Unterstützung von „continuous integration“ in verteilt entwickelnden Softwareprojekten

Studienarbeit

Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Softwaretechnik

Verfasser:

Fabian Dittberner
Matrikelnr.: 5014104

Betreuer:

Wolf-Gideon Bleek

Hamburg, 2005

1. Einleitung

1.1. Hintergrund der Studienarbeit

Große, verteilt entwickelnde Softwareprojekte stellen besondere Ansprüche an Entwicklerteams in Hinblick auf die gemeinsame Arbeit im Projekt sowie die kontinuierliche Sicherung der Softwarequalität. Zu den Merkmalen dieser verteilten Entwicklung gehört die gemeinsame Arbeit mehrerer Entwickler auf derselben Quellcodebasis. Damit geht die in Teilen gleichzeitige, überschneidende Entwicklung an denselben Systemkomponenten an unterschiedlichen Arbeitsplätzen einher. Schließlich findet noch die regelmäßige Integration der Änderungen in der zentralen Quellcodebasis statt.

Durch die verteilte Entwicklung ergeben sich an den einzelnen Arbeitsplätzen unterschiedliche Stände der Entwicklung, da Änderungen nicht sofort in die zentrale Quellcodebasis übernommen werden, sondern erst durch das Einspielen der Quelldateien ins System gelangen. Vor diesem Hintergrund spielen mehrere qualitätssichernde Maßnahmen eine Rolle, wie zum Beispiel die konsequente Ausführung von Tests (in dieser Arbeit mit Hilfe von *JUnit* [GB98]). Außerdem werden regelmäßig Abzüge von Zwischenständen des Systems erstellt - so genannte *Releases*[1] - die dem Auftraggeber zu Demonstrations- und Testzwecken zur Verfügung stehen.

Daraus resultieren für das Entwicklerteam kurz- sowie langfristig allerdings mehrere Probleme. Es erfordert ein hohes Maß an Disziplin, die Tests im System nach jeder Änderung konsequent auszuführen. Oftmals steht dem eine gegebenenfalls langwierige Ausführungszeit der Tests (vor allem in größeren Systemen) entgegen. Außerdem - gegeben durch die Verteilung des Quellcodes auf die einzelnen Plätze - kann es vorkommen, dass an einem einzelnen Arbeitsplatz die Tests erfolgreich abschließen. Gleichzeitig können diese auf der gesamten Quellcodebasis im Zusammenspiel mit den Änderungen der anderen Entwickler scheitern. Letztendlich ist der erfolgreiche Lauf der Tests eventuell an eine spezielle Umgebung gebunden, die auf einem Arbeitsplatzrechner eingerichtet wurde und so nicht oder nur in veränderter Form auf anderen Rechnern zur Verfügung steht. Arbeiten mehrere Entwickler zur selben Zeit an derselben Komponente im System, können Änderungen die zentrale Codebasis (engl. *repository*) in einen inkonsistenten, d.h. nicht mehr kompilierbaren Zustand versetzen.

Martin Fowler beschreibt in [FF] das Prinzip der *continuous integration*, die er zu den „best practices“ des *eXtreme Programmings* ([Bec05], S.49-50) zählt:

> *“Software development is full of best practices which are often talked about but seem to be rarely done. One of the most basic, and valuable, of these is a fully automated build and test process that allows a team to build and test their*

[1] Als *Release* bezeichnet man einen Abzug des Softwaresystems, der als neue Version dem Auftraggeber zu Testzwecken oder als fertiges Produkt übergeben wird. Welche Teile des gesamten Systems dabei im *Release* enthalten sein müssen, ist Bestandteil der vertraglichen Vereinbarung zwischen Auftraggebern und Entwicklern und orientiert sich an geplanten Teststellungen und den Terminen der Auslieferung von Systemteilen beziehungsweise des Gesamtsystems.

software many times a day. The idea of a daily build has been talked about a lot. McConnnell recommends it as a best practice and it's been long known as a feature of the Microsoft development approach. We agree with the XP community, however, in saying that daily builds are a minimum. A fully automated process that allows you to build several times a day is both achievable and well worth the effort.

We are using the term Continuous Integration, a term used as one of the practices of XP (Extreme Programming).

[…] Most people consider a compile and link to be a build. At the least we think a build should include starting the application and running some simple tests against it (McConnnell used the term "smoke test": switch it on and see if smoke comes out). Running a more exhaustive set of tests greatly improves the value of continuous integration, so we prefer to do that as well." - ([FF])

Dies umfasst das häufige Einspielen kleiner Änderungen in das Gesamtsystem, das regelmäßige, automatische Ausführen der Tests auf der zentralen Codebasis sowie die Erstellung von so genannten *builds* aus dem aktuellen Stand. Wie dabei das Entwicklerteam regelmäßig kleine Änderungen in das System einspielt und mit den damit verbundenen Konflikten in der Quellcodebasis umgeht, wird im Buch von Lippert, Roock und Wolf ([LRW02], S. 98-107) beschrieben. Dies soll zum einen die fachliche Korrektheit des Systems sicherstellen, zum anderen Auskunft darüber geben, ob sich das Gesamtsystem kompilieren und somit ausliefern lässt.

1.2. Inhalt der Studienarbeit

In dieser Arbeit möchte ich eine Lösung vorstellen, mit der sich *continuous integration* in einem verteilt entwickelnden Projekt unterstützen lässt: die *OpenSource*-Software *CruiseControl* (siehe [CC]). Zum einen möchte ich dabei die Mechanismen der *continuous integration* beleuchten, mit denen in einem verteilt entwickelnden Softwareprojekt die Qualität des Projekts fortwährend beobachtet und sichergestellt werden kann. Zum anderen möchte ich dafür konkrete Umsetzungen in *CruiseControl* betrachten. Dabei stehen Art und Umfang der von *CruiseControl* sowie der von eigenen Werkzeugen gelieferten Informationen für die Entwickler und die Projektleiter im Vordergrund.

Der Arbeit liegen Erfahrungen zugrunde, die ich in den letzten Jahren in mehreren, größeren Softwareprojekten gesammelt habe. Die beiden Projekte, mit denen ich den Einsatz von *CruiseControl* beispielhaft beschreiben möchte, sind dabei mit meiner aktuellen beruflichen und studentischen Tätigkeit eng verbunden.

1.3. Aufbau der Studienarbeit

Eingangs möchte ich die verschiedenen Formen der Qualitätssicherung in einem Softwareprojekt vorstellen. Dabei werde ich eine kurze Beschreibung der jeweiligen Mechanismen geben und darstellen, welchen Aspekt des Entwicklungsprozesses der jeweilige Mechanismus abdeckt. Dies wird Thema des zweiten Kapitels sein.

Zudem werde ich ausführen, auf welche Probleme ein Entwicklerteam bei der Anwendung der beschriebenen Mechanismen während des Entwicklungsprozesses stößt beziehungsweise stoßen kann. Zu den einzelnen Punkten werde ich jeweils einige typische Fälle aufzeigen. Diese Probleme haben sich in der oder in einer ähnlichen Form schon in verschiedenen Softwareprojekten ergeben, die ich im Laufe meiner beruflichen Tätigkeit als Softwareentwickler begleitet habe.

Den grundlegenden Aufbau sowie die Einbindung von *CruiseControl* im Entwicklungsprozess im Allgemeinen möchte ich im dritten Kapitel beschreiben. Die Darstellung der notwendigen Infrastruktur und der Komponenten von *CruiseControl* und deren Zusammenspiel sind dort zu finden. Für eine detaillierte Betrachtung der einzelnen Komponenten habe ich mich auf jeweils eine Alternative festgelegt. Diese Alternativen werde ich dann jeweils kurz beschreiben. Dabei handelt es sich um *CVS* [CVS] als Versionsmanagementsystem, *Ant* [ANT] als ausführende Komponente sowie *JUnit* [GB98] für die Umsetzung der Tests.

Beispielhafte Anwendungen von *CruiseControl* zur Unterstützung von *continuous integration* anhand zweier Projekte sind Inhalt des vierten Kapitels. An einem Projekt möchte ich auch die Stellen im Entwicklungsprozess beleuchten, die *CruiseControl* einnimmt, um die Entwicklung der Systemkomponenten im Sinne der Qualitätssicherung zu steuern. Am zweiten Beispielprojekt möchte ich die Einführung von *continuous integration* unter Verwendung von *CruiseControl* in ein bestehendes Projekt beschreiben. Insbesondere spielt dabei die nachträgliche Einbindung in den Entwicklungsprozess eine Rolle.

Abschließend werde ich im fünften Kapitel einen Ausblick darauf geben, was im Zusammenhang mit *CruiseControl* noch erwähnenswert und interessant ist, allerdings in dieser Arbeit nicht abgedeckt wurde. Des Weiteren möchte ich auch herausstellen, wozu *CruiseControl* im Projekt nicht angewendet wird und wo man mit *CruiseControl* an die Grenzen bei der Unterstützung des Entwicklungsprozesses stößt.

2. Qualitätssicherung in Softwareprojekten

Um die Qualität eines Softwareprojekts zu begutachten und somit deren Sicherung voranzutreiben zu können, lassen sich mehrere Merkmale eines Projekts während der Entwicklung betrachten. Die Erstellung und Ausführung von Tests dient der fachlichen und technischen Kontrolle der erstellten Komponenten. Die Betrachtung der Architektur erlaubt Rückschlüsse auf die Güte der Gesamtkonstruktion sowie Aufbau und Struktur des Gesamtsystems und seiner Komponenten.

2.1. Tests und Testabdeckung

Tests gehören zu den grundlegenden Mechanismen, die Qualität und fachliche Korrektheit[1] in einem Softwareprojekt kontinuierlich auf hohem Niveau zu halten. Mit Unit-Tests, Mock-Tests, Integrationstests (vgl. [Link02], S. 5-6) sowie manuellen Drehbuchtests und Akzeptanztests[2] wird das Softwaresystem während der Entwicklung regelmäßig getestet, so dass bei einer hohen Testabdeckung ein dementsprechendes Maß an Zuverlässigkeit, Robustheit sowie ein minimales Auftreten von programmatischen Fehlern garantiert werden kann. Ebenso gestaltet sich der Entwicklungsprozess weit resistenter gegen Fehlerquellen, die während der Neuentwicklung und Änderung am System auftreten. Programmierfehler werden durch konsequentes Testen früh aufgedeckt und können somit zeitnah korrigiert werden.

Johannes Link begründet in seinem Buch über Unit-Tests ([Link02], S. 8-10) den hohen Stellenwert von Tests bei der *continuous integration* vor allem durch das Vertrauen, welches die Entwickler durch die Tests in das von ihnen entwickelte System aufbauen. Durch die Automatisierung und somit regelmäßige Ausführung der Tests können Änderungen am System kontrolliert eingespielt werden und im Falle von Fehlern zeitnah die fehlerhaften Teile korrigiert werden. Somit ist die Testautomatisierung eine der Voraussetzungen für qualitätssichernde Maßnahmen durch die fortlaufende Integration. Des Weiteren ist eine hohe Testabdeckung eine Grundvoraussetzung für die beim *eXtreme Programming* systematisch durchgeführten *Refactorings*[3]. In einem schlecht getesteten System können diese oftmals massiven Änderungen an Systemteilen viele neue Fehler in das Projekt bringen, die durch die fehlenden Tests dann unbemerkt bleiben. Die kontinuierliche Qualitätsverbesserung, die durch die *Refactorings* in das System fließen soll, führte man somit ad absurdum.

Die Testabdeckung sowie der erfolgreiche Lauf der Tests sind zudem Qualitätsmerkmale, die Bestandteil eines Entwicklungsauftrags sein können. Tatsächlich sind eine prozentuale Mindestabdeckung sowie das fehlerfreie

[1] Im Sinne der Übereinstimmung von Spezifikation und Implementation eines Softwaresystems.

[2] Akzeptanztests betrachtet man bei der automatisierten *continuous integration* nicht, da diese Tests speziell durch Anwender an einer ausgelieferten Version (*Release*) im produktiv-ähnlichen Betrieb vorgenommen werden.

[3] *„Refactoring ist die Umstrukturierung eines Softwaresystems unter Beibehaltung seiner Funktionalität. […] Durch Refactoring kann die Struktur eines Softwaresystems flexibel gestaltet und ständig mit wenig Aufwand an die aktuellen Bedürfnisse angepasst werden.“* – ([LRW02], S.76)

Durchlaufen der Tests unter anderem Inhalt des Vertrags eines größeren Softwareprojekts, an dem ich zur Zeit der Erstellung dieser Studienarbeit mitarbeite.

2.2. Architektur

Unter dem Begriff Architektur fasse ich im Kontext dieser Studienarbeit den strukturellen Aufbau eines Softwaresystems zusammen, insbesondere in Hinblick auf die Komponenten des Systems, die Schichten und die unabhängigen Subsysteme zusammen sowie deren klare Trennung voneinander. Folgende Definition kann man dem Informatik-Handbuch entnehmen:

> *„Im engeren Sinne wird unter einer Architektur die Aufteilung eines Softwaresystems in seine Komponenten (meist Module), deren Schnittstellen, die Prozesse und Abhängigkeiten zwischen ihnen, sowie die benötigten Ressourcen verstanden (siehe [McD94]). Allgemeiner umfasst der Architekturbegriff auch die strukturellen, formalen, nicht an anwendungsfachlichen Inhalten orientierten Prinzipien und Organisationsformen von Software."* - ([PR+94], S. 784)

Die Architektur eines Softwareprojekts ist ein weiteres wichtiges Qualitätsmerkmal. Eine saubere Trennung einzelner, fachlich sowie technisch voneinander klar unterscheidbarer Komponenten fördert eine bessere Handhabbarkeit des Softwaresystems, erleichtert die Wartung und die fortlaufende Weiterentwicklung des Systems.

Ein messbarer Aspekt der Architektur sind die Beziehungen von unterschiedlichen Komponenten zueinander, zum Beispiel in Form einer *benutzt*-Beziehung der Teile einer Komponente durch Teile einer anderen Komponente[1]. Dabei lassen sich Anzahl, Richtung und Form dieser Beziehungen betrachten. Eine Kategorisierung in erlaubte und verbotene Beziehungen lässt zudem eine Einordnung in *saubere* oder *unsaubere* Architekturen zu. Die Zahl der unerlaubten Beziehungen zwischen Komponenten kann ein Maß bilden, über das regelmäßig der Zustand der Architektur im Projekt *gemessen* wird. Bei großen Änderungen im Softwaresystem, zum Beispiel, wenn größere Komponenten in kleinere zerteilt werden, kann somit der aufwändige Zerteilungsprozess gesteuert werden. Schritt für Schritt können dann, wenn nötig, unerlaubte Beziehungen zwischen den neu entstandenen Komponenten und dem restlichen System entfernt und die Architektur somit bereinigt werden.

Diesen Aspekt der Architekturbetrachtung möchte ich in den vorgestellten Beispielprojekten noch einmal hervorheben, da *CruiseControl* im Zusammenspiel mit [xRadar] hierfür die nötige Werkzeugunterstützung liefert.

[1] Es werden explizit die Benutzungen der Klassen einer Komponente oder Schicht durch Klassen einer anderen Komponente oder Schicht betrachtet. Diese Beziehungen erlauben eine Aussage darüber, wie gut die Komponenten voneinander getrennt sind. Die Beziehungen von Klassen innerhalb einer Komponente bleiben für diese Betrachtung außer Acht.

2.3. Regelmäßige *builds*

Das regelmäßige Erstellen von Abzügen des Entwicklungsstands, sogenannten *builds*, ermöglicht den Entwicklern, ihren Auftraggebern fortlaufend Testversionen zur Verfügung zu stellen. Beim *eXtreme Programming* nach [Bec05] sind solche regelmäßigen Zwischenreleases sogar fester Bestandteil des Entwicklungsprozesses. Das zeitnahe Feedback eines Kunden steht dabei im Vordergrund, um kurzfristig auf Änderungswünsche oder neue Anforderungen eingehen und reagieren zu können. Dabei entspricht es auch der Philosophie des *eXtreme Programmings*, frühzeitig mit der Entwicklung zu beginnen und frühe Versionen der zu entwickelnden Software für die weitere Planung und Feinabstimmung zu verwenden. Die Voraussetzung dafür sind lauffähige Testversionen; das regelmäßige Erstellen der *builds* bietet somit eine Grundlage, kompilierbare und getestete Softwareversionen bereitstellen zu können.

Schließlich beschreibt Beck den Prozess der verteilten Entwicklung nicht alleine als ein Problem, Aufgaben in einem Softwareprojekt zwischen den Entwicklern im Team aufzuteilen und erledigen zu lassen, sondern vor allem als Herausforderung, die Änderung aller Entwickler fortwährend in das Gesamtsystem einzubringen (vgl. [Bec05], S. 49-50). Je größer dabei die Zeitabstände zweier aufeinanderfolgender Integrationsschritte sind, desto kostenintensiver ist die erfolgreiche Integration von Änderungen ins System. Entwickler haben umso mehr Änderungen aus dem Team bei der Integration zu berücksichtigen, je mehr Zeit seit der letzten Integration der eigenen Änderungen beziehungsweise Erweiterungen vergangen ist. Die regelmäßigen, automatisierten *builds* können dazu dienen, kurze Integrationszyklen zu unterstützen, da die häufigere Integration von Änderungen nicht gleichzeitig mit einer häufigeren Ausführung der Tests seitens der Entwickler einhergehen muss. Die Zyklen werden somit von der Zeit, die für das Testen benötigt wird, freigehalten. Die Tests werden durch die automatisierten *builds* mit ausgeführt. Die Entwickler bekommen so die nötige Rückmeldung über die Qualität des Gesamtsystems, ohne regelmäßig einen hohen Zeitaufwand dafür betreiben zu müssen, vor jeder Integration die zeitaufwändigen Tests selber auszuführen. Zudem müssen die Entwickler nur reagieren, wenn tatsächlich ein Fehler auftritt.

2.4. Schwierigkeiten bei der Qualitätssicherung

Bei der Nutzung der oben genannten Mechanismen zur Beobachtung und Sicherung der Systemqualität ergeben sich kurz- wie langfristig Probleme, die die konsequente Durchführung der Maßnahmen erschweren beziehungsweise sogar gegenteilige Effekte hervorrufen können.

