MSO
Refactoring

Hans Philippi

October 2, 2018

This lecture

- What is *refactoring*?
- ... or how to deal with the horrible code your colleagues have created
- ... or how to deal with the horrible code you created yourself

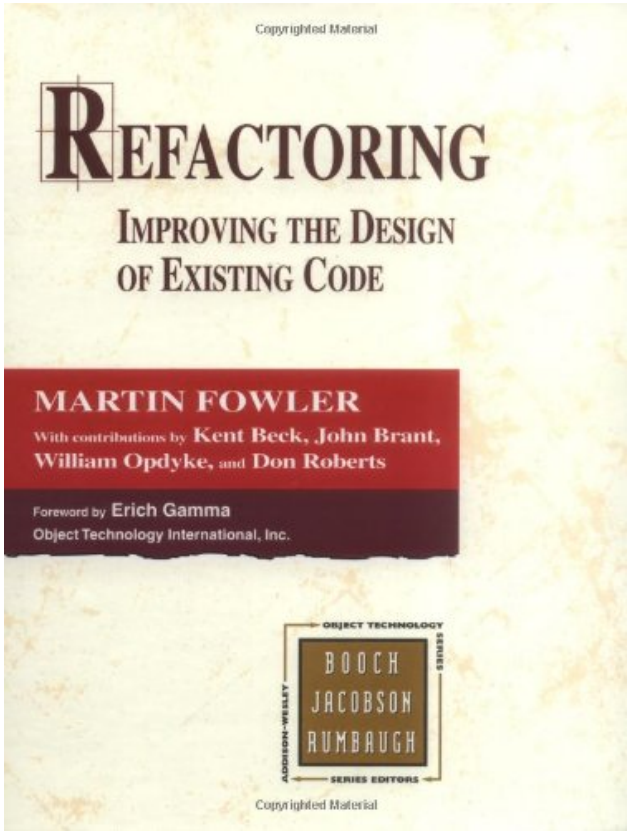
Applying design patterns

- So far, we have often considered the situation where we can start designing new software from scratch
- In practice, this is uncommon – you usually have to develop with existing systems
- Does that mean that design patterns are not useful in practice?

Iteratively improving designs

You can still apply design patterns to existing software:

- to improve the internal structure
- to facilitate testing
- to pave the way for new features

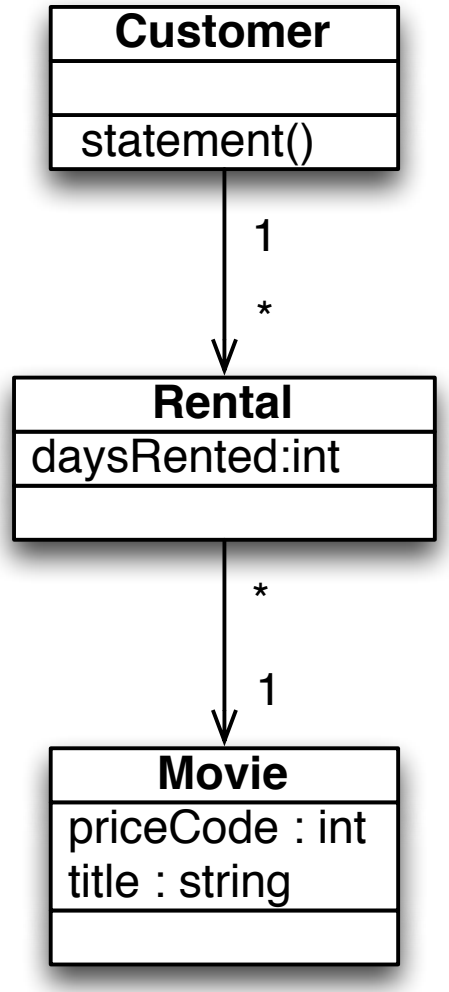


Refactoring *(noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour*

Case study: refactoring a movie rental system

- Let us have a look at a refactoring case study taken from the first chapter of Fowler's book
- Starting with some poorly designed code, we will refactor this step by step

