Lecture # 1

Refactoring

اَللّٰھُمَّ فَقِّھْنِا فِی الدِّ یْنِ

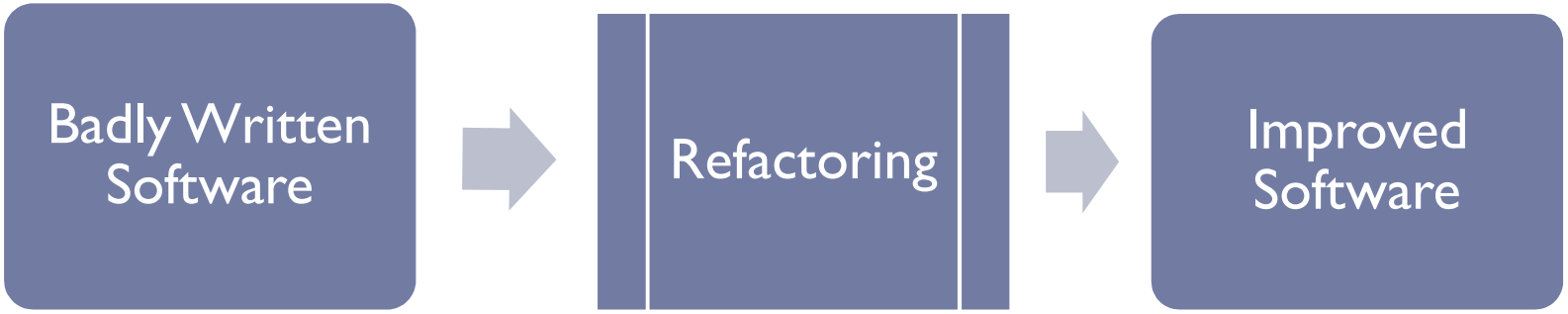
اے اللہ! ہمیں دین کی سمجھ عطا فرما۔

O Allah! Grant us understanding of Deen.

(Sahih Al-Bukhari)

What is Refactoring?

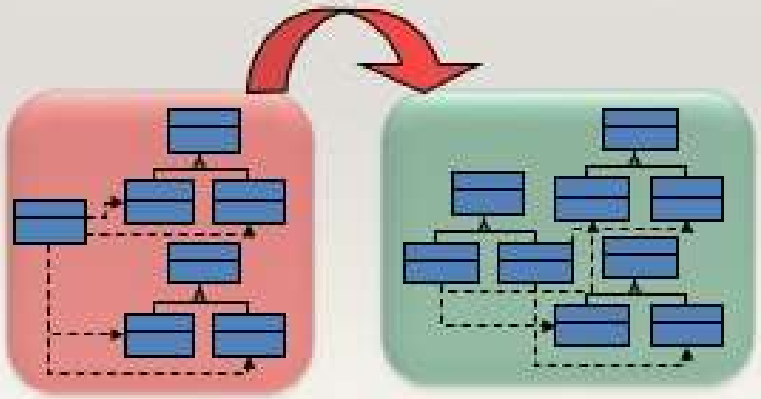
Refactoring is a disciplined process of applying structural transformations in the code such that the program is improved in terms of quality and its external behavior is preserved.



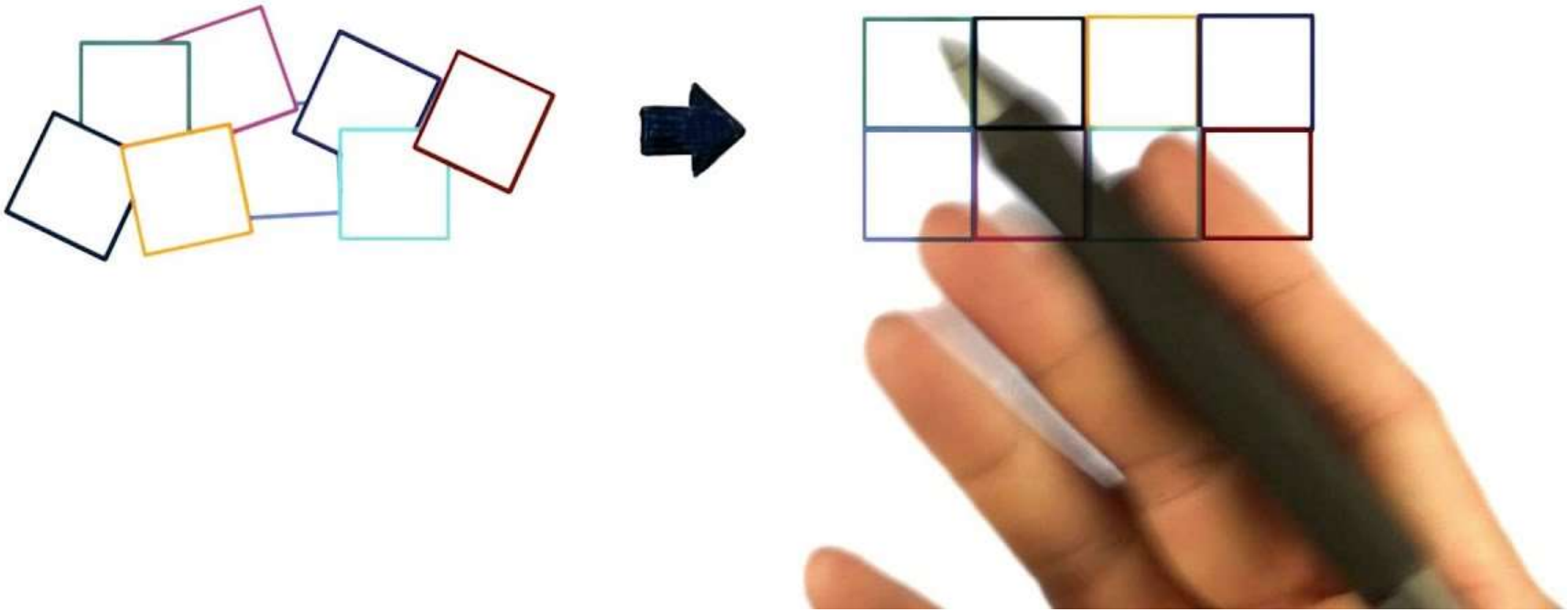
What is refactoring?