2.4.1. Ausführung der Tests

Die kontinuierliche Ausführung der Tests fordert vom Entwicklerteam ein sehr hohes Maß an Disziplin. Konsistenz und fachliche Korrektheit des Projekts sind wiederum nur durch die Tests messbar. Dem stehen lange Ausführungszeiten, die

aufwändige Bereitstellung einer angemessenen Testumgebung sowie die für jeden Test notwendige Vor- und Nachbereitung entgegen. Die Testausführung kann somit einen wesentlichen Zeitanteil in der Entwicklung einnehmen, der ab einer gewissen Projektgröße der Gesamtgröße des Projekts nicht mehr angemessen ist. Tests können dabei je nach Art auch nach Laufzeit klassifiziert werden. Die Unit-Tests benötigen bei der Ausführung am wenigsten Zeit, da sie auf das Testen von Methoden oder Klassen beschränkt sind. Integrationstests auf der anderen Seite behandeln das korrekte Zusammenspiel mehrerer Systemkomponenten als fachliche und technische Einheit (vgl. [Link02], S. 5). Diese Tests sind meist komplex, was mit langen Laufzeiten einhergeht. Einen Großteil der Ausführungszeit nehmen dabei nicht die eigentlichen Tests ein, sondern überwiegend das Aufsetzen und Starten der benötigten Umgebungssysteme, wie zum Beispiel Datenbanken. Das führt dazu, dass die Tests seltener ausgeführt werden, vor allem nicht mehr vor jedem Einspielen geänderter Systemkomponenten. Um die Zeit weiter zu verkürzen, die für die Tests aufgewendet werden muss, werden gegebenenfalls nicht alle Tests ausgeführt, sondern nur noch der schnell auszuführende Anteil der einfachen Unit-Tests.

Langfristig kann somit eine gleich bleibend hohe Qualität nicht mehr sichergestellt werden, ohne einen wesentlichen (und recht großen) Anteil an der Entwicklungszeit an die Ausführung des kompletten Testspektrums zu verlieren.

Beim Einsatz von Testabdeckungswerkzeugen ist es unerlässlich, sich mit den Mechanismen zu beschäftigen, mit deren Hilfe die Abdeckung in einem Softwareprojekt ermittelt wird. Die Testabdeckung dient zum einen als Qualitätsmerkmal, da sie angibt, für welche Teile des Systems schon Tests vorhanden sind und für welche noch nicht. Auch wird das Verhältnis dieser beiden Kennzahlen zueinander dargestellt. Zum anderen kann die Erhöhung der Abdeckung als Mittel eingesetzt werden, gezielt einen Aspekt der Qualität des Softwaresystems zu verbessern und damit auch die Gesamtqualität der Software zu erhöhen. Trotzdem kann die Verdichtung der Testergebnisse hinsichtlich der Codeabdeckung die Entwickler in eine falsche Sicherheit führen. Die Testabdeckung beinhaltet nämlich nicht nur alleine durch Tests direkt und durch die Entwickler beabsichtigt getesteten Code, sondern auch alle Systemkomponenten, die indirekt durch einen Test aufgerufen werden.

So werden zum Beispiel GUI-Komponenten[1] möglicherweise durch den Aufruf der zugrunde liegenden Funktionseinheiten während eines Tests zwar automatisch mit erzeugt und landen somit in dem Teil des testabgedeckten Codes. Dadurch werden die Komponenten selbst aber nicht im Sinne der Spezifikation getestet. Dem Entwickler steht dann mit der Testabdeckung keine Möglichkeit zur Verfügung zu erkennen, ob die GUI-Komponente speziell getestet wurde oder einfach durch eine getestete Funktionskomponente fälschlicherweise – also ohne selbst getestet zu werden – mit in der Abdeckung auftaucht.

Ein anderes Problem ergibt sich noch beim Testen von Systemteilen, in denen GUI-Elemente bei der Ausführung eine wesentliche Rolle spielen und die durch Fehler oder nicht im Test berücksichtigte Zustände blockieren können. So kann zum Beispiel ein modaler Dialog, der während des automatisierten Testlaufs erscheint, den gesamten Test blockieren. Zwar erlaubt die automatisierte Testausführung die Angabe von Timeouts, bis zu denen ein Test durchgelaufen sein muss oder aber

[1] Klassen beziehungsweise Komponenten, die die grafische Benutzeroberfläche (*Graphical User Interface, GUI*) kapseln.

abgebrochen wird, allerdings kann dies auch die Testergebnisse verfälschen, wenn solche Blockierungen bei Tests frühzeitig auftreten. Somit ist für die automatisierte Testausführung auf solche Fallstricke zu achten, da – je nach ausführendem System – solche Testblocker im System vorhandene Fehler verdecken. Außerdem wird die Testausführung behindert, speziell im Hinblick auf die Laufzeit der gesamten Tests. Trotz der automatisierten Testausführung sollte ein frühes Feedback über den Zustand der Quellcodebasis möglich sein. Eine unverhältnismäßig lange Laufzeit[1] der gesamten Tests limitiert dabei die Anzahl der pro Tag möglichen Integrationsläufe.

```
// zu testende Methode
public void doCloseAction()
{
      try
      {
            DBWorker worker = new DBWorker();

            // Zugriff auf eine Datenbankfunktion
            if(JoptionPane.showOptionDialog(null,"Speichern?",
                  "", JOptionPane.YES_NO_OPTION) == 1)
            {
                  worker.updateEntries();
            }
      }
      catch(SQLException sqlEx)
      {
            sqlEx.printStackTrace();
      }
}
```

Listing 2. a: Beispielcodefragment einer blockierenden Komponente (in Java)

In Listing 2. a ist ein solches kritisches Codefragment beispielhaft dargestellt. Die zu testende Methode (in diesem Beispiel in der Programmiersprache Java) führt einen Aufruf aus, bei dem eine Benutzerabfrage auftreten kann. Wenn der Testfall dieses Verhalten nicht berücksichtigt, dann erscheint hier ein modaler Dialog, der die weitere Ausführung des Tests blockiert.

[1] Am Entwicklerarbeitsplatz werden schon wenige Minuten Test-Laufzeit als unverhältnismäßig hoch empfunden, während bei der automatisierten Testausführung unter *CruiseControl* je nach Projektgröße auch Zeiten im Bereich einer Stunde toleriert werden können.

2.4.2. Einhaltung der Architektur

Die Architektur in einem Projekt dient nicht nur als Merkmal für eine hohe Qualität der Software, sondern ist auch essentiell für die Erweiterbarkeit und Wartbarkeit von Software. Während eine schlechte Architektur eine spätere Erweiterung erschweren, wenn nicht sogar unmöglich machen kann, sorgt eine gute Architektur für ein reibungsfreies Zusammenspiel der Softwarekomponenten, so dass Austausch und Erweiterung während der gesamten Entwicklung bei Einhaltung einer sauberen Architektur in der Regel mit geringem Mehraufwand zu leisten sind.

Abhängigkeiten zwischen Komponenten der Software können bei großen Projekten zu unangenehmen Kettenreaktionen führen, wenn an einer Komponente Änderungen vorgenommen werden, die sich unmittelbar oder indirekt auf viele oder alle anderen Komponenten des Systems auswirken. Bestehen unnötig viele Abhängigkeiten zwischen Systemkomponenten beziehungsweise sind die Komponenten nicht klar genug gegeneinander abgegrenzt, nimmt der Aufwand, der für Änderungen an einer oder mehreren Komponenten anfällt, mit ansteigender Projektgröße immer weiter zu. Des Weiteren verhindern unnötige Abhängigkeiten gegebenenfalls die fachlich lokal begrenzte Änderung einer Komponente. Die Architektur, genaugenommen die Güte der Umsetzung einer gewählten Architektur in einem Projekt kann somit grundlegend sein für eine erfolgreiche Projektarbeit.

2.4.3. Die Kostenfrage

Eine weitere wichtige Frage hinsichtlich der Qualitätssicherung in einem Projekt ist die nach den Kosten, die durch *continuous integration* entstehen oder eingespart werden können. Dabei werden die Kosten für die Einrichtung des automatisierten *build*- und Testprozesses mithilfe eines eigens dafür vorgesehenen Integrationsserver den Ausgaben für Fehleranalyse und Fehlerbeseitigung durch das Projektteam gegenübergestellt. Clark veranschaulicht das Einsparungspotenzial anhand eines einfachen Rechenbeispiels (vgl. [Cla05], S. 45). Demnach entsprechen die Kosten eines kleinen Teams von Entwicklern schon nach wenigen Stunden denen eines Servers, der sehr gut für automatisierte *build*-Prozesse geeignet ist. Wird wegen der Kosten der Serverhardware auf den automatisierten *build*-Prozess verzichtet, dann steigen, gemäß Clarks Beispiel, die Kosten für die Arbeit, die das Team für die Fehlerbeseitigung regelmäßig investieren muss. Dabei sind die Kosten für die Fehlerbeseitigung umso höher, je später die Fehler im System entdeckt werden[1]. Damit rechtfertigt schon das regelmäßige automatisierte Testen sowie die damit verbundene Konsistenzprüfung der Projekt-Quelldateien die Anschaffung eines Integrationsservers. Dadurch können Fehler früh erkannt und behoben werden, was die Kosten für die Fehlerbeseitigung niedrig hält.

[1] Unentdeckte und nicht behobene Fehler können sich während der Entwicklung im System ausbreiten, da neue Komponenten und Erweiterungen sich auch auf die fehlerhaften Teile abstützen. Je später also Fehler im System behoben werden, desto komplexer und zeitaufwändiger können sich die dafür nötigen Änderungen im System gestalten.

2.5. Zusammenfassung

Um die Qualität eines Softwaresystems zu betrachten, kann man unterschiedliche Aspekte des Systems herausgreifen. Tests begleiten den gesamten Entwicklungsprozess, und ihre regelmäßige Ausführung hilft, Fehler bei der Implementierung frühzeitig zu entdecken und somit die Korrektheit des Systems sicherzustellen. Die Ausführungszeit allerdings kann mit zunehmender Projektgröße – und somit zunehmender Anzahl an Tests – hinderlich für eine konsequente Testausführung durch die Entwickler werden.

Die Testabdeckung wiederum gibt an, wie gut und wie weit das Softwaresystem gestestet ist. Auch wenn die Qualität der einzelnen Tests von deren Implementation abhängig ist[1], so stellt die Testabdeckung doch ein Maß für die *Güte* des getesteten Softwaresystems dar. Bei der Betrachtung der Testabdeckung muss auch beachtet werden, dass Teile des Softwaresystems beim Testen durch andere Systemkomponenten gegebenenfalls mitgetestet werden, ohne dass ihre Funktionalität durch den Test abgedeckt wäre. Dies kann zu einer vermeintlich hohen Testabdeckung führen, die fälschlicherweise eine hohe Qualität vorgaukelt.

Des Weiteren lässt sich die Architektur des Softwaresystems betrachten, speziell das Zusammenspiel der einzelnen Komponenten beziehungsweise Schichten des Systems. Die Zahl und Richtung – und eine damit verbundene Wertung - der Beziehungen zwischen einzelnen Komponenten kann dabei als Maß für die Qualität der Architektur herangezogen werden. Anhand dieser Betrachtung kann das Softwaresystem architektonisch *bereinigt* werden, um die Qualität, Wartbarkeit und Änderbarkeit des Systems dauerhaft zu erhöhen. Wird dieser Aspekt des Softwaresystems außer Acht gelassen, können Kosten beziehungsweise Zeitaufwand für spätere Änderungen und Wartung sehr schnell zunehmen.

Regelmäßig erstellte Abzüge des Systems dienen dazu, dem Auftraggeber für ein Softwaresystem frühzeitig lauffähige Testversionen zur Verfügung zu stellen und dadurch den Entwicklern frühzeitiges Feedback zu geben. Vor allem in Hinblick auf die fachliche Qualität sind diese Testversionen wichtig. Mit ihrer Hilfe können Anwender nachprüfen, ob die bisherige Umsetzung der Anforderungen nicht nur technisch, sondern auch inhaltlich in Ordnung ist. Sie sind damit essentiell für ein mit *eXtreme Programming* (vgl. [Bec05]) entwickelndes Softwareprojekt. Weiterhin bedeuten erfolgreiche *builds*, dass sich die Quellcodebasis in einem konsistenten Zustand befindet und sich alle Quellen kompilieren lassen.

Schließlich beschreibt das Beispiel von Clark (in [Cla04], S. 45) anschaulich, welche Kostenersparnis der Einsatz eines Integrationsservers für ein Projekt bedeutet und wie sich dem gegenüber die kurzfristige Einsparung bei der Serverhardware in die steigenden Kosten für die Fehlersuche und –beseitigung niederschlägt. Durch eine kontinuierliche Qualitätssicherung früh entdeckte und behobene Fehler verursachen somit weit weniger Kosten für die Entwicklung.

[1] Man spricht in diesem Zusammenhang von dem den Tests zugrundeliegenden *Testmodell*, anhand dessen die Tests entworfen und implementiert werden und das sich auf die Stellen im Softwaresystem konzentrieren soll, wo Fehler am wahrscheinlichsten sind.

3. CruiseControl im Überblick

Mit Hilfe von *CruiseControl* können einige der Probleme gelöst werden, die durch die hohen Anforderungen an Qualitätssicherung, Zeitplanung und Produktivität gestellt werden. Dies geschieht durch die kontinuierliche, automatisierte Abarbeitung von Aufgaben wie der Ermittlung der Testabdeckung oder der Prüfung der Einhaltung von Architekturregeln. Einen detaillierten Blick auf diese Aufgaben und wie sie in den Entwicklungsprozess eingebunden sind, soll das folgende Kapitel vermitteln.

3.1. Die Anwendungsteile

Die Anwendung *CruiseControl* besteht aus mehreren Komponenten. Dabei gibt es zwei grundlegende Funktionseinheiten: die Ablaufsteuerung und die Webanwendung. Eine Reihe weiterer optionaler *plug-in*s und nützlicher Werkzeuge kann installiert werden. So gibt es unter anderem eine grafische Clientapplikation, die anstelle der Webanwendung zur Ansicht der *build*-Ergebnisse verwendet werden kann. Ebenso können Konfigurationen mit Hilfe einer grafischen *JavaWebStart*-Anwendung[1] angepasst werden. Das direkte Editieren der XML-Dateien wird dadurch unnötig. Des Weiteren gibt es verschiedene Monitoranwendungen, mit denen die Entwickler den aktuellen *build*-Status beobachten können. Es gibt diese sowohl als Einzelanwendung als auch als *plug-in* für Internet-Browser oder IDE[2]. Eine Auflistung der Drittanbieteranwendungen gibt es auf der *CruiseControl*-Internetseite von *ThoughtWorks* (siehe [Thwk]).

Die beiden wesentlichen Komponenten Ablaufsteuerung und Webanwendung übernehmen folgende Aufgaben: die Ablaufsteuerung dient der Automatisierung der *build*- und Testprozesse und sammelt die Ergebnisse dieser Läufe zusammen. Die zweite Komponente, die Webanwendung, ist zwar im Gegensatz dazu nicht unbedingt erforderlich, damit *CruiseControl* läuft, allerdings sind die Ergebnisse, die *CruiseControl* mit der *continuous integration* erzeugt, nur mit zusätzlichem Aufwand auszuwerten. Durch die Webanwendung werden Testergebnisse, Ausführungsprotokoll sowie Ausführungshistorie für die Entwickler übersichtlich aufbereitet.

3.2. Die Infrastruktur im Projekt

Die Ablaufsteuerung ist als Komponente alleine lauffähig, da es sich um eine ausführbare Java-Anwendung handelt. Auf einem System, auf dem *CruiseControl* laufen soll, wird nur eine *JavaRuntime*-Umgebung benötigt. Um die

[1] JavaWebStart-Anwendungen nutzen webbasierte Verteilungsmechanismen zur Installation und Ausführung (siehe [JWS]).
[2] IDE – *Integrated Development Environment*, Entwicklungsumgebung.

projektspezifischen Aufgaben im *build*-Prozess abzuarbeiten, stützt *CruiseControl* sich auf die Skriptsprache *ANT* (oder eine der unterstützten Alternativen). Außerdem wird der Client eines Versioning-Werkzeugs wie zum Beispiel *CVS* benötigt, um vor einem *build*-Prozess einen Abgleich mit dem aktuellen Projektstand machen zu können. Die Webanwendung wird als installierbares *WAR* mitgeliefert, das zur Ausführung auf einen Webserver aufgespielt werden muss. Die Entwickler müssen zudem natürlich von ihrem Arbeitsplatz aus Zugriff auf den Webserver haben, damit sie die Ergebnisse von *CruiseControl* über die Webanwendung abrufen können.

Zu den Voraussetzungen für den vollständigen Betrieb gehören also:

- die *CruiseControl*-Anwendung
- eine Java-Laufzeitumgebung (JRE oder JDK, Version 1.2 oder höher)
- eine Ausführungskomponente, zum Beispiel *ANT*
- eine Versioningkomponente, zum Beispiel *CVS*

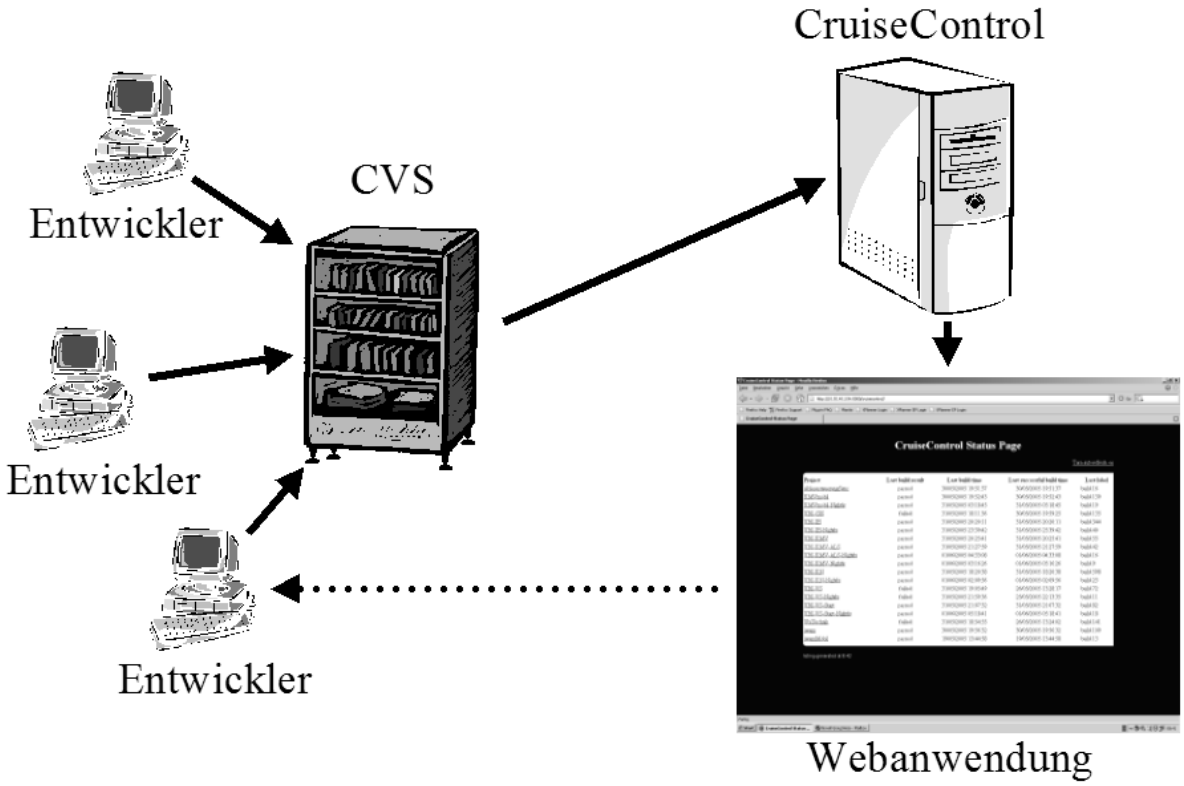

Abb. 3. a: Infrastruktur von CruiseControl im Projekt

Nach der Installation befindet sich auf dem System, auf dem *CruiseControl* ausgeführt wird, das Arbeitsverzeichnis von *CruiseControl*. In diesem Arbeitsverzeichnis ist für jedes in *CruiseControl* konfigurierte Projekt ein Unterverzeichnis eingerichtet. Dieses Unterverzeichnis wiederum enthält alle für einen vollständigen *build* des jeweiligen Projekts benötigten Komponenten. Der Inhalt des Verzeichnisses wird mit dem Stand in der zentralen Quellcodebasis, dem so genannten *Repository*, abgeglichen. Sind mehrere Teilprojekte vorhanden, deren

Komponenten in einem Projekt benötigt werden, so gibt es dementsprechende Verzeichnisse für jedes dazugehörige Projekt im *Repository*. Die einzelnen Komponenten werden dann ebenso mit dem aktuellen Stand im Versionierungssystem abgeglichen.