Starting situation



Initial classes

- The Movie class defines three price codes as integer constants:
  CHILDREN $= 0$
  STANDARD $= 1$
  NEW $= 2$
- The Rental class tracks the number of days a particular movie has been rented
- Finally, the Customer class records a list of Rentals

Statement computation

```
foreach (Rental v in rentals) {
  v_price = 0;
  switch (v.getMovie().getPriceCode()) {
    case Movie.STANDARD  :
      v_price = 3 euro  * v.daysRented; break;
    case Movie.NEW  :
      v_price = 5 euro  * v.daysRented; break;
    case Movie.CHILDREN  :
      v_price = 2 euro * v.daysRented; break;
   }
   totPrice += v_price;
   pts++;
   if (v.getMovie().getPriceCode() == Movie.NEW)
     pts++;
}
```

Question: What is wrong with this code?

Weak cohesion

- This fragment is computing the price of the rental and the customer points earned . . .
- . . . so it deals with *two responsibilities* at the same time!
- We will introduce a separate method for dealing with the details of rental prices

Extract method

```
foreach (Rental v in rentals) {
  v_price = calculatePrice (v);
  totPrice += v_price;
  pts++;
  if (v.getMovie().getPriceCode() == Movie.NEW) {
    pts++;
  }
...
```

... but we still are annoyed by the details of calculating the earned points

Extract method

```
foreach (Rental v in rentals) {
  v_price = calculatePrice(v);
  totPrice += v_price;
  pts = calculateEarnedPoints(v,pts);
...
```

- Note that we have to pass along enough information to compute the new running total of earned points
- In this example, we pass both the video `v` and current total points `pts` to `calculateEarnedPoints`
- We also notice that `v_price` is superfluous

Getting rid of temporary variables

```
foreach (Rental v in rentals) {
  totPrice += calculatePrice(v);
  pts += calculateEarnedPoints(v);
...
```

- Ah, that looks better!
- But should both these computations really be in the same loop?

Overloading loops

- But should both these computations really be in the same loop?
- Probably old school programmer habits . . .
- *Let us do two calculations in the same loop*
- *That saves processing time!*
- Yeah, let us save 0.02 msec processing time at the expense of losing cohesion . . .
- Wouldn't it be better to define separate `getCharge()` and `getRenterPoints()` methods?

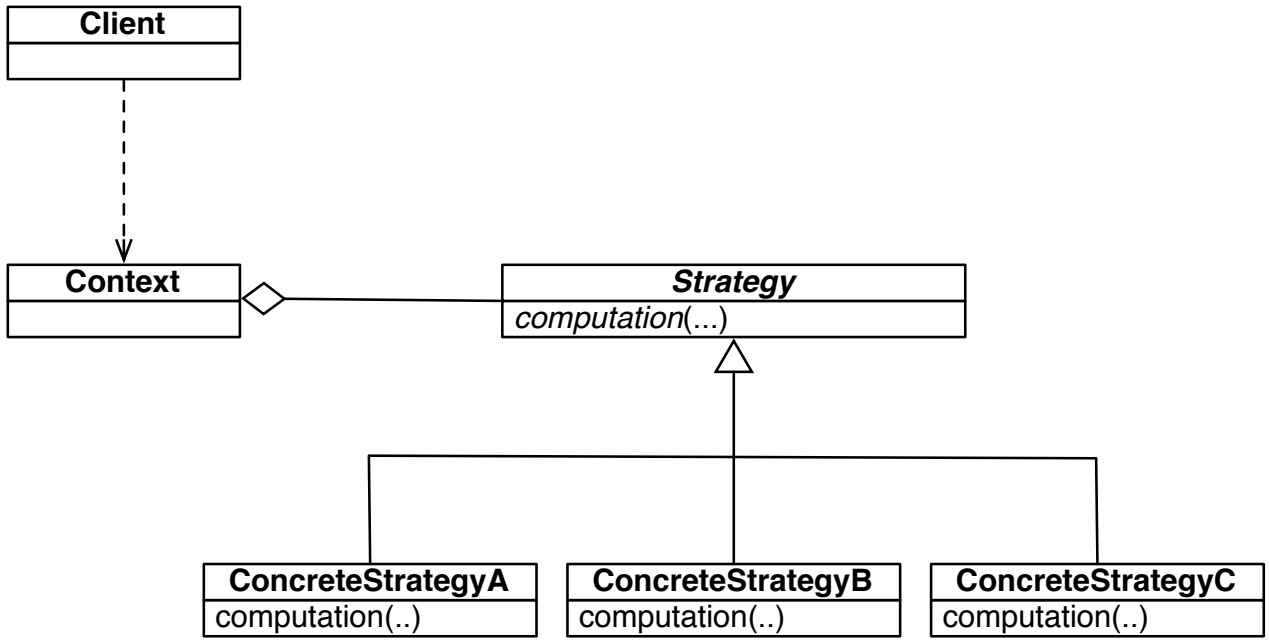
Getting rid of switches

```
switch (v.getMovie().getPriceCode()) {
  case Movie.STANDARD  :
    v_price = 3 euro  * v.daysRented; break;
  case Movie.NEW  :
    v_price = 5 euro  * v.daysRented; break;
  case Movie.CHILDREN  :
    v_price = 2 euro * v.daysRented; break;
 }
```

