DESIGN PATTERNS

Course 8

PREVIOUS COURSE CONTENT

- Behavioral patterns
  - Chaim of responsibility
  - Strategy
- Partitioning patterns
  - Filter
  - Composite object
  - Read-only Interface

CURRENT CURSE CONTENT

- Refactoring

SOFTWARE DESIGN PRINCIPLES

Open Close Principle

Software entities should be *open for extension* but *closed for modifications*.

- Design classes and packages so their functionality can be extended without modifying the source code

The Object Manifesto

Delegation and encapsulation:

- "Don't do anything you can push off to someone else."
- "Don't let anyone else play with you."

Programmer manifesto

Once and only once

- Avoiding writing the same code more than once

GOOD SIGNS OF OO THINKING

Short methods

- Simple method logic

Few instance variables

Clear object responsibilities

- State the purpose of the class in one sentence
- No super-intelligent objects
- No manager objects

SOME PRINCIPLES

The Dependency Inversion Principle

- Depend on abstractions, not concrete implementations
  - Write to an interface, not a class

The Interface Segregation Principle

- Many small interfaces are better than one “fat” one

The Acyclic Dependencies Principle

- Dependencies between package must not form cycles.
  - Break cycles by forming new packages

The Common Closure Principle

- Classes that change together, belong together
  - Classes within a released component should share common closure. That is, if one needs to be changed, they all are likely to need to be changed.

The Common Reuse Principle

- Classes that aren’t reused together don’t belong together
  - The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.

REFACTORING

The process of *changing a software system* in such a way that it *does not alter the external behaviour* of the code, yet *improves its internal structure*.

Refactoring is not just arbitrary restructuring

- Code must still work
- Small steps only so the semantics are preserved (i.e. not a major re-write)
- Unit tests to prove the code still works
- Code is
  - More loosely coupled
  - More cohesive modules
  - More comprehensible

WHAT IS NOT REFACTORING

- Adding new functionality is not refactoring
- Optimisation is not refactoring
- Changing code that does not compile is not refactoring
  (what would be the behaviour?)

WHEN TO REFACTOR

You should refactor:

- Any time that you see a better way to do things
  - "Better" means making the code easier to understand and to modify in the future
- You can do so without breaking the code
  - Unit tests are essential for this

You should not refactor:

- Stable code that won't need to change
- Someone else's code
  - Unless the other person agrees to it or it belongs to you
  - Not an issue in Agile Programming since code is communa

WHEN TO REFACTOR

When should you refactor?

- Any time you find that you can improve the design of existing code
- You detect a “bad smell” (an indication that something is wrong) in the code

When can you refactor?

- You should be in a supportive environment (agile programming team, or doing your own work)
- You are familiar with common refactorings
- Refactoring tools also help
- You should have an adequate set of unit tests

REFACTORING PROCESS

Make a small change

- a single refactoring

Run all the tests to ensure everything still works

If everything works, move on to the next refactoring

If not, fix the problem, or undo the change, so you still have a working system

CODE SMELLS EXAMPLES

If it stinks, change it

- Code that can make the design harder to change

Examples:

- Duplicate code
- Long methods
- Big classes
- Big switch statements
- Long navigations (e.g., a.b().c().d())
- Lots of checking for null objects
- Data clumps (e.g., a Contact class that has fields for address, phone, email etc.) - similar to non-normalized tables in relational design
- Data classes (classes that have mainly fields/properties and little or no methods)
- Un-encapsulated fields (public member variables)

A REFACTORING SHOULD BE USEFUL IF . . .

code is **duplicated**

a routine is **too long**

a loop is too long or **deeply nested**

a class has poor **cohesion**

a class uses too much **coupling**

inconsistent level of **abstraction**

too many **parameters**

to **compartmentalize** changes (change one place → must change others)

to modify an **inheritance hierarchy** in parallel

to **group related data** into a class

a **"middle man"** object doesn't do much

poor encapsulation of data that should be private

a **weak subclass** doesn't use its inherited functionality

a class contains **unused code**

SOME TYPES OF REFACTORING

refactoring to fit design **patterns**

renaming (methods, variables)

