UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

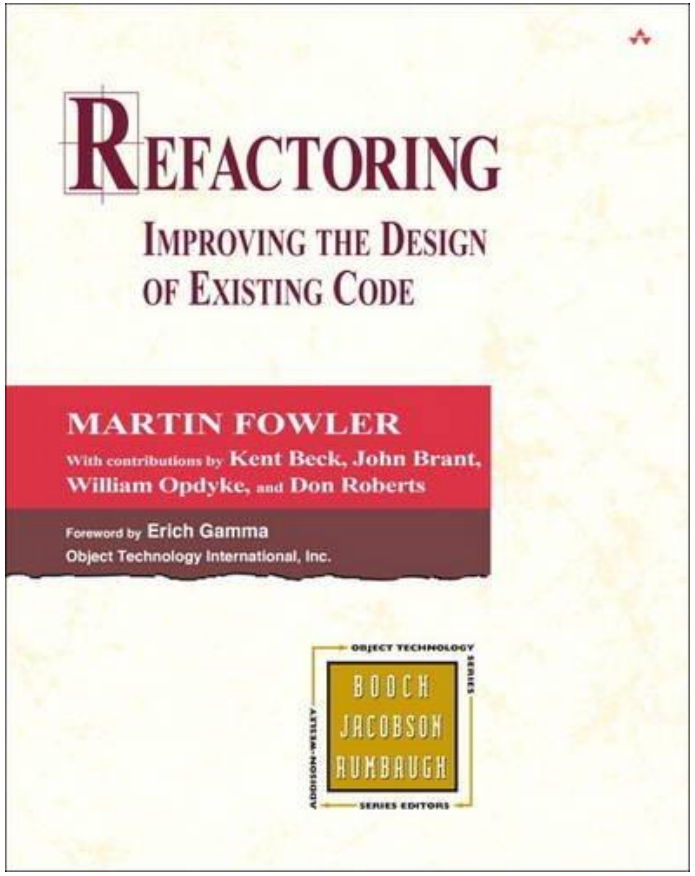
SOFTWARE ENGINEERING

Refactoring (and Bad Smells)

Prof. Sergio Di Martino

Refactoring

- Reference:
  - Refactoring: Improving the Design of Existing Code,
    - by Martin Fowler (et al.),1999, Addison-Wesley
- Fowler, Beck, et al.

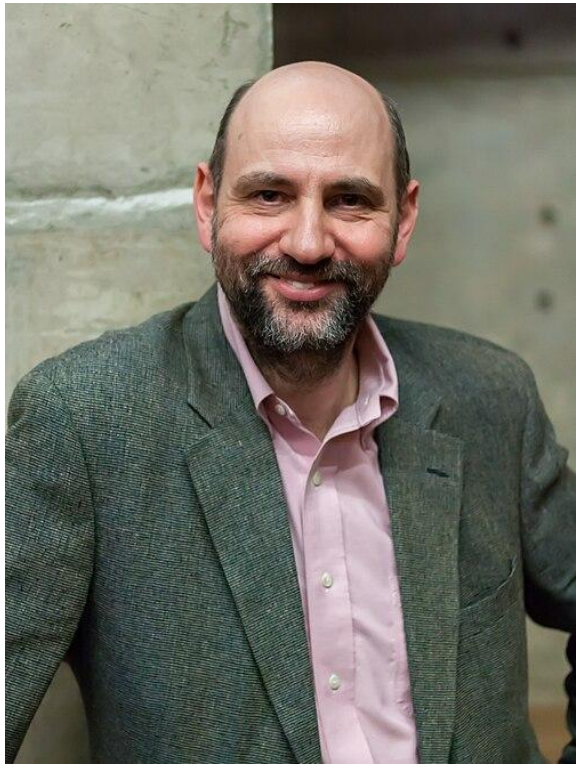


Motivation

“Any fool can write code that a computer can understand.

Good programmers write code that humans can understand.”

Martin Fowler



Defining Refactoring

- 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*."

  Martin Fowler

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

Refactoring Example: Rename variables/functions/classes

```
int n = numberOfRentals();
int n2 = numberOfDaysSinceStart();                    // Nope
double number = n / n2;


int numberOfRentals = numberOfRentals();
int numberOfDays = numberOfDaysSinceStart();          // Better
double averageRentalsPerDay = numberOfRentals / numberOfDays();
```

Refactoring

- The book is a pillar of a Computer Scientist's background
  - Mostly it’s a catalogue of small transformations that you can perform on your code, with motivation, explanation, variations, and examples

    e.g., *Extract interface, Move method, Pull up constructor body*
  - Refactorings often come as duals

    e.g., *Replace inheritance with delegation* and

    Replace delegation with inheritance

Refactoring

- Book has a catalogue of:
  - 22 “bad smells”

    i.e., things to look out for, “anti-patterns”
  - 72 “refactorings”

