Seminar: “Design Patterns”

“Streamed Lines” – Eine Pattern-Sprache für die Versionsverzweigung

17. Mai 1999

Frederic Schelling

Inhalt

1. Einleitung

2. Der Kontext
   - 2.1 Wozu Versionskontrollsysteme?
   - 2.2 Ein einfaches paralleles Szenario
   - 2.3 Seriell oder parallel arbeiten?
   - 2.4 Entscheidungsfaktoren
   - 2.5 Begriffe und Definitionen

3. Die Pattern-Sprache
   - 3.1 Einzelne Patterns im Detail
   - 3.2 Weitere Patterns

4. Die Anwendung
   - 4.1 Verzweigungsstrategien
   - 4.2 Entscheidungsfaktoren
   - 4.3 Auswirkungen

1. Einleitung

Thema des Vortrages ist die Beschreibung einer Pattern Language (d.h. einer bzgl. Inhalt, Benutzung und Interaktion miteinander verwobenen und aufeinander abgestimmten Sammlung von Patterns), die sich für den Einsatz im Rahmen des "Software Configuration Management", kurz SCM, eignet und zwar speziell im Bereich der parallelen Softwareentwicklung. Unter paralleler Softwareentwicklung verstanden sei hier die Arbeit eines **Teams** an einem Softwareprojekt, d.h. zeitgleich verlaufende Entwicklungsarbeit auf derselben Codebasis.

Unter dem Oberbegriff SCM versteht man die Menge aller Aufgaben und damit verbundenen Prozesse, die bei der Organisation der Systemstrukturen der zu erstellenden Software anfallen, bei der Überprüfung ihrer Funktionalität, bei ihrer Weiterentwicklung (Evolution) und bei der Koordination der Teamarbeit.

Um die stetig steigenden Ansprüche an die Qualität der Software gewährleisten zu können, wird es aufgrund der zunehmenden Komplexität moderner Software immer wichtiger, die Möglichkeiten der computergestützten Projektplanung auszuschöpfen. Diese kann speziell im Bereich der Softwareproduktion weit über die herkömmlichen Methoden hinausreichen und für das SCM spezialisierte Werkzeuge umfassen.

Ein wichtiges und zentrales Werkzeug für Softwareentwicklungsteams ist mit Sicherheit das Versionskontrollsystem. Diese Systeme kontrollieren, unter Vorgabe von Regeln, den Zugriff der Entwickler auf den aktuellen Stand und die Historie des gesamten Quellcodes eines Projekts ebenso wie zeitlich aufeinanderfolgende oder parallel verlaufende Abschnitte von Quellcode-Modifikationen.

Zeitlich parallel verlaufende Entwicklungsarbeit wird von diesen Systemen durch analog dazu verlaufende parallele "Versionsstränge" verwaltet. Da die gesamte, auch parallel verlaufende Arbeit am Projekt letztendlich eine einzige Idee als Ursprung und i.a. ein einziges Produkt zum Ziel hat, müssen diese parallelen Versionsstränge einerseits irgendwo entstehen und andererseits irgendwo auch wieder (zumindest zum Teil) zusammengeführt werden.

Das Anwendungsgebiet der hier vorgestellten Pattern Language "Streamed Lines" sind nun die typischen Probleme, die aus dem Aufteilen und Zusammenführen dieser Versionsstränge entstehen. Außerdem soll sie Hilfestellung beim Entscheidungsprozeß für eine sinnvolle Parallelisierungsstrategie leisten.

2. Der Kontext

2.1 Wozu Versionskontrollsysteme?

Wenn ein Projekt von einem Team durchgeführt wird, dann ist die Kooperation zwischen den Teammitgliedern immer ein zentraler Punkt in der Projektverwaltung. Kooperation bedeutet, daß die Berührungspunkte, die trotz Arbeitsaufteilung naturgemäß entstehen, von den Teammitgliedern möglichst reibungslos abgehandelt werden und den Projektfortgang so wenig wie möglich beeinträchtigen.

Ein Versionskontrollsystem gibt den Entwicklern zusätzlich zu der reinen Verwaltung der verschiedenen Versionen des Quellcodes die Möglichkeit, diese Situationen, die oft aus rein technischen Gründen entstehen (Quellcode-Konflikte), mit Hilfe einer technischen Lösung (eben dem Versionskontrollsystem) zu verwalten und damit die Möglichkeiten von menschlichen Fehlern in dem Verwaltungprozess zu reduzieren.

