Software Engineering

Key Refactorings

Software Engineering 2012-2013
Department of Computer Science Ben-Gurion university

Based on slides of: Mira Balaban Department of Computer Science Ben-Gurion university
F. Tip. IBM T J Watson Research Center.
Fowler’s Refactorings Catalogue (1)

- Composing Methods
  - Extract Method.
  - Inline Method.
  - Inline Temp.
  - Replace Temp with Query.
  - …
- Moving Features Between Objects.
  - Move Method.
  - Move Field.
  - Extract Class.
  - Inline Class.
  - …
- Organizing Data.
  - Replace Type Code with Class.
  - Replace type Code with Subclasses.
  - Replace type Code with State/Strategy.
  - …
Fowler’s Refactorings Catalogue (2)

- Simplifying Conditional Expressions.
  - Replace Conditional with Polymorphism
  - …

- Making Method Calls Simpler.
  - Rename Method.
  - Add Parameter.
  - Remove Parameter.
  - Introduce Parameter Object.
  - Replace Constructor with Factory Method.
  - …
Fowler’s Refactorings Catalogue (3)

- Dealing with Generalization.
  - Pull Up Field.
  - Pull Up Method.
  - Extract SuperClass.
  - Extract SubClass.
  - Extract Interface.
  - Form Template Method.
  - Replace Inheritance with Delegation.
  - ...

- Big Refactorings
  - Convert Procedural Design to Objects
  - Separate Domain from Presentation.
  - ...

Software Engineering, 2012
Key Refactorings
Key Refactorings

- Extract Method, Inline Method.
- Inline Temp, Replace Temp with Query.
- Replace Method with Method Object.
- Move Method, Move Field, Extract Class.
- Replace Type Code with Class, Replace Type Code with Subclasses, Replace Type Code with State/Strategy.
- Replace Conditional with Polymorphism
Extract Method

- Extracting a code fragment into a method
- goal:
  - enable reuse; avoid cut-and-paste programming
  - make long methods shorter and more readable

"If the code you want to extract is very simple, such as a single message or function call, you should extract it if the name of the new method will reveal the intention of the code in a better way. If you can't come up with a more meaningful name, don't extract the code."
Extract Method: Condition

• The extracted code is fully contained in an *enclosing lexical block*.
• The extracted code does **not** include a “return” statement.
• Consider:
  a. local variables to the source method
  b. parameters of the source method that occur in the extracted code, such that:
     • used in source method following the extracted code.
     • assigned in the extracted code
     → at most one such variable or parameter
Extract Method: Transformation (1)

- Two operations:
  - Add method.
  - Revise source and target methods code.

- Add a new method:
  - **Invent a new name** in the enclosing lexical scope. Pick a name that describes what the method does
  - **copy extracted code** from the source method into the new target method.
  - **Add the new method** in the enclosing lexical scope.
Extract Method: Transformation (2)

- Revise methods code: scan extracted code for
  1. variables local in scope to the source method
     • only used in target method: declare in target method
  2. variables local in scope to the source method, or parameters of the source method
     • (For variables only:) Declared and used outside the extracted code
     • (for variables and parameters:) used in the extracted code
        \Rightarrow pass in as parameters
  3. variables local in scope to the source method
     • Declared and used in the extracted code
     • Used outside the extracted code
        \Rightarrow declare outside the extracted code
        \Rightarrow Apply action (2).
Extract Method: Transformation (3)

- Revise methods code: scan extracted code for (cont)
  4. variables local in scope to the source method, or parameters of the source method
    - assigned in target method
    - used in source method following the extracted code
      - at most one variable can be modified
    \(\rightarrow\) return changed value
    \(\rightarrow\) Adapt return type of target method
  5. Extracted code replacement:
    - Case 4: \(\text{var} = \text{target method call.}\)
    - Otherwise: target method call.
void printOwing()
{
    // print banner
    Enumeration e = _orders.elements();
    Enumeration outstanding = 0.0;
    System.out.println("*************************");
    System.out.println("***** Customer Owes *****");
    System.out.println("*************************");

