Laboratorio di Programmazione

Prof. Marco Bertini
marco.bertini@unifi.it
http://www.micc.unifi.it/bertini/

Refactoring

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand." - Martin Fowler

The problem

- The overall goal of software engineering is to create high quality software efficiently.
- But there are always many reasonable reasons that make this goal hard to reach…
  - Requirements change over time, making it hard to update your code (leading to less optimal designs)
  - Time and money cause you to take shortcuts
  - You learn a better way to do something

The solution

- "Refactoring is 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. It is a disciplined way to clean up code that minimizes the chances of introducing bugs." (Martin Fowler, 1999)
- Refactoring modifies software to improve its readability, maintainability, and extensibility without changing what it actually does:
  - External behavior does NOT change
  - Internal structure is improved

The solution

Refactoring is a technique which identifies bad code (code that "smells") and promotes the re-structuring of that bad code into classes and methods that are more readable, maintainable, and generally sparse in code. Refactoring yields a "better" design of both your classes and methods.

Typically this means restructuring (rearranging) code in a series of small, semantics-preserving transformations (i.e. the code keeps working)…

Refactoring: goals

- The goal of refactoring is NOT to add new functionality
- The goal is refactoring is to make code easier to maintain in the future
- During refactoring you need to keep the code working
    - You need to have unit tests to prove the code works

Refactoring: risks

- Refactoring CAN introduce problems, because anytime you modify software you may introduce bugs!
- Management thus says:
  - Refactoring adds risk!
  - It's expensive – we're spending time in development, but not "seeing" any external differences? And we still have to retest?
  - Why are we doing this?

Refactoring: benefits

- We refactor because we understand getting the design right the first time is hard and you get many benefits from refactoring:
  - Code size is often reduced
  - Confusing code is restructured into simpler code
  - Both of these greatly improve maintainability, which is required because requirements always change.

How to reduce risks

- ensure rollback (e.g. via versioning): be able to roll back to a previous working version
- small steps, one at a time: if you make a mistake, it is easy to find the bug.
- always test (unit tests, and more): before you start refactoring, check that you have a solid suite of tests.

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
  - If you can do so without breaking the code
    - Unit tests are essential for this
- You should not refactor:
  - Stable code (code that won't ever need to change)
  - Someone else's code
    - Unless you've inherited it (and now it's yours)

When to refactor

Refactor when you add functionalities.

Before you add new features, make sure your design and current code is “good”: this will help the new code be easier to write

Refactor as you do a code review.

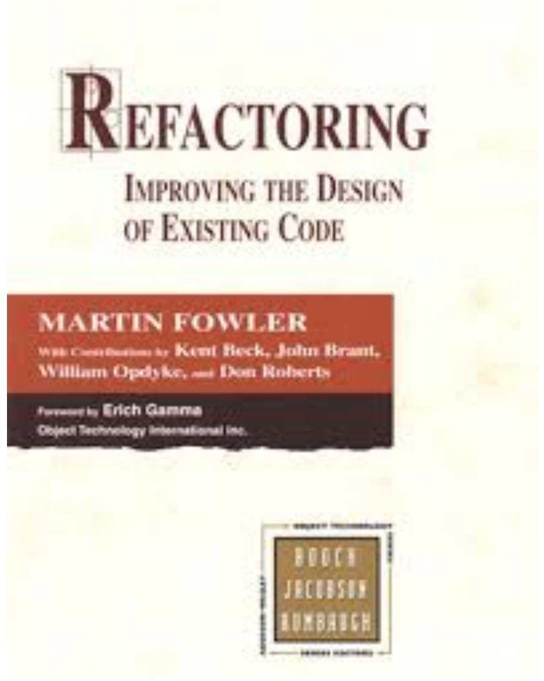
Refactor when you fix a bug.

- Stable code (code that won’t ever need to change)
- Someone else’s code
  - Unless you’ve inherited it (and now it’s yours)

What to refactor

- Martin Fowler uses "code smells" to identify when to refactor.
- Code smells are bad things done in code, somewhat like bad patterns in code
- Get a list here: http://wiki.c2.com/?CodeSmell

Refactoring catalog


REFACTORING
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER
With Contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International Inc.
BOOCH JACOBSON RUMBAUGH

- Martin Fowler has written a book reporting a catalog of refactoring operations
- Each refactoring entry is described with a name, summary of its operations, motivation, mechanics (how to do the refactoring), and examples (in Java)

Bad smells in code

- A “bad smell” in code is a structure of code that suggests the possibility of refactoring.
- There’s no formal description
- A list of bad smells with short informal descriptions and links to the refactoring that are more suitable is given on the book by Fowler mentioned in the previous slide.

Code smells

- Duplicated Code
  - bad because if you modify one instance of duplicated code but not the others, you (may) have introduced a bug!
- Long Method
  - long methods are more difficult to understand
  - rewrite a method as calls to many smaller well-named methods. They will become the documentation of the method itself

Code smells

- Large Class
    - classes try to do too much, which reduces cohesion. Does your class have too many instance variables or too much code ? Does it contains repetitions ?
- Long Parameter List
    - hard to understand, can become inconsistent.
    - Aren't you programming in procedural style ? Can't you pass an object or have an instance variable ?
- Divergent Change
    - Related to cohesion. Symptom: one type of change requires changing one subset of methods; another type of change requires changing another subset. Perhaps the class has too many responsibilities.

Code smells

- Lazy Class
  - A class that no longer "pays its way"
  - e.g. may be a class that was downsized by a previous refactoring, or represented planned functionality that did not pan out
- Speculative Generality
  - "Oh I think we need the ability to do this kind of thing someday"
- Temporary Field
  - An attribute of an object is only set in certain circumstances; but an object should need all of its attributes

Code smells

- Data Class
    - These are classes that have fields, getting and setting methods for the fields, and nothing else; they are data holders, but objects should be about data AND behavior
- Refused Bequest
    - A subclass ignores most of the functionality provided by its superclass
    - Subclass may not pass the "IS-A" test
- Comments (!)
    - Comments are sometimes used to hide bad code
        - "…comments often are used as a deodorant" (!)
        - a good comment should tell *why* code does something.

Code smells

- Inconsistent names
  - Pick a set of standard terminology and stick to it throughout your methods. For example, if you have Open(), you should probably have Close().
- Combinatorial Explosion
  - You have lots of code that does almost the same thing.. but with tiny variations in data or behavior. This can be difficult to refactor - perhaps using templates (generic programming)
- Dead code
  - just eliminate code that is not used. If you use version control there's no risk.

Code smells

- Shotgun surgery
  - Every time you make a kind of change in a class you have to make a lot of little changes to a lot of different classes
  - … you'll surely miss one of these changes sooner or later. Bring them together to avoid this risk.
- Parallel Inheritance hierarchies
  - Sort of a special case of the previous one: every time you make a subclass of one class, you also have to make a subclass of another one.

Code smells

- Inappropriate intimacy
    - Classes may make too much use of each other's fields and methods
    - Move methods and fields to the class or create a class that satisfies the needs of both classes
- Testing for null
- According to Fowler: "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."
    - Unfortunately, if the object might be null, you have to test it first
    - An occasional test for null isn't so bad, but a lot of them will clutter up the code. Solution: introduce a "null object"—a real object of the correct class (or a subclass of the correct class) that has the appropriate behavior

Code smells

… and many other “bad smells”…

The book by Fowler reports them.

Refactoring methods

Refactoring categories

- Refactoring methods are grouped in categories:
  - Composing Methods
    – Creating methods out of inlined code
  - Moving Features Between Objects
    – Changing of decisions regarding where to put responsibilities
  - Organizing Data
    – Make working with data easier

Refactoring categories

- Simplifying Conditional Expressions
- Making Method Calls Simpler
  – Creating more straightforward interfaces
- Dealing with Generalization
  – Moving methods around within hierarchies
- Big Refactorings
  – Refactoring for larger purposes

Extract method

- You have a code fragment that can be grouped together. Turn the fragment in to a method whose name explains the purpose of the method.
- Extract Method is one of the most common refactoring. Look at a method that is too long or look at code that needs a comment to understand its purpose. Then turn that fragment of code into its own method.