3.2.1. Die Einbindung von CruiseControl

Für die Verwendung von *CruiseControl* benötigt man in einem Projekt mindestens ein Rechnersystem, auf dem *CruiseControl*, ein Webserver sowie ein *CVS*-Server (oder ein anderes von *CruiseControl* unterstütztes Versioning-System) laufen können. Sind die einzelnen Anwendungen auf unterschiedliche Rechner verteilt, die miteinander vernetzt sind, so müssen einige grundlegende Verbindungen bestehen; *CruiseControl* benötigt Zugriff auf den *CVS*-Server, um die Projektdateien auf dem neuesten Stand halten zu können. Weiterhin muss die Webanwendung die durch *CruiseControl* erzeugten Ergebnisse lesen können, um diese für die Entwickler aufbereiten zu können. Daher muss ein Zugriff vom Web-Server aus auf den *CruiseControl*-Rechner erlaubt sein. Schließlich muss von den Entwicklerarbeitsplätzen aus eine Verbindung zum Web-Server ermöglicht werden, damit über einen Web-Browser die Resultate des *build*-Prozesses angezeigt werden können.

Im Entwicklungsprozess gelangen Änderungen von den Entwicklerplätzen in die Quellcodebasis im *CVS*-System. *CruiseControl* bezieht die für den *build* nötigen Quellen, Bibliotheken und Dateien vom *CVS*-Server. Die Ergebnisse des *builds* wie Tests und Dokumentation können dann über die Webanwendung am Entwicklerplatz zugänglich gemacht. Die Konfiguration von *CruiseControl* erfolgt entweder direkt auf dem *CruiseControl*-Server oder ferngesteuert vom Entwicklerplatz aus. Der oben beschriebene Konfigurationsclient kann dazu benutzt werden. Alternativ können Konfigurationsdateien über eine Terminalanwendung bearbeitet und somit *CruiseControl* entfernt gesteuert werden.

3.2.2. Die Ausführungskomponente – ANT

Für die Ausführung automatischer, geskripteter Arbeitsabläufe im Projekt nutzt *CruiseControl* das Werkzeug *ANT* [ANT], ein java-basiertes Skriptwerkzeug. Damit können viele Standardaufgaben wie Kompilieren, Kopieren, Verpacken in JARs, Ausführen von Unittests und das Erstellen von Enterprise-Anwendungen[1] ausgeführt werden. Für CruiseControl dient *ANT* daher als Skriptsprache, mit der der komplette automatisierte *build*-Lauf beschrieben werden kann. Die Erweiterbarkeit von *ANT* durch an besondere Aufgabenstellungen und Anforderungen angepasste Werkzeugkomponenten ermöglicht damit die Automatisierung kompletter Test- und Releaseprozesse. Die standardmäßig für *continuous integration* benötigte

[1] Als Enterprise-Anwendung werden hier Anwendungen bezeichnet, die einer mehrschichtigen Architektur (*multitier enterprise architecture*) folgend aufgebaut sind (vgl. [J2EE]).

Testunterstützung und Archivierung in Form von `war`- und `ear`-Dateien[1] sind in *ANT* enthalten.

Die beiden von *CruiseControl* unterstützten Alternativen *Maven* und *NAnt* werden in Kapitel 3.2.5. genauer betrachtet.

Im *Elvis*-Projekt[2] (siehe Kapitel 4.1) sollte die Unterteilung in Teilprojekte hinsichtlich des *build*-Prozesses dadurch vereinfacht werden, dass eine Hierarchie in der Ausführung der einzelnen *build*-Skripte geschaffen wurde. Dafür sind die *build*-Skripte abhängiger Projektteile mit denen verknüpft worden, die im *build*-Prozess vor diesen Projektteilen gebaut werden müssen. Damit sollte sichergestellt werden, dass beim *build* eines Projektteils immer auf den neuesten Versionen der darunter liegenden Teile gearbeitet wird. Im *Elvis*-Projekt führte diese Gliederung dann allerdings zu einer sehr rasch zunehmenden Komplexität der *build*-Skripte selbst und somit wiederum des gesamten build-Prozesses.

Die daraus resultierenden Probleme bei der Pflege des automatisierten *build*-Prozesses ergeben sich unter anderem aus der Feingranularität, mit der in *ANT*-Skripten die zu bewältigenden Aufgaben beschrieben werden. Die Entwickler haben detaillierte Kenntnisse des gesamten *build*-Prozesses, müssen deshalb auch die Komplexität bewältigen, die in einem großen Projekt und vielen Projektteilen sehr hoch werden kann. Der deskriptive Ansatz von *Maven* über das Projektmodell könnte hierbei eine besser handhabbare Alternative darstellen, da viele der Standardaufgaben eines *builds* vor den Entwicklern verborgen bleiben. Dementsprechend wird seitens der Entwickler meist auch keine detaillierte Kenntnis des build-Prozesses benötigt, um eine automatisierte *build*-Prozedur aufzusetzen. *Maven* abstrahiert von den Standardaufgaben und hilft somit, die Komplexität von *build*-Prozessen abzubauen.

3.2.3. Die Versionierungskomponente – CVS

In verteilt entwickelnden Softwareprojekten werden so genannte Versionierungssysteme eingesetzt, um die konkurrierende Entwicklung auf den Quelldateien des Softwaresystems zu überwachen und zu steuern. Werden Teile des Systems gleichzeitig an unterschiedlichen Entwicklerarbeitsplätzen verändert, können diese Teile mit Hilfe der Versionierungssoftware zusammengeführt werden.

Die Versionierungskomponente archiviert die Historie der Entwicklungsstände der Quelldateien des Systems. Werden durch fehlerhafte oder konfligierende Änderungen an den Quelldateien Teile des Softwaresystems in einen inkonsistenten Zustand versetzt, so kann ein vorhergehender und als fehlerfrei bekannter Stand aus dem Versionierungsarchiv wiederhergestellt werden. Wird auf einem Entwicklerrechner das Projekt erstmalig für die Entwicklung eingerichtet, so lässt sich aus dem System der aktuelle Entwicklungsstand beziehen.

[1] In vor allem auf der Basis von *J2EE* entwickelten Softwareprojekten werden Testversionen und fertige *Releases* häufig als Webarchive (Dateiendung `war`) und Applikationsserver-Anwendungen (Dateiendung `ear`) zur Verfügung gestellt.

[2] Der Name des Projekts ist anonymisiert und entspricht nicht dem wirklichen Projektnamen.

CruiseControl bezieht die für die *builds* benötigten Quelldateien und Bibliotheken von der Versionierungskomponente. Damit steht für die automatisierten Tests und *builds* immer der aktuelle Projektstand zur Verfügung.

Die OpenSource-Versionierungssoftware *CVS* („*Concurrent Versions System*“, [CVS]) ist eine weit verbreitete Versionierungssoftware und wird standardmäßig von *CruiseControl* für die Versionierung verwendet. In den Beispiel-Projekten in Kapitel 4 wird daher *CVS* als Versionierungskomponente eingesetzt. Die Konfigurationsbeispiele und Ablaufbeschreibungen werden sich daher auf die *CVS*-spezifische Art beschränken. Alternativen zu *CVS* werden später im Kapitel noch einmal kurz erläutert.

3.2.4. Die Webanwendung

Die Ausführung der Tests auf dem aktuellen Projektstand erzeugt über *JUnit* [GB98] ein Protokoll der Testergebnisse und –laufzeiten. Des Weiteren werden Ausgaben während der Ausführung des *build*-Skripts protokolliert. Über die Einbindung weiterer Werkzeuge können während des *build*-Laufes noch zusätzliche Informationen gesammelt werden, wie zum Beispiel Informationen über die Testabdeckung des Codes oder über Regelprüfungen bezüglich der Architektur des Projekts.

CruiseControl bereitet die gesammelten Informationen anhand einer mitgelieferten Webanwendung auf. Diese erzeugt anhand der während des *builds* erstellten Protokolldateien eine Informationsseite, auf der alle Daten in übersichtlicher Form den Entwicklern dargeboten werden. Die Webanwendung arbeitet dabei unabhängig von *CruiseControl*, so dass *CruiseControl* für das Funktionieren der Webanwendung nicht in Betrieb sein muss. Die Webanwendung wird beim Start mit dem Arbeitsverzeichnis von *CruiseControl* konfiguriert, in das *CruiseControl* die Protokolldateien für ein Projekt ablegt. Wird die Webanwendung aufgerufen, so werden die dort liegenden XML-Protokolle ausgewertet und in übersichtlicher Form dargestellt.

Die Webanwendung bietet zudem eine Übersicht über alle in *CruiseControl* konfigurierten Projekte und den dazugehörigen automatisierten *build*-Prozess. Für die einzelnen Projekte sind die Informationen des aktuellen *builds* aufrufbar ebenso wie die Daten vorheriger *builds*. Außerdem sind erfolgreiche *builds* besonders vermerkt. Während des *builds* erstellte Artefakte werden in ein Verzeichnis abgelegt, aus dem sie herunter geladen werden können.

3.2.5. Die Alternativen für CruiseControl-Komponenten

CruiseControl unterstützt zusätzlich zu den in dieser Arbeit genannten Komponenten *CVS* und *ANT* eine Reihe weiterer Alternativen. Die folgenden Anwendungen können unter *CruiseControl* als alternative Versioning-Komponenten verwendet werden:

- *Subversion*: ein *OpenSource-VCS*[1] der *Tigris.org-OpenSource Community* ([SVCS]), das grundlegend auf *CVS* aufbaut, aber einige wesentliche Unterschiede und Verbesserungen zu diesem aufweist. So werden unter Subversion zum Beispiel auch Verzeichnisse, Umbenennungen und Kopien versioniert.

- *AlienBrain*: ein kommerzielles Versioning-System ([AB]) der Firma *Avid Technologies, Inc*. Das System bietet besondere Unterstützung für grafische Quellen und wird somit laut Entwickler überwiegend in der Spieleentwicklung eingesetzt.

- *ClearCase*: eine kommerzielle Quelldatei- und Ressourcenverwaltung von *IBM* ([ClCa]), enthält unter anderem eine Versioning-Komponente, die unter *CruiseControl* genutzt werden kann.

- *MKS*: ein kommerzielles Softwarepaket *MKS Integrity Suite* der Firma *MKS* ([MKS]) mit verschiedenen Werkzeugen zur Unterstützung von Softwareentwicklung, u.a. eine Versioning-Komponente.

- *Perforce*: ein kommerzielles *Software Configuration Management System*[2] (SCM) der Firma *Perforce Software Inc.* ([PFC]), stellt eine ganze Reihe von Clientanwendungen für den Zugriff auf eine zentrale Quelldatenverwaltung zur Verfügung.

- *PVCS*: ein kommerzielles Versioning-System (*Poly Version Control System*) der Firma *Intersolve*.

- *SnapshotCM*: ein kommerzielles *Software Configuration Management System* (SCM) der *Firma True Blue Software* ([SnCM]), das als Besonderheit die verschiedenen Projekt-*Releases* und deren Beziehungen zueinander grafisch aufbereitet.

- *Star Team*: ein kommerzielles *Software Configuration Management System* (SCM) der Firma *Borland* ([ST]).

- *Visual SourceSafe*: die Versioning-Komponente von *Microsofts VisualStudio Developer Suite* ([VSS]).

Wie für die Versioning-Komponente lässt *CruiseControl* auch Alternativen für die Skriptsprache der *build*-Skripte zu. Zu dem hier hauptsächlich verwendeten *ANT* unterstützt *CruiseControl* noch folgende Alternativen:

- *Maven*: eine *OpenSource*-Software der *Apache Software Foundation* ([MV]) zur zentralen Verwaltung von Projekt-*builds*, Reporting und Dokumentation. Im Unterschied zu *ANT*, welches ein sehr detailliertes Wissen über den gesamten *build*-Prozess voraussetzt, wird *Maven* nur eine abstrakte Projektbeschreibung übergeben. In dieser

[1] *VCS – Version Control System.*
[2] Im Gegensatz zu einem *Version Control System* wie *CVS* überprüft ein *Source Configuration Management System* die Korrektheit der eingecheckten Quellen. Änderungen werden getestet und beim Scheitern der Tests abgewiesen (vgl. [Fish04]).

Projektbeschreibung findet sich das grundlegende Wissen über das Projekt selbst bezüglich Quellverzeichnissen und Bibliotheken. Alle Aufgaben, die während eines *build*-Prozesses bewältigt werden müssen, werden *Maven* mittels *plug-ins* zur Verfügung gestellt. Die Entwickler müssen nicht mehr jeden einzelnen Schritt des *build*-Prozesses wie in *ANT* nachbilden, sondern geben den jeweiligen *Maven-plug-ins* die für diese Standardaufgaben nötigen Informationen mit.

- *NAnt*: ein *OpenSource-build*-Werkzeug, welches auf *Microsofts .NET*-Rahmenwerk basiert und die *ANT*-Funktionalität auf diese Plattform portiert.

CruiseControl bietet damit eine breite Unterstützung gängiger *build*-Werkzeuge und *Version Control Systems*. Die Verfügbarkeit beider notwendiger Komponenten für *builds* und Versionierung als *OpenSource*-Software begünstigt den Einsatz von *CruiseControl* und der Anwendung von *continuous integration* in einer zunehmenden Zahl von Softwareprojekten.

3.3. Die Ablaufsteuerung

Die Ablaufsteuerung von *CruiseControl* ist für die Vorbereitung sowie die Ausführung des *build*-Prozesses verantwortlich. Dabei sind die Konfiguration der Projekte, deren Aktualisierung und die projekteigenen *build*-Skripte voneinander getrennt. Alleine die Projektkonfiguration wird beim Start von *CruiseControl* geladen. Dadurch können das Aktualisierungsskript und die *build*-Skripte während des laufenden Betriebs angepasst werden. Änderungen wirken sich dann auf den folgenden *build*-Prozess des jeweiligen Projekts aus. Die Konfiguration und der Ablauf eines *builds* sind im Folgenden beschrieben.

3.3.1. Die Projektkonfiguration

Die Steuerung von *CruiseControl* beziehungsweise den von *CruiseControl* verwalteten Projekten geschieht über eine zentrale Konfigurationsdatei, die `config.xml`. Über diese Datei wird der Anwendung mitgeteilt, welche Projekte gebaut werden sollen und wo die dazugehörigen *build*-Skripte liegen.

Listing 3 a. enthält die grundlegenden Elemente der Projektkonfiguration unter *CruiseControl*. Zur besseren Referenzierbarkeit ist das Listing zusätzlich mit Zeilennummern versehen, die im Originalskript nicht vorkommen. Eine vollständige Auflistung aller Konfigurationselemente kann in der Anwendungsdokumentation nachgeschlagen werden (vgl. [CC]). Das Beispiel verwendet als ausführende Skriptsprache für die *build*-Skripte *ANT*. In der Konfiguration kann im `schedule`-Element (Zeile 16-23) aber ebenso ein Aufruf in einer der beiden alternativen Skriptsprachen *Maven* und *NAnt* erfolgen.

Diese Konfigurationsdatei enthält die für das Projekt unter *CruiseControl* benötigten Informationen. In einer Textdatei werden Status und Uhrzeit des aktuellen *builds* gesammelt. Ort und Name dieser Textdatei werden in der Konfiguration

angegeben (Zeile 8-10). Die Webanwendung greift später bei der Anzeige der einzelnen *build*-Zustände darauf zu. Für die Prüfung auf Änderungen in der Quellcodebasis wird das Projektverzeichnis angegeben, in dem sich eine ausgecheckte[1] Version des Projekts befindet (Zeile 12-14). Auf dieser Version wird dann durch Vergleich mit der aktuellen Version im Versioningsystem auf Änderungen geprüft, um dann gegebenenfalls ein *build* zu starten, wenn eine neue Version vorliegt. Zudem wird die Zeit in Sekunden angegeben, die *CruiseControl* zwischen zwei aufeinander folgenden Prüfungen wartet.

```
1  <cruisecontrol>
2
3    <!-- ================================================== -->
4    <!-- Projekt MyProject                                  -->
5    <!-- ================================================== -->
6    <project name="MyProject" buildafterfailed="false">
7
8      <bootstrappers>
9        <currentbuildstatusbootstrapper file="logs/MyProject/buildstatus.txt" />
10     </bootstrappers>
11
12     <modificationset quietperiod="180">
13       <cvs localworkingcopy="MyProject/checkout/MyProjectTools" />
14     </modificationset>
15
16     <schedule interval="60">
17       <ant antWorkingdir="/home/dittberner/ccwork/MyProject"
18            buildfile="build-MyProject.xml"
19            target="cruisecontrol"
20            uselogger="false"
21            usedebug="false"
22            multiple="1" />
23     </schedule>
24
25     <!-- log verzeichnis -->
26     <log encoding="utf-8" dir="logs/MyProject">
27       <merge dir="MyProject/checkout/MyProjectTools/ccreports/xml" />
28     </log>
29
30     <!-- publisher laufen nach dem build  -->
31     <publishers>
32       <currentbuildstatuspublisher file="logs/MyProject/buildstatus.txt" />
33       <artifactspublisher
34                         dir="MyProject/checkout/MyProjectTools/ccreports/"
35                         dest="artifacts/MyProject/" />
36     </publishers>
37   </project>
38 </cruisecontrol>
```

Listing 3. a: Ein Ausschnitt aus der `config.xml` eines Beispielprojekts

Für jedes Projekt existiert ein delegierendes Aktualisierungsskript, welches dafür sorgt, dass aktuelle Versionen der Komponenten, die das Projekt benötigt, aus

[1] So nennt man einen kompletten Abzug eines Projekts aus dem Versioningsystem. Der Begriff „auschecken“ lehnt sich an das entsprechende Kommando an, bei *CVS* zum Beispiel `cvs checkout`.

dem Versioning-System geholt werden. Dieses delegierende Skript ist aber nicht dasselbe Skript wie das projekteigene *build*-Skript, welches die *Targets*[1] für das Kompilieren und Ausführen der Tests beinhaltet (Zeile 17-22). Letzteres steht der Ausführung unter *CruiseControl* erst zur Verfügung, nachdem ein aktueller Abzug aus dem Versioning-System geholt wurde. Ein Sammelverzeichnis für das Logging (Zeile 25-28) und Reporting (Zeile 30-36) eines Projekts wird ebenfalls in der Konfiguration von *CruiseControl* mit angegeben.

