DOKUMENTATION

zum Projekt

REFACTORING – am Beispiel eines PERT Chart

Markus Barth

Inhaltsverzeichnis

1. Inhalt des Projekts

Das Design einer Software zerfällt mit der Zeit. Häufige Änderungen ohne komplettes Verständnis des Codedesigns führen dazu, dass der Quellcode seine Struktur einbüsst. Der Aufbau des Programms erlaubt es nicht mehr, dieses auf einfache Art und Weise zu ändern.

Als Refactoring wird nun der Prozess bezeichnet, ein Softwaresystem so zu verändern, dass das externe Verhalten bestehen bleibt, der Quellcode aber eine bessere interne Struktur erhält. Die Software wird dadurch wieder verständlicher und somit leichter zu modifizieren.

Innerhalb dieses Projekts soll ein Refactoring an einem PERT Chart-Framework durchgeführt werden. Das PERT Chart ist Teil einer Fertigungssoftware für die Prozessplanung. Es bietet eine graphische Sicht auf Prozesse, um hierbei die Sequenz der einzelnen Prozesse zu planen, zu organisieren oder zu koordinieren.

2. Hintergrund / Umfeldbeschreibung

Refactoring ist ein diszipliniertes Vorgehen, um Quellcode zu bereinigen, so dass die Wahrscheinlichkeit, Fehler einzuführen, minimiert wird. Im Kern wird das Design verbessert, nachdem Quellcode geschrieben wurde.

Mit dem heutigen Verständnis von Softwareentwicklung wird angenommen, dass zuerst entworfen und dann programmiert wird. Erst kommt ein gutes Design und dann die Programmierung. Im Laufe der Zeit wird der Code jedoch verändert, und die Integrität des Systems, seine entwurfsmässige Struktur, schwindet. Die Qualität des Quellcodes von dem ursprünglichen Ingenieurs-Niveau sinkt immer weiter.

Refactoring ist genau das Gegenteil dieser Gepflogenheit. Mittels Refactoring kann sogar mit einem schlechten Design im Chaos begonnen und dieses dann zu gut strukturiertem Code umgearbeitet werden. Refactoring besteht aus vielen einfachen Schritten, wobei mit jedem der Zustand oder das Verhalten von Klassen verschoben wird. Das kumulierte Ergebnis dieser kleinen Änderungen soll dann das Design gründlich verbessern.

Die Methoden des Refactorings werden auf ein PERT Chart-Framework angewandt. Dieses Framework ist nach dem Model-View-Controller Prinzip aufgebaut. Der Controller enthält hierbei eine Klasse namens PertController, an welcher dann konkrete Beispiele für Refactoring-Methoden aufgezeigt werden. Geschrieben wurde die Software in C++ und somit werden auch die Beispiele in dieser Sprache aufgeführt.

Die Klasse PertController eignet sich für dieses Projekt, da sie genau die zuvor schon beschriebenen Eigenschaften aufweist. Das Design und damit die entwurfsmässige Struktur zerfällt. Hierfür sind Änderungen, um kurzfristige Ziele zu erreichen, oder Änderungen, ohne komplettes Verständnis des Codedesigns verantwortlich. Des Weiteren sind über die Jahre sehr viele Abhängigkeiten von Codezeilen entstanden, weshalb es sehr mühsam ist, das Framework oder speziell die PertController-Klasse zu modifizieren. Die Klasse verfügt über 4150 Zeilen Quellcode und über 61 Methoden. Dies allein macht sie schon äußerst unübersichtlich und schwer verständlich.

3. Zielsetzung

Refactoring ist ein Werkzeug, das für verschiedene Aufgaben eingesetzt werden kann. Zum einen soll das Refactoring das Design der Software verbessern. Aufgrund vieler getätigter Änderungen büßt der Code seine Struktur ein. Es wird schwieriger das Design zu erkennen, indem man den Quellcode liest. Und je schwieriger es ist, das Design des Codes zu verstehen, umso schwieriger ist es, dieses zu erhalten und umso schneller zerfällt es.

Weiter soll das Refactoring die Software leichter verständlich machen. Dies ist vor allem dann von Bedeutung, wenn über kurz oder lang andere Nutzer den Sourcecode modifizieren werden.

Zudem hilft es, Fehler in der Software zu finden und Redundanzen aufzulösen, da sich ein Entwickler während des Refactoring-Prozesses ein tiefgründiges Verständnis des Quellcodes erarbeitet und somit die Struktur besser versteht.

Letztendlich laufen alle genannten Punkte auf ein und dasselbe Ziel hinaus: Aufgrund des Refactorings soll es dem Entwickler ermöglicht werden, schneller und effektiver zu programmieren. In diesem speziellen Fall gilt dies für das PERT Chart-Framework bzw. die PertController-Klasse.

Ohne ein gutes Design können kurzfristig schnelle Fortschritte erzielt werden, bald aber wird das schlechte Design die weitere Entwicklung ausbremsen. Änderungen nehmen immer mehr Zeit in Anspruch, während der Entwickler damit beschäftigt ist, das System zu verstehen und den duplizierten Quellcode zu finden. Ein gutes Design ist entscheidend, um das Tempo der Softwareentwicklung aufrecht zu erhalten. Refactoring soll hier also helfen, Software schneller zu entwickeln, da der Verfall des Designs und somit des Systems gestoppt wird.

4. Anforderungen an die Dokumentation

Die Projektdokumentation wird separiert in eine Dokumentation der Projektabwicklung und in eine Dokumentation der Projektergebnisse.