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*

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



REFACTORING



Refactoring in General

- Opdyke (1992)
  - Presented preconditions for twenty-three primitive refactorings.
  - Definition of scope of refactoring involving a public variable is limited to the class which contains it and its subclasses.
- Roberts (1999)
  - extended the definition of refactoring by adding postcondition assertions.
- Fowler's refactoring catalog (1999)
  - Consists of 72 refactoring guidelines in natural language.
  - Consists of both low and high level refactorings, where bigger refactorings use smaller refactorings to complete the process.

Fowler's Catalogue

- Fowler *et al.*'s catalog on Refactoring
  - 22 Bad smells
  - 72 refactorings
    - Overview
    - Mechanics
    - Examples

- Non-formal presentation –widely adopted and used
- Most comprehensive text on refactoring available

Refactoring should include...

1. Evaluation of its preconditions
2. Execution of its mechanics
3. Execution of corrective actions
4. Verification

change_member_function_name

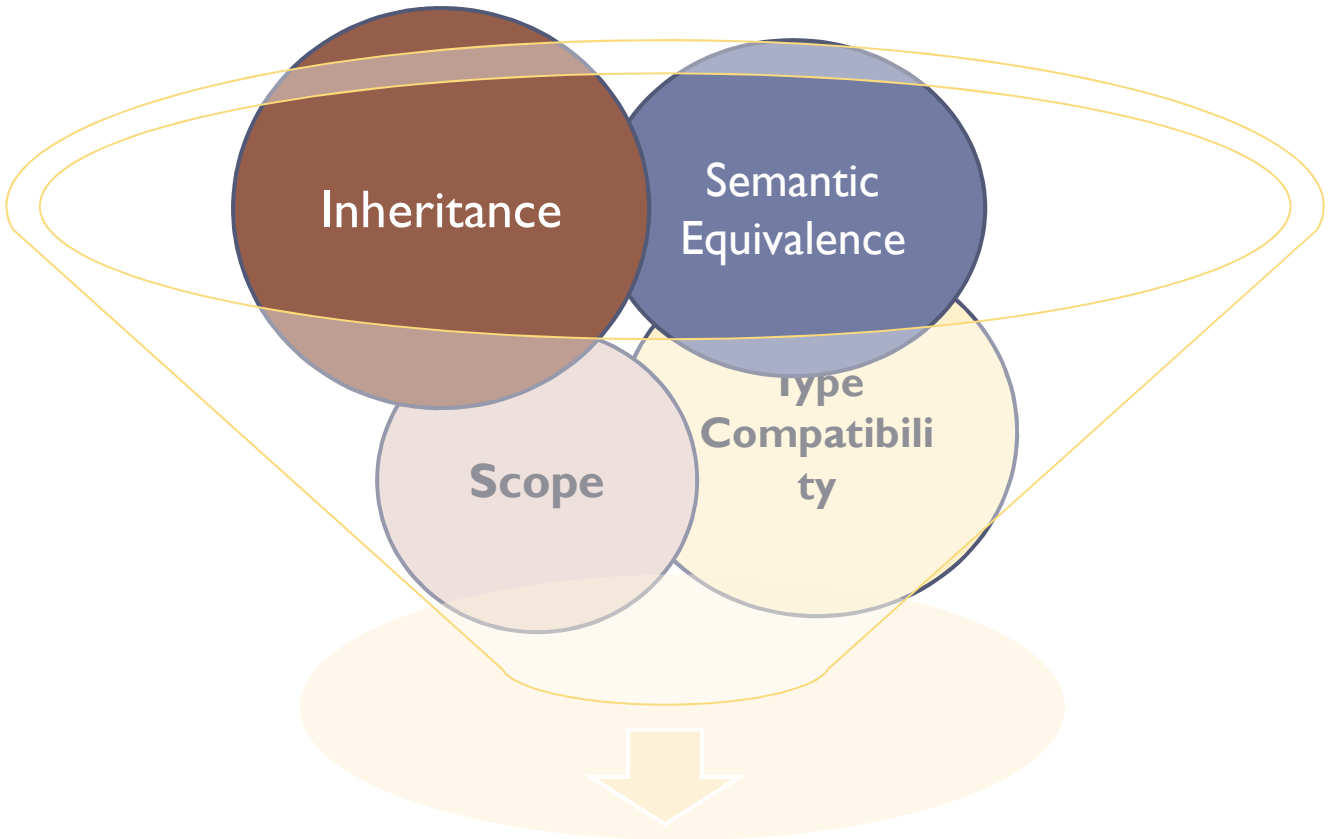
Change the name of a member function and any corresponding member functions defined in subclasses (and callers).

Arguments: function F, string newName.

Preconditions:

1. $\forall$ member $\in$ F.owner.locallyDefinedMemberFunctions,
   member.name $\neq$ newName.
   (a function with the same name is not already defined locally)
2. if F.accessControlMode $\neq$ private,
   $\forall$ subClass $\in$ subclassesOf(F.owner),
   $\forall$ member $\in$ subClass.locallyDefinedMemberFunctions,
   member.name $\neq$ newName.
   (if F is not private, a function with same name is not already defined locally in a subclass)
3. $\forall$ F2 $\in$ inheritedMemberFunctions(containingClass(F)),
   (F2.name = F.name) $\Rightarrow$
   matchingSignatureP (F, F2).
   (the signature matches that of any inherited function with the same name)
4. $\forall$ F2 $\in$ inheritedMemberFunctions(containingClass(F)),
   (F2.name = F.name) $\Rightarrow$
   ($\forall$ class $\in$ containingClass(F) $\cup$ subclassesOf(containingClass(F)),
   unrefdOnInstancesP(F2, class)) $\vee$
   (semantically EquivalantP F, F2).