Switches may cause maintenance headaches!

Inheritance

- Be sure that switches appear at the right place
- Often it is an indication to apply the Strategy pattern
- Separate the choice of an algorithm from the implementation of the algorithm



Refactoring safely

- It is all to easy to make mistakes or introduce bugs during refactoring
- Big refactorings can be really complex
- What best practices can help refactor effectively?

Refactoring

- Do not start refactoring complex code unless you have a good test suite – how else will you know that refactoring hasn't changed a program's behaviour?
- Refactor in the smallest possible steps
- Be sure that you can take a step back: version management
- Test after every step – catching bugs early will make your life a *lot* easier

Why refactor?

- Refactoring may be necessary to improve the software's design – eliminating duplicate code, for example
- Refactoring makes software easier to understand
- Refactoring makes software easier to adapt
- Refactoring makes software easier to test
- Refactoring helps spotting bugs
- Refactoring supports faster programming

When should you refactor?

- Rule of three
  - The first time you write something, just write it
  - The second time, wince at duplicate code
  - The third time, refactor
- Refactor when you add a new feature
- Refactor when you fix a bug
- Refactor when you do a code review

There is no golden rule

Bad code smells

- Fowler identifies a list of *bad code smells*
- These are not bugs, but symptoms in the source code that may indicate bad design, or the need for refactoring
- Whenever you are programming, or doing a code review, knowing about bad code smells can help identify problems, before they become hard to fix

Let us cover a few here and give some examples

Bad smell?

```
temp_c1 = (5/9)*(temp_f1-32);
temp_c2 = (5/9)*(temp_f2-32);
temp_c3 = (5/9)*(temp_f3-32);
avg_temp_c = (temp_c1 + temp_c2 + temp_c3) / 3;
```

Duplicated code

```
temp_c1 = (5/9)*(temp_f1-32);
temp_c2 = (5/9)*(temp_f2-32);
temp_c3 = (5/9)*(temp_f3-32);
avg_temp_c = (temp_c1 + temp_c2 + temp_c3) / 3;
```

Remedy: introduce a method

Long method (with vague name)

```
int doIt()
{
...
// 500 lines of weakly cohesive gibberish
...
}
```

- This usually indicates on method is doing too much
- Weak cohesion
- Remedy: split into smaller, more cohesive methods

Large class (with vague name)

```
public class TheSystem

...
// 1000 weakly cohesive methods
...
```

- This indicates a class is doing too much
- Weak cohesion
- Remedy: split into smaller, more cohesive, loosely coupled classes

Too many parameters

```
Result computeResult(DatabaseConnection db,
                     UserQuery uq,
                     ReceiptPrinter r,
                     UserName name,
                     CustomerReceiptHandler crh,
                     SalesRegister register,
                     CurrencyConverter cc,
                     MovieDatabase imdb,
                     CustomerIdNumber cin,
                     ...)
```

- This usually indicates that a method is trying to do too much
- Weak cohesion

Anti-pattern: shotgun surgery

- Suppose you want to make a change, for example, introduce a new kind of movie to the video store case study
- But to do this, you need to update the existing software in many different places – this is called *shotgun surgery*
- If this is not an unusual modification, the software is poorly designed!