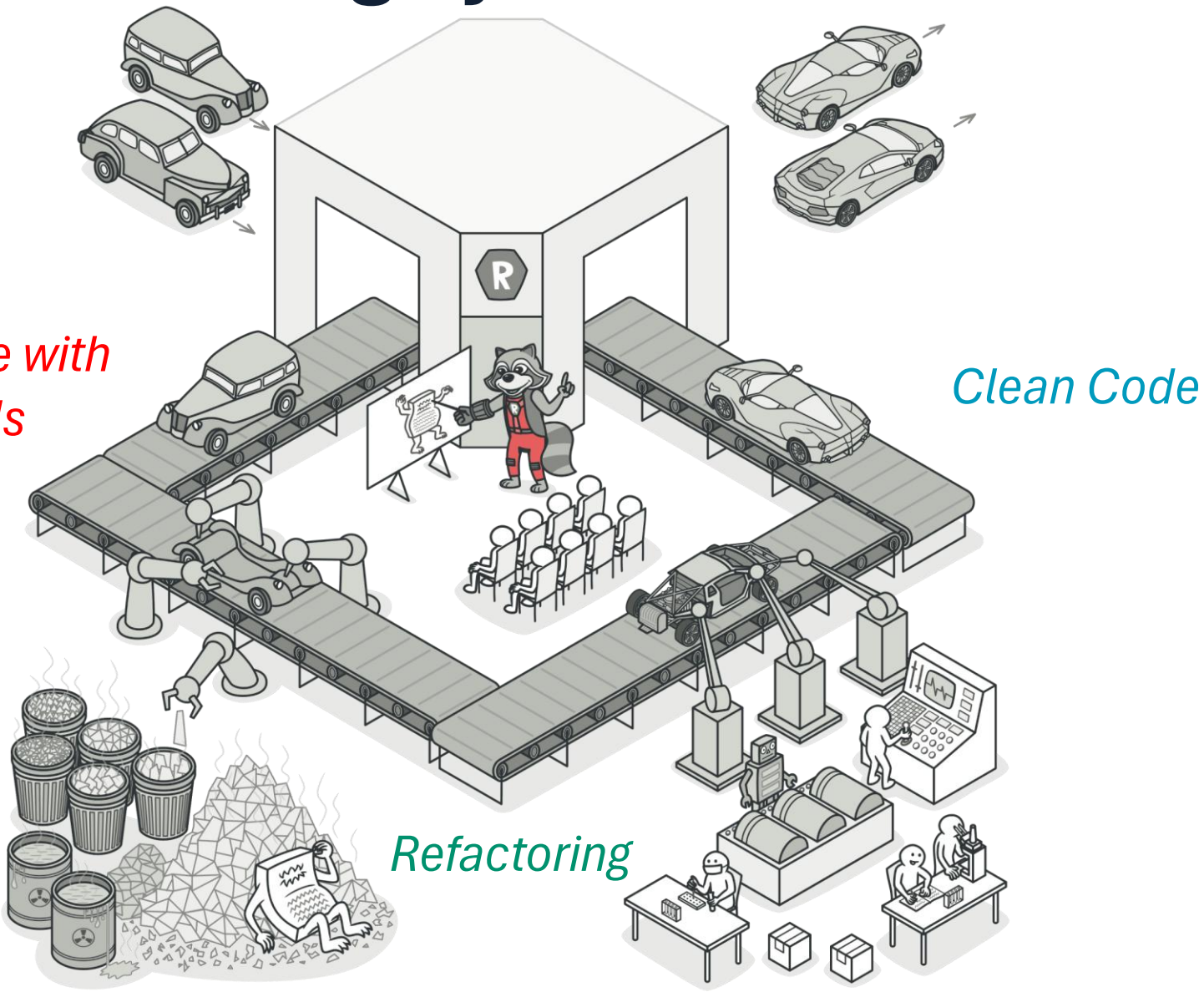
    i.e., what to do when you find them
  - As with GoF book on OODPs, there is overlap between the catalogue items, and the ideas start to blur.
  - We will look at some of the bad smells and what to do about them.

Bad Smells

Code Smells and Refactoring

- “**Code smells**” are indications that there are design problems in the system
- Refactoring techniques fix code smells.
- Most refactoring techniques are fairly straightforward, and there is often really good tool support in the IDEs (now)
- In both the case of code smells and refactoring techniques, there are new ones being “discovered” all the time, so the list of names is fairly long: we’ll look at a small subset of each
- https://refactoring.com/