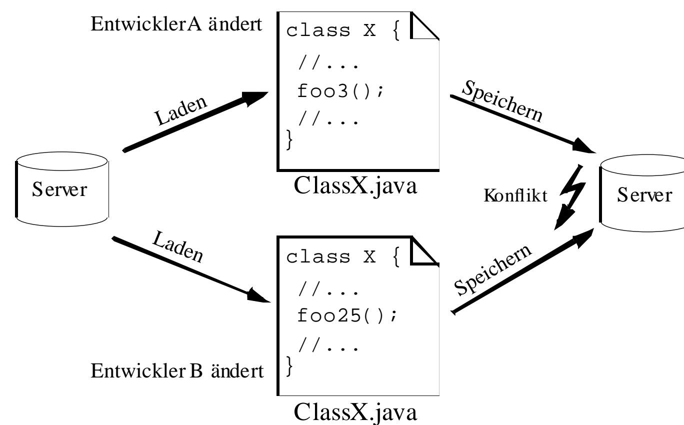
Da das Versionskontrollsystem nicht nur den aktuellen Stand des Quellcodes verwaltet, sondern auch sämtliche Änderungen, die jemals an dem Quellcode vorgenommen wurden, nimmt man es auch gerne in Anspruch, um die Ursachen von Programmfehlern und Abhängigkeiten zwischen Änderungen mit minimalem Aufwand nachvollziehen zu können.

Weiterhin stellt es ein wichtiges Dokumentationswerkzeug dar, da es den einzelnen Entwickler anhält, nachdem er neuen oder geänderten Code erstellt hat, diesen über den sogenannten Check-In in einer festgelegten Form einzubringen. Diese Form verlangt vom Entwickler ein Mindestmaß an sinnvoller Dokumentation, die dann ein wesentlicher Bestandteil des Check-Ins und damit der Historie des gesamten Entwicklungsprozesses wird.

Die in einem Softwareentwicklungsteam für die Produktivitätssteigerung gewünschte parallele Arbeitsweise setzt voraus, daß man ein Konzept für die Arbeitsaufteilung erarbeitet, um den zusätzlichen Koordinationsaufwand für die nebenläufigen Tätigkeiten möglichst gering zu halten. Versionskontrollsysteme unterstützen einerseits die Durchführung dieses Konzepts, da sie eine enge Kopplung der Organisation mit der relevanten Konzept-Dokumentation erlauben, andererseits geben sie jedoch keine Hinweise darauf, wie dieses Konzept aussehen muß. An dieser Stelle kommt die Pattern-Language “Streamed-Lines” ins Spiel.

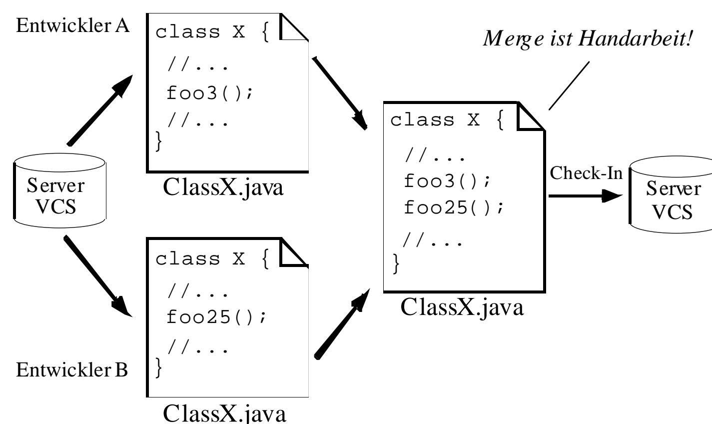
Aber zunächst wird im folgenden Abschnitt anhand eines einfachen Beispiels die beschriebene Grundproblematik erläutert:

2.2 Ein einfaches paralleles Szenario



In diesem Beispiel soll verdeutlicht werden, was passiert, wenn zwei Entwickler ohne (gegenseitige oder sonstwie geartete) Kontrolle Änderungen am Quellcode vornehmen. Die Arbeit sei gleichzeitig zu verrichten, um die Implementierung möglichst schnell abzuschließen.

Jeder der beiden Entwickler soll eine Methode der Klasse X implementieren, Entwickler A ist für **`foo3()`**, Entwickler B für **`foo25()`** zuständig. Der Quellcode ist (immerhin) auf einem zentralen Server abgelegt. Jeder der beiden Entwickler erstellt für sich selbst eine Kopie der Datei "ClassX.java" auf seinem Arbeitsplatz und modifiziert die Datei entsprechend seiner Vorgaben. Wenn nun der Entwickler A fertig ist und seine Version auf den Server zurückschreibt, dann wird diese Version überschrieben (gelöscht) werden, sobald der Entwickler B seine Version auf den Server schreibt. Es entsteht also ein Konflikt, der bereinigt werden muß.



Dies wird mit dem Vorgang des "Merging" (Zusammenmischen) bewerkstelligt. Einer der beiden Entwickler oder auch eine andere Person kümmert sich darum, die beiden parallel getätigten Änderungen in einer Datei zu vereinen. Dieser Vorgang kann auf der syntaktischen Ebene noch durch eine gewisse Automatisierbarkeit unterstützt werden, allerdings ist spätestens bei semantischen Konflikten vergleichsweise aufwendige Handarbeit angesagt.