Program Properties and Behavior Preservation



Rules for behavior Preservation

Why is Refactoring Useful?

- **The idea behind refactoring is to acknowledge that it will be difficult to get a design right the first time**
  - **and as a program's requirements change, the design may need to change**
  - **refactoring provides techniques for evolving the design in small incremental steps**
- **Benefits**
  - **Often code size is reduced after a refactoring**
  - **Confusing structures are transformed into simpler structures which are easier to maintain and understand**

Types of Refactoring (based on process)

- Floss refactoring:
  - Small.
  - Frequently performed
  - Mixed with some other tasks(it's not an exclusive task to refactor).
  - Keeps code healthy.
  - Perceived as the best practice.
  - Difficult to separate the impact of refactoring process from the other task
- Root canal refactoring:
  - Big.
  - Not frequent.
  - It is performed as just refactoring.
  - Corrective process.
  - Perceived as an emergency procedure.
  - Easy to verify the correctness of refactoring

Types of Refactoring (based on size)

- Primitive Refactoring- Small, atomic refactorings
  - Change method name
  - Create method etc.
- Composite Refactoring –These **composite refactorings** are usually defined as a sequence of **primitive refactorings as well as composite refactorings**, and reflect more complex behaviour-preserving transformations
  - Extract method
  - Move method
  - Pull up method

Types of Refactoring (based on impact)

- Refactoring inside method body
  - Limited scope
  - Clients outside method body not impacted
    - Replace temp with query
    - Substitute Algorithm
    - Extract method
- Refactoring impact the public interface for the clients
- Wide scope
  - Any change in the method signature
  - Moving a method/ field to another class
  - Extracting class

A (Very) Simple Example

Consolidate Duplicate Conditional Fragments

```
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 ();
```

A (Very) Simple Example

Better Still

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

send ();
```

```
if (isSpecialDeal())
    factor = 0.95;
else
    factor = 0.98;

total = price * factor;
send ();
```

Why should you refactor?

- **Refactoring improves the design of software without refactoring, a design will "decay" as people make changes to a software system**
- **Refactoring makes software easier to understand because structure is improved, duplicated code is eliminated, etc.**
- **Refactoring helps you find bugs**
- **Refactoring promotes a deep understanding of the code at hand, and this understanding aids the programmer in finding bugs and anticipating potential bugs**
- **Refactoring helps you program faster because a good design enables progress**

Refactoring versus Rewriting

- Code has to work mostly correctly before you refactor.
- If the current code just does not work, do not refactor. Rewrite instead!

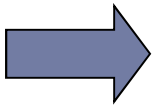
Optimization versus Refactoring

- The purpose of refactoring is to make software easier to understand and modify
- Performance optimizations often involve making code harder to understand (but faster!)

Optimization versus Refactoring

```
for (i=0; i < N; i++)
{
      // condition does not
      // change inside the
      // loop

      if (cond1)          {//1…}
      else if (cond2)     {//2…}
      else                {//3…}
      // xyz
}
```