Die Dokumentation der Projektabwicklung soll nochmals Aufschluss über den Ablauf des Projekts geben und auch auf etwaige Schwierigkeiten im Prozess des Refactorings hinweisen.

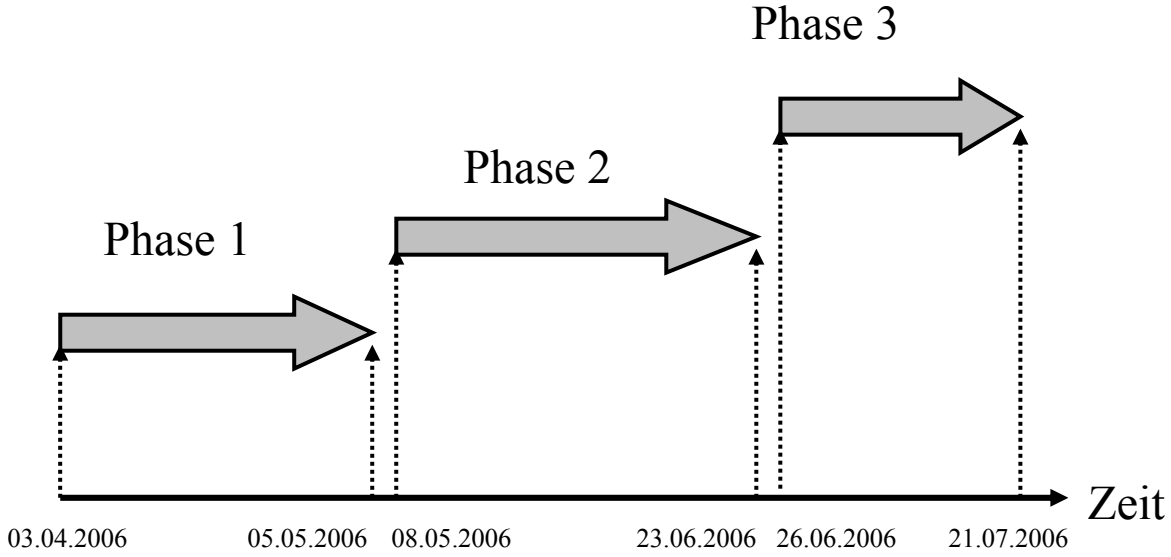
Die Dokumentation der Projektergebnisse erläutert wichtige Refactoring-Methodiken aus den Hauptprodukten der Phase 1, gibt Aufschluss über automatisierte Softwaretests und zeigt die implementierten Refactoring-Methodiken im PERT Chart-Framework bzw. in der PERTController-Klasse. Hierzu werden beispielhafte Auszüge aus dem Quellcode mit aufgeführt und kommentiert.

5. Dokumentation der Projektabwicklung

5.1 Gesamtprojektplanung und Ablauf des Projekts

Die generelle Projektgesamtaufgabe wurde in verschiedene Phasen unterteilt welche wiederum aus verschiedenen Teilaufgaben bestehen. Diese Teilaufgaben sind eigene, separate Teilprojekte und mussten nacheinander bzw. parallel durchgeführt und eingesetzt werden. Hieraus ergab sich folgender Phasenplan:

- **Phase 1: Methodiken des Refactorings erarbeiten**
  Start: 03.04.2006 Ende: 05.05.2006
- **Phase 2: Refactoring des PERT Charts**
  Start: 08.05.2006 Ende: 23.06.2006
- **Phase 3: Dokumentation**
  Start: 26.06.2006 Ende: 21.07.2006



Zuerst galt es die Methodiken des Refactorings zu erarbeiten, also eine Art Katalog mit Refactoring-Methoden anzulegen. Danach wurden diese dann Schritt für Schritt auf das PERT Chart-Framework bzw. auf die PertController-Klasse angewendet, um nach und nach das Design zu verbessern.

Bevor die erarbeiteten Refactoring-Methoden angewandt werden konnten, war es wichtig, eine Reihe von Testfällen zu erstellen. Diese sollten vollständig automatisiert und selbstüberprüfend arbeiten, um die Zeit für eine mögliche Fehlersuche zu reduzieren und um sofort festzustellen, wann Schnittstellen nach außen hin geändert wurden. Hierbei musste der zu refaktorierende Quellcode berücksichtigt werden.

5.2 Hauptprodukte

Das Projekt lässt sich in folgende Phasen und Hauptprodukte (Meilensteine) untergliedern:

Phase 1: Methodiken des Refactorings erarbeiten

- 1.1 Funktionen zusammenstellen
- 1.2 Eigenschaften zwischen Objekten verschieben
- 1.3 Daten organisieren
- 1.4 Bedingte Ausdrücke vereinfachen
- 1.5 Methodenaufrufe vereinfachen
- 1.6 Generalisierung
- 1.7 Große Refaktorierungen

Phase 2: Refactoring des PERT Charts

- 2.1 Tests aufbauen
- 2.2 Refactoring-Methodiken implementieren
- 2.3 Verbesserung des Designs

Phase 3: Dokumentation

- 3.1 schriftliche Ausarbeitung

5.3 Schwierigkeiten im Projekt

Das Refactoring ist eine neue Technik, welche jedoch die Produktivität stark erhöhen kann. Dennoch ist es nicht ganz einfach zu erkennen, wann der Einsatz weniger effektiv oder gar schädlich für das System werden kann. Vor 10 Jahren dürfte dies genauso bei Objekten der Fall gewesen sein. Man kannte sehr wohl die Vorteile der Objektorientierung, wusste aber noch nicht, wo die Grenzen lagen. Es gibt also auch beim Refactoring heutzutage noch keine hinreichenden Erfahrungswerte, um festzustellen, wo die Grenzen liegen.

