Refactoring-Tools

Manuel Haim, Andreas Schöneck

Fachbereich Mathematik und Informatik
Wintersemester 2006/07

Seminar: Software Refactoring

Dozentin: Prof. Dr. Gabriele Taentzer

Inhalt

Refactoring-Tools 1
Refactoring mit Eclipse 3
Übersicht über die möglichen Refactorings 3
Das Beispiel 4
Move (Verschieben) 5
Encapsulate Field (Feld umschließen) 6
Rename (Umbenennen) 7
Introduce Parameter (Parameter einführen) 8
Extract Constant (Konstante extrahieren) 9
Extract Method (Methode extrahieren) 9
Extract Local Variable (Lokale Variable extrahieren) 9
Change Method Singnature (Methodensignatur ändern) 11
Pull Up/Push Down 13
Use Supertype Where Possible 14
Extract Interface 14
Convert Anonymous Class to Nested 14
Convert Member Type to Top Level 15
Introduce Indirection 16
Extract Superclass 16
Introduce Factory 17
Inline 18
Generalize Declared Type 19
Convert Local Variable to Field 19
Infer Generic Type Arguments 19
Migrate JAR File 20
Create Script/Apply Script/History 20
Was nicht ohne Weiteres geht 20
Andere Tools 20
Literaturverzeichnis 20

Refactoring-Tools

Refactoring mit Eclipse

Übersicht über die möglichen Refactorings

In Eclipse sind die verschiedenen Refactoring-Methoden hauptsächlich über das Menü *Refactor* (abgebildet in Abbildung 1) und das Kontextmenü erreichbar. Je nach Kontext, d.h. abhängig davon ob beispielsweise eine lokale Variable, ein Programmblock oder eine Klasse ausgewählt bzw. markiert ist, ändern sich die Menüoptionen. Einige Refactorings wie Move sind aber auch per Drag & Drop durchführbar. Einen Überblick über alle verfügbaren Refactorings, ihren Effekt und wann sie wie angewendet werden können findet man in Eclipse unter *Help* -> *Help Contents*, dort in der Kategorie *Java Development User Guide* -> *Reference* -> *Refactoring* -> *Refactor Actions*. Die dort beschriebenen Aktionen sollen im Folgenden beleuchtet und an einem Beispiel Schritt für Schritt vorgeführt werden.

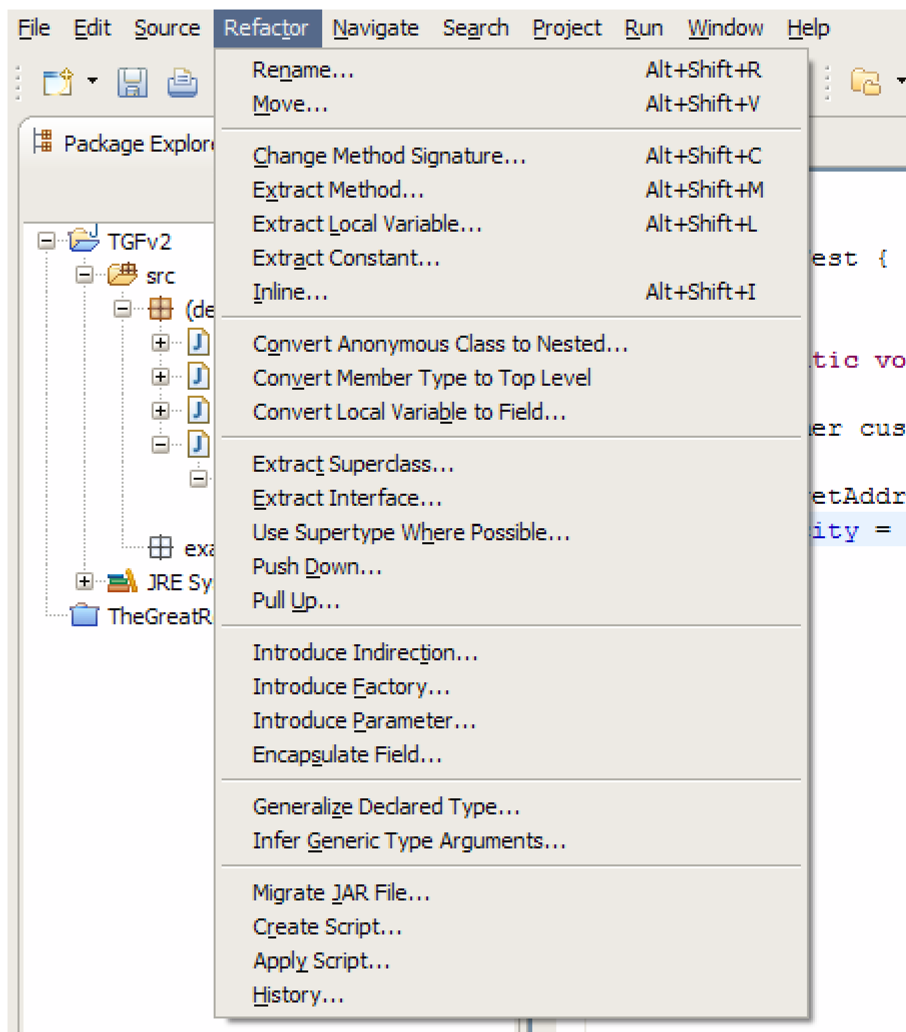
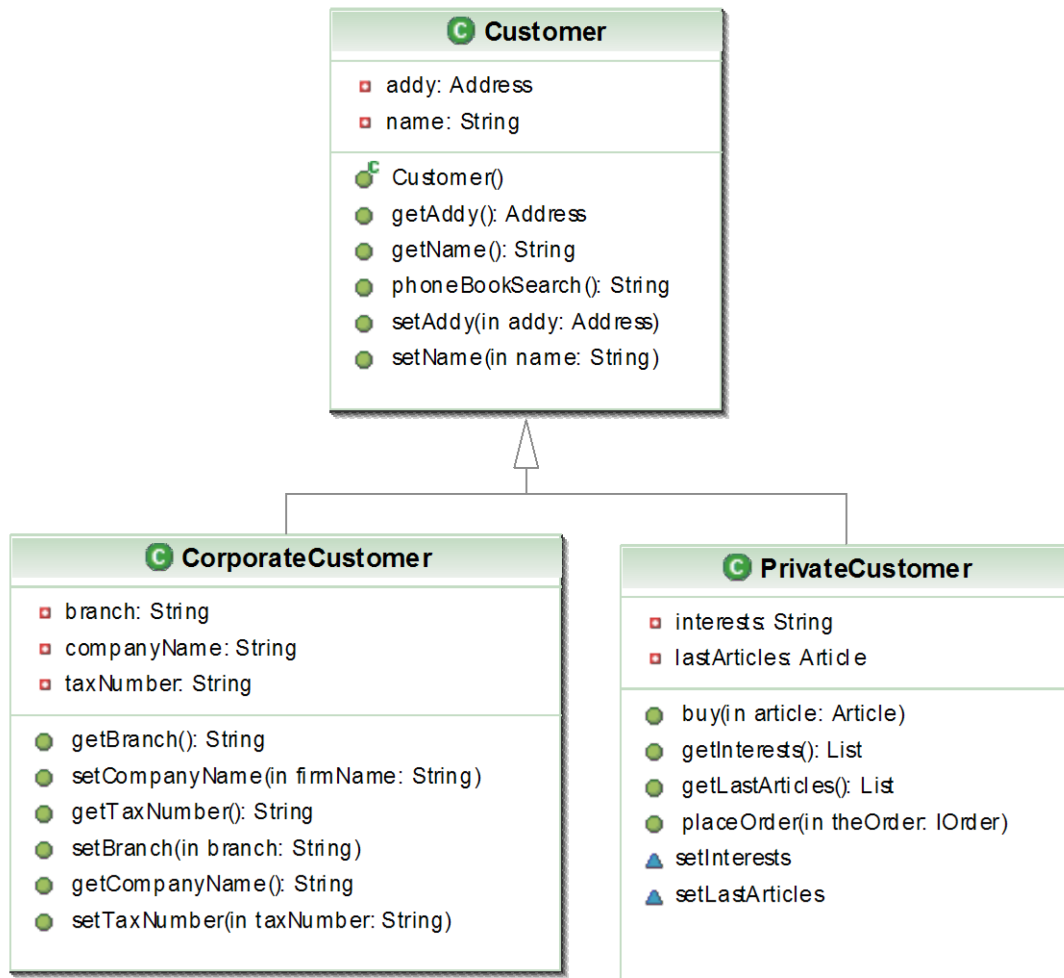


