows) are the main asset within IT environments in general, and in cloud environments in particular. The advent of cloud computing suggests the utility of standards that enable the (semi-) automatic creation and management of services (a.k.a. service automation). These standards describe a service and how to manage it independent of the supplier creating the service and independent of any particular cloud provider and the technology hosting the service. Making service topologies (i.e. the individual components of a service and their relations) and their orchestration plans (i.e. the management procedures to create and modify a service) interoperable artifacts, enables their exchange between different environments. This specification explains how to define services in a portable and interoperable manner in a Service Template document. ([1])

Mit TOSCA sollen folgende Ziele erreicht werden [12]:

- Einfache und unabhängige Installation bei kompatiblen Clouds
- Einfachere Migration existierender Anwendungen in die Cloud
- Flexible Lastspitzenskalierung (engl. bursting)
- Dynamische Anwendungen, welche viele Cloud-Plattformen unterstützen

Die vom Standard spezifizierten zu erzeugenden Dokumente stelle ich in den folgenden Abschnitten vor. Als Dateiformat kommt XML zum Einsatz.

3.2.1 Definitions

Das zentrale Dokument *Definitions* enthält benötigte Informationen, um ein *ServiceTemplate* definieren zu können. Dafür enthält es neben dem *ServiceTemplate* selbst, folgende Elemente: *NodeType*, *NodeTypeImplementation RequirementType*, *Capability Type*, *Relationship Type*, *RelationshipTypeImplementation ArtifactType*, *ArtifactTemplate*, *PolicyType* und *PolicyTemplate*.

3.2.2 Service Template

Im *ServiceTemplate* wird nun die Cloud-Anwendung selbst spezifiziert. In diesem Element können alle im *Definitions*-Dokument definierten Typen verwendet werden. Ein vollständiges imperatives TOSCA *ServiceTemplate* besteht selbst aus *TopologyTemplate* und *Plänen*. Diese Elemente können genau wie im *Definitions*-Dokument direkt im *ServiceTemplate* definiert, aber auch per *import*-Angaben aus weiteren separat abgelegten XML-Dokumenten importiert werden.

Das *Topology Template* beschreibt die Topologie einer modellierten Cloud-Anwendung. Die Topologie besteht aus den *NodeTemplate*-Elementen, welche die verwendeten Knoten, basierend auf entsprechenden *NodeType*-Elementen, definieren, und den *RelationshipType*-Elementen, welche die Beziehungen zwischen Knoten definieren. Das *RequirementType*-Element legt Anforderungen fest. Im Gegensatz dazu definiert *CapabilityType* welche Fähigkeiten oder Funktionalität bereitgestellt werden kann. *ArtifactType* definiert Typen für auslieferbare Artefakte, konkrete Ausprägungen werden durch *Artifact Template* definiert. Im Element *Plans* werden verschiedene mögliche Pläne definiert. Ein Plan kombiniert Operationsaufrufe zu einer höherwertigen Operation. Beispielsweise könnte es die Pläne *start* und *stop* geben, welche festlegen was beim Starten und beim Beenden der Clound-Anwendung gemacht werden muss.

Um die XML-Dateien nicht direkt bearbeiten zu müssen, wird VALESCA entwickelt.

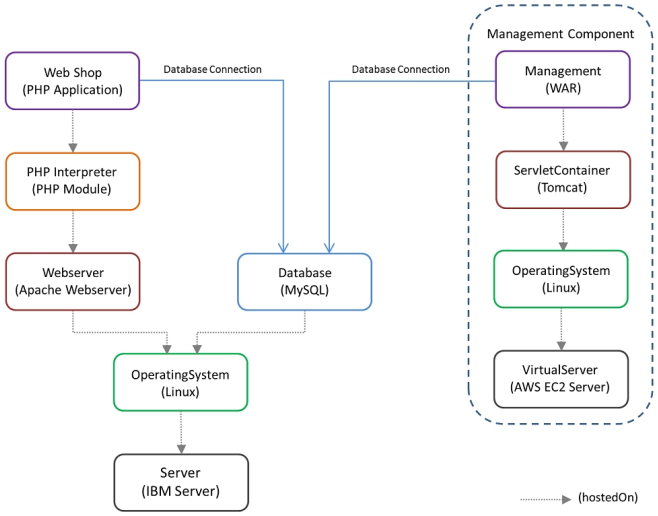

Abbildung 1. Topologie Ansicht in VALESCA. Beispiel für einen Webshop.

```
<namespace>
\--<name>
     |  ServiceTemplate.properties
     |
     \--topologytemplate
        |  parents.json
        |  TopologyTemplate.json
        |  TopologyTemplate.svg
        |
        +--grouptemplates
        |       grouptemplates.json
        |
        +--nodetemplates
        |       nodetemplates.json
        |
        \--relationshiptemplates
                realtionshiptemplates.json
```

Abbildung 2. Dateirepräsentation des Service-Templates im Ordner *servicetemplate*

3.3 Visual Editor for TOSCA (VALESCA)

Um die XML-Dateien zu spezifizieren, ist es möglich einen Editor mit graphischer Benutzungsoberfläche zu verwenden. Dieser wird eigens für TOSCA entwickelt. Dort kann die Topologie in einer Graph-Repräsentation bearbeitet werden.

3.3.1 Graphical User Interface (GUI)

VALESCA ist eine in Java und JavaScript geschriebene Webanwendung die über einen Browser bedient wird. In Abbildung 1 ist ein Webshop als Beispiel für die Topologie-Ansicht zu sehen. Eine nähere Beschreibung des GUI der Weboberfläche ist in Unterabschnitt 8.4 zu finden.

3.3.2 Dateirepräsentation des Service-Templates

Das Service-Template wird, wie in Abbildung 2 dargestellt, in Dateien unterhalb von *servicetemplate* gespeichert. Im Anhang ist in Abbildung 16 ein größeres Beispiel dargestellt.

Als Dateiformat werden Java properties-Dateien und die JavaScript Object Notation (JSON) [13] verwendet. Zusätzlich wird auch die grafische Repräsentation der Graphen in einer Scalable Vector Graphics (SVG)-Datei gespeichert, um eine Vorschau zu ermöglichen, ohne die Editor-Komponente laden zu müssen. Des Weiteren können noch beliebige andere Dateien, die in der Spezifikation verwendet oder benötigt werden, existieren. Dies sind häufig Binärdateien bei denen zwischen verschiedenen Versionen die Änderungen nicht einfach festgestellt werden können bzw. diese nicht so platzsparend wie bei Textdateien gespeichert werden können. Das ist später bei der Versionierung wichtig.

3.4 Repository

In einem Repository (engl. Lager), werden Daten verwaltet und gespeichert. Repository ist dabei nur eine Bezeichnung für den Ort an dem die jeweils von einem Programm zu speichernden Daten zu finden sind. Repositories können verschiedene Zwecke haben und auch unterschiedlich implementiert sein [14].

Beispielsweise spricht man im Rahmen der Versionsverwaltung (siehe Unterabschnitt 3.6) von einem Repository in dem die Versionen verwaltet und gespeichert werden. Genauso werden in Linux-Systemen häufig Repositories als Quelle für Software angegeben. Diese enthalten dann Daten, die zur Installation von der jeweiligen Software benötigt werden.

Ich verwende den Begriff *Repository* hier sowohl in den Grundlagen bei VVS, als auch später im Sinne von Speicher für die von VALESCA verwalteten Modelle.

3.5 Datenbank System (DBS)

Viele Programme müssen eine große Menge an Daten verarbeiten. Dabei soll es möglich sein, die Daten lesen sowie bearbeiten zu können. Da viele Programme ähnliche Anforderungen an die Verwaltung von Daten haben, wurden Datenbank Systeme (DBS) entwickelt. Diese bieten jeweils einheitliche Schnittstellen und Formate für das grundsätzliche Speichern und Verwalten der Daten. Programme greifen auf diese Schnittstellen zu, überlassen die Verwaltung aber dem DBS. Mit einem DBS ist es also möglich ein Repository, wie gerade in Unterabschnitt 3.4 definiert, zu implementieren.

3.5.1 Architektur

Ein DBS besteht aus einer Datenbank und einer Verwaltungskomponente, dem Datenbank Management System (DBMS). DBS können nur lokal, bzw. zentral auf einem einzigen Rechner laufen. Viele DBS sind aber auch dafür ausgelegt, auf mehreren Rechnern verteilt eingesetzt zu werden. Verteilt bedeutet hier aber nur, dass das DBS selbst verteilt arbeitet, um zum Beispiel mehr Festplatten

nutzen zu können die nicht in einen einzelnen Rechner passen würden oder auch für Replikation der Daten. Es ist aber nicht möglich, wie bei den in Unterabschnitt 3.6 beschriebenen, verteilten VVS, unterschiedliche Versionen der Daten vom DBS verwalten zu lassen. Als Speicherort für versionierte Objekte können DBS sehr gut dienen.

3.5.2 Ablagemöglichkeiten

Es gibt verschiedene Möglichkeiten die Daten in der Datenbank abzulegen [15]. Am häufigsten wird dazu ein **relationales** Datenbankmodell verwendet. Die Daten werden dabei in Tabellen gespeichert, welche in nahezu beliebigen Beziehungen zueinander stehen können. Populärer werden die **Dokumentorientierten** Datenbanken. Bei diesen werden komplette Dokumente, welche selber Daten in beliebigen Formen enthalten, in der Datenbank direkt gespeichert. Diese können auch zu den Not only SQL (NoSQL)-Datenbanken gezählt werden, welche ich hier aber nicht betrachte. Eine weniger populäre, aber trotzdem genutzte, Variante, ist die **objektorientierte** Datenbank. In diesen werden Objekte, im Sinne der Objektorientierung [16] inklusive ihrer Eigenschaften gespeichert. Die Datenbank verwaltet hier selbst die Objekteigenschaften wie Vererbungshierarchie und Identitäten.

3.5.3 Anforderungen

Datenbank Systeme (DBS) sollten Atomicity, Consistency, Isolation and Durability (ACID) vollständig unterstützen [17]:

- **Atomarität**: Zusammengehörende Operationen werden atomar ausgeführt. Das bedeutet entweder werden sie ganz oder, falls Teile der Operation nicht ausgeführt werden können oder ausgeführt werden konnten, gar nicht ausgeführt.
- **Konsistenz**: Jede ausgeführte Operation transformiert die verwalteten Daten von einem konsistenten wieder in einen konsistenten Zustand
- **Isolation**: Falls nebenläufig stattfindende Operationen erlaubt sind, beeinflussen diese sich nicht gegenseitig.
- **Haltbarkeit**: Nach einer erfolgreichen Operation sind die veränderten Daten sicher gespeichert. Sicher bedeuted, dass die Daten auch bei einem Systemausfall nicht mehr verloren gehen können, sich also in nicht flüchtigem Speicher befinden.

In diesem Zusammenhang möchte ich das von Eric Brewer aufgestellte CAP-Theorem [18] nennen. Dieses besagt, dass ein verteiltes System nicht in der Lage ist alle drei der folgenden Eigenschaften gleichzeitig zu erfüllen:

- **Konsistenz** (Consitency): Im Unterschied zu der Konsistenz von ACID ist hier Konsistenz auf alle am verteilten System beteiligten Knoten und die darauf gespeicherten Informationen bezogen. Diese müssen zu jedem Zeitpunkt, aus der Sicht jedes Knotens, gleich sein.
- **Verfügbarkeit** (Availability): Jede an das verteilte System gestellte Anfrage wird auch beantwortet.
- **Partitionstoleranz** (Partition tolerance): Das verteilte System funktioniert bei partiellen Ausfällen weiter.

Beispielsweise ist es für Cloud-Anwendungen oft ausreichend nach dem Basically Available, Soft State, Eventual consistency (BASE)-Prinzip zu arbeiten. Für diese sind Hochverfügbarkeit und Partitionstoleranz sehr wichtig, wohingegen es toleriert wird, wenn die gelieferten Daten inkonsistent bzw. noch nicht auf jedem beteiligten Knoten aktuell sind, dies aber nach möglichst kurzer Zeit wieder sind.

3.6 Versionsverwaltungssystem (VVS)

Bei der Entwicklung von Software ist es nötig, die erzeugten Daten, die oft hauptsächlich aus Quelltext und seiner Dokumentation bestehen, zu verwalten. Diese Verwaltung wird im großen Stil mittels Software Configuration Management (SCM) [19] Systemen realisiert. Der für diese Arbeit wichtigere Teil der Versionsverwaltung, wird von in SCM verwendeten VVS abgedeckt. Ich verwende in dieser Arbeit bewusst nicht den englischen Begriff Revision Control System (RCS) und Distributed Revision Control System (DRCS) weil ich den Leser durch die Verwendung des deutschen Begriffs darauf aufmerksam machen möchte, dass es um Versionsverwaltung geht.

VVS erfüllen folgende **grundsätzliche Anforderungen**:

- **Änderungsgeschichte**: Änderungen von verschiedenen Autoren an Daten werden protokolliert und sind für alle Beteiligten einsehbar.
- **Wiederherstellen**: Änderungen können rückgängig gemacht werden. Es kann zu früheren Entwicklungsständen zurückgekehrt werden.
- **Gleichzeitiges Bearbeiten**: Es wird dafür gesorgt, dass vorhergehende Änderungen, bei gleichzeitigem Bearbeiten eines versionierten Objekts, nicht verloren gehen.

