

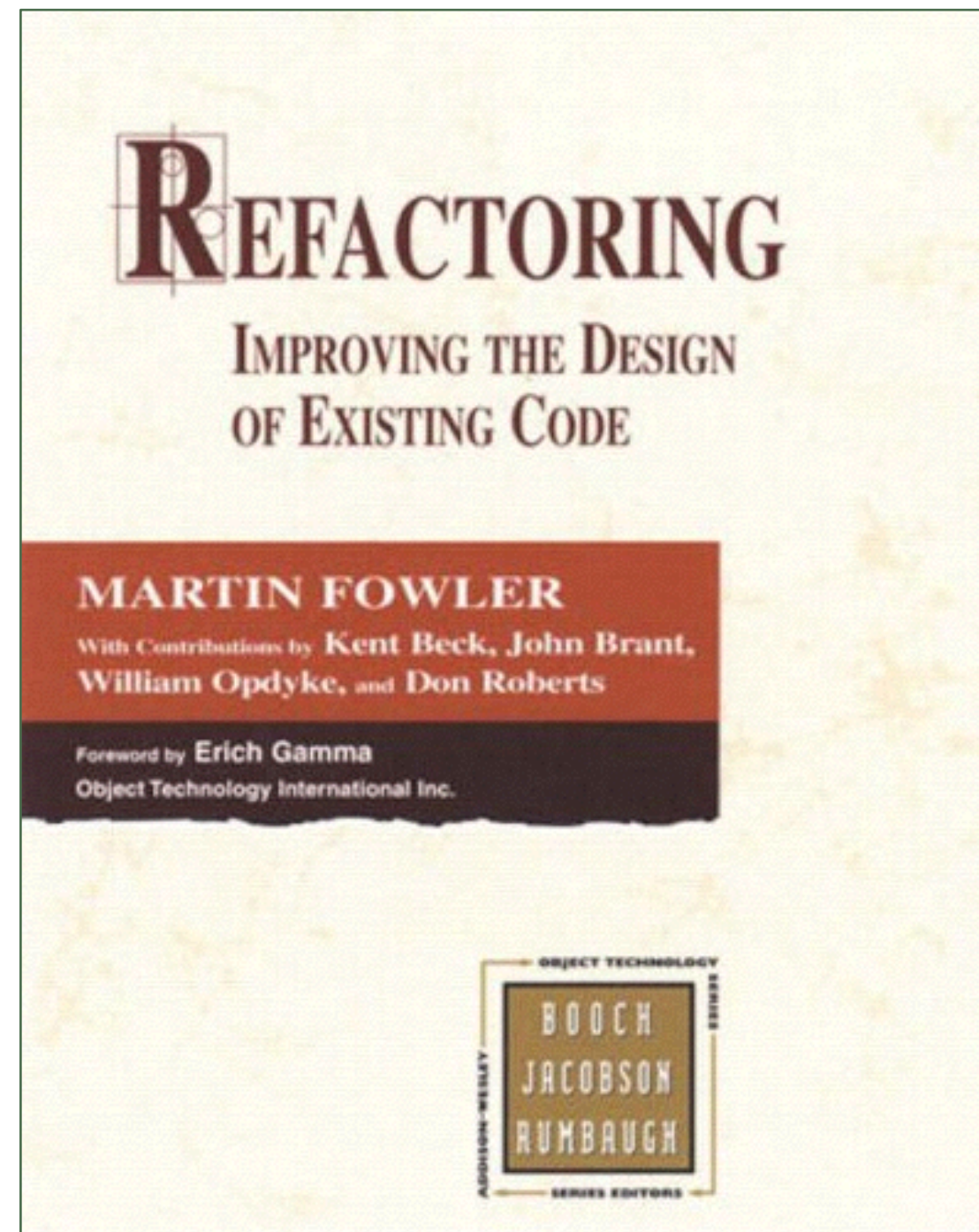
# REFACTORING

---

Allowing design to emerge!

# Good sources...

---



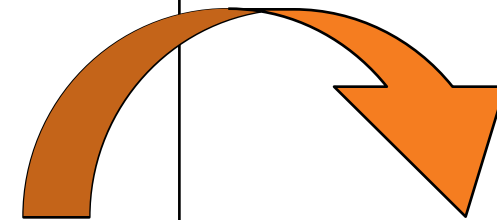
# Overview

---

- Introduction
- What is a “Code Smell”?
- Examples...
- What do you do in response to a code smell? Refactor!
- What is a “refactoring”?
  - When to do it... when not to do it...
- Good refactoring process
- Examples...
- A little practice
- Resources & Summary

# Change is constant

```
public String printStatement() {  
    double totalAmount = 0;  
    Enumeration rentals = _rentals.elements();  
    while (rentals.hasMoreElements()) {  
        double thisAmount = 0;  
        Rental each = (Rental) rentals.nextElement();  
  
        //determine amount for each movie  
        switch (each.getMovie().getPriceCode()) {  
            case Movie.REGULAR:  
                thisAmount += 2;  
                if (each.getDaysRented() > 2)  
                    thisAmount += (each.getDaysRented() - 2) * 1.5;  
                break;  
            case Movie.NEW_RELEASE:  
                thisAmount += each.getDaysRented() * 3;  
                break;  
        }  
  
        totalAmount += thisAmount;  
    }  
  
    return "Amount owed: " + totalAmount + "  
}
```





# ...becomes:

```
public String printStatement() {
    double totalAmount = 0;
    Enumeration rentals = _rentals.elements();
    while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();

        //determine amount for each movie
        switch (each.getMovie().getPriceCode()) {
            case Movie.REGULAR:
                thisAmount += 2;
                if (each.getDaysRented() > 2)
                    thisAmount += (each.getDaysRented() - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                thisAmount += each.getDaysRented() * 3;
                break;
        }

        totalAmount += thisAmount;
    }

    return "Amount owed: " + totalAmount + "\n";
}
```

frequent renter  
new release bonus

new reporting  
on rentals

```
public String printStatement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + "\n";

    while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();

        //determine amounts for each line
        switch (each.getMovie().getPriceCode()) {
            case Movie.REGULAR:
                thisAmount += 2;
                if (each.getDaysRented() > 2)
                    thisAmount += (each.getDaysRented() - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                thisAmount += each.getDaysRented() * 3;
                break;
            case Movie.CHILDRENS:
                thisAmount += 1.5;
                if (each.getDaysRented() > 3)
                    thisAmount += (each.getDaysRented() - 3) * 1.5;
                break;
        }

        // add frequent renter points
        frequentRenterPoints++;

        // add bonus for a two day new release rental
        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) && each.getDaysRented() > 1)
            frequentRenterPoints++;

        // show figures for this rental
        result += "\t" + each.getMovie().getTitle() + "\t" + String.valueOf(thisAmount) + "\n";
        totalAmount += thisAmount;
    }

    // add footer lines
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent renter points";

    return result;
}
```

frequent  
renter points

children's  
movies

frequent  
renter points

frequent  
renter points

# and then becomes...

```
public String printStatement() {
    double totalAmount = 0;
    Enumeration rentals = _rentals.elements();
    while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();

        //determine amount for each movie
        switch (each.getMovie().getPriceCode()) {
            case Movie.REGULAR:
                thisAmount += 2;
                if (each.getDaysRented() > 7)
                    thisAmount += (each.getDaysRented() - 7) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                thisAmount += each.getDaysRented() * 3;
                break;
        }

        totalAmount += thisAmount;
    }

    return "Amount owed: " + totalAmount;
}
```

```
public String printStatement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + "\n";

    while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();

        //determine amounts for each line
        switch (each.getMovie().getPriceCode()) {
            case Movie.REGULAR:
                thisAmount += 2;
                if (each.getDaysRented() > 2)
                    thisAmount += (each.getDaysRented() - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                thisAmount += each.getDaysRented() * 3;
                break;
            case Movie.CHILDRENS:
                thisAmount += 1.5;
                if (each.getDaysRented() > 3)
                    thisAmount += (each.getDaysRented() - 3) * 1.5;
                break;
        }

        // add frequent renter points
        frequentRenterPoints++;
        // add bonus for a two day new release
        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
            each.getDaysRented() > 1)
            frequentRenterPoints++;

        // show figures for this rental
        result += "\t" + each.getMovie().getTitle() + "\t" +
            String.valueOf(thisAmount) + "\t" +
            String.valueOf(frequentRenterPoints) + "\n";
        totalAmount += thisAmount;
    }

    // add footer lines
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent renter points";

    return result;
}
```

You might then add another method “printLongFormStatement” that reuses a lot of this code. And since you are in a hurry, you might just copy this method, and augment it.

*don't pretend you haven't done this*

# So the code changes...

---

- It's possible you (who are a perfect programmer) never introduces duplication into your code.
- But:
  - Others alter your code
  - You alter other people's code
  - This is good: collaboration = better product!

And code issues emerge

And are **expected** in agile methodologies!

# Code issues like:

---

- duplication
- rigidity
- lack of reusability
- mess

Gradually, code begins to rot in places.

Those places are said to “smell”

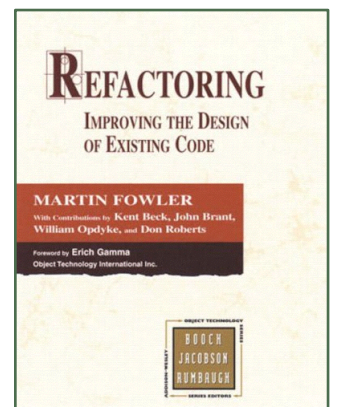
We, as designers/software developers, have to chase down these code smells and fix them.



# What is a “Code Smell”?

---

- A recognizable indicator that something may be wrong in the code
- Can occur in the product code as well as in the test code!

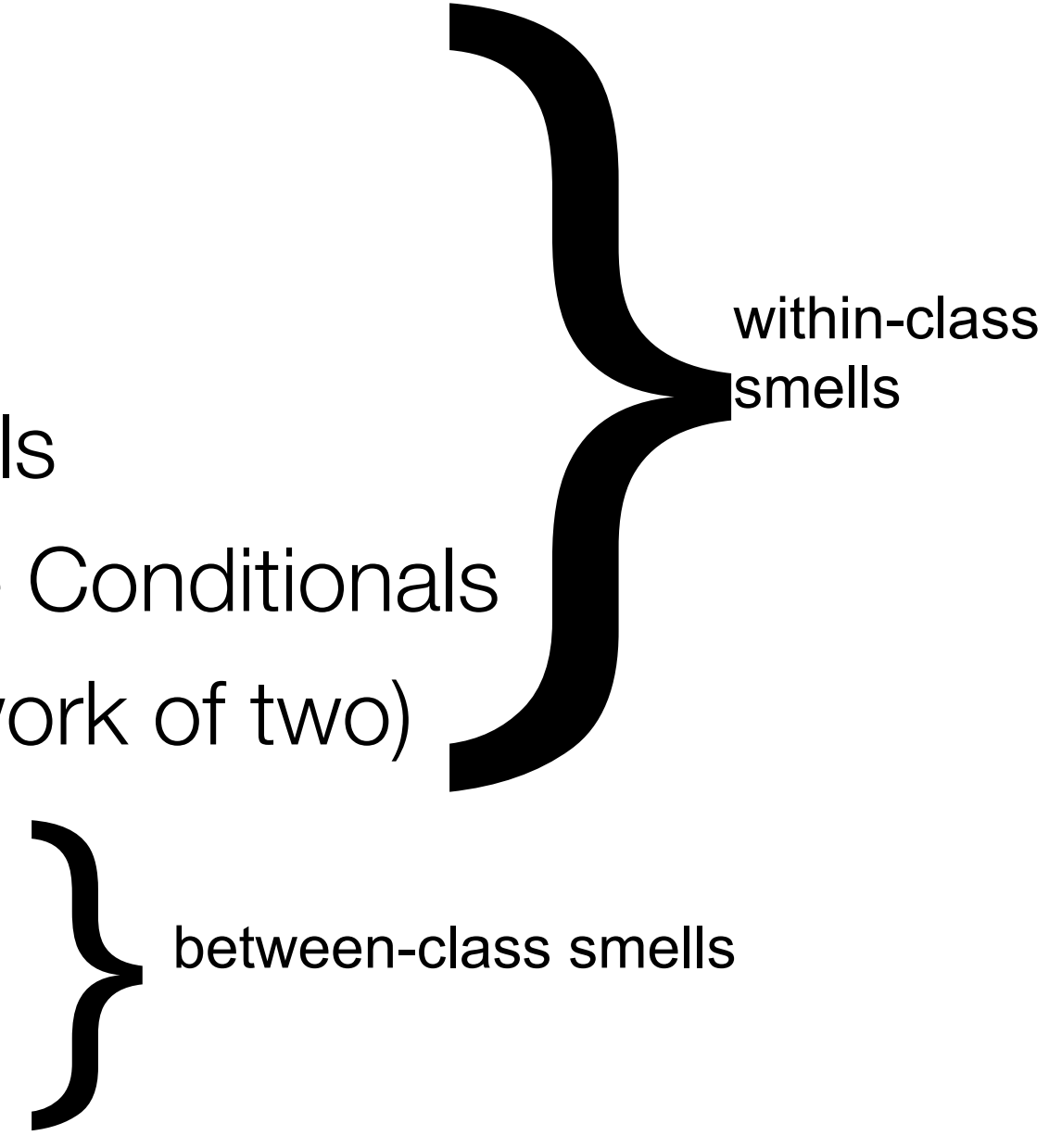


The smells/refactorings in the following slides are from Martin Fowler, Refactoring, “Improving the design of existing code”.

For test code smells: van Deursen et al. “Refactoring Test Code”.

# Some common issues

---

- Magic Numbers
  - Duplicated Code
  - Long Method
  - Complicated Conditionals
  - Switch Statements/Type Conditionals
  - Large class (doing the work of two)
  - Divergent Change
  - Shotgun Surgery
  - Comments
- 
- within-class smells
- between-class smells

# Let's look at a few...

---

# Magic Numbers

---

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

Any use of an actual  
number right in the code

# Duplicate Code

---

```
extern int array1[];
extern int array2[];

int sum1 = 0;
int sum2 = 0;
int average1 = 0;
int average2 = 0;

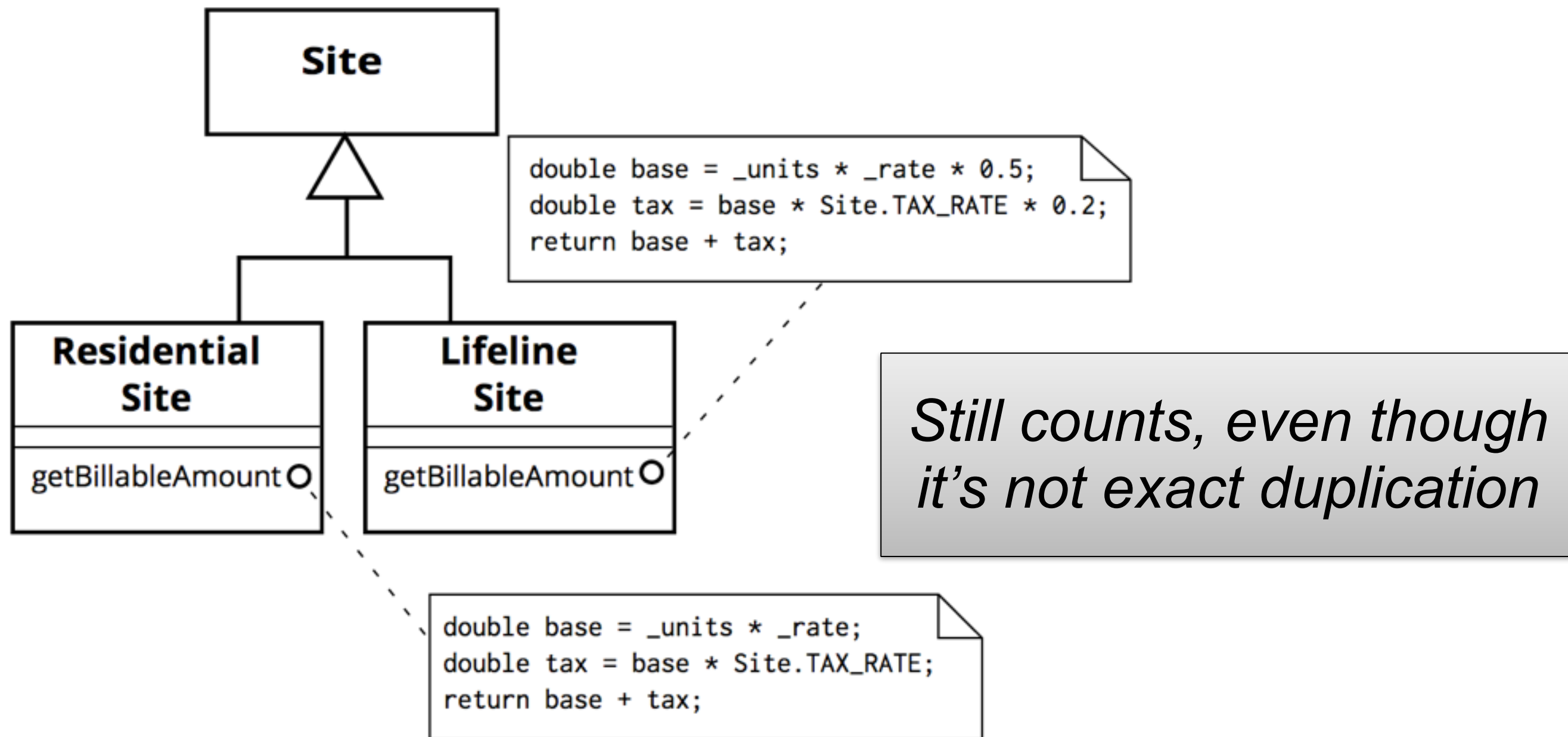
for (int i = 0; i < 4; i++)
{
    sum1 += array1[i];
}
average1 = sum1/4;

for (int i = 0; i < 4; i++)
{
    sum2 += array2[i];
}
average2 = sum2/4;
```

These two loops are the same!



# Almost-duplication



# Method too long...

some red flags...



**Deeply nested control structures:** e.g. for-loops 3 levels deep or even just 2 levels deep with nested if-statements that have complex conditions.



**Too many state-defining parameters:** By state-defining parameter, I mean a function parameter that guarantees a particular execution path through the function. Get too many of these type of parameters and you have a combinatorial explosion of execution paths (this usually happens in tandem with #1).



**Logic that is duplicated in other methods:** poor code re-use is a huge contributor to monolithic procedural code. A lot of such logic duplication can be very subtle, but once re-factored, the end result can be a far more elegant design.



**Excessive inter-class coupling:** this lack of proper encapsulation results in functions being concerned with intimate characteristics of other classes, hence lengthening them.



**Unnecessary overhead:** Comments that point out the obvious, deeply nested classes, superfluous getters and setters for private nested class variables, and unusually long function/variable names can all create syntactic noise within related functions that will ultimately increase their length.



**Your massive developer-grade display isn't big enough to display it:** Actually, displays of today are big enough that a function that is anywhere close to its height is probably way too long. But, if it is larger, this is a smoking gun that something is wrong.



**You can't immediately determine the function's purpose:** Furthermore, once you actually do determine its purpose, if you can't summarize this purpose in a single sentence or happen to have a tremendous headache, this should be a clue.

# Complicated Conditionals

---

```
if (date.before (SUMMER_START) || date.after(SUMMER_END))  
    charge = quantity * _winterRate + _winterServiceCharge;  
else charge = quantity * _summerRate;
```

# Switch Statements

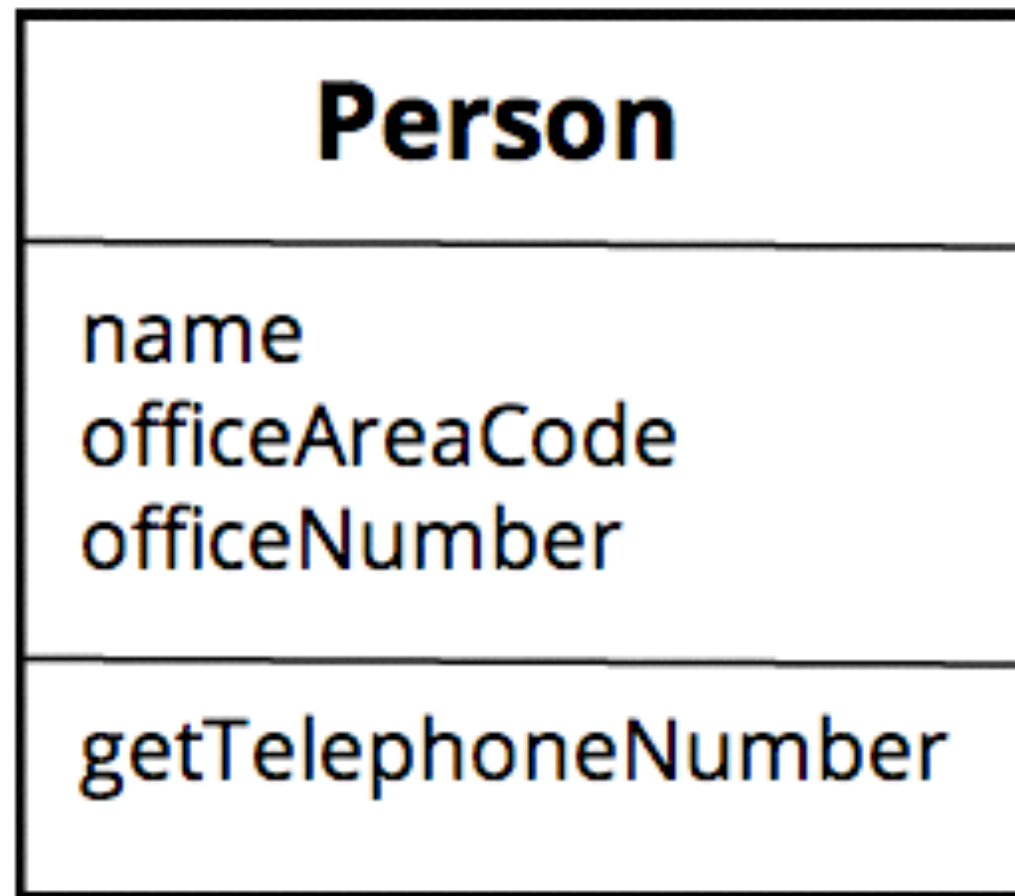
---

a conditional that chooses different behaviour depending on the type of an object (or a weird string representation of that type)

```
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");  
}
```

# One class is actually two

---





# Data Clump

---

*always passed around together*

```
public static void copyRange(int start, int end)
{
    //do something
}
```

*Sometimes combined with “Long Parameter List” where bunches of data clumps are passed into a method with just too many parameters*

“A good test is to consider deleting one of the data values: if you did this, would the others make any sense? If not, you have a data clump!

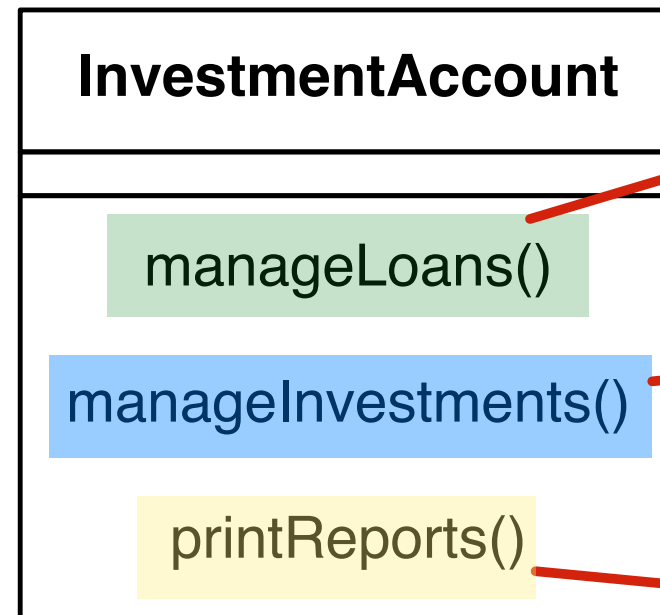
# Divergent Changes

*When you have to alter a class for more than one kind of change*

Divergent change occurs when one class is commonly changed in different ways for different reasons. ...

Any change to handle a variation should change a single class, and all the typing in the new class should express the variation.

## 3 reasons for change!



will need to change whenever the loans implementation is changed

will need to change whenever investment implementation is changed

will need to change every time the printing implementation is changed

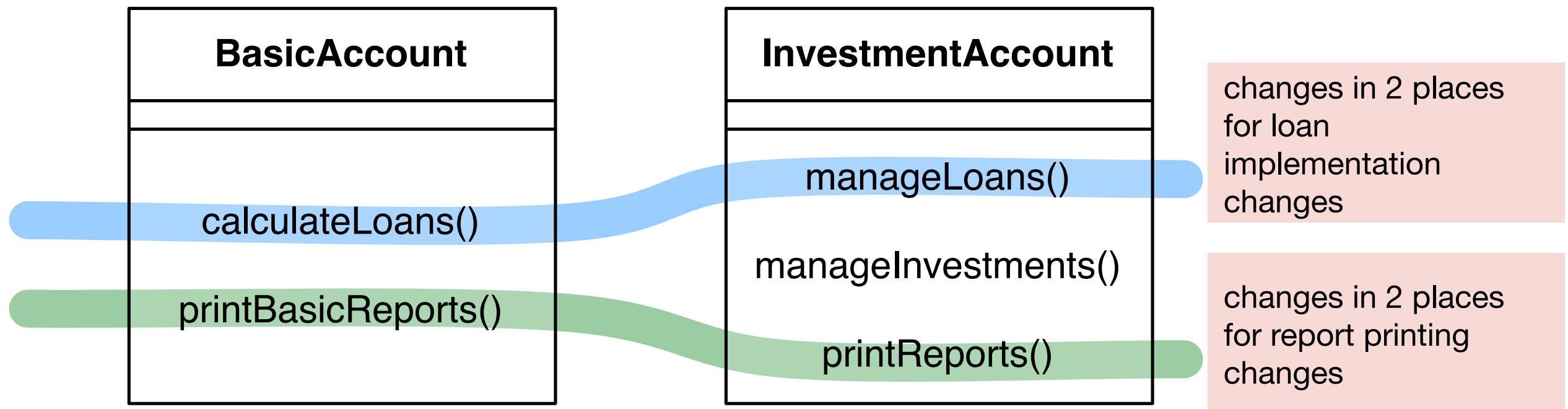
*If you look at a class and say, "Well, I will have to change these three methods every time I get a new database; I have to change these four methods every time there is a new financial instrument," you likely have a situation in which two objects are better than one. That way each object is changed only as a result of one kind of change. Of course, you often discover this only after you've added a few databases or financial instruments.*

*A change that alters many classes*

# Shotgun Surgeries

*You whiff this when every time you make a kind of change, you have to make a lot of little changes to a lot of different classes. When the changes are all over the place, they are hard to find, and it's easy to miss an important change.*

*this is the inverse of divergent change. One change in lots of places, versus one place with lots of changes*



*this isn't the same as the REQUIRES/MODIFIES/EFFECTS comments we used in 210!*

# Needing comments to explain the code

... comments often are used as a deodorant. It's surprising how often you look at thickly commented code and notice that the comments are there because the code is bad.

A good time to use a comment is when you don't know what to do. In addition to describing what is going on, comments can indicate areas in which you aren't sure. A comment is a good place to say why you did something. This kind of information helps future modifiers, especially forgetful ones.

```
# convert to cents
a = x * 100

# avg cents per customer
avg = a / n

# add to list
avgs < avg
t += 1
```

```
double getExpenseLimit() {
    // should have either expense limit or a primary project
    return (_expenseLimit != NULL_EXPENSE) ?
        _expenseLimit:
        _primaryProject.getMemberExpenseLimit();
}
```

# Okay...

---

- So now we know some symptoms of bad code.
- What do you do in response?

***REFACTOR!***

- Long-term investment in the quality of the code and its structure
- No refactoring may save costs / time in the short term but incurs a huge penalty in the long run



# What is “Refactoring”

---

*“[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” – Martin Fowler*

- Changes made to a system that:
  - **Do not** change observable behavior
    - (MEANING PRESERVING)
  - Remove duplication or needless complexity
  - Enhance software quality
  - Make the code easier and simpler to understand
  - Make the code more flexible
  - Make the code easier to change
- Requires Tests!

# Refactoring

---

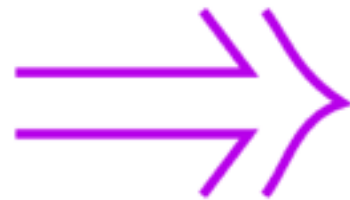
- At its simplest, it's just a small, behaviour-preserving, source-to-source transformation.
- Example:

```
extern int array1[];
extern int array2[];

int sum1 = 0;
int sum2 = 0;
int average1 = 0;
int average2 = 0;

for (int i = 0; i < 4; i++)
{
    sum1 += array1[i];
}
average1 = sum1/4;

for (int i = 0; i < 4; i++)
{
    sum2 += array2[i];
}
average2 = sum2/4;
```



```
int calcAverage (int* Array_of_4)
{
    int sum = 0;
    for (int i = 0; i < 4; i++)
    {
        sum += Array_of_4[i];
    }
    return sum/4;
}
```

# Definition of broken code:

---

Every module has three functions:

- To execute according to its purpose
- To afford change
- To communicate to its readers
- If it does not do one or more of these, it is broken.

```
q = ((p<=1) ? (p ? 0 : 1) : (p==-4) ? 2 : (p
+1));

while (*a++ = *b--) ;

char b[2][10000], *s, *t=b, *d, *e=b+1, **p; main(int
c, char**v) {int n=atoi(v[1]); strcpy(b, v[2]);
while(n--) {for(s=t, d=e; *s; s++) {for(p=v
+3; *p; p++) if(**p==*s) {strcpy(d, *p+2); d+=
strlen(d); goto x; } *d++=*s; x:} s=t; t=e; e=s; *d
++=0; } puts(t); }
```

# When to refactor?

---

- NOT: 2 weeks every 6 months
- Do it as you develop - Opportunistic Refactoring
- Boy Scout principle: leave it better than you found it.
- If you recognize a warning sign (a bad smell)
  - When you add a function
    - Before, to start clean and/or
    - After, to clean-up
  - When you fix a bug
  - When you code review
  - You can use The Rule of Three
    1. The first time, just do it!
    2. Need it somewhere else? Cut and paste it!
    3. The third time, refactor!

# When **not to** refactor?

---

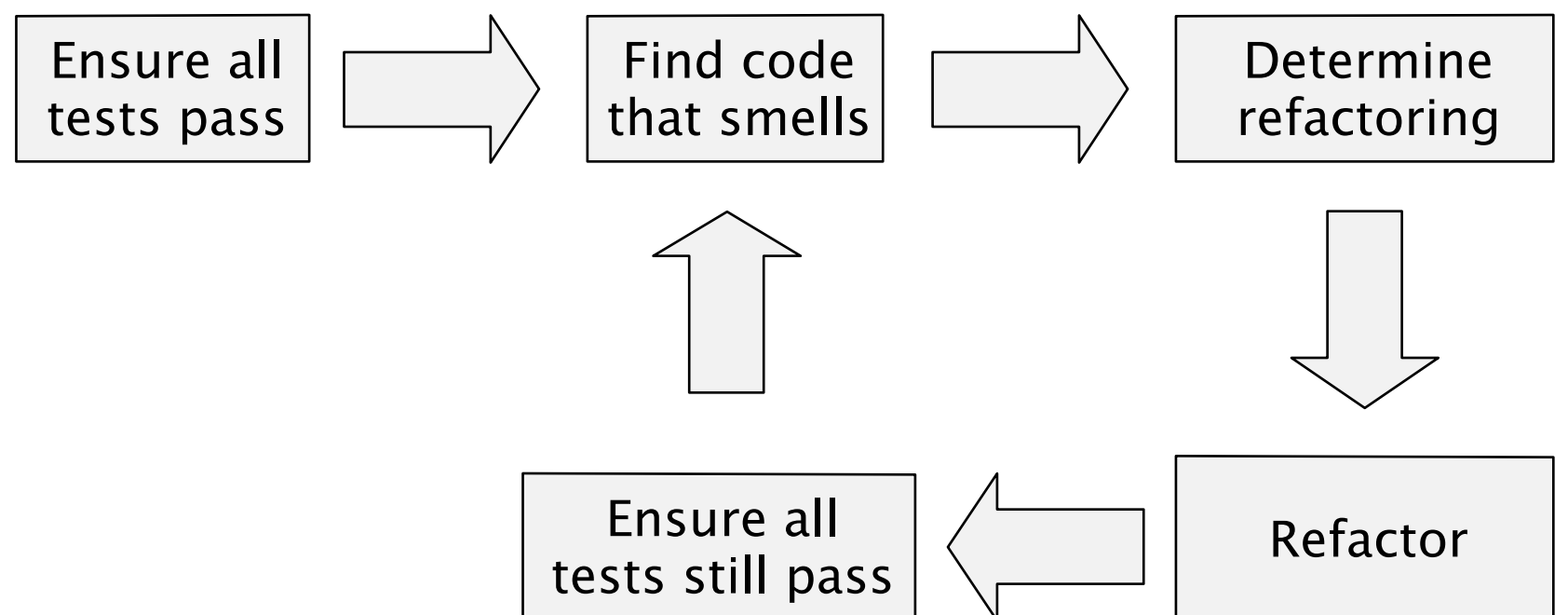
- When the tests are failing
- When you should just rewrite the code
- When you have impending deadlines



# How to refactor?

---

1. Make sure all your tests pass
2. Identify the code smell
3. Determine how to refactor this code
4. Apply the refactoring
5. Run tests to make sure you didn't break anything
6. Repeat until the smell is gone



# Refactorings fix Code Smells

---

- Add Parameter
- Change Bidirectional Association to Unidirectional
- Change Reference to Value
- Change Unidirectional Association to Bidirectional
- Change Value to Reference
- Collapse Hierarchy
- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Convert Procedural Design to Objects
- Decompose Conditional
- Duplicate Observed Data
- Encapsulate Collection
- Encapsulate Downcast
- Encapsulate Field
- Extract Class
- Extract Hierarchy
- Extract Interface
- Extract Method
- Extract Subclass
- Extract Superclass
- Form Template Method
- Hide Delegate
- Hide Method
- Inline Class
- Inline Method
- Rename Constant

Each of these is one predictably  
meaning preserving code  
transformation.

Online: <http://www.refactoring.com/catalog>

smell: magic numbers

refactoring: replace it with a symbolic constant

---

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



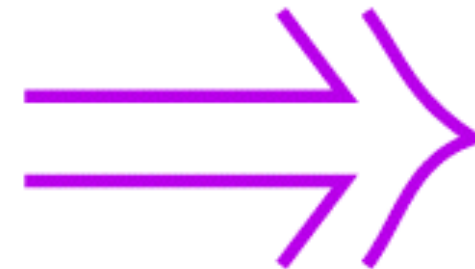
```
double potentialEnergy(double mass, double height) {  
    return mass * GRAVITATIONAL_CONSTANT * height;  
}  
  
static final double GRAVITATIONAL_CONSTANT = 9.81;
```

# Smell: Repeated Lines of Code

## Refactoring: Extract Method

*Not all duplicate code gets fixed by extracting a method. You need to carefully assess what kind of duplication you have.*

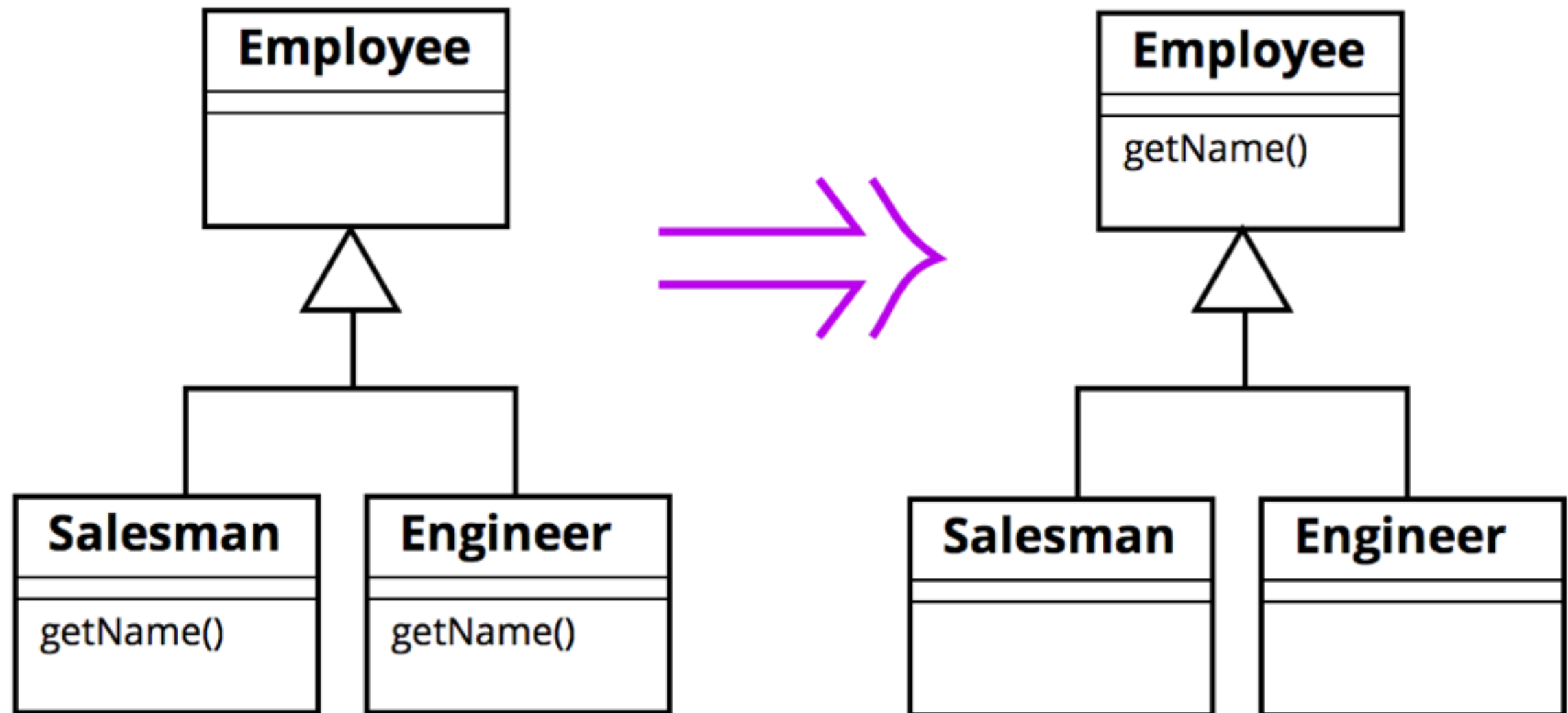
```
void printOwing() {  
    printBanner();  
  
    //print details  
    System.out.println ("name:  " + _name);  
    System.out.println ("amount " + getOutstanding());  
}
```



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

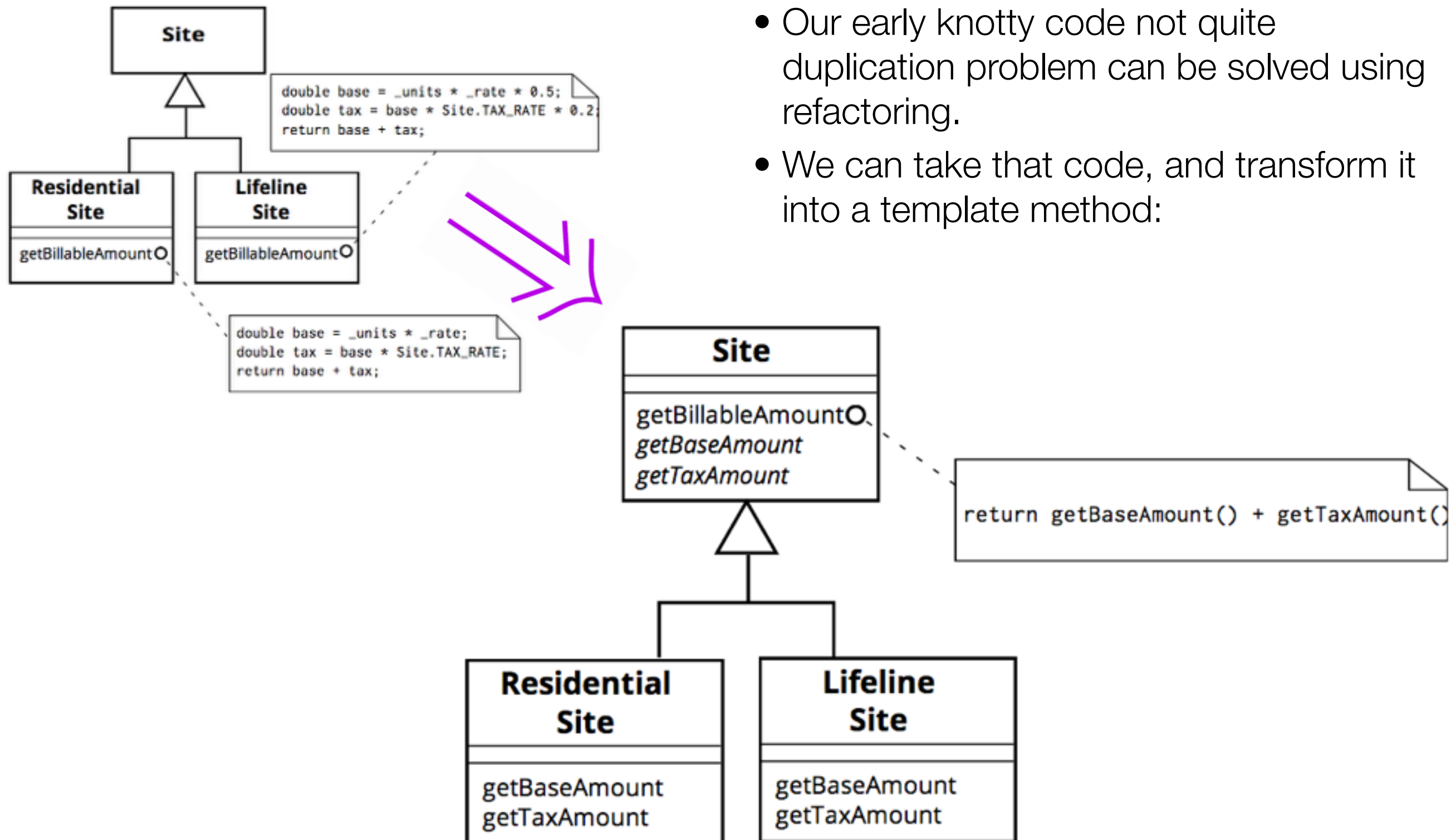
Smell: same method in two classes  
Refactoring: Pull up method

---



# Smell: almost duplicated code

## Refactoring: move to template method



# Smell: Long method

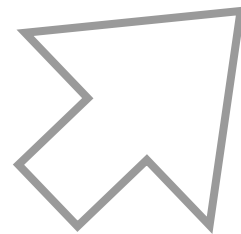
## Refactoring(s): Lots of options!

---

### ***Extract Method:***

- *Pull code out into a separate method when the original method is long or complex*
- *Name the new method so as to make the original method clearer*
- *Each method should have just one task*

▶ Extract Method



▶ Replace Temp with Query

▶ Replace Method with Method Object

▶ Decompose Conditional

▶ Consolidate Conditional Expression



# smell: Complicated conditional

## Refactoring: Decompose conditional

---

```
if (date.before (SUMMER_START) || date.after(SUMMER_END))  
    charge = quantity * _winterRate + _winterServiceCharge;  
else charge = quantity * _summerRate;
```



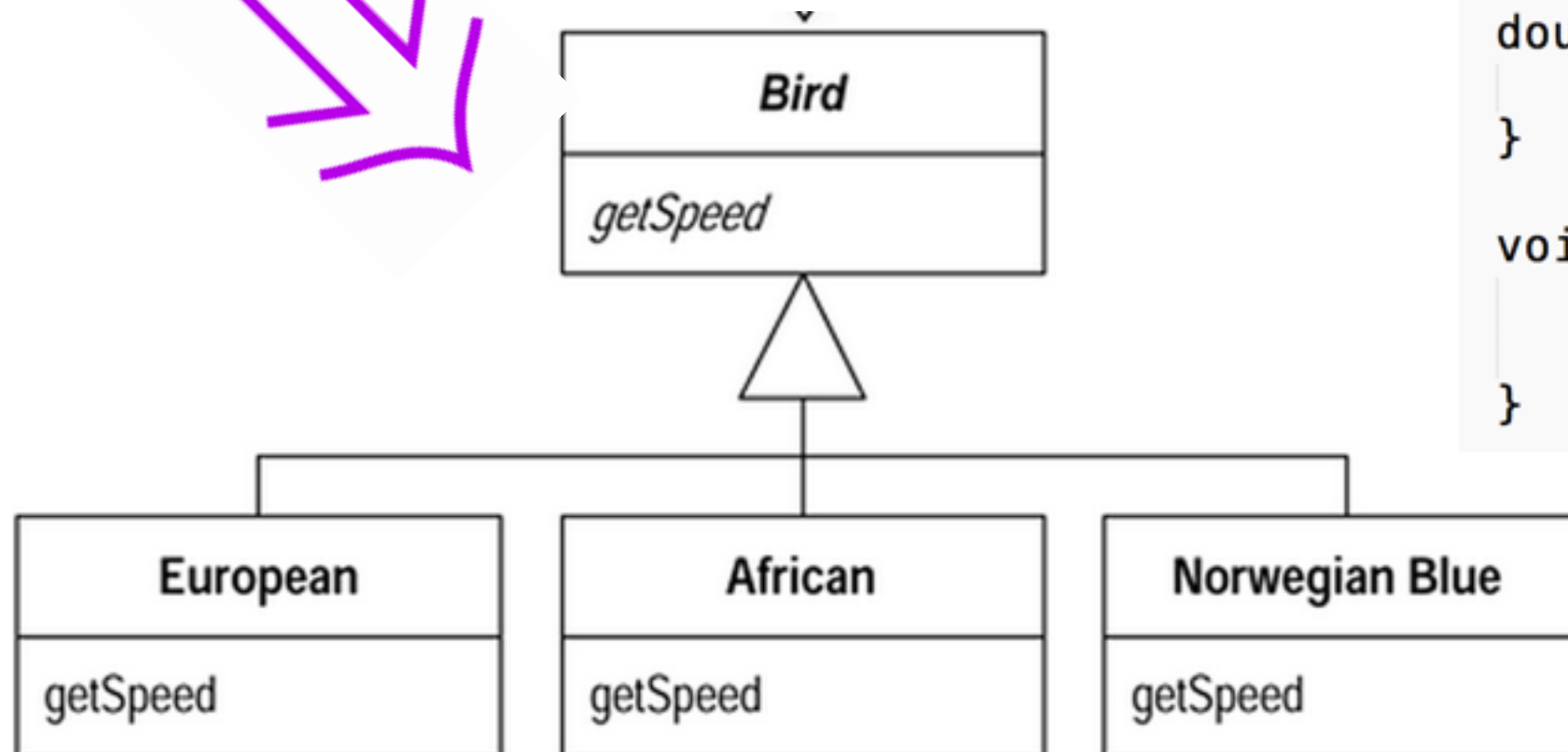
*extract methods from the condition, the “then” and the “else” parts.*

```
if (notSummer(date))  
    charge = winterCharge(quantity);  
else charge = summerCharge (quantity);
```

smell: switch statement/typed conditional  
refactor: replace conditional with polymorphism

---

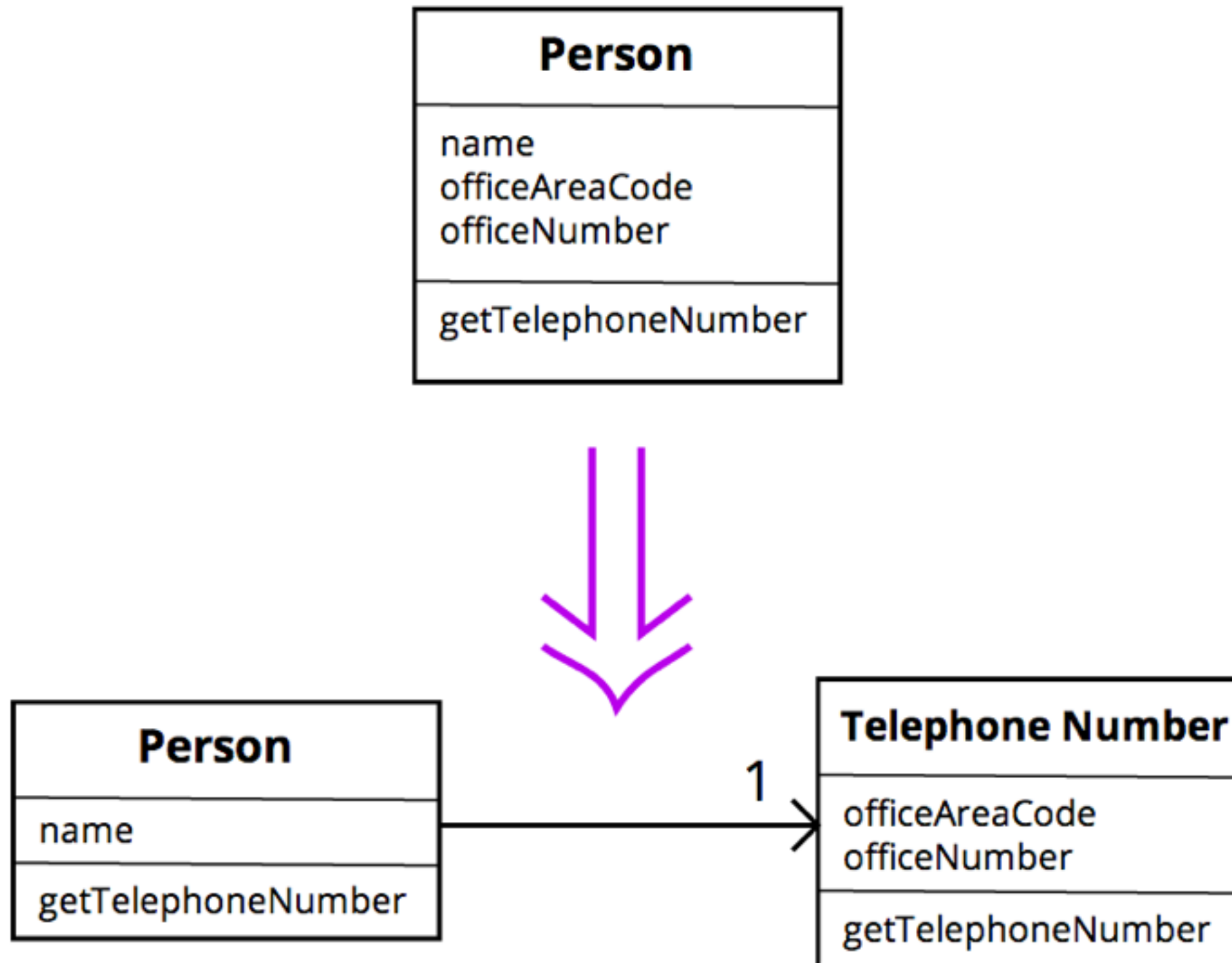
```
double getSpeed() {  
    switch (_type) {  
        case EUROPEAN:  
            return getBaseSpeed();  
        case AFRICAN:  
            return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;  
        case NORWEGIAN_BLUE:  
            return (_isNailed) ? 0 : getBaseSpeed(_voltage);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



```
double getSpeed(Bird bird) {  
    bird.getSpeed();  
}  
  
void main () {  
    Bird b = new African();  
    getSpeed(b);  
}
```

Smell: one class doing the work of two  
Refactoring: extract class

---



Smell: data clump (parameters that always go together)  
Refactoring: introduce parameter object

*Introduce parameter object - If you have a group of parameters that naturally go together then you can replace them with an object.*

always passed around together

```
public static void copyRange(int start, int end)
{
    //do something
}
```

```
public static void copyRange(Range r){
    //do something
}
```

Range
int getStart() int getEnd()

Customer
amountInvoicedIn (start : Date, end : Date) amountReceivedIn (start : Date, end : Date) amountOverdueIn (start : Date, end : Date)



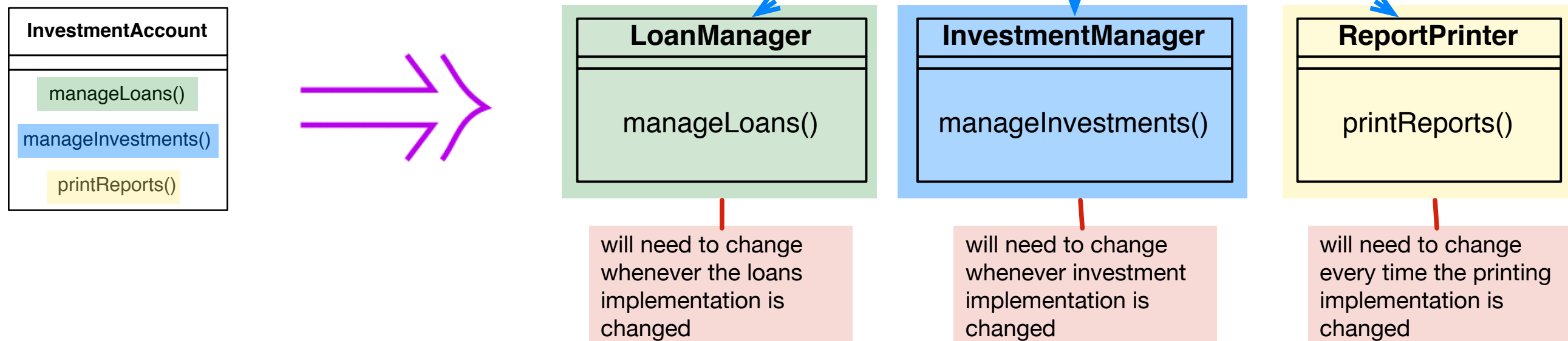
Customer
amountInvoicedIn (: DateRange) amountReceivedIn (: DateRange) amountOverdueIn (: DateRange)

# smell: divergent changes

## refactoring: extract class

*ideally, there is a one-to-one link  
between common changes and classes.*

*identify everything that changes for  
a particular cause and use Extract  
Class to put them all together*

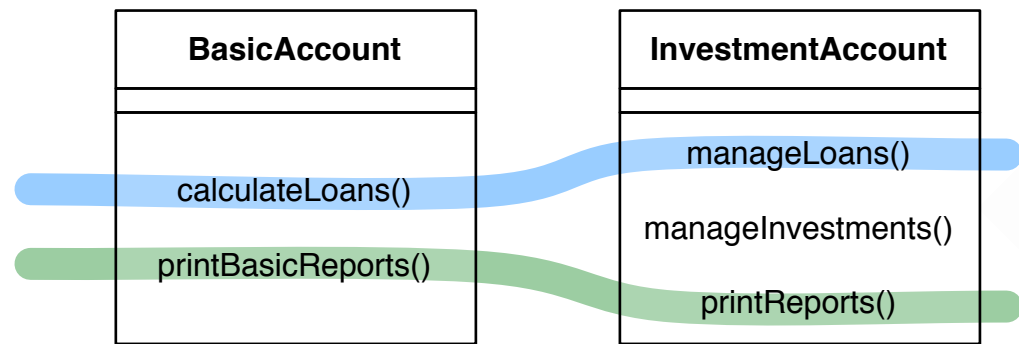


<https://www.youtube.com/watch?v=f2-6W8wxlj0>

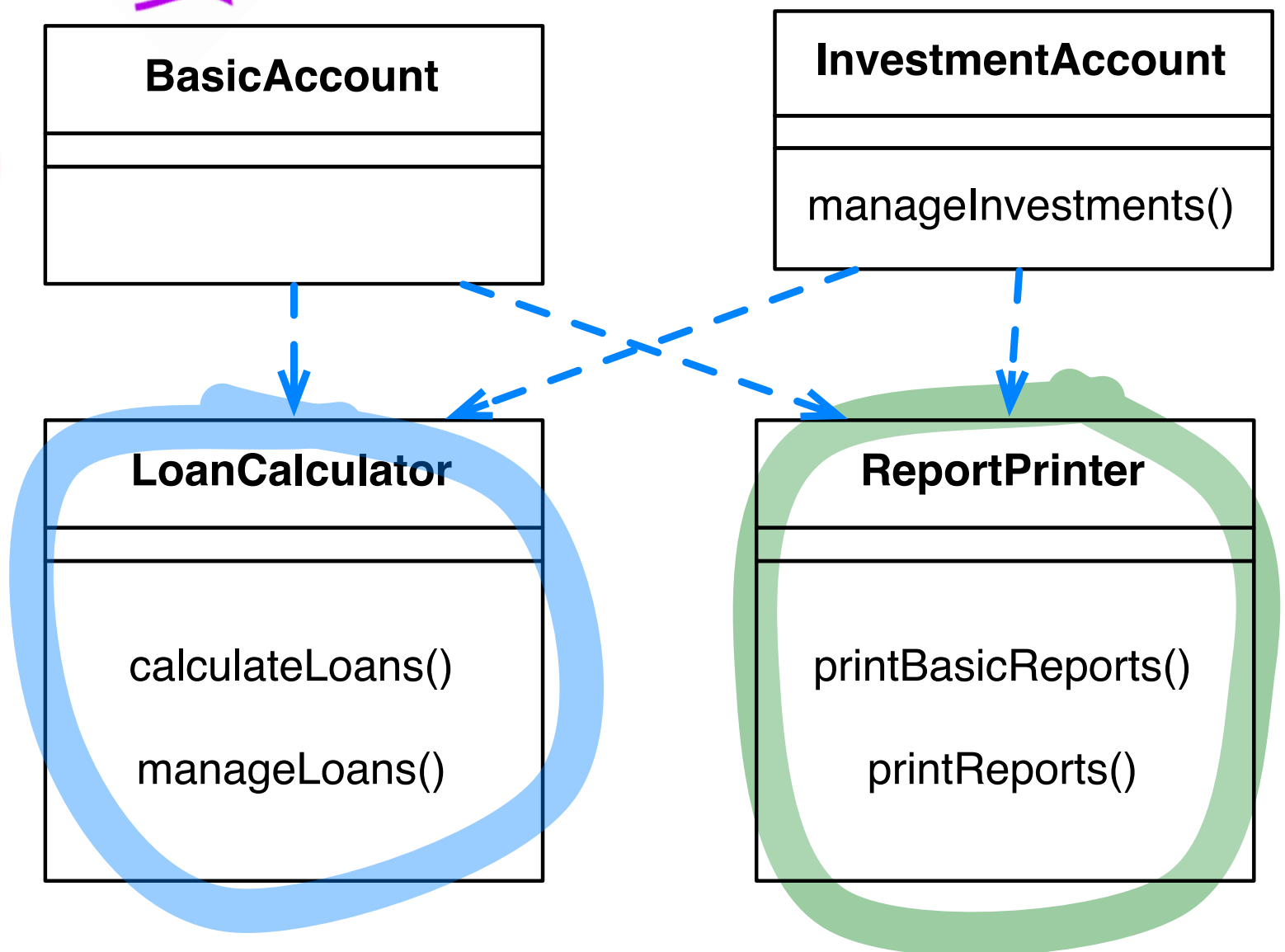
smell: shotgun surgery

refactoring: move method; move field

*ideally, there is a one-to-one link  
between common changes and classes.*



In this case you want to  
use Move Method and  
Move Field to put all the  
changes into a single  
class. If no current class  
looks like a good  
candidate, create one.

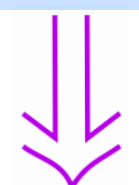


# smell: needing comments to explain the code

## refactoring: extract functionality/introduce assertion


*When you feel the need to write a comment, first try to refactor the code so that any comment becomes superfluous* ”

```
double getExpenseLimit() {  
    // should have either expense limit or a primary project  
    return (_expenseLimit != NULL_EXPENSE) ?  
        _expenseLimit:  
        _primaryProject.getMemberExpenseLimit();  
}
```



```
double getExpenseLimit() {  
    Assert.isTrue (_expenseLimit != NULL_EXPENSE || _primaryProject != null);  
    return (_expenseLimit != NULL_EXPENSE) ?  
        _expenseLimit:  
        _primaryProject.getMemberExpenseLimit();  
}
```

```
# convert to cents  
a = x * 100  
  
# avg cents per customer  
avg = a / n  
  
# add to list  
avgs < avg  
t += 1
```



```
total_cents = total * 100  
average_per_customer = total_cents / customer_count  
  
track_average(average_per_customer)
```

Now ... with all that under our belts...

---



# What smells?

---

```
class Account {
    float principal, rate;
    int daysActive, accountType;

    public static final int STANDARD = 0;
    public static final int BUDGET = 1;
    public static final int PREMIUM = 2;
    public static final int PREMIUM_PLUS = 3;
}

class Customer {
    public float calculateFee(Account accounts[]) {
        float totalFee = 0;
        Account account;
        for (int i=0; i<accounts.length; i++)
            if ( account.accountType == Account.PREMIUM ||
                account.accountType == Account.PREMIUM_PLUS ) {
                totalFee += .0125 * ( account.principal
                                     * Math.exp( account.rate * (account.daysActive/365.25) )
                                     - account.principal );
            }
        return totalFee;
    }
}
```

# What smells?

---

```
class Account {  
    float principal, rate;  
    int daysActive, accountType;  
  
    public static final int STANDARD = 0;  
    public static final int BUDGET = 1;  
    public static final int PREMIUM = 2;  
    public static final int PREMIUM_PLUS = 3;  
}
```

```
class Customer {  
    public float calculateFee(Account accounts[]) {  
        float totalFee = 0;  
        Account account;  
        for (int i=0; i<accounts.length; i++)  
            if ( account.accountType == Account.PREMIUM ||  
                account.accountType == Account.PREMIUM_PLUS ) {  
                totalFee += .0125 * ( account.principal  
                                     * Math.exp( account.rate * (account.daysActive/365.25) )  
                                     - account.principal );  
            }  
        return totalFee;  
    }  
}
```

Complicated  
Conditional

Magic  
Number

unreadable  
functionality

# Which refactorings would work?

```
class Account {  
    float principal, rate;  
    int daysActive, accountType;  
  
    public static final int STANDARD = 0;  
    public static final int BUDGET = 1;  
    public static final int PREMIUM = 2;  
    public static final int PREMIUM_PLUS = 3;  
}
```

```
class Customer {  
    public float calculateFee(Account accounts[]) {  
        float totalFee = 0;  
        Account account;  
        for (int i=0; i<accounts.length; i++)  
            if ( account.accountType == Account.PREMIUM ||  
                account.accountType == Account.PREMIUM_PLUS ) {  
                totalFee += .0125 * ( account.principal  
                                     * Math.exp( account.rate * (account.daysActive/365.25) )  
                                     - account.principal );  
            }  
        return totalFee;  
    }  
}
```

**Decompose  
Conditional**

**Complicated  
Conditional**

**Replace with  
Symbolic  
Constant**

**Magic  
Number**

**unreadable  
functionality**

**Extract  
Functionality**

# The end result.

```
class Account {
    float principal, rate;
    int daysActive, accountType;

    public static final int STANDARD = 0;
    public static final int BUDGET = 1;
    public static final int PREMIUM = 2;
    public static final int PREMIUM_PLUS = 3;
}
```

Decompose  
Conditional

```
class Customer {
    public float calculateFee(Account accounts[]) {
        float totalFee = 0;
        Account account;
        for (int i=0; i<accounts.length; i++)
            if ( account.accountType == Account.PREMIUM ||
                account.accountType == Account.PREMIUM_PLUS ) {
                totalFee += .0125 * ( account.principal
                                     * Math.exp( account.rate * (account.daysActive/365.25) )
                                     - account.principal );
            }
        return totalFee;
    }
}
```

Replace with  
Symbolic  
Constant

Magic  
Number

Complicated  
Conditional

unreadable  
functionality

Extract  
Functionality

```
private float interestEarned() {
    float years = daysActive / (float) 365.25;
    float compoundInterest = principal * (float) Math.exp( rate * years );
    return ( compoundInterest - principal );
}
```

Extract  
Functionality

```
private float isPremium() {
    if (accountType == Account.PREMIUM || accountType == Account.PREMIUM_PLUS)
        return true;
    else return false;
}
```

Decompose  
Conditional

```
public float calculateFee(Account accounts[]) {
    float totalFee = 0;
    Account account;
    for (int i=0; i<accounts.length; i++) {
        account = accounts[i];
        if ( account isPremium() )
            totalFee += BROKER_FEE_PERCENT * account.interestEarned();
    }
    return totalFee;
}
```

Replace with  
Symbolic  
Constant

```
static final double BROKER_FEE_PERCENT = 0.0125;
```

# Resources

---

- “The” Book, by Martin Fowler
  - Refactoring: Improving the design of existing code
- Smells to refactorings
  - <http://wiki.java.net/bin/view/People/SmellsToRefactorings>
- List of refactorings
  - <http://www.refactoring.com/catalog>
- A refactoring “cheat sheet”
  - <http://industriallogic.com/papers/smellstorefactorings.pdf>
- Use IDE support! Manual refactoring is hard and potentially error prone. (Eclipse/IntelliJ both provide automatic refactoring support)

# Remember:

---

- A potential for refactoring is not a smell
  - Just because you see a potential for refactoring doesn't mean you should apply it. Only refactor if the code suffers from a code smell.
  - Some refactorings are opposites of one another (you could get caught in a loop of refactorings if you do them just for the sake of it! Inline versus Extract method, for instance.)
- First smell, then refactor

# Summary

---

- Code decays for many reasons
  - Collaboration, rework, external conditions, agility
- Refactoring improves existing code
  - Does not change existing behaviour
- Refactoring improves maintainability and hence productivity
- Refactor continuously
- Refactoring is an iterative process
  - Tests pass → Find smell → Refactor → Repeat
- Many smells, even more refactorings!