Divergent change

- Suppose we have our Shapes in the CAD/CAM software again
- Over time, we add colours to our Shapes, methods for modifying colours (increasing transparency, brightness, etc.)
- But we also add information about borders – maybe sometimes the border should be dashed or dotted
- As time goes on, each individual Shape object is less and less cohesive
- Remedy: introduce separate classes to handle colours and borders
- Smells like Strategy

Bad smell?

```
switch (v.getMovie().getPriceCode()) {
  case Movie.STANDARD  :
    v_price = 3 euro  * v.daysRented; break;
  case Movie.NEW  :
    v_price = 5 euro  * v.daysRented; break;
  case Movie.CHILDREN  :
    v_price = 2 euro * v.daysRented; break;
 }
```

Switch statements

```
switch (v.getMovie().getPriceCode()) {
  case Movie.STANDARD  :
    v_price = 3 euro  * v.daysRented; break
  case Movie.NEW  :
    v_price = 5 euro  * v.daysRented; break
  case Movie.CHILDREN  :
    v_price = 2 euro * v.daysRented
 }
```

- The responsibility for this computation is in the wrong place
- Remedy: Add a method to the superclass (or introduce a Strategy)

Comments

- Comments are a Good Thing, right?
- This should be a sweet smell, not a bad smell. . .

Comments

- But if a method or objects needs lots of comments, maybe these comments are masking bad design.

```
public void compute() {
\\ This is a really complicated method
\\   that shouldn't be changed.
\\ It took me a long time to get it right,
\\   so DO NOT TOUCH IT!!!!!
```

- This is an excellent candidate for refactoring!

Bad names

- Excessively long identifiers: in particular, the use of naming conventions to provide disambiguation that should be implicit in the software architecture
  - `SystemControllerHandlerFactoryComponent`
- Excessively long identifiers may indicate an overload of responsibilities
- Excessively short identifiers: the name of a variable should reflect its function unless the function is obvious.
  - `z = h.getX() * i.getY() + k.getQ();`
- Excessive use of literals:
  - x = 3.14 * 12 + 13 / 256;

Fowler's refactoring catalogue

Every refactoring is described in a uniform format:

- The refactoring's **name**
- A **summary** of what the refactoring does;
- The **motivation** explaining when to apply the refactoring (and when not to apply it)
- The **mechanics** giving a recipe of how to apply a refactoring
- Finally, **examples** illustrating a refactoring

Example: Inline Temp

Summary: You have a temporary variable that is assigned to once with a simple expression, and the temp is getting in the way of other refactorings

Replace all references to that temp with the expression

```
double basePrice = anOrder.basePrice();
return (basePrice > 1000)
```

This becomes:

```
return (anOrder.basePrice() > 1000);
```

Example: Inline Temp

Mechanics

1. Declare the temp as a constant and compile
   // This tests that it is really only assigned once
2. Find all references to the temp and replace them with the right-hand side of the assignment
3. Compile and test after each change
4. Remove the declaration of the assignment of the temp
5. Compile and test

Example: Inline Temp

Motivation:
(fragment from Fowler)

The only time *Inline Temp* is used on its own is if you find a temp that is assigned the value of a method call. Often the temp isn't doing any harm and you can leave it there. If the temp is getting in the way of other refactorings, such as Extract Method, it's time to inline it.