Diese grundsätzlichen Anforderungen werden von vielen VVS durch zahlreiche zusätzliche Funktionen ergänzt. Unterschieden werden kann noch zwischen lokalen und zentralen VVS. Dabei findet bei der lokalen Versionsverwaltung die Versionierung oft in der Datei selbst statt. Die Änderungen werden im Dokument selber protokolliert und verwaltet. Bei der zentralen Versionsverwaltung muss es nicht zwangsläufig sein, dass das Repository auf dem lokalen Rechner liegt, es ist auch häufig so, dass es auf einem zentralen Server erreichbar ist.

3.6.1 Strategien für gleichzeitiges Bearbeiten

Um gleichzeitiges Bearbeiten von versionierten Daten zu gestatten, gibt es zwei verschiedene Ansätze [20] [21].

1) **Sperren**: Eine Möglichkeit ist es, die Dateien oder Daten ausschließlich für den Benutzer zu reservieren der eine Änderung vornehmen möchte. Dies bedeutet für andere Benutzer, dass wenn sie

dieselben Daten bearbeiten möchten, müssen sie warten bis der zur Zeit Änderungen durchführende Benutzer, seine Reservierung aufhebt bzw. vom VVS aufheben lässt. Dies ist häufig ein großer Nachteil, denn es kommt oft vor, dass an einer Datei mehrere Personen gleichzeitig arbeiten möchten und sollen. Zusätzlich kann es vorkommen, dass ein Benutzer vergisst, dass er bestimmte Daten für andere blockiert, wenn er anderen Aufgaben nachkommt. Diese Strategie wird auch als **pessimistische Versionsverwaltung** bezeichnet.

2) **Zusammenführen**: Die zweite Möglichkeit umgeht die durch das Sperren verursachten Probleme. Die Strategie besteht hier darin, dass jeder alle Daten editieren kann, wann er möchte. Aber sobald er die Änderungen dem VVS übergeben will, überprüft dieses ob zwischenzeitlich andere Änderungen gemacht wurden und versucht diese zu verschmelzen, falls das automatisch möglich ist und vom VVS unterstützt wird, oder fordert den Benutzer auf, seine Änderungen auf Basis der zwischenzeitlich gemachten Änderungen zu erneuern. Diese Strategie wird auch als **optimistische Versionsverwaltung** bezeichnet. Beim Zusammenführen gibt es zwei verschiedene Ansätze die im nächsten Unterunterabschnitt behandelt werden.

3.6.2 Zusammenführungsstrategien

Es existieren zwei verschiedenen Strategien dafür, wie man eine Zusammenführung durchführen kann (siehe Abbildung 3). Falls beim Zusammenführen nur die Unterschiede zwischen zwei Bäumen beachtet werden wird dies als *2-way merge* bezeichnet. Falls beim Zusammenführen auch die Informationen über den ursprünglich von beiden Änderungen veränderten Baum herangezogen werden, wird dies als *3-way merge* bezeichnet und wird von Tancred Lindholm wie folgt definiert:

> Assume T_1 and T_2 are ordered trees derived from the tree T_B. The 3-way merge of the trees T_1, T_B and T_2 is an ordered tree T_M, where the changes between T_B and T_1, as well as the changes between T_B and T_2 are incorporated. The tree T_B is called the base and the trees T_1 and T_2 are branches. ([9])

3.6.3 Probleme zentraler VVS

Die zentrale Herangehensweise, mit einem einzigen Repository, hat unter anderem aber folgende Probleme:

- Ohne Verbindung, zum Server bzw. dem zentralen Repository, ist es nicht möglich Änderungen zu protokollieren bzw. protokollieren zu lassen.
- Es gibt keine Möglichkeit, ohne ein zweites, mit zusätzlichem Synchronisationsaufwand verbundenen, Repository, Änderungen zu verwalten, welche nicht im Repository auftauchen sollen. (Dazu könnten beispielsweise Änderungen zählen, welche nicht kompiliert werden können oder unzureichend getestet sind.)

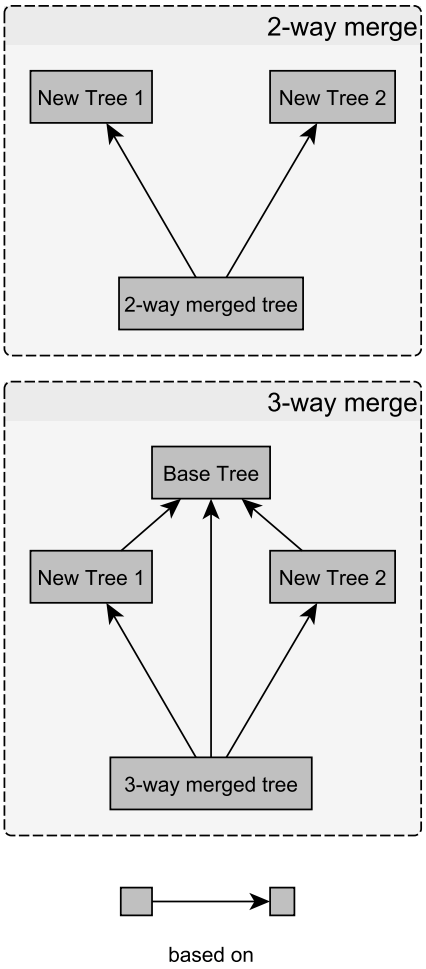

Abbildung 3. 3-way und 2-way Zusammenführungsstrategien

3.6.4 Verteilte VVS

Eine wichtige Entwicklung war es nun, die nur zentralisiert stattfindende Versionsverwaltung zu verteilen. Alte, aber in großen Firmen und Projekten immer noch eingesetzte, VVS, sehen es vor, die Datenhaltung an einer Stelle zu konzentrieren. An sich ist dieses Vorgehen auch gewünscht. Es soll nachvollziehbar sein, welche Änderungen stattgefunden haben. Es soll eben keine unterschiedlichen Versionen geben, die wegen *fehlendem Wissen* über andere Versionen, gleichzeitig und anscheinend gleichberechtigt, existieren und damit eindeutige Versionen und Entwicklungsstände verhindern.

Diesem *Unwissen* über andere existierende Versionen wird mit verteilten VVS entgegengetreten. Hier ist es gewollt, dass es mehrere Stellen gibt an denen die Versionierung stattfindet. Der Unterschied ist aber, dass dies nun koordiniert abläuft und das Wissen auf, an der Versionsverwaltung teilnehmenden Stellen, verteilt wird. So ist es möglich, dass Teilnehmer lokal andere Änderungen haben als andere Teilnehmer. Die Koordination der verteilten Entwicklung ist nun entscheidend. Norma-

lerweise wird eine Stelle zur zentralen, für andere als Ausgangs- und Zielpunkt dienende, Versionsverwaltungsstelle erklärt. Zu dieser werden gemachte Änderungen gegeben und von dort geholt. Eventuelle Fehlschritte die Entwickler in ihren lokalen Repositories gemacht haben, müssen nicht in der zentralen Entwicklungsgeschichte protokolliert und gespeichert werden. Verteilte VVS wurden kommerziell bereits in den Jahren vor 2000 entwickelt. Das erste Projekt für ein freie verteiltes VVS wurde 2001 mit GNU arch [22] ins Leben gerufen.

Im Vergleich zu lokalen und zentralen, nicht verteilten, VVS haben die **Verteilten** folgende **zusätzliche Eigenschaften** die auch als **Vorteile** gesehen werden können:

- **Mehrere Repositories**: Es gibt nicht nur eine zentrale/lokale Stelle an der die Änderungsgeschichte verwaltet wird.
- **Polyhierarchie**: Die Änderungsgeschichte entspricht keiner linearen Kette bzw. einem Baum mehr, sondern einem azyklischen und gerichteten Graph.
- **Offline Versionierung**: Es ist möglich Änderungen lokal zu Versionieren ohne zum zentralen oder anderen Repositories verbunden zu sein.

Diese Eigenschaften der verteilten VVS möchte man sich nun, bei der Verwendung als Back-End, zu Nutze machen.

4 Versionsverwaltungssysteme als Back-End

Die Idee, ein VVS als Back-End zum Verwalten bestimmter Daten zu verwenden, ist nicht neu, sondern wurde beispielsweise von JacobM aufgebracht [23]. Aktueller ist es , dafür ein **verteiltes Versionsverwaltungssystem (VVS)** einzusetzen. Verteilte VVS sind, wie in Unterunterabschnitt 3.6.4 beschrieben, später entwickelt worden und damit noch nicht so lange für alternative Einsatzmöglichkeiten verfügbar.

4.1 Front-End und Back-End

Oft nimmt man eine Unterteilung bei Software in bestimmte Zuständigkeitsbereiche vor [24]. In Client-Server Systemen kommen häufig die Begriffe *Front-End* und *Back-End* zum Einsatz. Ich beziehe den Begriff *Back-End* in dieser Arbeit auf den Datenverarbeitenden sowie Speichernden Teil einer Software, der getrennt von der Daten Ein- und Ausgabe ist.

4.2 Nutzen eines versionierten Back-Ends

Herkömmliche, im Back-End eingesetzte, Arten von Repositories sehen es nicht vor, verschiedene Versionen der abgelegten Daten vorzuhalten bzw. eine Änderungesgeschichte oder ähnliche von VVS gebotene Funktionalität bereitzustellen. Dabei ist es aus meiner Sicht nützlich, auch auf frühere Versionen zurückgreifen zu können oder verschiedene Versionen gleichzeitig der Daten vorhalten zu können.

4.3 Anforderungen an ein Back-End

Die an ein Back-End gestellten Anforderungen können unterschiedlich sein. Deshalb gibt es auch unterschiedliche Herangehensweisen, ein Back-End auf Basis eines VVS zu entwickeln. Grundsätzlich geht es darum, Daten speichern und bearbeiten zu können. Diesem Speichern und Bearbeiten können erweiterte Anforderungen wie gleichzeitiges oder verteiltes Bearbeiten hinzugefügt werden. Zusätzlich dazu müssen die Daten noch in unterschiedlichem Maße verwaltet werden. Im Folgenden gehe ich dazu auf häufige Anforderungen ein:

- **Speichern**: Manchmal ist es bereits ausreichend, ausschließlich Dateien auf der Festplatte schreiben und lesen zu können. Diese Anforderung zu erfüllen reicht aber häufig nicht aus.
- **Gleichzeitiges Bearbeiten**: Oft soll das gleichzeitige Bearbeiten der Dokumente bzw. verwenden eines Dienstes von mehreren Personen oder anderen Rechnern unterstützt werden. Dies ist eine fundamentale Anforderung an ein Back-End. Falls ein Back-End diese erfüllen soll, gilt es unter anderem das **Verlorengegangene Änderungen/lost update**-Problem [25], bei dem eine Änderung verloren geht, weil sie durch eine andere Änderung, welche von anderen Änderungen nichts weiß, überschrieben wird, zu verhindern.
- **Verteiltes Bearbeiten**: Gerade Anwendungen und Dienste die auf der Cloud ausgeführt werden, sollen verteiltes Bearbeiten unterstützen. Verteilt bedeutet hier, dass Änderungen an verschiedenen Eintrittspunkten der Cloud-Anwendung auftreten können, und dann als gespeichert gelten, aber die gemachten Änderungen nicht sofort für andere Knoten, an denen ebenfalls möglicherweise konkurrierende Änderungen gemacht werden, sichtbar sind. Auch dies ist eine Anforderung, die aufwändigere Implementierungen des Back-Ends erfordert.

5 Anforderungen an ein Back-End und Repository für VALESCA

Folgende Anforderungen sollen vom Back-End für VALESCA umgesetzt werden:

1) Benutzer sollen offline arbeiten können
2) Gespeicherte Änderungen sollen rückgängig gemacht werden können
3) Benutzer sollen ein neues Service-Template erstellen und speichern können.
4) Benutzer sollen Änderungen an einem Service-Template speichern können.
5) Benutzer sollen Service-Templates an einer zentralen Stelle speichern können
6) Benutzer sollen unabhängig voneinander parallel an einem Service-Template arbeiten können (Konfliktmanagement)
7) Benutzer sollen so gleichzeitig an einem Service-Template arbeiten können, dass Änderungen einer

bearbeitenden Person nahezu ohne Verzögerung bei allen anderen bearbeitenden Personen im Editor angezeigt wird (Live Editing).

6 KONFLIKTE IN VALESCA

Konflikte treten immer dann auf, wenn mindestens zwei Personen gleichzeitig Änderungen im Repository speichern wollen. Im Folgenden beschreibe ich, wie die daraus resultierende Konflikte in VALESCA aussehen und wie diese gelöst werden können.

6.1 Problem

Für gleichzeitiges Bearbeiten ist es erforderlich, dass sobald mehrere Änderungen in einem Repository eingetragen werden sollen, diese zusammengeführt werden müssen. Hierbei kann es vorkommen, dass die Änderungen sich nicht automatisch vom Repository bzw. im weiteren einem VVS vereinigen lassen und es Konflikte gibt, aber auch, dass es Änderungen gibt, die von ihrer Semantik her in Konflikt stehen und nicht vereint werden dürfen. Der erste Fall kann zum Beispiel auftreten, wenn an einer Datei zwei unterschiedliche Änderungen durchgeführt wurden, die sich auf eine bestimmte Zeile oder ein Element dieser Datei beziehen. Der zweite Fall tritt auf, wenn zum Beispiel in VALESCA zwei Änderungen an Typen durchgeführt wurden, die sich auf Dateibasis zwar vereinen lassen würden, aber von ihrer Bedeutung her widersprüchlich sind. Zur Anschaulichkeit führe ich zunächst ein kleines Beispiel ein.