Dieser **Merging-Aufwand** ist ein entscheidender Faktor bei der Festlegung des Arbeitsaufteilungs-Konzepts.

2.3 Seriell oder parallel arbeiten?

Dem beschriebenen Konflikt könnte man natürlich auch durch eine Serialisierung aus dem Wege gehen, d.h. der Entwickler A müßte seine Änderung vollständig abgeschlossen (d.h. auf den Server zurückgeschrieben) haben, bevor Entwickler B sich seine Version der Datei kopiert. Dies hätte allerdings eine Wartezeit zur Folge, d.h. der Entwickler B kann nicht sofort nach Auftragserteilung mit der Arbeit beginnen, sondern muß auf den Entwickler A warten. Es ist natürlich klar, daß dies nicht der gewünschte Effekt einer Arbeitsaufteilung sein kann.

Man kann nun das hier sehr einfach dargestellte Problem auf eine komplexere Situation übertragen: zwei voneinander abhängige Teilbereiche eines Projektes sollen gleichzeitig und möglichst separat erstellt werden. Änderungen in einem Teilbereich können Änderungen in dem anderen Teilbereich oder schlimmstenfalls zunächst unbemerkte Fehlfunktionen zur Folge haben. Wie geht man vor? Soll man trotzdem zugunsten **möglicherweise** kürzerer Entwicklungszeit (Merging-Aufwand!) parallel entwickeln oder soll man zugunsten mehr Sicherheit die Teilbereiche nacheinander fertigstellen? Mehr Sicherheit erhielte man dadurch, daß die Funktionalität des einen Bereichs größtenteils fertig- und sichergestellt ist und während der Entwicklungsdauer des zweiten Bereichs höchstens marginal verändert wird, d.h. man kann Fehlfunktionen klar zuordnen.

2.4 Entscheidungsfaktoren

Neben den schon genannten Faktoren wie Sicherheit und Produktivität gibt es noch eine Reihe anderer Einflüsse, die bei den Entscheidungen im Arbeitsaufteilungskonzept berücksichtigt werden sollten. Im folgenden sind die relevanten Einflüsse aufgelistet:

- *Sicherheit*
  - Konsistenz
    Alle Änderungen müssen vollständig vorhanden und untereinander konfliktfrei sein.
  - Zuverlässigkeit
    Entwickler müssen sich auf die Korrektheit einer bestimmten Menge an Code verlassen können. Im Allgemeinen ist natürlich die Korrektheit des gesamten Codes wünschenswert, allerdings ist dieser Zustand nur ein Endziel, deshalb kann die Korrektheit nur schrittweise erreicht werden.
  - Langfristige Stabilität
    Ein Code-Bereich muß auch in Kombination mit anderem Code richtig funktionieren.
  - Persistenz von eingebrachten Änderungen
    Eingebrachte Änderungen müssen auch nach Einbringen anderer parallel dazu entwickelter Änderungen noch vorhanden sein.
  - Keine “wiederauftauchenden” Fehler
    Einmal beseitigte Fehler dürfen nach Einbringen von parallel entwickeltem Code nicht wieder auftauchen.

- *Lebendigkeit*
  - Erhöhung der Effizienz/Produktivität
    Höhere Parallelität kann die Entwicklungsdauer verkürzen.
  - Erhöhung des Koordinations-Aufwands
    Beim Zusammenführen von Versionssträngen läßt zunehmende Parallelität den Aufwand steigen, der unternommen werden muß, um der Entstehung von Bugs entgegenzuwirken. Zuviel und schlecht geplante Parallelität kann die Vorteile der parallelen Softwareentwicklung reduzieren oder sogar völlig zunichte machen.
  - Gegenseitige Behinderung
    Zuviel parallele Arbeit an stark voneinander abhängigen Bereichen kann zu gegenseitiger Behinderung führen (file locking durch Checkouts) und die Ausschöpfung der Möglichkeiten paralleler Entwicklung mindern bis hin zur vollständigen Verklemmung

- *Wiederverwendbarkeit*
  - Nachvollziehbare Änderungen
    Die Nachvollziehbarkeit einer Änderung ist Voraussetzung für ihre Wiederverwendbarkeit.
  - Schnelle Übersicht
    Der Inhalt und zentrale Gehalt einer Änderung muß schnell und eindeutig auffindbar und identifizierbar sein, sonst ist die Wiederverwendbarkeit eingeschränkt.
  - Unabhängigkeit von Änderungen
    Änderungen sollten soweit wie möglich voneinander unabhängig sein von anderem, parallel dazu entwickeltem Code, damit man sie in andere Code-Bereiche übernehmen kann, ohne noch andere, dort nicht erwünschte Änderungen mit übernehmen zu müssen.