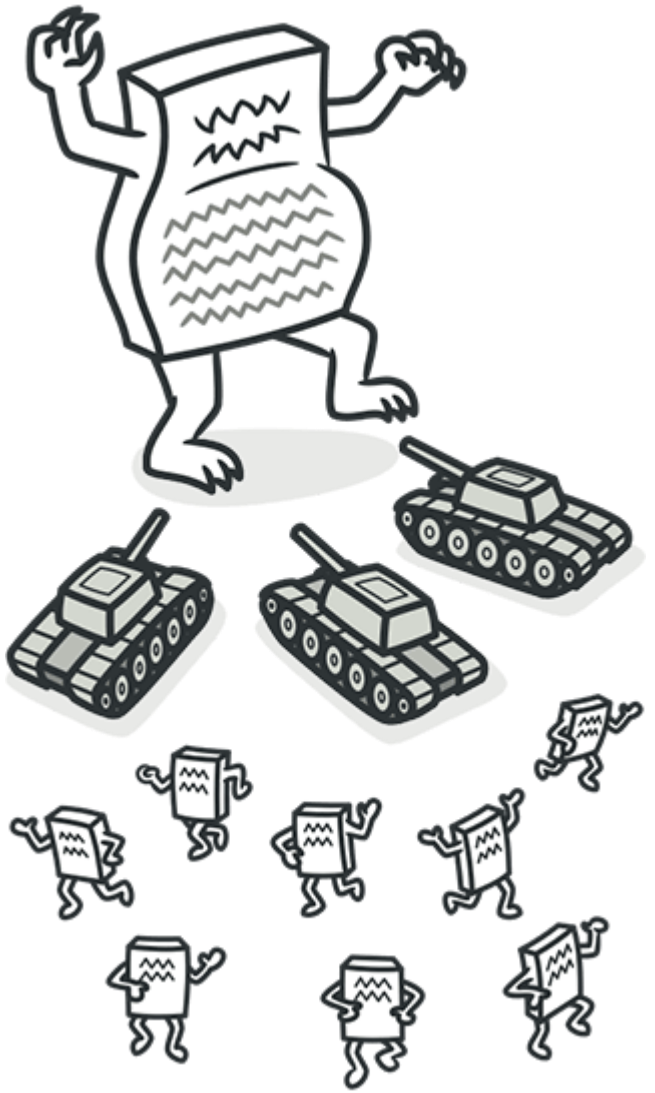
The typical working cycle



Da Refactoring.guru

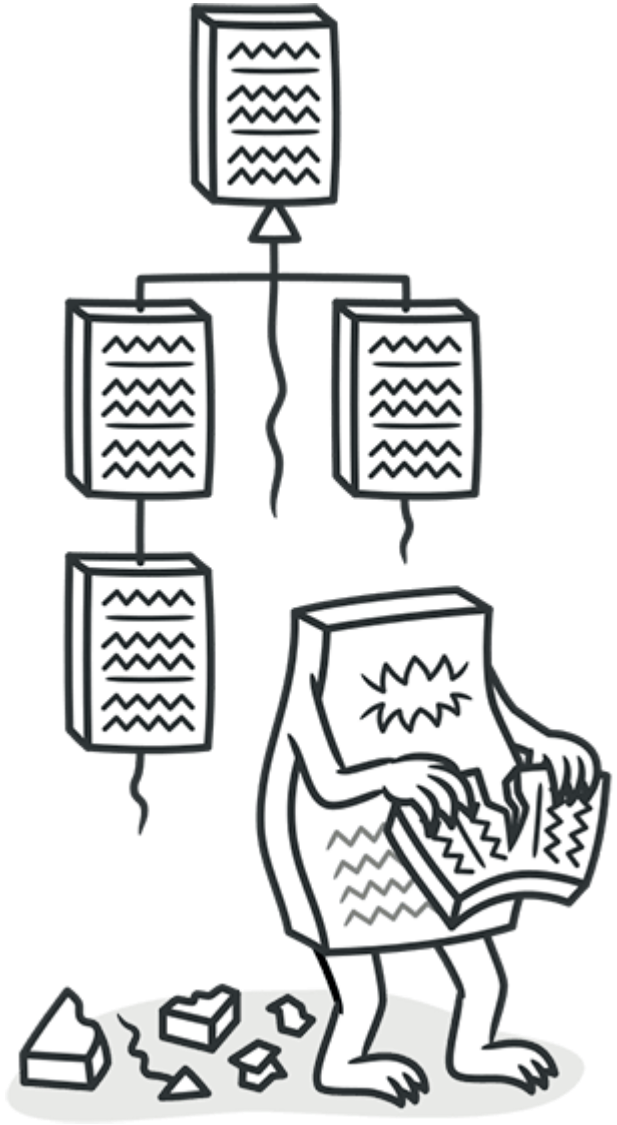
Types of Bad Smells: Bloaters

- Bloaters are code, methods and classes that have increased to such gargantuan proportions that they are hard to work with.
- Usually, these smells do not crop up right away, rather they accumulate over time as the program evolves (and especially when nobody makes an effort to eradicate them).
- Examples
  - Long Method
  - Large Class
  - Primitive Obsession
  - Long Parameter List
  - Data Clumps



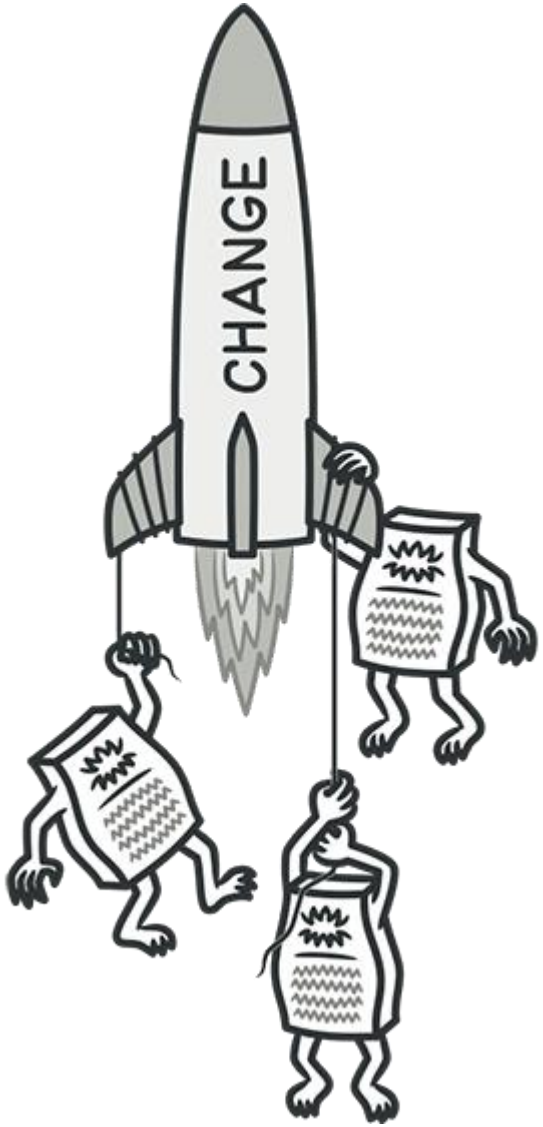
Types of Bad Smells: Object-Orientation Abusers