Ein weiteres Problem liegt in der Änderung von Schnittstellen. Wenn Zugriff auf den ganzen Quellcode besteht, ist es kein Problem, z. B. den Namen einer Methode zu ändern. Es gibt nur dann ein Problem, wenn die Schnittstellen von Quellcode benutzt werden, der nicht zu finden ist oder auf den kein Zugriff besteht und somit nicht geändert werden kann. Wenn sich also etwas an einer Schnittstelle ändert, sollte auch der veraltete Quellcode beibehalten werden, zumindest so lange, bis alle Anwender die Chance gehabt haben, die Änderung zu berücksichtigen. Glücklicherweise lässt sich eine veraltete Schnittstelle auch immer noch weiterverwenden, indem die alte Schnittstelle die neue verwendet. Zudem sollte dann auch die Möglichkeit in Betracht gezogen werden, eine Methode als ’deprecated’ (veraltet) zu kennzeichnen.

Einige Änderungen konnten auch nicht durchgeführt werden, da diese zugleich Auswirkungen auf andere Frameworks hatten, und somit vom Framework-Zuständigen noch zusätzliche Arbeit nötig gewesen wäre, um die Änderungen mit ein zu pflegen. Da jedoch diese zusätzliche Arbeit im aktuellen Release nicht mehr getätigt werden konnte, mussten die Änderungen wieder zurückgenommen werden.

Oberste Priorität hatte jedoch, dass das Refactoring die Funktionalität des PERT Charts in keinster Weise beeinflussen oder gar verändern durfte. Schnittstellen nach außen hin durften nicht verändert werden. Am Anfang galt es also zuerst einmal sicherzustellen, dass in den zu automatisierenden Testreihen, der komplette, zu refaktorierende Quellcode mit berücksichtigt wurde, um dann bei etwaigen Problemen sofort reagieren zu können. Dieser Schritt enthält zugleich einen nicht zu unterschätzenden Zeitaufwand, hat dann aber im späteren Verlauf den Vorteil, dass ein weitaus sicheres Refactoring möglich ist und nicht zu viel Zeit bei der Fehlersuche verschwendet wird.

6. Dokumentation der Projektergebnisse

6.1 Bedeutende Refactoring-Methodiken

Im Folgenden werden wichtige, erarbeitete Refactoring-Methodiken, sowie Methodiken, die auf das PERT Chart Framework angewandt wurden, benannt und in Kürze beschrieben. Hierbei handelt es sich um Refaktorierungen, die den in 5.2 aufgelisteten Hauptprodukten der Phase 1 zugeordnet wurden. Diese Dokumentation beschränkt sich auf die wesentlichen Refactoring-Methodiken, da der Katalog an Refaktorierungen sehr umfangreich ist.

6.1.1 Funktionen zusammenstellen

Ein großer Teil von Refaktorierungen stellt Funktionen zusammen, um den Quellcode ordentlich zu strukturieren. Die Probleme rühren daher, dass viele Methoden zu lang sind und sie zu viele Informationen enthalten, die unter der komplexen Logik begraben liegen. Die Schlüsselrefaktorierung hierfür ist *'Methode extrahieren'*. Diese nimmt ein Stück Code, um daraus eine eigene Methode zu machen. Der Name der Methode soll dabei zugleich die Aufgabe erklären. Ferner kann Quellcode, der eines Kommentars bedarf, um seine Aufgabe zu verstehen oder redundanter Quellcode mit *'Methode extrahieren'* aufgelöst werden.

'Methode integrieren' ist im Wesentlichen das Gegenteil davon. Ein Methodenaufruf wird durch den Rumpf der aufgerufenen Methode ersetzt. Wenn zu oft extrahiert wurde und einige der so entstandenen Methoden zu wenig tun oder wenn die Zerlegung in Methoden reorganisiert werden muss, kommt diese Refaktorierung zum Einsatz.

Das größte Problem bei *'Methode extrahieren'* ist der Umgang mit lokalen Variablen, und temporäre Variablen. Bei einer Methode sollte *'Temporäre Variable durch Abfrage ersetzen'* verwendet werden, um alle temporären Variablen los zu werden, die entfernt werden können. Wird eine temporäre Variable für viele Dinge verwendet, so sollte zunächst *'temporäre Variable zerlegen'* eingesetzt werden, um die temporären Variablen leichter ersetzbar zu machen.

Manchmal sind die temporären Variablen aber einfach zu verheddert, um sie zu ersetzen. Dann wird *'Methode durch Methodenobjekt ersetzen'* verwendet. Diese Refaktorierung ermöglicht, auch die verworrenste Funktion zu zerlegen, allerdings auf Kosten einer neuen Klasse.

6.1.2 Eigenschaften zwischen Objekten verschieben

Eine der wichtigsten, wenn nicht die wichtigste Entscheidung im Objektdesign, betrifft die Verteilung der Verantwortlichkeiten. Es ist schwierig diese gleich zu Beginn eines Softwareprojekts korrekt auf zu teilen. Oft lassen sich solche Probleme einfach lösen, indem *'Methode verschieben'* und *'Feld verschieben'* eingesetzt werden, um dann Verhalten zu verschieben. Das Verschieben ist hierbei abhängig davon, mit welchem Objekt die Methode bzw. das Feld am meisten interagiert.