- *Teamarbeit*
  - Zuständigkeit und Verantwortung
    Die Verantwortung für Meilensteine, Komponenten, Änderungen und Features der Software sollten jeweils eindeutig bestimmten, dafür geeigneten Personen zugewiesen werden.
  - Koordination
    Die Kooperation zwischen den einzelnen parallel arbeitenden Teammitgliedern muß gut koordiniert und ausbalanciert werden; zuviel Interaktion kann genauso wie zuwenig die Produktivität und die Qualität der Arbeit beeinträchtigen.
  - Komplexität
    Komplexe Änderungen und Aufgaben müssen sinnvoll in überschaubare Teilbereiche aufgeteilt werden, deren Zuständigkeit eindeutig feststeht. Abhängigkeiten zwischen den Teilbereichen müssen auf ein Minimum reduziert werden.

2.5 Begriffe und Definitionen

Im folgenden werden einige zentrale Begriffe aus dem Bereich der Versionsverwaltung angeführt und erklärt, da die später im einzelnen beschriebenen Patterns wiederholt Gebrauch von diesen Begriffen machen. Durch Beibehalten des englischen Ausdrucks soll unterstrichen werden, daß es sich hierbei um eindeutig definierte und klar abgegrenzte Begriffe handelt.

- *Codeline*

  Eine zeitlich aufeinanderfolgende und inhaltlich aufeinander aufbauende Kette von Änderungen am Code. Typischerweise wären auf ein und derselben *Codeline* z.B. neuer Code für ein neues Feature der Software zu finden, inklusive aller für dieses Feature notwendigen Änderungen an bereits bestehendem Code. Ob und wann sich dann diese *Codeline* evtl. in weitere *Codelines* verzweigt oder verzweigen sollte, ist gerade eine der zentralen Fragen, zu deren Beantwortung die Pattern Language Hilfestellung leisten soll.

- *Branching*

  Branching bezeichnet das Abzweigen, also das Erzeugen einer neuen *Codeline* von einer bestehenden. Als Beispiel käme ein neues umfangreiches Feature in Frage, welches mehrere Schritte bis zu seiner Fertigstellung benötigt. Auf der Ursprungs-*Codeline* kann weitergearbeitet werden, ohne daß die abgezweigte *Codeline* darauf Rücksicht nehmen muß, ebenso kann auf der abgezweigten *Codeline* “in aller Ruhe” das gewünschte Feature implementiert und ausgetestet werden, bevor es dann durch *Merging* in die Ursprungs-*Codeline* wieder eingebracht wird. Daß sich in der Zwischenzeit die Ursprungs-*Codeline* weiterentwickelt hat, ist letztendlich die Ursache dafür, daß das *Merging* normalerweise einen nicht vernachlässigbaren Arbeitsaufwand bereitet.

- *Merging*

  Code, der zu einem früheren Zeitpunkt durch *branching* in eine separate *Codeline* abgekoppelt und getrennt weiterbearbeitet wurde, wird durch den Vorgang des *Merging* (Zusammenführen) wieder in die Ursprungs-Codeline eingebracht. Durch die Abkopplung erreicht man eine gewisse Unabhängigkeit von den zeitgleich laufenden und oft noch mit Fehlern behafteten Codeänderungen anderer Teammitglieder und kann dadurch die eigene Entwicklung auf einer relativ soliden Codebasis aufbauen. Andererseits sind Änderungen, die seit der Abkopplung in anderen *Codelines* vorgenommen wurden, in der eigenen Codeline natürlich noch nicht enthalten (Ausnahme: *Propagating*, s.u.). Aufgrund dieser Tatsache kann es beim Zusammenführen zu sog. Code-Konflikten kommen, d.h. die beiden zu mergenden Codeteile passen syntaktisch und/oder semantisch nicht zusammen. Um die syntaktischen Konflikte (Beispiel: Benutzung einer Klasse, deren Definition sich nach dem *branching* ändert) aufzulösen, bedient man sich häufig Softwarewerkzeugen, die diese Arbeit zu einem guten Teil automatisch erledigen oder, falls der Konflikt nicht automatisch lösbar ist, zumindest die konfliktbehafteten Stellen im Code dem Entwickler in einer effizient zu handhabenden Form präsentieren. Das Auflösen der semantischen Konflikte ist i.d.R. nicht automatisierbar, sondern es muß sich ein Entwickler mit dem *Merge* beschäftigen, zu dem neben der Anpassung und Korrektur

des Codes natürlich auch noch die Testphase gehört, die erfolgreich abgeschlossen werden und dem eigentlichen Hinzufügen des Codes zur Codeline vorausgehen muß. Der zeitliche und personelle Aufwand des Merge-Vorgangs spielt aus diesem Grund eine wichtige Rolle, wenn es um die Entscheidung für oder gegen mehr Parallelität geht. Ebenfalls muß das Auswählen und Anpassen der allgemein betriebenen Strategie (s.u.) den Merging-Aufwand berücksichtigen, da die Faktoren, die den Aufwand eines einzelnen Merging-Vorgangs beeinflussen können, durch Mechanismen wie *Propagating* (s.u.) oft auf große Teile des gesamten Code-Baums verteilt sind (unter Code-Baum sei hier einfach die Gesamtheit aller Codelines verstanden). Das Ziel der allgemeinen Strategie sowie das der einzelnen "Codeline-Policy" (s.u.) ist es, den durch Parallelisierung entstandenen Merging-Aufwand wesentlich geringer zu halten als die dadurch gewonnene Effizienzsteigerung.