Die Abtrennung von Aktualisierungsskript und *build*-Skript ermöglicht somit an dieser Stelle die Wahl zwischen den drei von *CruiseControl* unterstützten Alternativen *ANT*, *Maven* und *NAnt*.

3.3.2. Das Aktualisierungsskript

Das Aktualisierungsskript enthält alle Aufrufe, die dazu dienen, eine aktuelle Version der Projektquellen sowie der dazugehörigen Bibliotheken und sonstigen Ressourcen[2] in das Arbeitsverzeichnis von *CruiseControl* zu kopieren.

```
1  <!--
2   Delegierendes build-script fuer Cruisecontrol -->
3  <project name="build-myproject" default="build"
4           basedir="/home/dittberner/ccwork/MyProject/checkout">
5
6   <property name="gui.dir" location="${basedir}/MyProjectGUI" />
7   <property name="tech.dir" location="${basedir}/MyProjectTech" />
8   <property name="tools.dir" location="${basedir}/MyProjectTools" />
9
10  <target name="cruisecontrol">
11
12    <cvs dest="${gui.dir}"
13         compressionlevel="9"
14         quiet="true"
15         command="update -P -C -R -d"  />
16
17    <cvs dest="${tech.dir}"
18         compressionlevel="9"
19         quiet="true"
20         command="update -P -C -R -d"  />
21    <echo>CVS ist fertig, starte das ANT-Target cruisecontrol </echo>
22
23    <!-- Das eigentliche build aufrufen -->
24    <echo>Starte delegierendes build-Skript</echo>
25    <echo>Rufe Target cruisecontrol auf</echo>
26    <ant dir="${tools.dir}"
27         antfile="build.xml" target="cruisecontrol"/>
28    <echo>Fertig!</echo>
29  </target>
30</project>
```

[1] *Target* wird eine einzelne, benutzerdefinierte Ausführungseinheit in *ANT* genannt. Ein *Target* kann aus einem oder mehreren *Tasks* (atomare Aufgaben in *ANT*) bestehen sowie andere abhängige *Targets* aufrufen.

[2] Dazu gehören zum Beispiel Dokumentationsdateien, Dokumententemplates, Konfigurationsdateien, Grafiken, Testdaten, Datenstrukturbeschreibungen etc.

Listing 3. b: Ein Ausschnitt aus der `build-project.xml` eines Beispielprojekts[1]

Listing 3. b enthält einige beispielhafte Aufrufe. Den Skriptzeilen sind wieder wie in Listing 3. a Zeilennummern vorangestellt, um einzelne Zeilen besser referenzieren zu können. Auch im Aktualisierungsskript gibt es diese im Original nicht.

Zuerst werden die Projekte aktualisiert, von denen das zu bauende Projekt abhängt (im Beispiel wären das die Komponenten `MyProjectGUI` und `MyProjectTech`, Zeilen 13-16 und 18-21). Dann wird im Projektverzeichnis das projekteigene *build*-Skript aufgerufen. Dort ist ein separates `cruisecontrol`-*Target* definiert, welches alle nötigen Arbeitsschritte des *builds* zusammenfasst und ausführt (Zeilen 27-28).

Diese zweischrittige Ausführungsfolge unter *CruiseControl* (erst Aktualisierungsskript, dann Projekt-*build*-Skript) ermöglicht es, dass die einzelnen Arbeitsschritte des *builds* in der projekteigenen `build.xml` definiert sind und auch dort eingepflegt werden. Dadurch können Änderungen und Erweiterungen des Prozesses jederzeit durch die Entwickler am Entwicklerplatz vorgenommen und ebenso wie die Quelldateien ins *Repository* eingespielt werden. Bei der nächsten Aktualisierung kopiert *CruiseControl* auch das geänderte *build*-Skript ins Arbeitsverzeichnis, so dass immer das aktuelle *build*-Skript durch *CruiseControl* aufgerufen wird.

Das in Listing 3. b dargestellte Skript enthält beispielhaft Aufrufe an ein *CVS-Repository*. An deren Stelle können auch die in Kapitel 3.2.5. beschriebenen alternativen Versioning-Systeme aufgerufen werden.

3.3.3. Der automatische Ablauf

Der automatische Ablauf eines *builds* unter *CruiseControl* gestaltet sich folgendermaßen – der Einfachheit halber sind in den einzelnen Schritten die Kommandos des Versioning-Systems *CVS* angegeben; bei Verwendung anderer Versioning-Systeme kommen die entsprechenden Kommandos zum Einsatz:

1. *CruiseControl* lädt die projektspezifischen Informationen über die mit einem Projekt assoziierten *CVS*-Module aus der Projektkonfiguration. Auf diesen prüft *CruiseControl* regelmäßig durch ein *CVS LOG*-Kommando (beziehungsweise dem Pendant in anderen Versioning-Systemen) auf Änderungen in der Quellcodebasis. *builds* können dementsprechend immer oder nur bei Änderungen am Projekt durchgeführt werden.

2. Soll ein *build* angefertigt werden, so beschafft sich *CruiseControl* über *CVS UPDATE* den aktuellen Stand von Quellcode, der Bibliotheken sowie der übrigen, für den *build* notwendigen Dateien. Im Aktualisierungsskript

[1] Die `build-project.xml` ist das delegierende Skript für *CruiseControl*; dieses ruft wiederum das *build*-Skript des Projekts auf – delegiert also den build-Prozess.

sind die einzelnen Module und die darauf auszuführenden *CVS UPDATE*-Operationen beschrieben.

3. Auf dem aktuellen Stand führt *CruiseControl* dann anhand des projekteigenen *build*-Skripts den *build* aus. Der Name des projekteigenen *build*-Skripts sowie das *ANT*-Target, welches den eigentlichen *build* durchführt, sind im oben genannten Aktualisierungsskript beschrieben. Von dort aus wird das *build*-Skript nach der Aktualisierung aufgerufen. Dabei wird der Quellcode kompiliert und gegebenenfalls als Projektarchiv, Webarchiv (WAR) oder Serverarchiv (EAR) verpackt. Auf dem kompilierten Projekt werden dann die Tests ausgeführt – je nach Aufbau des *build*-Prozesses nur Teile oder der komplette Satz an Tests, den das Projekt beinhaltet: Unittests sowie langwierige Integrationstests. Die Aufgaben, die während des *build*-Prozesses abgearbeitet werden, werden durch die Entwickler vorgegeben. So können während des *builds* zusätzliche Arbeitsschritte eingebaut werden, um die Testabdeckung zu ermitteln oder die Architektur zu analysieren. Alle benutzerdefinierten Aufgaben finden im Aufruf des projektspezifischen *build*-Skripts statt.

4. Nach Abschluss der Tests und der weiteren Aufgaben des *build*-Prozesses sammelt *CruiseControl* die Ergebnisse zusammen und erstellt einen Abschlußbericht sowie ein Ergebnispaket. Dieses Paket wird auch von der Webanwendung benutzt, um die Ergebnisse des *builds* für die Entwickler aufzubereiten und darzustellen.

5. Die während des *build*-Prozesses generierten XML-Dateien werden über XSLT (*XSL Transformations*, beschrieben in [XSLT]) in lesbare Ergebnisse transformiert.

Änderungen am oben genannten Aktualisierungsskript und an den projekteigenen *build*-Skripten werden bei jedem Start des *build*-Prozesses neu geladen und werden somit ohne Neustart aktiv. So können jederzeit von den Entwicklern die Arbeitsschritte eines *builds* angepasst werden. Ebenso können einzelne *CVS*-Module dem Projekt hinzugefügt werden oder aus diesem entfernt werden, ohne dass *CruiseControl* neu gestartet werden muss.

Neue Projekte werden in der Konfigurationsdatei `config.xml` (oder auch `configuration.xml`[1]) von *CruiseControl* definiert. Die Konfigurationsinformationen über die von *CruiseControl* verwalteten Projekte werden nur einmal beim Start von *CruiseControl* geladen. Änderungen an diesen Informationen kommen ohne Neustart nicht ins System, das heißt, dass Projekte im laufenden *CruiseControl* weder hinzugefügt noch entfernt werden können.

[1] Der Name der Konfigurationsdatei kann frei gewählt werden und wird *CruiseControl* beim Start der Anwendung mitgegeben.

3.3.4. Einbindung in den Entwicklungsprozess

CruiseControl ist über die mitgelieferte Webanwendung in den Entwicklungsprozess eingebunden. Die Entwickler können das Ergebnis des *builds*, der auf die Integration von neuen oder geänderten Quelldateien folgt, anhand der in dieser Anwendung dargestellten Informationen anschauen. Im Entwicklungsprozess kann das Team damit direkt auf unerwünschte Auswirkungen der Integration wie nicht kompilierbare Quelldateien oder fehlschlagende Tests reagieren. Vor allem haben Entwickler in einem aus mehreren Teilprojekten bestehenden Softwareprojekt über die Webanwendung die Möglichkeit, die *build*-Ergebnisse der jeweils anderen Teilprojekte einzusehen. Scheitern abhängige Teilprojekte, so können Entwickler darüber nachvollziehen, ob die von ihnen integrierten Änderungen dafür verantwortlich sein können.

Für die Einbindung in die Entwicklung lassen sich im Wesentlichen zwei unterschiedliche Arten von zu integrierendem Code betrachten: Änderungen und Neuentwicklungen. Die Auswirkungen durch die Einspielung der jeweiligen Form des Quelltextes ins *Repository* sind unterschiedlich weitreichend. Während Neuentwicklungen zumindest noch nicht von anderen abhängigen Teilprojekten genutzt werden können, sind Änderungen an Komponenten, von denen andere Komponenten abhängen, bei größeren Projekten schwieriger zu überblicken.

Neuentwicklungen können durch die eingeschränkte „Reichweite“ ihrer Auswirkungen nur die Komponente in einen inkonsistenten Zustand versetzen, in denen sie eingebunden sind. Änderungen hingegen können auch direkt den Zustand der *builds* abhängiger Komponenten beeinflussen.

Abhängig davon ist nicht nur das eigene Projekt in *CruiseControl* zu beobachten, sondern gegebenenfalls auch die abhängigen Projekte. Änderungen können diese in einen inkonsistenten Zustand überführen, auch wenn in die Projekte selbst keine Änderungen eingespielt wurden. Die Entwickler achten nach dem Einspielen von aktuellen Quelldateien auf die Informationen des nachfolgenden *builds* in *CruiseControl*. Treten Fehler während des Kompilierens auf oder schlagen die Tests fehl, können die Entwickler anhand der Fehler- und Testberichte der Webanwendung die fehlerhafte Stelle im Code identifizieren und ausbessern. Die Testberichte unter *CruiseControl* enthalten dafür dieselben Informationen, die das protokollierte Ausführen der Tests unter *JUnit* erzeugt.

Zusätzlich zur Kontrolle des *build*-Status über die Webanwendung kann *CruiseControl* die Entwickler noch über E-Mail über erfolgreiche und gescheiterte *builds* informieren. Dies ist besonders interessant, wenn nicht alle Entwickler Zugriff auf den *build*-Status beziehungsweise die Webanwendung haben. In *OpenSource*-Projekten können so zum Beispiel einzelne Entwickler mit administrativem Zugriff auf das Projekt darüber informiert werden, wenn etwas mit dem aktuellen *build* nicht stimmt, und entsprechende Maßnahmen einleiten. Des Weiteren ermöglicht die Benachrichtigung über E-Mail den Anschluss weiterer Kommunikationsmedien, zum Beispiel der Statusmeldung per SMS.

Es sei hier noch eine Alternative zur Webanwendung erwähnt, wie der aktuelle *build*-Status eines Projekts dem Team sehr effektvoll visualisiert werden kann: Clark (siehe [Cla04]) beschreibt in seinem Buch über Projektautomation ein kleines, einfach gebautes System von Lavalampen. Diese sind über funkgesteuerte Steckdosen und einer speziellen Steuerungssoftware mit dem *CruiseControl*-System

verbunden. Eine grüne Lavalampe dient dazu, erfolgreiche *builds* anzuzeigen, eine rote Lavalampe symbolisiert gescheiterte *builds*. Je nach Status wird über die Steuerung entweder die Steckdose der grünen oder die Steckdose der roten Lavalampe aktiviert. Der Zustand des *builds* kann somit gut sichtbar in den Entwicklerräumlichkeiten platziert werden. Die Lavalampen sind dabei nur eine Möglichkeit der Darstellung; in dem Beispiel von Clark wird auch die Idee einer Kopplung von *build*-Status und Tondateien erwähnt. Wie in Kapitel 3.1. schon beschrieben, gibt es zudem noch *Browserplug-ins* und Monitoranwendungen, mit denen der Zustand unter *CruiseControl* darstellen lässt, hauptsächlich in Form von roten und grünen Icons.

3.4. Zusammenfassung

CruiseControl läuft als eigenständige Anwendung, vorzugsweise auf einem dedizierten Rechner. Dieser benötigt Zugriff auf das *Version Control System Repository*, um sich für die *builds* aktuelle Versionen der unter *CruiseControl* eingerichteten Projekte besorgen zu können. Auf den während des *builds* generierten Informationen arbeitet eine Webanwendung, um die Ergebnisse des *builds* für die Entwickler aufzubereiten. Für die Webanwendung wird ein Webserver benötigt, auf den die Entwickler von einem ausgewiesenen Rechner oder aber ihrem Arbeitsplatz aus Zugriff haben.

CruiseControl arbeitet auf einem *build*-Werkzeug, mit dem in definierte *build*-Skripte ausgeführt werden können. Neben *ANT* werden auch *NAnt* und *Maven* durch *CruiseControl* unterstützt. Dabei bieten *ANT* und *Maven* zwei unterschiedliche Ansätze, um den *build*-Prozess zu definieren. Mit *ANT* wird der gesamte *build*-Prozess aus elementaren *Tasks* aufgebaut, während *Maven* eine Beschreibung des Projekts nutzt, um allgemeine *build*-Aufgaben erledigen zu können. Die Palette der Aufgaben kann dabei über *plug-ins* erweitert werden.

Um an die aktuellen Versionen von Quelldateien und Bibliotheken zu gelangen, benutzt *CruiseControl* das in verteilt entwickelnden Projekten eingesetzte Versioning-System. Die Versioning-Komponente wird dazu verwendet, um in regelmäßigen Abständen zu prüfen, ob ein *build* angestoßen werden muss. Liegen neue oder aktualisierte Quelldateien vor, wird der *build*-Prozess gestartet.

Die Konfiguration der Projekte, die unter *CruiseControl* gebaut werden, geschieht nach Konvention mittels zweier Konfigurationsdateien. In einer Datei sind die einzelnen Projekte, ihre Arbeitsverzeichnisse und weitere Informationen wie Prüfintervall und Logging hinterlegt. Eine weitere Datei enthält die Aktualisierungsinformationen, anhand derer *CruiseControl* die Teilprojekte aktualisiert, von denen das jeweilige Projekt abhängt. Zudem enthält dieses delegierende Skript den Aufruf im projekteigenen *build*-Skript, welches schließlich den *build*-Prozess steuert. Durch diese Ausführungshierarchie kann der *build*-Prozess von den Entwicklern innerhalb des Projekt-*build*-Skripts am Entwicklerarbeitsplatz angepasst werden, ohne die Konfiguration von *CruiseControl* abändern zu müssen.

Die Entwickler nutzen die Kommunikationskanäle von *CruiseControl*, um die Auswirkungen ihrer Änderungen und Neuentwicklungen auf den Zustand des *builds* zu beobachten. Bei Neuentwicklungen beschränken sich diese Auswirkungen

vornehmlich auf das eigene Projekt, bei Änderungen können sich diese auf abhängige Projekte ausweiten. Dementsprechend können bei Letzteren im Fehlerfall auch die *builds* der abhängigen Teilprojekte unter *CruiseControl* scheitern. Wenn ein build scheitert, können die Entwickler anhand der durch die Webanwendung aufbereiteten *build*-Informationen Fehleranalyse betreiben und die Fehler korrigieren beziehungsweise das verantwortliche Teilprojekt informieren.

4. CruiseControl in der Anwendung

4.1. Projekt A (Elvis)

Bei dem ersten Beispielprojekt handelt es sich um ein Softwaresystem zur Unterstützung der Betriebsführung eines großen regionalen Versorgungsunternehmens. Das Softwareprojekt unterstützt derzeit vier Geschäftsbereiche und besteht insgesamt aus zwölf Teilkomponenten. Die Komponenten erreichen zusammen eine Größe von über einer Million Zeilen Code, an denen etwa 20 Entwickler arbeiten. Die Entwicklung findet an 10 Rechnerarbeitsplätzen statt, die über das firmeninterne Netz miteinander und mit den Servern[1] verbunden sind. Von allen Rechnerarbeitsplätzen können Änderungen in das System über das *CVS-Repository* gelangen.

Das Softwaresystem ist nach der *J2EE*-Architektur[2] aufgebaut. Die serverseitigen Komponenten greifen dabei auf Datenbanken und Dateien zurück. Es bestehen weiterhin Verbindungen zu weiteren Drittsystemen wie *MapXtreme* als GIS[3] und *SAP/R3* als kaufmännisches Backendsystem. Als Datenbank kommt *DB2* von *IBM* zum Einsatz, als Applikationsserver *Websphere* von *IBM*.