Oft werden Klassen mit zu vielen Verantwortlichkeiten überladen. Dann sollte *'Klasse extrahieren'* verwendet werden, um einige dieser Verantwortlichkeiten abzutrennen. Wenn also z. B. eine Klasse die Arbeit von zwei Klassen macht, sollten relevante Felder und Methoden verschoben werden. Eine Klasse sollte nämlich eine glasklare Abstraktion darstellen. In der Praxis jedoch wachsen Klassen, da ständig neue Operationen, Methoden oder Felder mit hinzu kommen. Die Klasse wird irgendwann zu groß und zu komplex, um leicht verständlich zu sein.

Behält danach eine Klasse zu wenig Verantwortung, so kann sie mit *'Klasse integrieren'* in einer andere Klasse mit aufgenommen werden.

Die Refaktorierungen *'Fremde Methode einführen'* sowie *'Lokale Erweiterung einführen'* sind Spezialfälle dieses Hauptprodukts. Sie werden verwendet, wenn kein Zugriff auf den Sourcecode einer Klasse besteht, es aber trotzdem Verantwortlichkeiten in diese nicht veränderbare Klasse zu verschieben gibt. Handelt es sich nur um eine oder zwei Methoden, so kann *'Fremde Methode einführen'* verwendet werden; geht es um mehrere Methoden, so sollte besser *'Lokale Erweiterung einführen'* verwendet werden, da hier eine neue Klasse mit zusätzlichen Methoden erstellt wird. Die neue Klasse sollte dann als Unterklasse oder Hüllklasse (Wrapper) der Originalklasse erstellt werden.

6.1.3 Daten organisieren

Dieses Hauptprodukt soll den Umgang mit Daten vereinfachen. Hierzu zählt 'eigenes Feld kapseln', indem z. B. ein öffentliches Feld privat deklariert wird und der Zugriff auf die Membervariable nur noch mit so genannten set- und get-Methoden erfolgt. Der indirekte Variablen-Zugriff hat den Vorteil, dass die Flexibilität bei der Verwaltung der Daten erhöht wird. Dies zeigt sich z. B. dann bei der Vererbung, beim Zugriff auf ein Feld einer Oberklasse.

Eine der nützlichen Eigenschaften objektorientierter Sprachen ist es, dass sie es ermöglichen, neue Typen zu definieren, die über das hinausgehen, was mit den einfachen Datentypen traditioneller Sprachen gemacht werden kann. Oft wird mit einem einfachen Datenwert begonnen und im Nachhinein wird erkannt, dass ein Objekt nützlicher wäre.

'Wert durch Objekt ersetzen' ermöglicht es, simple Daten in klar erkennbare Objekte zu verwandeln. Dies macht sich spätestens dann zu Nutzen, wenn ein Datenelement weitere Daten oder zusätzliches Verhalten benötigt. Wenn dann das Objekt eine Instanz ist, welche in vielen Teilen des Programms benötigt wird, kann daraus durch den Einsatz von *'Wert durch Referenz ersetzen'* ein Referenzobjekt gemacht werden.

Links zwischen Objekten können in eine Richtung (unidirektional) und in beide Richtungen (bidirektional) benutzbar sein. Links in nur eine Richtung sind einfacher, aber manchmal wird *'gerichtete Assoziation durch bidirektionale ersetzen'* benötigt, um eine neue Funktion zu unterstützen. Es existieren also zwei Klassen, die Elemente der jeweils anderen benötigen. Nun gibt es aber nur eine Verbindung in eine Richtung. Hier gilt es also Rückverweise einzufügen und beide Verweise in den set-Funktionen zu aktualisieren.

'Bidirektionale Assoziation durch gerichtete ersetzen' entfernt unnötige Komplexität, wenn festgestellt wird, dass kein Link in beide Richtungen mehr benötigt wird.

6.1.4 Bedingte Ausdrücke vereinfachen

Die Kernrefaktorierung ist hier *'Bedingung zerlegen'*, die eine Bedingung in Teile zerlegt. Sie ist wichtig, da sie die Verzweigungslogik von den Details trennt, die ausgeführt werden. In einer Bedingung ist zwar oft ersichtlich was passiert, aber es wird leicht verdeckt, warum es passiert.

Die anderen Refaktorierungen dieses Hauptproduktes betreffen weitere wichtige Fälle. So kann *'redundante Bedingungsteile konsolidieren'* verwendet werden, um alle Redundanzen in Bedingungen zu vermeiden. Hier kommt das gleiche Codefragment in allen Zweigen eines bedingten Ausdrucks vor. Diesen Teil gilt es, aus den bedingten Ausdrücken herauszuziehen.

Wenn es um Code geht, der unter der Voraussetzung entwickelt wurde, dass eine Methode immer genau einen Ausgang haben muss, so werden häufig Steuerungsvariablen (oftmals boolesche Ausdrücke) verwendet, welche es ermöglichen, dass Bedingungen diese Regel einhalten. Wenn die Funktion zudem ein bedingtes Verhalten aufweist, welches den normalen Ablauf nicht leicht erkennen lässt, so können durch *'geschachtelte Bedingungen durch Wächterbedingungen ersetzen'* Spezialfälle klar herausgearbeitet werden. Mit *'Steuerungsvariable entfernen'* lässt sich dann noch die störende Variable beseitigen. Oftmals kann hier dann einfach nur 'break' oder 'return' verwendet werden.

6.1.5 Methodenaufrufe vereinfachen

Bei Objekten dreht sich alles um die Schnittstelle. Schnittstellen zu schaffen, die leicht zu verstehen und zu benutzen sind, ist eine Schlüsselqualifikation, um gute objektorientierte Software zu entwickeln. Dieses Hauptprodukt beschäftigt sich mit Refaktorierungen, um Schnittstellen zu vereinfachen.