Inverses

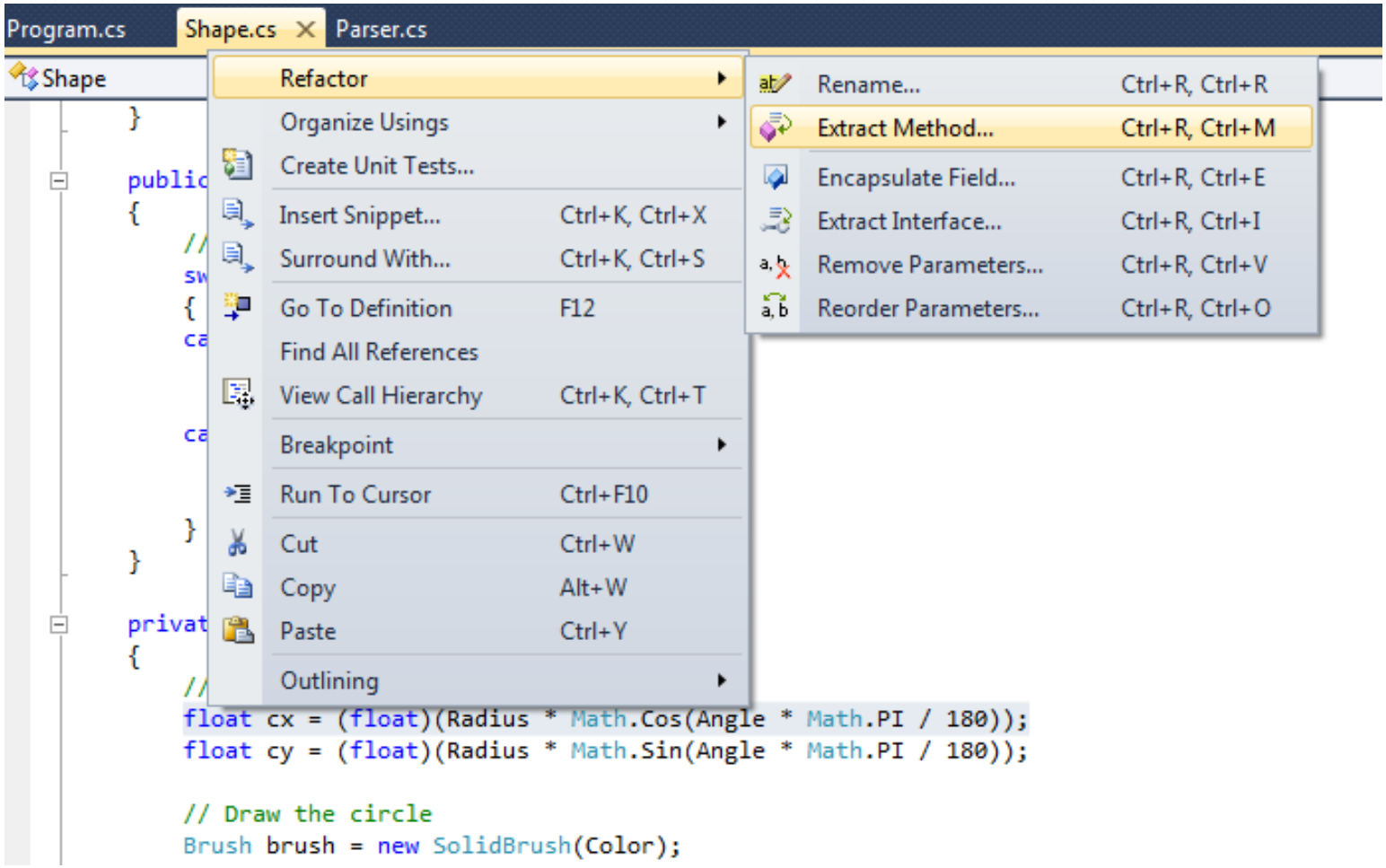
Many refactorings have a reverse refactoring as well. For example, *Introduce explaining variable* is the opposite of *Inline temp*

```
// base price - quantity discount + shipping
return quantity * itemPrice
  - Math.max(0, quantity - 500) * itemPrice * 0.05
  + Math.min(quantity * itemPrice * 0.1, 100);
```

Becomes:

```
double basePrice = quantity * itemPrice;
double discount = ...
double shippingFee = ...
return basePrice - discount + shippingFee;
```

Refactoring with Visual Studio



Refactoring overview

Finally, let us have a look at a quick overview of some other refactorings that Fowler identifies:

- Introduce parameter object
- Moving methods, fields, or attributes
- Splitting classes
- Hiding delegates

You might argue that these refactorings are on the level of *design* rather than *code*

Introduce parameter object

Methods with too many parameters are hard to call:

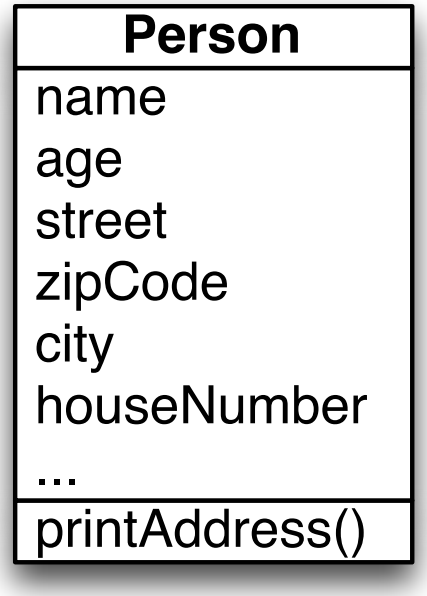
```
processCustomerOnDate(date, isOpen,
  age, hasDiscountCard, customerId)
```

Perhaps it is better to reorganize this into:

```
processCustomerOnDat(dateProfile, customerProfile)
```

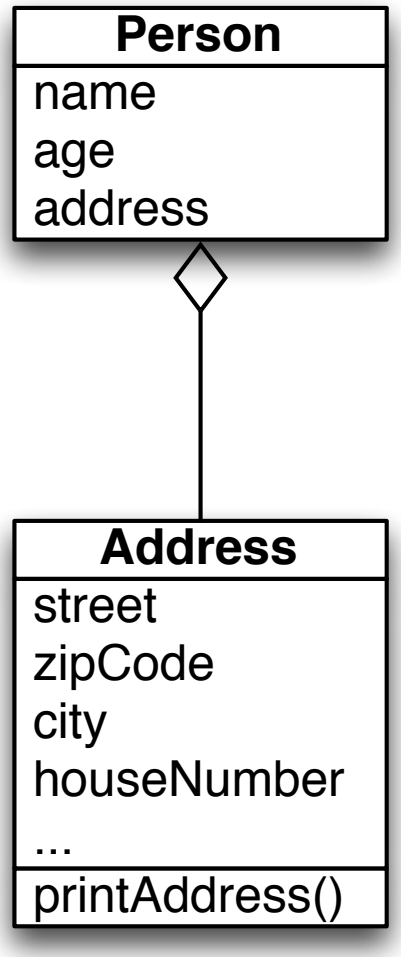
and introduce new objects `dateProfile` and `customerProfile` storing the associated data

Moving code



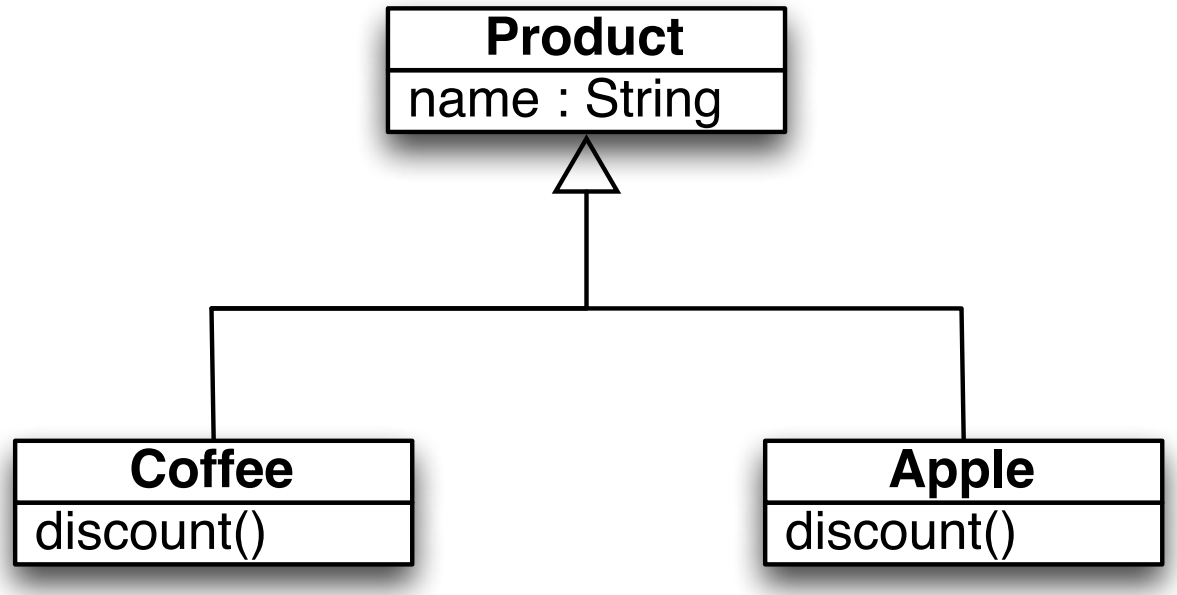
Perhaps this class is becoming too weakly cohesive . . .