4.1.1. Herausforderungen im Projekt

Das *Elvis*-Projekt besteht aus einer ganzen Reihe von eigenständigen Teilprojekten. Es bestehen Abhängigkeiten zwischen einigen dieser Teilprojekte. Beim Durchlaufen eines kompletten *build*-Zyklus müssen somit diese Abhängigkeiten berücksichtigt werden. Neue Versionen von Komponenten, die von anderen Komponenten verwendet werden, müssen im *build*-Prozess zu Beginn gebaut werden. Diese neuen Versionen werden beim Kompilieren und Testen von abhängigen Teilprojekten benutzt. Schließlich treten Änderungen von Schnittstellen zutage, die nicht vollständig abgeschlossen oder fehlerhaft umgesetzt wurden.

In Abb. 4. a sind die Teilprojekte des *Elvis*-Projekts und ihre Abhängigkeiten voneinander grafisch dargestellt. Alle aufgeführten Teilprojekte werden dabei im Laufe des Projekts weiterentwickelt. Verwendete Drittanbieterkomponenten sind in der Grafik nicht aufgeführt. Zwar sind auch diese Teile der Software nicht statisch und Änderungen unterworfen, aber beschränken sich die Anpassungen seitens des Projekts nur auf die aufgeführten Teilprojekte. Ändert sich die Komponente eines Drittanbieters, so kann zu einem definierten Zeitpunkt die neue Version dieser Komponente integriert werden. Die Komponenten, die durch die Entwickler im Projekt entwickelt werden, bringen ihre Anpassungen dahingegen kontinuierlich in das Gesamtprojekt ein.

---

[1] Dazu gehören unter anderem die Datenbankserver, die Webserver, die Applikationsserver, der CVS-Server sowie der dedizierte *CruiseControl*-Rechner.
[2] *J2EE*: „Java 2 Platform, Enterprise Edition“
[3] *GIS - Geographical Information System*.

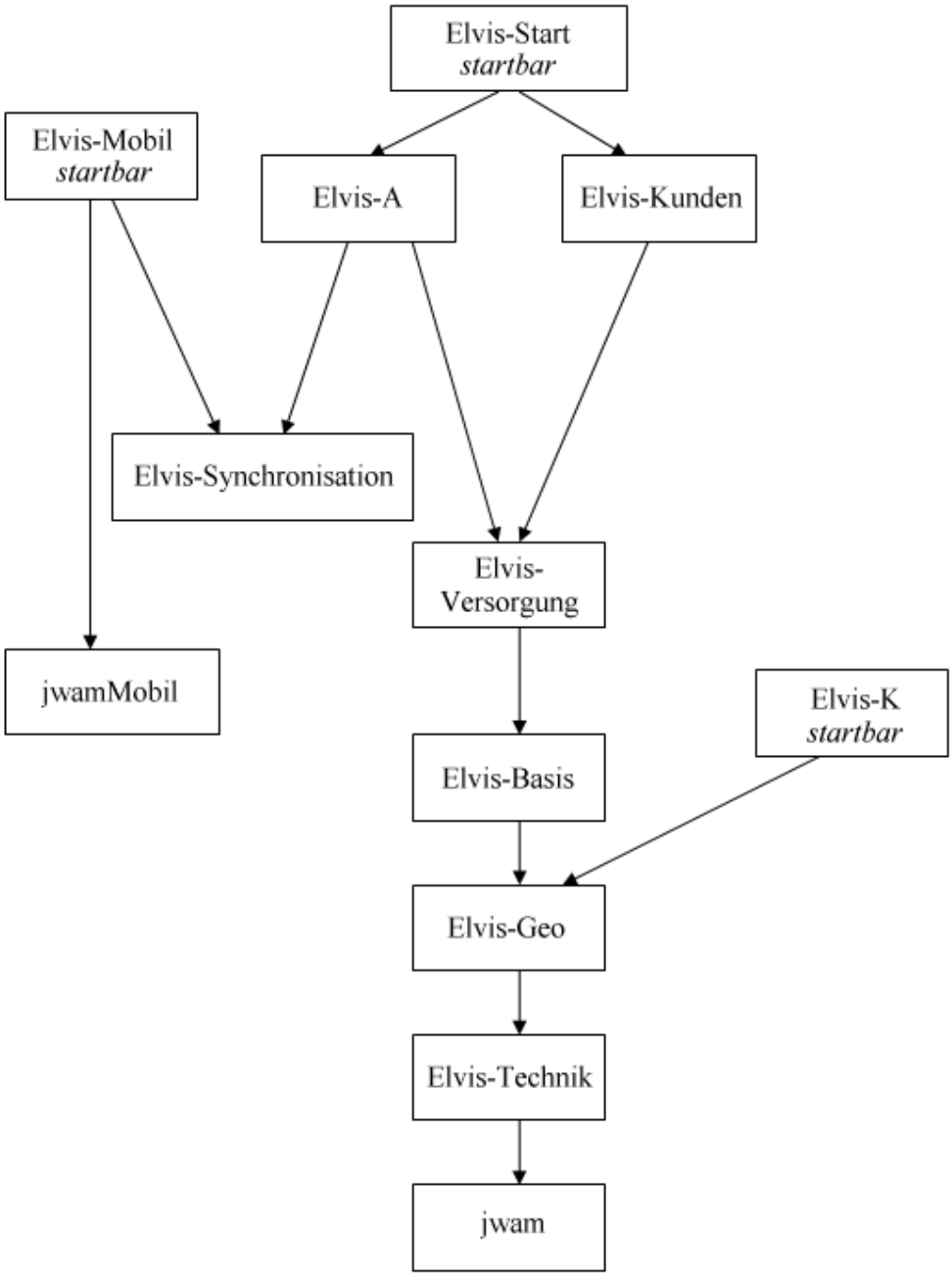

Abb. 4. a: Struktur im Elvis-Projekt; Abhängigkeiten zwischen den Teilprojekten

4.1.2. Ziele des Einsatzes von CruiseControl

Die im *Elvis*-Projekt realisierten Anforderungen der verschiedenen Geschäftsbereiche müssen Anwendern regelmäßig zum Testen und Bewerten der Software zur Verfügung gestellt werden. Zu Beginn des *Elvis*-Projekts bestand dieses nur aus wenigen Komponenten und wurde immer als Gesamtprojekt präsentiert. Der build-Prozess für diesen Anwendungsblock bestand aus einigen

wenigen, einfachen Aufgaben, da nur das gesamte Projekt kompiliert und getestet wurde. Die wenigen Komponenten, von denen das *Elvis*-Projekt abhing, wie zum Beispiel das verwendete *JWAM*-Rahmenwerk (siehe [JWAM]), unterlagen dabei nur marginalen Änderungen und stellten sich somit für das Projekt und den *build*-Prozess als nahezu statisch dar. Anfangs wurde *continuous integration* noch ohne zusätzliche Werkzeugunterstützung realisiert. Die Entwickler waren beim Einspielen von Änderungen und Neuerungen ins System angehalten, erst den Abgleich mit dem *Repository* zu machen. Bevor dann die Änderungen in das System übernommen wurden, führten die Entwickler alle Tests aus. Wenn dabei Fehler auftraten, wurden diese beseitigt, bevor die geänderten Quelldateien in das *CVS* übertragen wurden. Mit zunehmender Projektgröße und –komplexität wuchs der Zeitaufwand der Entwickler für den Abgleich mit dem *Repository* sowie der Anpassung der jeweiligen Quelldateien an die zwischenzeitlichen Änderungen der anderen Entwickler. Es kam häufiger zu Überschneidungen, die wiederum die nachträgliche Anpassung von geänderten Quelldateien aufwändiger machten.

Um trotz zunehmender Komplexität und der Aufteilung des großen Einzelprojekts in mehrere Teilprojekte den Qualitätsanforderungen weiterhin gerecht werden zu können, entschied sich das Projekt, die *continuous integration* mithilfe von *CruiseControl* zu unterstützen. Vor allem sollte das die Weiterentwicklung von Komponenten absichern, die sich im Kern des Systems befanden und von denen somit alle anderen Komponenten beziehungsweise Teilprojekte abhängig waren. Schließlich sollten durch den automatisierten *build*-Prozess auch automatisch *Tracking*-Informationen[1] über die Testergebnisse und Testabdeckung gesammelt werden, um die Effekte von qualitätssichernden Maßnahmen bewerten zu können. Langfristig wollte man damit die Entwicklung und Einhaltung der beobachteten Parameter (sog. *Metriken*, vgl. [LRW05], S. 202-206) im Blick behalten.

Die Aufteilung des Projekts auf verschiedene Teilprojekte bringt einen zusätzlichen Aspekt in den *build*-Prozess: die Ausführung mehrerer von einander abhängiger *builds*. Dadurch werden für den Prozess die neuesten Versionen der in Entwicklung befindlichen Systemteilen zur Verfügung gestellt, auf denen das eigene Teilprojekt aufbaut. Der Einsatz von *CruiseControl* im *Elvis*-Projekt sollte die Ausdehnung des automatisierten *build*-Prozesses auf alle Teilprojekte ermöglichen.

4.1.3. Arbeitsabläufe

Nach der Einführung von *CruiseControl* wurde der Projektzustand sehr genau beobachtet, wenn neue oder geänderte Quelldateien oder Bibliotheken in das System eingespielt wurden. Während im Hintergrund dann der *build*-Prozess anlief, wurde an den Entwicklerplätzen aber schon weiter entwickelt. Das vorherige Warten auf die Ausführung der gesamten Tests entfiel durch den Einsatz von *CruiseControl*. In regelmäßigen Abständen wurde dann kurz auf den *build*-Status geschaut. Trat durch die Änderungen ein Fehler auf, so wurde nach Beendigung der aktuellen Tätigkeit ein Bugfixing vorgenommen und zusammen mit den zusätzlichen

[1] In dem Buch von Lippert, Roock und Wolf (siehe [LRW05], S. 202-206) wird das *Tracking* über angeforderte, offene und erledigte Aufwände als der wichtigste Wert für das Projektcontrolling vorgestellt. Die Kennzahlen über erfolgreiche und gescheiterte Tests und über die Testabdeckung liefern weitere wichtige Informationen über den Ist-Zustand des Projekts.

Änderungen in das System eingespielt. Im Laufe des Projekts haben sich Integrationsintervalle und die Abstände zwischen zwei Kontrollen des *build*-Status vergrößert. So wird in der aktuellen Projektphase teilweise einmal bis zweimal am Tag der Projektstatus geprüft. Treten Fehler auf, so werden diese meist erst mit einiger Verzögerung korrigiert oder auf Phasen verschoben, in denen weniger Zeitdruck auf möglicherweise wichtigere Projektteile besteht. Wird dann schließlich ein Bugfixing eingespielt, wird in der Folge der *build*-Status häufiger geprüft, ob die Änderungen den Fehler tatsächlich beseitigt haben oder nicht. Zwar werden bei auftretenden Fehlern die Tests auch lokal ausgeführt, vor sowie nach dem Bugfix, um schneller an ein Ergebnis gelangen zu können. Allerdings soll somit auch sichergestellt werden, dass es sich bei Fehlern nicht um solche handelt, die nur in der besonderen Umgebung der Entwicklerarbeitsplätze vermieden werden oder sich aus anderen Gründen während des *build*-Prozesses manifestieren. Um durch die Testausführung die Entwicklung nicht zu unterbrechen, wird meist auf die lokale Ausführung der Integrationstests verzichtet. Diese werden weiterhin durch den Integrationsserver gefahren.

Änderungen im *build*-Skript können zu Fehlern im *build*-Prozess führen. Selbst die Tests können unter *CruiseControl* scheitern, während sie lokal durchlaufen. Solche Fehler sind auf Änderungen oder Neuerungen im *build*-Prozess zurückzuführen ebenso wie auf hinzugefügte Bibliotheken, die noch nicht im Klassenpfad der Tests auf dem Server liegen. Der Zeitraum zwischen Entdeckung und Korrektur solcher Fehler ist größer als bei Fehlern in Tests, da sich Fehler im *build*-Prozess nicht unbedingt im Scheitern einzelner Tests manifestieren, sondern eher weniger verständliche bis kryptische Fehlermeldungen unter *CruiseControl* verursachen. Die Fehlersuche und –analyse nimmt dabei in der Regel mehr Zeit in Anspruch als Fehler im Quellcode. Dementsprechend wird die Korrektur aufgeschoben, bis mehr Zeit zur Verfügung steht oder der Fehler korrigiert werden muss, um weiter entwickeln zu können. Der *Broken-window*-Theorie[1] folgend kann dieser Aufschub zu einer weiteren Verschlechterung des Codezustands führen. Der scheiternde *build*-Prozess verdeckt dabei neu hinzukommende Fehler, da sich der *build*-Status nun nicht mehr von *erfolgreich* nach *gescheitert* ändert.

Im Teilprojekt *Elvis-Kunden*-Webanwendung wurde zu Beginn ein besonderer Arbeitsablauf eingeführt, um den Zustand der Tests, der Testabdeckung sowie der Architektur zu verbessern. In so genannten *nightly-builds*[2] wurde unter anderem einmal täglich die Testabdeckung des Teilprojekts bestimmt sowie die Architektur bezüglich der Beziehungen der einzelnen Schichten untereinander geprüft. Die Testabdeckung im Projekt wurde mit dem Werkzeug *JCoverage* (siehe [JCov]) bestimmt. Die Prüfung der Architektur auf Architekturverletzungen erfolgte mit *xRadar* (siehe [xRadar]). Der neue Arbeitsablauf sah nun vor, dass der Arbeitstag entweder mit einem Bugfixing, dem Schreiben eines Tests oder der Behebung einer Architekturverletzung beginnt. Die von *JCoverage* erzeugte Testabdeckung enthält

[1] Die *Broken-window*-Theorie geht auf den Politikwissenschaftler James Q. Wilson und den Kriminologen George L. Kelling zurück und beschreibt, wie ein eher harmloser Defekt an einem leerstehenden Haus, nämlich besagtes kaputtes Fenster, zu einer zunehmenden Verwahrlosung des Hauses führen kann. Als Analogie dazu kann das Scheitern des *builds* durch einen einzigen Fehler gesehen werden, dessen längerfristiger Verbleib im System gegebenenfalls weitere Fehler unbemerkt in den Code einlässt – den Code zunehmend *verwahrlosen* lässt.

[2] Das Ermitteln von Testabdeckung und von Strukturen im Projekt nimmt besonders viel Zeit in Anspruch. Um tagsüber trotz der häufigen Integration von Änderungen in den Teilprojekten möglichst kurze Antwortzeiten beizubehalten, wurden diese Aufgaben im automatisierten Prozess in die Nacht verlegt. Daher die Bezeichnung *nightly-builds*.

dafür neben der Gesamtabdeckung eine detaillierte Auflistung der Abdeckung einzelner Pakete sowie der Abdeckung einzelner Klassen und deren Methoden. Anhand dieser Informationen konnte somit gezielt ungetesteter Code identifiziert werden und dafür ein Test geschrieben werden. Diese Vorgehensweise ist dem Umstand geschuldet, dass zum einen nicht durchweg nach der *Test-First*-Methode entwickelt wurde sowie Teile des Projekts aus der Prototypenphase übernommen worden waren. *Bugfixes* konnten einfach anhand der Testprotokolle aus dem aktuellen *build* vorgenommen werden. Die Architekturprüfung durch *xRadar* lieferte schließlich tabellarische Auswertungen der Beziehungen zwischen den einzelnen Komponenten und dabei festgestellter Verletzungen. Anhand dessen konnten dann die Klassen, die nicht erlaubte Zugriffe enthielten, entweder korrigiert oder aber verschoben werden, um die Architekturverletzung zu beseitigen.

4.1.4. Welche Ziele wurden erreicht

Datum	Elvis-I	Elvis-K	Elvis-Kunden	Elvis-A	Elvis-V	Elvis-Technik
02.05.05	66,00%	70,00%		58,00%	51,00%	
09.05.05	66,00%	73,00%		57,00%		
17.05.05	66,00%			58,00%	51,00%	
24.05.05				57,00%	51,00%	
31.05.05		76,00%		58,00%		43,00%
07.06.05					55,00%	44,00%
14.06.05				58,00%	57,00%	41,00%
21.06.05	62,00%	78,00%	16,00%	58,00%	57,00%	41,00%
28.06.05	64,00%	78,00%	17,00%	58,00%	58,00%	41,00%
04.07.05	64,00%	78,00%	19,00%	58,00%		41,00%
11.07.05	66,00%	78,00%	19,00%	59,00%		45,00%
18.07.05			22,00%	59,00%		
25.07.05			31,00%	60,00%		
01.08.05				60,00%		
08.08.05				60,00%		41,00%
15.08.05						
22.08.05		81,00%				42,00%

Tabelle 4. a: Entwicklung der Kennzahl Testabdeckung im *Elvis*-Projekt

In Tabelle 4. a sind beispielhaft einige Teilprojekte und die ermittelte Testabdeckung aufgeführt. Dabei ist pro Datum jeweils eine Abdeckung pro Teilprojekt in Prozent eingetragen. Die leeren Stellen bedeuten, dass für das jeweilige Teilprojekt zum Messzeitpunkt die Testabdeckung nicht ermittelt werden konnte. Das ist überwiegend aus einen der zwei folgenden Gründe der Fall: a) der Wert wurde nicht gemeldet. Umfang und Komplexität des Projekts verlangen, dass die Ermittlung der einzelnen Kennzahlen in der Verantwortung der einzelnen Teilprojekte liegt. Ein Qualitätsbeauftragter sammelt dabei die Kennzahlen des Projekts ein und wertet diese zentral aus. Die Ermittlung kann aber b) durch ein fehlerhaftes *build*-Skript oder aber ein scheiterndes *build* verhindert werden. Da die Werte nur zu bestimmten, festen Zeiten - einmal wöchentlich – ermittelt werden, wird

der Umstand eines fehlenden Wertes als leeres Feld notiert. Alternativ kann an diesen Stellen der letzte bekannte Wert angenommen werden, aber ermöglicht diese Notation eine Unterscheidung zwischen gleich gebliebener Kennzahl und nicht ermittelter Kennzahl. Im *Elvis*-Projekt sind die ausbleibenden Messungen hauptsächlich der zweiten Variante b) zuzuschreiben. Im Projekt ist ein Entwickler benannt, der sich explizit um das *Tracking* und das damit verbundene Einsammeln der benötigten Informationen kümmert. Wenn ein *build* scheitert und somit einige der benötigten Daten nicht zur Verfügung stehen, liegt das in der Verantwortung des jeweiligen Teilprojektteams. Das Fehlen von Messwerten liegt somit mehr in nicht ermittelten denn in vorhandenen, aber nicht gemeldeten Informationen begründet.