6.2 Beispielszenarien

Alice und Bob [26] modellieren eine Cloud-Anwendung. Sie haben die Topologie, welche in Abbildung 10 dargestellt ist, ihrer Anwendung bereits gemeinsam erstellt und sind für heute fertig mit ihrer Arbeit. Am nächsten Arbeitstag hat sich Bob dazu entschieden von zu Hause zu arbeiten. Morgens öffnen beide VALESCA und führen, ohne sich auszutauschen, eines der folgenden Szenarien aus:

- Alice fügt einen neuen Knoten für einen Load-Balancer hinzu. Bob fügt einen neuen Knoten für einen seperaten Datenbank-Cache-Server hinzu.
- Alice möchte als Webserver nginx verwenden und Bob lighttpd. Beide Ändern den Knoten entsprechend ab.
- Alice und Bob führen die gleiche Änderung aus. Beide möchten Node.js als Webserver verwenden und ändern den Knoten entsprechend.

Alice speichert ihre Änderungen im Repository als Erste. Als kurz darauf Bob seine Änderungen an der Topologie auch im Repository speichern möchte, kommt es zu einem Fehler, da seine Änderungen die von Alice überschreiben würden.

6.3 Unterschiedliche Behandlung von Konflikten

Die gerade beschriebenen drei Beispielabläufe müssen unterschiedlich behandelt werden. Der erste Ablauf erfordert, dass Bob den neuen Knoten von Alice in seine Topologie übernimmt und darauf aufbauend seinen eigenen neuen Knoten in diese einbaut. Der zweite Ablauf erfordert, dass entschieden werden muss, was für ein Webserver denn jetzt wirklich verwendet werden soll. Der dritte Ablauf ist offensichtlich kein Konflikt, das muss erkannt werden.

6.4 Automatische Konfliktlösung

Um solche Konflikte im Repository möglichst automatisch behandeln zu können, ist es im Allgemeinen hierfür erforderlich alle Informationen über den Inhalt der im Repository gespeicherten Daten diesem zur Verfügung zu stellen und das Repository dazu zu befähigen, diese Informationen verarbeiten und beurteilen zu können. Einem Computer dies beizubringen, ist sehr schwierig. Die Forschungen zur künstlichen Intelligenz, die für eine solche automatische Behandlung und das Treffen von Entscheidungen erforderlich sind, werden meines Wissens noch nicht erforscht.

6.5 Semi-Automatische Konfliktlösung

Es ist auch möglich ohne Wissen über die Semantik, der im Repository verwalteten Dateien, diese mit guten Ergebnissen zusammenführen zu können. Tancred Lindholm [9] hat in seiner Masterarbeit diese Problemstellung sehr genau untersucht. Seine Arbeit behandelt das Vereinen von allgemeinen Baumstrukturen ohne weitere Informationen wie deren Entstehungsgeschichte oder andere Meta-Informationen. Diese Informationen werden automatisch vom Algorithmus aus den Eingabedaten abgeleitet. Als Eingabedaten der Implementierung, genannt 3-way merging, Differencing and Matching (3DM), werden XML-Daten erwartet. Diese werden zunächst zu internen Baumstrukturen umgewandelt (geparst). Für die Java properties- und JSON-Dateien aus VALESCA müsste entweder ein JSON-zu-XML Konverter benutzt werden oder eine neue Parser-Komponente für 3DM hinzugefügt werden. Das den Algorithmus implementierende Programm enthält eine grafische Oberfläche (siehe Abbildung 4) in der das Ergebnis des Zusammenführens als auch nicht lösbare Konflikte betrachtet werden können. Im Folgenden erkläre ich wie der Algorithmus funktioniert.

6.5.1 3DM Algorithmus

Der grobe Ablauf des Algorithmus ist in Abbildung 5 dargestellt. Zunächst werden die XML-Dateien mit einem Parser zu einer internen Baumstruktur verarbeitet. Im nächsten Schritt wird vom `Tree matcher` aufgrund der Anzahl der Eingabedaten entschieden, ob ein *3-way merge* oder eine einfache Berechnung der Unterschiede (*2-way merge*) erfolgen muss. Falls ein *3-way merge* durchgeführt

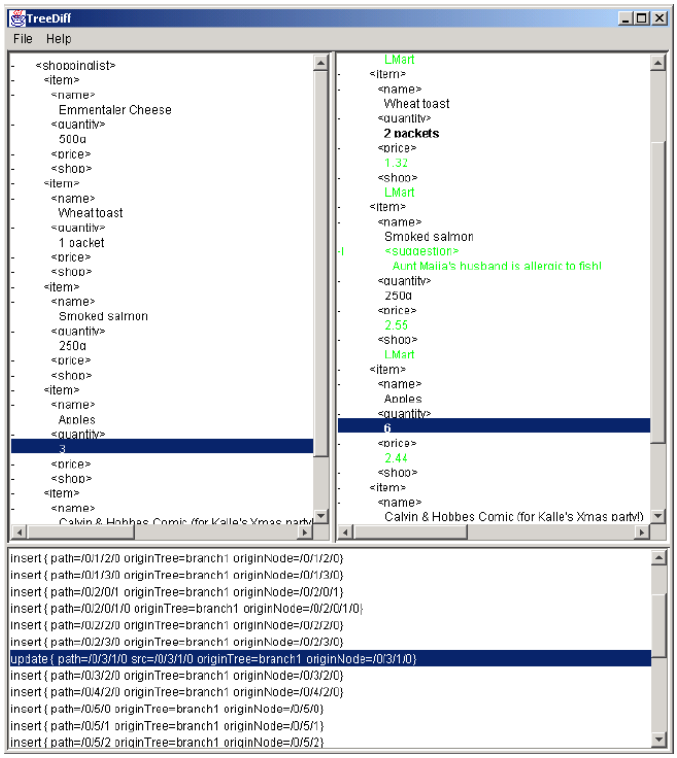

Abbildung 4. Von Tancred Lindholm [9] entwickelter Prototyp **3DM** für 3-way-merge von XML Dateien.

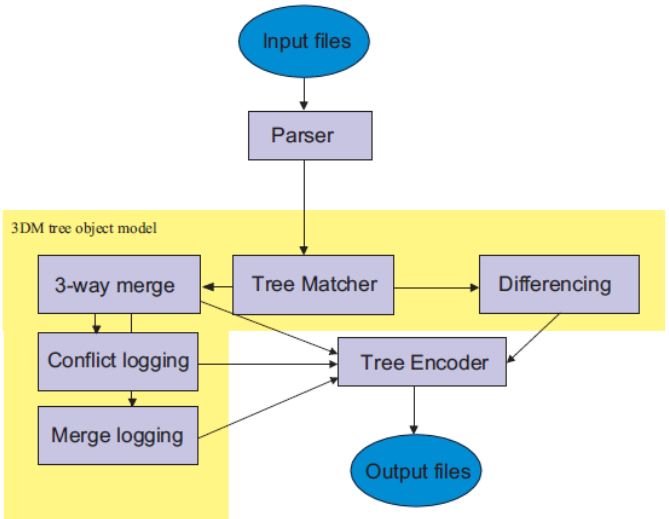

Abbildung 5. Architektur von 3DM aus [9]

wird, merkt sich der Algorithmus Konflikte und die durchgeführten Vereinigungsoperationen. Im abschließenden Schritt werden die Daten vom *Tree encoder* verarbeitet und die Ausgabe-Dateien erzeugt.

6.6 Manuelle Konfliktlösung in der GUI

Da der Umfang dieser Bachelorarbeit es nicht gestattet, entweder eine intelligente vollautomatische oder semi-automatische Konfliktlösung in das Repository zu integrieren, soll das Konfliktmanagement zunächst auf den Benutzer ausgelagert werden. Dies bedeutet, dass der Benutzer auftretende Konflikte manuell lösen muss. Um diese Konflikte trotzdem im gleichen Kontext, in der Weboberfläche, lösen zu können, kann eine Oberfläche wie zum Beispiel PrettyDiff [27] oder mergely [28] verwendet werden. Diese beiden Anwendungen bieten, basierend auf JavaScript, die Möglichkeit im Browser zwei Dokumente zu vergleichen und im Falle von mergely, diese auch direkt zusammenzuführen. Eine weitere Möglichkeit ist, das vorgestellte 3DM-Programm zu verwenden.

7 Evaluation und Auswahl VVS

In diesem Abschnitt untersuche ich verschiedene verfügbare VVS die zur Implementierung eines Back-Ends für VALESCA in Frage kommen. Zu Beginn erläutere ich die verwendete Terminologie. Als nächstes gehe ich noch einmal kurz auf die Anforderungen ein, welche von den VVS erfüllt werden sollten. Danach stelle ich diejenigen VVS, welche den Anforderungen stand halten konnten, vor und überprüfe genauer, warum oder warum sie nicht für eine mögliche Implementierung geeignet sind.

7.1 Terminologie

Um verschiedene Programme vergleichen zu können, wird eine einheitliche Terminologie benötigt. Die zu vergleichenden Programme benutzen für gleichwertige Operationen unterschiedliche Begriffe bzw. Befehle. Aus diesem Grund definiere ich hier, welche Operationen im weiteren wie bezeichnet werden. Bei der Evaluation der Programme werde ich die entsprechenden Befehle den von mir verwendeten Begriffen zuweisen, um Vergleiche der Operationen für den Leser so transparent wie möglich zu machen.

Benötigte Operationen eines verteilten VVS für ein VALESCA-Back-End:

- **Initialisieren** eines Repositories heißt, die erforderlichen Datei- und Ordnerstrukturen des jeweiligen VVS zu erstellen um Eintragungen machen zu können.
- **Hinzufügen** von Dateien die eingetragen werden sollen.
- **Eintragen** einer Änderung bedeutet, gemachte Änderungen wie hinzugefügte, geänderte sowie gelöschte Dateien der Versionsverwaltung bekannt zu machen. Nur durch **Hinzufügen** ausgewählte Änderungen, werden eingetragen. Sobald Änderungen eingetragen wurden, sind diese fest in der Versionsgeschichte vorhanden und ergeben eine neue Version.
- **Rückgängig machen** ermöglicht es, eingetragene Änderungen ungeschehen zu machen. Abhängig vom VVS kann dies einerseits bedeuten, die rückgängig zu machenden Änderungen als neue Eintragung einzutragen oder die Eintragung aus dem Repository zu entfernen.
- **Herausholen** bestimmter Änderungen bezeichnet das Verfügbarmachen eines bestimmten Standes der

versionierten Daten. Zum Beispiel der letzten eingetragenen Änderungen, aus der Versionsgeschichte. Verfügbarmachen kann hier bedeuten, dass noch nicht gemachte Änderungen im aktuellen Verzeichnis mit den Dateien zu einem bestimmten Änderungsstand ersetzt werden, oder dass diese an einer anderen Stelle verfügbar gemacht werden.
- **Versenden** bestimmter Änderungen ist bei verteilten VVS das Aktualisieren eines entfernten Repositories mit bestimmten oder allen Änderungen die im lokalen Repository gespeichert sind.
- **Herunterladen** bestimmter Änderungen bedeutet umgekehrt, Änderungen aus einem entfernten Repository in das lokale Repository zu übertragen.

Für diese Operationen ist am Ende in Tabelle 2 für jedes untersuchte VVS der entsprechende Befehl aufgeführt.

7.2 Anforderungen

Ausgehend von einer Liste verfügbarer VVS bei Wikipedia [29], untersuche ich davon nur diejenigen VVS, welche folgenden grundsätzlichen Anforderungen genügen:
- Aktive Entwicklung
- Verteiltes Repository Modell
- Lockfreie Unterstützung gleichzeitiger Bearbeitung
- Open-Source Lizenz und kostenfreie Nutzung

7.3 Ausgewählte VVS

Folgende Versionsverwaltungssysteme (VVS) erfüllen die gerade gestellten Anforderungen:
- Bazaar [30]
- darcs [31]
- Fossil [32]
- git [3]
- Mercurial [33]
- Monotone [34]
- Veracity [35]

7.4 Gemeinsamkeiten

Alle hier untersuchten VVS sind dafür optimiert, Quellcode-Dateien, d.h. Text-Dateien in ihrem Repository zu verwalten. Es können auch binäre Dateien gespeichert werden, hier ist es schwieriger Unterschiede zwischen zwei Versionen zu erkennen. Jedes der VVS speichert im Grunde Dateien ab und bietet dafür eine Schnittstelle an, die es ermöglicht verschiedene Stände einer Datei wiederherzustellen.

7.4.1 Behandlung großer Binärdateien

Eine schwierige Aufgabe mit dem alle VVS umgehen müssen, ist die Behandlung, im Vergleich zu den Textdateien großen, Binärdateien. Das Problem ist, dass wenn Änderungen an diesen im Repository gespeichert werden müssen, sollten möglichst nicht die komplette neue Version der Datei gespeichert werden, sondern nur das was geändert wurde. Dies kann bei Binärdateien nicht so einfach und effizient wie bei Textdateien gemacht werden.

7.4.2 Befehlsumfang

Die untersuchten VVS haben, für die hier nach der Terminologie relevanten Funktionen, einen gleich großen Befehlsumfang. Die Befehle unterscheiden sich hauptsächlich nur in den übergebenen Parametern und teilweise im Namen. Im Anhang ist in Tabelle 2 ist eine Tabelle aufgeführt in der äquivalente Befehle der VVS aufgelistet sind.

Abgesehen davon gibt es je nach Umsetzung nun kleinere Unterschiede in Bezug auf die genannten Eigenschaften.