Oft ist die einfachste und wichtigste Sache, die gemacht werden kann, den Namen einer Methode zu ändern. Namensgebung ist ein entscheidendes Mittel zur Kommunikation. Wenn verstanden wurde was ein Programm macht, sollte *'Methode umbenennen'* eingesetzt werden, um dieses Wissen weiter zu geben. Es können auch Variablen und Klassen umbenannt werden um die Absichten des Quellcodes deutlicher zu machen.

Parameter spielen ebenso eine wichtige Rolle in Schnittstellen. *'Parameter ergänzen'* und *'Parameter entfernen'* sind gängige Refaktorierungen. Programmierer, für die Objekte noch neu sind, verwenden oft lange Parameterlisten, da diese für andere Entwicklungsumgebungen typisch sind. Objekte dagegen ermöglichen es, Parameterlisten kurz zu halten. Und dies ist durchaus ein Vorteil, da jeder zusätzliche Parameter auch mehr Arbeit mit sich bringt. Es gilt also überflüssige oder gar nicht mehr verwendete Parameter zu entfernen.

Eine der nützlichsten Konventionen ist es, klar zwischen Methoden, die den Zustand ändern und solchen, die den Zustand abfragen, zu unterscheiden. Wenn dies vermischt wird, führt das nur zu Schwierigkeiten. Deshalb gilt es, die Refaktorierung *'Abfrage von Änderung trennen'* einzusetzen, um solche Aspekte zu entkoppeln. Eine Methode, die einen Wert zurückliefert, sollte nicht auch noch den Zustand des Objekts ändern.

Konstruktoren können ebenfalls ein störendes Element darstellen, da sie den Programmierer dazu zwingen, die Klasse eines Objekts zu kennen, das sie erzeugen. Oft muss man dies jedoch gar nicht wissen. Die Notwendigkeit, dies zu wissen besteht nicht mehr, wenn *'Konstruktor durch Fabrikmethode ersetzen'* angewendet wird. Bei der Erzeugung eines Objekts kann so mehr als nur eine einfache Konstruktion vorgenommen werden.

Ein weiterer Fluch ist die Typkonvertierung (casting). Es gilt zu vermeiden, dass die Anwender einer Klasse einen Downcast machen, wenn dieser auch woanders untergebracht werden kann, indem *'Downcast kapseln'* angewandt wird. Bei einer Methode, die ein Objekt liefert, auf das Clients einen Downcast anwenden müssen, sollte dieser besser in den Rumpf der Methode verschoben werden.

6.1.6 Generalisierung

Die meisten Refaktorierungen innerhalb dieses Hauptproduktes drehen sich um das Verschieben von Methoden in der Vererbungshierarchie. *'Feld nach oben verschieben'* und *'Methode nach oben verschieben'* befördern Funktionen die Hierarchie hinauf; *'Feld nach unten verschieben'* und *'Methode nach unten verschieben'* befördern sie nach unten. Hierbei ist zu beachten, dass gleiche Felder in verschiedenen Unterklassen oder Methoden mit identischen Ergebnissen in verschiedenen Unterklassen, besser in die Oberklasse verschoben werden sollten. Auf der anderen Seite sollte ein Feld einer Oberklasse, welches nur von wenigen Unterklassen verwendet wird, bzw. eine Methode, deren Verhalten in einer Oberklasse, nur für einige ihrer Unterklassen relevant ist, auch in die jeweiligen Unterklassen verschoben werden.

Es können jedoch nicht nur Methoden in der Hierarchie verschoben werden. Durch die Bildung neuer Klassen kann diese Hierarchie ebenfalls verändert werden. Hierfür sind *'Unterklasse extrahieren'*, *'Oberklasse extrahieren'* und *'Schnittstelle extrahieren'* verantwortlich, indem sie an verschiedenen Punkten ansetzen, um neue Elemente zu bilden. *'Oberklasse extrahieren'* ist dann besonders wichtig, wenn es z. B. zwei Klassen mit ähnlichen Elementen gibt. Dann kann eine Oberklasse und eine Unterklasse erstellt werden. In die Oberklasse werden dann ähnliche Elemente aufgenommen, in die Unterklasse kommen die Elemente, die sich unterscheiden. Der Vorteil dieser Refaktorierung ist wieder, dass redundanter Quellcode herausgefiltert wird.

6.1.7 Große Refaktorierungen

Während die vorhergehenden Hauptprodukte die individuellen Verfahren beim Refactoring repräsentieren, geht es bei diesem eher um den Blick auf den ganzen Refactoring-Prozess. Große Refaktorierungen sind jedoch eher etwas problematisch, da jedes System unterschiedlich aufgebaut ist, die Vorgehensweise somit stark variieren und es sehr viel Zeit in Anspruch nehmen kann, an laufende Systemen ein Refactoring durch zu führen, während die zuvor beschriebenen Refactoring-Methodiken nahezu problemlos durchgeführt werden können. Die Forschung und die Praxis hat sich bisher nur auf kleine Refaktorierungen spezialisiert, so dass an dieser Stelle nur Beispiele großer Refaktorierungen genannt werden können, jedoch keines Falls das ganze Gebiet dargestellt werden kann. Es bleibt zu erwähnen, dass sich große Refaktorierungen immer aus vielen Schritten, kleiner Refaktorierungen zusammensetzen.