- All these smells are incomplete or incorrect application of object-oriented programming principles.
  - Alternative Classes with Different Interfaces
  - Refused Bequest
  - Switch Statements
  - Temporary Field



Types of Bad Smells: Change Preventers

- These smells mean that if you need to change something in one place in your code, you have to make many changes in other places too. Program development becomes much more complicated and expensive as a result.
  - Divergent Change
  - Parallel Inheritance Hierarchies
  - Shotgun Surgery



Types of Bad Smells: Dispensables

- A dispensable is something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand.
  - Comments
  - Duplicate Code
  - Data Class
  - Dead Code
  - Lazy Class
  - Speculative Generality

Types of Bad Smells: Couplers

- All the smells in this group contribute to excessive coupling between classes or show what happens if coupling is replaced by excessive delegation.
  - Feature Envy
  - Incomplete Library Class
  - Message Chains
  - Middle Man

Bad Smells and the concept of Technical Debt

Technical Debt

- Whenever code meets functional requirements but is suboptimal or “quick and dirty.” E.g.:
  - “smelly” code
  - inefficient algorithms
  - sloppy design

- Might be fixed at code-review, might generate TODOs or new issues in the issue tracker

- Understanding, measuring and communicating about technical debt is critical in software industry

Debt Analogy

- Is debt always bad?
- Real life is about shipping code that works
- "*Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite – the danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt.*" – Ward Cunningham
- Can you afford to do it without debt?
- Can you afford the interest?
- Can you keep the debt and interest manageable?

Strategic, Intentional Debt

- Cunningham's original definition only considered intentional debt- debt can be "planned debt" or it can be "happened-upon" debt, both can be strategic

- "*Ship now and deal with the consequences later*"

- Examples of intentional or strategic debt:
  - Non-modular design
  - Purposefully too-simple/too-complex implementation
  - Performance indifference
  - Lack of generality or extensibility
  - Lack of scalability

Non-Strategic, Unintentional Debt

- Debt also accumulates unintentionally, no development process is perfect

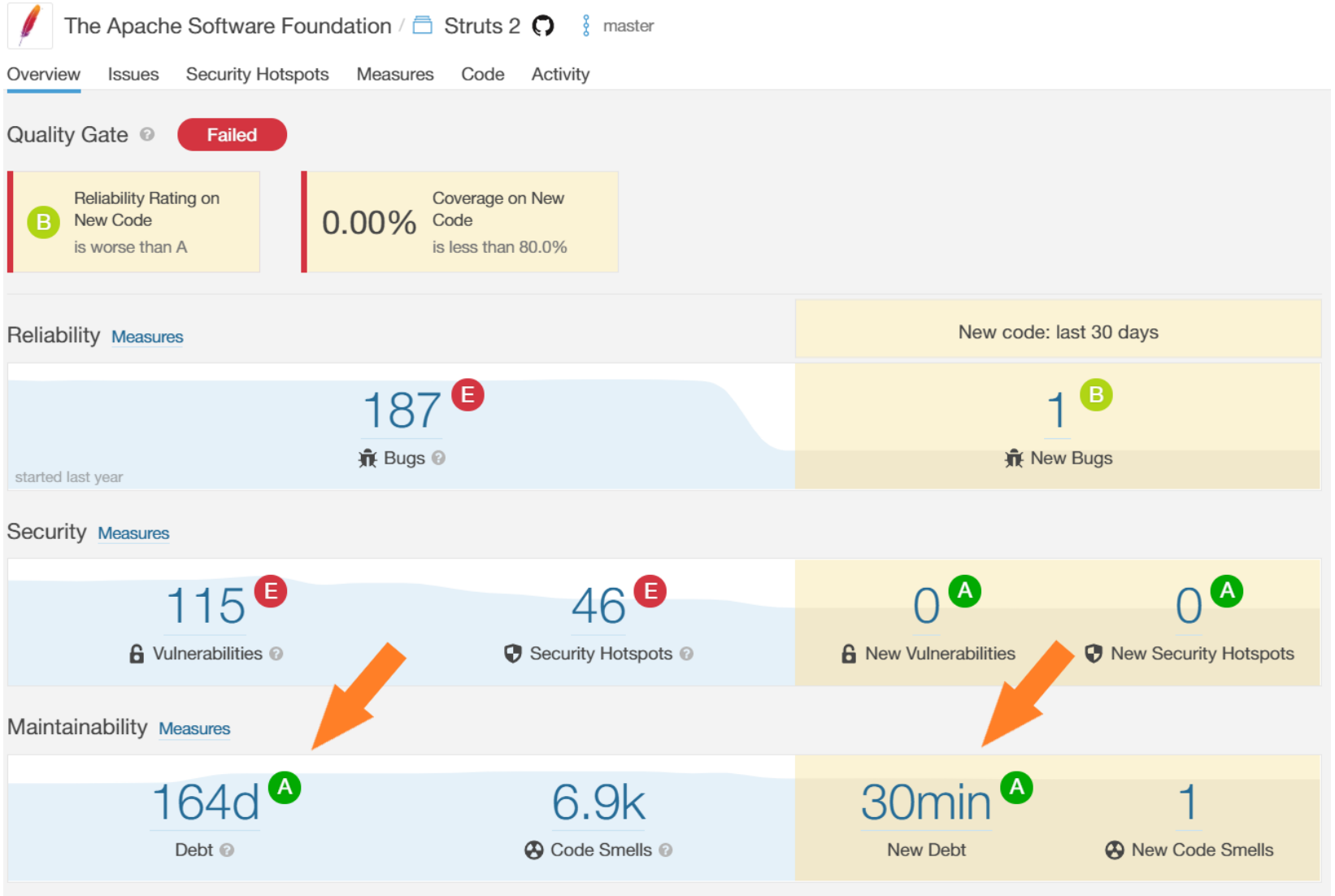
- Examples of unintentional or non-strategic debt:
  - Bad Smells
  - Memory leaks
  - Insufficient test coverage
  - Unintentionally complex implementation
  - Rigid/brittle architecture
  - Performance or scalability bottlenecks
  - Sloppy or difficult to maintain code