Eine Maßnahme zur Qualitätssteigerung konzentriert sich zum Beispiel auf die Erstellung und Vervollständigung der Tests im Teilprojekt. Ist ein Projekt unzureichend mit Tests ausgestattet, so können systematisch Klassen und Klassenpfad nach nicht getesteten Stellen durchsucht werden. Eine weitere Möglichkeit bieten Testabdeckungswerkzeuge wie *JCoverage* [JCov]. *JCoverage* liefert neben einer statistischen Auswertung der prozentuellen Abdeckung von Klassen und Paketen (engl. *packages*) auch eine Ansicht auf die Codeabdeckung. Mit Hilfe dieser Ansicht können die Entwickler sehen, welche Teile des Codes beim Testen mitgetestet wurden und welche während der Tests nie durchlaufen werden. Mit dieser Information lassen sich gezielt die vorhandenen Tests anpassen und erweitern. Dabei ist aber zu beachten, dass schon abgedeckter Code möglicherweise nicht richtig getestet wird, wie in Kapitel 2 beschrieben. Die Ermittlung der Testabdeckung alleine ersetzt also keinen systematischen Ansatz zur Erstellung, Pflege und Erweiterung von Tests in einem Projekt. Der *Test-First-Ansatz*[1] bietet hierbei eine Möglichkeit, von vornherein den Code vollständig und systematisch zu testen.

Das Teilprojekt *Elvis-K* zeigt dabei, wie eine Maßnahme zur Erhöhung der Testabdeckung zu einer kontinuierlichen Steigerung der Abdeckung führt. In diesem Fall wurde zur Erhöhung der Abdeckung eine zusätzliche Regel in den Entwicklungsprozess eingeführt: neben dem regulären Schreiben von Tests während der Weiterentwicklung sollte jeder Arbeitstag damit beginnen, einen fehlenden Test zu schreiben oder sich um einen fehlschlagenden Test zu kümmern. Das schließt mit ein, zu Beginn des Tages als Erstes den aktuellen Stand über die Webanwendung von *CruiseControl* abzufragen und sich gegebenenfalls über die ermittelte Testabdeckung eine Stelle im Code auszusuchen, für die noch kein oder nur ein unzureichender Test existiert. Wird diese Strategie konsequent im Teilprojekt verfolgt, kann eine Steigerung der Abdeckung, wie in der Statistik zu ersehen, erreicht werden.

Gleichbleibende oder sogar rückläufige Abdeckung sind damit zu erklären, dass zum einen trotz zunehmender Funktionalität zumindest soweit auch Tests implementiert werden, dass die Abdeckung auf gleichem Niveau bleiben kann. Ebenso werden auch trotz rückläufiger Zahlen kaum Tests aus dem Projekt entfernt, sondern eher für neue Klassen und Methoden keine Tests geschrieben. Bestehende und getestete Funktionalität bleibt in der Regel auch getestet. Ebenso beeinflusst das Verschieben von Klassen eines Teilprojekts in ein anderes unter Umständen die Abdeckung der einzelnen Projekte. In Ausnahmefällen werden Tests auch

[1] Der Test-First-Ansatz besagt, dass der Implementierung neuer Funktionalität oder der Änderung bestehender Funktionalität das Schreiben des dazugehörigen Tests vorausgeht (siehe [Bec05], S. 50-51).

auskommentiert, wenn sich die Behebung eines Fehlers kurzfristig als zu kompliziert erweist oder die Entwickler entscheiden, dass der Test infolge eines geplanten *Refactorings* obsolet wird.

Inzwischen haben weitere Teilprojekte eine Testabdeckung um die 80 % erreicht. Vor allem neuere Teilprojekte wurden dabei von Anfang an stärker nach dem *Test-First*-Ansatz entwickelt und zeigen nun eine hohe prozentuale Abdeckung. *CruiseControl* erlaubt durch die Ermittlung des *build*-Status für diese Projekte nun einen guten Überblick über den Gesamtzustand des Projektcodes.

Ein weiteres wichtiges Ziel, welches mit dem automatisierten *build*-Prozess verfolgt wurde, war die kontinuierliche Bereitstellung konsistenter *builds*. *CruiseControl* stellt zwei verschiedene Auswertungen der im Projekt erfolgten *builds* zur Verfügung, um einen Überblick über erfolgreiche und gescheiterte *builds* zu geben.

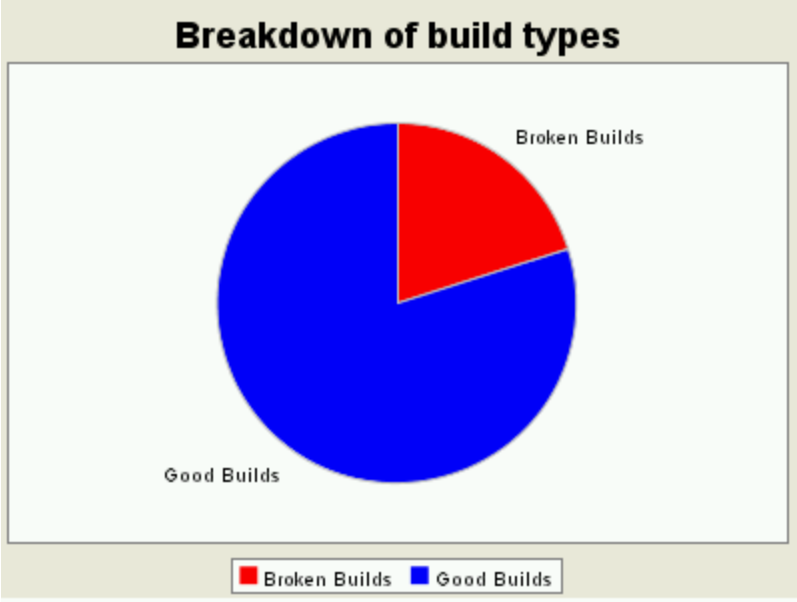

Abb. 4. b: Verhältnis von erfolgreichen zu gescheiterten *builds* im *Elvis-Basis*-Teilprojekt

Abbildung 4. b zeigt das Verhältnis von erfolgreichen (*good*) *builds* zu gescheiterten (*broken*) *builds* des Teilprojekts *Elvis-Basis*. Von den 109 bisher in *Elvis-Basis* durchgeführten und dieser Statistik zugrunde liegenden *builds* sind 87 erfolgreich verlaufen und 22 gescheitert. Diese Zahlen alleine sprechen zwar schon für einen stabilen Entwicklungsprozess, bei dem bei der Integration wenig Fehler in das System gelangen. Es ist aber noch keine zeitliche Zuordnung der Fehlerfälle möglich. Diese lässt Aussagen darüber zu, ob an bestimmten Zeitpunkten, an denen gegebenenfalls eine Maßnahme im Projekt durchgeführt wurde, eine besondere Häufung von Fehlern zu verzeichnen ist. Die Verteilung in Abbildung 4. c hingegen zeigt eine längerfristig stabile Phase erfolgreicher *builds*. Die gescheiterten *build*-Läufe finden sich gehäuft nach der Einrichtung des Projekts. Der Grund dafür sind die anfänglichen Arbeiten am *build*-Prozess, die zum Beispiel Anpassungen an den Klassenpfaden und das Einbinden von Bibliotheken umfassen.

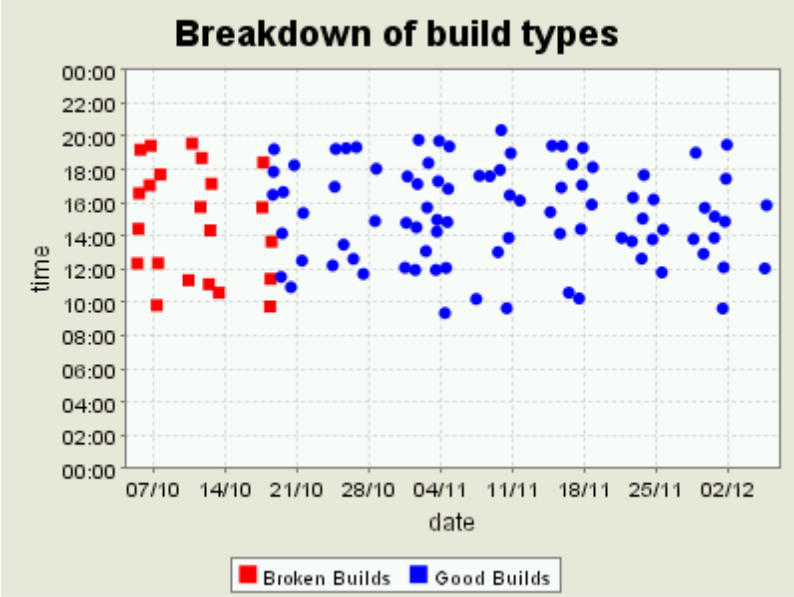

Abb. 4. c: Zeitliche Verteilung der *builds* im VS-Basis-Teilprojekt

Diese Kontinuität der *builds* konnte nicht für alle Teilprojekte erreicht werden. Im *Elvis-Kunden*-Teilprojekt finden sich mehrere Phasen, in denen Fehler – und somit scheiternde *builds* – sehr lange im System verweilen. Wie im vorhergehenden Kapitel beschrieben, ist dieser Umstand einem unnötig langen Aufschub von Korrekturen geschuldet. Das bereits beschädigte System wird dabei auch für weitere Fehler anfällig, die sich während einer „*roten Phase*“ in den Quellcode einschleichen können. Dadurch werden solche Phasen zusätzlich verlängert. In Abbildung 4. d sind diese langgezogenen Phasen vieler aufeinanderfolgender, gescheiterter *builds* sehr gut zu erkennen.

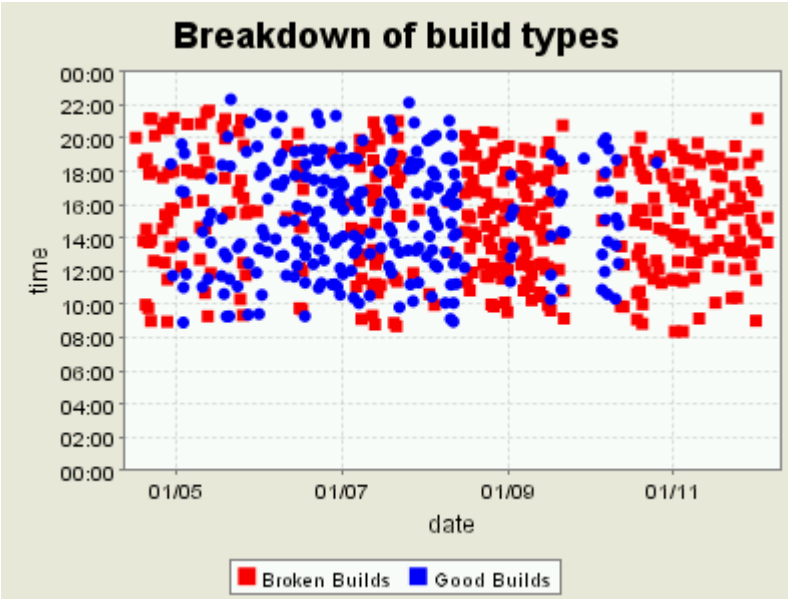

Abb. 4. d: Zeitliche Verteilung der *builds* im Elvis-Kunden-Teilprojekt

4.1.5. Welche Ziele konnten nicht umgesetzt werden

Im *Elvis*-Projekt wurde zu einem Zeitpunkt in etwa in der Mitte des bisherigen Projektzeitraums die Beobachtung der Architektur mit Hilfe von *xRadar* (siehe [xRadar]) eingeführt. Der Einsatz dieses Werkzeugs sollte dazu dienen, die Verletzungen der Architektur, die durch das Auseinanderschneiden des anfänglichen großen und einzelnen Anwendungsblocks entstanden sein mögen, einzugrenzen und zu beseitigen. Für die neuen Teilprojekte sollten mit der Zeit Regelsätze erstellt werden, die klar die durch die ursprünglich gewählte Architektur geforderten Bedingungen widerspiegeln. Anhand der durch *xRadar* ermittelten Metriken sollten die Entwickler gezielt fehlerhafte Beziehungen aus dem Projekt beseitigen.

Unerlaubte Beziehungen zwischen Subsystemen				
Datum	**Elvis-I**	**Elvis-K**	**Elvis-Versorgung**	**Elvis-Mobil**
01.09.2004	5	7	330	3
07.09.2004	5	0	322	3
14.09.2004	5	68	27	2
21.09.2004	2	62	29	0
30.09.2004	10	65	29	0
06.10.2004	10	65	29	0
20.10.2004	0	78	30	58
27.10.2004	0	68	28	58
03.11.2004	1	68	29	58
10.11.2004	1	66	29	58
17.11.2004	1	40	29	58
24.11.2004	1	32	29	58
01.12.2004	1	31	0	58
07.12.2004	1	35	0	58
15.12.2004	0	13	0	58
22.12.2004	0	13	0	58
05.01.2005	0	13	0	58
12.01.2005	0	13	0	58
19.01.2005	0	13	0	58
26.01.2005	0	13	0	58
02.02.2005	0	13	0	58
16.02.2005	0	19	0	58
22.02.2005	0	18	0	45
28.02.2005	0	19	0	45

Tabelle 4.b: Tracking über die Anzahl von Architekturverletzungen

Tabelle 4. a zeigt beispielhaft das *Tracking* über die Anzahl Architekturverletzungen einiger Teilprojekte. Auffällig sind dabei die teils extremen Sprünge in den Anzahlen zweier aufeinander folgender Messpunkte. Die plötzliche Zunahme der Verletzungen im Teilprojekt *Elvis-K* lässt sich mit der Verschärfung von Architekturregeln erklären, die zwischen der zweiten und der dritten Messung stattgefunden hat. Es wurden Subsysteme und Schichten identifiziert und durch die

unter *xRadar* benutzten Regeln gegeneinander abgegrenzt. Da die Regeln nicht von vornherein definiert waren, gab es einen Sprung in der Anzahl der Verletzungen zu dem Zeitpunkt, an dem die Regeln eingebunden wurden. Auch die großen Sprünge von *Elvis-Versorgung* und *Elvis-Mobil* lassen sich auf die Einführung beziehungsweise Verfeinerung von Architekturregeln für *xRadar* erklären. Durch das Zerteilen des Projekts sind außerdem teilweise noch Komponenten fälschlicherweise in Teilprojekten hängen geblieben, in die sie durch die Teilung eigentlich gar nicht mehr gehören sollten. Das nachträgliche Verschieben solcher Komponenten an die richtige Stelle sorgte ebenfalls für sprunghafte Änderungen der Anzahlen.

Ausgehend von einem Stand, in denen solche Umbauarbeiten abgeschlossen und die Architekturregeln definiert sind, sollten diejenigen Architekturverletzungen beseitigt werden, die sich dann noch im System befinden. Bei diesen handelte es sich um Fehlbenutzungen von einzelnen Projektteilen innerhalb eines Teilprojekts oder einer Komponente. Die Projekte *Elvis-K* und *Elvis-Mobil* erreichten dabei nicht den Zustand einer Architektur, die frei von solchen Verletzungen war. Vielmehr zeigt das *Tracking* vor allem im *Elvis-Mobil*-Projekt nach der anfänglichen Regeldefinition ein weitgehend statisches Bild ohne eine signifikante Verbesserung der Architektur.

4.1.6. Probleme in späteren Phasen des Projekts

Nachdem im Projekt über einen längeren Zeitraum von mehreren Wochen einige Probleme mit dem automatisierten *build*-Prozess und dem daraus resultierenden negativen *build*-Status unverändert bestehen blieben, wurde ein Treffen der Entwickler einberufen. Auf diesem Treffen sollten die Ursachen dieser Probleme identifiziert werden beziehungsweise die Gründe dafür, dass der *build*-Prozess noch nicht repariert worden war oder sich möglicherweise nicht reparieren ließ. Des Weiteren sollte dann darüber entschieden werden, ob der Einsatz von *CruiseControl* fortgeführt wird oder nach einer Alternative für die Qualitätssicherung gesucht werden soll.

Als Ursache für die fortwährend scheiternden builds wurden von Entwicklern der einzelnen Teilprojekte folgende Gründe genannt:

1. *„Wir haben nicht soviel Zeit für die Reparatur von Fehlern, die nur unter CruiseControl auftreten!“* - Der hohe Aufwand, der für die Behebung der auf dem *CruiseControl* auftretenden Fehler anfällt, ließ sich im Projektzeitplan nicht mehr unterbringen. Das entsprechende Teilprojekt hatte demnach nicht die benötigte Zeit, diese *CruiseControl*-seitigen Fehler zu beseitigen.
2. *„Wir führen immer die Tests lokal aus; dort laufen die durch. Das genügt eigentlich, da wir im HEAD*[1] *sowieso nur noch wenig ändern werden.“* - Bei regelmäßiger Ausführung der Tests an den Entwicklerarbeitsplätzen lohne eine Reparatur des automatisierten *build*-Prozesses möglicherweise nicht, vor allem, wenn nur noch wenige Änderungen an der aktuellen Version vorgenommen werden sollen. Deshalb habe das betroffene Teilprojekt bisher eine Reparatur aufgeschoben und halte einen weiteren Einsatz von *CruiseControl* für unnötig.

[1] Bezeichnung für die aktuelle Version in einem *CVS-Repository*.

3. *„Was können wir dafür, dass andere Teilprojekte, von denen wir abhängen, Fehler haben? Wir reparieren doch nicht deren Fehler!“* - Ein weiteres Teilprojekt wäre für den weiteren Einsatz von *CruiseControl*, sähe aber das Problem darin, dass zwar das eigene *build* scheitern würde, es aber an Fehlern in anderen Teilprojekten läge, von denen man abhinge. Für die Fehler in anderen Teilprojekten fühle man sich nicht zuständig.
4. *„Wir haben schon versucht, die Fehler zu korrigieren. Lokal laufen die Tests aber durch, und wir können nicht feststellen, warum das unter CruiseControl nicht läuft.“* - Zuletzt liefen dann einige der Tests unter *CruiseControl* nicht durch, während sie bei lokaler Ausführung keine Probleme machten. Der Fehler sei aber nicht festzustellen; irgendetwas laufe unter *CruiseControl* wohl einfach anders als in der Entwicklungsumgebung. Im Unterschied zur zweiten Aussage weiter oben wollte das Teilprojekt *CruiseControl* auch weiterhin nutzen, sah sich aber nicht in der Lage, das vermeintlich undurchschaubare Verhalten von *CruiseControl* in den Griff zu bekommen.