- *Propagating*

  Damit eine *Codeline* A sich nicht zu weit vom restlichen Code entfernt (also größeren Merging-Aufwand verursacht) kann man durch *Propagating* (Verteilen, Weiterreichen) Code von anderen *Codelines* integrieren oder auch Zwischenstufen dieser *Codeline* regelmäßig in eine oder mehrere andere *Codelines* verteilen. Als Beispiel sei hier eine Fehlerkorrektur (Bugfix) genannt, die z.B. in einer parallelen *Codeline* B vorgenommen wurde, aber Code betrifft, der in dem gemeinsamen Ursprung der beiden *Codelines* liegt, also demnach auch in *Codeline* A vorgenommen werden müßte. Damit dieser Bugfix auch der *Codeline* A zugute kommt, kann man nun diesen Bugfix von B nach A durch einen Merge-Vorgang einbringen. Die beiden *Codelines* bleiben auch weiterhin bestehen, es wurde also nur ein Stück Code weitergereicht. In komplexen Entwicklungsszenarien ist es auch durchaus möglich, daß solche *Propagating*-Schritte zwischen bestimmten *Codelines* in festgelegten, regelmäßigen Abständen durchgeführt werden müssen, um die Funktionalität und Aktualität der einzelnen *Codelines* zu gewährleisten. Wenn die gewählte allgemeine Verwaltungsstrategie des Code-Baums "schlecht" ist, dann kann es in einer solchen Situation schnell zu einem schwer zu durchschauenden Geflecht an Abhängigkeiten zwischen *Codelines* kommen, die zur Folge haben, daß eine einzige Änderung an einer *Codeline* mehrere Merge-Vorgänge nach sich ziehen kann.

3. Die Pattern-Sprache

Die Pattern-Sprache “Streamed Lines” besteht aus ca. 40 Patterns. Einige dieser Patterns bauen z.T. aufeinander auf, andere Patterns sind vom Rest unabhängig. Die Patterns sind grob in 3 Gruppen eingeteilt; dies sind “Branching Policy Patterns”, also Patterns, die sich um das Erstellen und Erhalten eines durchgängigen Konzepts für eine Codeline kümmern, “Branch Creation Patterns”, also Patterns, die Richtlinien für das Erzeugen einer neuen Codeline beschreiben und “Branch Structuring Patterns”, die von den Interaktionen zwischen mehreren Codelines handeln.

Es werden nun einzelne Patterns im Detail beschrieben sowie darauf folgend weitere Patterns kurz umrissen.

3.1 Einzelne Patterns im Detail

“Codeline Policy”

Dieses Pattern besagt, daß für jede Codeline eine klar festgelegte und einsichtige Richtlinie erstellt werden soll, an die sich die Entwickler halten sollen, wenn sie Code in diese Codeline einbringen wollen.

<u>Kontext:</u> es wird auf mehreren, parallel verlaufenden Codelines gleichzeitig entwickelt.

<u>Problem:</u> anhand welcher Kriterien soll ein Entwickler, der Code erstellt hat, entscheiden, in welche Codeline und zu welchem Zeitpunkt er diesen Code einbringen soll?

<u>Forces:</u>

- jede Codeline ist für einen spezielle Zweck erstellt worden (Bugfixing, Weiterentwicklungen, Portierungen usw.)
- der Name einer Codeline wird von den Entwicklern normalerweise als Hinweis auf ihren Zweck aufgefaßt
- andererseite sagt der Name einer Codeline längst nicht alles über deren Verwendungszweck aus, sowie über Einzelheiten zu deren Benutzung
- Aussuchen der “falschen” Codeline (d.h. einer Codeline, in der einige der einzubringenden Änderungen nichts zu suchen haben) wird sicherlich unnötigen Zusatzaufwand zur Folge haben, da beim Einbringen diese unerwünschten Änderungen eliminiert werden müssen.

<u>Lösung:</u> es reicht nicht aus, einer Codeline nur einen “sinnvollen” Namen zu geben, sondern es sollten noch konkrete und klar formulierte Benutzungs- und Einsatzhinweise zu der Codeline erstellt werden. Diese sollten wenigstens folgende Punkte umfassen:

- Welche Arbeit soll auf der Codeline verrichtet werden? Kategorisierungen sind z.B. Bugfixing, Weiterentwicklungen, ein spezielles Release, ein Teilprojekt, usw.
- Wann und wie Codeteile aus der Codeline entnommen und durch Merge zurückgeführt werden sollen. Z.B. könnte man hier festlegen, daß bestimmte Codeteile nur mit einem “exklusiven Checkout” verwendet werden dürfen, d.h. daß

immer nur ein einziger Entwickler zur gleichen Zeit dieses Stück Code ändern darf, da es sich um einen besonders kritischen Abschnitt handelt.