extracting code into a method or module

splitting one method into several to improve cohesion and readability

changing method **signatures**

performance **optimization**

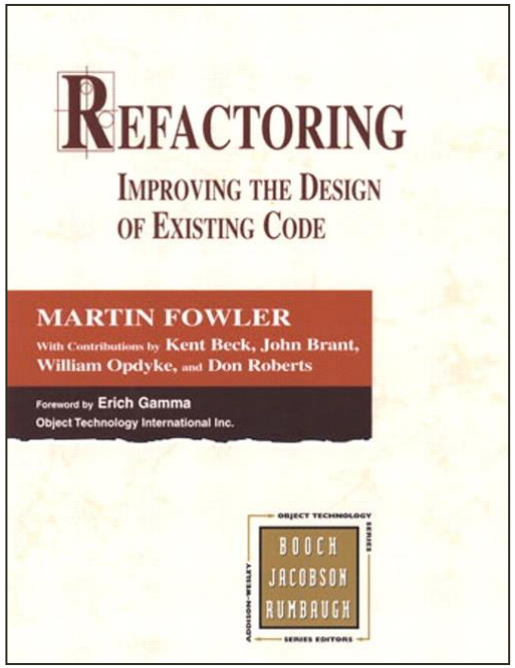
moving statements that semantically belong together near each other

naming (extracting) "magic" **constants**

exchanging idioms that are risky with safer alternatives

clarifying a statement that has evolved over time or is unclear

- See also http://www.refactoring.org/catalog/

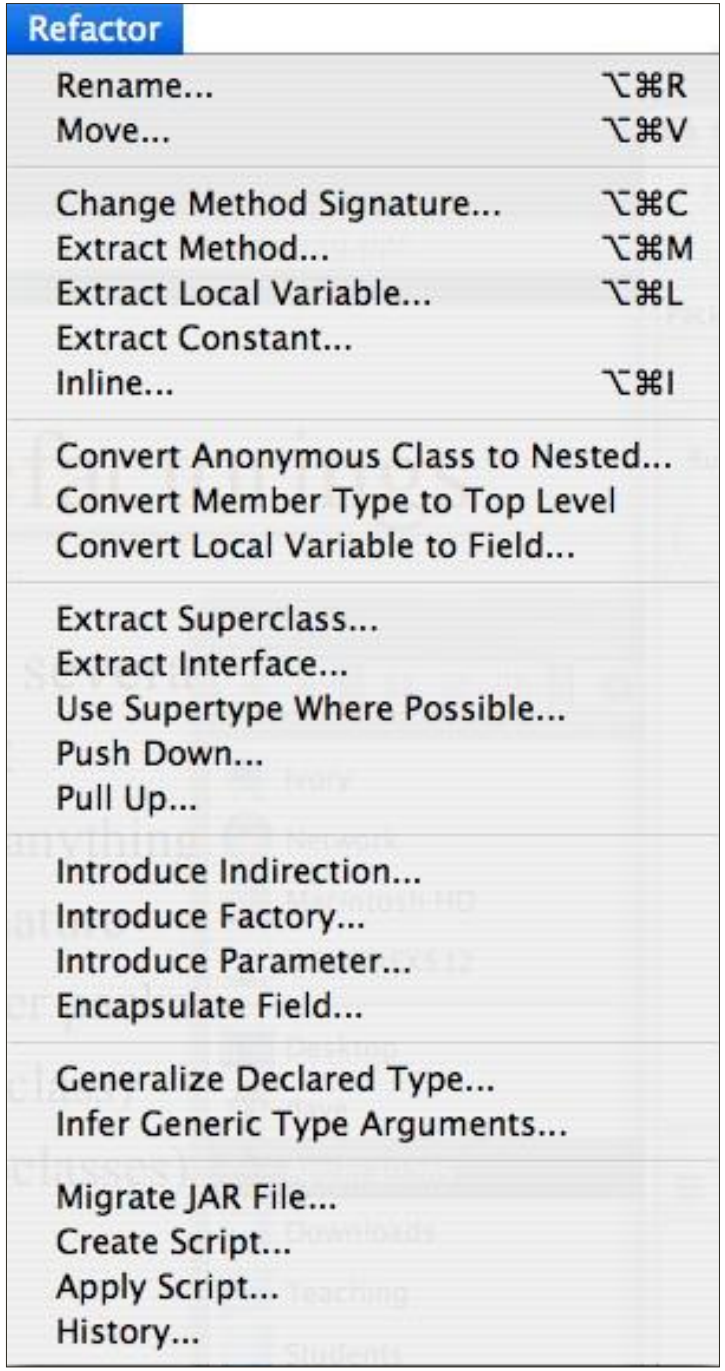


HOW TO REFACTOR

Manualy

Refactoring tool

Eclipse (and some other IDEs) provide significant support for refactoring



WAY REFACTORING?

To improve the software design
- Makes the program easier to change

To make the software easier to understand
- Write for people, not the compiler
- Understand unfamiliar code

To help find bugs
- Refactor while debugging to clarify the code

Some argue that good design does not lead to code needing refactoring but:
- It is extremely difficult to get the design right the first time
- Original design is often inadequate
- You may not understand user requirements, if he does!

Refactoring helps you to:
- Manipulate code in a safe environment
- Understand existing code

REFACTORING STEPS

Save / **backup** / checkin the code before you mess with it.