    // calculate outstanding
    while (e.hasMoreElements()){
        Order each = (Order)e.nextElement();
        outstanding += each.getAmount();
    }

    // print details
    System.out.println("name:" + _name);
    System.out.println("amount" + outstanding);
}
example (2)

```java
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    // print details
    System.out.println("name:" + _name);
    System.out.println("amount" + outstanding);
}

private void printBanner() {
    System.out.println("*************************");
    System.out.println("***** Customer Owes *****");
    System.out.println("*************************");
}
```
example (3)

```java
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    printDetails(outstanding);
}

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

public void printBanner() {
    // ...
}
```
example (4a)

```java
void printOwing() {
    double outstanding = 0.0;

    printBanner();
    outstanding = getOutstanding(outstanding);
    printDetails(outstanding);
}

private double getOutstanding(double outstanding) {
    Enumeration e = _orders.elements();
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }
    return outstanding;
}

private void printDetails(double outstanding) {
    ...
}
private void printBanner() {
    ...
}
```
example (4b)

```java
void printOwing() {
    double outstanding = 0.0;

    printBanner();
    outstanding = getOutstanding(outstanding);
    printDetails(outstanding);
}

private double getOutstanding(double outstanding) {
    Enumeration e = _orders.elements();
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }
    return outstanding;
}

private void printDetails(double outstanding) {
    ...
}
private void printBanner() {
    ...
}
```

Remove Assignments to Parameters refactoring
example (4c)

```java
void printOwing() {
    double outstanding = 0.0;
    printBanner();
    outstanding = getOutstanding(outstanding);
    printDetails(outstanding);
}

private double getOutstanding(double initialValue) {
    double result = initialValue;
    Enumeration e = _orders.elements();
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        result += each.getAmount();
    }
    return result;
}

private void printDetails(double outstanding) { ... }
private void printBanner() { ... }
```

In this case the outstanding variable is initialized only to an obvious initial value (0.0)

\(\Rightarrow\) We can initialize it only within the extract method.
variant: computed previous value

```java
void printOwing(double previousAmount) {
    Enumeration e = orders.elements();
    double outstanding = previousAmount * 1.2;

    printBanner();

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    printDetails(outstanding);
}
private void printDetails(double outstanding) {
    ...
}
private void printBanner() {
    ...
}
```
variant, refactored

```java
void printOwing(double previousAmount) {
    double outstanding = previousAmount * 1.2;
    printBanner();
    outstanding = getOutstanding(outstanding);
    printDetails(outstanding);
}

private double getOutstanding(double initialValue) {
    double result = initialValue;
    Enumeration e = _orders.elements();
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        result += each.getAmount();
    }
    return result;
}

private void printDetails(double outstanding) {
}

private void printBanner() {
}
```
Extract Method: Considerations

• if the extracted code changes multiple local variables:
  • pick different code to extract
    • aim for multiple methods, each with one return value
  • try applying Replace Temp With Query
    • in essence, recomputing the value each time
      • not always (directly) possible
      • performance issues
  • …or bring in the heavy machinery: Replace Method with Method Object
    • turns the method into its own object
Problem case

```java
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;
    int count = 0;

    printBanner();

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
        count = count + 1;
    }

    printDetails(outstanding, count);
}
```

```java
private void printDetails(double outstanding, int count) {
    ...
}
```

```java
private void printBanner() {
    ...
}
```

the extracted code changes multiple local variables
Inverse operation of Extract Method: Inline method

- replace a method call with the body of the called method
  - commonly applied by optimizing compilers
  - this is the inverse operation of Extract Method

- when to apply:
  - as an intermediate step when a group of methods is badly factored
  - when too much delegation makes code hard to understand

- various complicating issues:
  - polymorphic call sites
  - recursion
  - multiple return points
Inline method

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

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

**Mechanics**

- **Check** that the method is not polymorphic.
  - Don’t inline if subclasses override the method; they cannot override a method that isn’t there.
- Find all calls to the method.
- Replace each call with the method body.
- Compile and test.
- Remove the method definition.
Inline Temp

• you want to eliminate a temporary variable that is used to hold the result of a method call

• mechanics:
  • declare temp as final & compile
    • compiler will alert you if there are multiple assignments
  • replace all references to the temp with calls to that method.