7.5 Evaluation

Des Weiteren untersuche ich diese nun auf Erfüllung der in Abschnitt 5 gestellten Anforderungen im Bezug auf VALESCA. Dazu gehe ich in den folgenden Abschnitten genauer auf die ausgewählten VVS ein. Dabei stelle ich teilweise auch die interne Funktionsweise der Programme vor und vergleiche diese miteinander. Insbesondere betrachte ich folgende Eigenschaften der VVS:
- **Datenverwaltung und Vorgehen**: Hier untersuche ich wie das VVS die Dateien intern verwaltet und versioniert.
- **Geschwindigkeit und Skalierbarkeit**: Bei der Geschwindigkeit betrachte ich sowohl das Eintragen im lokalen Repository als auch die Übertragung von und zu einem entfernten Repository. Diese Betrachtung folgt nach der allgemeinen Untersuchung in Form von Vergleichen zwischen den VVS in Unterabschnitt 7.13.
- **Besonderheiten und Einschränkungen**: In diesem Teil hebe ich Stärken eines VVS hervor und zeige Einschränkungen auf, die bei der Verwendung beachtet werden müssen.
- **Java Anbindung**: Da es für VALESCA erforderlich ist, dass das Back-End über Java verfügbar ist, untersuche ich ob und wie diese möglich ist.

7.6 git

Git [3] wurde von Linus Torvalds, dem Schöpfer des Linux Kernels, 2005 entwickelt, nachdem der Entwicklungsgemeinschaft des Linux Kernels die Nutzungsrechte für das dort verwendete proprietäre BitKeeper [36] entzogen wurden. Bei der darauf folgenden Entwicklung eines neuen DRCS hat Linus besonderen Wert auf Geschwindigkeit, Einfachheit und nicht-lineare Zweige gelegt. Dies und mehr sind Stärken von git. Bei den Nachfolgenden Untersuchungen beziehe ich mich auf git in der Version 1.8.0 die am 21.10.2012 veröffentlicht wurde. Die hier komprimiert dargestellten Informationen können in der git Dokumentation [37] nachgelesen werden.

7.6.1 Datenverwaltung

Das Repository in dem git alle zu versionierenden Daten und die Verwaltungsinformationen dazu speichert, ist das .git-Verzeichnis. Parallel dazu existiert normalerweise der

sogenannte *working tree*, in dem sich die Daten befinden die herausgeholt wurden und bearbeitet werden können. Abbildung 6 zeigt die Struktur des .git-Verzeichnisses nach der Initialisierung.

In `objects` legt git alle Datenobjekte ab. Dabei gibt es neben den **Binary Large Objects (BLOBs)** noch 2 weitere Typen. Dazu gehören einmal die `tree`-Objekte. Diese speichern Meta-Informationen, wie Dateiname und Typ zu einem oder mehreren BLOBs. Der Dritte Typ sind *commit*-Objekte. In diesen werden Informationen zu einem `tree`, wie Datum und Autor, gespeichert. Im `refs`-Verzeichnis werden Zeiger auf *commit*-Objekte gespeichert. Zeiger auf `heads` zeigen auf die letzte Eintragung eines Zweigs und Zeiger im `tags`-Verzeichnis zeigen auf bestimmte Eintragungen um diese mit einem Alias bezeichnen zu können. In `index` werden Informationen zur *staging area* gespeichert. `HEAD` zeigt auf den Zweig der aktuell bearbeitet wird. Die restlichen erstellten Einträge im .git-Verzeichnis sind hier nicht weiter relevant.

Beim Speichern von großen Dateien wird im Grunde für jede Änderung die komplette Datei gespeichert. Dies ist erst einmal auch so gewollt um verschiedene Versionen wiederherstellen zu können. Bei einer Dateigröße von n Byte wird das Repository, bei jeder eingetragenen Änderungen an dieser Datei, um $O(n)$ größer. Bei k Eintragungen also um $O(kn)$. Diesem Wachstum wird in git durch die verlustfreie Komprimierung mit Deflate [38] und speichern im gzip-Format [39] entgegnet. Dabei verwendet es noch eine *idx*-Datei um über Offsets auf die in einer komprimierten *pack*-Datei gespeicherten Datenobjekte schneller zugreifen zu können. Genaueres ist unter [37] nachzulesen.

7.6.2 Vorgehen

Beim Hinzufügen und Eintragen geht git wie im Aktivitätendiagramm in Abbildung 7 vor. Teure Operationen sind dabei einmal das Erzeugen eines SHA-1 Hashes der Datei, bei der diese komplett eingelesen werden muss und das Speichern an sich, bei dem sie komplett geschrieben werden muss. Beim Hinzufügen und Eintragen einer Datei der Größe n liegt der Zeitaufwand also in $O(2n) = O(n)$. Somit steigt dieser linear mit der Datengröße. Der Zeitaufwand für das Hashen kann vernachlässigt werden, da dieses um mehrere Größenordnungen schneller geht als das eigentliche Lesen bzw. beim Lesen stattfindet und dieses unwesentlich verlangsamt.

Beim Übertragen von und zu einem entfernten Repository gibt es in git grundsätzlich zwei Möglichkeiten. Die eine verwendet zur Übertragung das Hypertext Transfer Protocol (HTTP). Dafür muss beim entfernten Repository nur ein HTTP-Server laufen. Der große Nachteil dieser Übertragungsart ist, keine Informationen zu git ausgewertet werden oder git direkt benutzt wird. Es werden viele verschiedene kleine Anfragen geschickt und viele Daten übertragen. Die andere Möglichkeit ist die Verwendung der intelligenteren Übertragungsarten. Genutzt werden dafür, neben dem direkten Zugriff auf Dateien über

```
.git
|-- branches
|-- config
|-- description
|-- HEAD
|-- hooks
|   |-- applypatch-msg.sample
|   |-- commit-msg.sample
|   |-- post-update.sample
|   |-- pre-applypatch.sample
|   |-- pre-commit.sample
|   |-- prepare-commit-msg.sample
|   |-- pre-rebase.sample
|   `-- update.sample
|-- info
|   `-- exclude
|-- objects
|   |-- info
|   `-- pack
`-- refs
    |-- heads
    `-- tags
```

Abbildung 6. .git-Verzeichnis Inhalt nach der Initialisierung mit `git init`

`file://`, hauptsächlich git eigenes Protokoll `git://` und dieses Protokoll getunnelt über andere Protokolle wie Secure Shell (SSH). Der Vorteil bei der Benutzung des `git://`-Protokolls ist, dass nur die Informationen übertragen werden müssen die benötigt werden. Dafür muss beim entfernten Repository `receive-pack` und `upload-pack` von git installiert sein. Diese Programme kümmern sich um das Senden und Empfangen. Es ist also geboten, bei Repositories die große Datenobjekte enthalten, nicht das HTTP-Protokoll sondern git eigenes zu benutzen.

7.6.3 Besonderheiten und Einschränkungen

Der `git add`-Befehl markiert, im Gegensatz zu den anderen VVS, nicht nur Dateien dafür, dass sie eingetragen werden sollen. Er erstellt und fügt auch schon die Objektdaten wie BLOB und Hash zum Index hinzu. Der `git commit`-Befehl muss danach nur noch den Baum aktualisieren und Commit-Objekte erstellen und diese verknüpfen.

7.6.4 Java Anbindung

Für Java gibt es verschiedene Bibliotheken die den Zugriff von Java auf git anbieten. Dazu gehören JGit [40] und JavaGit [41]. Dies unterscheiden sich dadurch, dass JGit selbst eine Implementierung von git ist, wohingegen JavaGit ein Wrapper für die originale Implementierung von git ist. Aus Optimierungs- und Geschwindigkeitssicht ist es klar, dass die C-Implementierung von git besser optimiert werden kann und ist, da der C-Code maschinennäher ist [42]. Dies liegt im Weiteren auch

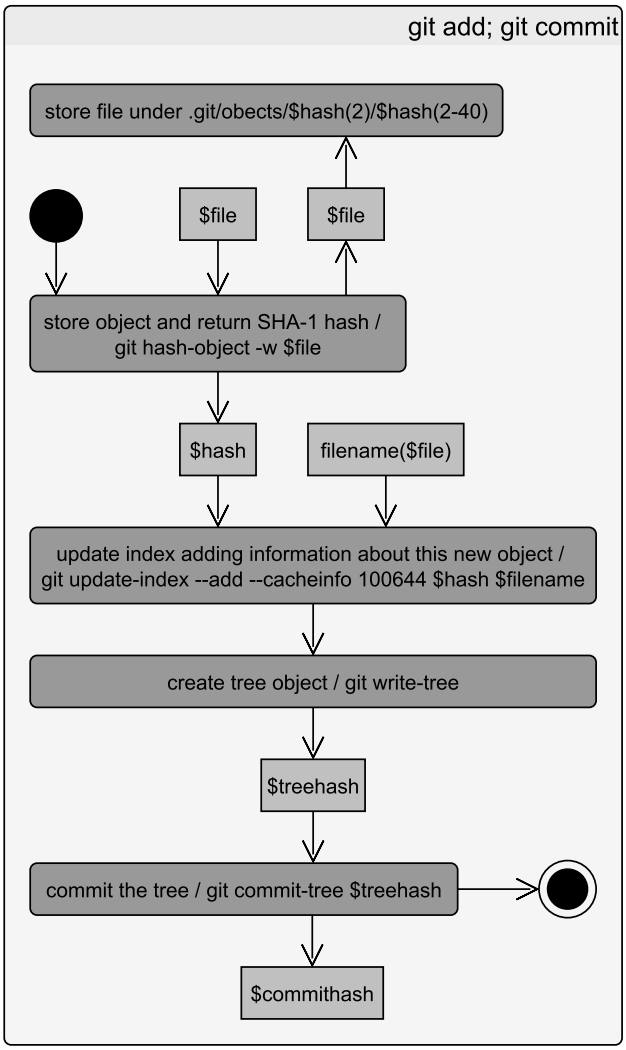

Abbildung 7. Aktivitätendiagramm für `git add; git commit`

daran, dass C eine zu kompilierende Sprache, Java aber eine interpretierte ist. Trotzdem kann es zum Beispiel, wenn die Ausführung nur in Java stattfindet, durch nicht auszuführende Kontextwechsel zur C-Implementierung, passieren, dass Zeit im Vergleich zur Nutzung der C-Implementierung gespart wird. Dies wirkt sich aber nur bei sehr kleinen Eintragungen aus. Zusätzlich ist die reine Java-Implementierung nicht mehr abhängig davon, dass eine git Installation vorhanden ist, sondern ist diese selbst.

7.7 Mercurial

Mercurial [33] funktioniert fast genauso wie git, deshalb verweise ich hier nur auf die Unterschiede, die Dustin Sallings [43] ermittelt hat.

7.8 Bazaar

Auch Bazaar [30] funktioniert ähnlich wie git. Es verwendet zum Beispiel statt SHA einfache Nummern um Versionen zu identifizieren und ist mehr dafür gedacht mit der integrierten GUI benutzt zu werden. Es ist auf eine einfache Benutzbarkeit mittels GUI ausgelegt.

7.9 darcs

Darcs advanced revision control system (Darcs) [31] ist ein von David Roundy entwickeltes verteiltes VVS, geht aber ein wenig anders als die hier vorgestellten VVS vor. Eintragungen sind Patches aber in beliebiger Reihenfolge. Es gibt keine hierarchischen Beziehungen zwischen Vereinigungen zweier Eintragungen. Genau diese fehlenden Beziehungen machen es schwierig konkrete Versionen zu bestimmen. Versionen beziehen sich immer auf den aktuellen Zustand des Repositories.

7.10 Fossil

Fossil [32] verwendet so wie git auch SHA-1 Hashes um Objekte im Repository zu identifizieren. Fossil verwendet SQLite-Datenbanken um die zu verwaltenden Repositories zu speichern. In diesen werden die Daten, auch mit Deflate [44] im zlib-Format [45], komprimiert gespeichert. Eine Besonderheit von Fossil ist, dass mehr als eine herausgeholte Kopie des Repositories unterstützt wird.

7.11 Monotone

Monotone [34] existierte vor git und wurde von Linus Torvalds als „die am meisten brauchbare Alternative“ [46] bezeichnet und auch von ihm ausgiebig untersucht. Deshalb verlasse ich mich auf seine Entscheidung, ein neues VVS zu entwickeln und belasse die Untersuchung mit dem Ergebnis, dass git fortgeschrittener als monotone ist.

7.12 Veracity

Veracity [35] ist ein junges, Ende 2010 veröffentlichtes, VVS. Es bietet einen integrierten Bug-Tracker und im Unterschied zu allen anderen hier vorgestellten VVS ist es möglich verwaltete Dateien sperren zu lassen [47].

7.13 Geschwindigkeits- und Skalierbarkeitsvergleich

In [48] wurden verschiedene Vergleiche zwischen VVS in Bezug auf ihre Geschwindigkeit durchgeführt. Es werden nicht alle hier erwähnten VVS direkt verglichen. Dort wurde angegeben, dass git zwar nicht das schnellste, aber im Bezug auf Speichereffizienz das beste VVS ist.

Die Schwierigkeit diese Aussagen zu verallgemeinern liegt darin, dass es für jedes Projekt sowohl andere Daten gibt die im Repository gespeichert werden, als auch, dass sich die Entwicklungsgeschichte dieser erheblich unterscheiden kann. Somit kann hier keine grundlegende Entscheidung getroffen werden, welches VVS sich am besten eignet.