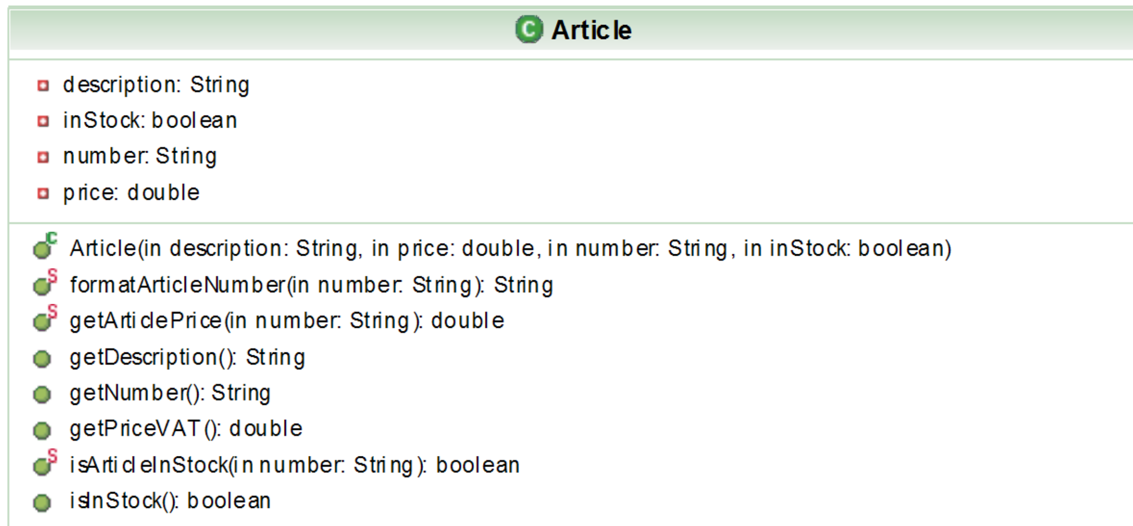
Abbildung 1

Das Beispiel

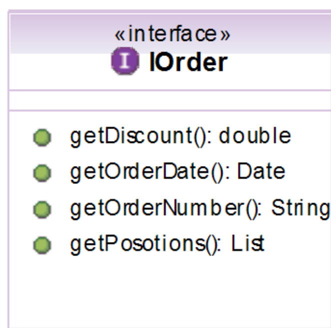
Das Beispiel simuliert ein Bestellsystem. Es gibt die Klasse *Customer*, die einen Käufer repräsentiert. Davon abgeleitet sind die Klassen *CorporateCustomer* und *PrivateCustomer*, die spezielle Felder und Methoden für Firmen- und Privatkunden beinhalten. Jeder Käufer hat eine Adresse, gekapselt in der Klasse *Address.* Außerdem gibt es die Klasse *Article*, die einen Artikel repräsentieren soll. Für Bestellungen existiert das Interface *IOrder*. In der Klasse *Test* wird die Funktionalität der Klassen demonstriert und getestet.



Die einzelnen Methoden sind weitestgehend selbsterklärend. Ein Kunde hat Name und Adresse, zusätzlich kann eine Telefonbuchsuche durchgeführt werden. Für Firmenkunden wird die Branche notiert, der Name der Firma und eine Steuernummer. Bei den Privatkunden werden die zuletzt gekauften Artikel und seine Interessen mitgespeichert, überdies kann er Artikel kaufen und Bestellungen aufgeben.



Die Klasse *Article* repräsentiert einen Artikel. Eine Beschreibung, eine Artikelnummer, der Bruttopreis und ein Status, ob der Artikel lieferbar ist, werden mitgespeichert. Über die statischen Methoden *formatArticleNumber*, *getArticlePrice* und *isArticleInStock* wird anhand der Artikelnummer deren Konformität, der Preis des assoziierten Artikels und die Verfügbarkeit geprüft. Die Methode *getPriceVAT* gibt den Preis inklusive der deutschen Mehrwertsteuer zurück.



Die Schnittstelle *IOrder* dient der Repräsentation von Bestellungen, die im System als Objekte verwaltet werden. Sie berücksichtigt einen Rabatt, das Bestelldatum, die Bestellnummer und die einzelnen Artikel, die bestellt werden sollen.

Der komplette Code ist verfügbar unter
http://www.mathematik.uni-marburg.de/~haim/it/refactoring/

Move (Verschieben)

Das Beispiel weist eine sehr mangelhafte Paketstruktur auf. Alle Klassen befinden sich innerhalb des Standardpakets. Eclipse weist zwar beim Erstellen neuer Klassen darauf hin, dass das Standardpaket nicht benutzt werden sollte, ein stures Ignorieren dieser Warnung führt aber schlussendlich zu einer unübersichtlichen und möglicherweise unlogischen Paketstruktur, sofern man davon überhaupt sprechen kann. Exemplarisch sollen nun die Klassen *Article*, *Customer, CorporateCustomer, PrivateCustomer* und *Address* in ein neues Paket verschoben werden. Hierzu ist es nicht erforderlich, zuerst die neuen Pakete anzulegen.

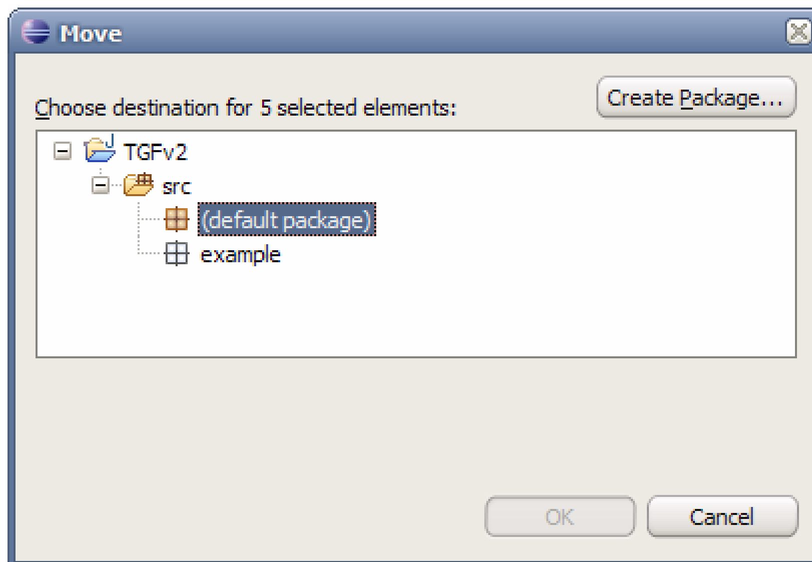


Abbildung 2

Man wählt zunächst die drei Klassen (genauer: deren Quelldateien) aus (mit gedrückter Strg-Taste). Sodann ruft man das Kontextmenü auf und wählt im Untermenü *Refactor* den Punkt *Move*. Alternativ kann stets das Menü *Refactor* im Hauptmenü verwendet werden oder aber die Tastenkombination Alt+Umschalt+V.

Abbildung 2 zeigt das erscheinende Fenster. Mit *Create Package* können nun neue Pakete angelegt werden. In unserem Fall geben wir als Name „*example*" ein, klicken auf *Finish* und auf *OK*. Die Klassen befinden sich nun im neu angelegten Paket, während alle Referenzen darauf (mit import-Anweisungen) angepasst wurden. Wer auf mehr Mausoperationen Wert legt, kann zunächst das neue Paket erstellen und danach die Klassen einzeln oder zusammen dort hinein ziehen.

Encapsulate Field (Feld umschließen)

Eine Klasse, die überwiegend Daten kapselt, sollte ihre Felder nicht öffentlich (public) machen. Liegt nun aber bereits eine solche Klasse vor, die auch irgendwo verwendet wird, ist es mühsam, alle Stellen aufzufinden, wo der direkte Zugriff auf ein Feld erfolgt und jeweils zu ändern. Das Refactoring Encapsulate Field bietet die Möglichkeit, ein öffentliches Feld in ein privates Feld mit getter- und setter-Methoden zu konvertieren und alle direkten Referenzen entsprechend und korrekt anzupassen.