Moving code



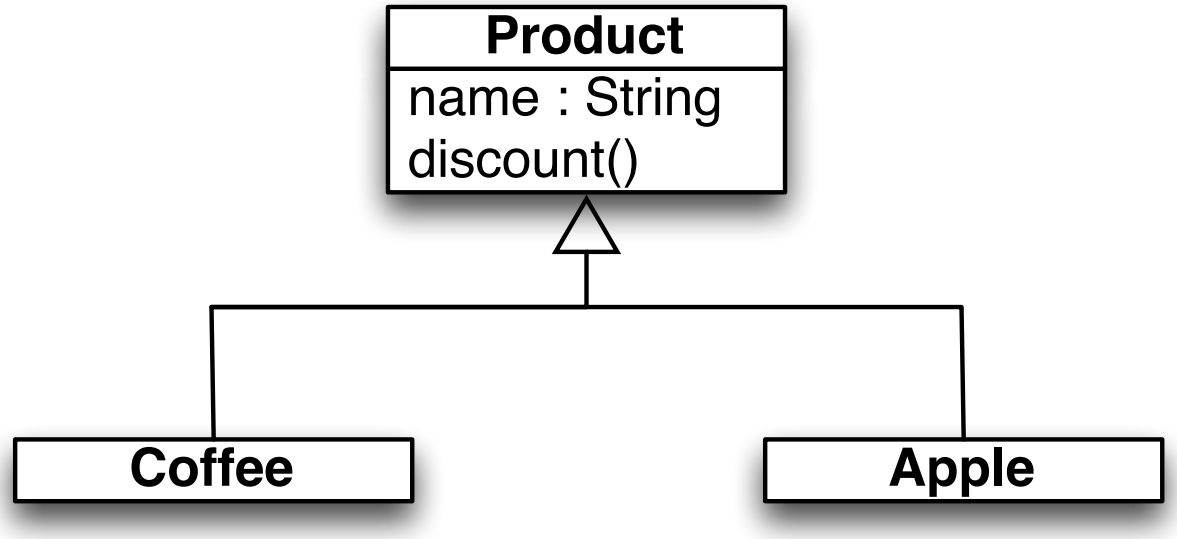
This is an example of the *Extract class* refactoring

Pull up



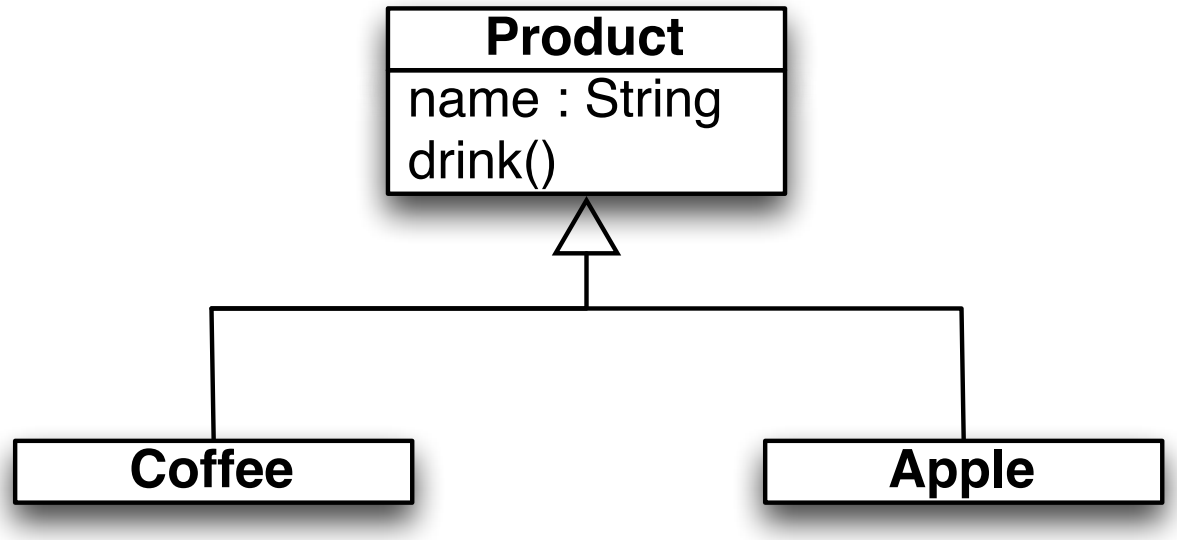
If all subclasses support the `discount` method, shouldn't this be defined in the superclass?