Consequences of Debt

- Too much debt => too much time paying interest
- Unpredictability in software planning stage, increased risk of investment
- Slows down future work
- More bugs, more expensive to fix them
- Frustrated, unhappy developers

Measuring Technical Debt

- Some Quality-related tools, like SonarQube, can provide an accurate estimate of the technical debt accomulated within a software repository



Fixing Bad Smells with Refactoring

Refactoring

- Basic metaphor:
  - Start with an existing code base and make it better.
  - Change the internal structure (in-the-small to in-the-medium) while preserving the overall semantics
    - i.e., rearrange the “factors” but end up with the same final “product”
- The idea is that you should improve the code in some significant way. For example:
  - Reducing near-duplicate code
  - Improved cohesion, lessened coupling
  - Improved parameterization, understandability, maintainability, flexibility, abstraction, efficiency, etc …

The Refactoring Cycle

- Basic pattern for refactoring with a working program:
  1. Choose the worse smell
  2. Select a refactoring that will address the smell
  3. Apply the refactoring.

- Select refactorings to improve code each trip through the cycle.

- Program's behavior is not changed.
- Thus, program remains in a working state.
  - So cycle improves the code but retains behavior.

The Refactoring Cycle

- Trickiest part of process: Identify the smell!

- When you start refactoring,
  - best to start with the easy stuff (for example, breaking up large routines or renaming things for clarity).

- This lets you see and fix the remaining problems more easily.

Refactoring and Unit Tests

- Refactoring is strongly dependent on having a good suite of unit tests
- With the unit tests, we can refactor
- Then run the automated tests
  - To verify that the behaviour is indeed preserved.
- Without good unit tests,
  - developers may shy away from refactoring
  - Due to the fear that they may break something.

Why Should You Refactor?

- The reality:
  - Extremely difficult to get the design “right” the first time
  - Hard to fully understand the problem domain
  - Hard to understand user requirements, even if the user does!
  - Hard to know how the system will evolve in *X* years
  - Original design is often inadequate
  - System becomes brittle over time, and more difficult to change
- Refactoring helps you to
  - Manipulate code in a safe environment (behavior preserving)
  - Recreate a situation where evolution is possible
  - Understand existing code

Why Should You Refactor?

- Refactoring improves the design of software
  - Without refactoring the design of the program will decay
  - Poorly designed code usually takes more code to do the same things, often because the code does the same thing in different places
- Refactoring makes software easier to understand
  - In most software development environments, somebody else will eventually have to read your code
- Refactoring helps you find bugs
- Refactoring helps you program faster

Typical Refactorings

Typical Refactorings

Class Refactorings	Method Refactorings	Attribute Refactorings
add (sub)class to hierarchy	add method to class	add variable to class
rename class	rename method	rename variable
remove class	remove method	remove variable
Extract class	push method down	push variable down
	push method up	pull variable up
	add parameter to method	create accessors
	move method	abstract variable
	extract code in new method	
	replace parameter with method	

Extract Method

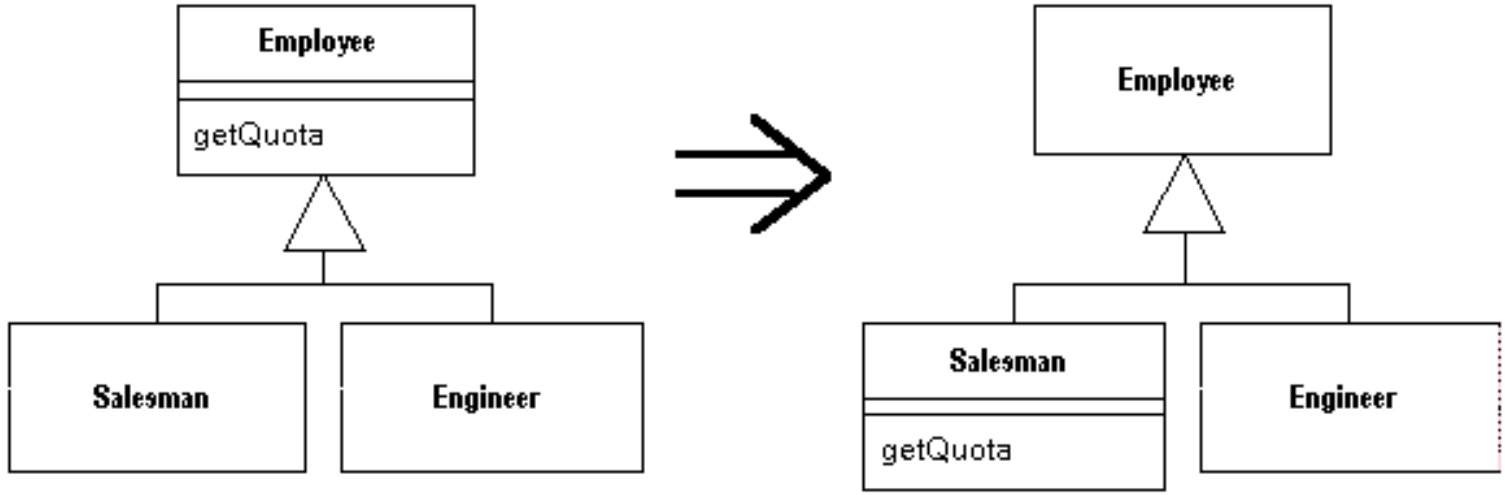
```
void printOwing() {
  printBanner();
  //print details
  System.out.println ("name:       " + _name);
  System.out.println ("amount     " + getOutstanding());
  }
```

```
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}


void printDetails (double outstanding) {
  System.out.println ("name: " + _name);
  System.out.println ("amount " + outstanding);
}
```