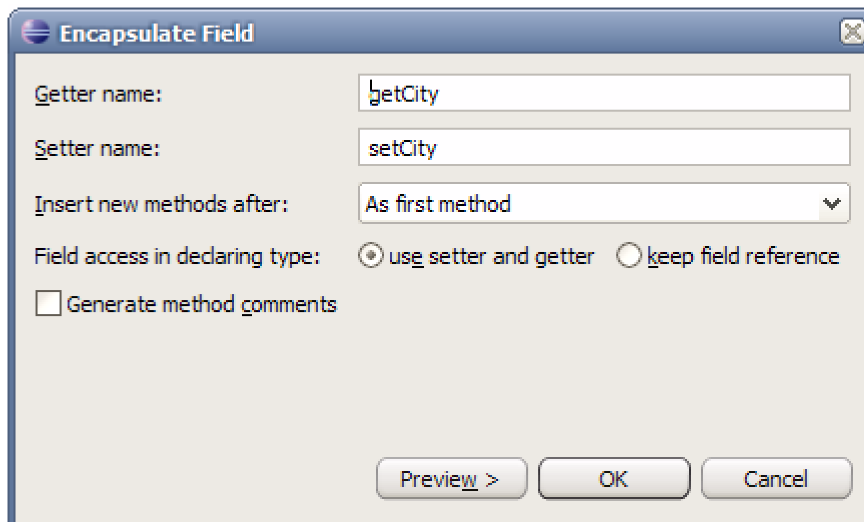


Abbildung 3

Im Beispiel hat die Klasse *Address* drei öffentliche Felder, auf die in der main-Methode in der Klasse *Test* direkt zugegriffen werden. Um den Zugriff über Methoden einzuführen, muss jedes Feld einzeln im Quelltext oder im Outline markiert werden und im Kontextmenü unter *Refactor* der Punkt *Encapsulate Field* gewählt werden. Wir wählen das Feld *city* im Outline und rufen über das Kontextmenü den obengenannten Menüpunkt auf. Das Fenster ist in Abbildung 3 dargestellt. Hier hat man zusätzlich noch die Wahl der Namen der Zugriffsmethoden, sowie den Ort der neuen Methoden im Quellcode und, ob die bestehenden Referenzen ersetzt oder beibehalten werden sollen. Kommentare können zudem hinzugefügt werden. Über *Preview* ist eine Vorschau der Aktionen einsehbar. *OK* übernimmt die Änderungen. Genauso soll mit den Feldern *street* und *zipcode* verfahren werden.

Rename (Umbenennen)

Für beispielsweise sehr ungeschickt benannte Felder oder Methoden, aber auch Klassen und andere Elemente steht das Refactoring Rename zur Verfügung. Möglicherweise sollen neuerdings eingeführte Codekonventionen eingehalten werden, die das eindeutige Benennen von Feldern nach Typ, Funktion und Zugriff vorsehen.

Betrachten wir als Beispiel das Feld *addy* in der Klasse *Customer*. Es beherbergt eine Referenz auf ein *Address*-Objekt. Der besseren Lesbarkeit und Verständlichkeit, sowie evtl. aus Professionalitäts-gründen soll dieses Feld schlicht in „*address*“ umbenannt werden. Dazu wird das bisherige Feld markiert. Dies geht sehr einfach mit der Maus, kann jedoch auch über das Menü *Edit* -> *Expand Selection To* geschehen. Steht der Cursor zwischen irgendwelchen Buchstaben des Feldes, wählt man im eben erwähnten Menü den Punkt *Enclosing Element* (die Tastenkombination ist Alt+Umschalt+Nach-oben). Weitere nützliche Befehle zur Markierung, die bei einigen Refactor-Aktionen sehr nützlich sind, findet man ebenfalls in diesem Menü. Ist das Feld markiert, kann man mithilfe der Tastenkombination Alt+Umschalt+R das Feld umbenennen. Im Kontextmenü findet man den Befehl unter *Refactor* -> *Rename*. Das Fenster zum Umbenennen ist in Abbildung 4 abgebildet und bietet dem Benutzer einige Optionen. Standardmäßig ist die Option *Update references* aktiviert. In unserem Fall sollen noch die Namen der getter- und setter-Methoden angepasst werden. Sollen zusätzlich alle Kommentare und sonstige Zeichenketten aktualisiert werden, erzwingt diese Aktion eine Vorschau, denn in den Kommentaren erfolgt eine rein textuelle Ersetzung (man denke an einen Kommentar, in dem der Name einer Variablen als Wort verwendet wird, d.h. die Semantik der Kommentare wird ggf. beeinflusst).

Klassen und die sie enthaltenden Dateien können mit F2 im Package Explorer umbenannt werden.

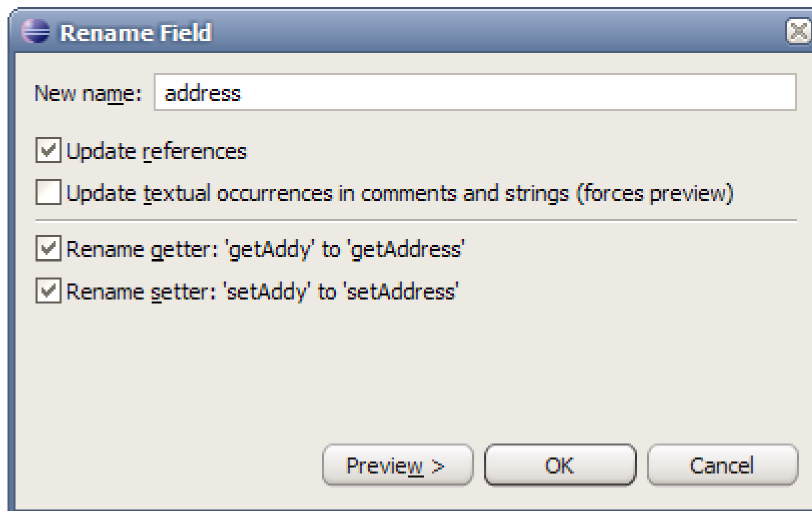


Abbildung 4

Introduce Parameter (Parameter einführen)

Wenn ein Wert innerhalb einer Methode fest ist, der von nun an aber variabel mit dem Aufruf der Methode sein soll, so wird ein Parameter benötigt.

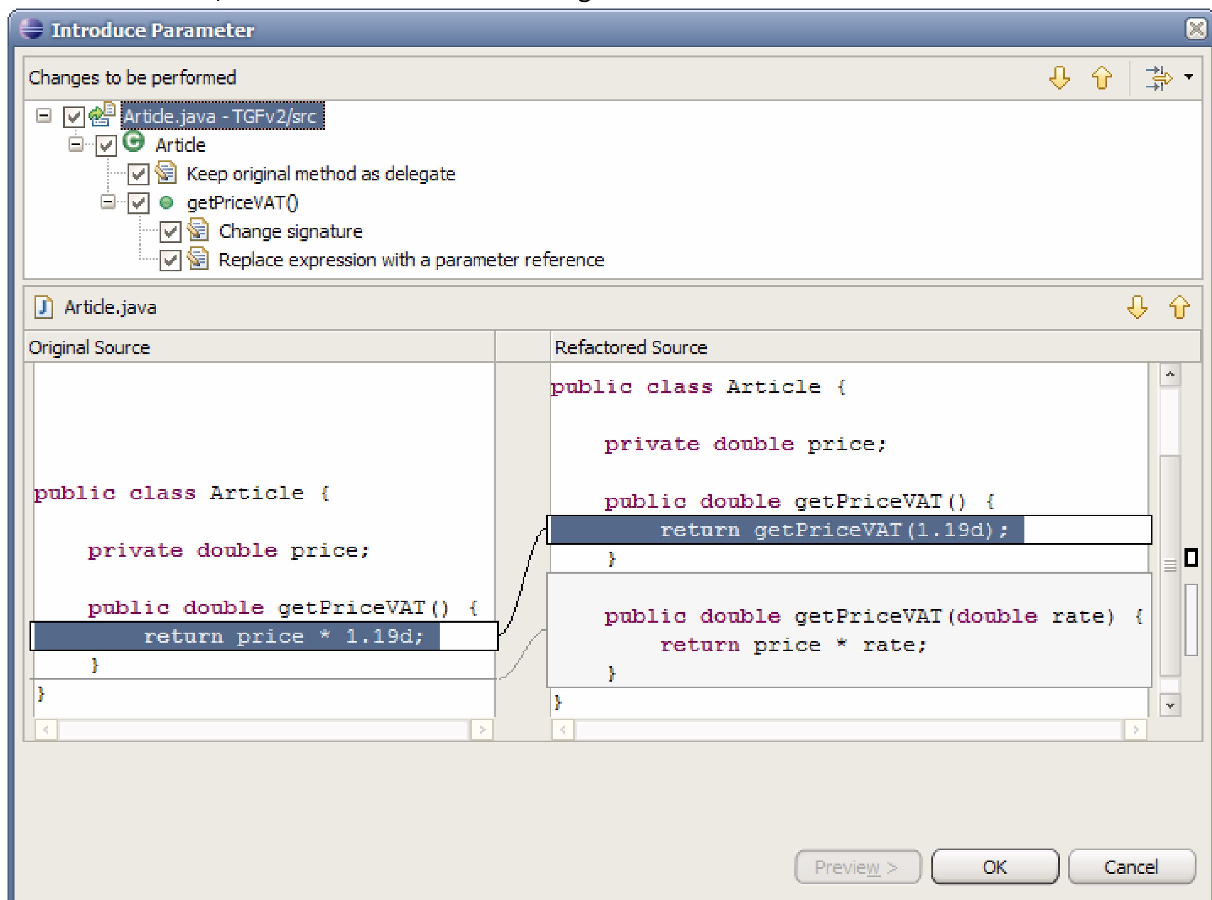


Abbildung 5

Im Beispiel ist in der Klasse *Article* die Methode *getPriceVAT*, die den Preis inklusive Mehrwertsteuer berechnet. Man stelle sich das Szenario vor, dass für den Artikel, da er von nun an in verschiedenen Ländern vertrieben werden soll, der Preis mit unterschiedlichen Mehrwertsteuern, je nach Land,

berechnet werden soll. Im ersten Schritt wird der fest-codierte Wert „1.19d“ in der Methode *getPriceVAT* ausgewählt und *Refactor* -> *Introduce Parameter* aufgerufen. Der Typ des neuen Parameters bleibt double, der Name soll zu *„rate“* geändert werden. Im Fenster wird die neue Signatur der Methode angezeigt. Es besteht darüber hinaus die Möglichkeit, die alte Methode als einen Spezialfall der neuen zu erhalten, was über *Keep original method as delegate to changed method* aktiviert werden soll. Das Markieren als deprecated ist nicht notwendig. Wir erhalten nun zwei Methoden gleichen Namens. Die alte Methode, die die neue mit entsprechendem Parameter aufruft, soll mittels *Refactor* -> *Rename* in *„getPriceGER“* umbenannt werden.

