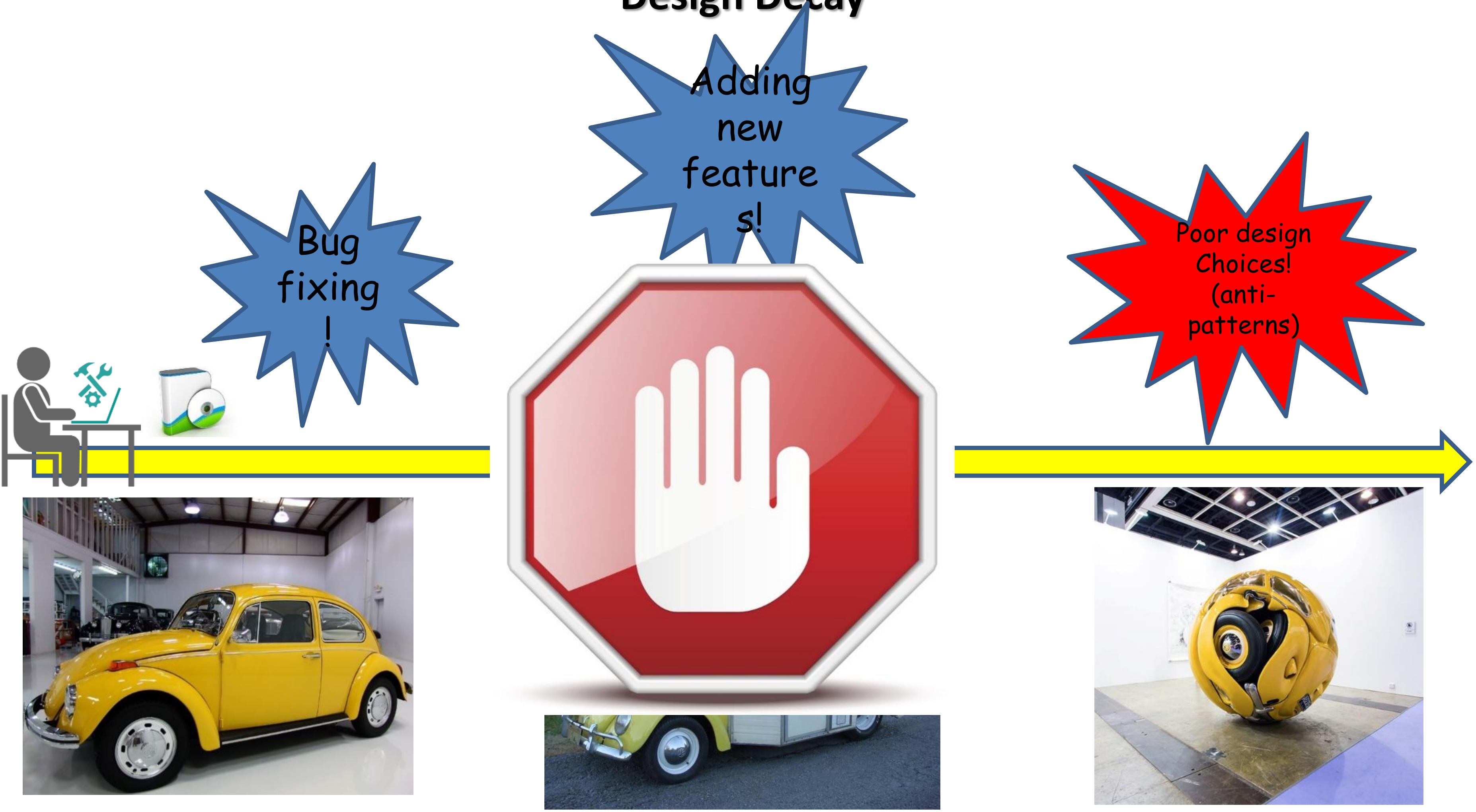
LOG6306 : Patrons pour la compréhension de programme

Foutse Khomh
foutse.khomh@polymtl.ca
Local M-4123

ÉCOLE POLYTECHNIQUE MONTRÉAL

Département de génie informatique et de génie logiciel
École Polytechnique de Montréal

Design Decay



Bug fixing!

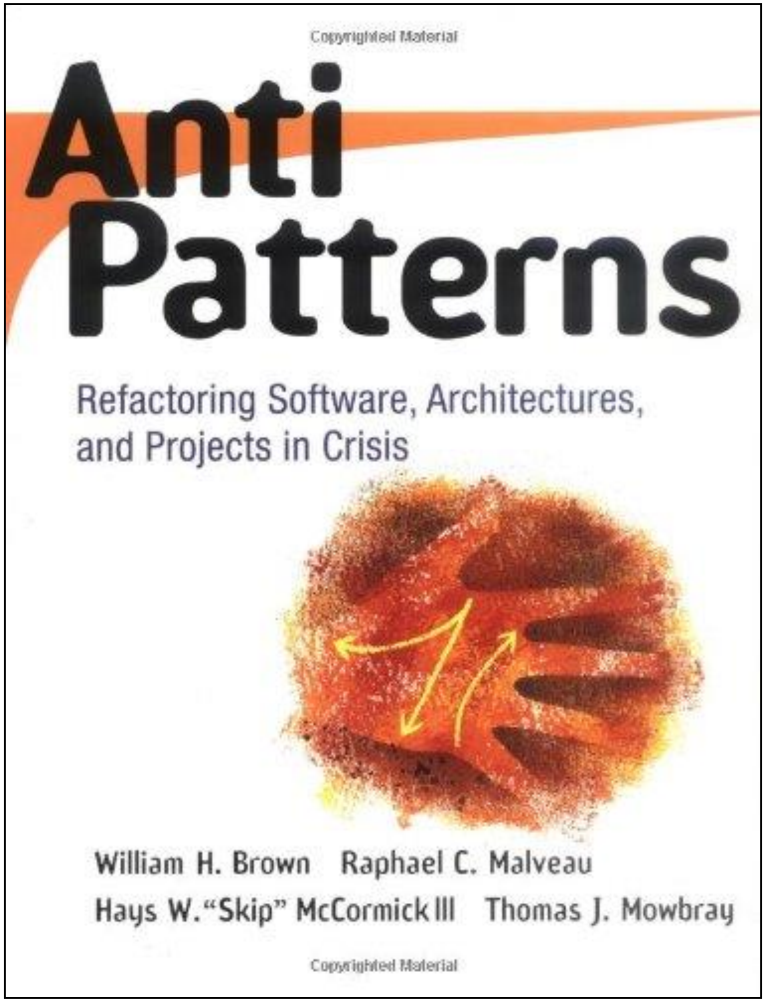
Adding new features!

Poor design Choices! (anti-patterns)

Design Decay

- Development team may implement software features with poor design, or bad coding…

- Code Smells (Low level (local) problems)
  - Poor coding decisions
- Lexical smells (Linguistic Anti-patterns)
  - Poor naming, commenting… of an entity
- Anti-patterns (High Level (global) problems)
  - Poor design solutions to recurring design problems



Copyrighted Material

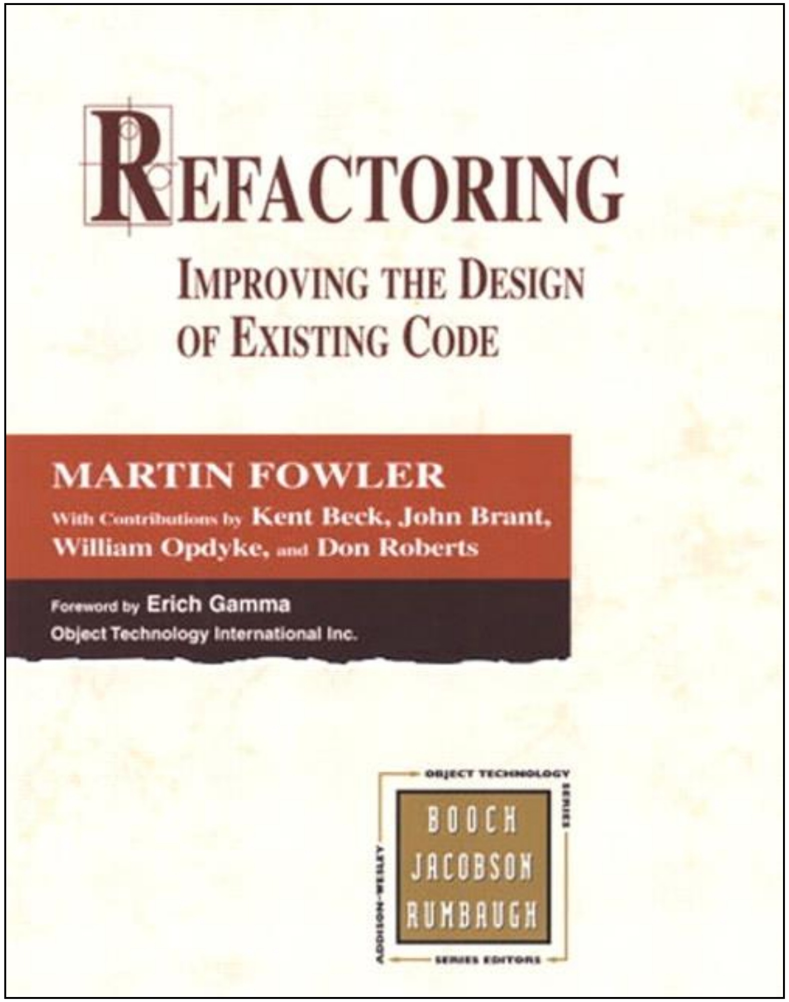
Anti Patterns

Refactoring Software, Architectures, and Projects in Crisis

William H. Brown Raphael C. Malveau

Hays W. "Skip" McCormick III Thomas J. Mowbray

Copyrighted Material



REFACTORING

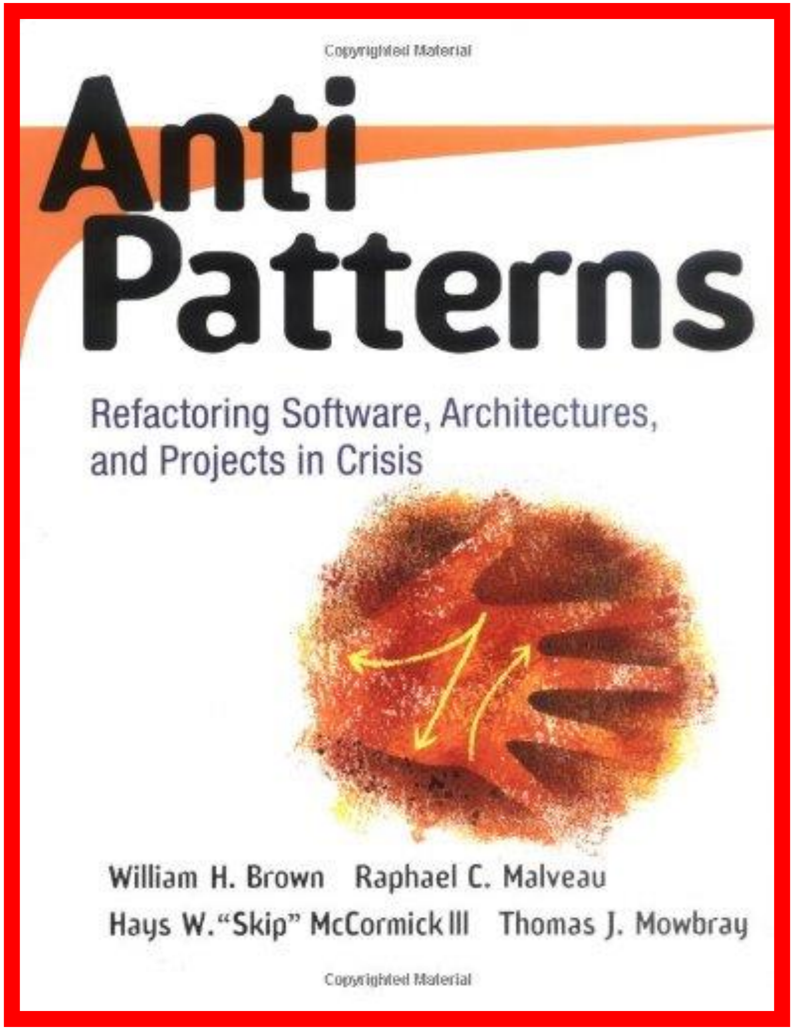
IMPROVING THE DESIGN OF EXISTING CODE

MARTIN FOWLER

With Contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts

Foreword by Erich Gamma

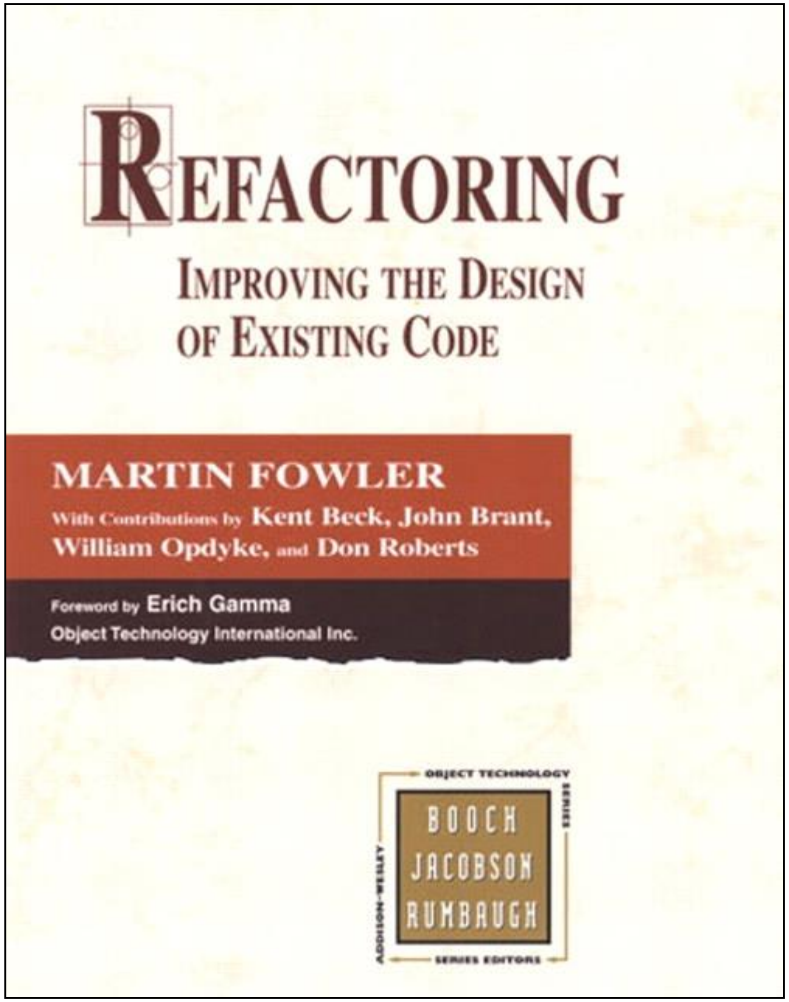
Object Technology International Inc.