Extract method

Category: Composing Methods

- You have a code fragment that can be grouped together. Turn the fragment in to a method whose name explains the purpose of the method.
- Extract Method is one of the most common refactoring. Look at a method that is too long or look at code that needs a comment to understand its purpose. Then turn that fragment of code into its own method.

Extract method

- Create a new method, and name it after the intention of the method (name it by what it does, not by how it does it)
- Copy the extracted code from the source method into the new target method
- Scan the extracted code for references to any variables that are local in scope to the source method
- See whether any temporary variables are used only within this extracted code. If so, declare them in the target method as temporary variables Look to see whether any local-scope variable are modified by the existing code
- Pass into the target method as parameters local scope variables that are read from the extracted code
- Replace the extracted code in the source method with a call to the target method

```
void printForm() {
  printBanner();

  //print details
  cout << "name: " << name << endl;
  cout << "amount: " << getOutstanding()
       << endl;
}
```

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

void printDetails(double outstanding) {
  cout << "name: " << name << endl;
  cout << "amount: " << outstanding
       << endl;
}
```

Inline method

- It is the opposite of Extract Method: substitute a call to a method with its own code
- When a method body is more obvious than the method itself, replace calls to the method with the method's content and delete the method itself.
    - Make sure that the method is not redefined in subclasses. If the method is redefined, refrain from this technique.
    - Find all calls to the method. Replace these calls with the content of the method.
    - Delete the method.

Inline method

Category: Composing Methods

- It is the opposite of Extract Method: substitute a call to a method with its own code
- When a method body is more obvious than the method itself, replace calls to the method with the method's content and delete the method itself.
  - Make sure that the method is not redefined in subclasses. If the method is redefined, refrain from this technique.
  - Find all calls to the method. Replace these calls with the content of the method.
  - Delete the method.

Inline method

Category: Composing Methods

```
class PizzaDelivery {
  //...
  int getRating() {
    return moreThanFiveLateDeliveries() ? 2 : 1;
  }
  bool moreThanFiveLateDeliveries() {
    return numberOfLateDeliveries > 5;
  }
}
```

the content of the method.

- Delete the method.

Inline method

Category: Composing Methods

```
class PizzaDelivery {
  //...
  int getRating() {
    return numberOfLateDeliveries > 5 ? 2 : 1;
  }
}
```

- Delete the method.

Split temporary variable

- You have a local variable that is used to store various intermediate values inside a method (except for cycle variables).
- Use different variables for different values. Each variable should be responsible for only one particular thing.
  - If you are skimping on the number of variables inside a function and reusing them for various unrelated purposes, you are sure to encounter problems as soon as you need to make changes to the code containing the variables.
  - Find the first place in the code where the variable is given a value. Here you should rename the variable with a name that corresponds to the value being assigned.
  - Use the new name instead of the old one in places where this value of the variable is used.
  - Repeat as needed for places where the variable is assigned a different value.

Split temporary variable

```
double temp = 2 * (height + width);
cout << temp << endl;
temp = height * width;
cout << temp << endl;
```

problems as soon as you need to make changes to the code containing the variables.

- Find the first place in the code where the variable is given a value. Here you should rename the variable with a name that corresponds to the value being assigned.
- Use the new name instead of the old one in places where this value of the variable is used.
- Repeat as needed for places where the variable is assigned a different value.

Split temporary variable

```
double temp = 2 * (height + width);
cout << temp << endl;
temp = height * width;
cout << temp << endl;
```

problems as soon as you need to make changes to the code containing the variables.

```
double perimeter = 2 * (height + width);
cout << perimeter << endl;
double area = height * width;
cout << area << endl;
```

- Repeat as needed for places where the variable is assigned a different value.

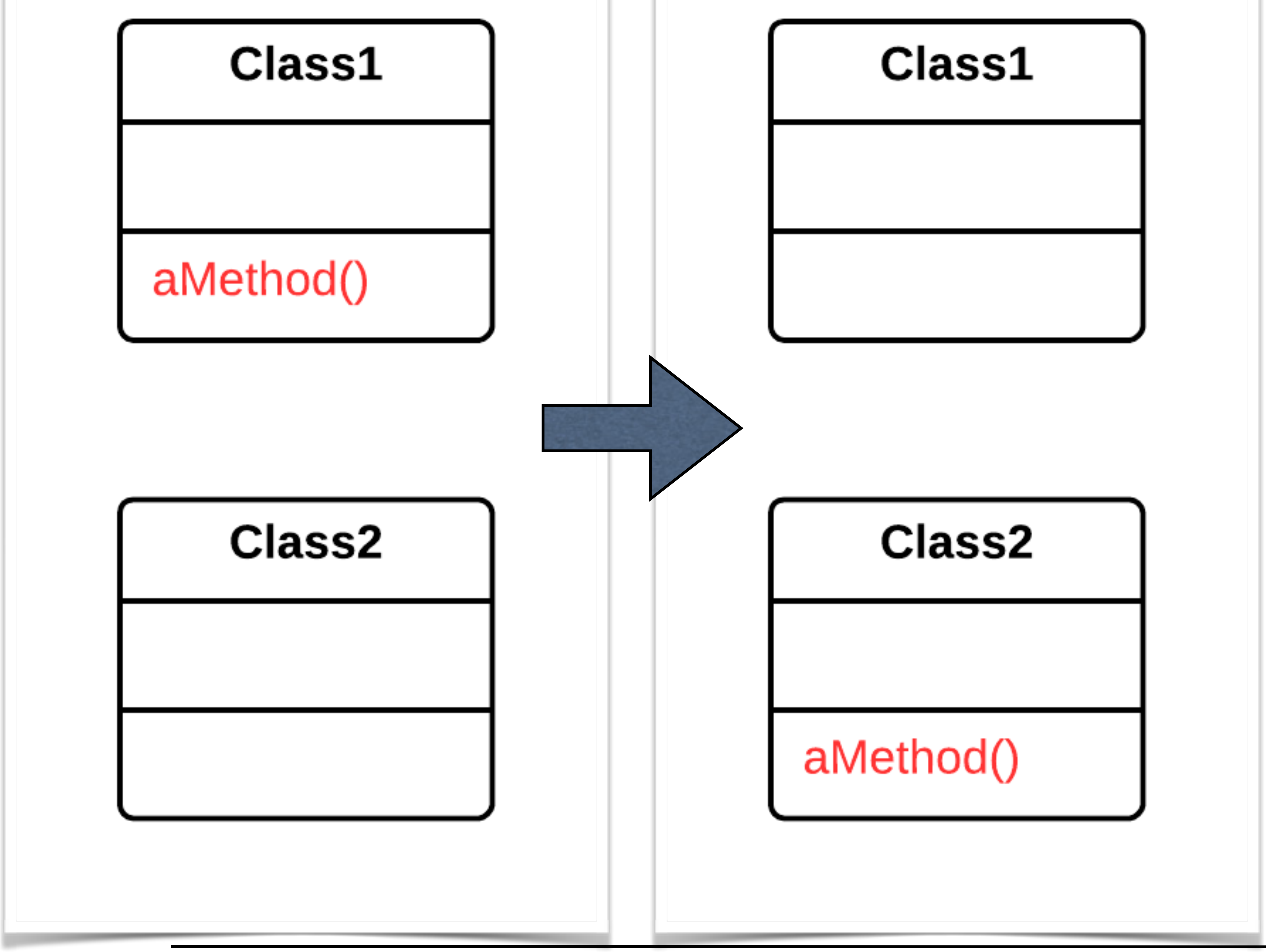
Move method

- A method is used more in another class than in its own class.
- Create a new method in the class that uses the method the most, then move code from the old method to there. Turn the code of the original method into a reference to the new method in the other class or else remove it entirely.

Move method

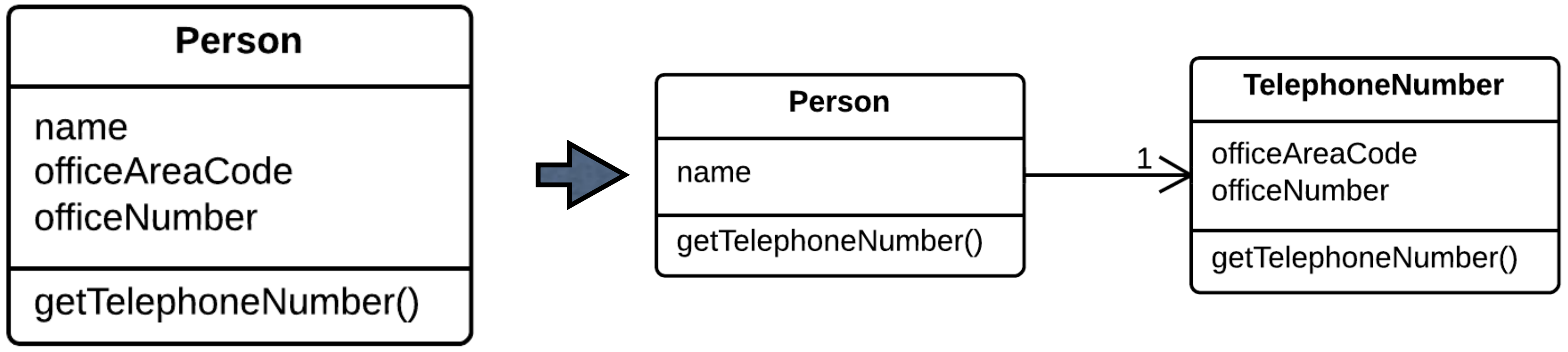
Category: Moving Features between Objects

- A method is used more in another class than in its own class.
- Create a new method in the class that uses the method the most, then move code from the old method to there. Turn the code of the original method into a reference to the new method in the other class or else remove it entirely.


Class1
aMethod()
Class2
Class1
Class2
aMethod()

Extract class

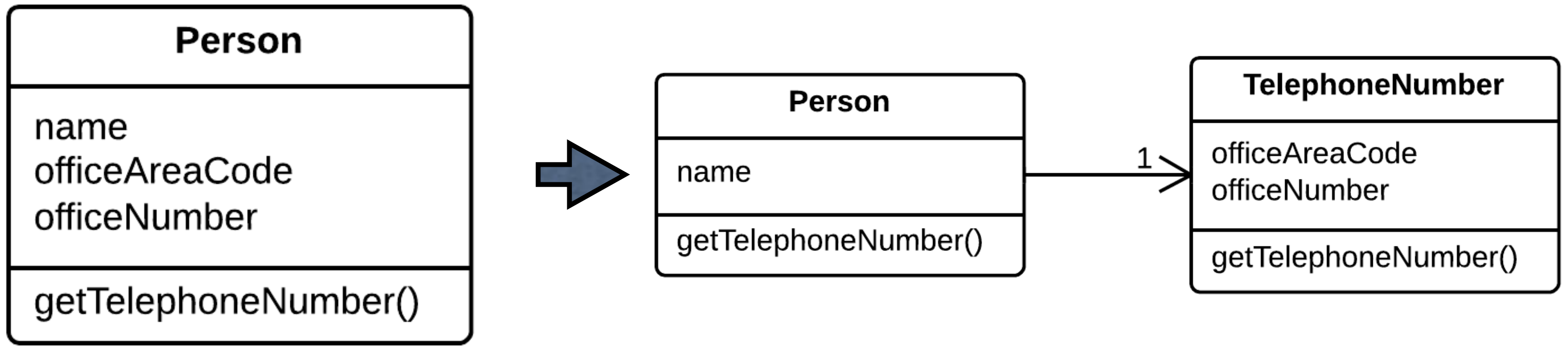
- When one class does the work of two, awkwardness results.
- Instead, create a new class and place the fields and methods responsible for the relevant functionality in it.
  - This refactoring method will help maintain adherence to the Single Responsibility Principle.



Extract class

Category: Moving Features between Objects

- When one class does the work of two, awkwardness results.
- Instead, create a new class and place the fields and methods responsible for the relevant functionality in it.
  - This refactoring method will help maintain adherence to the Single Responsibility Principle.



Move field

- Similar to move method… A field is used more in another class than in its own class.
- Create a field in a new class and redirect all users of the old field to it
- Often fields are moved as part of the Extract Class technique. Deciding which class to leave the field in can be tough. Rule of thumb: put a field in the same place as the methods that use it (or else where most of these methods are).

Move field

Category: Moving Features between Objects

- Similar to move method… A field is used more in another class than in its own class.
- Create a field in a new class and redirect all users of the old field to it
- Often fields are moved as part of the Extract Class technique. Deciding which class to leave the field in can be tough. Rule of thumb: put a field in the same place as the methods that use it (or else where most of these methods are).

Replace Magic Number with Symbolic Constant

- Your code uses a number that has a certain meaning to it.
- Replace this number with a constant that has a human-readable name explaining the meaning of the number.

Replace Magic Number with Symbolic Constant

Category: Organizing Data

- Your code uses a number that has a certain meaning to it.
- Replace this number with a constant that has a human-readable name explaining the meaning of the number.

Replace Magic Number with Symbolic Constant

Category: Organizing Data