```
if (cond1) {
      for (i=0; i < N; i++) {
            //1…
            // xyz
      }
else if (cond2) {
      for (i=0; i < N; i++) {
            //2…
            // xyz
      }
else  {
      for (i=0; i < N; i++) {
            //3…
            // xyz
      }
}
```

Optimization versus Refactoring

There is no apriori reason to believe that refactored code is "more optimized" in terms of resources consumed during execution;

Refactoring is really about improving code maintainability.

Often more maintainable code is less resource efficient than highly-tuned code.

Anyone can write code that computers can understand, good programmers write code that humans can understand!

Refactoring: Where to Start?

- How do you identify code that needs to be refactored?

"Bad Smells" in Code

Refactoring

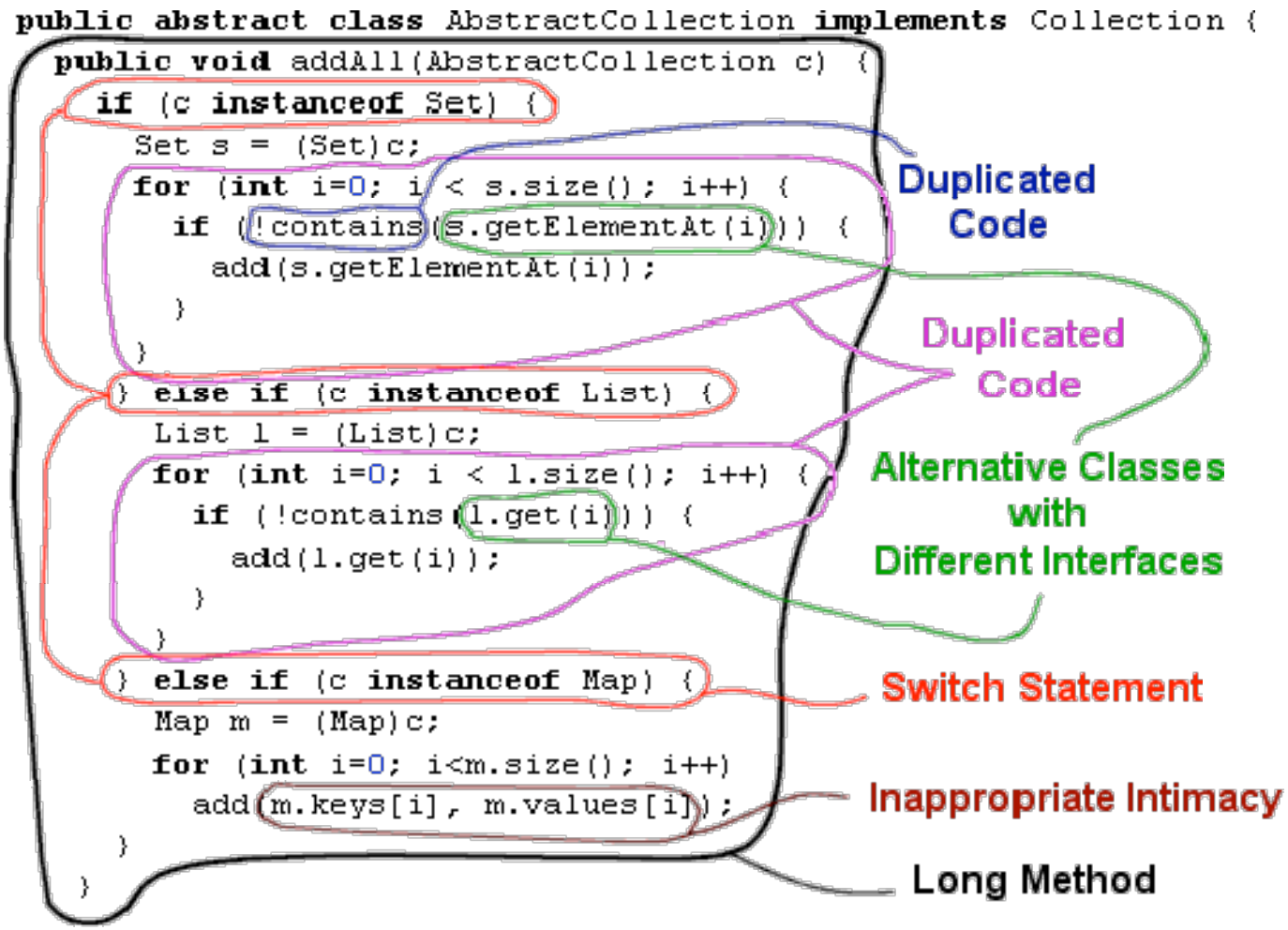
- Unreadable Code …
- Duplicated Code …
- Complex Code …

…

Bad Smell Code is

HARD to MODIFY

Identifying bad smells



Duplicated Code

If the same code structure is repeated. It is bad because if you modify one instance of duplicated code but not the others, you (may) have introduced a bug!

- Extract Method - gather duplicated code
  Simplest – duplication in the same class.
- Pull Up Method - move to a common parent
  In sibling classes. Extract method + Pull Up Method.
- Form Template Method - gather similar parts, leaving holes.
  Similar but not equal code in sibling classes.
- Substitute Algorithm - choose the clearer algorithm
- Extract class - create a new class with the duplicated code.
  For duplication in unrelated classes.

Duplicated Code

- DRY: Don't Repeat Yourself

Long Method

If the body of a method is over a page (choose your page size) .

Long methods are more difficult to understand; performance concerns with respect to lots of short methods are largely obsolete

- Extract Method - extract related behavior. The need for comments is a good heuristic.
- Replace Temp with Query - remove temporaries when they obscure meaning.
  - Might enable extract method.
- Introduce Parameter Object / Preserve Whole Object - slim down parameter lists by making them into objects.
  - Extract Method might lead to long parameter lists.
- Replace Method with Method Object – If still too many parameters. Heavy machinery.
- Decompose Conditionals - conditional and loops can be moved to their own methods

Extract Method

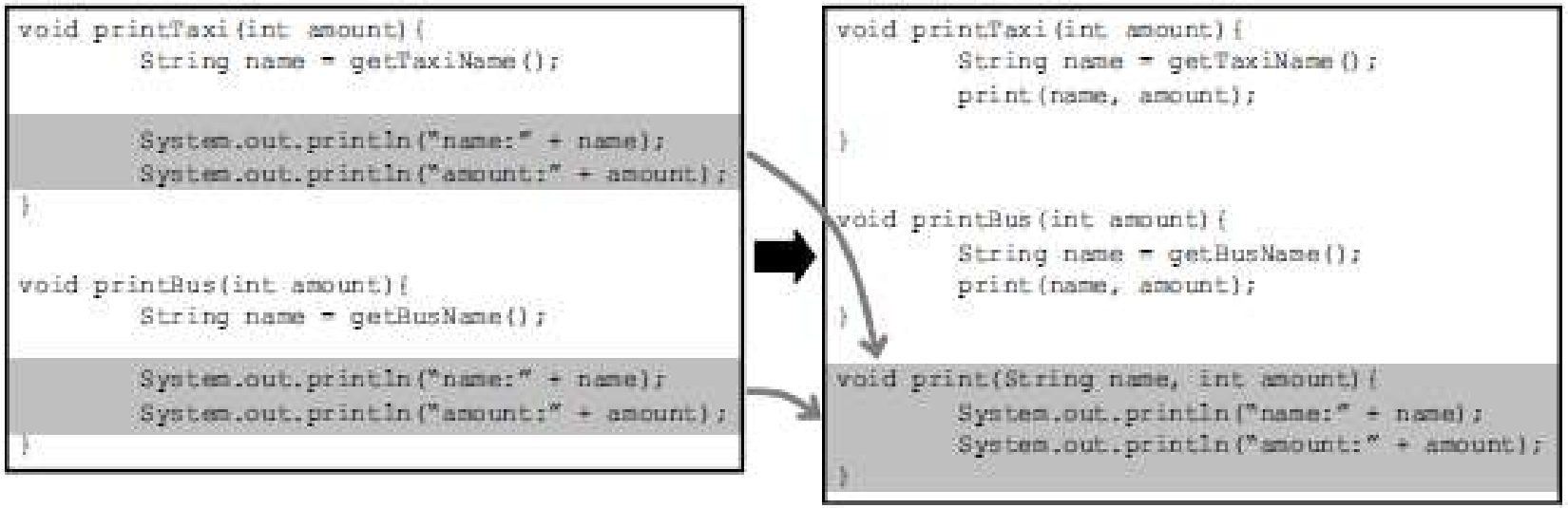


Figure 2: Example of Extract Method

Breaking a Method