• complicating issues:
  • Method call may have side-effects
    • try applying Separate Query from Modifier
  • Method call may have a different value at the places where the temp is used
  • temps that are assigned more than once (in loops)
  • performance issues
Example: Inline Temp

```java
boolean checkBasePrice() {
    int basePrice = anOrder.basePrice();
    return (basePrice > 1000);
}
```

```
boolean checkBasePrice() {
    final int basePrice = anOrder.basePrice();
    return (basePrice > 1000);
}
```

```
boolean checkBasePrice() {
    return (anOrder.basePrice() > 1000);
}
```

```java
...
    (anOrder.basePrice() > 1000);
...
Replace Temp with Query

- you want to eliminate a temporary variable that is used to hold the result of an expression
- mechanics:
  - declare temp as `final` & compile
    - compiler will alert you if there are multiple assignments
  - extract the expression into a method.
  - replace all references to the temp with calls to that method.
- complicating issues (same as in Inline temp):
  - expression may have side-effects
    - try applying Separate Query from Modifier
  - expression may have a different value at the places where the temp is used
  - temps that are assigned more than once (in loops)
  - performance issues
Example: Replace Temp with Query

double getPrice() {
    int basePrice = _quantity * _itemPrice;
    double discountFactor;
    if (basePrice > 1000)
        discountFactor = 0.95;
    else
        discountFactor = 0.98;
    return basePrice * discountFactor;
}

Example (2)

double getPrice() {
    final int basePrice = _quantity * _itemPrice;
    double discountFactor;
    if (basePrice > 1000)
        discountFactor = 0.95;
    else
        discountFactor = 0.98;
    return basePrice * discountFactor;
}

private int basePrice() {
    return _quantity * _itemPrice;
}
Example (2)

double getPrice() {
    final int basePrice = basePrice();
    double discountFactor;
    if (basePrice > 1000)
        discountFactor = 0.95;
    else
        discountFactor = 0.98;
    return basePrice * discountFactor;
}

private int basePrice() {
    return _quantity * _itemPrice;
}
Example (3)

double getPrice() {
    final int basePrice = basePrice();
    double discountFactor;
    if (basePrice() > 1000)
        discountFactor = 0.95;
    else
        discountFactor = 0.98;
    return basePrice() * discountFactor;
}

public int basePrice() {
    return _quantity * _itemPrice;
}
```java
double getPrice() {
    final int basePrice = basePrice();
    double discountFactor;
    if (basePrice() > 1000) {
        discountFactor = 0.95;
    } else {
        discountFactor = 0.98;
    }
    return basePrice() * discountFactor;
}

public int basePrice() {
    return _quantity * _itemPrice;
}
```

**Example (3)**

- **Remove temp Declaration**: The temporary variable `discountFactor` can be removed.
- **Repeat the process for discountFactor**: The discount factor can be calculated directly using an if-else statement instead of a separate variable declaration.
Example (4)

double getPrice() {
   return basePrice() * discountFactor();
}

private double discountFactor() {
   double discountFactor;
   if (basePrice() > 1000)
      discountFactor = 0.95;
   else
      discountFactor = 0.98;
   return discountFactor;  \textbf{extract the expression} into a method
                           \textbf{Conditional expression}
}

private int basePrice() {
   return _quantity * _itemPrice;
}
Example (5)

double getPrice() {
    return basePrice() * discountFactor();
}

private double discountFactor() {
    return (basePrice() > 1000) ? 0.95 : 0.98;
}

private int basePrice() {
    return _quantity * _itemPrice;
}
Replace Temp with Query – compositional view

- Consists of three primitive refactorings:
  - Finalize temp.
  - Extract method
  - Inline temp.

Mechanics of Replace Temp with Query:
- declare temp as `final` & compile
- extract the expression into a method.
- replace all references to the temp with calls to that method.
Replace Method with Method Object

- you have a long method that uses local variables in such a way that you cannot apply Extract Method

- Turn the method into its own object so that:
  - local variables become fields
  - you can then decompose the method into other methods on the same object
Replace Method with Method Object

class Order...
   double price() {
   double primaryBasePrice;
   double secondaryBasePrice;
   double tertiaryBasePrice;
   // long computation;
   ...
   
   return new PriceCalculator(this).compute();
Creating a Method Object: Mechanics

- **Create a new class**, name it after the method. This class should contain:
  - **final field** for the “source object” that hosted the original method
  - **field** for each temporary variable and parameter in the method
  - **constructor** that takes the source object and each parameter; initializes corresponding fields
  - a new method called compute() with the same return type as the source method

- **copy body of original method** into compute()
  - use source object field for any invocations of methods on the original object

- **Compile.**

- **Replace the old method** with one that creates the new object and calls compute.
Example