Anti Patterns

Refactoring Software, Architectures, and Projects in Crisis

William H. Brown Raphael C. Malveau
Hays W. "Skip" McCormick III Thomas J. Mowbray



REFACTORING

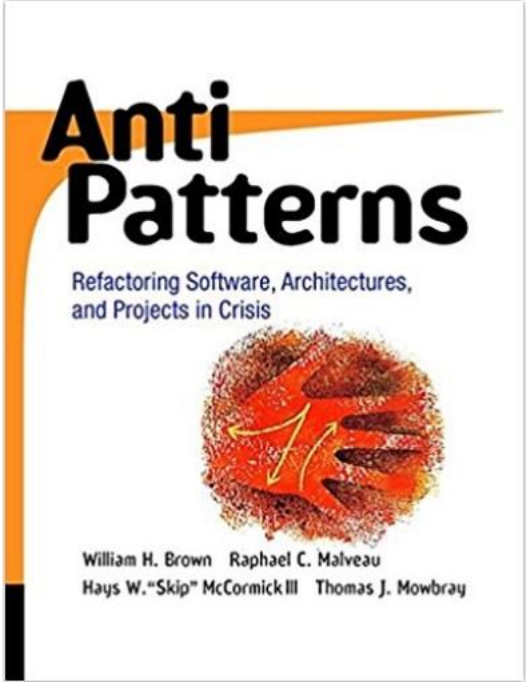
IMPROVING THE DESIGN OF EXISTING CODE

MARTIN FOWLER
With Contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts

Foreword by Erich Gamma
Object Technology International Inc.

OBJECT TECHNOLOGY SERIES
BOOCH
JACOBSON
RUMBAUGH
SERIES EDITORS
ADDISON-WESLEY

Anti-patterns


Anti Patterns
Refactoring Software, Architectures, and Projects in Crisis
William H. Brown Raphael C. Malveau
Hays W. "Skip" McCormick III Thomas J. Mowbray

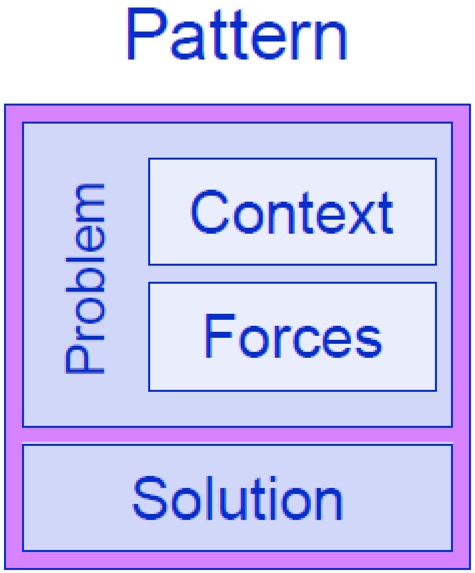
Anti-patterns are "poor" solutions to recurring design and implementation problems

- Impact program comprehension, software evolution and maintenance activities
- Important to detect them early in software development process, to reduce the maintenance costs

—William H. Brown, 1998

Anti-patterns

- **Design patterns** are “good” solutions to recurring design issues, but on the other side,..
- **Anti-patterns** are “bad” design practices that lead to negative consequences.



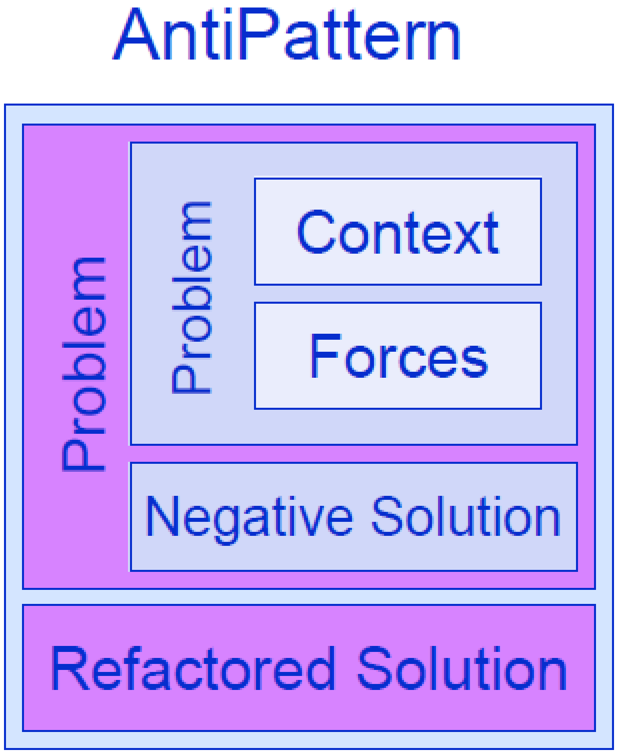
Pattern

Problem

Context

Forces

Solution



AntiPattern

Problem

Problem

Context

Forces

Negative Solution

Refactored Solution

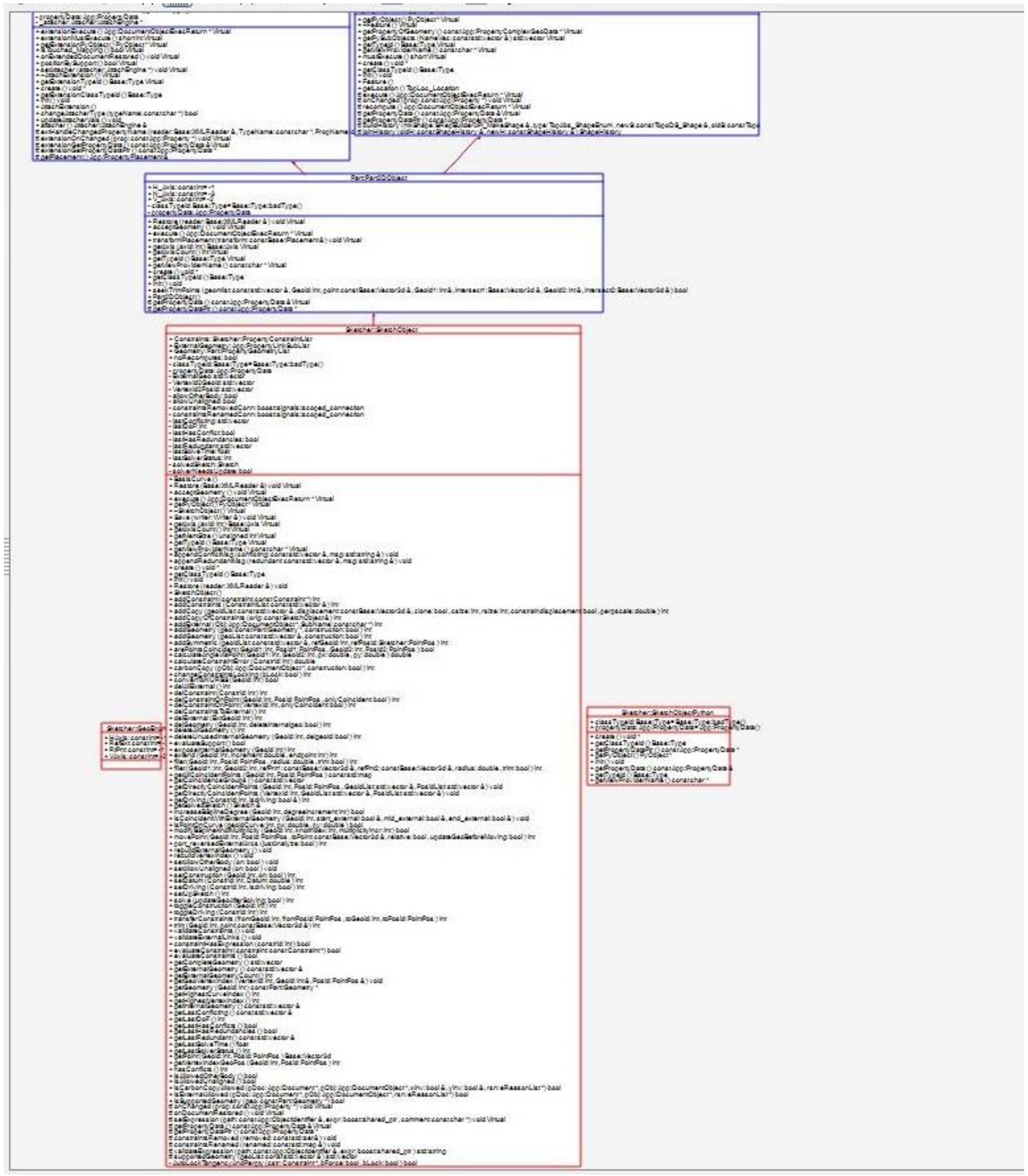
Blob (God Class)

- Blob (God Class)
  - “Procedural-style design leads to one object with a lion’s share of the responsibilities while most other objects only hold data or execute simple processes”
  - Conception procédurale en programmation OO
  - Large classe contrôleur
  - Beaucoup d’attributs et méthodes avec une faible cohésion*
  - Dépend de classes de données

* À quel point les méthodes sont étroitement liées aux attributs et aux méthodes de la classe.

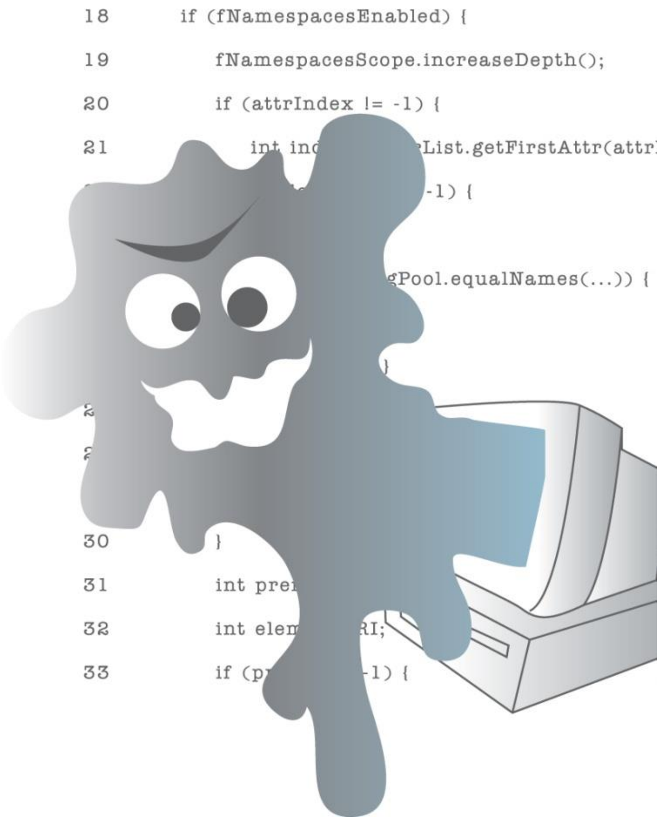
Blob (God Class)

- FreeCAD project
- 2,540,559 lines of code



Blob (God Class)

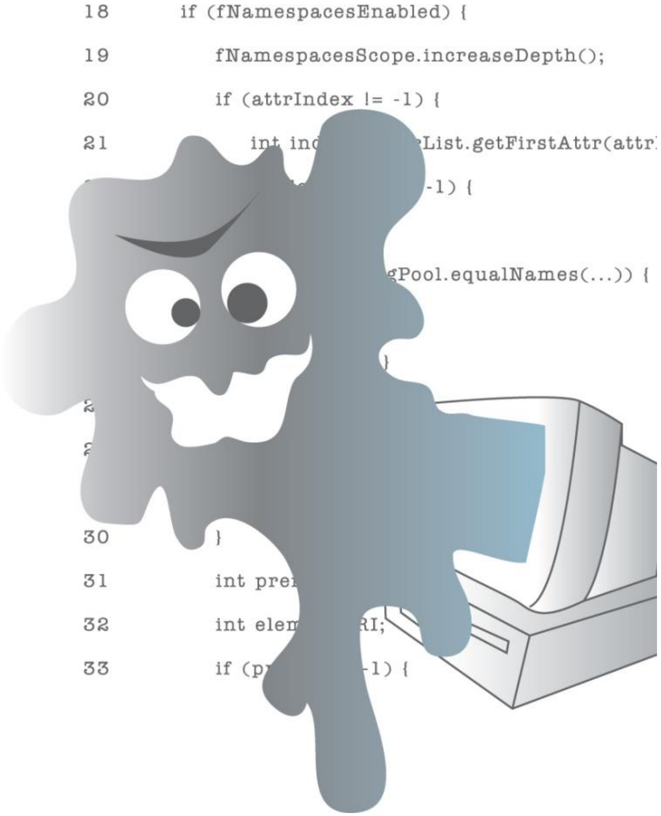
- **Symptoms:**
  - Large controller class
  - Many fields and methods with a low cohesion*
  - Lack of OO design.
  - Procedural-style than object oriented architectures.



*How closely the methods are related to the instance variables in the class.
Measure: LCOM (Lack of cohesion metric)

Blob (God Class)

- **Consequences:**
  - Lost of the benefits of using Object Oriented programming!
  - Too complex to reuse or test.
  - Expensive to load.
  - …



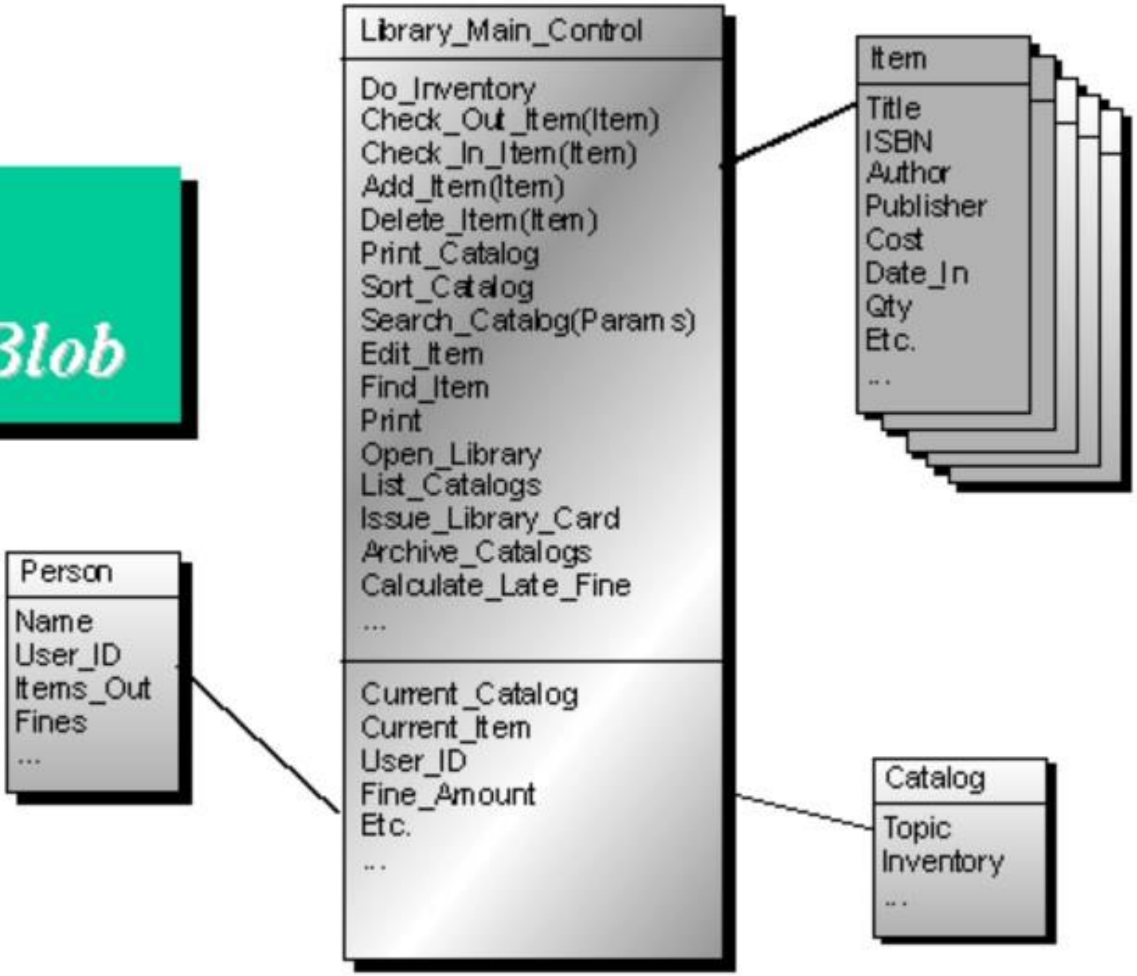
```
if (fNamespacesEnabled) {
    fNamespacesScope.increaseDepth();
    if (attrIndex != -1) {
        int ind        List.getFirstAttr(attr
                       -1) {

                       gPool.equalNames(...)) {

                       }

    }
    int pre
    int elem           RI;
    if (p              -1) {
```

*How closely the methods are related to the instance variables in the class.
Measure: LCOM (Lack of cohesion metric)

Blob (God Class)

Example:

The Library Blob



Library_Main_Control

Do_Inventory
Check_Out_Item(Item)
Check_In_Item(Item)
Add_Item(Item)
Delete_Item(Item)
Print_Catalog
Sort_Catalog
Search_Catalog(Params)
Edit_Item
Find_Item
Print
Open_Library
List_Catalogs
Issue_Library_Card
Archive_Catalogs
Calculate_Late_Fine
...