- Festlegen, wer auf diese Codeline zugreifen darf und wer nicht. Es können Einzelpersonen oder Gruppen angegeben, aber auch eine bestimmte Personengruppe anhand ihrer Funktion oder Stellung benannt werden.
- Abhängigkeiten von anderen Codeline klar herausstellen, d.h. von welchen anderen Codeline Änderungen übernommen werden dürfen oder müssen, oder welche anderen Codelines Änderungen in dieser Codeline ebenfalls erhalten müssen.
- Voraussichtliche Dauer und Bedingungen für ein Beenden der Codeline

"Policy Branch"

Dieses Pattern bietet einen Lösungsweg an für den Fall, daß Entwickler im Sinne des oben beschriebenen Patterns "Codeline Policy" mit der Richtlinie einer Codeline in Konflikt geraten.

Kontext: mehrere Entwickler arbeiten auf einer Codeline, für die im Sinne von "*Codeline Policy*" eine Richtlinie festgelegt wurde.

Problem: Einige der Entwickler sind mit der Richtlinie der Codeline nicht einverstanden und würden sie gerne ändern.

Forces:

- Ein Ändern der Codeline-Richtlinie hätte nachteilige Konsequenzen für die anderen Entwickler.
- Hinzufügen von weiteren Termen zur Codeline-Richtlinie kann die gesamte Richtlinie der Codeline unklar werden lassen und langfristig die Zweckmäßigkeit der Codeline in Frage stellen.

Lösung: Eine neue Codeline erzeugen mit einer neuen "*Codeline Policy*", auf der die Entwickler arbeiten können, die mit der alten Richtlinie unzufrieden waren.

"Branch Per Task"

Dieses Pattern besagt, was zu tun ist, wenn bei einer Menge an gleichzeitig auszuführenden Änderungen Eingriffe in die gleichen Dateien nötig werden.

Kontext: Eine Änderung, die an der Codeline gemacht werden muß, betrifft Dateien, die gleichzeitig auch für andere Änderungen gebraucht werden.

Problem: Wie bringt man alle Änderungen konflikt- und fehlerfrei in die Codeline ein?

Forces:

- Gleichzeitige Codeentwicklung ohne kontrollierte Interaktion kann unbrauchbaren oder fehlerhaften Code zur Folge haben und damit Arbeitsverlust.
- Zuviele "Dateisperren" (d.h. daß nur noch ein einziger Entwickler auf eine Datei zugreifen kann) verursacht lange Wartezeiten und kann schlimmstenfalls bis zur Verklemmung führen (A braucht B, B braucht aber A).

Lösung: Für jede einzelne Änderung eine neue "kurze" Codeline von der Hauptcodeline abzweigen. Sobald die Änderung inkl. Testen vollständig implementiert worden ist, diese neue Codeline zurückmergen. Dies kann entweder vom Autor der Änderung oder auch vom Owner (Betreuer, Inhaber) der Hauptcodeline vorgenommen werden.

"Docking Line"

Dieses Pattern beschreibt eine Methode, wie Konflikte bei der Zuständigkeit für einen Merge-Vorgang beseitigt werden können.

Kontext: Eine Änderung, die von einem Entwickler durchgeführt wurde (der "Change-Owner") soll in eine sehr kritische Codeline gemerged werden. Für diese Codeline ist ein anderer Entwickler zuständig (der "Codeline-Owner"). Es ist noch wichtiger als bei anderen Codelines, daß die Codeline nach dem Merge-Vorgang in einem korrekten und konsistenten Zustand ist.

Problem: Wer soll den Merge durchführen und wer ist für den nachfolgenden Zustand der Codeline verantwortlich?

Forces:

- Es ist wichtig, daß Change-Owner die Auswirkungen seiner Arbeit mitverfolgen kann, damit er Verantwortungsgefühl für seine Arbeit behält. Wenn er regelmäßig von der Pflicht entbunden wird, seine Änderungen konflikt- und fehlerfrei in die Hauptcodeline zurückzumergen, wird er früher oder später das Interesse an diesem Vorgang verlieren. Da man eine Änderung mit vergleichsweise wenig Zusatzaufwand so gestalten kann, daß der Merge-Vorgang deutlich einfacher und sicherer vonstatten geht, ist es aber sinnvoll, daß der Entwickler an diesen Prozeß beteiligt wird.
- Im Endeffekt ist es immer der Codeline-Owner, der für das Funktionieren der Codeline geradestehen muß, d.h. falls der Change-Owner den Merge durchführt, ist er letztendlich trotzdem verantwortlich, d.h. er möchte den Merge eigentlich am liebsten selbst durchführen.