Abbildung 5 zeigt noch einmal das Fenster zur Parametereinführung mit aktivierter Vorschau (*Preview*).

Extract Constant (Konstante extrahieren)

Hat man innerhalb eines Programmes an mehreren Stellen ein und dieselbe Konstante verwendet, so kann man diese (auch in Bedacht, dass sie sich möglicherweise ändern könnte, wenn eine neue Programmversion dies verlangt) für eine Klasse und somit für alle diese Klasse referenzierenden Klassen verfügbar machen.

Im Beispiel könnte der deutsche Mehrwertsteuersatz eine solche Konstante darstellen. Zunächst wird diese Konstante wieder wie bei Introduce Parameter markiert. Im Kontextmenü wird *Refactor* -> *Extract Constant* gewählt. Das Fenster ist in Abbildung 6 abgebildet und bietet die Optionen, den Zugriff zu bestimmen, alle Vorkommen der Konstanten sofort zu ersetzen und überdies alle Vorkommen mit vollqualifizierendem Namen (also inklusive Klassen- und Paketnamen) zu versehen. Als Name wählen wir *„GER_VAT_RATE“* und bestätigen die übrigen Standardeinstellungen.

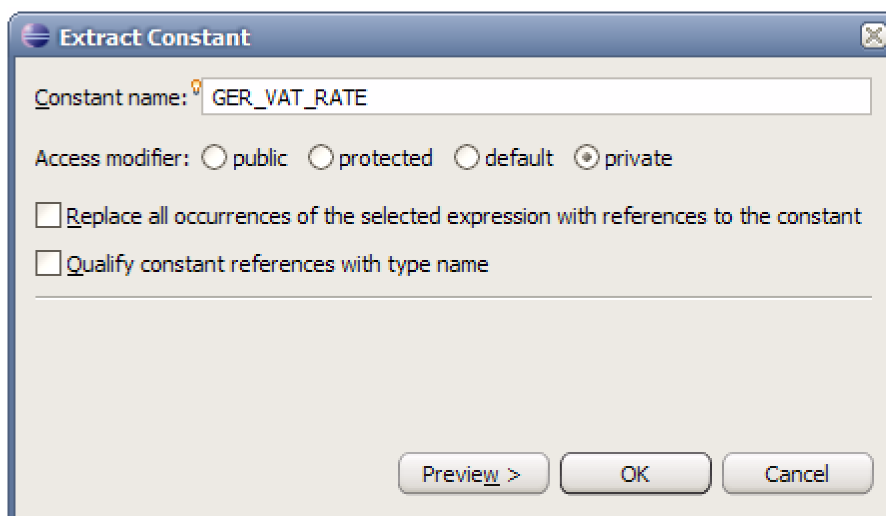


Abbildung 6

Extract Method (Methode extrahieren)

Mit Extract Method können ausgewählte Anweisungen zu einer neuen Methode zusammengefasst werden, etwa, wenn diese mehrmals im Programmtext vorkommen. Alle Vorkommnisse können ersetzt werden. Variable Anteile werden als Parameter übernommen, aber keine Strings, wenn sie direkt im Programmtext stehen.

Extract Local Variable (Lokale Variable extrahieren)

Mit Extract Local Variable kann feststehender Programmtext, wie z.B. Strings variabel gemacht werden, d.h. es wird eine neue Variable hinzugefügt, die mit dem originalen Wert initialisiert wird.

Das folgende Beispiel ist etwas komplexer, weil es mehrere Refactorings einsetzt. Zunächst schauen wir in der Klasse *Test* die *main*-Methode an; wir möchten eine Methode zur Telefonbuchsuche eines *Customer*s haben, da wir möglicherweise nach mehreren verschiedenen Personen suchen möchten. Wir beginnen damit, die festen Strings in dieser Methode in Variablen zu verwandeln. Dazu muss zunächst der ganze erste String einschließlich Anführungszeichen markiert werden, was wieder leicht mit den Markierungsoptionen geschehen kann. Mit Alt+Umschalt+L oder Refactor -> *Extract Local Variable* wird das Fenster zur Variablenextraktion geöffnet, welches in Abbildung 7 abgebildet ist. Hier gibt es die Option, die Variable zu einer Konstanten (final) zu machen und als Erstes wieder die standardmäßige Option, alle Referenzen zu aktualisieren. Wir belassen es bei den Standardoptionen und geben der neuen Variablen den Namen „*customer_name*“. Analog verfahren wir mit den anderen Strings. Ungeschickterweise müssen die Variablendeklarationen jetzt allesamt vor die Deklaration und Initialisierung des *Customer*. Am Leichtesten geht das mit gedrückter Alt-Taste und der Nach-oben-Taste; hierbei wird die aktuelle Zeile um eine Zeile nach oben verschoben. Wir drücken für jede Zeile mit einer neuen Variablendeklaration solange mit gehaltener Alt-Taste die Nach-oben-Taste, bis wir die Deklarationen an der gewünschten Stelle haben.

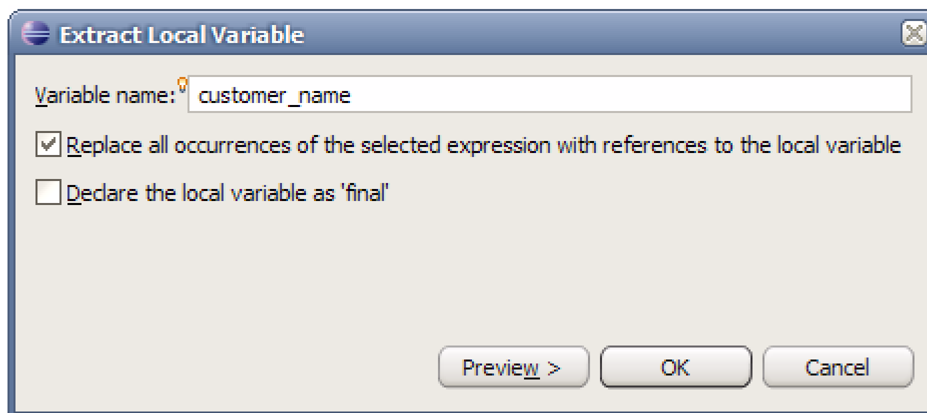


Abbildung 7

Im nächsten Schritt soll die Suchmethode extrahiert werden. Dazu markiert man zuerst die Zeilen von der Deklaration des *Customer* bis einschließlich dem Aufruf von *phoneBookSearch*. Dann wählt man *Refactor* -> *Extract Method* (Alt+Umschalt+M) aus und erhält das Fenster wie in Abbildung 8. Als Namen wählen wir „*searchCustomer*“, bestätigen dies und erhalten die neue Funktion und den korrekten Aufruf. Die Methode ist wie erwartet statisch, zudem hat man die Möglichkeit, Ausnahmen zu berücksichtigen und Kommentare einzufügen. Man kann auch weitere Parameter hinzufügen. Kämen weitere solcher Such-Blöcke vor, könnte man diese ebenfalls gleich ersetzen.