7.14 Auswahl

Wegen der guten Anbindung durch JGit in Java und der Popularität verwende ich im folgenden git als VVS das für das Back-End verwendet werden soll.

8 Entwurf eines Repositories mit git

Dieser Abschnitt geht auf den Entwurf ein. Es wird zunächst dargelegt, was unter Live Editing zu verstehen ist und wie es umgesetzt werden kann. Darauf folgen andere Design Entscheidungen und ein Konzept wie git in die GUI integriert werden könnte.

8.1 Live Editing

Live Editing, wie in den Anforderungen unter Punkt 7 beschrieben, wird vom verwendeten Editor selbst nicht unterstützt. Es wäre durch folgenden Ablauf möglich, das durch das Repository zu unterstützen:

Sobald eine Änderung im Editor gemacht wird, wird diese Änderung eingetragen und im zentralen Repository bekannt gemacht. Gleichzeitig wird entweder ein aktives Polling durchgeführt um eingetragene Änderungen im zentralen Repository zu registrieren und diese Änderungen dann zu holen, oder dies geschieht durch verschiedene git-hooks welche ausgeführt werden sobald im zentralen Repository eine Änderung eingetragen wird. Active Polling ist grundsätzlich eine schlechte Strategie und erzeugt unnötigen Overhead. Andererseits ist die Variante mit git-hooks auch aufwändig zu implementieren. Ein weiteres Problem ist die Anzahl der Eintragungen. Es würde für jede gemachte Änderung, auch jede noch so kleine Anpassung im Editor, eine Eintragung erzeugt werden. Weiterhin kommt es noch zu Problemen, wenn mehrere Personen gleichzeitig Änderungen vornehmen, die nicht durch automatisches Verschmelzen von git gelöst werden können.

Aufgrund der genannten Probleme sehe ich davon ab diese Funktionalität mit dem Repository zu unterstützen. Es wäre sinnvoller dies im Editor zu implementieren, da hier die Ursache der Probleme liegt und somit die erste Stelle, an der die Probleme behandelt werden sollten. Bei Übereinstimmung der beteiligten Personen abgeschlossene Änderungen können danach gespeichert bzw. im Repository eingetragen werden.

8.2 Speichern entspricht Commit ohne Push

Das Drücken des Speichern-Knopfs im Editor bewirkt eine Eintragung in das lokale Repository. Um die Änderungen für andere im zentralen Repository bereitzustellen, gibt es einen separaten Knopf.

8.3 Architektur von VALESCA mit git

Um offline und gleichzeitiges Arbeiten in VALESCA zu unterstützen, soll dieses auf die in Abbildung 8 dargestellte Architektur portiert werden. Dabei muss zum einen die Anbindung an git entworfen bzw. entschieden werden, wie ein auf git basierendes Back-End

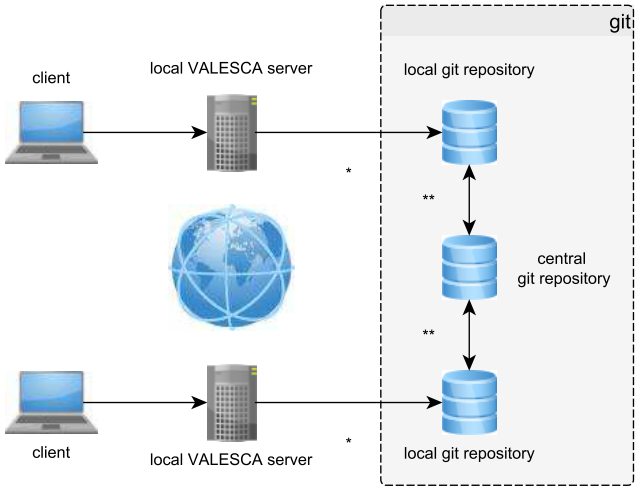

Abbildung 8. VALESCA Architektur mit git Back-End. Bei * sind die Änderungen für ein auf git basierendes Repository nötig, bei ** die Umsetzung der Kommunikation und Verwaltung mit einem zentralen Repository.

in VALESCA integriert werden kann. Im zweiten Schritt muss entschieden werden wie ein zentrales Repository aus Sicht von VALESCA verwaltet wird.

8.4 GUI Konzept für Versionierung

In diesem Abschnitt beschreibe ich, wie ein versioniertes Repository im Front-End von VALESCA umgesetzt werden könnte.

8.4.1 Bisherige Oberfläche

In VALESCA gibt es zwei grundsätzliche Layouts. Die Oberfläche zur Verwaltung von allgemeinen Komponenten (Abbildung 9) für TOSCA und die Ansicht des Editors (Abbildung 10) für die Bearbeitung dieser Komponenten.

8.4.2 Oberfläche mit Integration von Versionskontrolle

Für ein versioniertes Repository sollten in der Oberfläche mindestens folgende Interaktionen und Informationen über das Repository für den Benutzer einfach zugänglich sein:

- Aktuelle Version (Bezeichnung einer Eintragung)
- Aktueller Zweig
- Wie viele Eintragungen noch nicht versendet wurden und wie viele noch nicht heruntergeladen wurden
- Anzeige welche Dateien bei der Eintragung hinzugefügt/geändert/gelöscht werden.
- Eintragen inklusive Autorinformationen und Beschreibungstext
- Versenden
- Herunterladen
- Rückgängig machen
- Versionsgeschichte ansehen

Eine auch für andere Webanwendungen geeignete Implementierung könnte so aussehen: Im oberen Bereich

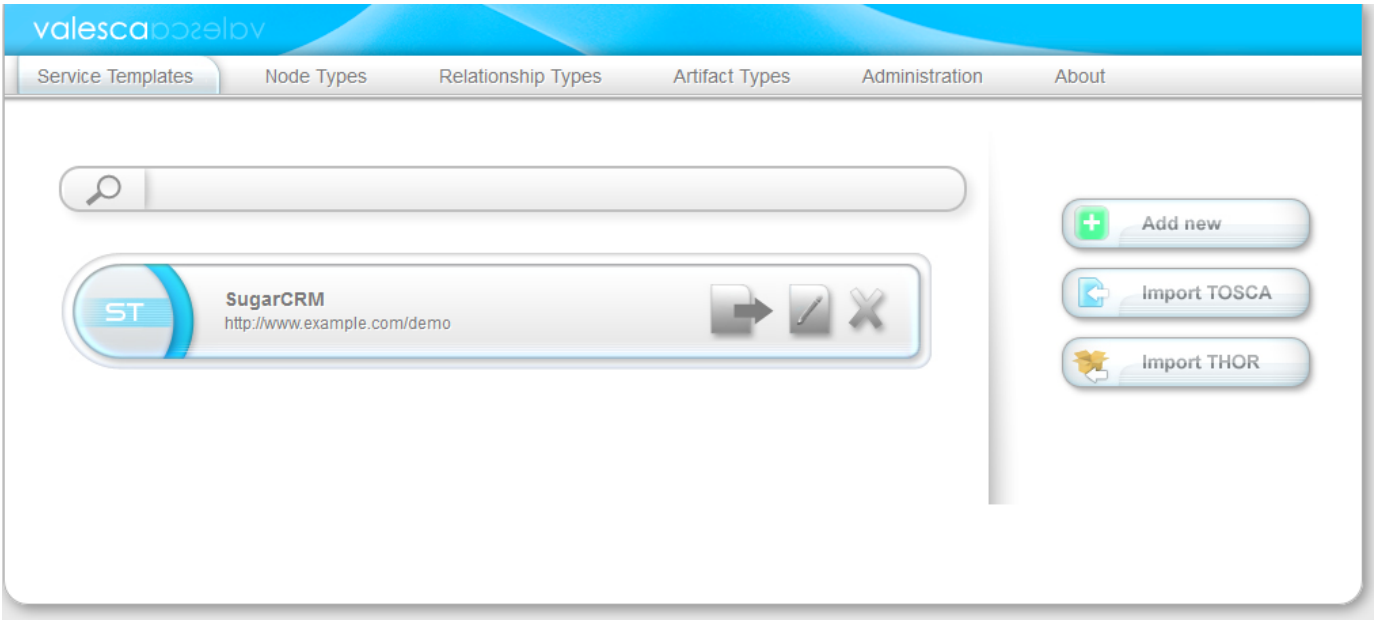

Abbildung 9. VALESCA Weboberfläche

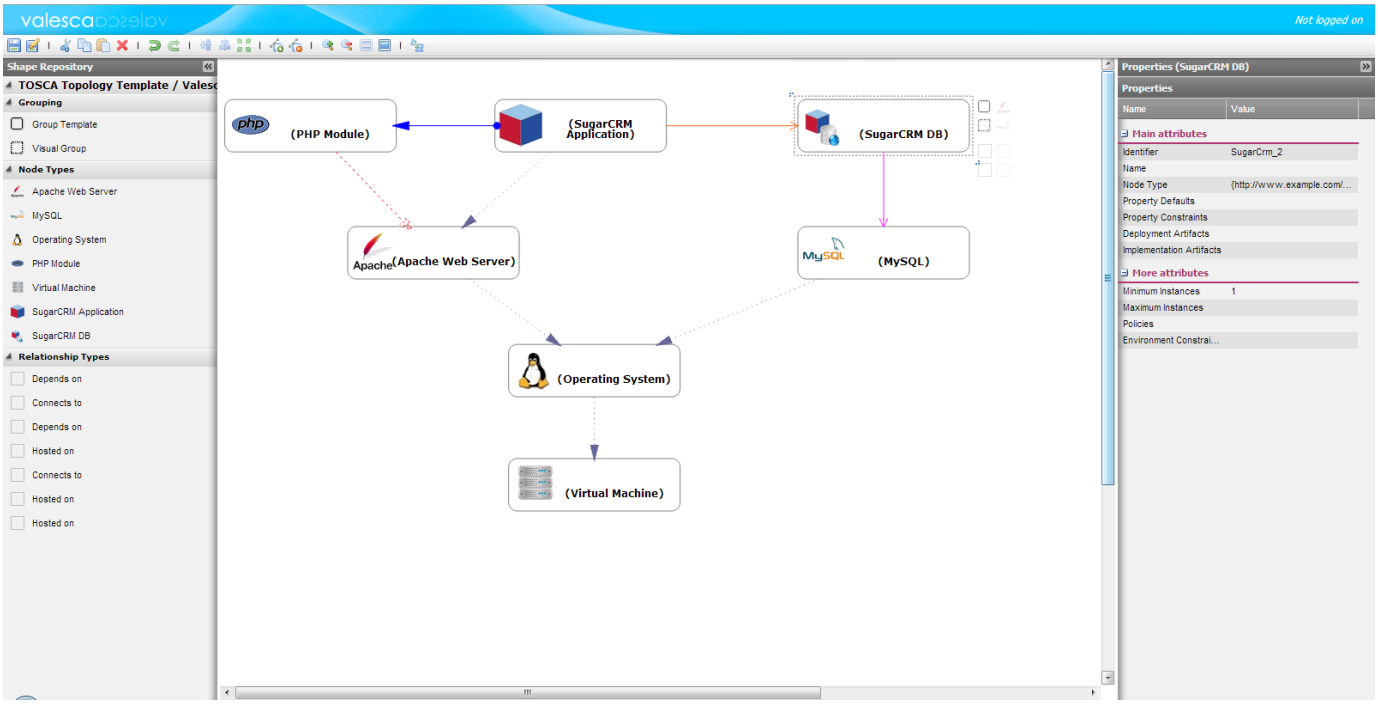

Abbildung 10. VALESCA Editor

der GUI wird eine absolut positionierte Leiste wie in Abbildung 11 eingefügt auf der die genannten Informationen immer sichtbar platziert und die Interaktionsmöglichkeiten durch Knöpfe abrufbar sind. Dazu gehört eine Anzeige wie viele Dateien sich geändert haben, aber noch nicht eingetragen sind und eine Anzeige wie viele Eintragungen noch nicht in ein entferntes Repository übertragen wurden.

Der Quellcode für die hier skizzierte *GitBar* ist in Abbildung 17 dargestellt.

9 Implementierung eines Repositories mit Git

Bei der Implementierung des Repositories setze ich auf einen agilen Ansatz [49]. Ich iteriere durch aufeinander aufbauende Implementierungsphasen/Iterationsschritte. In jeder dieser Phasen werden bestimmte neue Funktionalitäten ergänzt und die in vorhergehenden Phasen implementierten Dinge korrigiert und verbessert. So entsteht nach und nach die komplette Implementierung des Repositories.

Abbildung 11. GitBar mit Knöpfen

```
1  FileRepositoryBuilder builder = new FileRepositoryBuilder();
2  Repository repository = builder.setGitDir(new File("/my/git/directory/.git")).
       readEnvironment().findGitDir().build();
3  Git git = new Git(repository);
4
5  // add changes to staging area
6  AddCommand add = git.add();
7  add.addFilepattern(".").call();
8
9  // commit staged changes
10 CommitCommand commit = git.commit();
11 commit.setMessage("commitmessage").call();
```

Abbildung 12. JGit API Beispiel mit existierendem git-Repository

9.1 JGit

Um vorhandenen Quelltext wiederzuverwenden, entwickle ich keine eigene Anbindung an git, sondern verwende die vorhandene Bibliothek JGit [40], welche eine git-Schnittstelle aus Java-Sicht zur Verfügung stellt. JGit bietet dazu einerseits verschiedene Application Programmable Interface (API)-Methoden zum direkten Aufruf von entsprechenden git-Befehlen, als auch andererseits einige zusammenfassende und abstrahierende Methoden zur effizienten Nutzung von git an.