```
double potentialEnergy(double mass, double height) {
  return mass * height * 9.81;
}
```

- Replace this number with a constant that has a human-readable name explaining the meaning of the number.

Replace Magic Number with Symbolic Constant

Category: Organizing Data

```
double potentialEnergy(double mass, double height) {
  return mass * height * 9.81;
}
```

```
const double GRAVITATIONAL_CONSTANT = 9.81;

double potentialEnergy(double mass, double height) {
  return mass * height * GRAVITATIONAL_CONSTANT;
}
```

Introduce Null Object

- Since some methods return "null" instead of real objects, you have many checks for "null" in your code.
- Instead of "null", return a null object that exhibits the default behavior.
  - The price of getting rid of conditionals is creating yet another new class.

Introduce Null Object

Category: Simplifying Conditional Expressions

- Since some methods return “null” instead of real objects, you have many checks for “null” in your code.
- Instead of “null”, return a null object that exhibits the default behavior.
  - The price of getting rid of conditionals is creating yet another new class.

```
class Animal {
public:
  virtual void make_sound() = 0;
};
class Dog : public Animal {
  void make_sound() { cout << "woof!" << endl; }
};
void make_default_sound() {
  cout << “...” << endl;
}

Animal* a = getAnimal(...);
if (a != nullptr)
  a->make_sound();
else
  make_default_sound();
```

```
class Animal {
public:
  virtual void make_sound() = 0;
};
class Dog : public Animal {
  void make_sound() { cout << "woof!" << endl; }
};
class NullAnimal : public Animal {
  void make_sound() { cout << “...” << endl;}
};