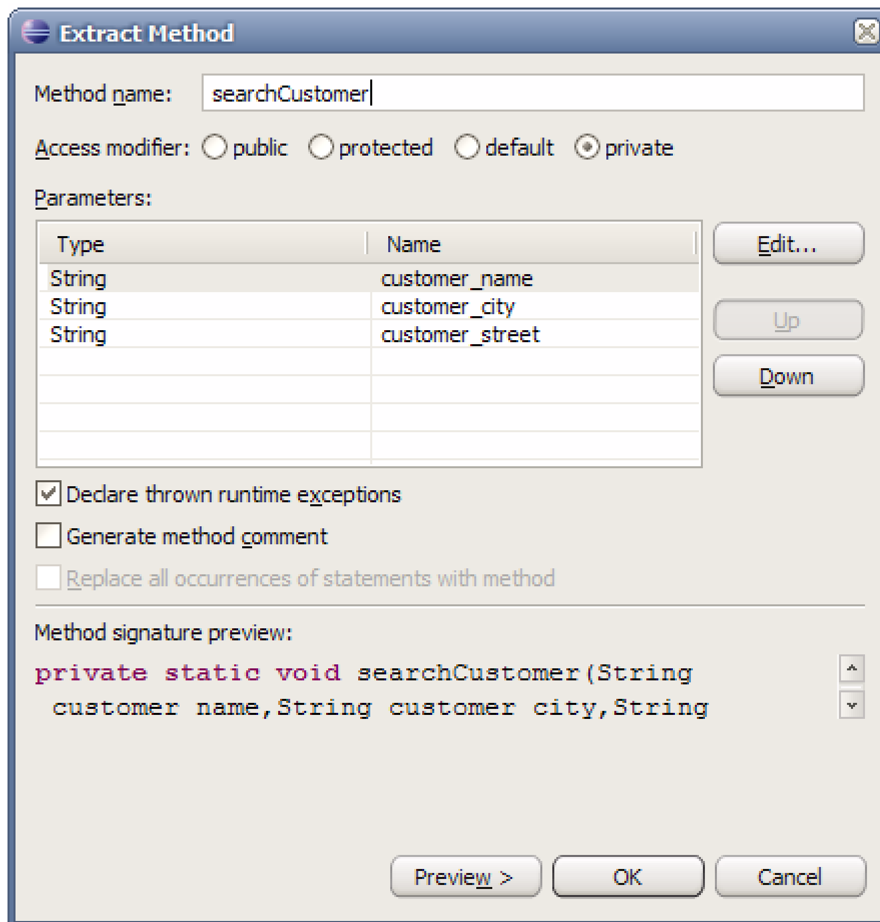


Abbildung 8

Change Method Singnature (Methodensignatur ändern)

Das Refactoring Change Method Signature ist verwandt mit Refactorings wie Rename oder Introduce Parameter. Die Möglichkeiten zum Ändern der Methodensignatur sind vielfältig. Es können beliebig Parameter hinzugefügt oder gelöscht werden. Rückgabewert, Methodenzugriff und ein Name können neu festgelegt werden. Praktisch erscheint vor allem der Reiter *Exceptions*, in dem man die von der Methode geworfenen Ausnahmen spezifizieren kann. Natürlich gibt es wie bei allen Refactorings dieser Art die Option, dass die bestehende Methode als Delegat zur veränderten Methode beibehalten und dabei wahlweise als veraltet gekennzeichnet wird.

Die eben erstellte Methode soll einen Rückgabewert bekommen. Im Outline wird die Methode ausgewählt, danach ruft man im Kontextmenü *Refactor* -> *Change Method Signature* (Alt+Umschalt+C) auf. Zuerst ändern wir den Zugriff auf public, dann den Rückgabewert zu String. Hier könnten jetzt noch die Parameter nach Belieben umgeordnet werden. Das Fenster für Change Method Signature ist in Abbildung 9 abgebildet.

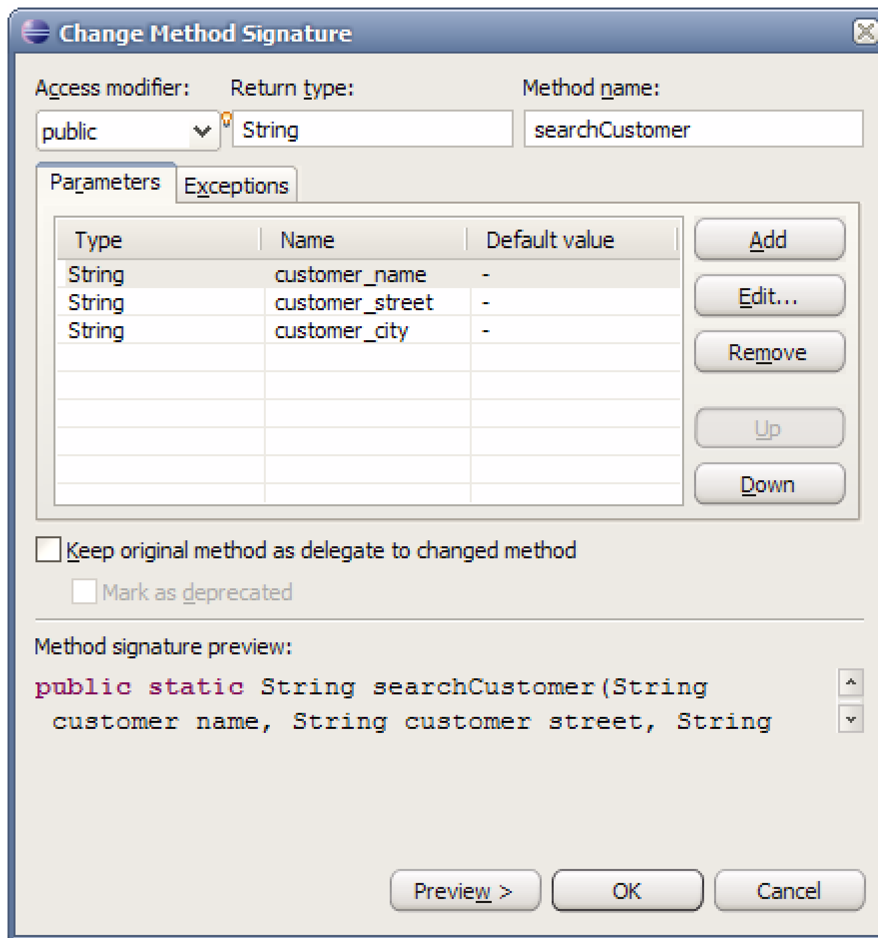


Abbildung 9

Wenn wir jetzt bestätigen, erhalten wir einen Fehler wie in Abbildung 10, weil unsere Methode keinen Wert zurückgibt. Mit einem Klick auf *Continue* ignorieren wir diese Meldung zunächst und fügen noch ein return-Statement vor den Aufruf von *phoneBookSearch*. Man hätte die Rückgabeanweisung auch schon vor dem Durchführen des Refactorings einfügen können, wodurch man sich einen Klick erspart.

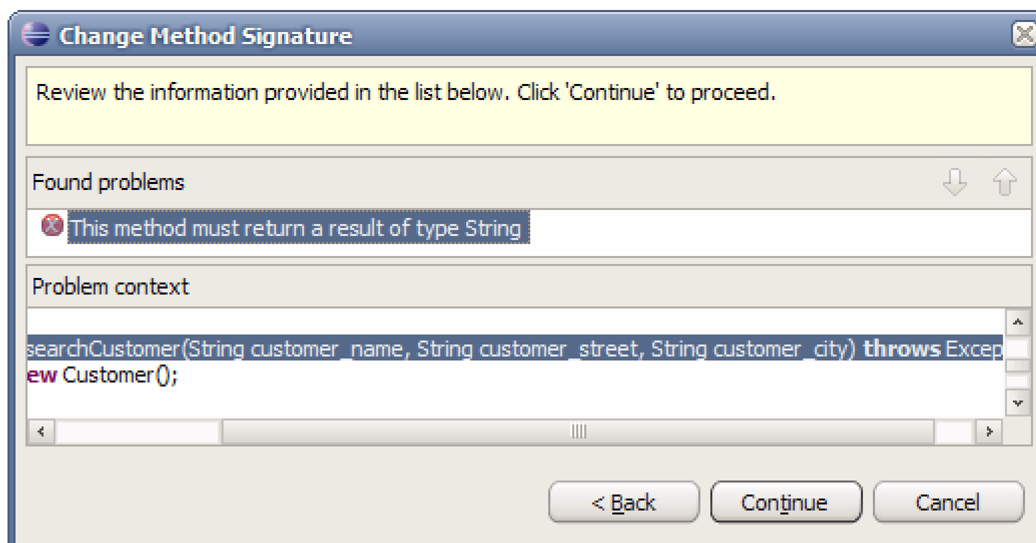


Abbildung 10

Wir hätten uns diesen Schritt sparen können, wenn wir den Rückgabewert von *phoneBookSearch* zuerst in einem String gespeichert hätten, den wir dann beispielsweise auf der Konsole ausgeben. Dies kann durch Löschen des Kommentars bei den entsprechenden Anweisungen nachvollzogen werden. Man markiert für das Extract Method dann alles inklusive der Zeile mit der Rückgabe-variablen *ret*.

Pull Up/Push Down

In einer Klassenhierarchie ist es manchmal sinnvoll, Felder oder Methoden in eine Ober- oder Unterklasse zu verschieben, weil man beispielsweise festgestellt hat, dass eine Methode für alle auf der aktuellen Stufe einer Klassenhierarchie liegenden Klassen benötigt wird, nicht nur in einer. Mit Pull up kann man diese Methode dann in eine zu wählende Oberklasse verschoben werden. Ebenso ist es möglich, dass z.B. Felder, die nur in bestimmten Unterklassen (einem bestimmten Zweig der Hierarchie) benutzt werden in eine speziellere unterliegende Klasse mit Push down zu verschieben.