```
int minimum (int a[ ], int from, int
to)
{
      int min = from;
      for (int i = from; i <= to; i++)
            if (a[i] < a[min]) min = i;
      return min;
}
```

```
void swap (int &x, int &y)
{
      int temp = x;
      x = y;
      y = temp;
}
```

```
for (i=0; i < N-1; i++) {
      min = i;
      for (j = i; j < N; j++)
            if (a[j] < a[min]) min = j;
      temp = a[i];
      a[i] = a[min];
      a[min] = temp;
}
```

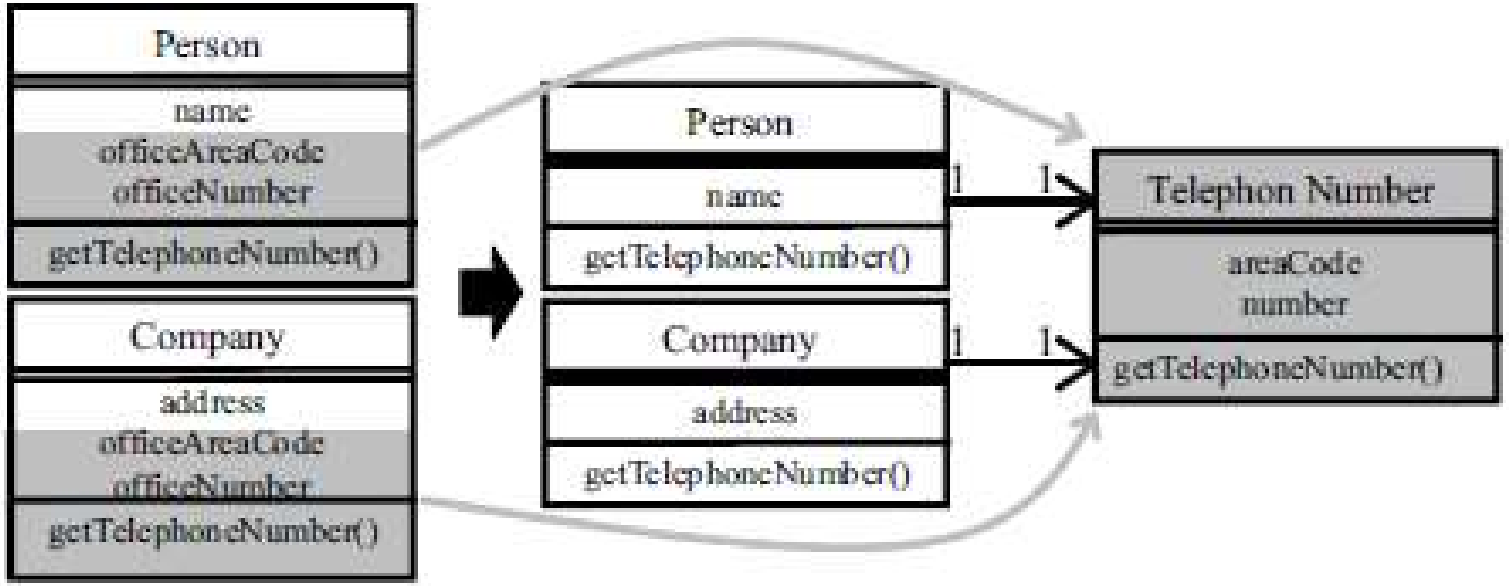
```
for (i=0; i < N-1; i++) {
      min = minimum (a, i, N-
      1);
      swap(a[i], a[min]);
}
```

Large Class

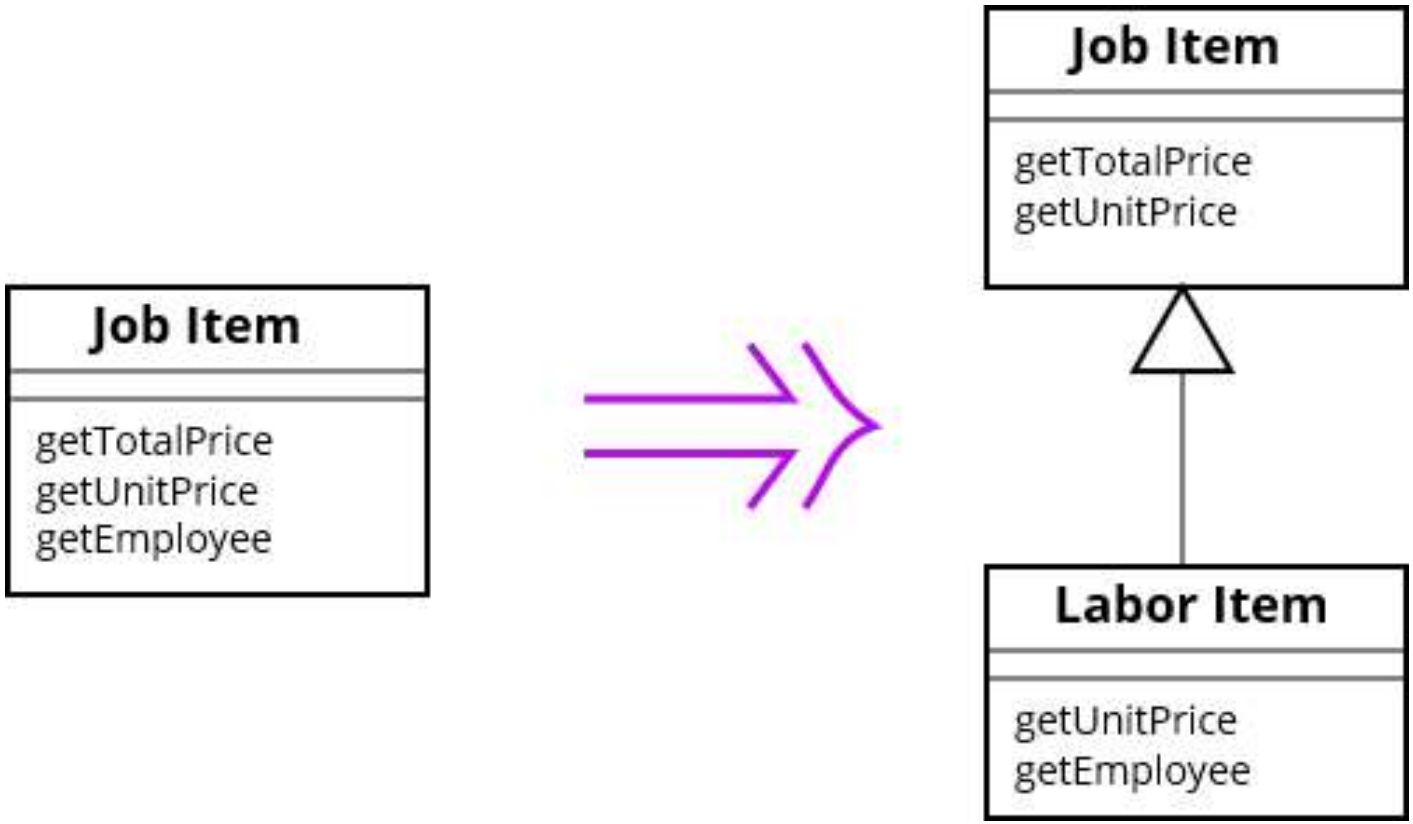
If a class is doing too much: has too many variables or too many methods. Large classes try to do too much, which reduces cohesion

- Extract Class - to bundle variables or methods.
- Extract Subclass – A class has features that are used only by some instances.
- Extract interface – determine how clients use the class. Provide ideas on breaking the class.