// may return null_animal
Animal* a = getAnimal(...);
a->make_sound();
```

Rename method

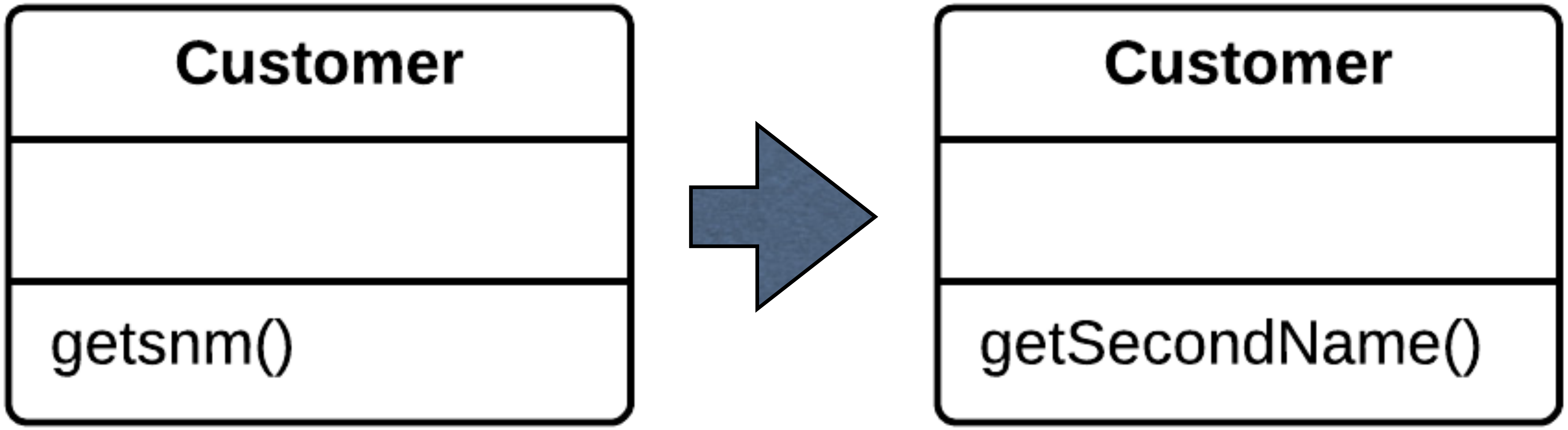
- The name of a method does not explain what the method does.
- Rename the method.
  - Perhaps a method was poorly named from the very beginning – for example, the method was created in a rush without giving proper care to naming it well.
  - Or perhaps the method was well named at first but as its functionality changed and its name stopped being a good descriptor.

Rename method

Category: Simplifying Method Calls

- The name of a method does not explain what the method does.
- Rename the method.
  - Perhaps a method was poorly named from the very beginning – for example, the method was created in a rush without giving proper care to naming it well.
  - Or perhaps the method was well named at first but as its functionality changed and its name stopped being a good descriptor.

Rename method



It is not a simple find/rename: we have to find all references to the old method and replace them with references to the new one.

Pull Up Field

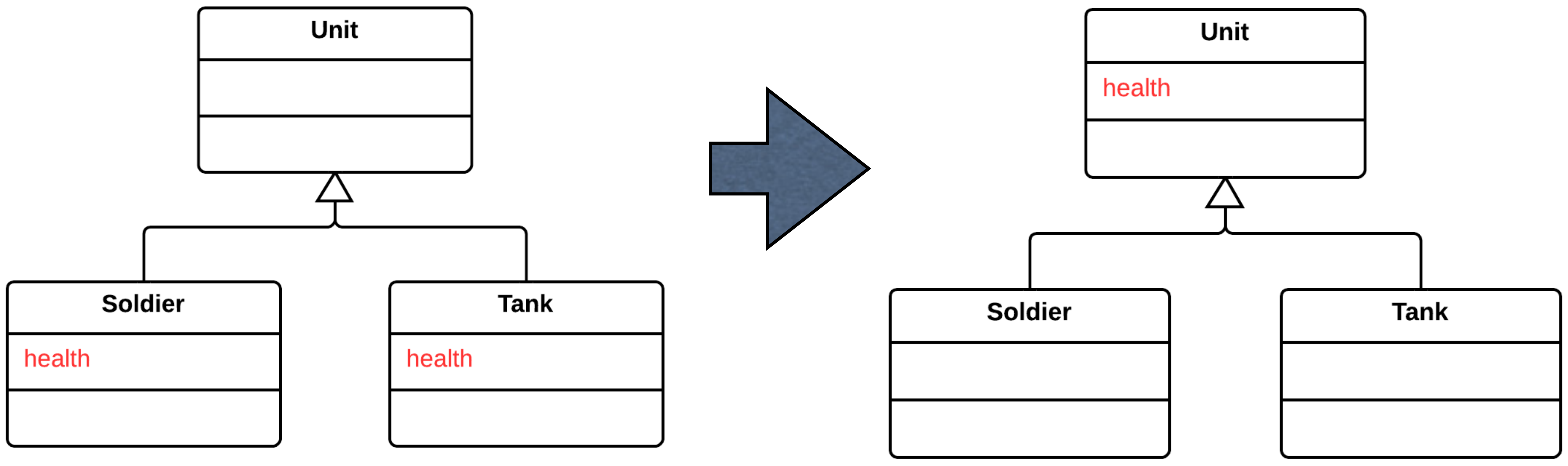
- Two classes have the same field.
- Remove the field from subclasses and move it to the superclass.
  - Subclasses grew and developed separately, causing identical (or nearly identical) fields and methods to appear.
  - Eliminates duplication of fields in subclasses.
  - Eases subsequent relocation of duplicate methods, if they exist, from subclasses to a superclass.

Pull Up Field

Category: Dealing with Generalisation

- Two classes have the same field.
- Remove the field from subclasses and move it to the superclass.
  - Subclasses grew and developed separately, causing identical (or nearly identical) fields and methods to appear.
  - Eliminates duplication of fields in subclasses.
  - Eases subsequent relocation of duplicate methods, if they exist, from subclasses to a superclass.

Pull Up Field



Pull Up method

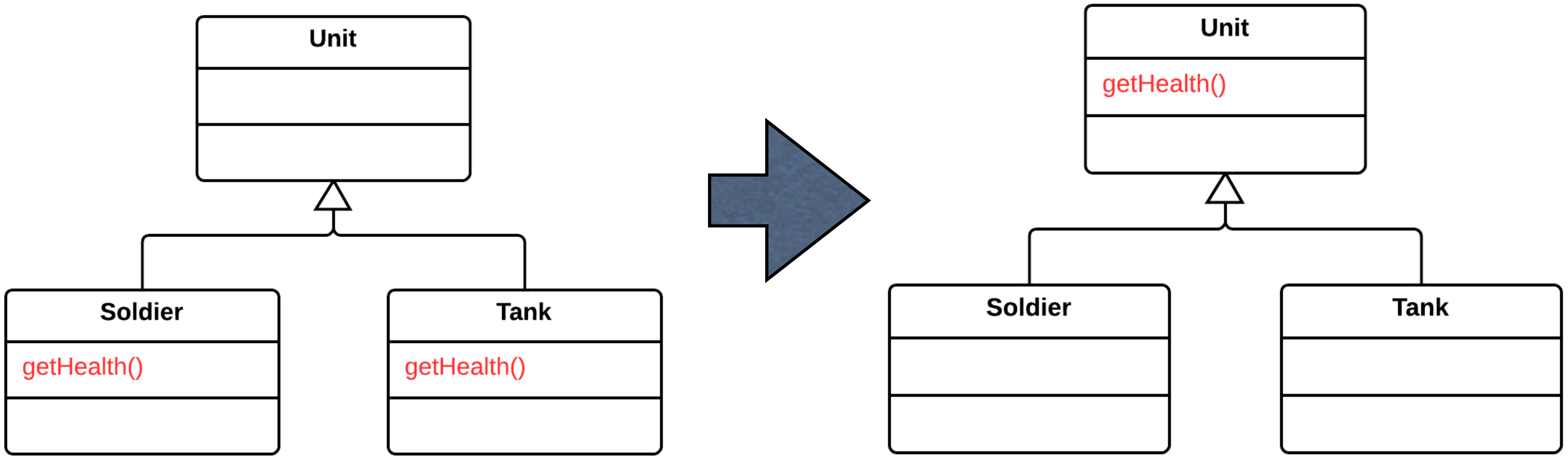
- Similar to pull up field… Your subclasses have methods that perform similar work.
- Make the methods identical and then move them to the relevant superclass.
    - Gets rid of duplicate code. If you need to make changes to a method, it's better to do so in a single place than have to search for all duplicates of the method in subclasses.

Pull Up method

Category: Dealing with Generalisation

- Similar to pull up field… Your subclasses have methods that perform similar work.
- Make the methods identical and then move them to the relevant superclass.
  - Gets rid of duplicate code. If you need to make changes to a method, it's better to do so in a single place than have to search for all duplicates of the method in subclasses.

Pull up method



Push down field/method

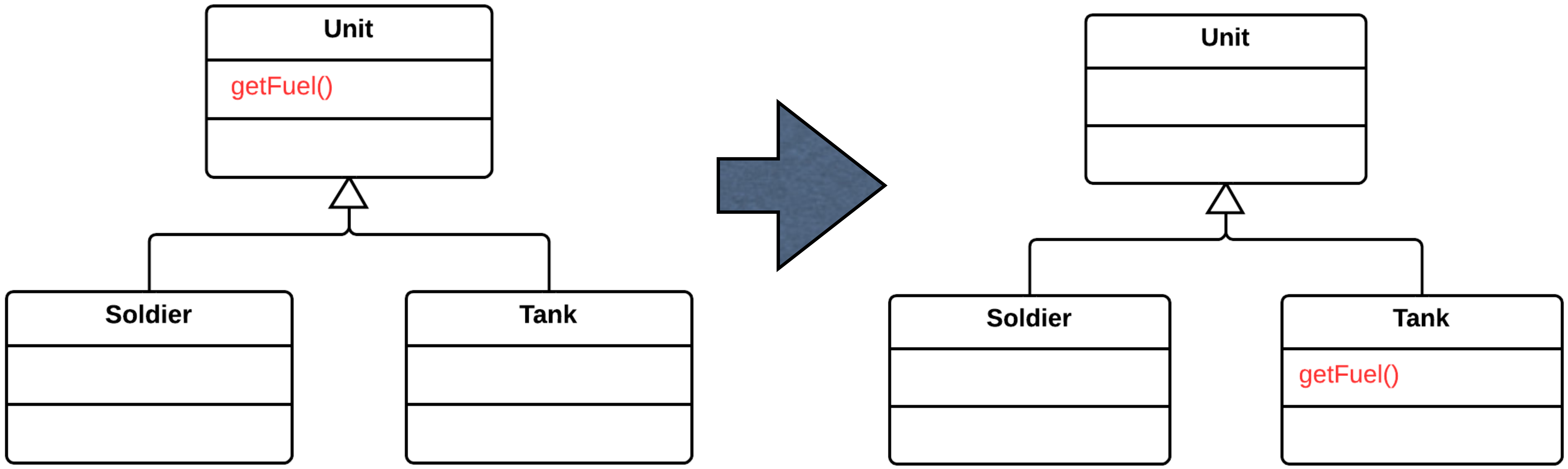
- It’s the opposite of the previous case: is a field used only in a few subclasses? Is behavior implemented in a superclass used by only one (or a few) subclasses?
- Then move the attribute or behavior to the subclass
  - Perhaps at first a certain method was meant to be universal for all classes but in reality is used in only one subclass. This situation can occur when planned features fail to materialize.
  - Such situations can also occur after partial extraction (or removal) of functionality from a class hierarchy, leaving a method that is used in only one subclass