Pull up



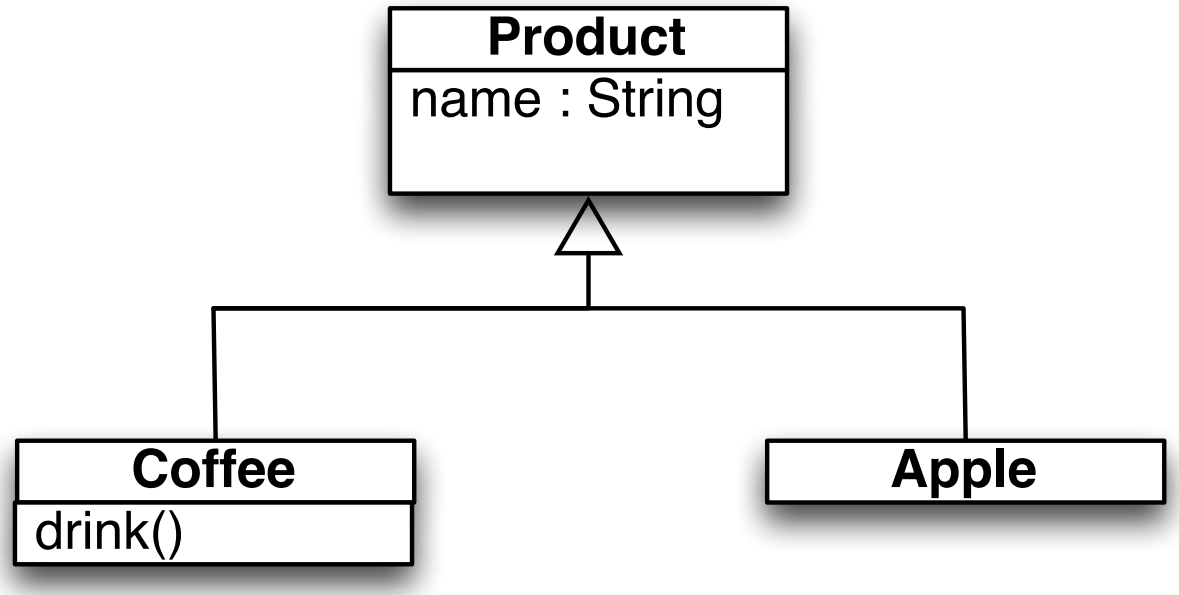
This is the *Pull-up* refactoring

Pull down



Why should you define a dummy implementation for the `drink` method of the `Apple` class?

Pull down



It's better to apply the *Pull-down* refactoring

Refactorings in all shapes and sizes

There are lots of other kinds of refactorings:

- Extract a common superclass when several classes share the same attributes and methods
- Introduce a new subclass of ClassX, to distinguish a specific kind of ClassX
- ...

Refactoring – lessons

- Refactoring helps keep code clean
- Aim for self-documenting code:
  - choose meaningful names
  - lift complex computations into separate methods
  - keep control flow simple
  - ...
- When refactoring, take small steps and test all the time