Current_Catalog
Current_Item
User_ID
Fine_Amount
Etc.
...

Item

Title
ISBN
Author
Publisher
Cost
Date_In
Qty
Etc.
...

Person

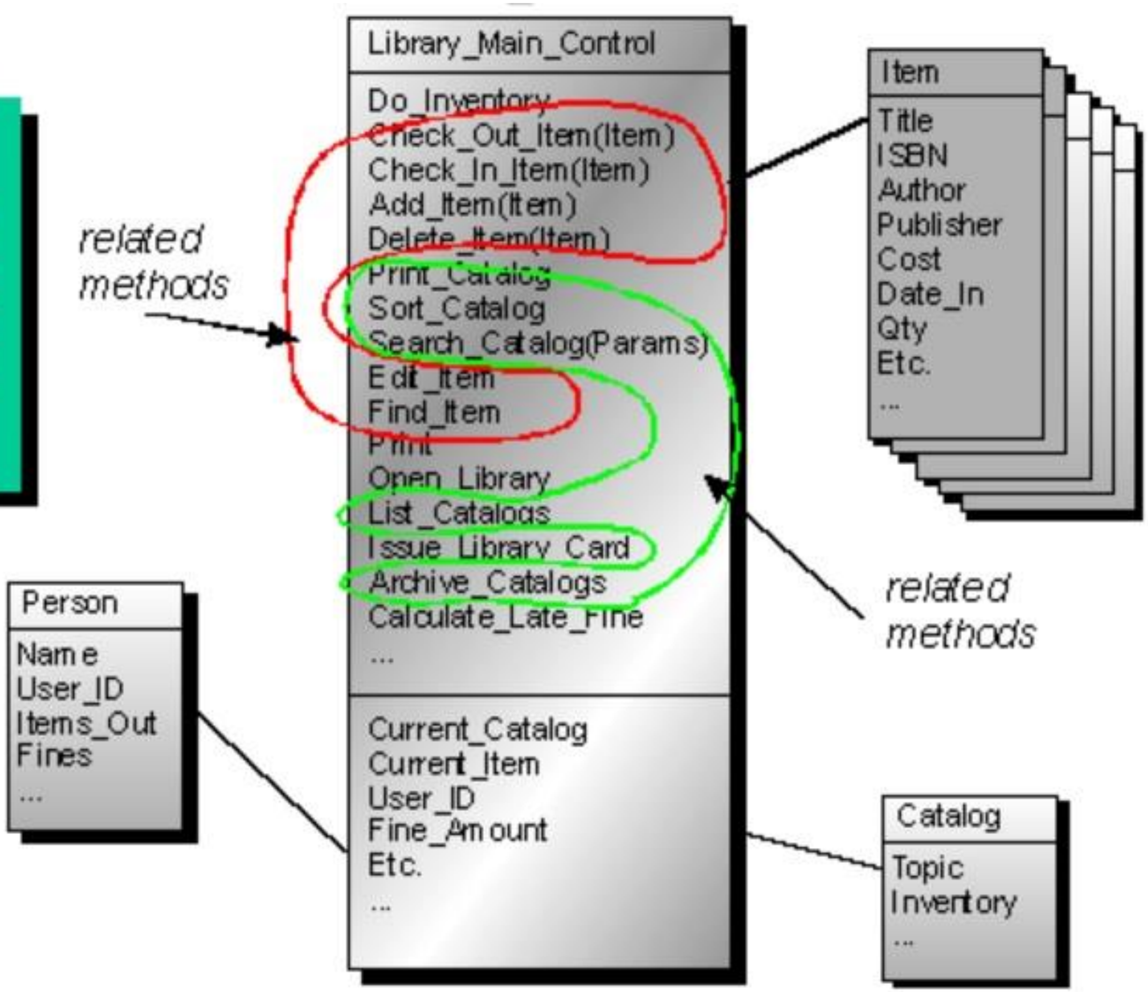
Name
User_ID
Items_Out
Fines
...

Catalog

Topic
Inventory
...

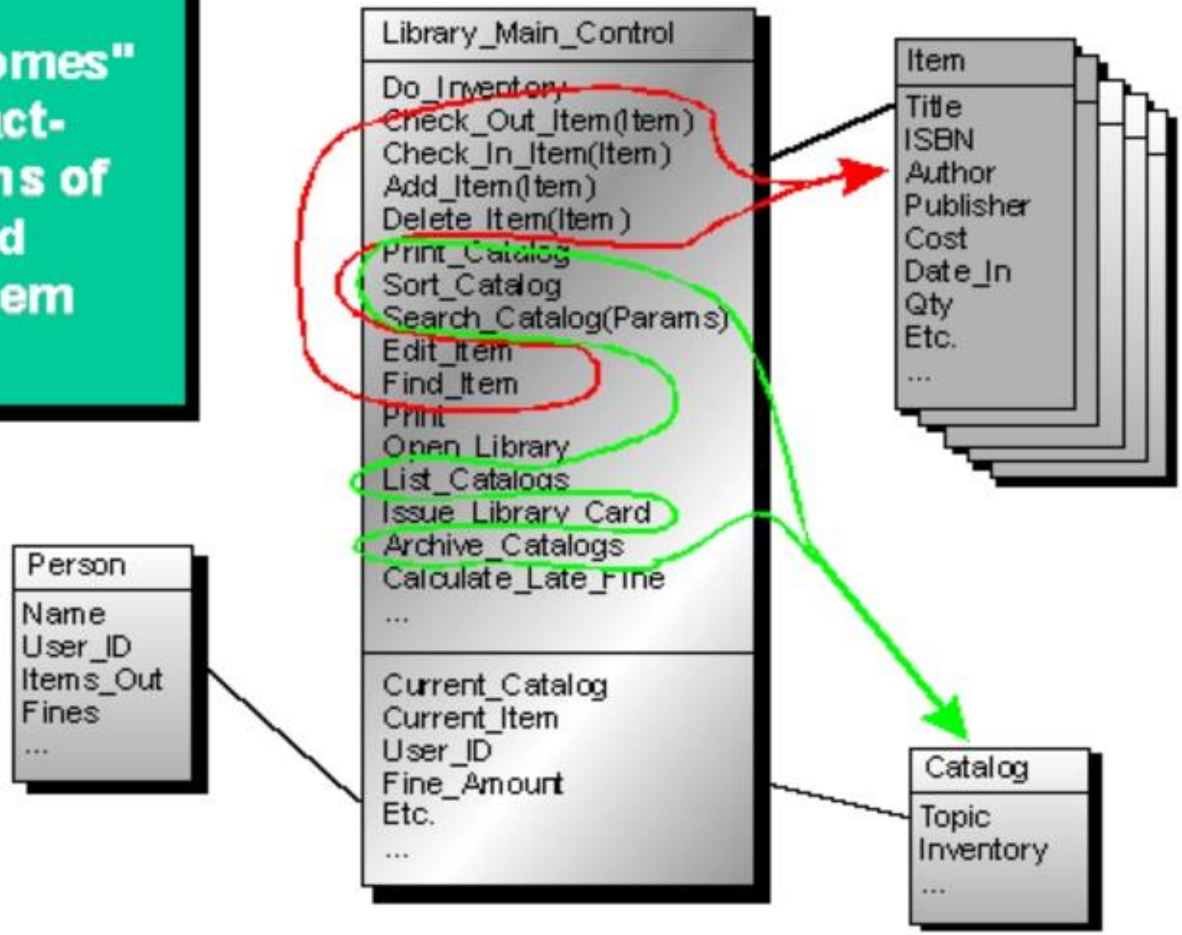
Blob (God Class) : Correction

Step 1:
Identify or categorize related attributes and operations according to contracts.



Blob (God Class) : Correction

Step 2:
Find "natural homes" for these contract-based collections of functionality and them migrate them there



Library_Main_Control

Do_Inventory
Check_Out_Item(Item)
Check_In_Item(Item)
Add_Item(Item)
Delete_Item(Item)
Print_Catalog
Sort_Catalog
Search_Catalog(Params)
Edit_Item
Find_Item
Print
Open_Library
List_Catalogs
Issue_Library_Card
Archive_Catalogs
Calculate_Late_Fine
...

Current_Catalog
Current_Item
User_ID
Fine_Amount
Etc.
...

Item

Title
ISBN
Author
Publisher
Cost
Date_In
Qty
Etc.
...

Person

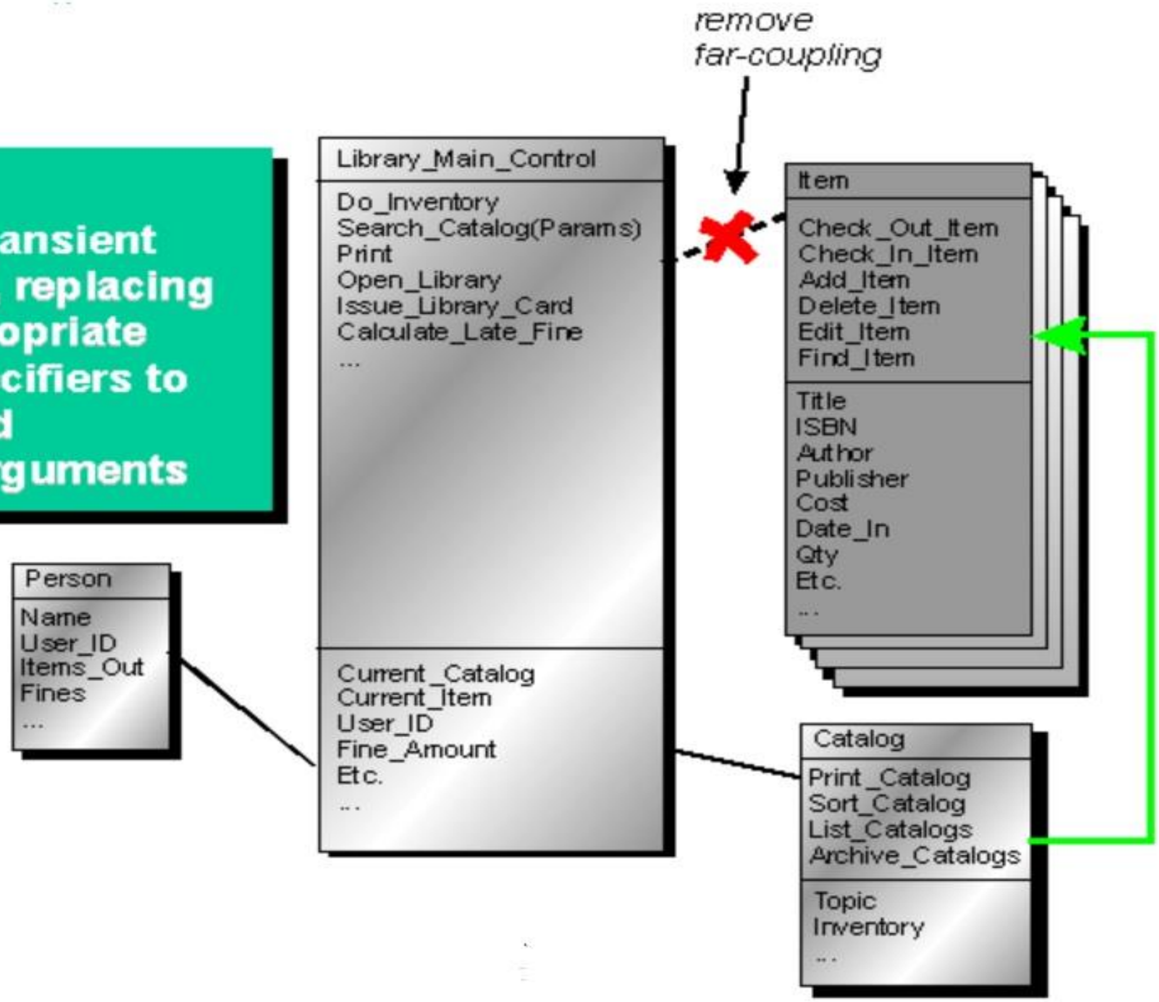
Name
User_ID
Items_Out
Fines
...