```java
public class Account {

    int gamma(int inputValue, int quantity, int yearToDate) {
        int importantValue1 = inputValue * quantity + delta();
        int importantValue2 = inputValue * yearToDate + 100;
        if ((yearToDate - importantValue1) > 100) {
            importantValue2 -= 20;
            importantValue1 += 20;
        }
        int importantValue3 = importantValue2 * 7;
        return importantValue3 - 2 * importantValue1;
    }
}
```
Step 1: create a new class

- named after the original method
- field for “source object” of type Account
- field for each parameter/temp

```java
class Gamma {
    private final Account _account;
    private int inputVal;
    private int quantity;
    private int yearToDate;
    private int importantValue1;
    private int importantValue2;
    private int importantValue3;
}
```
Step 2: add constructor & compute()

class Gamma {
    public Gamma(Account source, int inputValArg,
                  int quantityArg, int yearToDateArg) {
        _account = source;
        inputVal = inputValArg;
        quantity = quantityArg;
        yearToDate = yearToDateArg;
    }
    public int compute() { ... }
    private final Account _account;
    private int inputVal;
    private int quantity;
    private int yearToDate;
    private int importantValue1;
    private int importantValue2;
    private int importantValue3;
}

Step 3: move code into compute() & update source method

```java
public class Account {
    int gamma(int inputVal, int quantity, int yearToDate) {
        return new Gamma(this, inputVal, quantity, yearToDate).compute();
    }
}

class Gamma {
    public Gamma(Account source, int inputValArg, int quantityArg, int yearToDateArg) {...
    
    public int compute() {
        importantValue1 =
            (inputVal*quantity) + account.delta();
        importantValue2 = (inputVal*yearToDate) + 100;
        if ((yearToDate - importantValue1) > 100) {
            importantValue2 -= 20;
            importantValue1 += 20;
        }
        int importantValue3 = importantValue2 * 7;
        return importantValue3 - 2*importantValue1;
    }
    ...
```
Decompose source method: Apply Extract Method

class Gamma {
    public Gamma(Account source, int inputValArg,
                  int quantityArg, int yearToDateArg) {...}

    public int compute() {
        importantValue1 =
            (inputVal*quantity) + _account.delta();
        importantValue2 = (inputVal*yearToDate) + 100;
        importantThing();
        int importantValue3 = importantValue2 * 7;
        return importantValue3 - 2*importantValue1;
    }

    private void importantThing(){
        if ( (yearToDate - importantValue1) > 100)
            {importantValue2 -= 20;
             importantValue1 += 20;}
    }

    ...
}
Move Method

- A method is using or is used by more features of another class than the class on which it is defined.
- Create a new method with a similar body in the class it uses most.
  - either turn the old method into a simple delegation, or remove it altogether.
Move Method - thoughts

• Moving methods is the bread and butter of refactoring.

• Move methods when
  • classes have too much behavior
  • classes are collaborating too much and are too highly coupled.

  o Once I see a likely method to move, I take a look at the methods that call it, the methods it calls, and any redefining methods in the hierarchy.

  o I assess whether to go ahead on the basis of the object with which the method seems to have more interaction.

  o It's not always an easy decision to make.

    If I am not sure whether to move a method, I go on to look at other methods. ..

    Sometimes the decision still is hard to make. ..

    If it is difficult to make the decision, it probably does not matter that much.

    Then I choose according to instinct; after all, I can always change it again later.
Move Method: Mechanics

- Examine features used by the source method that are defined on the source class. Consider whether they should be moved as well.
- check for overriding definitions in subclasses/superclasses
  - in general, cannot move if there is polymorphism
- declare method in the target class (choose different name?)
  - copy code from source method
  - adjust references to source
- determine how to access target object from source
- turn source method into delegating method
  - removal is optional
  - If you remove the source method, replace all the references with references to the target method.

Compile target class

Compile and Test
Move Method: Example