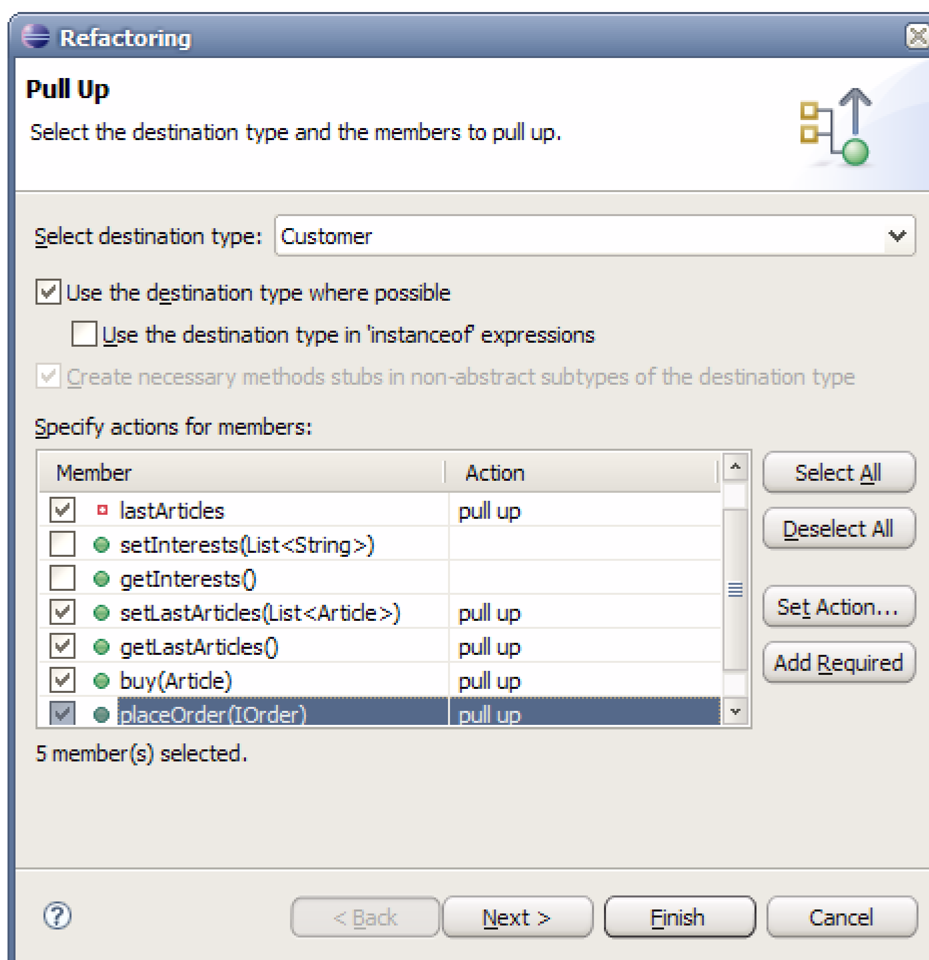


Abbildung 11

Im Beispiel gibt es zwei von *Customer* abgeleitete Klassen: *CorporateCustomer* für Firmenkunden (mit Firmenname, Umsatzsteuer-Identifikationsnummer und einer Branchenbezeichnung) und *PrivateCustomer* (mit einer Liste von gesammelten Interessen und zuletzt gekauften Artikeln) für Privatkunden. Fortan soll für alle Typen von Kunden eine Liste der zuletzt gekauften Artikel festgehalten werden. Dazu führen wir ein Pull Up auf das Feld *lastArticles* in *PrivateCustomer* durch. Zunächst wird das Feld ausgewählt. Dann ruft man im Kontextmenü den Punkt *Refactor* -> *Pull Up* auf. Das Fenster in Abbildung 11 lässt eine Auswahl der zu verschiebenden Member und des

Verschiebungsziels zu. Zusätzlich zum vormarkierten *lastArticles* wählen wir dessen getter- und setter-Methoden, *buy* und *placeOrder* aus. Auf der nächsten Seite, auf die man mit *Next* gelangt, wird zusätzlich angeboten, die alten getter- und setter-Methoden zu löschen. Hier behalten wir die Standardmarkierung. Ein Klick auf *Next* zeigt eine Vorschau, *Finish* übernimmt zu jeder Zeit sofort die Änderungen.

Use Supertype Where Possible

Dieses Refactoring versucht, an allen möglichen Stellen eine Oberklasse des gewählten Typs zu benutzen. Betrachtete man die Pull Up Aktion weiter oben, so findet man in dessen Optionen den Punkt *Use the destination type where possible*. Hat man hier die Option nicht gesetzt, erreicht man nachträglich das Durchführen durch das Refactoring Use Supertype Where Possible.

Im Beispiel wählt man dann die Klasse *PrivateCustomer*, deren Member ohne die angesprochene Option verschoben worden sind, aus und selektiert im Kontextmenü unter *Refactor* den Punkt *Use Supertype Where Possible*.

Extract Interface

Will man zu einer Klasse deren Schnittstelle haben, so steht die Option Extract Interface zur Verfügung.

Im Beispiel könnte man ein Interface aus *Article* generieren. Das Prozedere ist denkbar einfach. Man markiert den Klassennamen und wählt im Menü *Refactor* den Unterpunkt *Extract Interface*. Neben einem Namen (z.B. „IArticle“) für die Schnittstelle hat man noch diverse andere Optionen, insbesondere die Wahl der Methoden und deren Modifikatoren, die die Schnittstelle deklariert. Wir wählen alle möglichen Methoden aus und aktivieren zudem *Use the extracted interface type where possible*, was analogen Effekt zu Use Supertype Where Possible hat.

Convert Anonymous Class to Nested

Wer in Java anonyme Klassen verwendet, tut dies in der Absicht, diese nur einmal zu verwenden. Kommt jedoch ans Licht, dass die anonyme Klasse doch mehrmals verwendet wird, dann sollte sie, um den Code übersichtlich zu halten, in eine Member-Klasse umgewandelt werden, die dann an mehreren Stellen Verwendung finden kann.

Im Beispiel finden wir das Interface *IOrder*. Es dient dazu, Bestellungen zu repräsentieren. Die Methode *test3* der Testklasse benutzt eine anonyme Klasse für eine Bestellung. Es könnte sich mit der Zeit herausgestellt haben, dass öfter die in der Bestellung angegebenen Artikel bestellt werden. Die Klasse wird also tendenziell im System öfter benötigt. Im ersten Schritt konvertieren wir diese anonyme Klasse also in eine Member-Klasse. Dazu markiert man den Namen der Schnittstelle in der Deklaration der anonymen Klasse und wählt im Kontextmenü *Refactor* -> *Convert Anonymous Class to Nested*. Es erscheint das Fenster aus Abbildung 12. Als Name soll *„DVDOrder“* dienen. Die neue Klasse soll default werden, ansonsten bleiben die Standardeinstellungen. Wir erhalten die gewünschte Member-Klasse, die Deklaration der anonymen Klasse wurde entsprechend ersetzt.

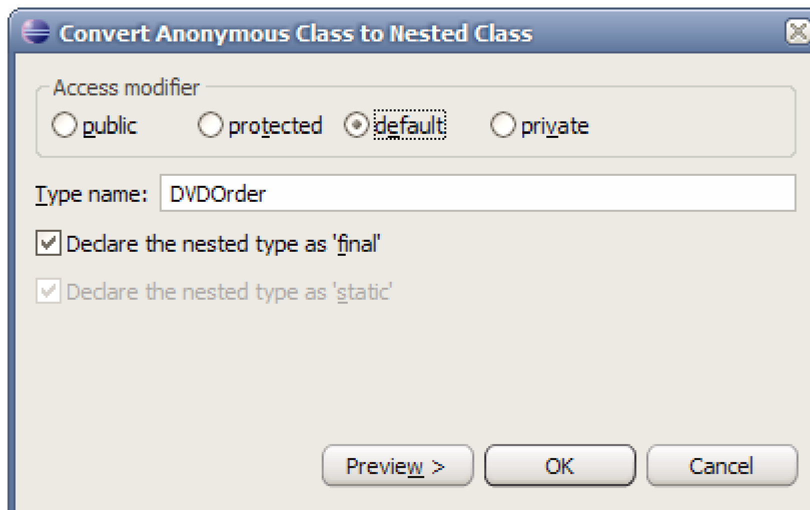


Abbildung 12

Convert Member Type to Top Level

Convert Member Type to Top Level

Changes to be performed

Test.java - TGFv2/src
Update imports
Test
Convert Member Type to Top Level
test3()
Create 'DVDOrder.java' - TGFv2/src

Test.java

Original Source

```
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

public class Test {

    static final class DVDOrder
        public Date getOrderDate

            return new Date(Syst
        }

        public String getOrderNu
```

Refactored Source

```
public class Test {

    public static void main(S

        PrivateCustomer cust

        cust.setName("J. Emar
        cust.getAddy().street
        cust.getAddy().city =

        cust.phoneBookSearch(
```

OK
Cancel

Abbildung 13

Wenn eine Member-Klasse oft benötigt wird, kann es lästig sein, immer den kompletten Qualifizierer für die eingebettete Klasse anzugeben. Möglicherweise ist die Member-Klasse noch nicht einmal von außen zugreifbar. In diesem Fall ist Convert Member Type to Top Level anzuwenden.