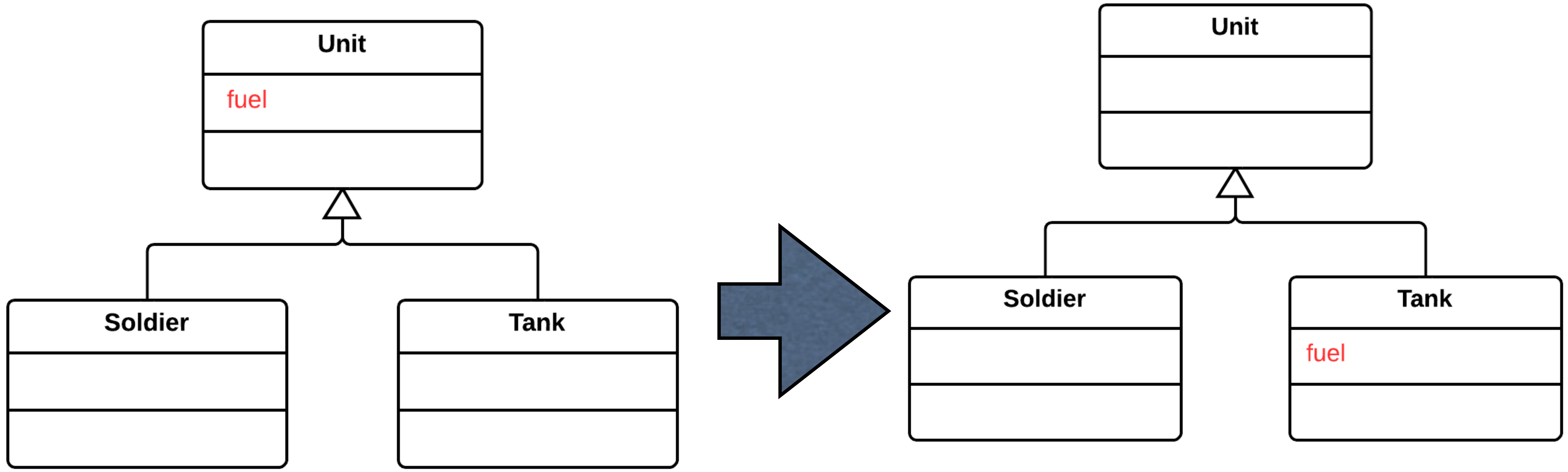
Push down field/method

Category: Dealing with Generalisation

- It's the opposite of the previous case: is a field used only in a few subclasses? Is behavior implemented in a superclass used by only one (or a few) subclasses?
- Then move the attribute or behavior to the subclass
  - Perhaps at first a certain method was meant to be universal for all classes but in reality is used in only one subclass. This situation can occur when planned features fail to materialize.
  - Such situations can also occur after partial extraction (or removal) of functionality from a class hierarchy, leaving a method that is used in only one subclass

Push down field/method





Extract Subclass

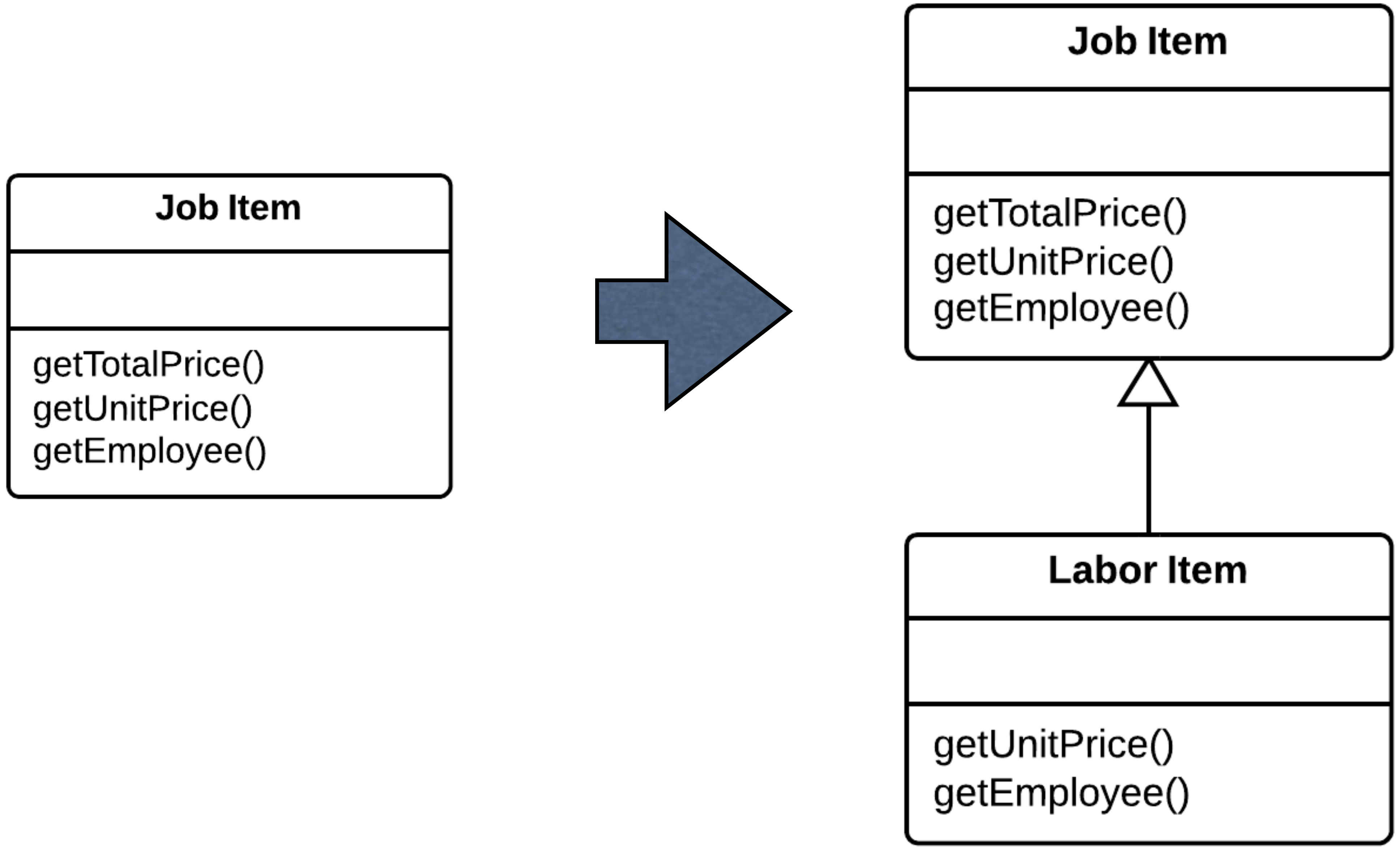
- A class has features that are used only in certain cases.
- Create a subclass and use it in these cases.
    - Your main class has methods and fields for implementing a certain rare use case for the class. While the case is rare, the class is responsible for it and it would be wrong to move all the associated fields and methods to an entirely separate class. But they could be moved to a subclass.

Extract Subclass

Category: Dealing with Generalisation

- A class has features that are used only in certain cases.
- Create a subclass and use it in these cases.
  - Your main class has methods and fields for implementing a certain rare use case for the class. While the case is rare, the class is responsible for it and it would be wrong to move all the associated fields and methods to an entirely separate class. But they could be moved to a subclass.

Extract subclass



Extract Superclass

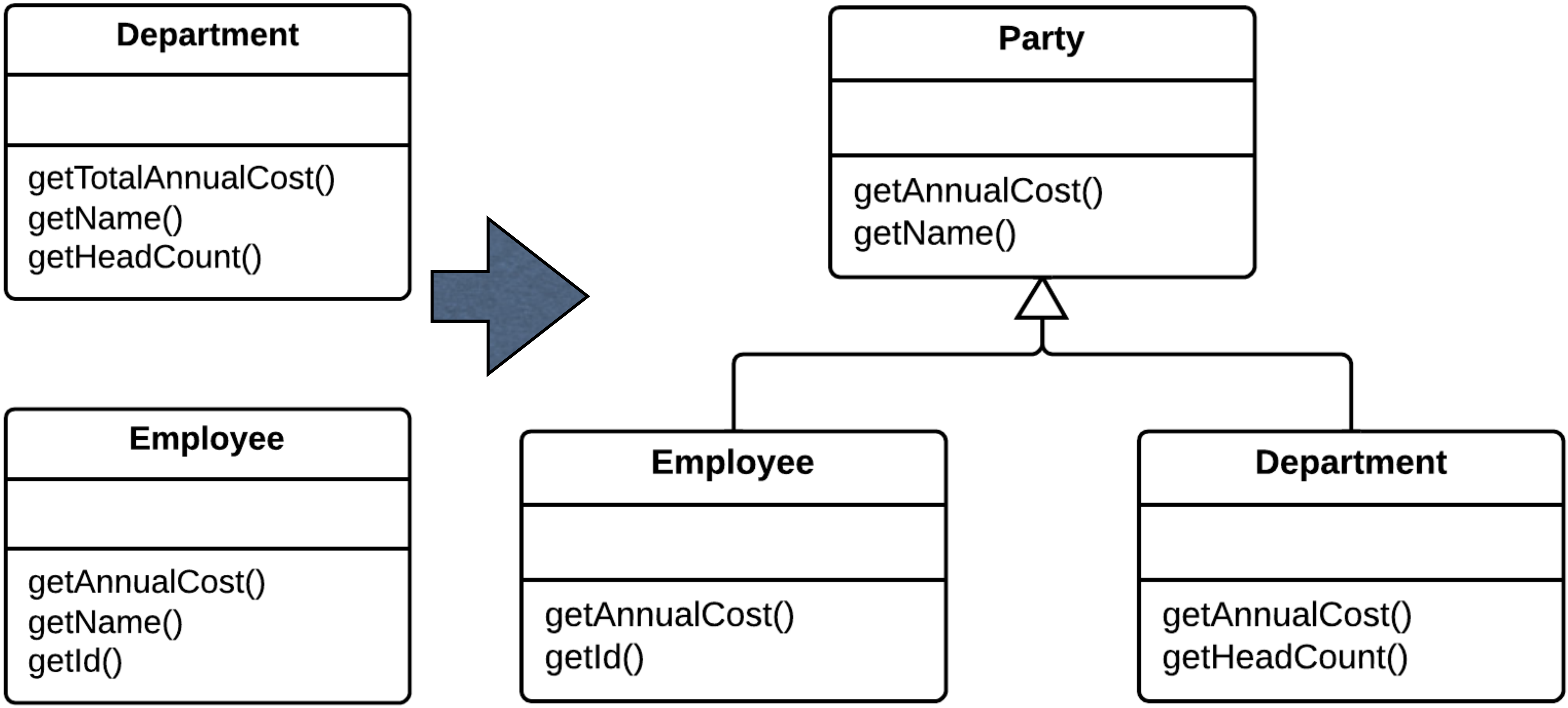
- You have two classes with common fields and methods.
- Create a shared superclass for them and move all the identical fields and methods to it.
  - One type of code duplication occurs when two classes perform similar tasks in the same way, or perform similar tasks in different ways. This should be handled via inheritance. But oftentimes this similarity remains unnoticed until classes are created, necessitating that an inheritance structure be created later.

Extract Superclass

Category: Dealing with Generalisation

- You have two classes with common fields and methods.
- Create a shared superclass for them and move all the identical fields and methods to it.
  - One type of code duplication occurs when two classes perform similar tasks in the same way, or perform similar tasks in different ways. This should be handled via inheritance. But oftentimes this similarity remains unnoticed until classes are created, necessitating that an inheritance structure be created later.

Extract Superclass



Convert Procedural Design

- Take each record type and turn it into a "dumb" data object with accessors
- Take all procedural code and put it into a single class
- Take each long method and apply Extract Method and the related factorings to break it down. As you break down the procedures use Move Method to move each one to the appropriate dumb data class
- Continue until all behavior is removed from the original class

Convert Procedural Design

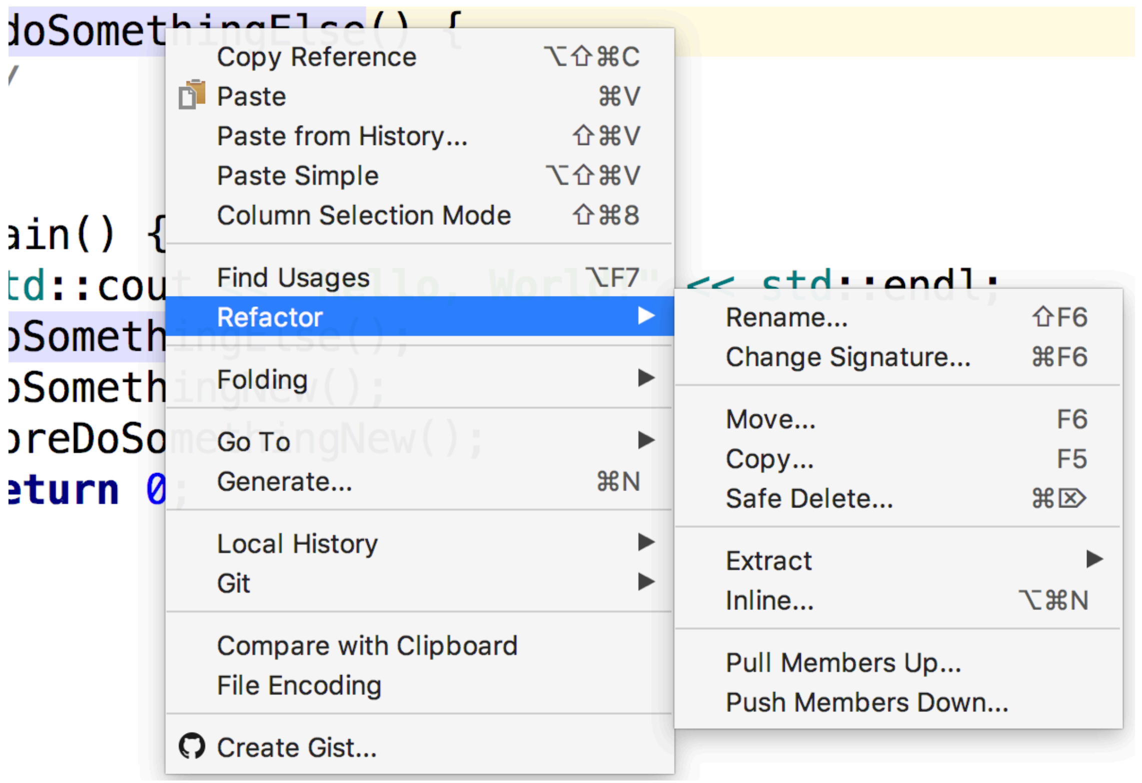
Category: Big refactoring