Beispiele großer Refaktorierungen:

- *'Vererbungsstruktur entzerren'*, um undurchschaubare, komplexe Vererbungshierarchien aufzulösen.
- *'Prozedurale Entwürfe in Objekte überführen'*, um prozedurale Konzepte in objektorientierte Programmiersprachen umzuwandeln.
- *'Anwendung von der Präsentation trennen'*, um die Anwendungslogik von der Benutzerschnittstelle zu trennen. Diese Trennung ist lebenswichtig für ein langlebiges und florierendes System.
- *'Hierarchie extrahieren'*, um eine übermäßig komplexe Klasse zu vereinfachen, indem sie in eine Gruppe von Unterklassen zerlegt wird.

6.2 automatisierte Softwaretests

Bevor die erarbeiteten Refactoring-Methoden angewandt wurden, war es wichtig, eine Reihe von Testfällen zu erstellen. Diese sollten vollständig automatisiert und selbstüberprüfend arbeiten, um die Zeit für eine mögliche Fehlersuche zu reduzieren und um sofort festzustellen, wann Schnittstellen nach außen hin geändert wurden. Hierbei musste der Quellcode, der refaktoriert wird, berücksichtigt werden.

Die Tests wurden nach größeren Änderungen, während der Implementierungsphase, jedoch mindestens einmal täglich ausgeführt. Zudem wurden die Testreihen unterteilt in funktionale Tests und Komponententests:

- Funktionale Tests
  Testen die Software als ganzen und prüfen die weitere Funktionsfähigkeit des Moduls bzw. des Frameworks. Aufgrund des Refactorings, darf das bisherige Verhalten der Software in keinster Weise verändert werden. Genau dies wird durch funktionale Tests sichergestellt.

- Komponententests
  Testen Abhängigkeiten zwischen Modulen und überprüfen die korrekte Bedienung von Schnittstellen bzw. von Schnittstellen zu anderen Paketen.