- Duplicate Observed Class – For a presentation class that includes domain functionality. Move functionality to a domain object. Set up an *Observer*.

Extract Class



Extract subclass



Long Parameter List

A method does not need many parameters, only enough to be able to retrieve what it needs. Long parameter lists are hard to understand and maintain. They can become inconsistent.

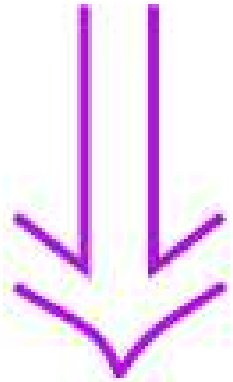
The clue – pass objects: Use objects for packing data.

Penalty – might increase dependency among objects.

- Replace Parameter with Method - parameters result from a method call on a reachable object → remove the parameters; let the object invoke the method.
- Preserve Whole Object – replace parameters that result from an object by the object itself.
- Introduce Parameter Object - turn several parameters into an object.

Preserve whole object

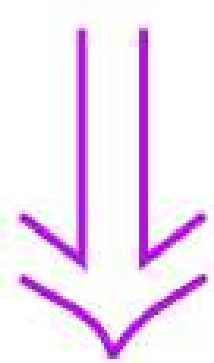
```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```



```
withinPlan = plan.withinRange(daysTempRange());
```