- Take each record type and turn it into a “dumb” data object with accessors
- Take all procedural code and put it into a single class
- Take each long method and apply Extract Method and the related factorings to break it down. As you break down the procedures use Move Method to move each one to the appropriate dumb data class
- Continue until all behavior is removed from the original class

CLion and refactoring

CLion and refactoring

- CLion offers you a set of code refactorings, which track down and correct the affected code references automatically.



  - other modern IDEs provide similar services, e.g. Eclipse.

  - some of these refactoring apply also to files and project structure, e.g. renaming/deleting files

CLion and refactoring

- Rename: renames symbols, automatically correcting all references in the code for you.
- Change Signature: helps you add/remove/reorder function parameters, change the result type or update the name of the function, all usages will be fixed as well.
- Move: moves files or directories, as well as methods, variables or constants.
- Copy: creates a copy of file or directory.

CLion and refactoring

- Safe Delete: safely removes files and symbols from your code.
- Inline: replaces redundant variable usage/ method calls with its initializer/declaration.
- Extract refactoring – CLion analyses the block of code where the refactoring was invoked, detects input and output variables, together with the usages of the selected expression to replace them with the newly created

CLion and refactoring

Extract can be applied to:

- Variable
- Constant
- Parameter
- Typedef
- Define
- Method
- Superclass
- Subclass

CLion and refactoring

- Pull Members Up safely moves class members to a superclass.
- Push Members Down safely moves class members to a subclass.

How to refactor

- Select (or hover caret on) a symbol or code fragment to refactor. The set of available refactorings depends on your selection.
- On the main Refactor menu or on the context menu of the selection, choose the desired refactoring
  - For certain refactorings, there is an option of previewing the changes prior to actually performing the refactoring.
  - If conflicts are expected after the refactoring, CLion displays a dialog with a brief description of the encountered problems.

Change signature

- The Change Signature refactoring combines several different modifications that can be applied to a function signature. It allows:
  - To change the function name.
  - To change the function return type.
  - To add new parameters and remove the existing ones.
  - To reorder parameters.

Extract constant/define

- Implements the “Replace Magic Number with Symbolic Constant” refactoring substituting a number with a constant or define.
  - Select the expression to be changed, if more than one expression is detected for the current cursor position, the Expressions list appears. If this is the case, select the required expression.
  - If more than one occurrence of the expression is found within the class, specify whether you wish to replace only the selected occurrence, or all the found occurrences with the new constant.

Extract function

- Creates a function from a fragment of code, detecting variables that are the input for the selected code fragment and the variables that are output for it.