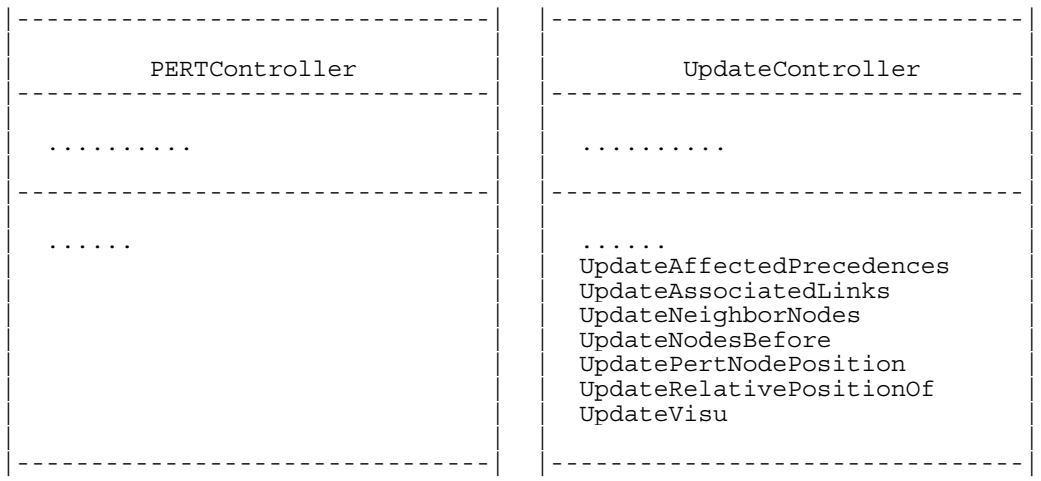
6.3 Quellcode-Beispiele

Im Folgenden werden Beispiele aus dem Quellcode der PERTController-Klasse aufgeführt, die einige der Refactoring-Methodiken aufzeigen. Da der Quellcode der betreffenden Software nicht veröffentlicht werden darf, wurde dieser aus dem Kontext genommen und etwas abgeändert. Das Prinzip der dargestellten Refactoring-Methodiken sollte jedoch immer noch verständlich sein.

```
//////////////////////////////////////////////////////////////////////
// I. Funktionen zusammenstellen
//////////////////////////////////////////////////////////////////////

//===========================================
// Methode extrahieren:
// VOR dem Refactoring
//===========================================

der Status (NONE, PLUS, MINUS) eines Nodes wird immer wieder abgefragt
und danach gesetzt
    --> NONE: PERT Node hat keine Kinder
    --> PLUS: PERT Node hat Kinder, ist jedoch zugeklappt
    --> MINUS: PERT Node hat Kinder. Kinder müssen angezeigt werden.
    --> dies soll nun anhand einer Methode passieren
    --> Redundanz wird eleminiert

    Code wiedeholt sich sehr oft an mehreren Stellen:

    IPertNodeExtension pertNodeExtension = modelNode;
    pertNodeExtension -> GetPertNodeState(nodeState, _enumIndex);

    //Set the State of the Node to PLUS, others to NONE
    if(nodeState == PertNodeExtension::NONE)
    {
        PertNodeState currentState = PertNodeExtension::PLUS;
        pertNodeExtension -> SetPertNodeState(nodecurrState);
    }

    else if(nodeState == PertNodeExtension::MINUS)
    {
        PertNodeState currentState = PertNodeExtension::NONE;
        pertNodeExtension -> SetPertNodeState(nodecurrState);
    }

    else if(nodeState == PertNodeExtension::PLUS)
    {
        PertNodeState currentState = PertNodeExtension::NONE;
        pertNodeExtension -> SetPertNodeState(nodecurrState);
    }


//===========================================
// Methode extrahieren:
// NACH dem Refactoring
//===========================================
void PertController::SetNodeState(PertNodeState nodeState,
                                  IPertNodeExtension pertNodeExtension)
{
    if(nodeState == PertNodeExtension::NONE)
    {
        PertNodeState currentState = PertNodeExtension::PLUS;
        pertNodeExtension -> SetPertNodeState(nodecurrState);
    }

    else if(nodeState == PertNodeExtension::MINUS)
    {
        PertNodeState currentState = PertNodeExtension::NONE;
        pertNodeExtension -> SetPertNodeState(nodecurrState);
    }

    else if(nodeState == PertNodeExtension::PLUS)
    {
        PertNodeState currentState = PertNodeExtension::NONE;
        pertNodeExtension -> SetPertNodeState(nodecurrState);
    }
}

// Aufruf:
    IPertNodeExtension pertNodeExtension = modelNode;
    pertNodeExtension -> GetPertNodeState(nodeState, _enumIndex);

    SetNodeState(nodeState, pertNodeExtension);
```

```
//=============================================
// Temporäre Variable integrieren:
// VOR dem Refactoring
//=============================================

es existiert eine temporäre Variable, der einmal
ein Wert zugewiesen wurde. Diese temporäre Variable
steht bei anderen Refakorierungen im Weg

--> alle Referenzen der Variablen durch den Ausdruck
    ersetzen


    IGENode hParentGENode = hChildGENode->GetParent();
    return hParentGENode;



//=============================================
// Temporäre Variable integrieren:
// NACH dem Refactoring
//=============================================


    return hChildGENode->GetParent();




//////////////////////////////////////////////////////////////////////
// II. Eigenschaften zwischen Objekten verschieben
//////////////////////////////////////////////////////////////////////

//=============================================
// Klasse extrahieren:
// VOR dem Refactoring
//=============================================


-> PERTController-Klasse macht die Arbeit von mehreren Klassen.
-> Es wird eine neue Klasse erstellt, in welche sämtliche
   Update-Methoden verschoben werden.


                |--------------------------------|
                |                                |
                |         PERTController         |
                |--------------------------------|
                |                                |
                |  ..........                    |
                |                                |
                |--------------------------------|
                |                                |
                |  ......                        |
                |  UpdateAffectedPrecedences     |
                |  UpdateAssociatedLinks         |
                |  UpdateNeighborNodes           |
                |  UpdateNodesBefore             |
                |  UpdatePertNodePosition        |
                |  UpdateRelativePositionOf      |
                |  UpdateVisu                    |
                |                                |
                |--------------------------------|
```

```
//============================================
// Klasse extrahieren:
// NACH dem Refactoring
//============================================
```



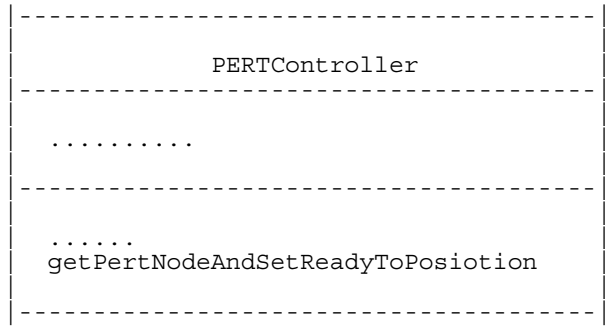

```
////////////////////////////////////////////////////////////////////////
// III. Daten organisieren
////////////////////////////////////////////////////////////////////////


//============================================
// Objekt kapseln
// VOR dem Refactoring
//============================================
```

Zugriff auf ein Objekt erfolgt dirket. Besser ist es, set- und get-
Methoden zu erstellen und nur über diese auf das Objekt zuzugreifen

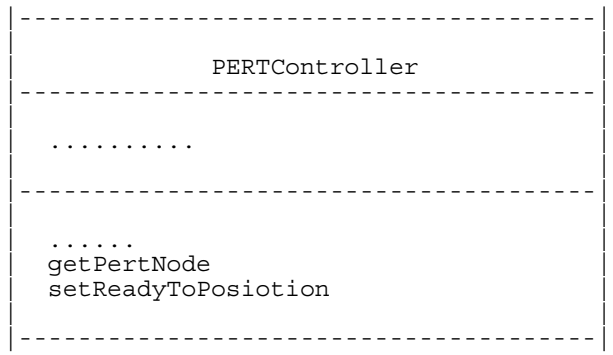
```
    //Member-Variable
    PertWindow* _pPertWindow;

    //Zugriff erfolgt direkt
    PertWindow* pertWindow = _pPertWindow;                  .
                    .
    _pPertWindow = pertWindow;



//============================================
// Objekt kapseln
// NACH dem Refactoring
//============================================

    //Member-Variable
    PertWindow* _pPertWindow;

    PertWindow * getPertWindow() { return _pPertWindow; };
    void setPertWindow( PertWindow pertWindow ) { _pPertWindow = pertWindow; };


    //Zugriff nur noch über set- und get-Methoden
    PertWindow* pertWindow = getPertWindow();
                    .
                    .
                    .
    setPertWindow(pertWindow);
```

```
//===========================================
// Zahlen durch symbolische Konstanten ersetzen
// VOR dem Refactoring
//===========================================

-> numerisches Literal mit besonderer Bedeutung.
-> Konstante erstellen, gemäß ihrer Bedeutung benennen und die Zahl
   dadurch ersetzen

   double getXNodeCoordinate(double xBorder)
   {
        return 25.0 / 2.0 - xBorder;
   }


//===========================================
// Zahlen durch symbolische Konstanten ersetzen
// NACH dem Refactoring
//===========================================

   const double XGRIDLAYOUT_CONSTANT = 25.0;

   .....


   double getXNodeCoordinate(double xBorder)
   {
        return XGRIDLAYOUT_CONSTANT / 2.0 - xBorder;
   }



////////////////////////////////////////////////////////////////////
// IV. Bedingte Ausdrücke vereinfachen
////////////////////////////////////////////////////////////////////


//===========================================
// Redundante Bedingungsteile konsolidieren
// VOR dem Refactoring
//===========================================

-> das gleiche Codefragmente kommt in allen Bedingungsteilen vor
-> dieses Codefragment aus dem bedingten Ausdruck herausziehen


        IPertNodeExt::PertNodeState nodeState = IPertNodeExt::NONE;
        hParentGENodeExt -> GetPertNodeState(nodeState);

        if(nodeState == IPertNodeExt::NONE)
        {
            nodeState = PertNodeExt::PLUS;
            hParentGENodeExt -> SetPertNodeState(nodeState);
            UpdateVisu(hParentGENode);
        }

        else if ( nodeState == DNBIPertNodeExt::NONE_IMPACTED )
        {
            nodeState = PertNodeExt::PLUS_IMPACTED;
            hParentGENodeExt -> SetPertNodeState(nodeState);
            UpdateVisu(hParentGENode);
        }
```