Catalog

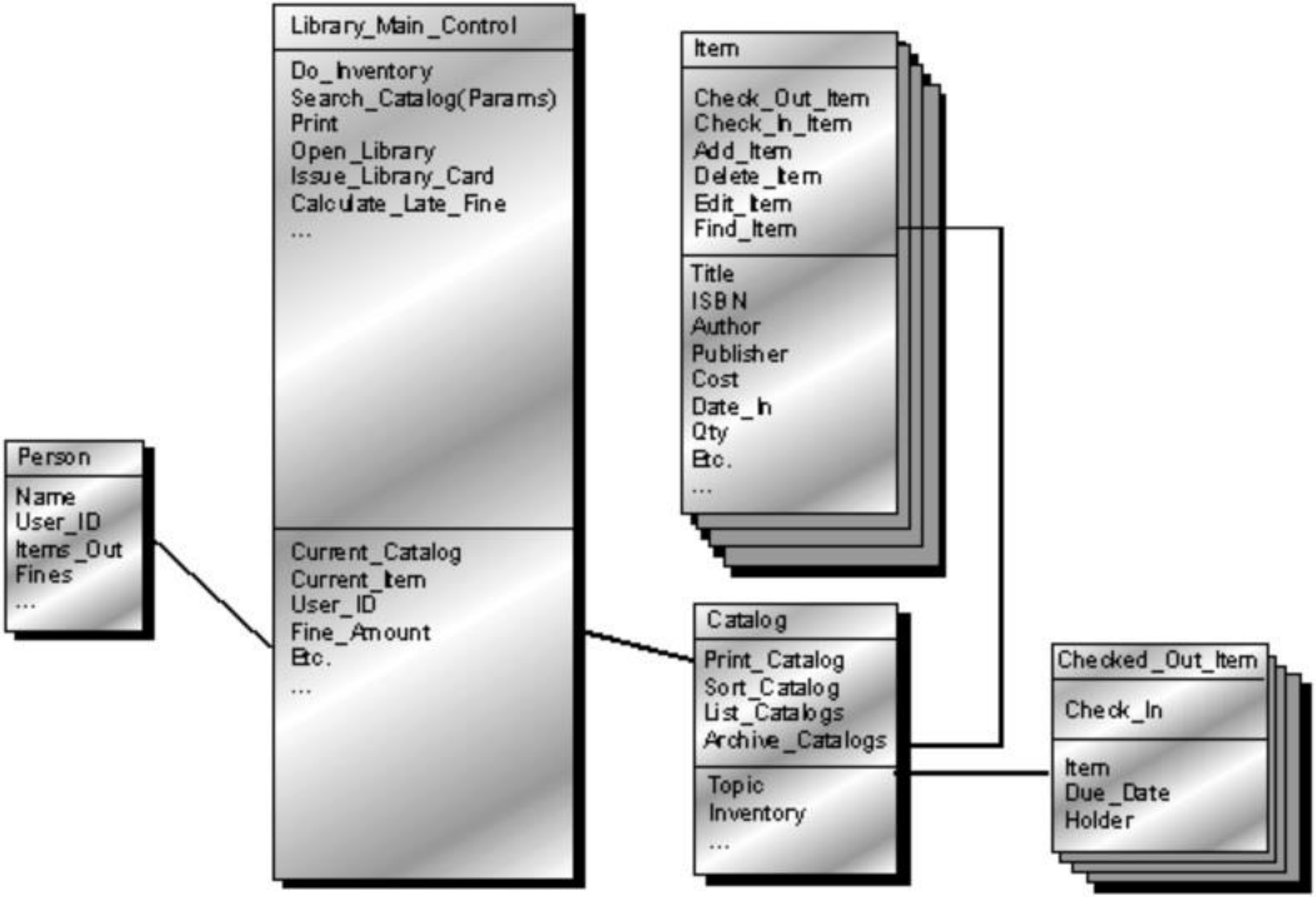
Topic
Inventory
...

Blob (God Class) : Correction

Final Step:
Remove all transient associations, replacing them as appropriate with type specifiers to attributes and operations arguments



Blob (God Class) : Correction



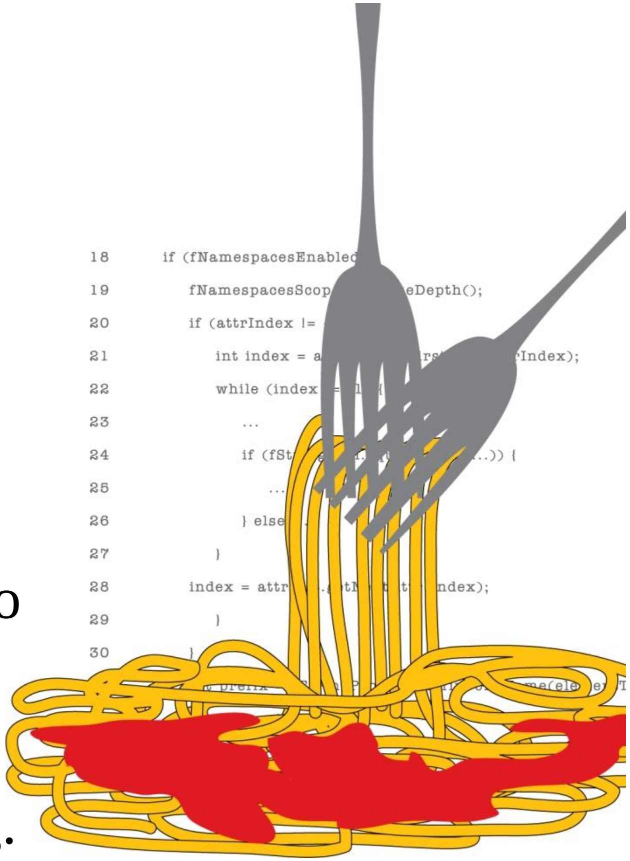
Spaghetti Code

- “Ad hoc software structure makes it difficult to extend and optimize code.”
- Manque de structure : pas d’héritage, pas de réutilisation, pas de polymorphisme

- Conception procédurale en programmation OO
- Noms des classes suggèrent une programmation procédurale
- Longues méthodes sans paramètres avec une faible cohésion
- Utilisation excessive de variables globales

Spaghetti Code

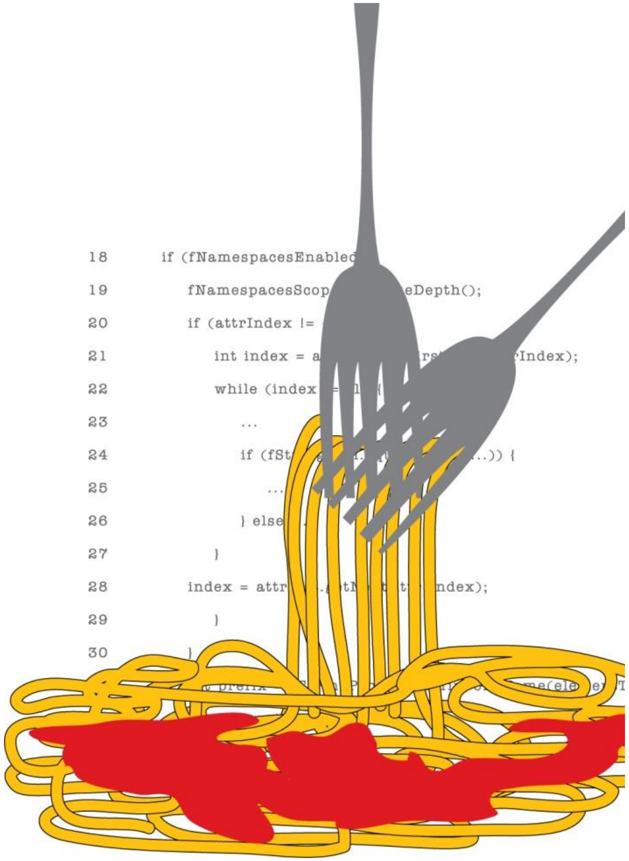
- **Symptoms :**
  - Many object methods with no parameters.
  - Lack of structure: no inheritance, no reuse, no polymorphism.
  - Long process-oriented methods with no parameters and low cohesion.
  - Procedural thinking in OO programing.



```
if (fNamespacesEnabled
    fNamespacesScop          eDepth();
    if (attrIndex !=
        int index = a                    Index);
        while (index
            ...
            if (fSt                    ...)) {
                ...
            } else
        }
        index = attr                ndex);
    }
}
```

Spaghetti Code

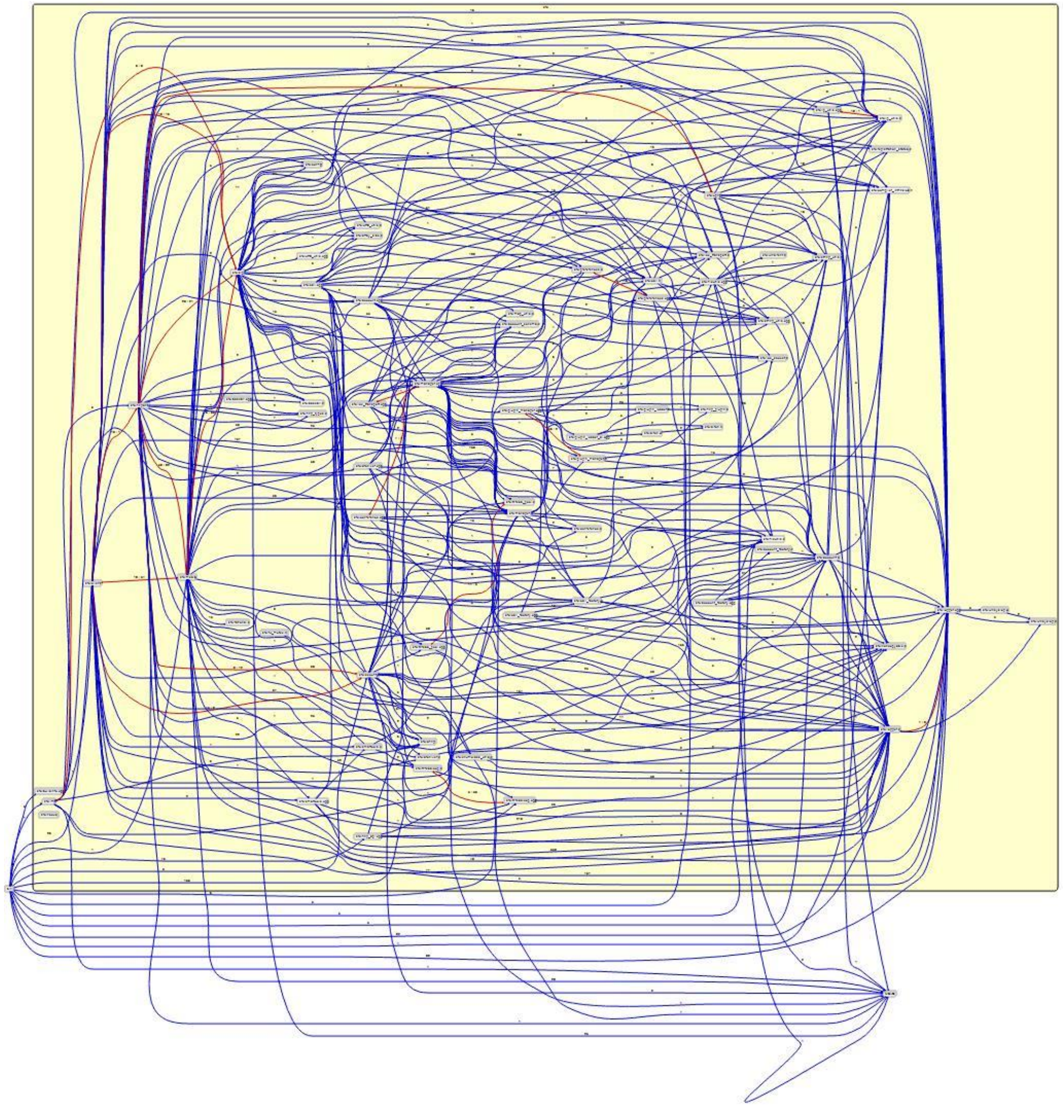
- **Consequences :**
  - The pattern of use of objects is very predictable.
  - Code is difficult to reuse.
  - Benefits of OO are lost; inheritance is not used to extend the system; polymorphism is not used.
  - Follow-on maintenance efforts contribute to the problem.



```
if (fNamespacesEnabled
    fNamespacesScop          eDepth();
    if (attrIndex !=
        int index = a              Index);
        while (index
            ...
            if (fSt                 ...)) {
                ...
            } else
        }
    index = attr                ndex);
    }
}
```