Um die API benutzen zu können kann man wie in Abbildung 12 vorgehen. Beispielsweise kann man nun mit `RevTree` und `RevWalk` einen Baum von Revisionen objekt-orientiert verarbeiten.

9.2 Iteration 1: Basis git Anbindung

In der ersten Iteration ändere ich jeden vorhandenen Aufruf in VALESCA, bei dem ein Service-Template erstellt oder geändert wird. Ich füge die nötigen git-Befehle hinzu, damit die vorhandene Versionierung der Dateien von VALESCA automatisiert wird.

Da für diesen Iterationsschritt bereits ein git-Repository von Hand erstellt wurde, ist es noch nicht erforderlich, dass dieses erstellt und eingerichtet wurde. Zur Trennung von Verantwortlichkeiten [16] gibt es die Klasse *valesca.filesystem.FileHandling*. In dieser werden Zugriffe auf Dateien gekapselt. Dort erweitere ich die Methode *writeStringToFile* um Aufrufe zum Hinzufügen der Änderungen und zum Eintragen dieser Änderungen.

9.3 Iteration 2: Generalisierung des Dateizugriffs

Um dem Ziel der Speicherung direkt im git-Repository näher zu kommen, ist der nächste Schritt, statt Zugriffe auf das Dateisystem direkt zu verwenden, diese als Zugriffe auf das git-Repository zu kapseln. Diese Kapselung sollte möglichst generell strukturiert sein, um spätere Änderungen am Quelltext so gering wie möglich zu halten. Deshalb erweitere ich VALESCA um die in Abbildung 13 dargestellte Klassenhierarchie.

Ab jetzt wird gefordert, dass die Schnittstelle `IResourceRepository` von allen Klassen, welche die Ressourcen verwalten, implementiert werden muss. Somit ist für den Aufrufenden Code sichergestellt, dass dieser unabhängig von der Implementierung funktioniert. Ressourcen sind in diesem Fall unter anderem die JSON- und SVG-Dateien die von VALESCA erzeugt werden. Dies ist in der bisherigen Implementierung von `FileHandling` anzupassen. Für versionierte Repositories gibt es ab jetzt die Möglichkeit die abstrakte Klasse `AbstractVersionedResourceRepository` zu konkretisieren. Diese implementiert auch `IResourceRepository`. Für Aufrufe wird eine neue Klasse `ResourceRepository` hinzugefügt. Diese neue Klasse weiß, definiert über eine Konfigurationsoption in VALESCA, welche Ausprägung der Schnittstelle `IFileHandling` verwendet werden soll, und führt die Aufrufe auf dieser Instanz aus. Alle Klassen sind nach dem Singleton-Entwurfsmuster aufgebaut. Dieses definiert, dass von einer solchen Klasse nur

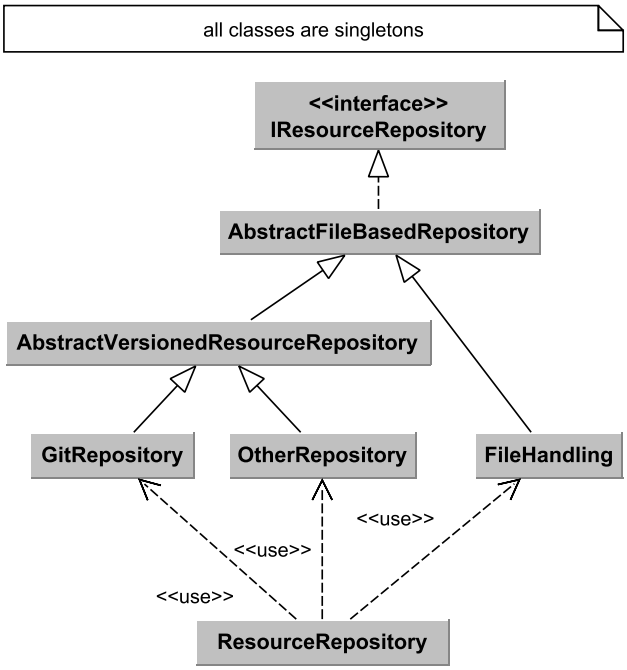

Abbildung 13. Neue Klassenhierarchie für die Dateiverwaltung

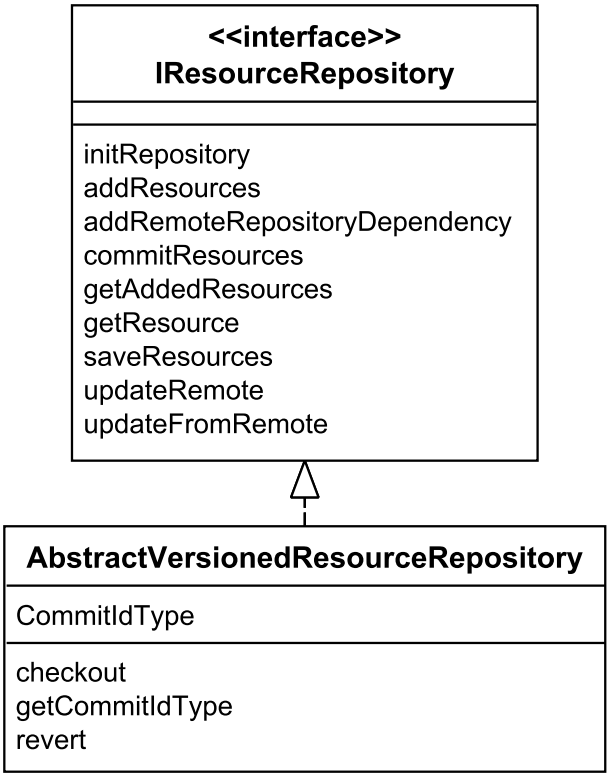

Abbildung 14. Klassendiagramm mit Methoden und Attributen

eine Instanz existieren darf. Da alle hier verwendeten Implementierungen auf Dateien und Ordnern arbeiten, gibt es die Klasse `AbstractFileBasedRepository`.

9.4 Iteration 3: Einbinden der neuen Klassenhierarchie in VALESCA

Einige Zugriffe auf Dateien, welche im Repository abelegt werden sollen, finden nicht über Methoden in `FileHandling` statt, sondern es gibt auch Stellen die direkt Dateizugriffe ausführen oder eine weitere Klasse `filesystem.Utils` verwenden. In dieser Iteration gilt es alle relevanten Zugriffe auf Dateien so umzuleiten, dass die Zugriffe über das Repository laufen, um in späteren Iterationen diese nach und nach verallgemeinern zu können.

9.5 Iteration 4: Repository Schnittstellenmethoden definieren

In dieser Iteration geht es darum die Methoden des Java-Interfaces `IResourceRepository` zu definieren. Auch mit Blick auf die in Abschnitt 5 gestellten Anforderungen, werden nun jeweils entsprechende Methoden, die von jedem in VALESCA verwendeten Repository unterstützt werden müssen, erstellt. In Abbildung 14 sind diese dargestellt.

- `initRepository`: Erstellt die benötigten Dateien und Ordner für die Verwaltung dieses Repositories
- `addRessources`: Markiert Ressourcen zum Eintragen.
- `addRemoteRepositoryDependency`: Fügt eine optionale Abhängigkeit von einem entfernten Repository hinzu.
- `commitResources`: Trägt eingetragene Änderungen im Repository ein
- `getAddedResources`: Listet zur Eintragung hinzugefügte Ressourcen auf
- `getResource`: Gibt eine bestimmte Ressource zurück
- `saveResources`: Kombination von `addResources` und `commitResources`
- `updateRemote`: Lokale Eintragungen an ein entferntes Repository übertragen
- `updateFromRemote`: Eintragungen aus einem entfernten Repository holen.

9.6 Iteration 5: GitFileSystem / WatchService

Um Datei-Änderungen in einem Git-Repository zu speichern gibt es zwei Möglichkeiten. Entweder ich entwickle ein mit git versioniertes Dateisystem (`class GitFileSystemProvider extends FileSystemProvider`) oder ich verwende einen `WatchService` um das Repository-Verzeichnis zu überwachen und bei Änderungen diese in git einzutragen. Es soll folgende Möglichkeiten geben, wann eine Eintragung für geänderte Dateien ausgeführt wird:

- Nicht automatisch, nur manuell eintragen

- Nach dem Ändern einer Datei

9.6.1 GitWorkingTreeWatcher

Bei dieser Implementierung lasse ich von einem `WatchService` jedes Verzeichnis im Repository überwachen. Die zu überwachenden Ereignisse sind dabei die `StandardWatchEventKinds` `ENTRY_CREATE`, `ENTRY_DELETE` und `ENTRY_MODIFY`. In meiner Implementierung füge ich die, bei einem dieser Ereignisse geänderten, Dateien zum Index hinzu (`git add`) und trage die Änderungen, falls so konfiguriert, im Repository ein (`git commit`). Um den GitWorkingTreeWatcher zu verwenden, muss man diesen Instantiieren und ihm dabei den Pfad zu dem Verzeichnis, welches versioniert werden soll, übergeben.

9.6.2 GitFileSystem(-Provider)

In Java 7 gibt es erweiterte Möglichkeiten ein benutzerdefiniertes Dateisystem anzubieten. Hierfür muss man einen `FileSystemProvider` erstellen der ein `FileSystem` anbietet. Auf dieses neue Dateisystem kann aus Java heraus mit der Angabe des für das neue Dateisystem verwendeten Schemas zugegriffen werden. Per Voreinstellung benutzt Java bei Dateizugriffen das der Laufzeitumgebung bereitgestellte Dateisystem des Hosts. Dieses wird über das Schema `file:///` adressiert. Beispielsweise könnte man ein auf git basiertes versioniertes Dateisystem wie folgt erstellen:

Zunächst leitet man `GitFileSystemProivder` von `FileSystemProvier` ab und implementiert die abstrakten Methoden. Dazu `newFileChannel` welches ein `FileChannel`-Objekt zurückgibt über das auf eine Datei zugegriffen werden kann. Abhängig vom Pfad der benutzt wird muss der `FileSystemProvider` entscheiden, welches `FileSystem` benutzt wird. Dies ist zum Beispiel nützlich, weil nicht ein globales versioniertes Dateisystem implementiert werden soll, sondern eines welches für bestimmte Verzeichnisse git-Repositories implementiert. Das von `FileSystem` abgeleitete `GitFileSystem` bietet in jeder Instanz ein neues Dateisystem an. Das Wurzelverzeichnis bezieht sich auf dieses und nicht auf ein eventuelles anderes Dateisystem auf dem es basiert.

9.7 Vergleich der möglichen Ansätze

Die drei von mir beschriebenen Ansätze setzen jeweils auf einer anderen Ebene an. Sehr abstrahiert von der Anwendung selbst, kann durch ableiten von `FileSystemProvider` und `FileSystem` in Java 7 ein neues Dateisystem erstellt werden. Die zweite Möglichkeit ist es, in der Anwendung selbst ein Repository zu erstellen was die anfallenden Zugriffe selbst verwalten und entsprechend versionieren kann. Dies erfordert in der Anwendung eine starke Konzentration der Dateizugriffe um diese koordiniert verarbeiten zu können. Dieser Ansatz ist eine Einzelfalllösung die im Allgemeinen nicht auf andere Anwendungen übertragen werden kann. Somit muss für diesen Ansatz bei jeder Anwendung neu entwickelt werden. Die dritte Möglichkeit ist eine Zwischenschicht. Diese verwendet das vorhandene Dateisystem, das in der Anwendung benutzt wird, und beobachtet Zugriffe. Bei bestimmten Ereignissen werden Methoden aus dieser Zwischenschicht aufgerufen um auf die Ereignisse entsprechend zu reagieren. Dieser Ansatz ermöglicht es, ihn völlig unabhängig von der Anwendung in diese zu integrieren. Allerdings ist dieser Ansatz abhängig von dem in der Anwendung verwendetem Dateisystem und der dafür bereitgestellten Bibliotheken um auf Ereignisse zu reagieren. In Tabelle 1 sind diese Untersuchungen tabellarisch dargestellt. Es wird deutlich, dass die Verwendung des `WatchServices` eine relativ einfache Implementierung ermöglicht, bei der vorhandener Quelltext wenig modifiziert werden muss, da einfach nur ein Verzeichnis überwacht werden muss. Schwieriger wird es, wenn sich die zu versionierenden Daten nicht in einem gemeinsamen Verzeichnis befinden. Dies behandle ich nicht.

10 Zusammenfassung und Ausblick

In dieser Bachelorarbeit habe ich in Abschnitt 9 gezeigt, welche Möglichkeiten es für die Implementierung eines verteilten Back-Ends gibt. Dabei habe ich zunächst verschiedene Anforderungen (Abschnitt 5), welche an ein Back-End gestellt werden sollten, erklärt und bin darauf eingegangen wie diese von verschiedenen verteilten VVS erfüllt werden. Das Ergebnis dieser Arbeit ist eine Zwischenschicht, *GitWorkingTreeWatcher* (Unterunterabschnitt 9.6.1), zwischen dem Dateisystem und einer Anwendung, welche beliebige Daten in einem Ordner ablegt. Diese abgelegten Daten werden von meiner Zwischenschicht automatisch versioniert. Weiterhin habe ich dargestellt, wie es möglich ist, entweder auf Dateisystemebene oder in der Anwendung ein versioniertes Repository umzusetzen. Für die Integration in die GUI von VALESCA habe ich eine Leiste (*GitBar* in Unterunterabschnitt 8.4.2), welche Informationen zum aktuellen Stand des Repositories anzeigt, vorgeschlagen. Diese ist nur ein Konzept und müsste zur Integration mit Funktionalität versorgt werden. Des Weiteren war es in der mir zur Verfügung stehenden Zeit nicht möglich, die vorgeschlagenen Änderungen für eine einheitliche Schnittstelle zum Repository (`IResourceInterface`) in VALESCA zu integrieren. Dies ist mit größeren Aufwand verbunden, wie auch schon in Tabelle 1 beschrieben.