- If you use a well-managed version control repo, this is done.

Write **unit tests** that verify the code's external correctness.

- They should pass on the current poorly designed code.
- Having unit tests helps make sure any refactor doesn't break existing behavior (regressions).

Analyze the code to decide the **risk** and benefit of refactoring.

- If it is too risky, not enough time remains, or the refactor will not produce enough benefit to the project, don't do it.

REFACTORING STEPS

Refactor the code.

- Some unit tests may break. Fix the bugs.
- Perform functional and/or integration testing. Fix any issues.

Code review the changes.

Check in your refactored code.

- Keep each refactoring **small**; refactor one issue / unit at a time.
  - helps isolate new bugs and regressions
- Your checkin should contain *only* your refactor.
  - NOT other changes such as adding features, fixing unrelated bugs.
  - Do those in a separate checkin.
    - (Resist temptation to fix small bugs or other tweaks; this is dangerous.)

EXTRACT METHOD

You have a code fragment that can be grouped together.

Turn the fragment into a method whose name explains the purpose of the 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); }
```

inverse of Inline Method

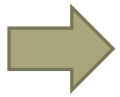
INLINE METHOD

A method's body is just as clear as its name.

Put the method's body into the body of its callers and remove the method.

```
int getRating() {
        return (moreThanFiveLateDeliveries()) ? 2 : 1;
}

 boolean moreThanFiveLateDeliveries() {
         return _numberOfLateDeliveries > 5;
}
```



```
int getRating() {
      return (_numberOfLateDeliveries > 5) ? 2 : 1;
}
```

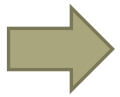
inverse of Exact Method

RENAME METHOD

The name of a method does not reveal its purpose.

Change the name of the method.

```
class Customer {
        double getInvcdtlmt();
}
```

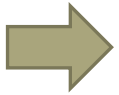


```
class Customer {
      double getInvoiceableCreditLimit();
}
```

REMOVE PARAMETER

A parameter is no longer used by the method body.

Remove it.

Customer getContact(Date)  Customer getContact()

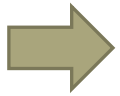
inverse of Add Parameter

Naming: In IDEs this refactoring is usually done as part of "Change Method Signature"

ADD PARAMETER

A method needs more information from its caller.

Add a parameter for an object that can pass on this information.

Customer getContact()  Customer getContact(data)

inverse of Remove Parameter

Naming: In IDEs this refactoring is usually done as part of "Change Method Signature"

EXTRACT CLASS

You have one class doing work that should be done by two.

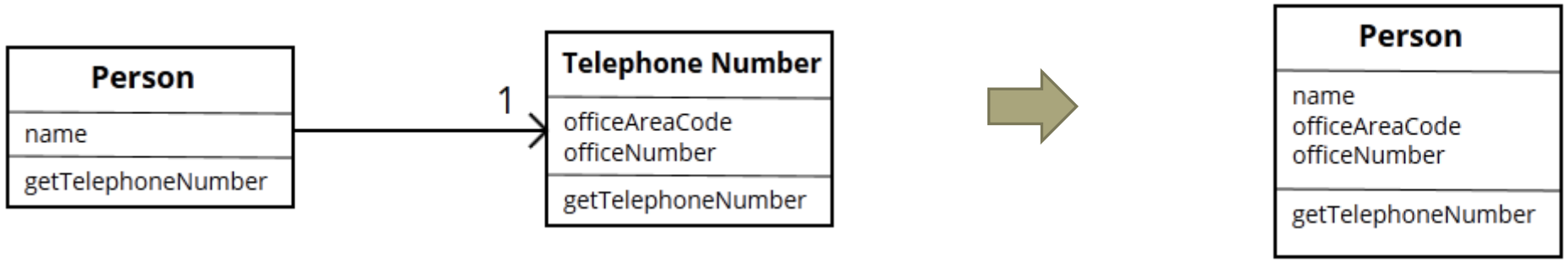
Create a new class and move the relevant fields and methods from the old class into the new class.

inverse of Inline Class

INLINE CLASS

A class isn't doing very much.

Move all its features into another class and delete it.

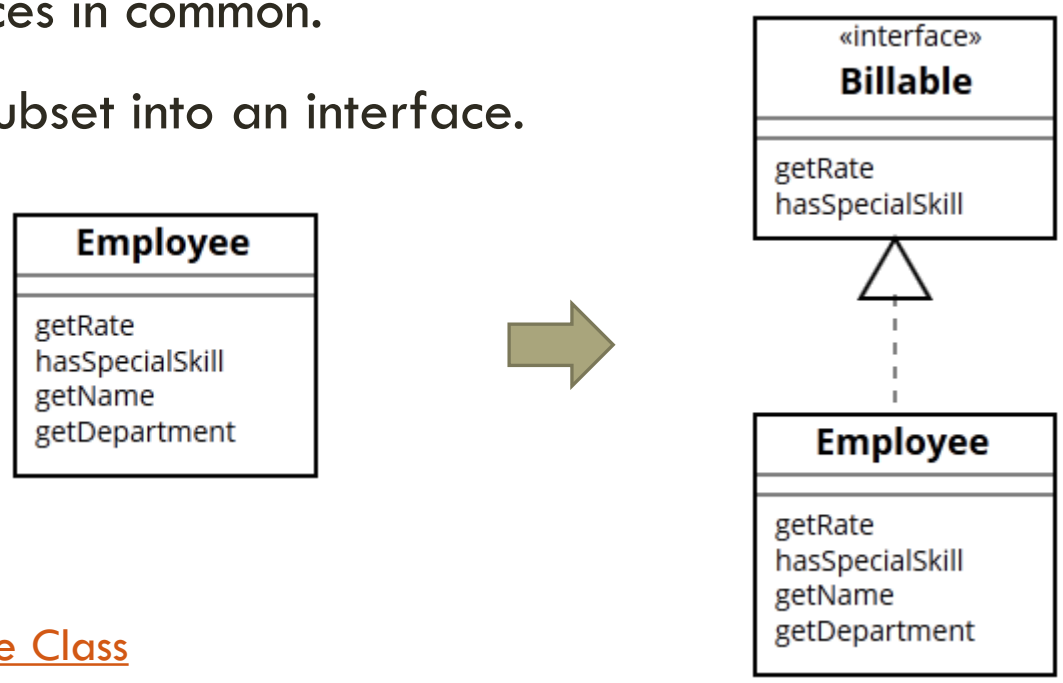


inverse of Extract Class, Extract Interface

EXTRACT INTERFACE

Several clients use the same subset of a class's interface, or two classes have part of their interfaces in common.

Extract the subset into an interface.



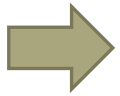
inverse of Inline Class

REPLACE ERROR CODE WITH AN EXCEPTION

A method returns a special code to indicate an error.

Throw an exception instead.