```
int main() {
    int x = 10;
    int y = 15;

    //before Extract Method refactoring
    int z = y - x;

    return 0;
}
```

```
int subtract(int x, int y);

int main() {
    int x = 10;
    int y = 15;

    //after Extract Method refactoring
    int z = subtract(x, y);

    return 0;
}

int subtract(int x, int y) { return y - x; }
```

Extract Superclass/Subclass

- Perform the corresponding refactorings
  - Select the desired class in one of the views, or just open it in the editor, then select the refactoring
  - Specify names of new classes and which methods are to be moved

Extract Superclass: example

```
//Initial class InitialClass before Extract
// Superclass refactoring.
class InitialClass {
private:
    int field;

    void foo();
    int bar();
};

void InitialClass::foo() {
}

int InitialClass::bar() {
    return field;
}
```

```
// A new base class had been created
// including the selected members of the
// initial class
class BaseClass : public InitialClass {
private:
    int field;

    int bar();
};

//The function declaration had changed
int BaseClass::bar() {
    return field;
}

// Initial class after refactoring.
// The selected members had been moved to the
// created super class
class InitialClass {
private:
    void foo();
};

void InitialClass::foo() {
}
```

Extract Subclass: example

```
//Initial class InitialClass before
// Extract Subclass refactoring.
class InitialClass {
private:
    int field;

    void foo();
    int bar();
};

void InitialClass::foo() {
}

int InitialClass::bar() {
    return field;
}
```

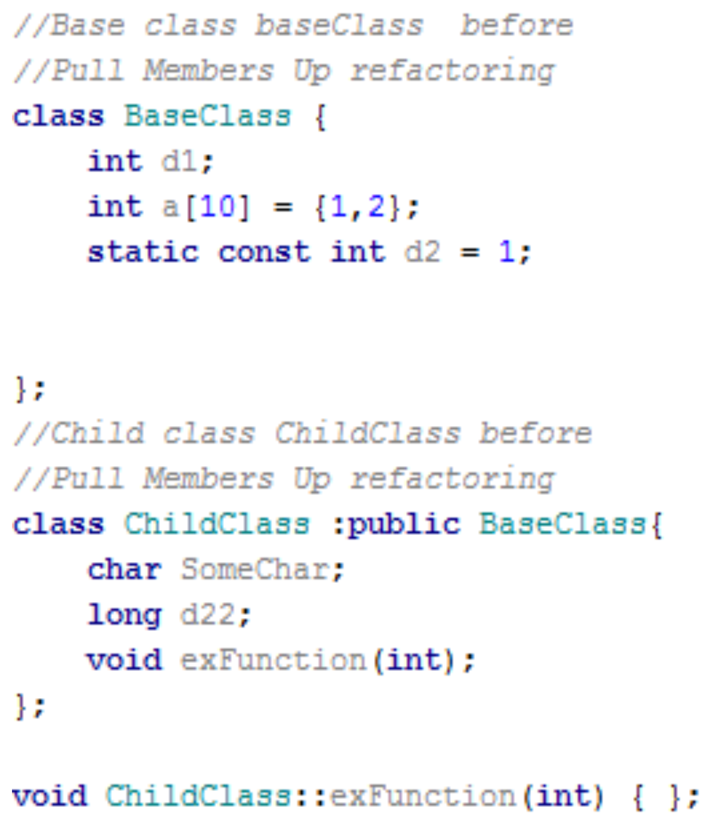
```
// Initial class after refactoring.
// The selected members had been moved
// to the created sub class
class InitialClass {
private:
    void foo();
};

void InitialClass::foo() {
}

// A new sub class had been created including
// the selected members of the initial class
class SubClass : public InitialClass {
private:
    int field;

    int bar();
};

//The function declaration had changed
int SubClass::bar() {
    return field;
}
```

Move

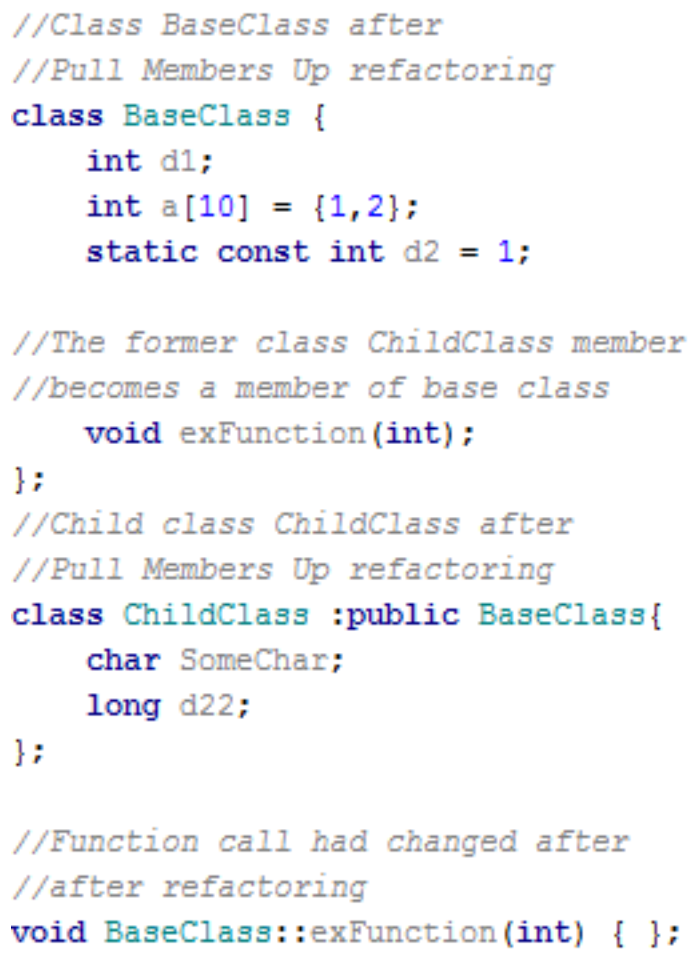
- Move methods and declarations as the corresponding refactoring
- Moves also files, to reorganize a project

Pull / push members

- Perform the corresponding refactorings



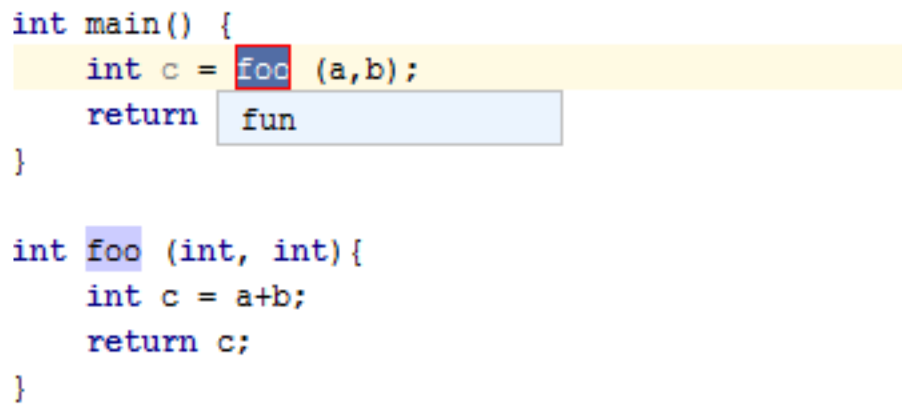
```
//Base class baseClass  before
//Pull Members Up refactoring
class BaseClass {
    int d1;
    int a[10] = {1,2};
    static const int d2 = 1;