Lösung: Der Change-Owner merged seine Änderungen in eine permanent zur Codeline mitgeführte "Integrations-Codeline" und sorgt dafür, daß seine Änderungen mit dem aktuellen Entwicklungsstand kompatibel, d.h. konflikt- und fehlerfrei sind. Der Codeline-Owner ist dann dafür zuständig, diese dann ja schon größtenteils angepaßten Änderungen von der Integrations-Codeline in seine Codeline zu mergen.

3.2 Weitere Patterns

"Platform Line"

Dieses Pattern besagt, daß es sinnvoll ist, für jede Zielplattform, für die man eine Software entwickelt (Cross-Platform), eine eigene Codeline anzulegen. Die Änderungen, die auf der Codeline für eine der Plattformen gemacht wird, kann man durch *propagating* in die Codelines der anderen Plattformen übernehmen. Auch ist es sinnvoll, eine zentrale Codeline für plattform-unabhängigen Code zu halten, von dem aus dann dieser Code in die plattformspezifischen Codeline gemerged wird. Die plattformspezifischen Codelines werden naturgemäß nicht mehr zur Haupt-Codeline zurückgemerged, sondern haben eine unbegrenzte Lebensdauer.

"Merge Your Own Code"

Nach diesem Pattern ist es im Normallfall immer am besten, wenn der Autor einer Änderung (Change-Owner) seinen eigenen Code in eine Codeline merged, da er den zu mergenden Code logischerweise am besten kennt und aufgrund des Neueinbringens evtl. auftretende Fehler auch schneller (dem Change-Owner) zugeordnet werden können. Weiterhin hält es den Autor des Codes auch mehr in seiner Verantwortung, was letztendlich auch die Voraussetzung für die zukünftige Qualität des Codes dieses Entwicklers ist. Nur in Ausnahmefällen sollte der Codeline-Owner selbst den Merge durchführen. In schwer entscheidbaren Fällen sollte das Pattern "*Docking Line*" angewendet werden.

"Subproject Line"

Wenn eine Änderung aus mehreren Teiländerungen besteht, die z.T. auch parallel abgearbeitet werden können, und das Einbringen der Hauptänderung nur als Ganzes Sinn macht, dann sollte für diese Änderung eine eigene Codeline erzeugt werden, in der dann die Teiländerungen schrittweise (seriell oder parallel) durchgeführt und gemerged werden. Wenn alle Teiländerungen implementiert sind, dann wird die neue Codeline als Gesamtänderung wieder in die Hauptcodeline zurückgemerged.

"Staged Integration Lines"

Wenn z.B. die Qualitätssicherung einer Softwareprojektes aus mehreren hierarchisch aufgebauten Stufen besteht, oder z.B. die Zuständigkeiten und damit die Verantwortlichkeiten der beteiligten Entwickler auf einer Hierarchie basieren, die geschriebener Code durchlaufen muß, dann ist es sinnvoll, diesen Hierarchien entsprechende Codelines anzulegen. So kann ein Stück neuer Code durch einen Merge-Vorgang einfach in die "nächste" Stufe gebracht werden und seine Verantwortlichkeit dem nächsten Teammitglied übertragen werden.

4. Die Anwendung

Die hier vorgestellten Patterns stellen natürlich nur einen Auszug aus der umfangreichen Pattern-Sprache dar. Außerdem ist die Sprache nicht so aufgebaut, daß man unbedingt jedes einzelne der Patterns auch zur Anwendung bringen muß, um die Sprache “sinnvoll” einsetzen zu können, sondern man kann für den Anfang eine Untermenge von etwa 5 bis 6 Patterns zusammenstellen, die eher zu den allgemeineren “Haupt”-Patterns gehören und am Anfang nur mit diesen arbeiten. Wenn das Projekt wächst, kann man für differenzierte Aufgaben noch weitere Patterns hinzunehmen oder Patterns aus dieser Teilmenge gegen andere Patterns austauschen, die einem zu den wechselnden Ansprüchen jeweils passend erscheinen.

Ebenfalls ist es wichtig, die Auswahl an Patterns ständig daraufhin zu überprüfen, ob sie mit der gewünschten Arbeitsweise überhaupt in Einklang zu bringen sind, denn nicht alle Patterns zielen auf die gleiche Strategie ab. Es kann nämlich auch durchaus sein, daß während der Projektlaufzeit ein Strategiewechsel erforderlich wird und dementsprechend muß man auch die dazu passenden Patterns auswählen.

Im folgenden Abschnitt werden einige der Möglichkeiten angesprochen, mit Hilfe der Sprache eine differenzierte und auf das Projekt zugeschnittene Strategie zu entwickeln.

4.1 Verzweigungsstrategien

“Branch per Major Task / Branch per Minor Task”

Man kann für große oder kleine Änderungen/Implementierungen eine neue Codeline erzeugen, je nachdem, ob man mehr Sicherheit oder mehr Effizienz erreichen will.