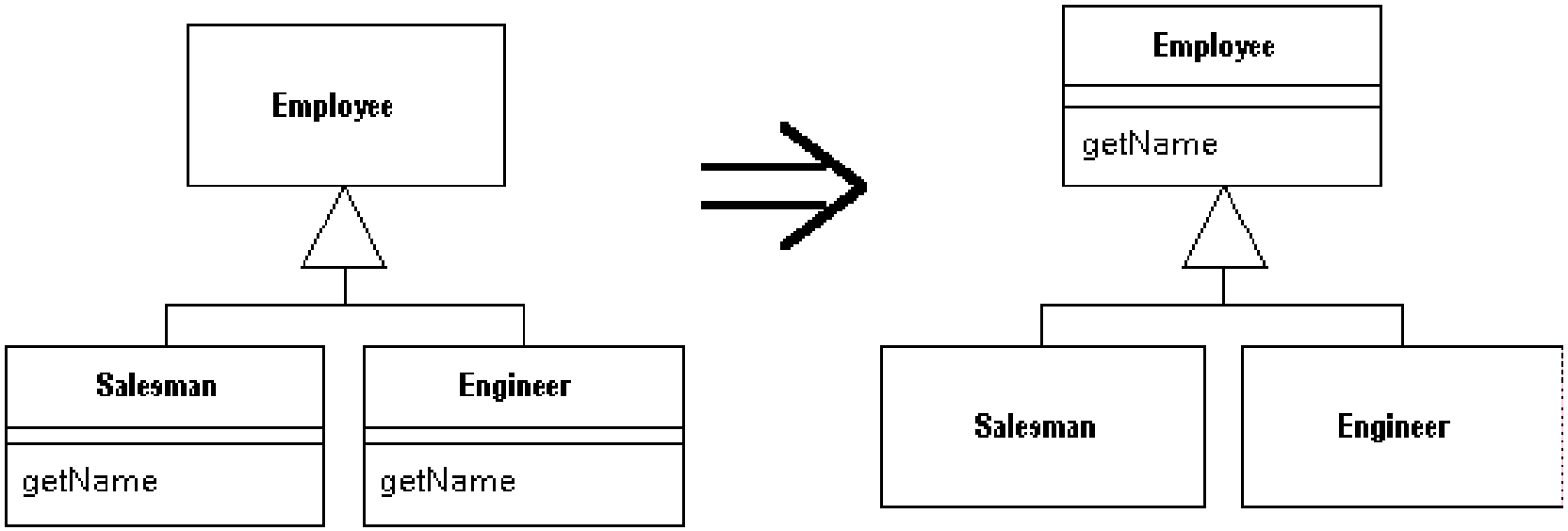
Push method down

- Behavior of a superclass is relevant only for some of its subclasses.
- Move it to those subclasses.



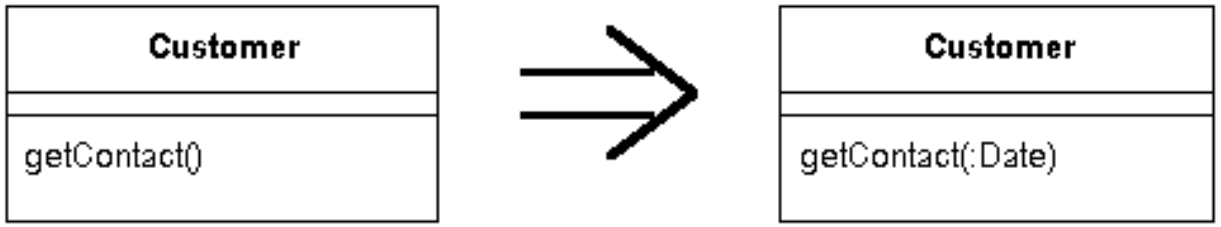
Push method up

- You have methods with identical results on subclasses.
- Move them to the superclass



Add parameter to method

- A method needs more information from its caller.
- Add a parameter for an object that can pass on this information.



Replace Parameter with Method

```
int basePrice = _quantity * _itemPrice;


discountLevel = getDiscountLevel();


double finalPrice = discountedPrice
  (basePrice, discountLevel);
```

```
int basePrice = _quantity * _itemPrice;

double finalPrice =
  discountedPrice (basePrice);
```

//An object invokes a method, then passes the result as a parameter for a method. The receiver can also invoke this method.