Ein Großteil der Entwickler sah demnach als Ursache für die auftretenden Fehler *CruiseControl* selbst. Vor allem die Tatsache, dass oftmals die Tests lokal durchliefen, dann aber unter *CruiseControl* scheiterten, wurde als Indiz verstanden, dass unter *CruiseControl* noch etwas geschieht, was sich der Kontrolle der Entwickler entzog. Dieses *BlackBox*-Verhalten wurde auch zum Anlass genommen, gegen Reparaturen des bisherigen automatisierten *build*-Prozesses zu argumentieren. Schließlich ließe sich schwer abschätzen, wie aufwändig es sein würde, die Testumgebung unter den vermeintlich speziellen Gegebenheiten von *CruiseControl* anzupassen.

Zurückzuführen ist dieses Missverständnis darauf, dass die genaue Funktionsweise von *CruiseControl* den Entwicklern nicht bekannt war. Die Tatsache, dass *CruiseControl* die automatisierten Aufgaben des *build*-Prozesses durch Aufrufe der dafür vorgesehenen *ANT*-Skripte ausführt, lässt den Schluss zu, dass der Prozess nicht allein *CruiseControl*-seitig scheitert, sondern durch die besondere Umgebung auf den Entwicklerrechnern begründet ist. So können serverseitige Tests aus verschiedenen Gründen scheitern, während sie lokal am Entwicklerplatz durchlaufen:

- Beim Einspielen von Änderungen werden zwar lokal die Tests ausgeführt, aber die Strukturen auf der *build*-Testdatenbank nicht angepasst. Dann scheitern die Tests erst bei der Ausführung unter *CruiseControl*.
- In besonderen Fehlersituationen, die auf Entwicklerrechnern möglicherweise nur selten auftreten, können serverseitig die Tests blockieren, vor allem, wenn ein Fehlerdialog den Test blockiert.
- Der im *Elvis*-Projekt eingesetzte Integrationsserver besitzt nicht dieselben Zugriffsrechte auf das Netzwerk wie die Entwicklerrechner. In einigen Fällen führte dies zu besonderen Fehlern bei der Ausführung des build-Prozesses.
- Die im *build*-Skript verwendeten Klassenpfade laufen mit der Zeit mit den in der IDE eingestellten Pfaden auseinander; während die Tests lokal laufen, können sie auf dem Integrationsserver gegebenenfalls nicht mehr auf alle benötigten Ressourcen zugreifen.

- Im Allgemeinen finden die Tests bei der Ausführung über die IDE und bei der Ausführung per *ANT*-Skript in zwei unterschiedlichen Umgebungen statt. Im *Elvis*-Projekt sind dies das eingebettete *JUnit* in *Eclipse* sowie der *JUnit-Task* unter *ANT*.

Trotz der Feststellung, dass das Scheitern des *builds* nicht auf *CruiseControl*, sondern auf Fehler im *build*-Prozess zurückzuführen sei, wollten sich trotzdem noch nicht alle Teilprojekte um die Reparatur kümmern. Das Beispiel des Projekts, welches kurz vor der Fertigstellung nicht mehr allzu vielen Änderungen unterliegen sollte, zeigt, dass es vielleicht nicht immer sinnvoll erscheint, einen hohen Aufwand für den automatisierten *build*-Prozess zu betreiben, wenn die Entwickler das Projekt trotzdem diszipliniert mit lokal ausgeführten Tests begleiten. Zudem wollte das Teilprojektteam sich noch zusätzlich auf manuelle Tests stützen, da man vermutete, durch das Ausprobieren und Benutzen der Software eher noch weitere Fehler entdecken zu können.

Im *Elvis*-Projekt wurde daher zu diesem Zeitpunkt ein Kompromiss beschlossen: *CruiseControl* wird weiterhin eingesetzt; somit sind Reparatur und Wartung des automatisierten *build*-Prozesses verpflichtend für Teilprojekte, von denen andere Teile des Projekts abhängen. Allein die Projekte an der Spitze, die nicht mehr von anderen Projektteilen benutzt werden, steht es frei, selbst zu entscheiden, ob sie die Qualitätssicherung weiter mit *CruiseControl* betreiben wollen oder der Projektleitung beziehungsweise dem *Tracking* eine Alternative anbieten wollen. Da sich beim Einsatz von *xRadar* und *JCoverage* noch zusätzliche Probleme ergeben können, wurde deren automatisierte Ausführung unter *CruiseControl* deaktiviert. Architektur und Testabdeckung sollten zwar weiterhin manuell ermittelbar sein, aber um den Umfang zu pflegenden Skripts zu verringern, sollten diese beiden *Trackings* aus dem *build*-Prozess entfernt werden. Die so genannten *nightly builds* wurden somit wieder aus dem Projekt genommen. Eine tägliche Erhebung von Testabdeckung und Architekturverletzungen wäre nicht mehr notwendig, da die Zahlen für das *Tracking* nur einmal wöchentlich ausgewertet werden mussten.

Zu der Ansage, dass nun jedes Team dafür verantwortlich sei, den Status unter *CruiseControl* „grün“ zu halten, kam der Zusatz, dass ein Team nun auch die Verantwortung dafür habe, bei Fehlern in anderen Teilprojekten, die dadurch den eigenen *build*-Prozess zum Scheitern bringen, bescheid zu sagen und die Behebung der Fehler in die Wege zu leiten. Zur Erleichterung der Fehleranalyse und der Behebung von Fehlern, die nur auf dem Integrationsserver auftreten, sollte eine projektinterne *Wiki*-Seite eingerichtet werden, mit deren Hilfe Hinweise und Tipps gesammelt werden sollten zu besonders häufigen und kryptischen Fehlern und deren Lösung. Schließlich sollte noch jede Tätigkeit, bei der am Integrationsserver selbst Eingriffe vorgenommen werden müssen, dokumentiert werden. Dadurch sollte später ermittelt werden, ob tatsächlich ein signifikanter Mehraufwand durch das Beheben von *CruiseControl*-seitigen Problemen entsteht. Mehrere Entwickler hatten während des Meetings geäußert, dass sie das Gefühl hätten, es würde sehr viel Aufwand in die Wartung von *CruiseControl* gesteckt für Probleme, die der Einsatz von *CruiseControl* selbst aufgeworfen hat.

In diesem Meeting stellte das Projekt für die Benutzung von *CruiseControl* fest, dass im *Elvis*-Projekt hauptsächlich teils unklare Zuteilung von Verantwortlichkeiten sowie teils lückenhaftes Verständnis für den *build*-Prozess und

die Funktionsweise von *CruiseControl* für den zunehmenden Zerfall des *build*-Prozesses verantwortlich gemacht werden können.

4.1.7. Bewertung des Einsatzes im Projekt

Die *build*-Skripte im Griff zu behalten, kann sehr aufwändig werden, da diese bei sehr großen und komplexen Projekten die Projektabhängigkeiten berücksichtigen müssen. Der Einsatz von *ANT* als *build*-Werkzeug im *Elvis*-Projekt hat die Komplexität zudem noch vergrößert, da der *build*-Ablauf detailliert beschrieben werden musste und somit auch sehr fehleranfällig vor allem bei Anpassungen war.

Die Bedeutung von fehlschlagenden *builds* war den Entwicklern nicht klar genug; dadurch wurden Fehler im Projekt teils sehr lange verschleppt. Bequemlichkeit kann dem Einsatz von *CruiseControl* gerade in diesem Punkt zusätzlich im Wege stehen. Häufig haben Fehler deshalb unnötig lange im System existiert, da der Aufwand, den *build*-Prozess zu korrigieren, die Entwickler erst abschreckte. Dazu kam noch, dass einige Entwickler *CruiseControl* als *BlackBox* verstanden.

Trotzdem hat der Einsatz von *CruiseControl* dem Projekt aus meiner Sicht sehr geholfen. Ohne *CruiseControl* hätte sich die Arbeit an einem Teilprojekt von vielen als sehr viel schwieriger, wenn nicht sogar zum unlösbaren Integrationsproblem entwickeln können. Die fortwährende Ungewissheit, in welchem Zustand sich das Gesamtprojekt befindet gepaart mit der Unsicherheit, ob eigene Änderungen das Projekt in einen inkonsistenten Zustand versetzen, hätte die Entwicklung irgendwann in eine Aufwandsfalle getrieben, in der sehr viel Zeit zum Integrieren und Durchtesten der Gesamtanwendung hätte aufgebracht werden müssen.

Erfolgreiche builds haben im Projekt die Moral gehoben. Andererseits wurde *CruiseControl* teils als notwendiges Übel empfunden, vor allem, wenn für ein scheiterndes *build* auf die Schnelle keine Lösung gefunden werden konnte. Dazu kommt, dass Schwierigkeiten bei der Fehlersuche oftmals auch dazu führten, die Ursache für gescheiterte *builds* bei *CruiseControl* zu suchen. Probleme im build-Prozess wurden dadurch verschleiert und erst verspätet korrigiert. Von den genannten Problemen mit dem *build*-Skript und der Fehlersuche bei angeblich *CruiseControl*-seitigen Problemen abgesehen, fügte sich *CruiseControl* aber dennoch sehr gut in den grauen Entwickleralltag und den Entwicklerarbeitsplatz ein. Bei der normalen Entwicklertätigkeit musste nicht zusätzlich auf Besonderheiten von *CruiseControl* geachtet werden. Beim Schreiben von Tests musste nur besonders auf Lauffähigkeit auf dem Integrationsserver geachtet werden, was aber in der Art der Anbindung des Integrationsservers in das Netz begründet lag. Änderungen und Neuerungen wurden einfach wie gehabt ins *Repository* eingespielt, das Feedback bekam der Entwickler automatisch. Schließlich holt *CruiseControl* sich die Informationen dann ab, wenn der *build*-Prozess sie benötigt.

Die Testausführung und der automatisierte *build*-Prozess konnten im *Elvis*-Projekt sehr gut durch *CruiseControl* unterstützt werden. Die automatisierte Erhebung zusätzlicher Metriken erwies sich allerdings im Kontext des *Elvis*-Projekts als weniger ergiebig beziehungsweise nützlich als erhofft. Zudem sollte die Wartung

und Erweiterung der *build*-Skripte vereinfacht werden, weshalb deren Umfang wieder verringert wurde. Die Ermittlung der Testabdeckung sowie die Ermittlung von Architekturverletzungen wanderten daher nur kurz in den automatischen Prozess. Diese Informationen wurden im späteren Abschnitt des Projekts wieder durch manuelle Ausführung der entsprechenden *build*-Targets gesammelt, die bei Bedarf für das *Tracking* angestoßen werden konnten.

4.2. Projekt B (WebMig)

An der Universität Hamburg wird vom Arbeitsbereich Softwaretechnik des Fachbereichs Informatik seit Mai 1999 das Community-Werkzeug *Commsy* [COMM] entwickelt. Die auf PHP basierende Webanwendung soll durch das *WebMig* genannte Projekt nach und nach in die Java-basierte Anwendung *JCommsy* portiert werden. Anfänglich wurde das Projekt als Hauptstudiumsprojekt des Arbeitsbereiches als Weiterentwicklung durch Studenten und wissenschaftliche Mitarbeiter der Universität eingeleitet und steht inzwischen als *OpenSource*-Anwendung der Weiterentwicklung im Internet zur Verfügung.

Der automatisierte *build*-Prozess soll dabei die Qualität sichern, indem jederzeit und kurzfristig beim Scheitern des *builds* beziehungsweise der Tests die Projektleiter und –betreuer benachrichtigt werden und damit zeitnah auf eine unerwünschte Ausbreitung von Inkonsistenzen reagiert werden kann.

4.2.1. Besonderheiten der nachträglichen Einführung

Im *WebMig*-Projekt wurde *CruiseControl* erst später im Projekt eingeführt. Dementsprechend existierte zum Einführungszeitpunkt noch kein Target im *build*-Skript, mit dem automatisiert der gesamte *build*-Prozess durchlaufen werden konnte. *Targets* für das Testen der Projektklassen und den Zusammenbau von Projektteilen, zum Beispiel der Webkomponente, waren aber schon vorhanden.

Eine Herausforderung der nachträglichen Einführung war nun, die bisher lokal laufende Testumgebung auf dem *CruiseControl*-Server zu reproduzieren. Im Projekt wurde für die Entwicklung die Entwicklungsumgebung *Eclipse* von *IBM* (siehe [Ecl]) eingesetzt. Diese führt Tests in der *JUnit*-Umgebung aus; Tests lassen sich bequem aus dem Java-Editor heraus starten. Welche Bibliotheken und sonstigen Ressourcen für einen Tests sichtbar sind und in den Klassenpfad geholt werden, ist in der jeweiligen Projektkonfiguration von *Eclipse* festgeschrieben. Das kann aber dazu führen, dass in einer anderen als der lokalen Testumgebung unter *Eclipse* Bibliotheken und Ressourcen nicht im Klassenpfad sichtbar sind. Das liegt daran, dass für das Testtarget im *build*-Skript der Klassenpfad per Hand eingetragen wird. So kann es passieren, dass mit der Zeit die Klassenpfade in der Projektkonfiguration und im *build*-Skript auseinander laufen, wenn bei Änderungen beide separat angepasst werden müssen.

Erwartungsgemäß scheiterten die Tests in den ersten Tagen nach Einführung von *CruiseControl* erst einmal. Zurückzuführen war dies zum einen auf einige noch

im Projekt vorhandene Fehler, aber hauptsächlich auf die geänderte Testumgebung auf dem Integrationsserver. Somit mussten Änderungen am *build*-Skript und an einigen Tests vorgenommen werden. Um für die Zeit der Umstellung aber nicht die Ausführung der lokalen Tests zu stören, wurde zusätzlich zum neuen `cruisecontrol`-Target im *build*-Skript ein zusätzliches `cruisecontrol-tests`-Target eingeführt. Dieses enthält alle nötigen Schritte, um die Testumgebung für den automatisierten *build*-Prozess auf dem Server vorzubereiten. Dazu gehören unter anderem das Aufsetzen von benötigten Testverzeichnissen oder das Kopieren von notwendigen Ressourcen in den Klassenpfad. Bei der lokalen Ausführung in der IDE sind oft schon alle Ressourcen per Projektkonfiguration eingebunden. Auf dem Server müssen die Ressourcen hingegen erst zusammenkopiert werden, damit sie in der Testumgebung des *ANT*-Skripts zur Verfügung stehen.

Eine weitere Herausforderung stellte die Prüfung der im Projekt entwickelten *JSP*-Seiten[1] dar. Dazu wurde ein Testtarget eingeführt, welches die *JSPs* während des *build*-Prozesses kompiliert, um deren Konsistenz zu prüfen. Ebenso wurde für die Ermittlung der Testabdeckung *Cobertura* (vgl. [Cob]) in den *build*-Prozess eingeführt.

4.2.2. Probleme bei der Einführung von CruiseControl

Anfangs war der automatisierte *build*-Prozess nicht zum Laufen zu bringen; nach dem Start von *CruiseControl* lief das *build* einmal korrekt durch, sobald aber danach Änderungen ins *Repository* eingespielt wurden, reagierte *CruiseControl* auf diese nicht mehr. Als Auslöser konnte ein Abbruch des CVS LOG-kommandos identifiziert werden, mit dessen Hilfe *CruiseControl* prüft, ob Änderungen ins *CVS-Repository* eingespielt worden sind.

Die Tests, die bis dahin nur lokal auf den Entwicklerrechnern ausgeführt worden waren, legten unter anderem Testverzeichnisse und Testdateien im Klassenpfad an. Das führte dazu, dass *CVS* beim Prüfen des Projekts auf dem Integrationsserver auf Dateien und Verzeichnisse stieß, die nicht im *Repository* vermerkt waren. *CVS* bricht dann den Vergleichslauf ab, um den Nutzer darauf hinzuweisen, dass Verzeichnisse und Inhalte im Weg seien, die nicht mit dem *Repository* abgeglichen werden könnten. Die Tests des Projekts mussten somit erst einmal dahingehend angepasst werden, dass ihre Testdaten nach Beendigung des jeweiligen Tests wieder gelöscht werden müssen. Alternativ könnte der *build*-Prozess nach Beendigung des *builds* immer alle erstellten Verzeichnisse und Dateien löschen. Wichtig dabei ist, dass die Dateien auch dann entfernt werden müssen, wenn der Testlauf oder das *build* ansonsten wegen eines Fehlers zwischendrin kontrolliert abbricht.

[1] *JavaServerPages*, Unterstützung von dynamischer Webseitenentwicklung unter Java.

4.3. Zusammenfassung

Im *Elvis*-Projekt wurde *CruiseControl* eingeführt, um *continuous integration* in einem immer größer und komplexer werdenden, verteilt entwickelnden Projekt zu unterstützen. Das Projekt entschied sich für einen automatisierten *build*-Prozess, da das Projekt auf mehrere Teilprojekte aufgeteilt wurde und der bis dahin manuelle Prozess der Integration immer langwieriger wurde. Dabei waren mehrere Teilprojekte zu berücksichtigen, die teilweise von einander abhingen und aufeinander aufbauten. Für einen automatisierten Prozess bedeutete dies, dass einem Teilprojekt immer alle aktuellen Versionen der benötigten anderen Teilprojekte zur Verfügung stehen mussten.

Zusätzlich zur Automatisierung des *builds* und der Testausführung wurden mit der Einführung von *CruiseControl* verschiedene *Metriken* erhoben. Sie sollten den Zustand des Projekts anhand einiger Kennzahlen darstellen. Mit Hilfe dieser Zahlen sollte weiterhin der Projektfortschritt sowie der Effekt von unterschiedlichen Maßnahmen zur Qualitätssicherung beobachtet werden. Nach der Einführung wurde der Projektzustand noch sehr häufig beobachtet, um die anfänglichen Korrekturen machen zu können, die nötig waren, den *build*-Prozess zum Laufen zu bringen. Im weiteren Verlauf des Projekts nahm die Häufigkeit, mit der der Projektzustand überprüft wurde, ab, so dass später nur noch ein- bis zweimal am Tag der build-Status geprüft werden musste.

Fehler im *build*-Prozess des *Elvis*-Projekts stellten sich als schwierig zu finden und zu beheben heraus. Anders als die Tests im Projekt führten Fehler im Prozess zu Fehlermeldungen, die nicht direkt auf die fehlerhafte Stelle führten. Dadurch wurde die Fehleranalyse erschwert und damit die Korrektur der Fehler hinausgezögert. Die Wartung und Erweiterung des *build*-Prozesses nahm daher oft mehr Zeit in Anspruch als das Korrigieren von Fehlern, die von den Tests des Systems gefunden wurden. Auch wurde in einem Teilprojekt für eine Weile ein besonderer Integrationsprozess eingeführt, der den Arbeitstag mit dem Schreiben eines Tests, der Erhöhung der Testabdeckung oder der Korrektur einer Architekturverletzung einleiten sollte. Die Besonderheiten im *Elvis*-Projekt waren der Einsatz von *JCoverage* als Testabdeckungswerkzeug und *xRadar* als Architekturbewertungssoftware. Außerdem wurden im *Elvis*-Projekt mehrere Teilprojekte im *build*-Prozess integriert und deren Abhängigkeiten abgebildet.