“Early / Deferred Branching”

Abzweigen einer neuen Codeline kann man eher früher vornehmen, z.B. wenn Konflikte schon absehbar sind, oder eher später durchführen z.B. wenn diese Konflikte erst eingetreten sind.

“Relaxed / Restricted Merging Style”

Ebenfalls kann man festlegen, ob die doch recht zeitkritischen und fehleranfälligen Merging-Vorgänge von einem ziemlich beliebigen Entwickler durchgeführt oder eher nur von ausgewählten Personen vorgenommen werden darf.

“Relaxed / Restricted Access Line”

Zu den möglichen Strategien gehört auch das Festlegen von Zugriffsrechten. Diese kann man eher lax halten und jedem Entwickler den Zugriff auf den größten Teil des Codes erlauben (was durchaus kreativitätsfördernd sein kann) oder eher restriktiv, was das Risiko von durch mangelnde Code-Kenntnisse eingebrachten Fehlern verringert und auf einen relativ engen Personenkreis beschränkt.

4.2 Entscheidungsfaktoren

Welche Strategie man wählt, hängt letztendlich von einer Entscheidung für die Menge an Risiko ab, die man zur Durchführung des Projektes auf sich nehmen will. Wenn mehr Effizienz, d.h. Auslastung der zur Verfügung stehenden Entwickler gewünscht ist, muß man versuchen, dies durch einen stärkeren Einsatz von Parallelität verwirklichen, d.h. es wird mehr gleichzeitige Arbeit am Code notwendig, um das gesteckte Ziel zu erreichen. Dies führt aber auch zu einem höheren Risiko, da bei gleichzeitigen Entwicklungen erstens die Wahrscheinlichkeit von (evtl. sogar unbemerkten) Fehlern wächst und zweitens auch immer die latente Gefahr besteht, durch überhöhte Parallelität einen sprunghaftt anwachsenden Koordinationsaufwand (dies beinhaltet den Merging-Aufwand und den Team-Koordinationsaufwand) bewältigen zu müssen. Ob dieser Koordinationsaufwand letztendlich vielleicht sogar die zulässige Grenze übersteigt und damit das gesamte Projekt in Gefahr bringt, weiß man oft erst, wenn es fast zu spät ist. Die "zulässige Grenze" ist dann erreicht, wenn der Implementierungsaufwand zuzüglich des Koordinationsaufwands den Aufwand übersteigt, der bestünde, wenn man das Projekt mit einem eher "sicheren", also wenig parallelen Fahrplan bearbeiten würde. Das höhere Risiko stärkerer Parallelisierung nimmt man ja schließlich nur gegen eine möglicherweise höhere Effizienz in Kauf.

4.3 Auswirkungen

Die Erfahrungen und die Methoden, um "typische" Fehler, die während eines parallelisierten Projektablaufs gemacht werden, zu vermeiden, will die Pattern-Sprache in eine verwertbare Form bringen. Wichtig ist die Anwendung der Patterns deshalb, weil mit ihnen schon in der Initialphase des Projekts bestimmte Dinge "richtig" gemacht werden können, auch ohne daß das Team ein Feedback bekommt, das ja erfahrungsgemäß erst nach einer gewissen Zeit zur Verfügung steht. Zum Feedback gehört u.a. das sich ja ständig ändernde Verhältnis zwischen der einzelnen Arbeitsaufgaben eines Teammitglieds, zu denen u.a. einerseits das Codieren von "neuer" Funktionalität und andererseits das Bugfixing (Fehlerkorrektur) gehört. Letzteres ist eben auch stark abhängig von der Menge und dem Umfang von Merging-Aktivitäten. Wenn der Aufwand für Bugfixing nach Merge-Aktivitäten stark zunimmt, dann ist dies oft ein Zeichen dafür, daß die Aufgabenverteilung und Planung der Parallelisierung nicht optimal auf das Projekt abgestimmt sind und die notwendige Koordination den Projektfortgang eher behindert als fördert.

Eine ganz allgemeine Aussage, die die Sprache in diesem Zusammenhang macht, ist, daß die Integrationsaktivitäten (also das Mergen und das Propagaten) möglichst häufig und so früh wie möglich geschehen sollen. Diese Vorgehensweise kann vorhandene Fehler frühzeitig aufdecken und hält das Projekt "am Leben", indem auch die Kommunikation zwischen den Entwicklern gefördert wird, weil sie an den Auswirkungen ihrer Arbeit intensiver und direkter beteiligt sind (siehe Pattern "*Merge Your Own Code*"). Entsprechend ist auch das Ausbleiben von häufiger Integration oft ein Vorzeichen für den Mißerfolg eines Projekts.

Unabhängig davon läßt die Sprache dem Anwender jedoch die Freiheit, den Grad und die Art der Parallelität für das Projekt seinen persönlichen Vorlieben, Erfahrungen und auch dem Aufbau und den Eigenschaften des Teams anzupassen, ohne ihn weder in eine bestimmte Richtung zu drängen noch ihn an den entscheidenden Stellen im Stich zu lassen.