```
int withdraw(int amount) {
        if (amount > _balance)
            return -1;
        else {
             _balance -= amount; return 0;
        }
}
```



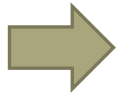
```
void withdraw(int amount) throws BalanceException {
        if (amount > _balance)
              throw new BalanceException();
         _balance -= amount;
}
```

REPLACE EXCEPTION WITH TEST

You are throwing an exception on a condition the caller could have checked first.

Change the caller to make the test first.

```
double getValueForPeriod (int periodNumber) {
    try {
        return _values[periodNumber];
    } catch (ArrayIndexOutOfBoundsException e) {
        return 0;
    }
}
```



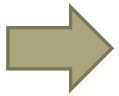
```
double getValueForPeriod (int periodNumber) {
    if (periodNumber >= _values.length)
        return 0;
    return _values[periodNumber];
}
```

CONSOLIDATE CONDITIONAL EXPRESSION

You have a sequence of conditional tests with the same result.

Combine them into a single conditional expression and extract it.

```
double disabilityAmount() {
        if (_seniority < 2) return 0;
        if (_monthsDisabled > 12) return 0;
        if (_isPartTime) return 0;
        // compute the disability amount
```



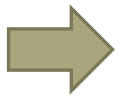
```
double disabilityAmount() {
        if (isNotEligableForDisability()) return 0;
         // compute the disability amount
```

CONSOLIDATE DUPLICATE CONDITIONAL FRAGMENTS

The same fragment of code is in all branches of a conditional expression.

Move it outside of the expression.

```
if (isSpecialDeal()) {
  total = price * 0.95;
  send();
}  else {
  total = price * 0.98;
  send();
}
```



```
if (isSpecialDeal())
  total = price * 0.95;
else
  total = price * 0.98;
send();
```

OBSTACLES TO REFACTORING

Complexity
- Changing design is hard
- Understanding code is hard

Possibility to introduce errors
- Run tests if possible
- Build tests

Cultural Issues
- *"We pay you to add new features, not to improve the code!"*

Performance issue
- Refactoring may slow down the execution

Normally only 10% of your system consumes 90% of the resources so just focus on 10 %.
- Refactorings help to localize the part that need change
- Refactorings help to concentrate the optimizations

Development is always under time pressure
- Refactoring takes time
- Refactoring better after delivery

SUMMARY

"The process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure" [Fowler]

Refactor to:

- Improve the software design
- Make the software easier to understand
- Help find bugs

A catalog of refactoring exists: Extract Method, Move Method, Replace Temp with Query, etc…

Refactoring has some obstacles