Im Beispiel wollen wir die eben erstellte Member-Klasse *DVDOrder* generell verfügbar machen. Im Outline wird zuerst die Klasse markiert, dann im Kontextmenü *Refactor* -> *Convert Member Type to Top Level*. Im Fenster in Abbildung 13 sehen wir sofort eine Vorschau. Alle nur für diese Klasse benötigten Importe werden entfernt und in eine neue Datei für die Klasse übernommen.

Introduce Indirection

Das Refactoring Introduce Indirection erstellt eine statische Methode, die mit einer gegebenen Referenz und entsprechenden Parametern die ausgewählte Objektmethode aufruft. Bei dieser Aktion können auch alle auftauchenden Aufrufe durch die Indirektion ersetzt werden.

Eine Indirektion soll für *Article.isInStock* erstellt werden. Wir markieren die Methode und wählen *Refactor* -> *Introduce Indirection*. Man hat nun die Wahl eines Namens („*isArticleInStock*“) und der Klasse (bleibt *Article*), die diese Methode deklariert. Zusätzlich wird eine Warnung angezeigt, sollte die Methode dadurch überladen werden.

Extract Superclass

Eine Auswahl von Klassen soll eine gemeinsame Oberklasse bekommen. Hierzu benötigt man das Refactoring Extract Superclass.

Stellen wir uns vor, *PrivateCustomer* und *CorporateCustomer* hätten keine gemeinsame Oberklasse. Um dies zu simulieren, kommentieren wir den Rumpf des Konstruktors von Customer aus, dies geht sehr einfach mit *Source* -> *Toggle Comment*. Anschließend benennen wir *Customer* in *„OldCustomer“* um. Danach sollen alle Felder mit Pull Down in die Unterklassen verschoben werden. Die Fehler in der Testklasse können dadurch behoben werden, dass man in der *main*-Methode *PrivateCustomer* einsetzt. Zusätzlich sollte die Vererbungsstrukturgelöscht werden, indem die extends-Anweisungen entfernt werden. Im Menü *Refactor* wählt man nun den Punkt *Extract Superclass*, idealerweise hat man zuvor *CorporateCustomer* oder *PrivateCustomer* markiert. Im Fenster, welches in Abbildung 14 gezeigt ist, wählt man durch Klicken von *Add* die jeweils andere Klasse hinzu. Die neue Klasse soll wieder *„Customer“* heißen. Als Member wählen wir alle zuvor mittels Push Down verschobenen Felder und Methoden. Mit *Next* geht es weiter auf die Seite, auf der spezifiziert wird, welche Member der Subklassen gelöscht werden sollen. Sinnigerweise wählen wir an dieser Stelle alle Member und klicken auf *Finish*. Würde man nur eine Teilauswahl treffen, würde der Memberzugriff angepasst werden (auf protected). Die resultierende Klasse *Customer* ist nun ihrem Original sehr ähnlich. Lediglich der Konstruktor muss noch angepasst werden.

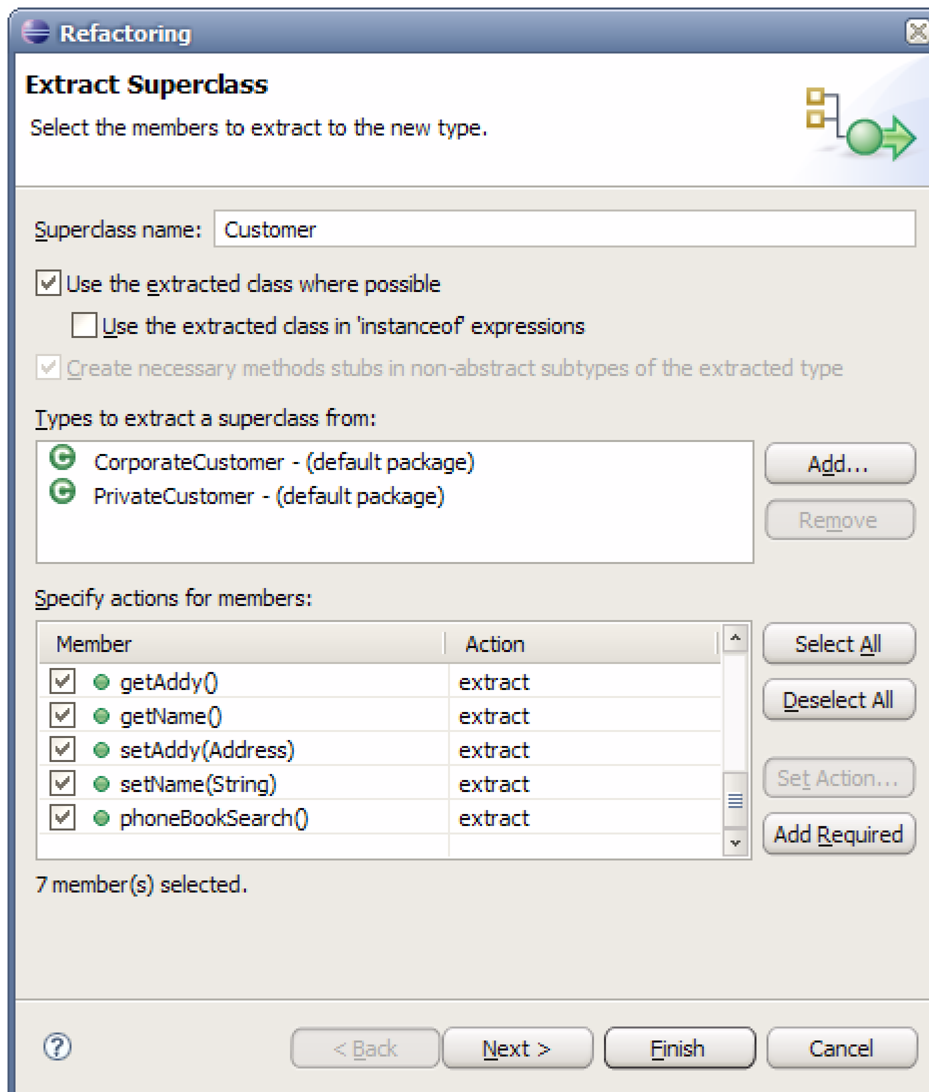


Abbildung 14

Introduce Factory

Introduce Factory erstellt eine dem Factory-Entwurfsmuster folgende Methode (eventuell in einer neuen Klasse), die Objekte bestimmter Typen, deren Konstruktoren verborgen sein sollen, erzeugt.

Im Beispiel soll eine Factory für die Klasse Article erstellt werden, die das Anlegen neuer Artikel ermöglicht und gleichzeitig prüft, ob die Artikelnummer, der automatisch „TGF-" vorangestellt wird, schon dieses Präfix enthält und ob der Artikel vorrätig ist oder bestellt werden muss; dies und der Preis, der in Zukunft nachgeschaut wird, sollte bei Konstruktion nicht mehr angegeben werden. Als erstes wird der Konstruktor markiert und im Menü *Refactor* der Punkt *Introduce Factory* ausgewählt. Im Dialog (siehe Abbildung 15) kann man den Namen der Methode und die Klasse, in der die Factory erzeugt wird angeben. Der Konstruktor kann privaten Zugriff erhalten – welchen Zugriff der Konstruktor letzten Endes erhält, hängt allerdings davon ab, in welcher Klasse die Factory-Methode sein wird. Hier sollen die Standardeinstellungen behalten werden. Alle Aufrufe des Konstruktors werden durch Aufrufe der Factory-Methode ersetzt.