Replace parameter with method

```
int basePrice = _quantity * _itemPrice;
discountLevel = getDiscountLevel();
double finalPrice = discountedPrice (basePrice, discountLevel);
```



```
int basePrice = _quantity * _itemPrice;
double finalPrice = discountedPrice (basePrice);
```

Divergent Change

1. If you find yourself repeatedly changing the same class for different requirement variations – then there is probably something wrong with it.
2. A class should react to a single kind of variation – *cohesion principle*.
3. Deals with cohesion; symptom: one type of change requires changing one subset of methods; another type of change requires changing another subset

- Extract Class - group functionality commonly changed into a class

Shotgun Surgery

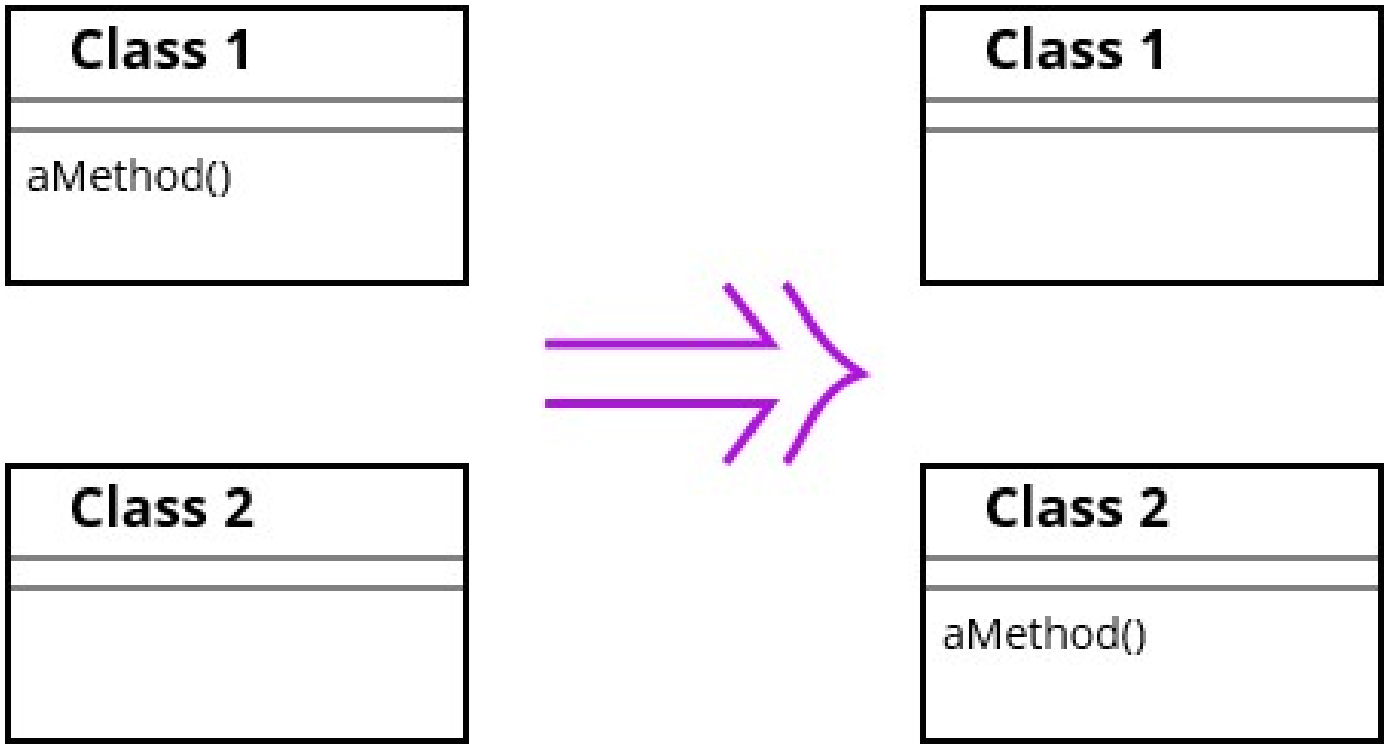
1. If you find yourself making a lot of small changes for each desired change.
2. Small changes are hard to maintain. a change requires lots of little changes in a lot of different classes

Opposite of divergent change. Ideal:

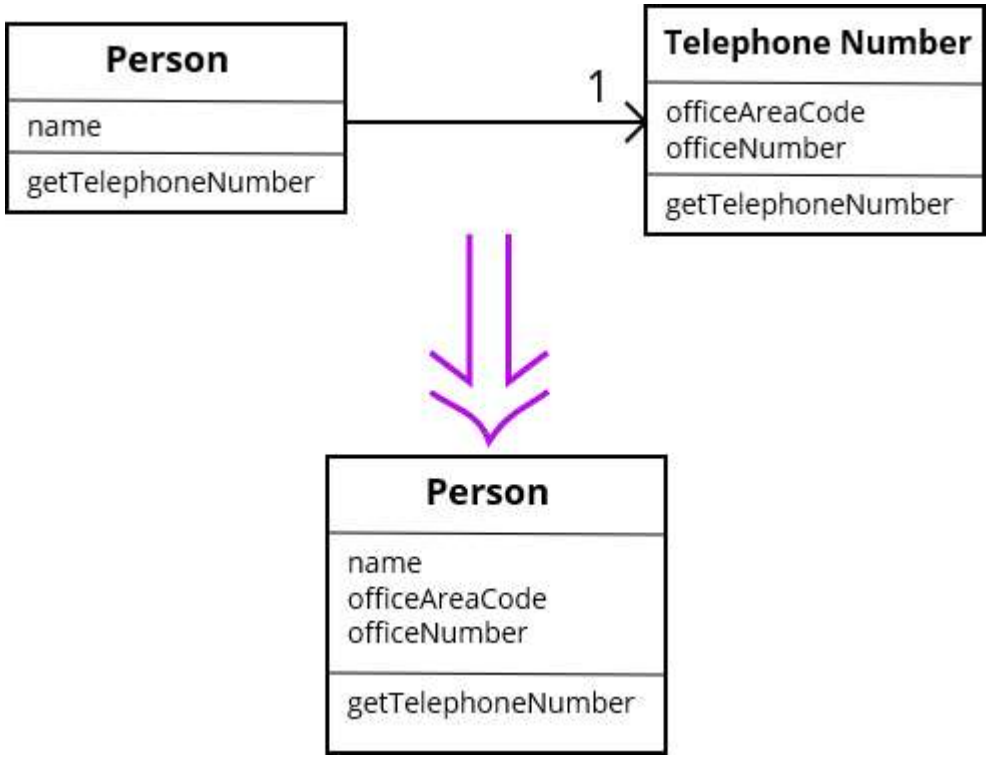
common changes ←→ classes is a 1:1 relationships.

- Move Method/Field - pull all the changes into a single class (existing or new).
- Inline Class - group a bunch of behaviors together in an existing class (might imply divergent change).

Move method



Inline Class



Feature Envy

1. A method requires lots of information from some other class (move it closer!)
2. If a method seems more interested in a class other than the class it actually is in – move it to where it belongs.
3. *Strategy* and *Visitor* break this rule – separate behavior from the data it works on. Answer the *Divergent Change* smell.

- Move Method - move the method to the desired class.
- Extract Method + Move Method - if only part of the method shows the symptoms Or if the method uses data from several classes.