From Website

- https://refactoring.guru/
- https://github.com/nerdschoolbergen/code-smells/tree/master/assignment/src/main/java/nerdschool/bar

Duplicated Code

- If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them
- The simplest duplicated code problem is when you have the same expression in two methods of the same class
  - Perform *Extract Method* and invoke the code from both places
- Another common duplication problem is having the same expression in two sibling subclasses
  - Perform Extract Method in both classes then Pull Up Field
- If you have duplicated code in two unrelated classes, consider using Extract Class in one class and then use the new component in the other

Long Method

- The longer a procedure is, the more difficult it is to understand
- Object programs live best and longest with short methods.
- Nearly all of the time, all you have to do to shorten a method is *Extract Method*
- If you try to use Extract Method and end up passing a lot of parameters, you can often create a new Class encapsulating the parameters

Large Class

- When a class is trying to do too much, it often shows up as too many instance variables. When a class has too many instance variables, duplicated code cannot be far behind
- A class with too much code is also a breeding ground for duplication
- In both cases *Extract Class* and *Extract Subclass* will work

Long Parameter List

- Old school: pass in everything as parameters. Was better than global data.
- With objects you don't need to pass in everything the method needs, instead you pass in enough so the method can get to everything it needs
- This is goodness, because long parameter lists are hard to understand, because they are inconsistent and difficult to use, because you are forever changing them as you need more data
- Use *Replace Parameter with Method* when you can get the data in one parameter by making a request of an object you already know about
- Use *Introduce Parameter Object*

Long Parameter List

- Problem: You have a method that requires a long parameter list
  - You may have a group of parameters that go naturally together
  - If so, make them into an object
- Example: Replace

```
public void marry(String name, int age,
                                boolean gender, String name2, int age2,
boolean gender2) {...}
```

with

```
public void marry(Person person1, Person person2) {...}
```

Divergent Change

- Divergent change occurs when one class is commonly changed in different ways for different reasons
  - If you're doing this, you're almost certainly violating the principles of one key abstraction and separation of concerns, and you should refactor your code
- To clean this up you identify everything that changes for a particular cause and use *Extract Class* to put them all together

Shotgun Surgery

- This situation occurs when every time you make a kind of change, you have to make a lot of little changes to a lot of different classes
  - The same rate of change in different objects, particularly if they are disconnected.
- A conceptually simple change that requires modification to code in many places. The result of Copy&Paste Programming
- When the changes are all over the place they are hard to find, and its easy to miss an important change.
- You want to use *Move Method* and *Move Field* to put all the changes in a single class
  - If no current class looks like a good candidate then create one

Feature Envy

- “*A classic [code] smell is a method that seems more interested in a class other than the one it is in. The most common focus of the envy is the data*."

Fowler.

- “Feature envy” is when a method makes heavy use of data and methods from another class
  - Use *Move Method* to put it in the more desired class
- Sometimes only part of the method makes heavy use of the features of another class
  - Use *Extract Method* to extract those parts that belong in the other class

Data Clumps

- Often you will see the same three or four data items together in lots of places:
  - Fields in a couple of classes
  - Parameters in many method signatures
- Bunches of data that hang around together really ought to be made into their own object
  - E.g.: x,y → Point
- The first step is to look for where the clumps appear as fields and use *Extract Class* to turn the clumps into an object
- For method parameters use *Introduce Parameter Object* or *Preserve Whole Object* to slim them down

Primitive Obsession

- People new to objects are sometimes reluctant to use small objects for small tasks, such as money classes that combine numbers and currency ranges with an upper and lower, and special strings such as telephone numbers and ZIP codes
- Many programmers are reluctant to introduce “little” classes that represent things easily represented by primitives—telephone numbers, zip codes, money amounts, ranges (variables with upper and lower bounds)
- If your primitive needs any additional data or behavior, consider turning it into a class
  - For example, you may want to format your primitive in a special way, such as (215)898-0587 or 19104-6389

Type Tests

- Most times when you see a switch statement on a type you should consider polymorphism
  - “The essence of polymorphism is that instead of asking an object what type it is and then invoking some behavior based on the answer, you just invoke the behavior. The object, depending on its type, does the right thing.”
- Use *Extract Method* to extract the switch statement and then *Move Method* to get it into the class where the polymorphism is needed

Parallel Inheritance Hierarchies

- Is really a special case of shotgun surgery
- In this case every time you make a subclass of one class, you have to make a subclass of another
- *There are two ways to go. To remove the parallel: refactor either or both hierarchies until their members are congruent, then collapse pairwise. To remove duplication between the parallels: define distinct responsibilities refined by each hierarchy and relocate methods as appropriate.* -- WardCunningham
- If you use *Move Method* and *Move Field*, the hierarchy on the referring class disappears

Lazy Class

- Each class you create costs money and time to maintain and understand
- A class that is not carrying its weight should be eliminated
- If you have subclasses that are not doing enough try to use *Collapse Hierarchy* and nearly useless components should be subjected to *Inline Class*

Speculative Generalization

- You get this smell when someone says "I think we need the ability to do this someday" and you need all sorts of hooks and special cases to handle things that are not required
- This smell is easily detected when the only users of a class or method are test cases
- If you have abstract classes that are not doing enough then use *Collapse Hierarchy*
- Unnecessary delegation can be removed with *Inline Class*
- Methods with unused parameters should be subject to *Remove Parameter*
- Methods named with odd abstract names should be repaired with *Rename Method*

Temporary Field

- Sometimes you will see an object in which an instance variable is set only in certain circumstances
- This can make the code difficult to understand because we usually expect an object to use all of its variables
- Use *Extract Class* to create a home for these orphan variables by putting all the code that uses the variable into the component