Abbildung 15

Nun soll die Factory-Methode angepasst werden. Über Change Method Signature sollen die Parameter *inStock* und *price* gelöscht werden. Als Parameter im Konstruktoraufruf werden die Methoden *isArticleInStock* anstatt *inStock* und *getArticlePrice* anstatt *price* dienen. Für number führen wir jetzt noch eine lokale Variable ein, die anstatt bloß mit *number* mit dem Ergebnis des Aufrufs von *formatArticleNumber* initialisiert wird. Die Codevervollständigung kommt dem Programmierer hier sehr zur Hilfe. Als Parameter haben die angegebenen Methoden jeweils lediglich die Artikelnummer. Die Fabrikmethode sieht am Ende etwa wie folgt aus:

```
public static Article createArticle(String description, String number) {
            String artno = formatArticleNumber(number);
            return new Article(description, getArticlePrice(artno), artno,
                        isArticleInStock(artno));
      }
```

Inline

Methoden, die eigentlich nur Standardaufgaben, wie das Addieren zweier Zahlen ausführen und/oder sehr selten verwendet werden, können überflüssig sein. Hierzu löscht das Refactoring Inline die Methodendeklaration und fügt an die Stelle der Methodenaufrufe den jeweiligen Methodenrumpf. Inline stellt den vorigen Zustand nach einem Extract Method wieder her.

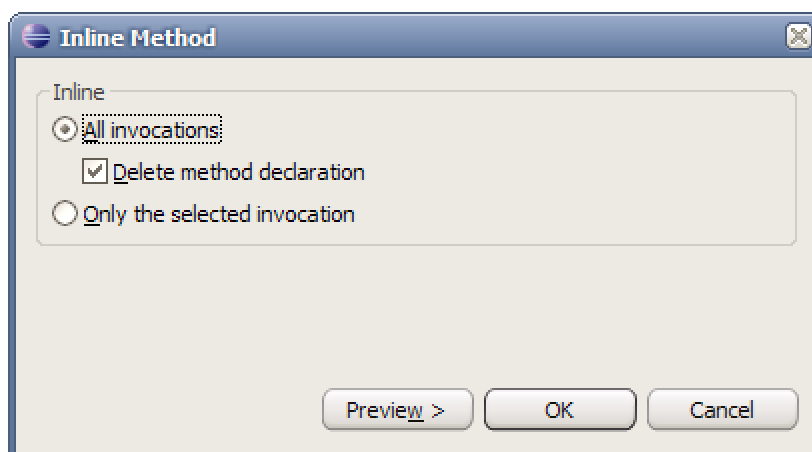


Abbildung 16

Die eben verwendete Methode formatArticleNumber macht nichts weiter als eine Zeichenkettenersetzung und wird zudem nur einmal gebraucht. Wir können den Aufruf markieren und Refactor -> Inline wählen. Ohne weiteres Zutun wird lediglich dieser Aufruf durch den

Methodenrumpf (mit entsprechend eingesetzten Parametern) ersetzt. Man kann aber auch alle Aufrufe der Methode, in unserem Fall ist das nur einer, ersetzen und die Methode abschließend über Delete method declaration komplett entfernen (siehe

Abbildung 16).

Generalize Declared Type

Bei Parametern, Feldern oder lokalen Variablen die vom Typ eines Referenztyps (einer Klasse) sind, ist es manchmal nötig, einen Obertyp des aktuellen Verweistyps zu benutzen, um mehr Flexibilität zu erhalten. Hierzu wird das Refactoring Generalize Declared Type angewendet. Kann die ausgewählte Referenz problemlos zu einer Referenz des Obertyps gemacht werden, so wird das Refactoring durchgeführt.

Um die Verwendung ein wenig zu illustrieren betrachte man die Methode *test* der Klasse *Test*. Sie sucht nach einer Firma Sun in Marburg. Wahrscheinlich wurde diese Testmethode von jemandem geschrieben, der nicht unbedingt die Spezifikation der Klassen kannte. In jedem Fall ist hier auch die Verwendung der Oberklasse *Customer* möglich, da auf kein spezielles Feld von *CorporateCustomer* zugegriffen wird. Markiert man den Typ der Variablen *fc* und wählt aus dem Kontextmenü *Refactor* -> *Generalize Declared Type*, so bestätigt Eclipse, dass tatsächlich *Customer* ausreicht. Die einzig wählbare Klasse ist *Customer*, wie Abbildung 17 beweist.

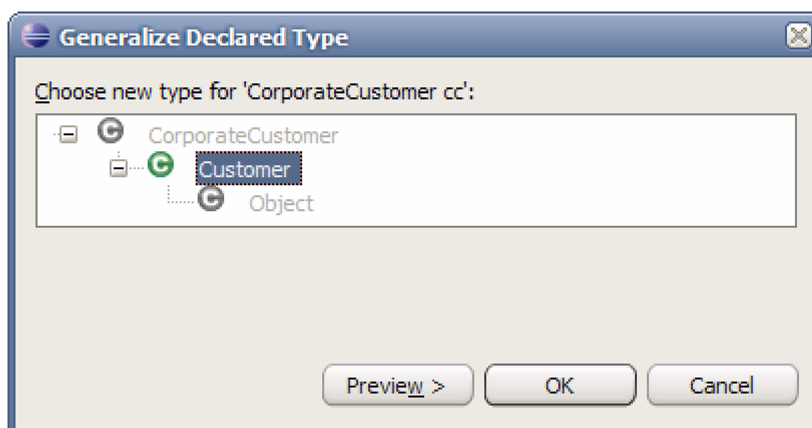


Abbildung 17

Convert Local Variable to Field

Soll eine lokale Variable in ein Feld umgewandelt werden, etwa, wenn der Wert global für eine Klasse gesetzt werden können soll, so sollte das Refactoring Convert Local Variable to Field angewendet werden. Als Beispiel kann man sich vorstellen, dass Konfigurationseinstellungen aus Performance-gründen nicht in jeder Methode einer Klasse neu ausgelesen werden sollen, sondern lediglich einmal beim Erstellen eines Objektes, d.h. im Konstruktor. Man wählt für dieses Refactoring zuerst die lokale Variable aus und dann den entsprechenden Menüpunkt.

Infer Generic Type Arguments

Dieses Refactoring sucht zuerst alle Vorkommnisse eines bestimmten Typs und ersetzt diese dann durch einen generischen Parameter. Somit fallen umständliche Typkonvertierungen (Casts) weg, man spart sich möglicherweise die Mehrfachdeklaration ein und derselben Methode, die lediglich andere Typen benutzt, sonst aber vielleicht sogar textuell gleich ist.

Migrate JAR File

Wenn man innerhalb eines Projekts eine neue Version einer Bibliothek in Form einer JAR-Datei verwenden möchte, kann Migrate JAR File hilfreich sein. Sind im neuen JAR-Archiv Refactoring-Informationen gespeichert, so werden diese übernommen, damit keine Fehler beim Kompilieren – wegen beispielsweise umbenannter Methoden oder veränderter Schnittstellen – auftreten.

Create Script/Apply Script/History

Create Script erstellt ein Refactoring-Skript, das aus ausgewählten Refactorings, die im aktuellen Workspace durchgeführt worden sind beruht. So können bestimmte Refactorings leicht wieder mit Apply Script angewendet werden. Mit History kann der Verlauf der durchgeführten Refactorings eingesehen werden; einzelne Einträge könne hieraus gelöscht werden.

Was nicht ohne Weiteres geht

in Beispiel für eine Aktion, die sich nicht auf die verfügbaren Refactorings zurückführen lässt (z.B. in Form eines Refactoring-Skripts), ist das Zusammenfassen bestimmter Felder einer Klasse zu einem Feld einer neuen Klasse inkl. Anpassung der Referenzen. So kann eine Klasse *Customer*, die eine komplette Adresse enthält, weniger aufwändig durch Kopieren der bestehenden Klasse oder Neuerstellen dahingehend verändert werden, dass die Adresse in eine neue Klasse *Address* ausgelagert wird, als mit Refactoring-Methoden.

Andere Tools

Heute unterstützen alle namhaften Entwicklungsumgebungen zumindest einen gewissen Grundumfang von Refactoring-Methoden. Für Java gibt es deren reichlich, als da wären: IntelliJ Idea und CodeGear (Borland) JBuilder. Überdies existieren noch verschiedene Plugins für die Entwicklungsumgebungen: JFactor (JBuilder), XRefactory (Emacs), RefactorIt (diverse, darunter: NetBeans, Eclipse, JDeveloper), JRefactory (JBuilder, NetBeans) und andere.

Auch für andere Laufzeitumgebungen wie .NET existieren entsprechende Tools bzw. gibt es Unterstützung in den jeweiligen IDEs. Ein prominentes Beispiel für ein Add-In, welches die Refactoring-Möglichkeiten von Visual Studio erheblich erweitert, ist Refactor! Pro.

Für C/C++ (Visual C++) gibt es mit Ref++ ein Add-In für Visual Studio. SlickEdit ist ein Editor, der ebenfalls Refactoring-Unterstützung anbietet und den es auch als Plugin für Eclipse und Visual Studio gibt.

Literaturverzeichnis

Fowler, Martin. *Refactoring Tools.* 23. Februar 2007. http://www.refactoring.com/tools.html.

Omondo. *Eclipse - Omondo - The Live UML Company.* 23. Februar 2007.
http://www.omondo.de/default.asp.

The Eclipse Foundation. *Eclipse.org home.* 23. Februar 2007. http://www.eclipse.org/.