	GitFileSystemProvier	GitWorkingTreeWatcher	GitRepository
Implementierung	extend FileSystemProvider und extend FileSystem	benutzt WatchService	eigene Implementierung
Dateisytemnähe	Sehr nah, eigenes Dateisystem	zwischen Dateisystem und Anwendung.	Nicht nah
Abhängigkeit von Dateisystem	Kann unabhängig sein, da es ein eigenes Dateisystem ist. Ich würde in einer ersten Implementierung zunächst auf das vorhandene Dateisystem `file:///` zurückgreifen.	Abhängig von auf Dateien basiertem Dateisystem. Kann auf Änderungen in diesem bestimte Reaktionen ausführen.	Unabhängig von einem Dateisystem.
Umfang	Sehr umfangreich, da ein komplettes Dateisystem erstellt wird.	Kleiner Umfang	Umfang abhängig von benötigten Operationen im Repository und der Anforderungen der Anwendung.
Integration	Aufwändig, siehe Umfang.	Einfache Integration möglich, da ein Ordner nur überwacht werden muss	Kann aufwändig sein, je nach existierender Konzentration von Zugriffen auf Dateisystem/ anderes Speichersystem

Tabelle 1
Vergleich möglicher Implementierungen in Java für ein versioniertes Repository

LITERATUR

[1] TOSCA TC Mitglieder, *Topology and Orchestration Specification for Cloud Applications (TOSCA)*, English, Working Draft 13, OASIS Technical Commitee, Okt. 2012. Adresse: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca (siehe S. 3, 4).

[2] IAAS, Universität Stuttgart. (Juli 2012). Visual Editor for TOSCA, Adresse: http://www.cloudcycle.org/valesca/ (besucht am 07.07.2012) (siehe S. 3).

[3] Git Mitwirkende. (Juli 2012). Git, Adresse: http://git-scm.com/ (besucht am 07.07.2012) (siehe S. 3, 11).

[4] T. van Lessen, „Konzipierung und Entwicklung eines Repository für Geschäftsprozesse“, Diplomarbeit, Universität Stuttgart, 2006 (siehe S. 3).

[5] C. Roberts, „Integrating version control into ownCloud“, Prifysgol Aberystwyth University, Progressreport, Nov. 2011. Adresse: http://craig0990.files.wordpress.com/2011/11/cgr9_progressreport_v1.pdf (siehe S. 3).

[6] ownCloud Community. (6. Nov. 2012). ownCloud. English, Adresse: http://owncloud.org/ (besucht am 06.11.2012) (siehe S. 3).

[7] R. Grant, „Filesystem Interface for the Git Version Control System Final Report“, PENN ENGINEERING, University of Pennsylvania, Report, Apr. 2009. Adresse: http://www.seas.upenn.edu/~cse400/CSE400_2008_2009/websites/grant/final.pdf (siehe S. 3).

[8] S. Fish. (7. Nov. 2012). Better SCM Initiative : Comparison, Adresse: http://better-scm.shlomifish.org/comparison/comparison.html (besucht am 25.08.2012) (siehe S. 3).

[9] T. Lindholm, „A 3-way Merging Algorithm for Synchronizing Ordered Trees — the 3DM merging and differencing tool for XML“, Master's thesis, Helsinki University of Technology, Dept. of Computer Science, Sep. 2001. Adresse: http://tdm.berlios.de/3dm/doc/thesis.pdf (siehe S. 3, 7, 9, 10).

[10] P. Mell und T. Grance, *The NIST Definition of Cloud Computing*, English, Special Publication, U.S. Department of Commerce, Sep. 2011. Adresse: http://csrc.nist.gov./publications/nistpubs/800-145/SP800-145.pdf (siehe S. 4).

[11] Wikipedia. (6. Nov. 2012). Cloud-Computing — Wikipedia, Die freie Enzyklopädie, Adresse: http://de.wikipedia.org/w/index.php?title=Cloud-Computing&oldid=110166147 (besucht am 06.11.2012) (siehe S. 4).

[12] OASIS TOSCA Committee. (Aug. 2012). Tosca overview, Adresse: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca (besucht am 08.08.2012) (siehe S. 4).

[13] D. Crockford, *The application/json Media Type for JavaScript Object Notation (JSON)*, RFC 4627 (Informational), Internet Engineering Task Force, Juli 2006. Adresse: http://www.ietf.org/rfc/rfc4627.txt (siehe S. 5).

[14] H.-J. Habermann und F. Leymann, *Repository. Eine Einführung*. München: Oldenbourg, 1993, ISBN: 9783486222005 (siehe S. 5).

[15] R. Elmasri und S. Navathe, *Grundlagen von Datenbanksystemen*, Ser. Paerson Studium. Pearson Studium, 2009 (siehe S. 6).

[16] B. Lahres und G. Raỳman, *Praxisbuch Objektorientierung: Von den Grundlagen zur Umsetzung*, Ser. Galileo computing. Galileo Press, 2006 (siehe S. 6, 16).

[17] P. O'Neil und E. O'Neil, *Database–principles, Programming, and Performance*, Ser. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, 2001 (siehe S. 6).

[18] E. A. Brewer, „Towards robust distributed systems (abstract)", in *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, Ser. PODC '00, Portland, Oregon, United States: ACM, 2000, S. 7–, ISBN: 1-58113-183-6. DOI: 10.1145/343477.343502. Adresse: http://doi.acm.org/10.1145/343477.343502 (siehe S. 6).

[19] W. F. Tichy, „Tools for Software Configuration Management", in *SCM*, J. F. Winkler, Hrsg., Ser. Berichte des German Chapter of the ACM, Bd. 30, Teubner, 1988, S. 1–20 (siehe S. 6).

[20] G. Hohpe und B. Woolf, *Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solutions*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003 (siehe S. 6).

[21] M. Fowler, *Patterns of Enterprise Application Architecture*, Ser. The Addison-Wesley Signature Series. Prentice Hall, 2003, ISBN: 9780321127426 (siehe S. 6).

[22] T. Lord. (21. Mai 2008). GNU arch, Adresse: http://www.gnu.org/software/gnu-arch/ (besucht am 28.11.2012) (siehe S. 8).

[23] JacobM. (März 2010). Using a version control system as a data backend, Adresse: http://stackoverflow.com/questions/2519252/using-a-version-control-system-as-a-data-backend (besucht am 03.08.2012) (siehe S. 8).

[24] J. Dunkel, A. Eberhart, S. Fischer, C. Kleiner und A. Koschel, *Systemarchitekturen für Verteilte Anwendungen: Client-Server, Multi-Tier, SOA, Event Driven Architectures, P2P, Grid, Web 2.0*. Hanser Fachbuchverlag, 2008 (siehe S. 8).

[25] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade und V. Watson, „System r: relational approach to database management", *ACM Trans. Database Syst.*, Bd. 1, Nr. 2, S. 97–137, Juni 1976, ISSN: 0362-5915. DOI: 10.1145/320455.320457. Adresse: http://doi.acm.org/10.1145/320455.320457 (siehe S. 8).

[26] R. L. Rivest, A. Shamir und L. Adleman, „A method for obtaining digital signatures and public-key cryptosystems", *Commun. ACM*, Bd. 21, Nr. 2, S. 120–126, Feb. 1978, ISSN: 0001-0782. DOI: 10.1145/359340.359342. Adresse: http://doi.acm.org/10.1145/359340.359342 (siehe S. 9).

[27] A. Cheney. (3. März 2009). Pretty Diff - The difference tool, Adresse: http://prettydiff.com/ (besucht am 08.11.2012) (siehe S. 10).

[28] J. Peabody. (3. Nov. 2012). mergely - Diff and merge offonline, Adresse: http://www.mergely.com/ (besucht am 08.11.2012) (siehe S. 10).

[29] Wikipedia. (Aug. 2012). Comparison of revision control software — Wikipedia(,) The Free Encyclopedia, Adresse: http://en.wikipedia.org/w/index.php?title=Comparison_of_revision_control_software&oldid=505233869 (besucht am 03.08.2012) (siehe S. 11).

[30] Canonical. (Aug. 2012). Bazaar, Adresse: http://bazaar.canonical.com/ (besucht am 03.08.2012) (siehe S. 11, 13).

[31] DarcsTeam. (Aug. 2012). Darcs, Adresse: http://darcs.net (besucht am 03.08.2012) (siehe S. 11, 13).

[32] Fossil. (Aug. 2012). Fossil, Adresse: http://www.fossil-scm.org (besucht am 03.08.2012) (siehe S. 11, 13).

[33] M. Community. (Aug. 2012). Mercurial, Adresse: http://mercurial.selenic.com (besucht am 03.08.2012) (siehe S. 11, 13).

[34] monotone. (Aug. 2012). Monotone, Adresse: http://www.monotone.ca (besucht am 03.08.2012) (siehe S. 11, 13).

[35] SourceGear. (Okt. 2012). Veracity - The Next Step in DVCS, Adresse: http://veracity-scm.com/ (besucht am 23.10.2012) (siehe S. 11, 13).

[36] BitMover, Inc. (26. Okt. 2012). Bitkeeper, Adresse: http://www.bitkeeper.com/ (besucht am 26.10.2012) (siehe S. 11).

[37] S. Chacon. (26. Okt. 2012). git Book, Adresse: http://git-scm.com/book/en (besucht am 26.10.2012) (siehe S. 11, 12).

[38] P. Deutsch, *GZIP file format specification version 4.3*, RFC 1952 (Informational), Internet Engineering Task Force, Mai 1996. Adresse: http://www.ietf.org/rfc/rfc1952.txt (siehe S. 12).

[39] J. loup Gailly und M. Adler. (27. Juli 2003). The gzip home page, Adresse: http://www.gzip.org (besucht am 31.10.2012) (siehe S. 12).

[40] The Eclipse Foundation. (Sep. 2012). Jgit, Adresse: http://www.eclipse.org/jgit/ (besucht am 15.10.2012) (siehe S. 12, 16).

[41] J. Project. (). Javagit. English, Adresse: http://javagit.sourceforge.net/ (besucht am 29.10.2012) (siehe S. 12).

[42] S. O. Pearce. (30. Apr. 2009). Why Git is so fast, Was: re: eric sinks blog - notes on git, Adresse: http://marc.info/?l=git&m=124111702609723&w=2 (besucht am 29.10.2012) (siehe S. 12).

[43] D. Sallings. (6. Apr. 2008). The Differences Between Mercurial and Git, Adresse: http://www.rockstarprogrammer.org/post/2008/apr/06/differences-between-mercurial-and-git/ (besucht am 26.11.2012) (siehe S. 13).

[44] P. Deutsch, *DEFLATE Compressed Data Format Specification version 1.3*, RFC 1951 (Informational), Internet Engineering Task Force, Mai 1996. Adresse: http://www.ietf.org/rfc/rfc1951.txt (siehe S. 13).

[45] P. Deutsch und J.-L. Gailly, *ZLIB Compressed Data Format Specification version 3.3*, RFC 1950 (Informational), Internet Engineering Task Force, Mai 1996. Adresse: http://www.ietf.org/rfc/rfc1950.txt (siehe S. 13).

[46] L. Torvalds. (6. Apr. 2005). Lkml, Adresse: http://lkml.org/lkml/2005/4/6/121 (besucht am 26.11.2012) (siehe S. 13).
[47] SourceGear. (2. Juli 2012). How do file locks work?, Adresse: http://veracity-scm.com/qa/questions/1105/how-do-file-locks-work (besucht am 26.11.2012) (siehe S. 13).
[48] Git Community. (2. Feb. 2010). GitBenchmarks - Git SCM Wiki, Adresse: https://git.wiki.kernel.org/index.php/GitBenchmarks&oldid=8548 (besucht am 29.10.2012) (siehe S. 13).
[49] J. Ludewig und H. Lichter, *Software Engineering - Grundlagen, Menschen, Prozesse, Techniken*. dpunkt.verlag, 2007, S. I–XXI, 1618–, ISBN: 978-3-89864-268-2 (siehe S. 15).
[50] M. Shell. (März 2007). Manuscript Templates for Conference Proceedings, Adresse: http://www.ctan.org/tex-archive/macoros/latex/contrib/IEEEtran/ (besucht am 10.08.2012) (siehe S. 21).
[51] B. van der Zander und F. T. Authors. (22. Nov. 2012). TeXstudio. Version 2.5, Adresse: http://texstudio.sourceforge.net/ (besucht am 27.11.2012) (siehe S. 21).
[52] JabRef Authors. (18. Nov. 2012). JabRef reference manager. Version 2.9 beta 1, Adresse: http://jabref.sourceforge.net (besucht am 27.11.2012) (siehe S. 21).
[53] yWorks. (Okt. 2012). yEd Graph Editor, Adresse: http://www.yworks.com/de/products_yed_about.html (besucht am 24.10.2012) (siehe S. 23).