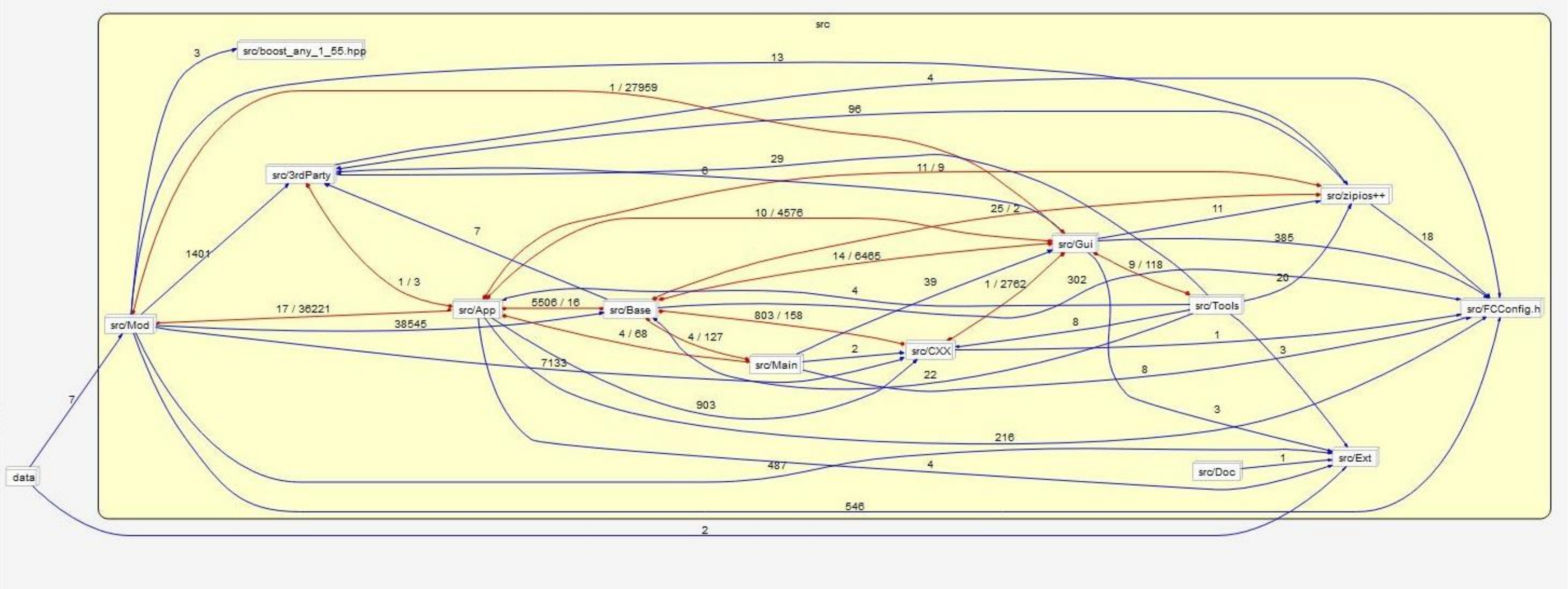
Spaghetti Code

- Ring project
- 233,492 lines of code

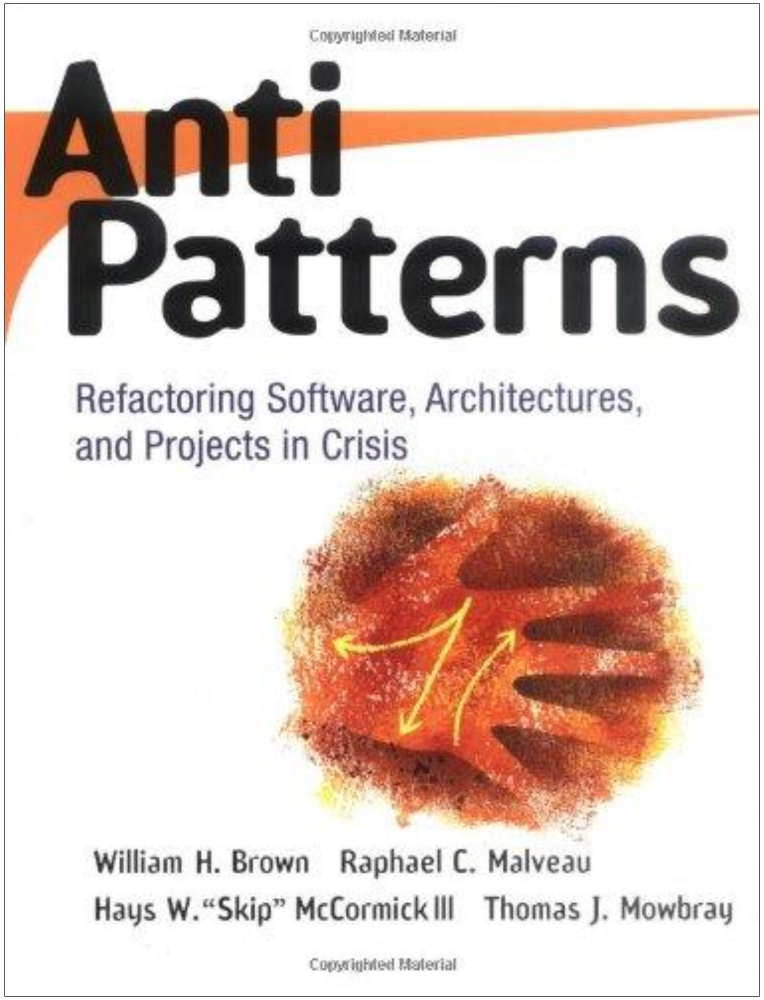


Spaghetti Code

- FreeCAD project
- 2,540,559 lines of code



Refactoring



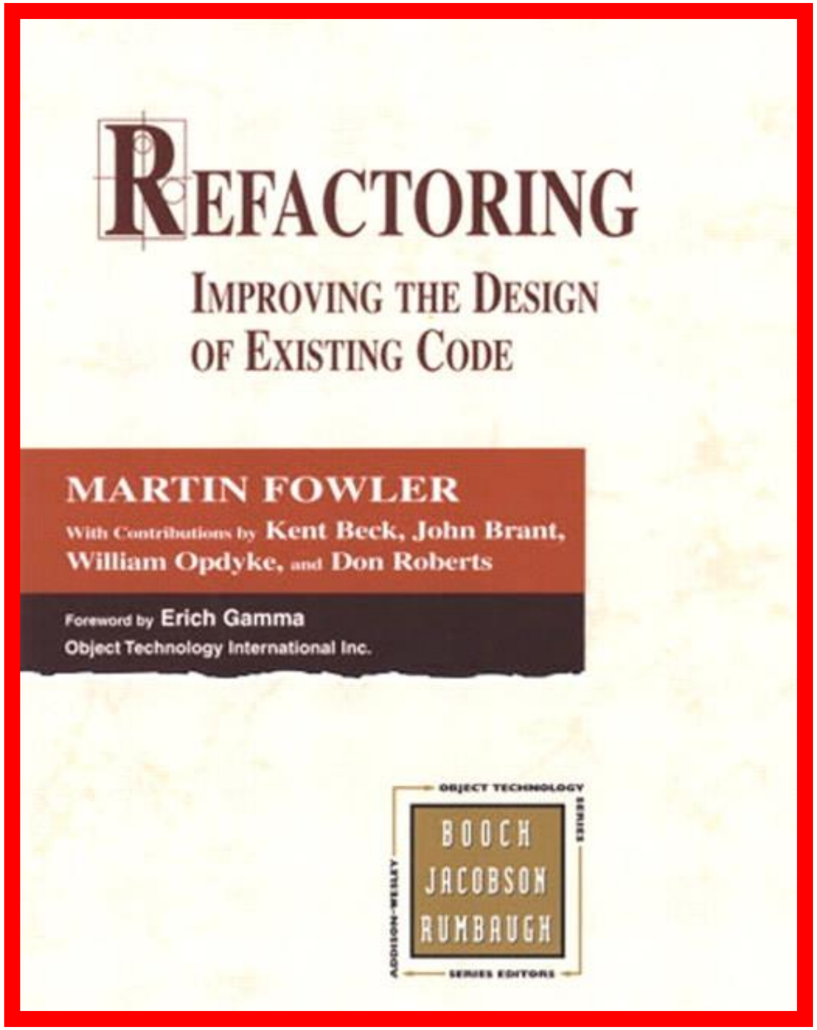
Copyrighted Material

Anti Patterns

Refactoring Software, Architectures, and Projects in Crisis

William H. Brown Raphael C. Malveau
Hays W. "Skip" McCormick III Thomas J. Mowbray

Copyrighted Material



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.

Reverse Conditional

You have a conditional that would be easier to understand if you reversed its sense.

Reverse the sense of the conditional and reorder the conditional's clauses.

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

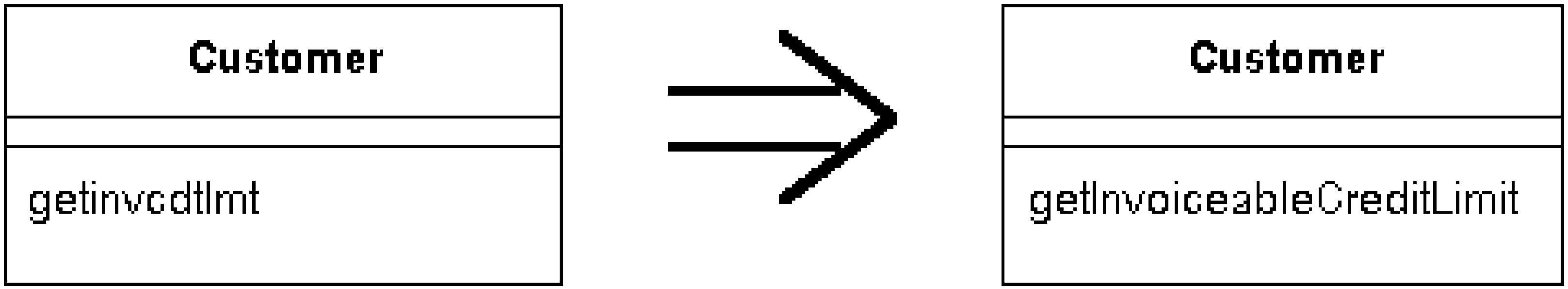
↓

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

Rename Method

The name of a method does not reveal its purpose.

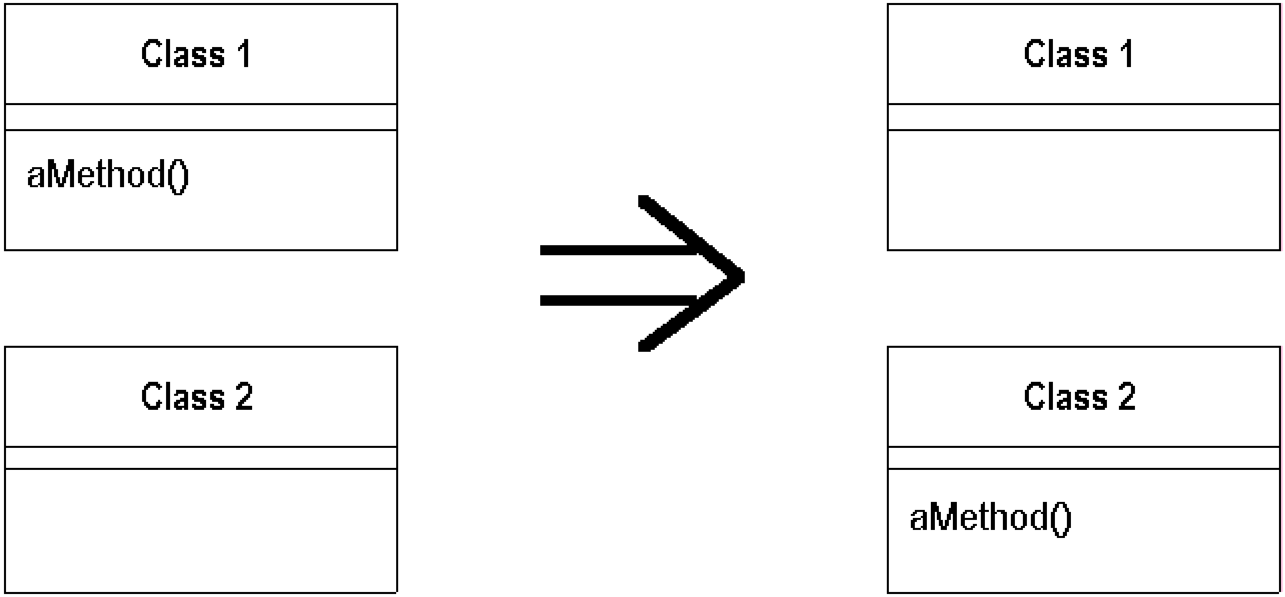
Change the name of the method.



Move Method

A method is, or will be, (using or) used by more features of another class than the class on which it is defined.

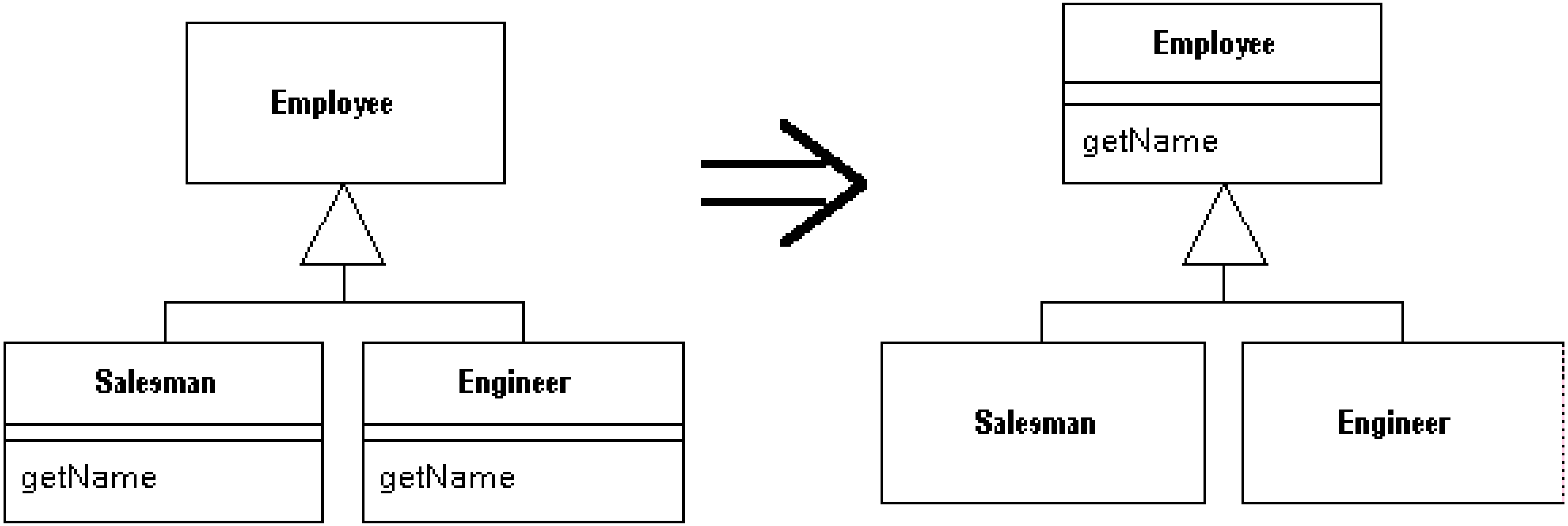
Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether.



Pull Up Method

You have methods with identical results on subclasses.

Move them to the superclass.



Extract Method

You have a code fragment that can be grouped together.

Turn the fragment into a method whose name explains the purpose of the method.

```
void printOwing() {
   printBanner();

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

↓

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

Extract Method

You have a code fragment that can be grouped together.

Turn the fragment into a method whose name explains the purpose of the method.

```
void printOwing() {
   printBanner();

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

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

Inline Method

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

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

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

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

Inline Method

Opposite to Extract Method

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

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

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

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

Replace Conditional with Polymorphism

You have a conditional that chooses different behavior depending on the type of an object.

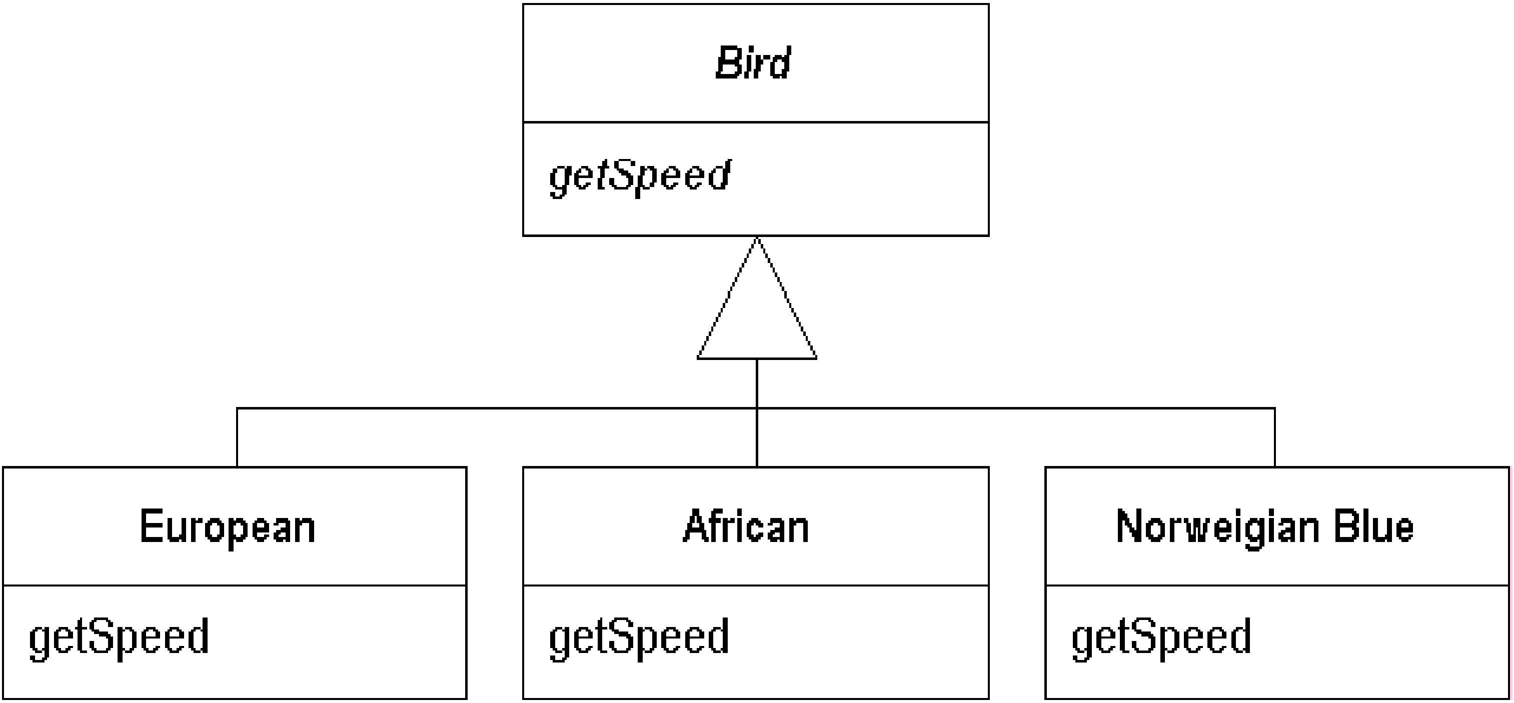
Move each leg of the conditional to an overriding method in a subclass. Make the original method abstract.