Pull up method

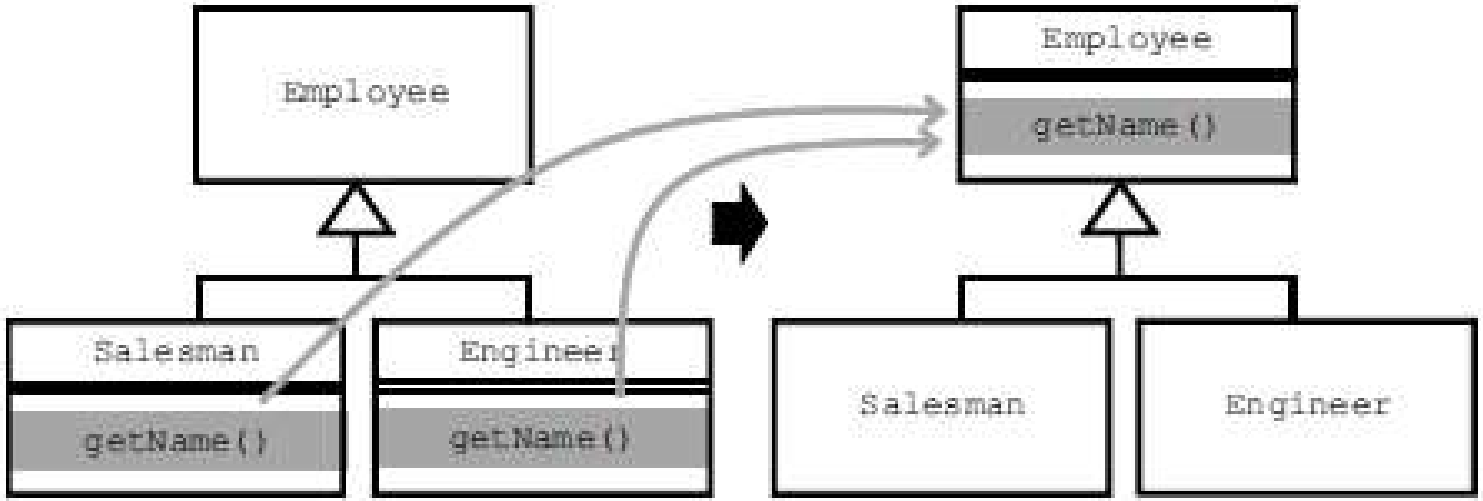


Figure 3: Example of Pull Up Method

Form Template Method

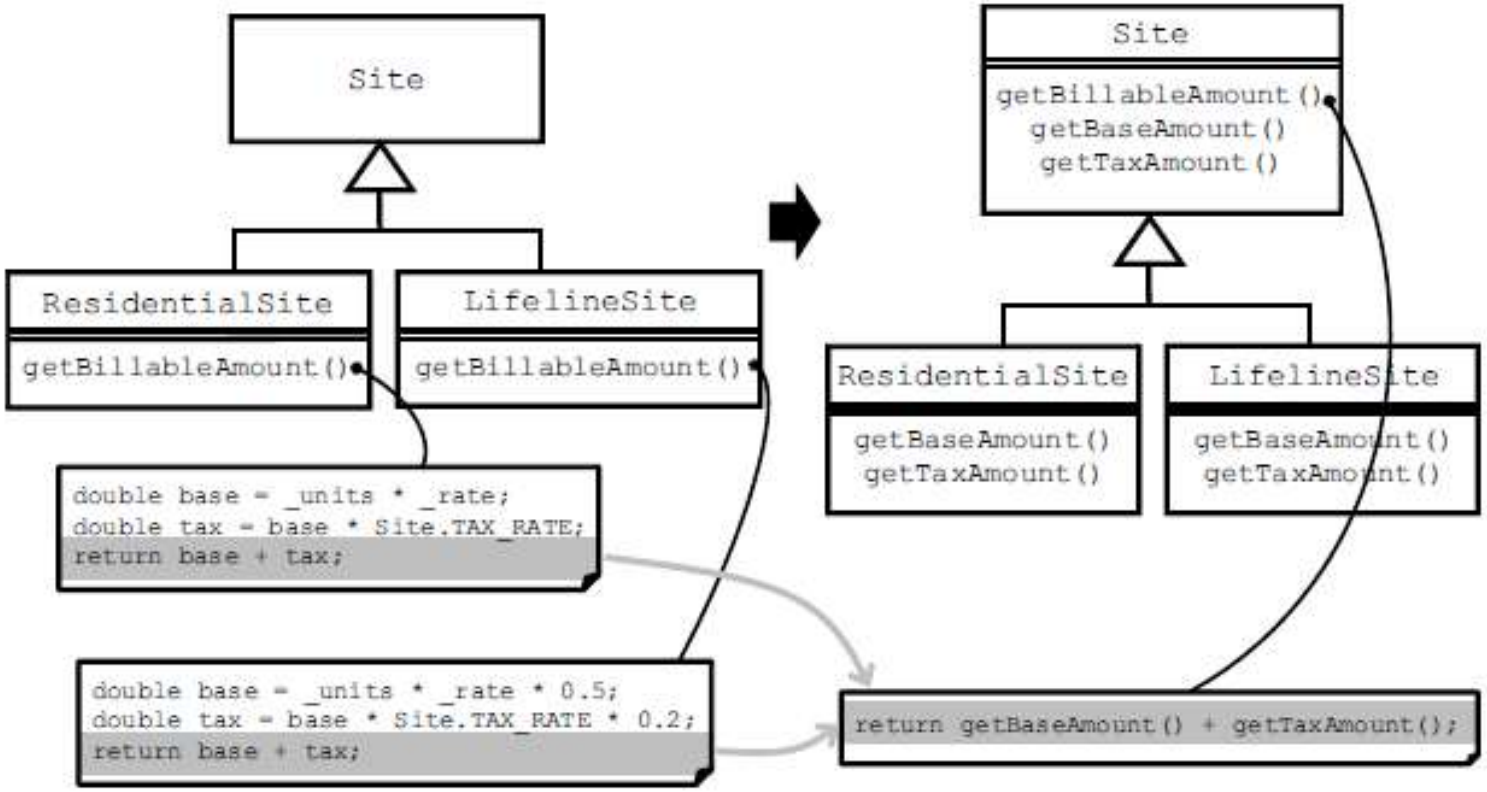


Figure 4: Example of Form Template Method

Push Down Method