};
//Child class ChildClass before
//Pull Members Up refactoring
class ChildClass :public BaseClass{
    char SomeChar;
    long d22;
    void exFunction(int);
};

void ChildClass::exFunction(int) { };
```



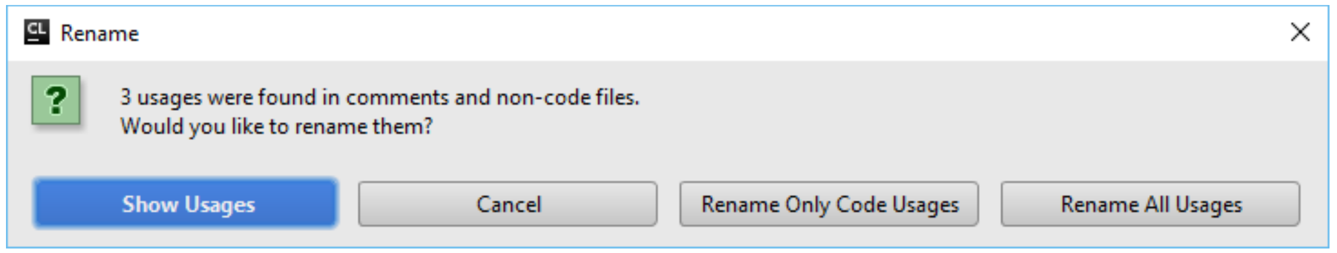
```
//Class BaseClass after
//Pull Members Up refactoring
class BaseClass {
    int d1;
    int a[10] = {1,2};
    static const int d2 = 1;

//The former class ChildClass member
//becomes a member of base class
    void exFunction(int);
};
//Child class ChildClass after
//Pull Members Up refactoring
class ChildClass :public BaseClass{
    char SomeChar;
    long d22;
};

//Function call had changed after
//after refactoring
void BaseClass::exFunction(int) { };
```

Rename

- Perform the corresponding refactoring if applied to a method, allows to rename everything and to re-organize the project applying it to files or directories.



```
int main() {
    int c = foo (a,b);
    return  fun
}

int foo (int, int){
    int c = a+b;
    return c;
}
```

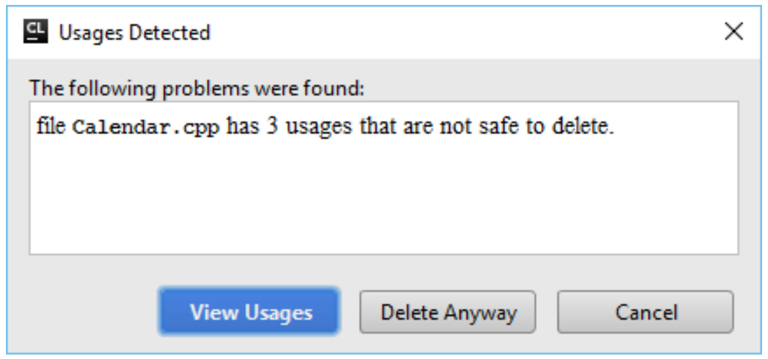


Rename

3 usages were found in comments and non-code files.
Would you like to rename them?

Safe delete

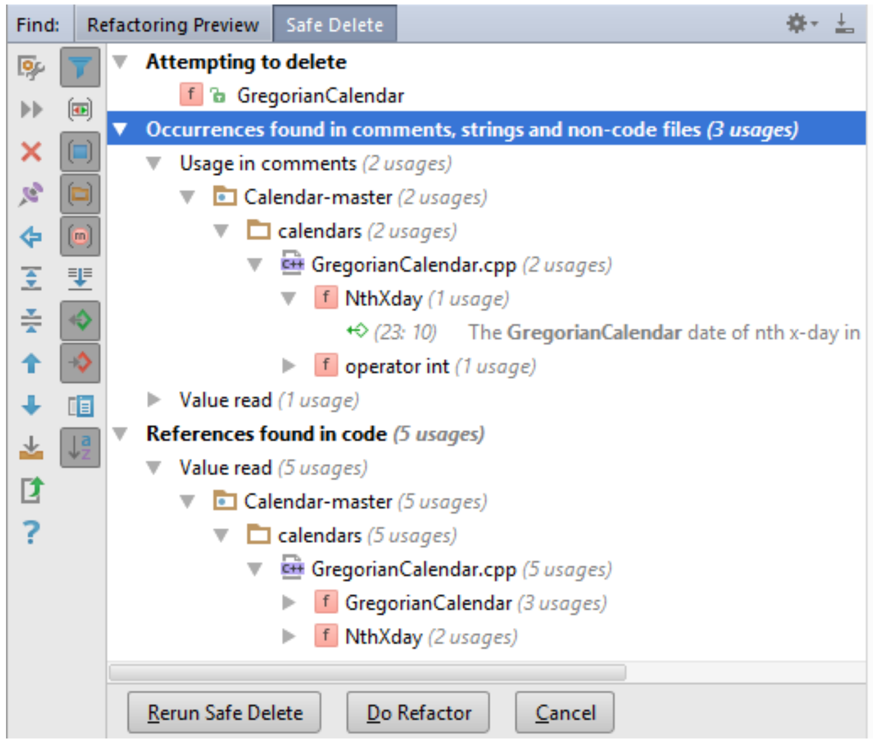
- Allows to remove files, checking if there some use for the code they contain.
  - Partially updates the CMake instructions. In some cases the resulting CMake may not be correct and has to be adjusted manually.



Usages Detected

The following problems were found:

file `Calendar.cpp` has 3 usages that are not safe to delete.



```
Attempting to delete
    GregorianCalendar
Occurrences found in comments, strings and non-code files (3 usages)
  Usage in comments (2 usages)
    Calendar-master (2 usages)
      calendars (2 usages)
        GregorianCalendar.cpp (2 usages)
          NthXday (1 usage)
            (23: 10)   The GregorianCalendar date of nth x-day in
          operator int (1 usage)
  Value read (1 usage)
References found in code (5 usages)
  Value read (5 usages)
    Calendar-master (5 usages)
      calendars (5 usages)
        GregorianCalendar.cpp (5 usages)
          GregorianCalendar (3 usages)
          NthXday (2 usages)
```

Reading material

- M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, "Refactoring: Improving the Design of Existing Code", Addison-Wesley.
- Tutorial: https://sourcemaking.com/refactoring

Credits

- These slides are based on the material of:
  - David Matuszek, Univ. of Pennsylvania
  - Dan Fleck, George Mason University
  - Jonathan I. Maletic, Kent State University