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

Replace Conditional with Polymorphism

You have a conditional that chooses different behavior depending on the type of an object.

Move each leg of the conditional to an overriding method in a subclass. Make the original method abstract.



How To Perform Refactorings

- Either manually or automatically.

- When done manually, it is always done in small steps (called refactorings).

- Larger refactorings are sequences of smaller ones

Manual Refactoring

- Manual refactoring steps should always be **small**, because:
  - They are safer this way, because the steps are simpler
  - It is easier to backtrack

  Pay attention to the mechanics:
  - Mechanics should stress **safety**

Automatic Refactoring

- When automatic support is available, it should be preferred, but ...
- ... **only if the tool is really safe.**
- Example: *Rename Method*
  - Does it check for another method with the same name?
  - Does it account for overloading?
  - Does it account for overriding?

Testing is Key When Refactoring

- Tests warn programmers of problems if they unknowningly break other parts of the application
- Tests give an immediate/quick analysis of the effects of a change

When to Refactor?

We should refactor when the code **stinks**.

"If it stinks, change it."

Grandma Beck,
discussing child-rearing philosophy

A simple example

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList)
    if (x[0] == 4)
      list1.add(x);
  return list1;

}
```

This code is quite simple
but what does it do?

https://www.slideshare.net/mariosangiorgio/clean-code-and-code-smells (slide30-46)

A simple example

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList)
    if (x[0] == 4)
      list1.add(x);
  return list1;

}
```

This code is quite simple but what does it do?

Looking at it we can’t tell what it is actually doing!

A simple example

```
public List<int[]> getFlaggedCells() {
  List<int[]> flaggedCells = new ArrayList<int[]>();
  for (int[] cell : gameBoard)
    if (cell[STATUS_VALUE] == FLAGGED)
      flaggedCells.add(x);
  return flaggedCells;
}
```

Is this code any better?

A simple example

```
public List<Cell> getFlaggedCells() {
  List<Cell> flaggedCells = new ArrayList<Cell>();
  for (Cell cell : gameBoard)
    if (cell.isFlagged())
      flaggedCells.add(x);
  return flaggedCells;
}
```

What about this?

A   simple example

What we have done:

used intention revealing names

flaggedCells rather than list1

A simple example

What we have done:

used intention revealing names	flaggedCells rather than list1
replaced *magic numbers* with constants	cell[STATUS_VALUE] rather than x[0]

A simple example

What we have done:

used intention revealing names	flaggedCells rather than list1
replaced *magic numbers* with constants	cell[STATUS_VALUE] rather than x[0]
created an appropriate abstract data type	Cell cell rather than int[] cell

Benefits

- more flexible thanks to use of objects instead of primitives int[ ].
- Better understandability and organization of code.

  Operations on particular data are in the same place, instead of being scattered.
- No more guessing about the reason for all these strange constants and why they are in an array.

Another example

```
int d;
```

What does it mean?
Days? Diameter? ...

Another example

```
int d;
```

What does it mean?

Days? Diameter? …

```
int d;        //elapsed  time in days
```

Is this any better?

Another example

```
int d;
```

What does it mean?
Days? Diameter?

..

```
int d;      //elapsed time in days
```

Is this any better?

```
int elapsedTimeInDays;
```

What about this?

One more Example

```
public bool isEdible() {
  if (this.ExpirationDate > Date.Now &&
       this.ApprovedForConsumption ==  true &&
       this.InspectorId != null) {
    return true;
  } else { return
    false;
  }
}
```

How many things is the function doing?

One more Example

```
public bool isEdible() {
  if (this.ExpirationDate > Date.Now &&
       this.ApprovedForConsumption ==  true &&
       this.InspectorId != null) {
    return true;
  } else { return
    false;
  }
}
```

1. Check expiration
2. Check approval
3. Check inspection
4. Answer the request

Can we implement it better?

Do one thing

```
public bool isEdible() {
    return isFresh()    &&
           isApproved() &&
           isInspected();
}
```

Is this any better? Why?

- Now the function is doing one thing!
- Easier to understand (shorter method)
- A change in the specifications turns into a single change in the code!

Long Method example (~1622 LOC)
https://github.com/dianaelmasri/FreeCadMod/blob/master/Gui/ViewProviderSketch.cpp

One more take…

```
public void bar(){
  foo(“A”);
  foo(“B”);
  foo(“C”);
}
```

What about this?

Don’t Repeat Yourself

```
public void bar(){
  foo(“A”);
  foo(“B”);
  foo(“C”);
}
```

```
public void bar(){
  String [] elements = {“A”, “B”, “C”};

  for(String element : elements){
    foo(element);
  }

}
```

Now the logic to handle the elements is written once for all

Avoid copy and past, it smells!!!

Refactoring and code smells

Refactorings remove *Bad Smells in the Code*
i.e., potential problems or flaws

- Some will be strong, some will be subtler
- Some smells are obvious, some aren't
- Some smells mask other problems
- Some smells go away unexpectedly when we fix something else

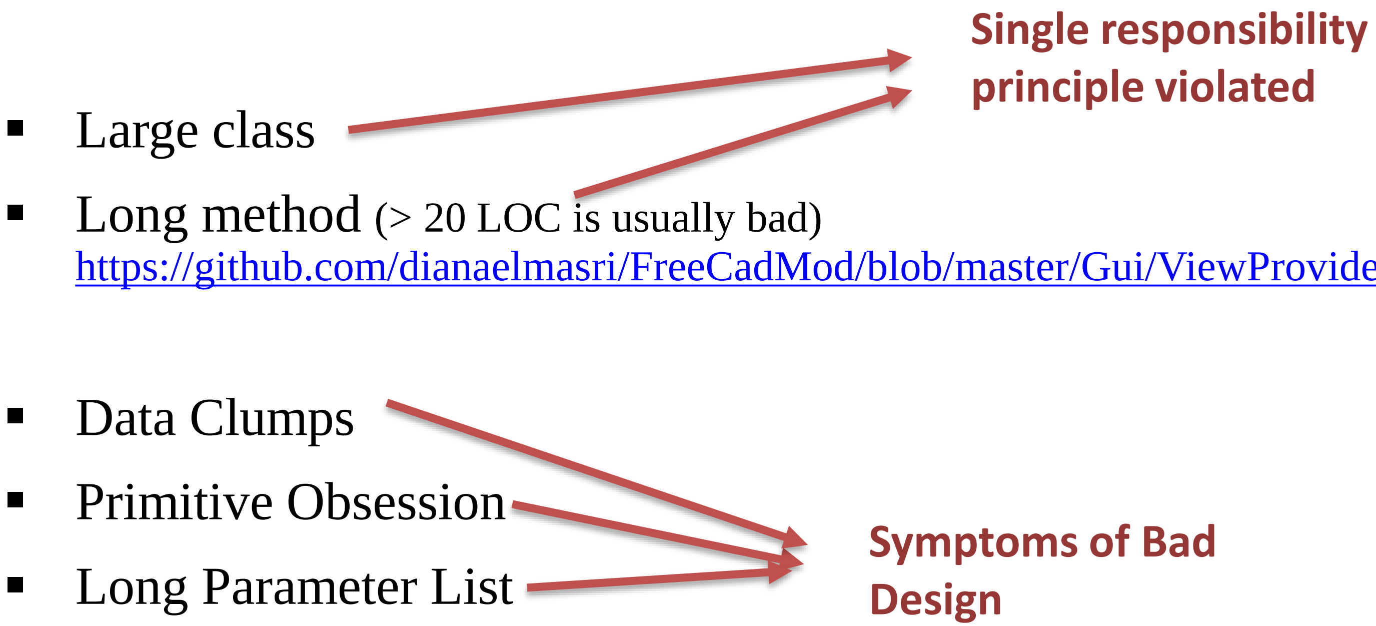
22 Code Smells

What we don’t want to see in your code

- Inappropriate naming
- Comments
- Dead code
- Duplicate code
- Primitive obsession
- Large class
- God class
- Lazy class
- Middle Man
- Data clumps
- Data class
- Long method
- Long parameter list
- Switch statements
- Speculative generality
- Oddball solution
- Feature Envy
- Refuse bequest
- Black sheep
- Contrived complexity
- Divergent change
- Shotgun surgery

Bloaters

Bloaters are code, methods and classes that have increased to such gargantuan proportions that they are hard to work with.



- Large class
- Long method (> 20 LOC is usually bad)
  https://github.com/dianaelmasri/FreeCadMod/blob/master/Gui/ViewProviderSketch.cpp

Single responsibility principle violated

- Data Clumps
- Primitive Obsession
- Long Parameter List

Symptoms of Bad Design

https://sourcemaking.com/refactoring/smells

Primitive obsession

```
public Class Car{
    private int red, green, blue;

    public void paint(int red, int green, int blue){
        this.red   = red;
        this.green = green;
        this.blue  = blue;
    }
}
```

```
public Class Car{
    private Color color;

    public void paint(Color color){
        this.color = color;
    }
}
```

https://www.slideshare.net/mariosangiorgio/clean-code-and-code-smells

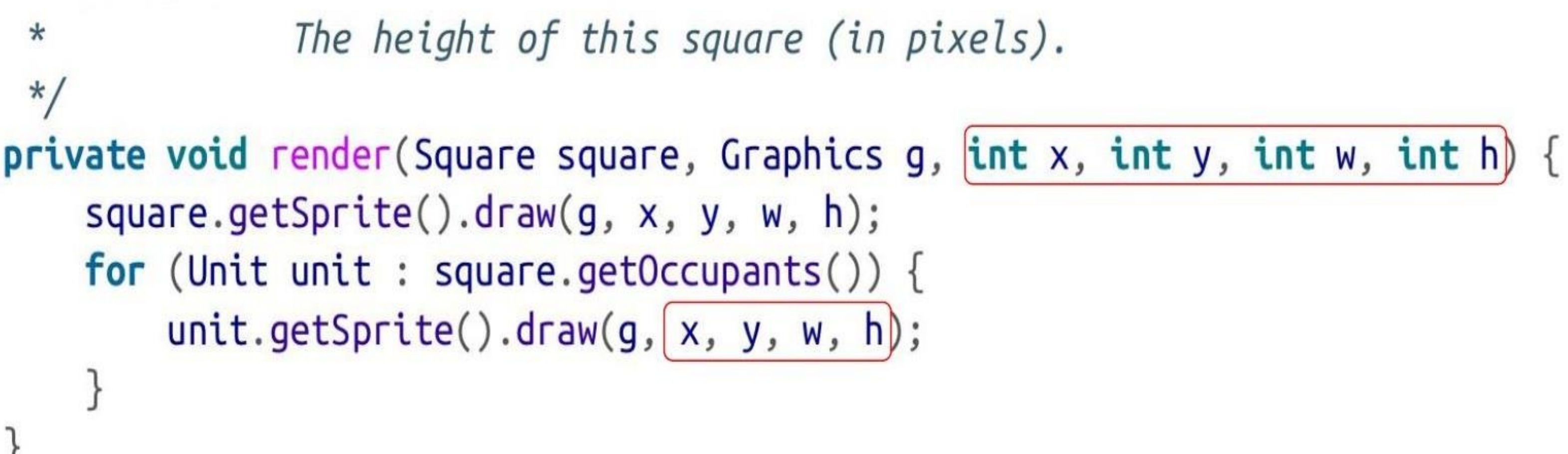
Data Clumps

```
bool SubmitCreditCardOrder (string creditCardNumber, int expirationMonth, int expirationYear,
double saleAmount)
{ }

bool Isvalid (string creditCardNumber, int expirationMonth, int expirationYear)
{ }

bool Refund(string creditCardNumber, int expirationMonth, int expirationYear, double Amount)
{ }
```

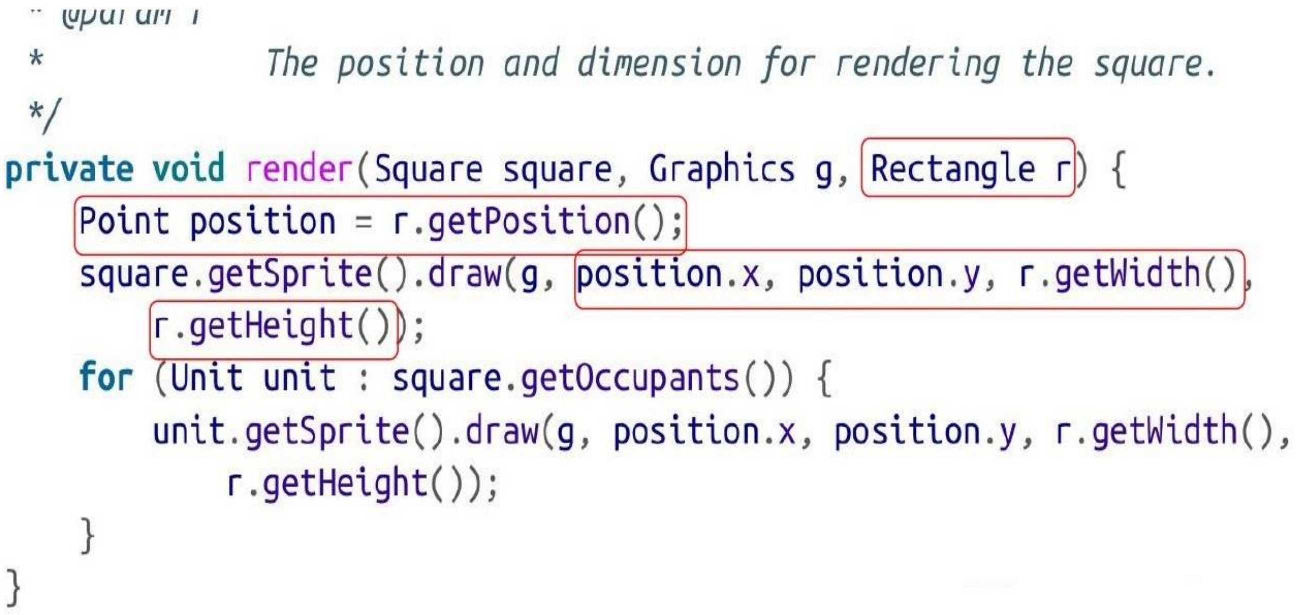
```
bool SubmitCreditCardOrder (string creditCardNumber, int expirationMonth, int expirationYear, double
saleAmount)
{     }
bool Isvalid (string creditCardNumber, int expirationMonth, int expirationYear)
{     }
bool Refund(string creditCardNumber, int expirationMonth, int expirationYear, double Amount)
{    }
```

```
class CreditCard {
private:
string creditCardNumber;,
int expirationMonth;
int expirationYear;
};
bool SubmitCreditCardOrder ( CreditCard card, double saleAmount)
{    }
bool Isvalid (CreditCard card)
{    }
bool Refund(CreditCard card , double Amount)
{      }
```