Die Auswertung der Testabdeckung wurde eingesetzt, um gezielt die Testabdeckung im Projekt zu erhalten beziehungsweise zu erhöhen. Ebenso wurde das Testen während der Aufteilung des Projekts auf mehrere Teilprojekte begleitet. Die angestrebte Bereinigung der Architektur wurde nur in Teilen umgesetzt. Als die Architekturbewertung aus dem automatisierten *build*-Prozess wieder entfernt wurde, gab es im System noch an vielen Stellen Verletzungen der Architektur.

Einige Missverständnisse über die Funktionsweise von *CruiseControl* haben einige Entwickler im Projekt dazu verleitet, Mehraufwand für die Behebung von Fehlern unter *CruiseControl* anzunehmen, welche ihrer Meinung nach nur für das fehlerfreie Funktionieren von *CruiseControl* anfallen müssten. Der Mehrgewinn beim Einsatz von *CruiseControl* wurde damit in Frage gestellt, ohne nach den wirklichen Ursachen – wie zum Beispiel Fehlern im *build*-Prozess – zu fahnden.

Auch im *WebMig*-Projekt wurde *CruiseControl* zu einem späten Zeitpunkt ins Projekt eingeführt und erforderte somit einige Anpassungsarbeiten am *build*-Prozess.

Als Besonderheiten werden im *WebMig*-Projekt im automatisierten Prozess *JSP*-Seiten kompiliert und somit auf Konsistenz geprüft. Außerdem werden die Projektbetreuer per E-Mail über aktuelle Projekt-*build*-Zustände und –änderungen informiert.

5. Fazit

5.1. Die Grenzen von CruiseControl – wozu dient es nicht?

In verteilten Softwareprojekten kann der Anspruch bestehen, dass nicht nur fehlerhafte *builds* angezeigt werden, sondern dass Änderungen, die zum Scheitern des *builds* führen, gar nicht in das *Repository* gelangen können. Besonders interessant ist das bei Projekten, bei denen sichergestellt werden muss, dass sich jederzeit von der aktuellen Version im *HEAD* eine lauffähige und getestete Produktversion erstellen lässt. Ebenso erlaubt diese Vorgehensweise bei sehr großen und verteilt entwickelnden Projekten, zum Beispiel *OpenSource*-Projekte, an denen unzählige Entwickler einer großen Entwicklergemeinschaft oder im Internet miteinander entwickeln, die aktuelle Version des Projekts gegen Änderungen oder Neuerungen abzuschirmen, die das Projekt in einen inkonsistenten Zustand versetzen könnten. Zudem bleibt die Verantwortung mindestens solange beim Entwickler, bis dieser eine fehlerfreie Version seiner Änderungen einspielen kann, die nicht vom Integrationsserver abgewiesen wird. *CruiseControl* kann nicht für solche Zwecke eingesetzt werden, da es auf schon geänderten Quellen eines *Repository* arbeitet und Fehler im *build* erst anzeigt, wenn diese schon in die aktuelle Version des Systems integriert wurden. Ein Nachteil eines solchen Systems besteht allerdings darin, dass die Entwickler nach dem Einspielen der Änderungen erst Erfolg oder Scheitern des *check ins* abwarten müssten, bevor sie weiterarbeiten könnten. Wenn nach dem Einspielen weiter Änderungen vorgenommen werden, kann das möglicherweise die nachträgliche Fehleranalyse und –korrektur erschweren, da sich das lokale System nicht mehr auf dem Stand befindet, auf dem es sich zum Zeitpunkt der Integration befand und gegebenenfalls weitere Änderungen, die zwischenzeitlich gemacht wurden, berücksichtig werden müssen.

Auch wenn *CruiseControl* vielfältige Möglichkeiten bietet, die Qualität des entwickelten Systems kontinuierlich zu überprüfen, so befreit es das Team keineswegs vom Prozess der regelmäßigen Qualitätsbeurteilung. Die Beobachtung der Testabdeckung sowie der Architekturgüte ist allein nicht ausreichend und ersetzt zum Beispiel nicht *Reviews* des Projekts. Ebenso bedeutet eine hohe Testabdeckung nicht automatisch, dass der abgedeckte Code optimal getestet wird noch dass die Tests die Funktionalität sinnvoll und nah an der fachlichen Wirklichkeit testen. Hier müssen die Entwickler den von ihnen geschriebenen Code fortwährend kritisch bewerten und weiterhin darauf achten, schon vorhandene Teststrukturen kontinuierlich beziehungsweise bei Bedarf zu verbessern. Eine Einhaltung der vorhandenen Architekturregeln allein genügt gegebenenfalls nicht, wenn sich ändernde Anforderungen eine Änderung der schon vorhandenen Regeln nötig macht. *CruiseControl* befreit das entwickelnde Team nicht von der Aufgabe, den im automatisierten Prozess kontinuierlich erhobenen Kennzahlen eine kritische Bewertung folgen zu lassen, selbst wenn die Zahlen fortwährend eine hohe Qualität widerspiegeln.

5.2. Alternativen zu CruiseControl

Es seien an dieser Stelle noch vier Alternativen zu *CruiseControl* genannt; es handelt sich dabei um Werkzeuge, die zur Ausführung automatisierter *build*-Prozesse dienen. Daher können sie auch zur Unterstützung von *continuous integration* eingesetzt werden, da mit ihrer Hilfe eine kontinuierliche Prüfung der Konsistenz der Projektquellen möglich ist. Als Alternativen seien hier kurz die Werkzeuge Anthill, *DamageControl*, *Tinderbox* und *Continuum* erwähnt.

Anthill (siehe [Anthill]) ist ähnlich wie *CruiseControl* ein Werkzeug zur Unterstützung automatisierten *build*-Prozessen. Im Gegensatz zu *CruiseControl* erhält man Anthill auch in einer kommerziellen, kostenpflichtigen Variante, die weit mehr Funktionalität wie die kostenlose Grundvariante bietet.

Tinderbox (siehe [TBox]) stellt auf besondere Art und Weise „*Missetäter*“ im *build*-Protokoll dar; nach dem Einspielen von Änderungen erscheint der Entwickler, der diese eingespielt hat, in der Integrationsansicht in einer Spalte namens „guilty“ (=“schuldig“). Sind die *builds* danach „grün“, so hat derjenige funktionierende Änderungen eingespielt. Schlägt der *build* allerdings fehl, so kann man sofort erkennen, wessen *check in* für das Scheitern verantwortlich zu machen ist. Allerdings geschieht dies gegebenenfalls wohl auch bei scheiternden Tests, die durch Nebenläufigkeit und nicht berücksichtigten Seiteneffekten nur ab und zu mal scheitern.

DamageControl (siehe [DC]) ist ein weiteres *build*-Werkzeug, welches im Vergleich zu *CruiseControl* nur eine stark eingeschränkte Auswahl an Versioning-Systemen unterstützt. Dafür bietet es über das Webinterface unter anderem auch mehr Administrationsmöglichkeiten. Einige wichtige Funktionen zum Erstellen von neuen und Übernehmen von vorhandenen Projekteinstellungen stehen dort zur Verfügung.

Continuum (siehe [Cont]) ist ein zum jetzigen Zeitpunkt noch relativ neues *continuous integration*-Werkzeug des *Apache Maven Projects*. Laut eigener Beschreibung soll es einen ähnlichen Funktionsumfang bieten wie *CruiseControl* und *Anthill*. Als Besonderheit zu diesem Werkzeug sei seine *SOAP*[1] *API*[2] genannt, mit dessen Hilfe Anwendungen Projekte unter *Continuum* verwalten und steuern können. *Continuum* liefert selbst einen solchen Anwendungsclient mit.

5.3. Ausblick

Neben der automatisierten Ausführung der Tests im Projekt sowie der Erhebung von Metriken zur Bewertung des Projektzustands und der Projektqualität kann *CruiseControl* auch für weitere automatisierbare Prozesse genutzt werden, so zum Beispiel zur Erstellung von Projektdokumentation oder zur automatischen Visualisierung von Projektfortschritt und Systemänderungen. Im vorgestellten Elvis-Projekt wird die Dokumentation über *DocBook* (siehe [DocBook]) generiert, allerdings geschieht dies nur zu vereinzelten Release-Terminen. Über den automatisierten

[1] *SOAP* ist eine Weiterentwicklung des ursprünglich von Dave Winer und Microsoft entwickelten *XML-RPC* (XML-*Remote Procedure Call*) für entfernte Funktionsaufrufe via XML-Protokoll.
[2] API - *Application Programming Interface*, Anwendungsprogrammierschnittstelle.

Prozess wäre aber eine regelmäßige Aktualisierung der Dokumentation parallel zum Projektfortschritt möglich.

Wie die automatisierte Dokumentation wäre eine Unterstützung des Abnahmeprozesses denkbar. So könnten Anforderungen und umgesetzte Funktionalität entweder im Quellcode oder in zusätzlichen Dokumenten im Projekt notiert werden. Für die *Releases* werden dann Abzüge dieser Dokumentation für das Abnahmeprotokoll genutzt, um einen Abgleich zu ermöglichen zwischen den in einer Iteration geforderten Aufwandspunkten und der erfüllten Funktionalität.

Ein weiterer Aspekt, der nicht in dieser Arbeit betrachtet wurde, ist die automatisierte Messung der Performanz (engl. *performance*) verschiedener Systemteile. Mit Hilfe von *CruiseControl* könnte so durch die *continuous integration* fortlaufend das Systemverhalten unter Last beobachtet werden, so dass erkennbar wird, wie sich Änderungen am System beziehungsweise an Teilen davon auf die Performanz auswirken. Durch das stetige Durchmessen des Systems mittels geeigneter Lasttests bekommt die Qualitätssicherung auch eine Möglichkeit geboten, Verbesserungen und Verschlechterungen der Performanz des Systems relativ zu einer Systemversion zu bestimmen. Die Herausforderung an dieser Stelle ist vor allem das Finden und Schreiben geeigneter Tests, mit deren Hilfe man einen realistischen Eindruck über die Systembelastbarkeit gewinnen kann.

Der Einsatz von *CruiseControl* zusammen mit *Maven* als *build*-Werkzeug wurde in dieser Arbeit nur am Rande beschrieben. Der abstrakte Ansatz der Projektbeschreibung bei *Maven* liefert einige Vorteile beim Erstellen und Warten des *build*-Prozesses unter *Maven*. Inwieweit sich dadurch die in dieser Arbeit beschriebenen Probleme, die sich in den Projekten durch den Einsatz von *ANT* ergeben können, gegebenenfalls vermeiden lassen, wurde in dieser Arbeit nicht behandelt.

5.4. Schlusswort/Fazit

Die continuous integration hat sich als wichtiger Bestandteil und nicht umsonst als eine der best practices in mit XP entwickelten Projekten etabliert. Die Unterstützung qualitätssichernder Maßnahmen in einem verteilt entwickelnden Softwareprojekt wird durch die kontinuierliche Integration und gleichzeitigen Erhebung von projektbezogenen Metriken gefördert. CruiseControl bietet durch die Einbindung vieler gängiger Versionsmanagementsysteme und build-Werkzeuge für eine große Zahl von heutigen Projekten eine Möglichkeit, den automatisierten build-Prozess im Projekt einzuführen. Diese breite Unterstützung gängiger Werkzeuge nimmt den Entwicklern aber weder die Verantwortung für die Qualität des erstellten Systems noch wird die Arbeit am Prozess selber durch den Einsatz erleichtert. CruiseControl bewahrt das Entwicklerteam nicht davor, bei komplexer werdenden Projekten mit den Problemen kämpfen zu müssen, die sich durch eine komplizierte Architektur im Softwareprojekt ergibt. Allerdings ist dem Projekt selbst geholfen, welches trotz einer hohen Komplexität weiterhin Qualitätsansprüchen genügen möchte, ohne einen sehr großen Anteil der Entwicklungszeit mit Tests oder Deployment und regelmäßigen Releases verbringen zu müssen.

Für mich stellt sich die Verwendung von CruiseControl als zu bewältigende Aufgabe dar. CruiseControl bietet dafür auch einen leichten Einstieg, so dass auch ohne großen Konfigurationsaufwand Projekte mit Hilfe von CruiseControl einen

automatisierten build-Prozess erhalten können. Allerdings gestaltet sich meiner Meinung nach die Beherrschung des build-Prozesses selbst als beliebig schwierig; eine Tatsache, die schon wiederholt in Projekten, in denen ich tätig war, dazu führten, dass CruiseControl als Aufwandsfresser gebrandmarkt wurde. Schließlich ist CruiseControl das Werkzeug, welches das Scheitern des build-Prozesses visualisiert. Und der Status eines gescheiterten Prozesses bleibt solange bestehen, wie der build-Prozess oder aber das Projekt selber nicht repariert wurden. Entwickler neigen dann vereinzelt dazu, CruiseControl nicht als Unterstützung zu sehen, sondern eher als notwendiges Übel.

6. Literaturverzeichnis

[AB] *AlienBrain*, Avid Technologies, Inc. - http://www.alienbrain.com, Stand: 17.09.2005

[ANT] *Apache Ant*, http://ant.apache.org, Stand: 05.08.2005

[Anthill] *Anthill*, Urbancode Software Development - http://www.urbancode.com/projects/anthill/default.jsp, Stand: 26.11.2005

[Bec05] BECK, Kent: *Extreme Programming Explained: Embrace Change*, *2nd Edition*, Addison-Wesley, 3. Auflage, 2005.

[CC] *CruiseControl*, http://cruisecontrol.sourceforge.net, Stand: 03.09.2005

[Cla04] CLARK, Mike: *Pragmatic Project Automation – How to Build, Deploy and Monitor Java Applications.* Pragmatic Programmers, Juli 2004.

[ClCa] *IBM Rational ClearCase*, IBM Software – http://www-306.ibm.com/software/awdtools/clearcase/, Stand: 17.09.2005

[Cob] *Cobertura*, http://cobertura.sourceforge.net, Stand: 04.11.2005

[COMM] *CommSy*, http://www.commsy.de, Stand: 23.10.2005

[Cont] *Continuum*, http://maven.apache.org/continuum/, Stand: 11.12.2005

[CVS] *CVS Concurrent Versions System* - https://www.cvshome.org, Stand: 05.08.2005

[DC] *DamageControl*, http://damagecontrol.codehaus.org/, Stand: 26.11.2005

[DocBook] *DocBook*, http://docbook.sourceforge.net/, Stand: 27.11.2005

[Ecl] *Eclipse*, IBM Software – http://www.eclipse.org, Stand: 04.11.2005

[Edl02] EDLICH, Stefan: *Ant – kurz & gut*. O'Reilly, 1.Auflage, 2002.

[FF] FOWLER, Martin; FOEMMEL, Matthew: *Continuous integration* (http://www.martinfowler.com/articles/continuousIntegration.html), Stand: 12.06.2005.

[GB98] GAMMA, Erich; BECK, Kent: *Test Infected: Programmers Love Writing Tests*. In: *Java Report 3* (1998), Nr. 7, S. 37-50.

[J2EE] *Designing enterprise applications with the J2EE platform* - http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/technologies/technologies.html, Stand: 04.11.2005

[JCov] *JCoverage* - http://www.jcoverage.com, Stand: 26.10.2005

[JWAM] *JWAM Framework* - http://www.jwam.de, Stand: 16.10.2005

[JWS] *JavaWebStart*, Sun Microsystems, Inc. – http://java.sun.com/products/javawebstart/developers.html, Stand: 04.11.2005

[Kos] KOSKELA, Lasse: *Driving on CruiseControl – Part 1* (http://www.javaranch.com/journal/200409/DrivingOnCruiseControl_Part1.html), Stand: 12.09.2005.

[Link02] LINK, Johannes: *Unit Tests mit Java – Der Test-First-Ansatz.* dpunkt.verlag, 1.Auflage, 2002.

[LRW02] LIPPERT, Martin; ROOCK, Stefan; WOLF, Henning: *Software entwickeln mit eXtreme Programming*. dpunkt.verlag, 1.Auflage, 2002.

[LRW05] LIPPERT, Martin; ROOCK, Stefan; WOLF, Henning: *eXtreme Programming – Eine Einführung mit Empfehlungen und Erfahrungen aus der Praxis*. dpunkt.verlag, 2.Auflage, 2005.

[McD94] MCDERMID, J.A.: *Software engineer's reference book.* Butterworth-Heinemann, 1994.

[MKS] *MKS Integrity Suite*, MKS - http://www.mks.de/solutions/index.jsp, Stand: 17.09.2005

[MV] *Maven, Apache Maven Project*, Apache Software Foundation - http://maven.apache.org/, Stand: 18.09.2005

[NAnt] *NAnt, "Not Ant"* - http://nant.sourceforge.net/, Stand: 18.09.2005

[Fish04] FISH, Shlomi: *The New Breed of Version Control Systems.* O'Reilly ONLamp, 2004 - http://www.onlamp.com/pub/a/onlamp/2004/01/29/scm_overview.html, Stand: 05.11.2005

[PFC] *Perforce*, Perforce Software Inc. - http://www.perforce.com/perforce/products.html, Stand: 17.09.2005

[PR+02] POMBERGER, Gustav; RECHENBERG, Peter et al.: *Informatik-Handbuch.* Carl Hanser Verlag, 3. Auflage, 2002.

[SnCM] *SnapshotCM*, True Blue Software - http://www.truebluesoftware.com/, Stand: 17.09.2005

[ST] *StarTeam*, Borland - http://www.borland.com/us/products/starteam/, Stand: 17.09.2005

[SVCS] *Subversion Version Control System* - http://subversion.tigris.org, Stand: 17.09.2005

[TBox] *Tinderbox*, http://www.mozilla.org/tinderbox.html, Stand: 11.12.2005

[Thwk] *ThoughtWorks CruiseControl Home* – Drittanbieterwerkzeuge, http://confluence.public.thoughtworks.org/display/CC/Home#Home-thirdparty, Stand: 04.11.2005

[VSS] *Visual SourceSafe*, Microsoft - http://msdn.microsoft.com/vstudio/previous/ssafe/, Stand: 17.09.2005

[xRadar] *xRadar*, http://xradar.sourceforge.net, Stand: 03.09.2005

[XSLT] *XSL Transformations (XSLT)* - http://www.w3.org/TR/xslt, Stand: 05.08.2005