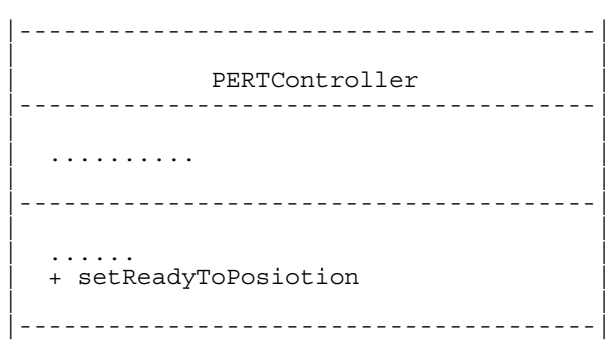
```
//=============================================
// Redundante Bedingungsteile konsolidieren
// NACH dem Refactoring
//=============================================

        IPertNodeExt::PertNodeState nodeState = IPertNodeExt::NONE;
        hParentGENodeExt -> GetPertNodeState(nodeState);

        if(nodeState == IPertNodeExt::NONE)
        {
            nodeState = PertNodeExt::PLUS;
        }

        else if ( nodeState == DNBIPertNodeExt::NONE_IMPACTED )
        {
            nodeState = PertNodeExt::PLUS_IMPACTED;
        }

        hParentGENodeExt -> SetPertNodeState(nodeState);
        UpdateVisu(hParentGENode);



////////////////////////////////////////////////////////////////////
// V. Methodenaufrufe vereinfachen
////////////////////////////////////////////////////////////////////


//=============================================
// Abfrage von Veränderung trennen
// VOR dem Refactoring
//=============================================

-> es existiert eine Methode, die einen Wert zutrückliefert, aber auch
   einen Zustand ändert.
-> zwei Methoden erstellen, eine für die Abfrage und eine für die Änderung.
```



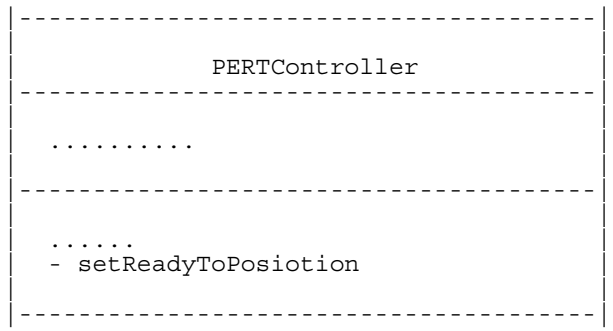

```
//=============================================
// Abfrage von Veränderung trennen
// NACH dem Refactoring
//=============================================
```



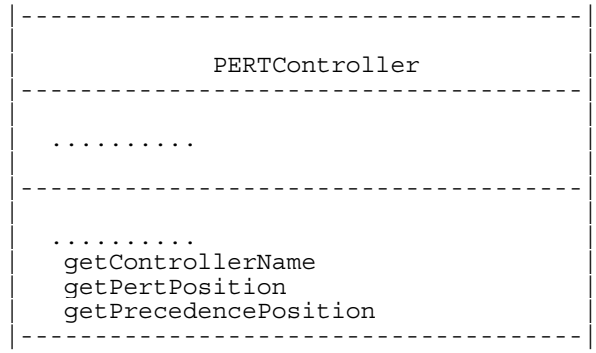
```
//===========================================
// Methode verbergen
// VOR dem Refactoring
//===========================================

-> eine Methode wird von keiner anderen Klasse verwendet
-> Methode als private deklarieren
```




```
//===========================================
// Methode verbergen
// NACH dem Refactoring
//===========================================
```




```
////////////////////////////////////////////////////////////////////
// VI. Ändern der Vererbungshierarchie
////////////////////////////////////////////////////////////////////


//===========================================
// Oberklasse extrahieren
// VOR dem Refactoring
//===========================================

-> es existieren mehrere PERT-Viewer (PERT Chart, Precedence Viewer),
   welche alle die selbe PERTController-Klasse verwenden
-> teilweise gibt es für die Viewer ähnliche Elemente, es gibt
   aber auch Elemente, die sich unterscheiden.
-> ähnliche Elemente bleiben in der Oberklasse
-> Elemente, die sich unterscheiden kommen in verschiedene,
   spezialisierte Unterklassen
```



```
//=============================================
// Oberklasse extrahieren
// NACH dem Refactoring
//=============================================
```