```java
public class Account {
    double overdraftCharge() {
        if (_type.isPremium()) {
            double result = 10;
            if (_daysOverdrawn > 7)
                result += (_daysOverdrawn - 7) * 0.85;
            return result;
        } else return _daysOverdrawn * 1.75;
    }

    double bankCharge() {
        double result = 4.5;
        if (_daysOverdrawn > 0)
            result += overdraftCharge();
        return result;
    }

    private AccountType _type;
    private int _daysOverdrawn;
}
```

move this method into class AccountType

...there are going to be several new account types, each of which has its own rule for calculating the overdraft charge
Example

• observations
  • reference to field _daysOverdrawn in source class;
    • we assume that this field needs to stay in source class
  • source class already has pointer to target class

• options for making a feature available to the moved method in its new position:
  1. move it to the target class as well
  2. create or use a reference to the source class from the target class
  3. pass the whole source object as a parameter
     • needed if moved method calls methods on source object
  4. if the feature is a field, pass it as a parameter
Move Method: Example (2)

class AccountType {
    boolean isPremium() { ... }

    double overdraftCharge(int daysOverdrawn) {
        if (isPremium()) {
            double result = 10;
            if (daysOverdrawn > 7)
                result += (daysOverdrawn - 7) * 0.85;
            return result;
        } else
            return daysOverdrawn * 1.75;
    }
}

Option 4

remove the _type from uses of features of the account type

copy the method body over to the account type and get it to fit.
Move Method: Example (3)

- replace source method body with delegation
- the code now compiles and can be used as-is

```java
public class Account {

    double overdraftCharge() {
        return _type.overdraftCharge(_daysOverdrawn);
    }

    double bankCharge() {
        double result = 4.5;
        if (_daysOverdrawn > 0)
            result += overdraftCharge();
        return result;
    }

    private AccountType _type;
    private int _daysOverdrawn;
}
```
Move Method: Example (4)

- removing the source method requires updating all references to it (apply Inline Method)

```java
public class Account {
    double bankCharge() {
        double result = 4.5;
        if (_daysOverdrawn > 0)
            result += _type.overdraftCharge(_daysOverdrawn);
        return result;
    }

    private AccountType _type;
    private int _daysOverdrawn;
}

class AccountType { ... }

double overdraftCharge() {
    return _type.overdraftCharge(_daysOverdrawn);
}
```
Move Method: Related Refactorings

- **Move Field**
  - move a field from source class to target class
  - similar issues

- **Extract Class**
  - break up a single class into two classes
    - create a new class that will contain some of the functionality of the source class
    - create link between source and target class (e.g., in constructor of source class)
    - move functionality to target class with repeated applications of Move Method and Move Field
Refactorings for Eliminating Type Codes

• used for restructuring “procedural” programs that use type codes
• Replace Type Code with Class
  • for modeling enumerated types
  • assumes no conditional logic involving type codes
• Replace Type Code with Subclasses
  • conditional logic involving type codes
  • assumes type code is frozen at object creation time
  • enables use of Replace Conditional with Polymorphism to replace conditional logic with dynamic dispatch
• Replace Type Code with State/Strategy
  • can accommodate more dynamic situations where the type code changes at run-time
Replace TC with Class

- Symbolic name is only an alias
  - The compiler still sees the underlying number.
  - The compiler type checks using the number not the symbolic name.
- Any method that takes the type code as an argument expects a number, and there is nothing to force a symbolic name to be used.
  - This can reduce readability and be a source of bugs.
Replace TC with Class: Mechanics

1. **create a new class** for the type code
   - with code field that matches type code
   - static variables for allowable instances
   - static method for retrieving appropriate instance given type code

2. **modify implementation** of source class to use the constants defined in new class
   - maintain old type-code based interface

3. **update methods on source class** that use the type code so they use the new class instead
   - apply Rename Method to update names of accessor methods
   - add accessor method that uses new class
   - add constructor that uses new class

4. **update clients** of source class, one by one

5. after updating all clients, **remove old interface**
Replace TC with Class: Example