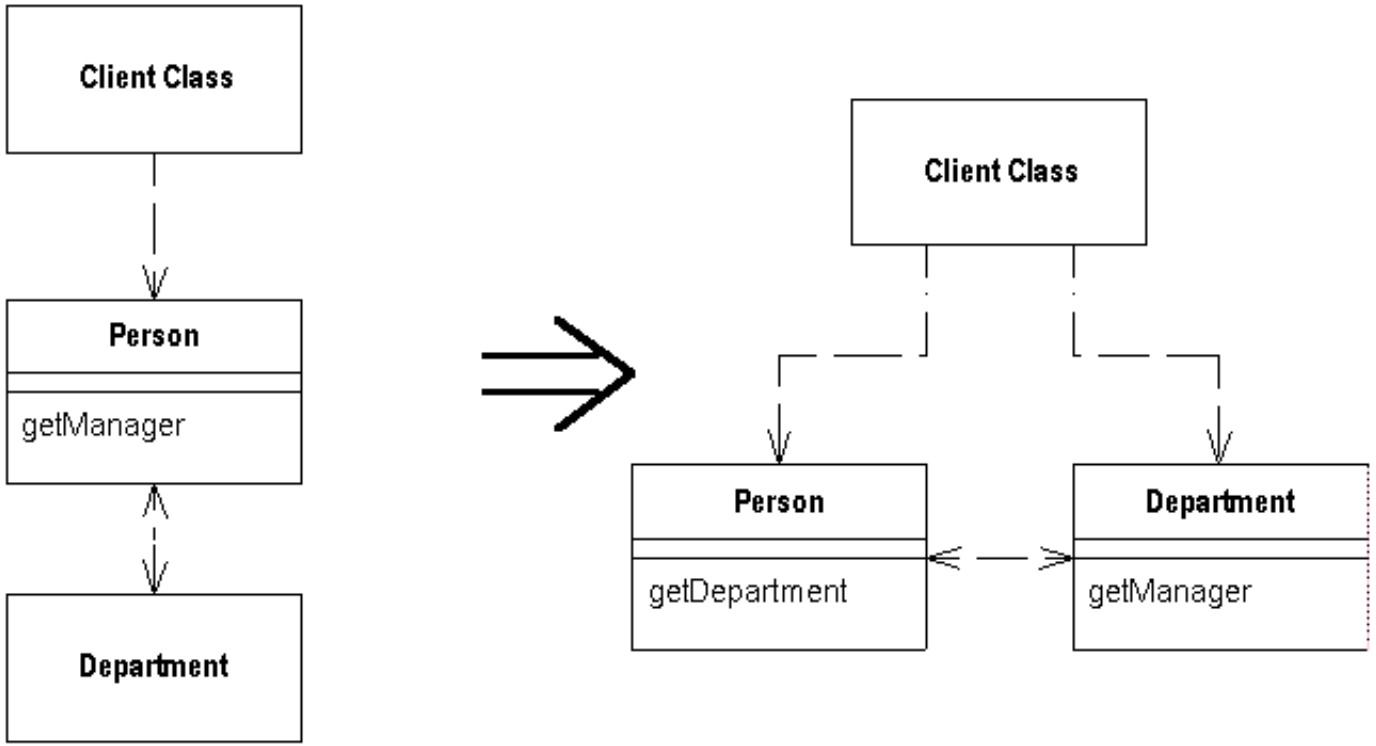
Message Chains

- Message chains occur when you see a client that asks one object for another object, which the client then asks for yet another object, which the client then asks for yet another object, etc.
  - `intermediate.getProvider().doSomething()`
- or:
  - ```
    ch = vehicle->getChassis();
    body = ch->getBody();
    shell = body->getShell();
    material = shell->getMaterial();
    props = material->getProperties();
    color = props->getColor();
    ```
- Navigating this way means the client is structured to the structure of the hierarchy
    ```

Middle Man

- One the major features of Objects is encapsulation
- Encapsulation often comes with delegation
- Sometimes delegation can go to far
- For example if you find half the methods are delegated to another class it might be time to use Remove Middle Man and talk to the object that really knows what is going on
- If only a few methods are not doing much, use Inline Method to inline them into the caller
- If there is additional behavior, you can use Replace Delegation with Inheritance to turn the middle man into a subclass of the real object

Remove Middle Man



Esercizi

Rifattorizzare il seguente codice

```
void printProperties(IList users)
{
  for (int i = 0; i < users.size(); i++)
  {
    string result = "";
    result += users.get(i).getName();
    result += " ";
    result += users.get(i).getAge();
    Console.WriteLine(result);

    // ...
  }
}
```

Rifattorizzare il seguente codice

```
public bool Login(string username, string password){
    var user = userRepo.GetUserByUsername(username);

    if(user == null)
        return false;

    if (loginValidator.IsValid(user, password))
        return true;

    user.FailedLoginAttempts++;

    if (user.FailedLoginAttempts >= 3)
        user.LockedOut = true;

    return false;
}
```

Final Remarks

Refactoring Malapropism (by Martin Fowler)

- [...] the term "refactoring" is often used when it's not appropriate. If somebody talks about a system being broken for a couple of days while they are refactoring, you can be pretty sure they are not refactoring. If someone talks about refactoring a document, then that's not refactoring. Both of these are restructuring.
- I see refactoring as a very specific technique to do the more general activity of restructuring. Restructuring is any rearrangement of parts of a whole. It's a very general term that doesn't imply any particular way of doing the restructuring.
- Refactoring is a very specific technique, founded on using small behavior-preserving transformations (themselves called refactorings).
- If you are doing refactoring your system should not be broken for more than a few minutes at a time, and I don't see how you do it on something that doesn't have a well defined behavior.