Acknowledgments

Als Erstes möchte ich meiner Familie für Ihre Unterstützung während der gesamten Arbeitszeit danken. Es ist wichtig in Zeiten großer Erschöpfung von Verwandten sowie auch Freunden motiviert zu werden. Deshalb gebührt hier ein weiterer Dank meinen Freunden und Kommilitonen Daniel Maurer, Marius Kleiner und Florian Straßer, welche mir wichtige Anregungen zur Gestaltung sowie zum Inhalt gaben und auch für die nötige Ablenkung und Abwechslung abseits dieser Arbeit sorgten. Abschließend möchte ich natürlich noch meinem Betreuer, Dipl.-Inf. Oliver Kopp, für seine Geduld und Unterstützung danken.

Anhang A
Abkürzungsverzeichnis

3DM	3-way merging, Differencing and Matching
ACID	Atomicity, Consistency, Isolation and Durability
API	Application Programmable Interface
BASE	Basically Available, Soft State, Eventual consistency
BLOB	Binary Large Object
Darcs	Darcs advanced revision control system
DBS	Datenbank SystemDatenbank Systeme
DBMS	Datenbank Management System
DRCS	Distributed Revision Control System
FUSE	Filesystem in Userspace
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IaaS	Infrastruktur as a Service
IEEE	Institute of Electrical and Electronics Engineers
JSON	JavaScript Object Notation
NIST	National Institute of Standards and Technology
NoSQL	Not only SQL
PaaS	Platform as a Service
RCS	Revision Control System
SCM	Software Configuration Management
SaaS	Software as a Service
SHA	Secure Hash Algorithm
SSH	Secure Shell
SVG	Scalable Vector Graphics
TOSCA	Topology and Orchestration Specification for Cloud Applications
VALESCA	Visual Editor for TOSCA
VVS	Versionsverwaltungssystem
XML	eXtended Markup Language

Anhang B
LaTeX Vorlage

Die für diese Arbeit verwendete Vorlage basiert auf der vom Institute of Electrical and Electronics Engineers (IEEE) veröffentlichten LaTeX-Vorlage *IEEEtran* [50]. Diese habe ich noch um die in Abbildung 15 dargestellten Angaben direkt nach der **documentclass** Definition ergänzt (http://www.michaelshell.org/tex/ieeetran/).

Da die mitgelieferten Literatur-Styles nur für BibTex sind, wurde das Paket `biblatex-ieee` durch die Option `style=ieee` geladen.

Anhang C
Verwendete Software

C.1 Ordner und Dateibaumstrukturen

Die Darstellung der in einen Ordner enthaltenen Dateien und Ordner habe ich unter Linux mit dem Befehl `tree folder -charset=ASCII > asciitreeoffolder.txt` erstellt. Der Parameter `-charset=ASCII` ist notwendig damit die erzeugte Ausgabedatei nur die vom *listings*-Paket verwendbaren Zeichen (ASCII) enthält. Will man die dargestellte Tiefe ändern, verwendet man den Parameter `-T x`, dabei bezeichnet `x` die Tiefe. In LaTeXwerden die erzeugten Textdateien mit dem Befehl `verbatiminput` eingebunden.

C.2 LaTeX

Zur Bearbeitung der LaTeX-Dateien kam TeXstudio [51] und JabRef [52] zum Einsatz.

```
\documentclass[10pt,journal,compsoc,a4paper,twoside]{IEEEtran}
% !!!!!!
% NEEEDED FOR ngerman and babel to work!!!
% !!!!!!
\makeatletter
\def\markboth#1#2{\def\leftmark{\@IEEEcompsoconly{\sffamily}\MakeUppercase{\protect#1}}%
\def\rightmark{\@IEEEcompsoconly{\sffamily}\MakeUppercase{\protect#2}}}
\makeatother
\usepackage[utf8]{inputenc}
\usepackage[T1]{fontenc}
\usepackage[ngerman]{babel}
\usepackage{microtype}
\usepackage{listings}
% for index in pdf (highlights and linkifies links aswell in pdf)
% hidelinks removes coloured boxes around links in pdf, links stay clickable
\usepackage[hidelinks]{hyperref}
\usepackage[]{acronym}
\usepackage[]{csquotes}
\usepackage{verbatim}
\usepackage{tabularx}
\usepackage[style=ieee,backref=true,backend=bibtex8]{biblatex}
\bibliography{literatur}
%colors
\usepackage[usernames,dvipsnames,table]{xcolor}
\definecolor{lightlight-gray}{gray}{0.95}
\definecolor{light-gray}{gray}{0.8}
\usepackage{datetime}
\usepackage[
        title={Verteiltes Modellrepository fuer TOSCA},
        author={Kai Mindermann},
        type=bachelor,
        institute=iaas,
        number=8,
        course=cs,
        examiner={Prof.\ Dr.\ Frank Leymann},
        supervisor={Dipl.-Inf.\ Oliver Kopp},
        startdate={1.~Juni~2012},
        enddate={1.~Dezember~2012},
        crk={
                H.3.2, % H.3.2 Information Storage   File organization
                H.3.4, % H.3.4 Systems and Software Distributed systems  (new)
                H.3.5, % H.3.5 Online Information Services  Web-based services  (new)
                H.5.2 % H.5.2 User Interfaces
        },
        language=german
        ]{uni-stuttgart-cs-cover/uni-stuttgart-cs-cover}
\usepackage[pdftex]{graphicx}
\usepackage{epstopdf}
%define settings for listings
\lstset{
        basicstyle=\ttfamily,
        stepnumber=1,
        xleftmargin=2em,
        numbers=left,
        breaklines=true,                % sets automatic line breaking
        breakatwhitespace=false,
        showstringspaces=false,
}
\pdfinfo{
/CreationDate (D:20121125155000)
/ModDate (D:\pdfdate)
}
\hypersetup{pdfinfo={
Title={Verteiltes Modellrepository fuer TOSCA},
Author={Kai Mindermann}
}}
```

Abbildung 15. Ergänzungungen der LaTeX-Vorlage

C.3 Graphen

Für die Modellierung der Graphen habe ich den yEd [53] von yWorks verwendet.

ANHANG D
GROSSE ABBILDUNGEN

In diesem Abschnitt werden Abbildungen und Tabellen dargestellt.

D.1

Abbildung 16 zeigt ein Beispiel der Datei- und Ordnerstruktur Repräsentation in VALESCA.

D.2

Tabelle 2 mit der Terminologie entsprechenden Befehle der untersuchten VVS.

D.3

Abbildung 17 zeigt den Quelltext der als Konzept vorgestellten GitBar als git-Integration in VALESCA.

Kai Mindermann hat im Jahr 2008 sein Abitur abgeschlossen. Direkt im Anschluss leistete er Wehrdienst bei der Bundeswehr und erhielt dort das Abzeichen für Leistungen im Truppendienst der Stufe 3. Im Oktober 2009 begann er sein Informatikstudium an der Universität Stuttgart. Neben den obligatorischen Grundlagen erlangte er dort unter anderem Kenntnisse im Bereich eingebetteter als auch verteilter Systeme sowie in vertieften Grundlagen der Rechnernetze. Mit Abschluss dieser Arbeit erreicht er mit 23 Jahren den Hochschulabschluss, Bachelor of Science. Gleichzeitig ist er bereits jetzt dabei, sich auf den nächsten Abschluss, Master of Science, vorzubereiten.

```
.
|-- artifacttemplates
|   `-- http%3A%2F%2Fwww.example.com%2Fdemo
|       |-- EC2ControlArtifact
|       |   |-- artifacttemplate.properties
|       |   `-- EC2-VM-Service.war
|       `-- linuxControlArtifact
|           |-- artifacttemplate.properties
|           `-- EC2-Linux-Service.war
|-- artifacttypes
|   `-- http%3A%2F%2Fexample.com%2FToscaTypes
|       `-- WAR
|           `-- artifacttype.properties
|-- imports
|   `-- http%3A%2F%2Fschemas.xmlsoap.org%2Fwsdl%2F
|       |-- http%3A%2F%2Fec2linux.aws.ia.opentosca.org
|       |   `-- EC2LinuxIAService.wsdl
|       `-- http%3A%2F%2Fec2vm.aws.ia.opentosca.org
|           `-- EC2VMIAService.wsdl
|-- namespaces.properties
|-- nodetypes
|   |-- http%3A%2F%2Fexample.com%2FToscaTypes
|   |   |-- ApacheWebServerType
|   |   |   |-- NodeType.properties
|   |   |   `-- scc-data
|   |   |-- MySqlType
|   |   |   |-- NodeType.properties
|   |   |   `-- scc-data
|   |   |-- OperatingSystemType
|   |   |   |-- interfaces
|   |   |   |-- NodeType.properties
|   |   |   `-- scc-data
|   |   |-- PhpModulType
|   |   |   |-- NodeType.properties
|   |   |   `-- scc-data
|   |   `-- VirtualMachineType
|   |       |-- interfaces
|   |       |-- NodeType.properties
|   |       `-- scc-data
|   `-- http%3A%2F%2Fwww.example.com%2FToscaComponents%2FSugarCrmTypes
|       |-- SugarCrmApplicationType
|       |   |-- NodeType.properties
|       |   `-- scc-data
|       `-- SugarCrmDbType
|           |-- NodeType.properties
|           `-- scc-data
|-- relationshiptypes
|   `-- http%3A%2F%2Fexample.com%2FToscaTypes
|       |-- AppDependsOnPhpRuntimeType
|       |   |-- RelationshipType.properties
|       |   `-- scc-data
|       |-- ConnectsToType
|       |   |-- RelationshipType.properties
|       |   `-- scc-data
|       |-- DependsOnType
|       |   |-- RelationshipType.properties
|       |   `-- scc-data
|       |-- HostedOnType
|       |   |-- RelationshipType.properties
|       |   `-- scc-data
|       |-- MySqlDbConnectionType
|       |   |-- RelationshipType.properties
|       |   `-- scc-data
|       |-- MySqlDbHostedOnMySqlType
|       |   |-- RelationshipType.properties
|       |   `-- scc-data
|       `-- PluginHostedOnContainerType
|           |-- RelationshipType.properties
|           `-- scc-data
`-- servicetemplates
    `-- http%3A%2F%2Fwww.example.com%2Fdemo
        `-- sugarCrm
            |-- ServiceTemplate.properties
            `-- topologytemplate
```

Abbildung 16. Beispiel der Datei- und Ordnerstruktur Repräsentation in VALESCA eines kompletten Servicetemplates (*SugarCRM* aufgelistet nur bis Tiefe 4)

	Bazaar	darcs	Fossil	git	Mercurial	Monotone	Veracity
Initialisieren	init	initialize	init	init	init	setup	init
Hinzufügen	add	add	add	add	add	add	add
Eintragen	commit	record	commit	commit	commit	commit	commit
Rückgängig machen	revert	revert	revert	revert	revert	revert	revert
Herausholen	checkout	apply	open	checkout	update	checkout	checkout
Versenden	push	push	push	push	push	sync	push
Herunterladen	pull	pull	pull	pull	incoming	sync	pull

Tabelle 2
Der Terminologie entsprechende Befehle der untersuchten VVS

```
<!DOCTYPE html><html><head>
<link href="bootstrap/css/bootstrap.min.css" rel="stylesheet" media="screen">
<style type="text/css">
.gitbar {padding-bottom:30px; /* need so much space for this bar vertically*/}
.gitbarcontent {
font-family: Verdana,sans-serif;font-size: 0.8em;
background-color:#92D6FF;
position:fixed;
padding:5px 8px 5px 28px;
top:0px;
right:0px;
left:0px;
border-width:1px 0px 1px 0px;
border-style:solid;
border-color:#2F7AA6;}
</style>
</head>
<body>
<div class="gitbar"><div class="gitbarcontent">
<strong>GitBar</strong> -
<button class="btn btn-primary btn-mini disabled" type="button">Commit</button> |
<button class="btn btn-warning btn-mini" type="button">Reset</button> /
<button class="btn btn-danger btn-mini" type="button">Revert</button> /
<button class="btn btn-default btn-mini disabled" type="button">Push</button> /
<button class="btn btn-default btn-mini" type="button">Pull</button> /
View History - <span class="label label-info">master</span>
</div></div>

<div class="gitbar"><div class="gitbarcontent" style="top:50px;">
<strong>GitBar</strong> -
<span class="badge badge-warning">4</span> &rarr; <button class="btn btn-primary btn-mini" type="button"
    >Commit</button> |
<button class="btn btn-warning btn-mini" type="button">Reset</button> /
<button class="btn btn-danger btn-mini" type="button">Revert</button> /
<button class="btn btn-default btn-mini" type="button">Push</button> /
<button class="btn btn-default btn-mini" type="button">Pull</button> /
View History - <span class="label label-info">master</span>
</div></div>

<div class="gitbar"><div class="gitbarcontent" style="top:100px;">
<strong>GitBar</strong> -
<span class="badge badge-warning">1</span> &rarr; <button class="btn btn-primary btn-mini" type="button"
    >Commit</button> |
<button class="btn btn-warning btn-mini" type="button">Reset</button> /
<button class="btn btn-danger btn-mini" type="button">Revert</button> /
<button class="btn btn-default btn-mini" type="button"><span class="badge badge-warning">13</span> Push<
    /button> /
<button class="btn btn-default btn-mini" type="button">Pull</button> /
View History - <span class="label label-info">master</span>
</div></div>
<script src="jquery.min.js"></script><script src="bootstrap/js/bootstrap.min.js"></script>
</body></html>
```

Abbildung 17. Quelltext der als Konzept vorgestellten GitBar als git-Integration in VALESCA

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift