Seminar Software Engineering
WS 03/04

Refactoring und Patterneinsatz in evolutionärer Umgebung

Ausarbeitung zum Vortrag vom
20. Januar 2004

von

Jan Krebs
98-918-055

Institut für Informatik
Universität Zürich

Dozent: Prof. Dr. Martin Glinz
Betreuender Assistent: Christian Seybold

Inhaltsverzeichnis

1. Einführung

In den Augen vieler Kritker scheinen Agile Methoden (insbesondere XP)n den Tod von Software Design zu fordern. Stimmt das, oder haben agile Methoden nur ein neues Verständnis von Design?

Klassische Software-Entwicklungsprozesse fordern, dass zu Beginn eines Entwicklungszyklus aufgrund der Gesamtheit der erhobenen Anforderungen eine Grobarchitektur des Systems erstellt wird und zu verwendende Entwurfsmuster festgelegt werden. Wenn das Gerüst steht, entworfen von "erfahrenen Entwicklern" und "Architekten", werden die Klassen und Methoden anschliessend vom "einfachen" Programmierer implementiert. Während des Implementierens werden dabei keine Änderungen am ursprünglichen Design mehr vorgenommen.

Agile Methoden, wie Kent Becks eXtreme Programming (XP), haben allerdings einen anderen Ansatz gewählt. Sie verzichten auf grosse Designaktivitäten zu Projektbeginn ("Big Up Front Design") und verabscheuen das Verfassen von grafischen Entwurfsdokumenten z.B. mittels UML. Auf den Einsatz von Entwurfsmustern und flexiblen Frameworks wird nicht viel Wert gelegt, ja ihr Einsatz ist gar verboten, wenn man dafür heute Arbeit investieren müsste, deren Ertrag man erst in Zukunft ernten könnte (oder auch nicht). Obwohl es scheint, dass in XP kein Platz für Design ist, wird natürlich auch in XP Design gemacht, nur an anderer Stelle: XP hat die Idee von evolutionärem Design wiederaufgegriffen. XP benutzt Praktiken die es erlauben, dass Evolution selbst zur Design-Strategie wird. "Simple Design", kontinuierliches Refactoring, kontinuierliche Integration, systematisches Testen, sind alles Techniken, die es erlauben, dass das Design aus dem Code entstehen kann, und sind nicht zu verwechseln mit "code and fix".

In Kapitel 2 werden die verschiedenen Designstrategien von klassischen und agilen Methoden behandelt, während Kapitel 3 Refactoring als eine effektive Methode zur Verbesserung des Design von existierendem Code vorstellt. Sie ist nicht nur zentral für XP, sondern die Verwendung einer disziplinierten Methode für Code-Redesign kann meines Erachtens auch in vielen anderen Gebieten (z.B. Wartung) nützlich sein. Kapitel 4 behandelt dann die Design-Überlegungen von XP und geht auf einige immer wieder kritisierte Punkte ein.

2. Verschiedene Design-Strategien

Nach [Fowler01] existieren grundsätzlich 2 Arten von Design: Geplantes Design (*planned design*) und evolutionäres Design (*evolutionary design*).

2.1. Evolutionäres Design

Evolutionäres Design bedeutet, dass das Design wächst, während das System implementiert wird. Design ist Teil des Programmierprozesses und während das Programm entsteht, ändert sich das Design.

Normalerweise ist evolutionäres Design ein Desaster. Das Design am Schluss ist eine Aggregation von taktischen ad-hoc-Entscheidungen, jede dieser Entscheidungen macht es schwieriger den Code in Zukunft zu ändern.

Die Hauptaufgabe von Design ist, gemäss Kent Beck (in [Fowler00]), es zu ermöglichen, das Programm langfristig einfach ändern zu können. Mit einem schlechten Design gerät man in den Zustand von Software-Entropie: Das schlechte Design ist schwieriger zu ändern und mit jeder Änderung wird es schlechter und schlechter, nun schleichen sich auch die Fehler ein und werden

immer mehr. Die Fehlerkosten nehmen exponentiell zu während das Projekt fortschreitet.

2.2. Geplantes Design (big up front design)

Geplantes Design ist eine Ingenieursdisziplin. Oftmals wird es verglichen mit dem Bau eines Hauses, wo Architekten und Ingenieure sich um die grossen Entwurfsentscheidungen kümmern, und als Abschluss ihrer Aktivität die Pläne den Handwerkern (und einem Bauleiter, der die Aktivitäten auf der Baustelle koordiniert) übergeben.

Weil die Architekten sich auf einer höheren Abstraktionsebene bewegen als die Implementierer, können sie es vermeiden diese Serie von ad-hoc-Entscheidungen zu machen, die bei evolutionärem Design zu Software-Entropie führen. Und wenn die Programmierer genau dem Bauplan folgen, dann hat man am Schluss ein gut designtes System.

Doch geplantes Design hat auch Nachteile, mit denen es zu kämpfen hat:

- *Es ist nicht möglich im Vornherein alle Probleme und Fragen, die beim Implementieren auftauchen werden durchzudenken.*
  Die Programmierer werden beim Implementieren sehr wahrscheinlich auf Probleme stossen, die das Design in Frage stellen. Was dann? Ändern die Designer die Pläne (zeit- und kostenintensiv) oder lösen die Programmierer die Probleme einfach selbst (code and fix). Auch beim geplanten Design ist Software-Entropie ein mögliches Endstadium.
- *Kulturelles Problem.*
  Designer werden Designer wegen ihrer Fähigkeiten und Erfahrung. Aber wenn sie nur noch designen, dann haben sie keine Zeit mehr zum Programmieren. Die Werkzeuge und Materialien in der Software-Entwicklung ändern sich schnell. Und wer nicht mehr dabei ist, verliert nicht nur den Anschluss an die technologische Entwicklung, sondern oft auch den Respekt vor den Programmierern.
- *Ändernde Anforderungen*
  Wie soll auf ändernde Anforderungen reagiert werden?
  Eine Möglichkeit ist, Flexibilität ins Design einzubauen, so dass man einfach Veränderungen vornehmen kann, wenn die Anforderungen ändern. Dieser Ansatz setzt aber voraus, dass man weiss, welche Änderungen an den Anforderungen man erwartet.
  Viele dieser Probleme mit Anforderungen kommen aber daher, dass man die Anforderungen einfach nicht klar genug verstanden hat. Viele richten ihr Augenmerk daher auf Requirements Engineering Prozesse, die bessere Anforderungen liefern, in der Hoffnung, dass damit das Design später nicht mehr geändert werden muss.
  Aber was ist, wenn sich beim Auftraggeber die Geschäftsprozesse ändern und sich daraus neue , unvorhersehbare Anforderungen ergeben? Dies kann auch mit dem besten Requirements Engineering Prozess nicht verhindert werden.

2.3. Der Ansatz von agilen Methoden

Agile Methoden verzichten auf “big up front design” und benutzen einen evolutionären Design-Ansatz aus 2 Gründen:

- Geplantes Design hat viele Nachteile (siehe oben)

- Agile Methoden wie XP versprechen, dass sie die Kostenkurve für Änderungen so flach biegen können, dass evolutionäres Design nicht mehr teuer ist.
  Um die Kostenkurve zu biegen, kennt XP verschiedene Praktiken (*enabling practices*). Die wichtigsten sind wohl die kontinuierliche Integration und das rigorose Testregime. Andere Praktiken versuchen dann die Vorteile der so gebogenen Kostenkurve auszunützen (*exploiting practices*).

Das Testen sorgt für die nötige Sicherheit, dass spätere Änderungen die Funktionalität nicht verändern; und über die kontinuierliche Integration werden alle Mitglieder des Entwicklungsteams "synchronisiert", so dass späte Änderungen möglich sind, ohne fürchten zu müssen, dass die Änderungen Probleme bei der Integration aufwerfen werden.

Eine weiter Praktik, Refactoring, erlaubt es einem das Design des Codes im Nachhinein auf eine disziplinierte, fehlervermeidende Art zu ändern (im Gegensatz zu "code and fix").

Kontinuierliche Integration, Testen und Refactoring schaffen eine neues Umfeld, dass evolutionäres Design plausibel macht.

3. Refactoring

3.1. Definition

Für das Wort "Refactoring" gibt es zwei Definitionen, abhängig davon, ob es als Nomen oder als Verb gebraucht wird:

3.1.1. Nomen

Refactoring(noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior. [Fowler00]

- Ein Refactoring ist also eine kleine Änderung an der Struktur der Software (wie. z.B "Extrahieren einer Methode" (*Extract Method*)), die das externe Verhalten aber nicht verändert.
- Einige komplexere Refactorings beinhalten mehrere einfacherer Refactorings. ("Extrahieren einer Klassse" (*Extract Class*) beinhaltet normalerweise "Bewege Methode" (*Move Method*) und "Bewege Instanz-Variable" (*Move Field*).

3.1.2. Verb

Refactor(verb): to restructure software by applying a series of refactorings without changing its observable behavior. [Fowler00]

- Man kann also eine gewisse Zeit damit verbringen zu "refaktorieren" (das Verb), während der man eine gewisse Anzahl von Refactorings (das Nomen) auf dem Code anwendet.

3.2. Was ist Refactoring?

"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It's a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written" [Fowler00]

Fowler wurde einmal gefragt, ob es denn bei Refactoring nur darum gehe, den Code etwas aufzuräumen (*to clean up code*). Die Antwort war ein "ja, aber...":

- Refactoring ist eine Technik, mit der man Code in einer weitaus effizienteren und kontrollierteren Art und Weise aufräumt.
- Man räumt seinen Code viel wirkungsvoller auf, weil man weiss, welche Refactorings man gebrauchen soll, und weil man sie so gebraucht, dass die Entstehung von Fehlern im Code (*bugs*) möglichst vermieden werden kann.

3.3. Das Ziel von Refactoring

Das Ziel von Refactoring ist, Software so zu verändern, dass sie nachher einfacher zu verstehen und auch einfacher zu ändern ist.

Neben Refactoring gibt es auch andere Techniken, z.B. Performance-Optimierung, die nur die interne Struktur von Code verändern, während das Verhalten (bis auf die Geschwindigkeit) gleich bleibt. Performance-Optimiertung hat aber ganz ein anderes Ziel und verändert den Code so, dass er danach meist schwerer zu verstehen ist als zuvor. Man muss es aber manchmal tun, um die Leistung zu erreichen, die gefordert ist.

3.4. Warum soll man Refactoring betreiben?

Refactoring ist nicht die ultimative Lösung (*silver bullet*) für jedes Problem, aber es ist ein gutes Werkzeug, das für verschiedene Zwecke genutzt werden kann und soll [nach Fowler00]:

- *Refactoring verbessert des Software-Design*
  Programmierer verändern den existierenden Code, um kurzfristige Ziele zu erreichen ("noch schnell ein Feature einbauen..") oder ohne das Design des Ganzen genügend zu verstehen. Als Folge verliert der Code seine Struktur. Der Verlust von Struktur hat einen kumulativen Effekt: Je schwieriger es ist, das Design im Code zu erkennen, umso schwieriger ist es, das Design zu bewahren, und umso schneller verdirbt (*to decay*) das Design. Wenn man regelmässig Refactoring betreibt, dann bleibt der Code in Form.

- *Refactoring macht Software leichter verständlich*
  Schlecht verständlicher Code kann ins Geld gehen. Vorallem wenn ein anderer Programmierer den Code, den man geschrieben hat, verändern muss, und anstatt Stunden Wochen braucht, um überhaupt zu verstehen wie er funktioniert. Refactoring überführt schlecht strukturierten Code, der läuft, in besser strukturierten Code; Nur ein wenig Zeit mit Refactoring zugebracht kann helfen, dass der Code selbst besser kommuniziert, was sein Zweck sein soll.
  Fowler gebraucht angeblich Refactoring sogar um Code, den andere geschrieben haben, besser zu

verstehen (etwa für ein Review). Wenn er versucht Code zu verstehen, dann belässt er es nicht dabei seine Erkenntnisse darüber, wie er denkt, dass das Programm funktioniere, in seinem Kopf herumschwirrren zu lassen oder ein Dokument zu verfassen. Nein, er ändert die Struktur des Codes so mittels Refactoring, dass der Code für sich selbst spricht. Zuerst macht er Refactoring auf kleinen Details, und dann, wenn der Code immer klarer wird, lassen sich auch Probleme auf höherer Ebene erkennen. Und diese Probleme würde er sonst nie sehen, da er sich das ganze nämlich gar nie in seinem Kopf visualisieren könnte.

- *Refactoring hilft einem Fehler im Programm (bugs) zu finden*
  Indem einem Refactoring hilft, den Code besser zu verstehen, hilft es einem, Fehler zu finden. Meist muss man den Code nämlich verstehen, damit einem die Fehler auffallen.

- *Refactoring hilft einem schneller zu programmieren*
  Verbessern des Designs, verbessern der Leserlichkeit und vermindern der Fehlerrate erhöht die Qualität der Software. Die Qualität ist essentiell für eine schnelle Software-Entwicklung.
  Ohne gutes Design kommt man zwar anfangs schnell voran, doch nach einer gewissen Zeit beginnt schlechtes Design den Software-Entwicklungsprozess zu verlangsamen. Man verwendet viel Zeit darauf, Fehler zu finden und zu beheben, anstatt, dass man neue Funktionalität hinzufügt. Veränderungen brauchen mehr Zeit, weil man viel Zeit aufwenden muss, um den Code erst zu verstehen und duplizierten Code aufzuspüren.
  Ein gutes Design ist essentiell, um eine hohe Geschwindigkeit im Software-Entwicklungsprozess zu bewahren. Refactoring hilft einem, Software schneller zu entwickeln, da es das Design des Systems davon abhält zu verderben und es möglicherweise sogar verbessert.

3.5. Entstehungsgeschichte

Die Refactoring-Bibel wurde von Martin Fowler [Fowler00] geschrieben, unter Mithilfe einiger der Überväter der Software-Entwicklung wie Kent Beck und Erich Gamma.

Wann genau der Term “Refactoring” entstanden ist, ist unklar, aber guten Programmierern sei es schon immer klar gewesen, dass sie nicht immer “sauberen” Code von Anfang an schreiben, und dass es wichtig ist, den Code von Zeit zu Zeit zu “reinigen”.

Kent Beck und Ward Cunningham, die seit den 1980igern mit Smalltalk arbeiteten, sollen unter den Ersten gewesen sein, die die Bedeutung von Refactoring erkannt haben.

1992 hat dann Bill Opdyke, ein Doktorand von Ralph Johnson (Gang Of Four), der an der Entwicklung von effizienten und flexiblen Frameworks interessiert war, eine Doktorarbeit über Refactoring geschrieben.

Und da niemand von den grossen (auch nicht Kent Beck, von dem er es gelernt hatte beim Pair-Programming im Flugzeug) bereit war, ein Buch nur über Refactoring zu schreiben, hat Fowler sich schliesslich dazu überwunden.

3.6. Vorgehensweise

3.6.1. Die zwei Hüte

Wenn man Refactoring als Methode gebraucht, um Software zu entwickeln, dann muss man seine Zeit strikt in 2 Aktivitäten unterteilen:

- *Hinzufügen von neuer Funktionalität*
  Wenn man neue Funktionalität hinzufügt, dann sollte der schon bestehende Code nicht geändert werden. Man misst den Fortschritt, indem man für die neuen Funktionen Tests schreibt, und diese Tests dann zum Laufen bringt.
- *Refactoring*
  Wenn man in der Refactoring-Phase ist, dann fügt man keine neue Funktionalität hinzu, man restrukturiert nur den Code. Man schreibt keine neuen Tests (es sei denn man findet einen Testfall, den man zuvor vergessen hatte abzudecken). Man ändert Tests nur, wenn man das auf jeden Fall machen muss, weil man im Zuge des Refactorings eine Schnittstelle geändert hat.

Kent Beck (XP) gebraucht dafür die Metapher von den zwei Hüten. Entweder trägt man den einen, oder den anderen, aber sicher nie beide gleichzeitig. Wenn man wechseln will, dann muss man den einen ablegen, und den anderen aufsetzen. Des weiteren sollte man sich auch immer bewusst sein, welchen von beiden Hüten man im Moment gerade trägt, so dass man nichts durcheinander bringt.

Meist wechselt man vom einen zum anderen recht häufig. Man möchte eine neue Funktionalität hinzufügen, und merkt dabei, dass das einfacher ginge, wenn der Code anders/besser strukturiert wäre. Also Hut wechseln und Code restrukturieren, dann wieder wechseln und neue Funktionalität hinzufügen. Wenn die neue Funktionalität dann hinzugefügt ist, merkt man vielleicht, dass man das gar nicht sauber gemacht hat, und der Hut wird wieder gewechselt...

3.6.2. Testen nach jedem Schritt

Tests sind das A und O des Refactorings.

Kritiker von agilen Methoden (XP insbesondere) und von Refactoring als Design-Methode sind angeblich häufig solche, die die Methoden ausprobiert und sich dabei die Finger verbrannt haben. Dabei haben sie versucht nur die Teil-Methoden auszunutzen die Vorteile versprechen, ohne auch die Teil-Methoden anzuwenden, die die Vorteile erst ermöglichen.

Man kann nicht einfach Martin Fowlers Buch über Refactoring [Fowler00] kaufen, den Katalog auf Seite x aufschlagen, und das vorgeschlagene Refactoring Muster auf seinen Code anwenden, ohne eine Möglichkeit zu haben, um sicherzustellen, dass sich das externe Verhalten des Systems durch die Änderung am Design nicht ändert. Wenn man es so macht, dann kann man ein gut funktionierendes, aber schlecht designtes System geradewegs in ein nicht mehr funktionierendes System verwandeln, und sich im alten Grundsatz “never change a running system” bestätigt fühlen.

Daher ist Testen ein existentieller Bestandteil von Refactoring. Nach jedem noch so kleinen Refactoring-Schritt soll kompiliert werden und es soll getestet werden. Oftmals, wenn man seinen Code restrukturieren möchte, sieht man plötzlich Tausende von Sachen, die man besser machen könnte. Hier was ändern, da was ändern, und hier auch noch gerade..Kompilieren, Testen...und es tut nicht mehr was es sollte. Refactoring als Methode braucht Disziplin, und man darf sich nicht zu voreiligem Handeln verleiten lassen.

Um aber nach jedem noch so kleinen Schritt zu testen, braucht man auf jeden Fall automatische, selbstprüfende Tests, wie man sie z.B. mit Hilfe des JUnit-Frameworks für Java einfach erstellen kann. JUnit ist ein open-source Test-Framework, das von Erich Gamma und Kent Beck entwickelt wurde, natürlich mittels eXtreme Programming!

Nicht-selbstprüfende Tests sind nicht allzu hilfreich. Dies sind Tests, die zwar Testfälle einlesen und zu testende Methoden aufrufen, die aber die Ergebnisse der Methodenaufrufe einfach auf der Konsole ausgeben, und es dann dem menschlichen Tester überlassen, die Ergebnisse mit den erwarteten Ergebnissen manuell abzugleichen. Eine zeitraubende Angelegenheit.

3.6.3. Wann soll man Refactoring machen?

Nach [Fowler00] ist es nicht angebracht, für Refactoring eine bestimmte Zeit im Projektplan zur Verfügung zu stellen (z.B. eine Woche alle 2 Monate), sondern Refactoring sollte fortlaufend gemacht werden. Man soll Refactoring nicht um den Selbstwillen tun, sondern weil man etwas anderes tun will und Refactoring einem hilft, das andere zu tun.

Die üblichen Aktivitäten, bei denen einem ein Refactoring helfen kann, sind:

- *Hinzufügen einer neuen Funktionallität*
  Bevor man etwas am Code ändern kann, muss man ihn verstehen. Mit Refactoring kann man den Code besser verständlich machen.
  Desweiteren kann man auf Code stossen, der es einem nicht einfach macht, eine neue Funktionalität hinzufügen, und man macht zuerst ein Refactoring. Anschliessend geht das Hinzufügen der neuen Funktionalität dann einfacher und schneller.
- *Fehlerbehebung (bug fixing)*
  Auch hier wird Refactoring vor allem dazu gebraucht den Code verständlicher zu machen. Aktive Auseinandersetzung mit dem Code hilft, den Fehler zu finden.
- *Code Review*
  Nach Fowler sollten Code Reviews nur in 2er-Gruppen erfolgen, bestehend aus dem Autor selbst und einem Reviewer.
  Ohne Refactoring kann der Reviewer den Code lesen, bis zu einem gewissen Grad verstehen, und Vorschläge machen.
  Mit Refactoring kann sich der Reviewer, wenn er eine gute Idee hat, gleich überlegen ob diese Idee auch einfach implementiert werden kann. Der Reviewer macht Vorschläge, und beide zusammen entscheiden, ob diese einfach mittels Refactoring implementiert werden könnten. Wenn ja, wird das Refactoring gemacht. So muss sich der Reviewer nicht nur *vorstellen*, wie der Code mit seiner vorgeschlagenen Änderung aussehen würde, sondern er kann gleich *sehen* wie er aussieht.
  Diese Idee weitergedacht führt dann zu XP's Pair Programming, das eigentlich ein fortlaufendes, in den Entwicklungsprozess eingebettetes Code Review ist.

3.6.4. Ändern, wenn es stinkt

Etwas vom wichtigsten, und auch etwas vom schwierigsten, ist es zu wissen, wann und wo man genau ein Refactoring machen soll, wo also das Desiign so schlecht ist, dass es nach einem

Refactoring schreit. Die Kenntnis von den Refactoring-Mustern allein hilft einem dabei nicht gross. Die kommen erst zum Einsatz, nachdem man erkannt hat, dass schlechtes Design vorliegt.

Gute Entwickler verstehen natürlich etwas von gutem Design und "unästhetisches" Design sticht ihnen sofort ins Auge. Der Begriff "Ästhetik" war Fowler im Zusammenhang mit Design allerdings doch etwas zu schwammig als er sein Buch über Refactoring [Fowler00] schreiben wollte und er suchte nach handfesteren Kriterien, mit denen man schlechtes Design im Code erkennen sollte. So setzte er sich mit dem frisch Vater gewordenen Kent Beck in Zürich zusammen und die beiden entwickelten das Konzept der "Bad Smells", also schlechter Gerüche, nach einer Analogie zum Wechseln von Windeln bei Kleinkindern (*"If it stinks, change it. - Grandma Beck, discussing child-rearing philosophy"* [Fowler00])

Die beiden entwickelten dann einen Katalog von Code-Strukturen (eben dieser "Bad Smells"), die Anzeichen dafür sein können, dass ein Refactoring durchgeführt werden sollte. Dabei verzichteten sie bestimmt auf die Angabe von exakten Metriken (z.B. wie viele Linien Code in einer Methode zu viel sind und die Methode damit zu lang ist) sondern setzen auf die menschliche Intuition. Die "Bad Smells" sind nur Indikatoren, und wenn es irgendwo stinkt, dann muss man selbst entscheiden ob es genug stinkt, um ein Refactoring durchzuführen.

Beispiele von "Bad Smells": [Fowler00]

- *Duplizierter Code*
  Duplizierter Code ist die absolute Nummer 1 auf Fowlers "stink parade". Sollte die gleiche Code-Struktur an mehr als nur einem Ort im Code auftauchen, dann kann man sich sicher sein, dass man besser designten Code erhält, wenn man es irgendwie schafft, diese Code-Struktur an einem einzigen Ort zu vereinen.
  Wenn diese Fragmente in zwei verschiedenen Methoden derselben Klasse auftauchen, dann sollte man sie in eine neue, einzige Methode extrahieren, die dann von den bisherigen Orten aufgerufen wird.
  Wenn die Fragmente in zwei Subklassen auftauchen, dann kann man sie in eine neue Methode extrahieren, und diese dann in die Superklasse "hinaufschieben".
  Natürlich sind dies die zwei einfachsten Fälle von dupliziertem Code, aber Fowlers Katalog der Refactoring-Muster birgt auch für schwierigere Fälle anspruchsvollere Lösungen.
- *Lange Methoden*
  Die objekt-orientierte Programme, die am längsten leben, haben die kürzesten Methoden. Programmierer, die erstmals objekt-orientiert programmieren, haben das Gefühl, dass niemals etwas wirklich berechnet wird, sondern dass objekt-orientierte Programme eine endlose Folge von Delegationen darstellen. Aber diese Umwege machen sich bezahlt, und kurze Methoden sind meist selbsterklärend und können von verschiedenen Aufrufern geteilt werden.
  Fowler benutzt eine einfache Heuristik, wann eine Methode zu lang ist, und ein Teil von ihr vielleicht in eine neue, eigene Methode extrahiert werden sollte: Sobald man das Gefühl hat, etwas in einer Methode kommentieren zu müssen, wird stattdessen der zu kommentierende Code extrahiert, und die Methode wird nach ihrem Zweck benannt. Der Aufruf der neuen Methode in der alten Methode erspart dann das kommentieren, da der Namen der neuen Methode selbsterklärend ist. Sobald man kommentieren muss, genügt wohl der Name der Methode nicht mehr zur Erklärung was sie tut, und es gibt eine semantische Differenz zwischen dem, was die Methode tut und wie sie es tut.
  Um eine Methode dann zu kürzen gibt es viele Techniken, vom einfachen extrahieren einer

Methode in der gleichen Klasse, bis zum Ersetzen einer Methode mit einem "Methoden-Objekt".

- *Grosse Klassen*
  Wenn eine Klasse versucht zu viele Funktionen abzudecken, dann hat sie oft zu viele Instanzvariablen. Und vielleicht brauchen Instanzen der Klasse jeweils nur einen Teil der Instanzvariablen. Das Design könnte verbessert werden indem eine neue Klasse oder eine neue Subklasse aus der zu grossen Klasse extrahiert wird.
- *Lange Parameterlisten*
  Lange Parameterlisten sind gut, denn wenn wir einer Methode nicht alle Parameter übergeben, die sie braucht, dann gebraucht sie globale Daten, und globale Daten sind schlecht (Koppelung) und der Gebrauch von globalen Daten kann zuweilen schmerzhaft enden.
  Aber lange Parameterlisten sind auch schwer verständlich: 7 Integer-Werte in einer Reihe, wer kann da beim Aufrufen der Methode noch die richtige Reihenfolge einhalten? Zum Glück müssen wir bei einer objekt-orientierten Sprache einer Methode nicht alle Parameter als primitive Typen übergeben. Wir könnten ihr auch ein Objekt übergeben, und die Methode kann dann dieses Objekt nach den einzelnen Werten zur Berechnung fragen.
- *Switch Statements*
  Switch Statements sollten in objekt-orientierten Programmen eine rare Erscheinung sein. Oft findet man das gleiche Switch Statement an verschiedenen Stellen im Programm (→ Duplikation) und wenn eine neue Klausel bei einem Switch Statement hinzugefügt werden soll, dann muss man alle Stellen wo das Switch Statement auftritt suchen und dies an allen Stellen ändern. Ein mühsames Unterfangen. Häufig lassen sich solche Switch Statements mit der Einführung von Polymorphismus zum Verschwinden bringen, wobei dieses Unterfangen aber nicht immer trivial ist: Zuerst müssen die Switch Statements in eigene, neue Methoden extrahiert werden, diese anschliessend in die richtige Klasse geschoben werden, wo sie hingehören. Schliesslich muss dann entschieden werden, ob die Typ-Codes, auf denen das Switch-Statement operiert, direkt durch Subklassen ersetzt werden können, oder ob man ein State- oder Strategy-Pattern [Gamma94] verwenden soll.

Insgesamt haben Fowler und Kent in [Fowler00] 22 solcher "Bad Smells" aufgeführt, natürlich eine nicht abschliessende Liste. Die obigen 5 Beispiele sind willkürlich gewählt.

3.6.5. Refactorings – der Muster-Katalog

In [Fowler00] werden über 70 einzelne "Refactorings" in einem Muster-Katalog beschrieben. Alle "Refactorings" überführen den ursprünglichen Code in verhaltenstechnisch äquivalenten Code, der aber anders strukturiert ist. Ein solches "Refactoring" ist entweder ein ganz kleiner Schritt, der das Design des Codes noch nicht gross verbessert, oder es ist eine Aneinanderreihung mehrer kleiner "Refactorings" in einer bestimmten Reihenfolge.

Jedes der "Refactorings" wird im Katalog in einem Standard-Format beschrieben durch:

- Name (*name*): Ein Bezeichner, mit dessen Hilfe das Refactoring referenziert werden kann
- Zusammenfassung (*summary*): Die Zusammenfassung beschreibt die Situation, in der man das Refactoring anwenden kann und fasst zusammen, was das Refactoring tut.
- Motivation (*motivation*): Die Motivation beschreibt, warum das Refactoring gemacht werden soll und unter welchen Umständen darauf verzichtet werden sollte.

– Vorgehensweise (*mechanics*): Eine kurze, prägnante, Schritt-für-Schritt Anleitung.

– Beispiel (*example)*: Einfaches Beispiel an einem Stück Code, das zeigt wie es geht.

Ein Beispiel eines solchen "Refactorings" zeigt [Anhang1].

3.7. In welchen Fällen soll man kein Refactoring machen?

Plötzlich erscheint einem Refactoring als Allerwelts-Heilmittel, und man hat das Gefühl, jedes schlechte Design könnte durch Refactoring wieder auf Vordermann gebracht werde. Es gibt aber durchaus Situationen, wo Refactoring nicht angewendet werden sollte:

- *Wenn der Code so schlecht ist, dass es schneller gehen würde, nochmals ganz von vorne anzufangen*. Ein klares Anzeichen dafür ist, wenn der Code nicht funktioniert. Wenn man ihn testet und merkt, dass er voll von Fehlern ist und man es nicht schafft, ihn zu stabilisieren. Um ein Refactoring zu machen, muss man Code haben, der die funktionalen Anforderungen erfüllt, der aber schlecht strukturiert ist.
- *Wenn man kurz vor einem Abgabetermin steht*: Der Produktivitätsgewinn durch Refactoring würde sich erst nach dem Abgabetermin zeigen, und damit zu spät

4. Simple Design und andere Aspekte von Design in XP

Die Hauptdesignstrategie von XP, die natürlich immer wieder Kritiker auf den Plan ruft und ellenlange Diskussionen auf entsprechenden Foren verursacht, ist die Praktik vom Simple Design. Ihre Schlagworte oder besser -sätze sind "Do the Simplest Thing that Could Possibly Work" und "You Aren't Going to Need It", auch bekannt als YAGNI.

Simple Design sagt uns, dass wir keinen Code schreiben sollen, der erst von einer Funktionalität benutzt werden wird, die wir in Zukunft implementieren werden müssen.

Eigentlich tönt das recht einfach und einleuchtend. Doch was ist denn mit all den guten Sachen wie Frameworks, wiederverwendbaren Komponenten, flexiblem Design, Entwurfsmustern? In klassischen Prozessen haben wir doch schon längst herausgefunden, dass wir das Rad nicht jedesmal neu erfinden müssen, und auf bewährte Muster zurückgreifen können. Das Ganze ist aber nicht ganz gratis zu haben, denn solche Sachen sind meist schwierig zu erstellen. Zu Beginn muss man Zeit und Geld investieren, in der Hoffnung, dass es sich später auszahlen wird.

XP hingegen verlangt ausdrücklich, dass man keine flexible Komponenten und Frameworks gebraucht, wenn es auch einfacher geht. Sollte man sie später doch brauchen, dann wächst man quasi automatisch rein mittels Refactoring.

4.1. Die Motivation von Simple Design

Wo steckt die Motivation für eine solche Philosophie?

Ein Grund ist ökonomischer Natur: Wenn ich Arbeit in etwas hineinstecke, das erst in der nächsten Iteration gebraucht werden wird, dann habe ich weniger Zeit für Sachen, die in dieser Iteration

erledigt werden müssen. Und dieses Verhalten erhöht das Risiko, dass man die Sachen für diese Iteration nicht rechtzeitig fertig stellen kann.
Noch schlimmer als an etwas nur zu früh zu arbeiten ist es, es auch noch falsch zu machen. Und die Chancen es falsch zu machen sind umso höher je früher man damit beginnt, weil die Anforderungen zu einem solch frühen Zeitpunkt noch gar nicht genug bekannt sind oder weil sie sich in Zukunft noch ändern werden

Ein zweiter Grund ist, dass ein komplexeres Design schwerer zu verstehen ist als ein einfaches Design. Die zusätzliche Komplexität macht Modifikation am System darum schwieriger und verursacht zusätzliche Kosten in der Periode zwischen dem Zeitpunkt, wo das komplexere Design implementiert wurde und dem Zeitpunkt, ab dem man es wirklich gebraucht hätte.

Viele empfinden obige Aussagen als Nonsens und Fowler gibt sogar zu, dass sie recht haben. [Fowler01]. Aber nur wenn die Praktiken, die XP erst ermöglichen (*enabling practices*), nicht angewendet werden. Werden sie aber angewendet, und verläuft die Kostenkurve dadurch praktisch horizontal, dann macht Simple Design Sinn.

4.2. Verletzt Refactoring YAGNI?

Refactoring braucht Zeit (kostet also) und doch werden dabei keine neuen Funktionalitäten implementiert. Man macht also etwas, was einem vielleicht erst morgen etwas bringen wird (besseres Design macht das Implementieren neuer Funktionalität in Zukunft einfacher). Verletzt die Refactoring-Praktik also YAGNI?

Solche Fragen tauchen von Zeit zu Zeit in XP-Mailing-Listen auf, können aber einfach beantwortet werden: YAGNI bedeutet, dass man nicht ein komplexeres Design verwendet, als dass es nötig wäre um die aktuelle “Story” zu implementieren. Und Refactoring braucht man, um das Design so einfach wie möglich zu halten. Darum soll man immer ein Refactoring machen, sobald man erkennt, dass man es auch einfacher gestalten könnte.

Simple Design ist eine Praktik, die XP sowohl ausnützt als auch erst ermöglicht. Nur mit Testen, kontinuierlicher Integration und mit Refactoring kann Simple Design wirksam angewendet werden. Aber gleichzeitig ist Simple Design nötig, um die Kostenkurve flach zu halten. Jegwelche unnötige Komplexität macht es schwieriger, das System in irgendeine Richtung zu ändern, ausser in die Richtung, auf die man vorgreift (*to anticipate*) mit der zusätzlichen komplexen Flexibilität. Aber da wir Menschen gar nicht gut im vorhersehen (*to anticipate*) sind, ist es besser, wenn wir Einfachheit anstreben. Und da wir Menschen die einfachste Lösung nicht beim ersten Mal finden, müssen wir Refactoring machen damit wir näher an unser Ziel kommen.[Fowler01]

4.3. Architektur in XP

In XP wird möglichst auf ein Software-Architektur Dokument zu Beginn der Entwicklung verzichtet. Aber gibt es nicht Sachen, die einfach wahnsinnig schwer zu ändern oder hinzuzufügen sind am Schluss? Ist es nicht erlaubt, sich zu Beginn zu überlegen, wie man das ganze System im Groben aufbauen will?

XP sagt nein. Keine Datenbank, solange man sie nicht braucht und es auch mit Dateien geht. Braucht man sie nachher doch, macht man ein Refactoring.

Fowler bringt in [Fowler01] auch die Geschichte von dem Projekt, in dem ein System mit EJB (Enterprise Java Beans) gestaltet wurde, und kurz vor der Produkteinführung entschieden wurde, dass sie eine solch komplexes Framework eigentlich gar nicht brauchen und es mittels Refactoring wieder entfernt haben. Es war ein ziemlich grosses Refactoring und es wurde spät durchgeführt, aber es hatte geklappt. Doch dann stellt sich die Frage, ob es genauso schmerzlos gehen würde, wenn man ein System ohne EJB entwickelt und sehr spät merkt, dass man es eigentlich doch braucht. Wenn man für Sicherheit, Transaktionsmanagement, Persistenz und des weiteren schon selbst eine Lösung gebastelt hat und dann erkennt dass alles einfacher ginge mit EJB? Ist es dann genauso einfach, EJB hinzuzufügen wie EJB zu entfernen?

Fowler widerspricht daher dem XP-Ansatz und plädiert für eine Grob-Architektur bei Projektstart. Um XP aber gerecht zu werden, soll sie nicht wie bei klassischen Methoden in Stein gemeisselt sein, sondern darf natürlich geändert werden, sobald man beim Programmieren erkennt, dass etwas überflüssig ist, oder es auch einfacher ginge.

4.4. Wer macht denn nun das Design in XP?

Design "entsteht" in XP während dem Implementieren durch Refactoring. Jeder Programmierer (oder besser: jedes Programmierer-Pärchen) trägt seinen Teil dazu bei.

Und wo bleiben die Software-Architekten, die Leute die in klassischen Projekten für das Design zuständig waren? Werden die aus dem Haus gejagt?

Für "Architekten", die sich für etwas besseres als der "einfache" Programmierer halten, und sich zu schade sind jemals wieder eine Zeile Code zu schreiben, weil sie über dieses Stadium hinaus sind, hat es in XP wahrlich keinen Platz.

Aber für erfahrene Entwickler, die bereit sind an das Simple Design zu glauben, und weiterhin Einfluss nehmen wollen und das Design entscheidend mitgestalten, hat XP eine Rolle vorgesehen, die des "Coach".

XP braucht erfahrene, gute Entwickler, die auch etwas von Design verstehen, und die voneinander lernen und zusätzlich braucht es einen noch erfahreneren und noch besseren Entwickler als Coach.

4.5. Patterneinsatz in XP

Oftmals entsteht nach Fowler [Fowler01] der Eindruck, dass XP davon abrät Entwurfsmuster überhaupt zu benutzen. Und dies, obschon die meisten XP-Verfechter auch Vorreiter in der "Pattern-Bewegung" waren. Warum ist das so? Weil sie über die Entwurfsmuster hinaus blicken können, oder weil die Muster schon so in ihr Denken eingebettet sind, dass sie es gar nicht mehr merken? Wer weiss.

XP ist ein Software-Entwicklungs-Prozess, aber Entwurfsmuster sind ein Rückgrat der Design-Kenntnisse. Und diese Kenntnisse sind wertvoll, ob für klassische Software-Entwicklungs-Prozesse oder für einen agilen Prozess. [Fowler01]

Und XP sagt nur:

– Gebrauche kein Entwurfsmuster bis Du es benötigst.

– Finde Deinen Weg in ein Entwurfsmuster über eine einfache Implementation

Eine Theorie, wieso Entwurfsmuster in XP nicht explizit empfohlen werden, ist, dass die Kräfte von Simple Design einen automatisch zu den Entwurfsmustern führen. Einige Refactorings tun das sogar explizit, aber sogar ohne diese Refactorings ergeben sich die Muster wie von allein, wenn man die Regeln von Simple Design anwendet, sogar wenn man die Muster gar nicht kennt.

Besser wäre, wenn man schon im Voraus eine gewisse Ahnung von den gängigsten Entwurfsmustern, wie sie z.B. in der Entwurfsmuster-Bibel [Gamma94] beschrieben sind, hätte und dann, wenn einem Simple Design dazu hinführt, sie erkennen und nachschlagen könnte, anstatt das Rad wieder neu zu erfinden.

5. Fazit und Ausblick

Design ist nicht tot bei den agilen Methoden (oder zumindest nicht bei XP, das ich etwas näher untersucht habe). Design hat ein neues Verständnis gefunden.

Tot wäre es, wenn man das herkömmliche “geplante Design” über den Haufen geworfen und sich keine Gedanken mehr gemacht hätte. Stattdessen wurden disziplinierte Techniken erfunden (oder gefunden), die es ermöglichen, das Design des Systems so spät wie möglich oder es gar nicht festzulegen, sondern hineinzuwachsen. Damit will man ein gutes System-Design nicht entwerten; im Gegenteil, ein gutes Design ist überaus wichtig, denn ohne gutes Design wären späte Änderungen ein Greuel.

Während man bei XP-Praktiken wie “Simple Design” durchaus anderer Meinung sein kann und sich auf jeden Fall fragen darf inwieweit XP die Kostenkurve abflachen kann, und wie sie bei XP genau aussieht, haben uns die agilen Methoden auf jeden Fall ein mächtiges Werkzeug gebracht, das einem erlaubt das Design eines Stücks Software im Nachhinein zu ändern, ohne das Verhalten zu verändern: Refactoring. Diese Technik, diszipliniert angewendet, wird sicher in der Zukunft bei Wartungsprozessen von objekt-orientierter Software eine entscheidende Rolle spielen, gleich ob sie ursprünglich mit einer agilen Methode oder mit einem klassischen Entwicklungsprozess entwickelt wurde.

Literatur

[Fowler00] Fowler, M., Refactoring: Improving the Design of Existing Code, 2000, Addison-Wesley

[Fowler 01] Fowler, M., Is Design Dead?, 2001, http://martinfowler.com/articles/designDead.html

[Gamma94] Gamma, E. et alteri, Design Patterns: Elements of Reusable Object-Oriented Software, 1994, Addison-Wesley