```java
public class Person {
    public static final int O = 0;
    public static final int A = 1;
    public static final int B = 2;
    public static final int AB = 3;
    private int _bloodGroup;

    public Person(int bloodGroup) {
        _bloodGroup = bloodGroup;
    }

    public void setBloodGroup(int arg) {
        _bloodGroup = arg;
    }

    public int getBloodGroup() {
        return _bloodGroup;
    }
}
```
Step 1: create new class BloodGroup

class BloodGroup {
    public static final BloodGroup O = new BloodGroup(0);
    public static final BloodGroup A = new BloodGroup(1);
    public static final BloodGroup B = new BloodGroup(2);
    public static final BloodGroup AB = new BloodGroup(3);
    private static final BloodGroup[] _values = {O, A, B, AB};

    private final int _code;
    private BloodGroup (int code){
        _code = code;
    }
    public int getCode(){
        return _code;
    }
    public static BloodGroup code(int arg){
        return _values[arg];
    }
}

used for the intermediate stage of refactoring
Step 2: update code in Person to use constants in class BloodGroup

```java
public class Person {
    public static final int O = BloodGroup.O.getCode();
    public static final int A = BloodGroup.A.getCode();
    public static final int B = BloodGroup.B.getCode();
    public static final int AB = BloodGroup.AB.getCode();
    private BloodGroup _bloodGroup;
    public Person(int bloodGroup) {
        _bloodGroup = BloodGroup.code(bloodGroup);
    }
    public void setBloodGroup(int arg) {
        _bloodGroup = BloodGroup.code(arg);
    }
    public int getBloodGroup() {
        return _bloodGroup.getCode();
    }
}
```
Step 3: rename old accessor methods; add new constructor & accessors

```java
public class Person {
    
    // old constructor
    public Person(int bloodGroup){
        _bloodGroup = BloodGroup.code(bloodGroup);
    }
    
    // old accessor (renamed)
    public void setBloodGroupCode(int arg){
        _bloodGroup = BloodGroup.code(arg);
    }
    
    // new constructor
    public Person(BloodGroup bloodGroup){
        _bloodGroup = bloodGroup;
    }
    
    // new accessor
    public void setBloodGroup(BloodGroup arg){
        _bloodGroup = arg;
    }
    
    ...
}
```
Step 5: after updating all clients (step 4), remove all type-code based interface

```java
public class Person {
    private BloodGroup _bloodGroup;
    public Person(BloodGroup bloodGroup) {
        _bloodGroup = bloodGroup;
    }
    public void setBloodGroup(BloodGroup arg) {
        _bloodGroup = arg;
    }
}

class BloodGroup { ... }
```
Replace TC with Subclasses

- If the type code affects behavior - use polymorphism to handle the variant behavior.
- This situation usually is indicated by the presence of case-like conditional statements.
Replace TC with Subclasses

- apply when you have an immutable type code with conditional control flow based on its value

- mechanics:
  1. preparation:
     - redirect access to type-code field via getter/setter method
       (see: Self-Encapsulate Field)
     - if a constructor uses type code as a parameter, replace it with a factory method
       (see Replace Constructor with Factory Method)
  2. for each value of the type code, create new subclass
  3. override getting method in each subclass to return the relevant value
Replace TC with Subclasses: Example

```java
public class Employee {
    private int _type;
    static final int ENGINEER = 0;
    static final int SALESMAN = 1;
    static final int MANAGER = 2;

    Employee(int type){    // constructor uses type code as a parameter
        _type = type;
    }
}
```
step 1: preparation

public class Employee {
    private int _type;
    static final int ENGINEER = 0;
    static final int SALESMAN = 1;
    static final int MANAGER = 2;

    // add getter method – “Self Encapsulate Field”
    int getType(){ return _type; }

    // factory method to replace constructor
    static Employee create(int type){
        return new Employee(type);
    }
    // constructor now made private
    private Employee(int type){_type = type;}
}
step 2: introduce subclasses

```java
public abstract class Employee {
    static final int ENGINEER = 0;
    static final int SALESMAN = 1;
    static final int MANAGER = 2;
    abstract int getType();
    static Employee create(int type){
        switch(type){
            case ENGINEER: return new Engineer();
            case SALESMAN: return new Salesman();
            case MANAGER: return new Manager();
            default: throw new IllegalArgumentException("incorrect type value");
        }
    }
}

class Engineer extends Employee {
    int getType(){ return Employee.ENGINEER; }
}

class Salesman extends Employee { ... }

class Manager extends Employee { ... }
```
Replace TC with State/Strategy (1)

- This is similar to Replace Type Code with Subclasses
  used if:
  - the type code changes during the life of the object
  - another reason prevents subclassing
- It uses either the state or strategy pattern
Replace TC with State/Strategy (3)

- apply when you have a mutable type code with conditional control flow based on its value

- mechanics:
  - redirect access to type-code field via getter/setter method (see: Self-Encapsulate Field)
  - create State class, name it after purpose of type code with abstract query to return the type code (getTypeCode())
  - add a subclass of the State for each type code
    - override query method of State class
Replace TC with State/Strategy (2)

- Mechanics (continued):
  - Add type code information to **State class**
    - Static variables for code values.
    - Create a factory method for State subclasses.
    - Change type query methods (getTypeCode) in State subclasses.
  - Update clients of source class, one by one.
  - update source class
    - create field to refer to State object
    - update TC query method to delegate to State object
    - update TC setting method to change State
Replace TC with State/Strategy:
Example

```java
public class Employee {
    private int _type;
    static final int ENGINEER = 0;
    static final int SALESMAN = 1;
    static final int MANAGER = 2;

    Employee(int type){ _type = type; }

    int payAmount(){
        switch (_type){
            case ENGINEER: return _monthlySalary;
            case SALESMAN: return _monthlySalary+_commission;
            case MANAGER: return _monthlySalary + _bonus;
            default: throw new RuntimeException("Incorrect employee");
        }
    }
}
```
Step 1: Self-encapsulate field

```java
public class Employee {
    private int _type;
    static final int ENGINEER = 0;
    static final int SALESMAN = 1;
    static final int MANAGER = 2;

    Employee(int type) { setType(type); }
    int getType() { return _type; }
    void setType(int type) { _type = type; }

    int payAmount() {
        switch (getType()) {
            case ENGINEER: return _monthlySalary;
            ...
        }
    }
}
```
Step 2: Declare State Class, subclasses, and query methods

```java
abstract class EmployeeType {
    abstract int getTypeCode();
}
class Engineer extends EmployeeType {
    int getTypeCode(){ return Employee.ENGINEER; }
}
class Manager extends EmployeeType {
    int getTypeCode(){ return Employee.MANAGER; }
}
class Salesman extends EmployeeType {
    int getTypeCode(){ return Employee.SALESMAN; }
}
```
Step 3: Copy type definitions to State class; introduce factory method

abstract class EmployeeType {
    // add factory method
    static EmployeeType newType(int code) {
        switch (code) {
            case ENGINEER: return new Engineer();
            case SALESMAN: return new Salesman();
            case MANAGER: return new Manager();
            default: { throw new IllegalArgumentException("bad code"); }
        }
    }

    static final int ENGINEER = 0;
    static final int SALESMAN = 1;
    static final int MANAGER = 2;
    abstract int getTypeCode();
}

class Engineer extends EmployeeType {
    int getTypeCode() { return ENGINEER; }
}

Static variables for code values.
Create a factory method for State subclasses.
Change type query methods (getTypeCode) in State subclasses.
Step 4: update source class (Employee)

```java
public class Employee {
    private EmployeeType _type;

    int getType() {
        return _type.getTypeCode();
    }

    void setType(int arg) { // change state
        _type = EmployeeType.newType(arg);
    }

    int payAmount() {
        switch (getType()) {
            // update references to constants
            case EmployeeType.ENGINEER:
                return _monthlySalary;
            case EmployeeType.SALESMAN:
                return _monthlySalary + _commission;
            case EmployeeType.MANAGER:
                return _monthlySalary + _bonus;
            default:
                throw new RuntimeException( "Incorrect employee");
        }
    }
}
```

- create field to refer to State object
- update TC query method to delegate to State object
- update TC setting method to change State
Replace Conditional with Polymorphism

double getSpeed() {
    switch (_type) {
    case EUROPEAN:
        return getBaseSpeed();
    case AFRICAN:
        return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
    case NORWEGIAN_BLUE:
        return (_isNailed) ? 0 : getBaseSpeed(_voltage);
    }
    throw new RuntimeException("Should be unreachable");
}
Replace Conditional with Polymorphism

- often used in combination with Replace TC with Subclasses and Replace TC with State/Strategy
  - moves each “leg” of conditional to an overriding method in a subclass
- mechanics
  - if necessary, use Move Method to place the conditional in a method at the top of the inheritance structure
    - may require adding get/set methods for fields of source class accessed in conditional
  - pick a subclass, create overriding definition and move appropriate leg of conditional into it
    - may require making private members of superclass protected
Step 1: Move `payAmount()` to `EmployeeType`

```java
abstract class EmployeeType {

    // moved to EmployeeType
    int payAmount(Employee emp) {
        switch (getTypeCode()) {
            case ENGINEER:
                return emp.getMonthlySalary();
            case SALESMAN:
                return emp.getMonthlySalary() + emp.getCommission();
            case MANAGER:
                return emp.getMonthlySalary() + emp.getBonus();
            default: throw new RuntimeException("Incorrect employee");
        }
    }
}
```
Step 2: Delegate `payAmount()` to `EmployeeType`

```java
public class Employee {
    private EmployeeType _type;
    int getType(){ return _type.getTypeCode(); } 
    void setType(int arg){ // change state
        _type = EmployeeType.newType(arg);
    }

    // now delegates to EmployeeType
    int payAmount(){ return _type.payAmount(this); }

    private int _monthlySalary;
    private int _commission;
    private int _bonus;

    // accessors added
    int getMonthlySalary(){ return _monthlySalary; }
    int getCommission(){ return _commission; }
    int getBonus(){ return _bonus; }
}
```
Step 3: Now start overriding payAmount()

abstract class EmployeeType {
    ...

    int payAmount(Employee emp){
        switch (getTypeCode()){
            case ENGINEER: throw new RuntimeException("should be overridden");
            case SALESMAN: return emp.getMonthlySalary() + emp.getCommission();
            case MANAGER: return emp.getMonthlySalary() + emp.getBonus();
            default: throw new RuntimeException("Incorrect employee");
        }
    }
}

class Engineer extends EmployeeType {
    int payAmount(Employee emp){
        return emp.getMonthlySalary();
    }
}
Step 4: Final situation, after removing body of method in State class

abstract class EmployeeType {
    abstract int payAmount(Employee emp);
}

class Engineer extends EmployeeType {
    int payAmount(Employee emp){
        return emp.getMonthlySalary();
    }
}

class Manager extends EmployeeType {
    int payAmount(Employee emp){
        return emp.getMonthlySalary() + emp.getCommission();
    }
}

class Salesman extends EmployeeType {
    int payAmount(Employee emp){
        return emp.getMonthlySalary() + emp.getBonus();
    }
}