Long Parameter List

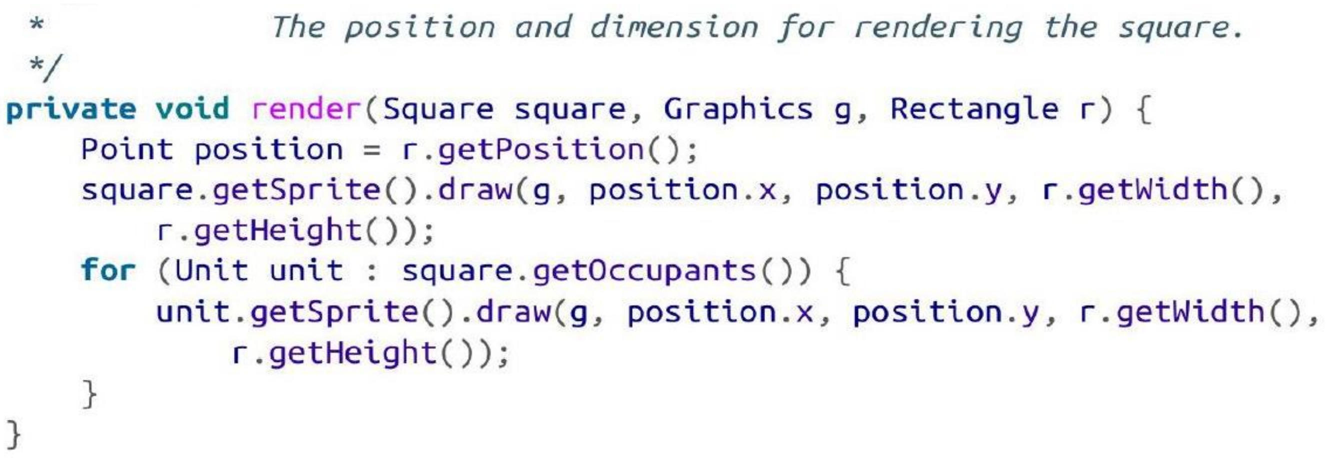


```
 *            The height of this square (in pixels).
 */
private void render(Square square, Graphics g, int x, int y, int w, int h) {
    square.getSprite().draw(g, x, y, w, h);
    for (Unit unit : square.getOccupants()) {
        unit.getSprite().draw(g, x, y, w, h);
    }
}
```

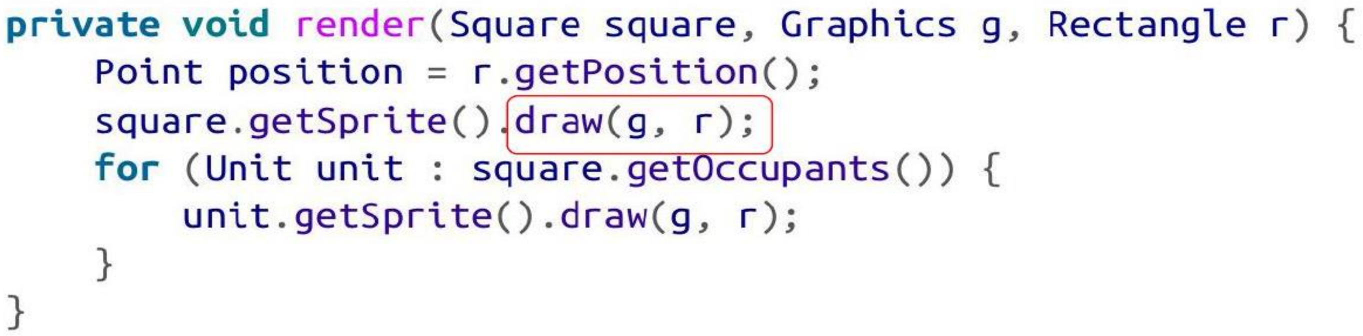


```
 * @param r
 *            The position and dimension for rendering the square.
 */
private void render(Square square, Graphics g, Rectangle r) {
    Point position = r.getPosition();
    square.getSprite().draw(g, position.x, position.y, r.getWidth(),
        r.getHeight());
    for (Unit unit : square.getOccupants()) {
        unit.getSprite().draw(g, position.x, position.y, r.getWidth(),
            r.getHeight());
    }
}
```

Long Parameter List



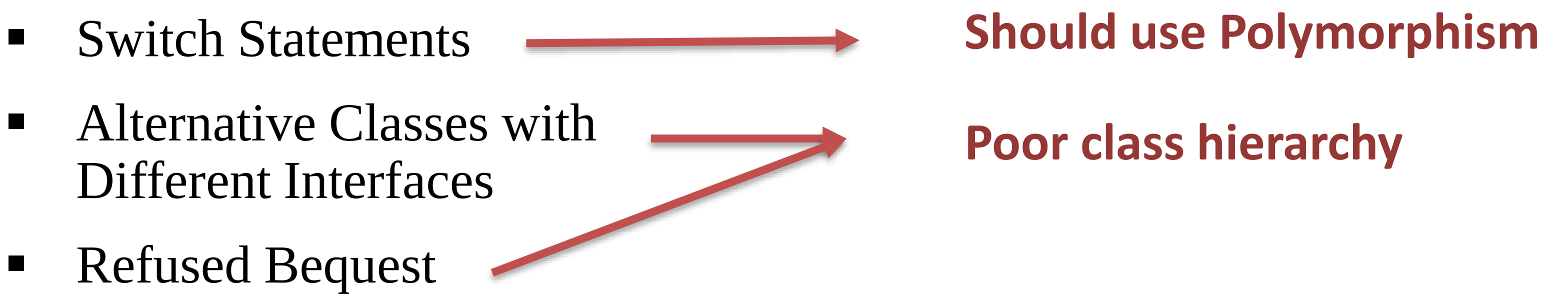
```
 *              The position and dimension for rendering the square.
 */
private void render(Square square, Graphics g, Rectangle r) {
    Point position = r.getPosition();
    square.getSprite().draw(g, position.x, position.y, r.getWidth(),
        r.getHeight());
    for (Unit unit : square.getOccupants()) {
        unit.getSprite().draw(g, position.x, position.y, r.getWidth(),
            r.getHeight());
    }
}
```



```
private void render(Square square, Graphics g, Rectangle r) {
    Point position = r.getPosition();
    square.getSprite().draw(g, r);
    for (Unit unit : square.getOccupants()) {
        unit.getSprite().draw(g, r);
    }
}
```

Object-Orientation Abusers

All these smells are incomplete or incorrect application of object-oriented programming principles.



- Switch Statements
- Alternative Classes with Different Interfaces
- Refused Bequest

Should use Polymorphism

Poor class hierarchy

https://sourcemaking.com/refactoring/smells

Switch statements

- Why is this implementation bad? How can you improve it?

```
class Animal {
 int MAMMAL = 0, BIRD = 1, REPTILE = 2;
 int myKind;  // set in constructor
 ...
 string getSkin() {
   switch (myKind) {
      case MAMMAL: return "hair";
      case BIRD: return "feathers";
      case REPTILE: return "scales";
      default: return "integument";
    }
  }
}
```

http://slideplayer.com/slide/7833453/ slides 59 to 62

Switch statements

Bad Implementation because

- A switch statement should not be used to distinguish between various kinds of object

- What if we add a new animal type?
- What if the animals differ in other ways like “Housing” or “Food:?

Switch statements

- Improved code: The simplest is the creation of subclasses

```
class Animal
{
    string getSkin() { return "integument"; }
}
class Mammal extends Animal
{
    string getSkin() { return "hair"; }
}
class Bird extends Animal
{
    string getSkin() { return "feathers"; }
}
class Reptile extends Animal
{
    string getSkin() { return "scales"; }
}
```

Switch statements

How is this an improvement?

– Adding a new animal type, such as Insect

➢ does not require revising and recompiling existing code

– Mammals, birds, and reptiles are likely to differ in other ways : class “housing” or class “food”

➢ But we’ve already separated them out so we won’t need more switch statements

✓ **we’re now using Objects as they were meant to be used**

Refused bequest

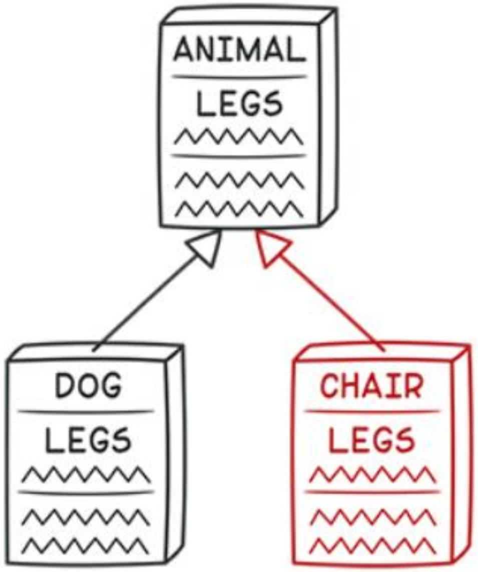
Subclass doesn’t use superclass methods and attributes

```
public abstract class Employee{
    private int quota;
    public  int getQuota();
    ...
}

public class Salesman extends Employee{ ... }

public class Engineer extends Employee{
    ...
    public int getQuota(){
        throw new NotSupportedException();
    }
}
```

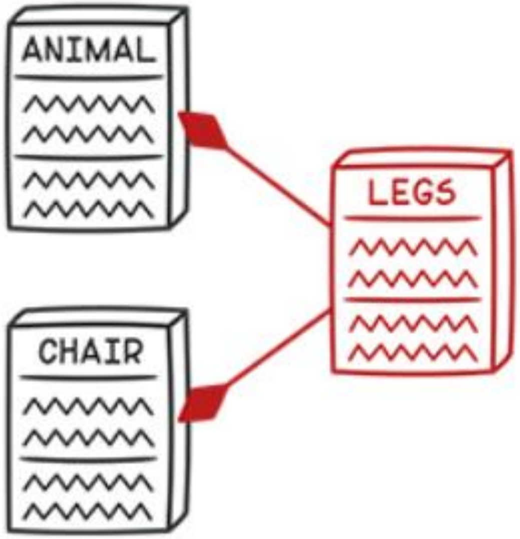
Engineer does not use quota. It should be pushed down to Salesman

Refused Bequest

Inheritance (is a …).
Does it make sense??



Delegation (has a…)



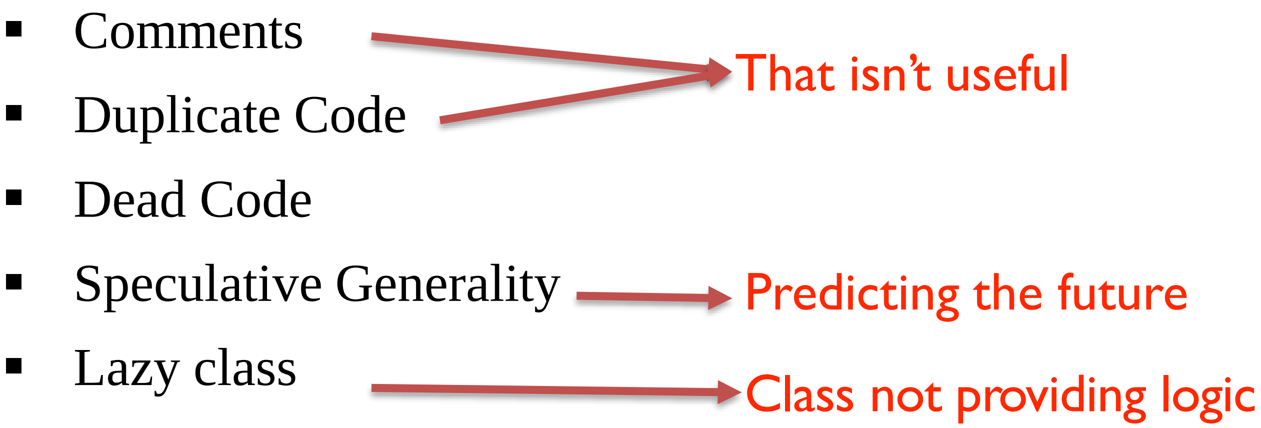
Refused Bequest

How this is an improvement?

- Won’t violate ***Liskov substitution principle,*** *i.e., if inheritance was implemented only to combine common code but not because the subclass is an extension of the superclass.*
- The subclass uses only a portion of the methods of the superclass.
  - ➢ No more calls to a superclass method that a subclass was not supposed to call.

Dispensable

Something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand.



- Comments
- Duplicate Code
- Dead Code
- Speculative Generality
- Lazy class

That isn't useful

Predicting the future

Class not providing logic

https://sourcemaking.com/refactoring/smells

Comments

Explain yourself in the code

Which one is clearer?

(A)

```
//Check to see if the employee is eligible for full benefits
if((employee.flags & HOURLY_FLAG)&&(employee.age > 65))
```

(B)

```
if(employee.isEligibleForFullBenefits())
```

Duplicate Code: In the same class

```
int a [ ];
int b [ ] ;
int sumofa = 0;
for (int i=0; i<size1; i++){
  sumofa += a[i];
}
int averageOfa= sumofa/size1;
….
int sumofb = 0;
for (int i = 0; i<size2; i++){
  sumofb += b[i];
}
int averageOfb = sumofb/size2;
```

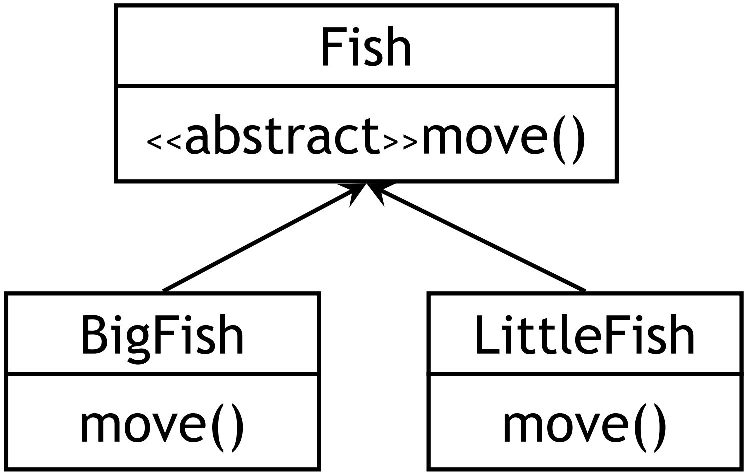
Refactor: Extract method

```
int calcAverage(int* array, int size) {

int sum= 0;
 for (int i = 0; i<size; i++)
        sum + =array[i];
 return sum/size;
}

int a[];
int b[];
int averageOfa = calcAverage(a[], size1)
int averageOfb = calcAverage(b[], size2)
```

https://www.slideshare.net/annuvinayak/code-smells-and-its-type-with-example

Duplicate Code: In the same class

- Example: consider the ocean scenario:
    - Fish move about randomly
    - A big fish can move to where a little fish is (and eat it)
    - A little fish will *not* move to where a big fish is



- General move method:

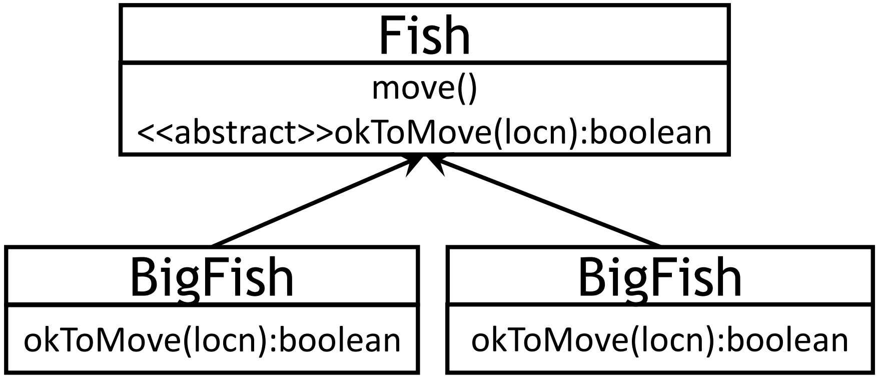
```
public void move() {
    choose a random direction;            // same for both
    find the location in that direction;  // same for both
    check if it's ok to move there;       // different
    if it's ok, make the move;            // same for both
}
```

http://slideplayer.com/slide/7833453/

Duplicate Code

Refactoring solution:

- Extract the check on whether it's ok to move
- In the Fish class, put the actual move() method
- Create an abstract okToMove() method in the Fish class
- Implement okToMove() in each subclass


Fish
move()
<<abstract>>okToMove(locn):boolean

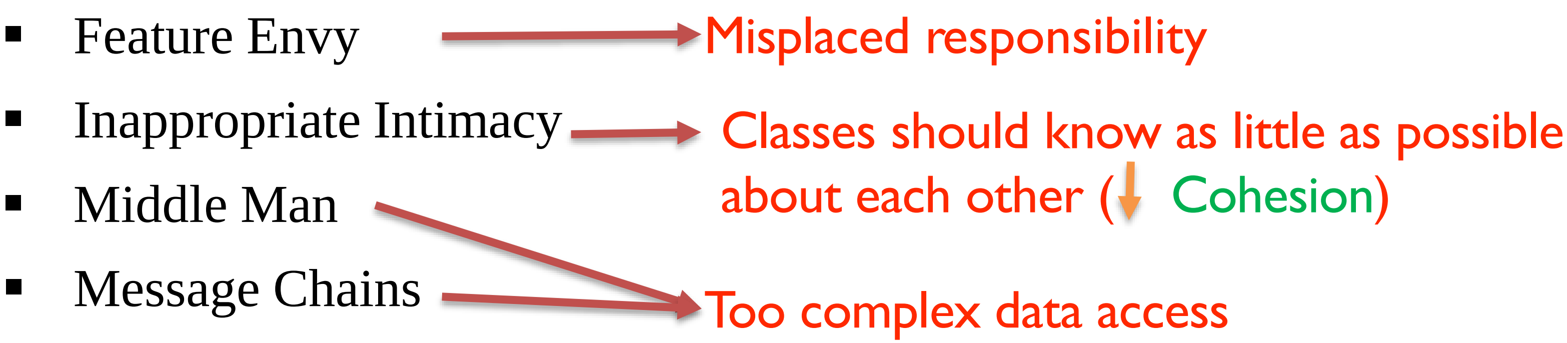
BigFish
okToMove(locn):boolean

BigFish
okToMove(locn):boolean

http://slideplayer.com/slide/7833453/

Couplers

All the smells in this group contribute to excessive coupling between classes or show what happens if coupling is replaced by excessive delegation.



- Feature Envy
- Inappropriate Intimacy
- Middle Man
- Message Chains

Misplaced responsibility

Classes should know as little as possible about each other (↓ Cohesion)

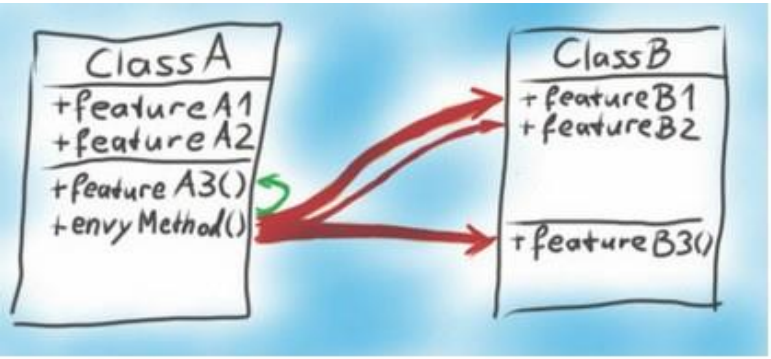
Too complex data access

https://sourcemaking.com/refactoring/smells

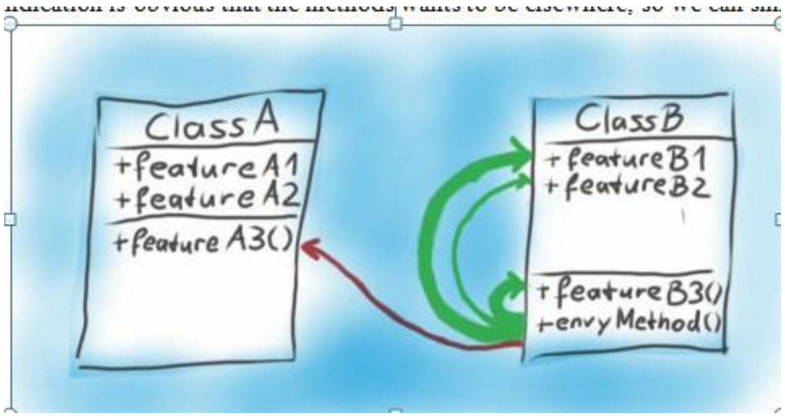
Feature Envy

It’s obvious that the method wants to be elsewhere, so we can simply use **MOVE METHOD** to give the method its dream home.

Before



Refactored



✓ We are reducing the **coupling** and enhancing the **cohesion**

https://www.slideshare.net/annuvinayak/code-smells-and-its-type-with-example

Feature Envy

- A method in one class uses primarily data and methods from another class to perform its work
  - ➢Indicates the method was incorrectly placed in the wrong class
- Problems:
  - High class coupling
  - Difficult to change , understand, and reuse
- Refactoring Solution: Extract Method & Method Movement
  - Move the method with feature envy to the class containing the most frequently used methods and data items

Feature Envy

```
class OrderItemPanel {
private:
 itemPanel _itemPanel;
 void updateItemPanel( ) {
   Item item = getItem();
   int quant = getQuantity( );
   if (item == null)
     _itemPanel.clear( );
   else{
     _itemPanel.setItem(item);
     _itemPanel.setInstock(quant);
    }
 }
}
```

http://slideplayer.com/slide/7833453/ slides 75 to 77

Feature Envy

- Method updateItemPanel is defined in class OrderItemPanel, but the method interests are in class ItemPanel

```
class OrderItemPanel {
private:
 itemPanel _itemPanel;
 void updateItemPanel( ) {
   Item item = getItem();
   int quant = getQuantity( );
   if (item == null)
     _itemPanel.clear( );
   else{
     _itemPanel.setItem(item);
     _itemPanel.setInstock(quant);
    }
 }
}
```

- *Refactoring* solution:
  - – Extract method doUpdate in class OrderItemPanel
  - – Move method doUpdate to class ItemPanel

```
class OrderItemPanel {
private:
 itemPanel _itemPanel;
 void updateItemPanel( ) {
   Item item = getItem();
   int quant = getQuantity( );
   _itemPanel.doUpdate(item, quant);
 }
}
class ItemPanel {
public:
 void doUpdate(Item item, int quantity){
    if (item == null)
     clear( );
    else{
     setItem(item);
     setInstock(quantity);
    }
 }
}
```

Message chains

```
a.getB().getC().getD().getTheNeededData()
```

```
a.getTheNeededData()
```

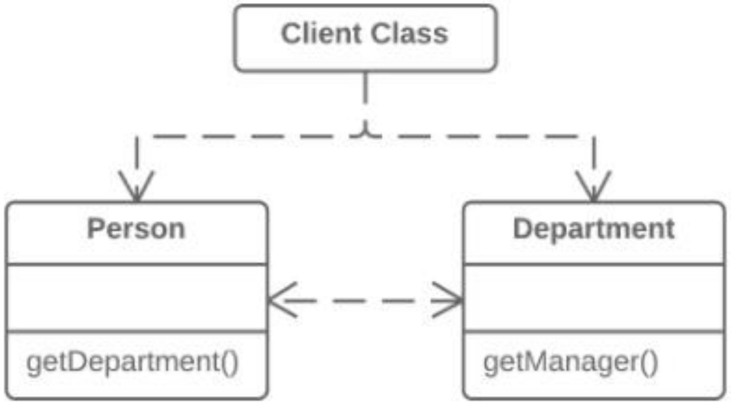
Law of Demeter: Each unit should only talk with friends

https://www.slideshare.net/mariosangiorgio/clean-code-and-code-smells

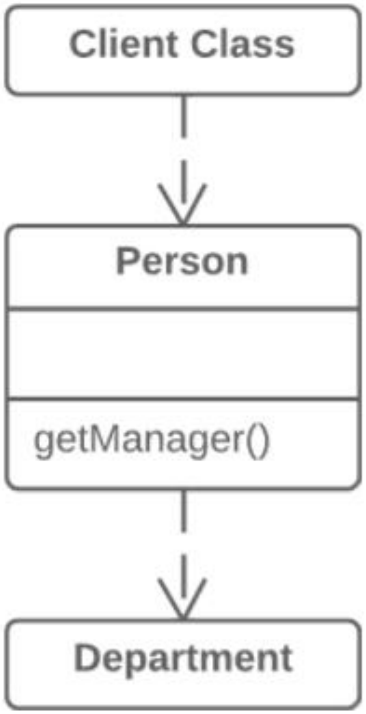
Message chains

- To refactor a message chain, use Hide Delegate.

Message chains



Refactor: Hide delegate



https://sourcemaking.com/refactoring/smells

Change preventers

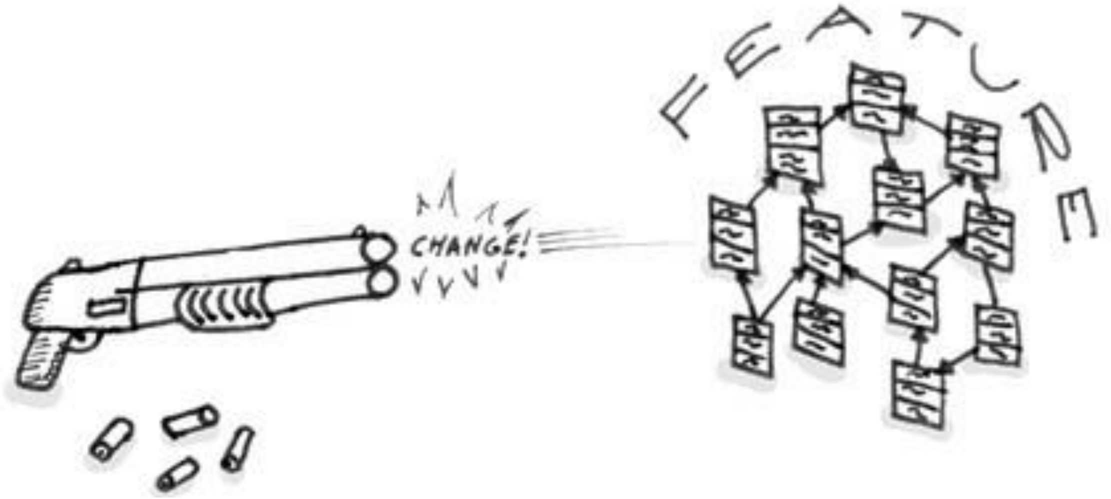
if you need to change something in one place in your code, you have to make many changes in other places too.
Program development becomes much more complicated and expensive as a result.

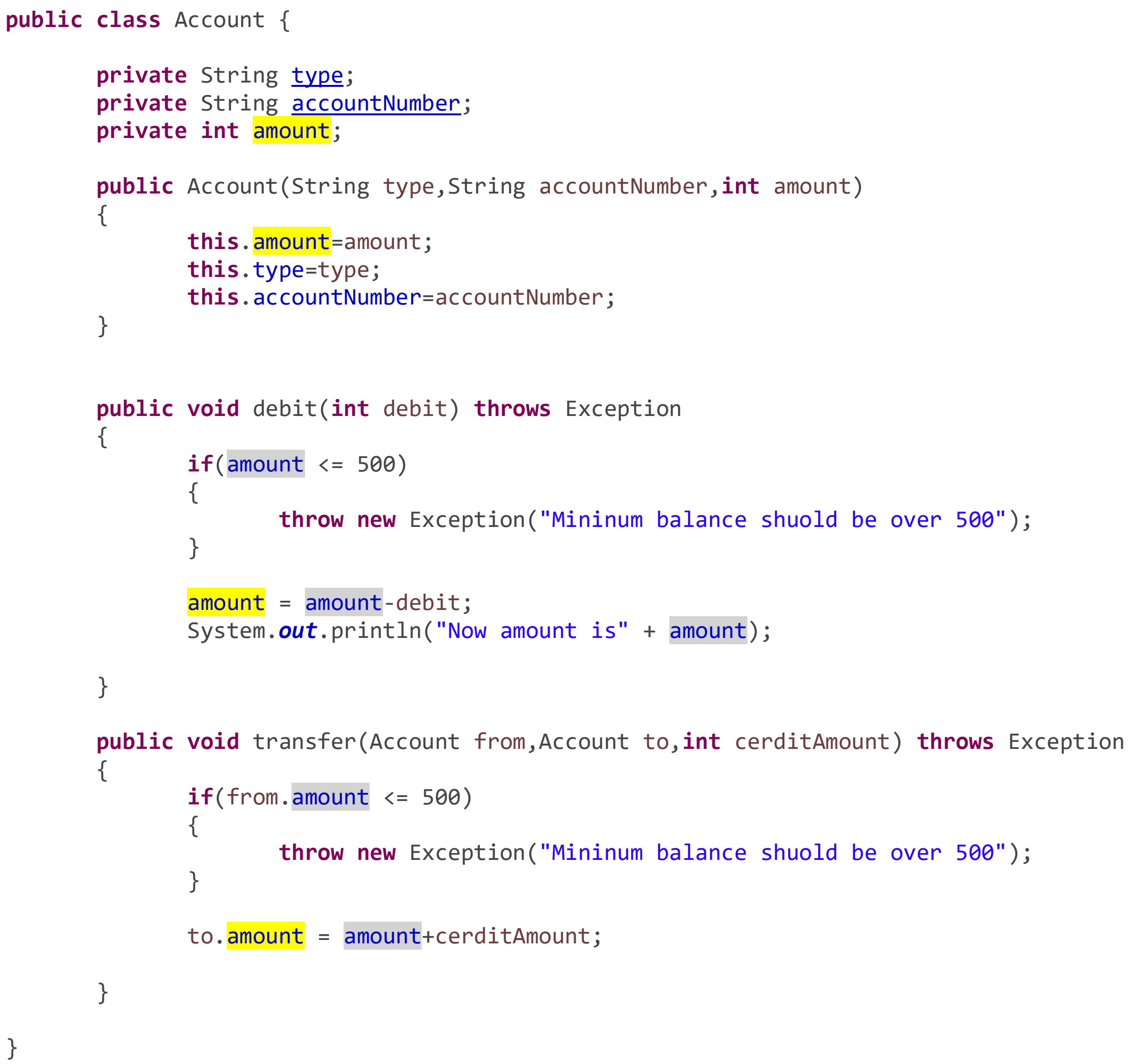
- Divergent change → A class has to be changed in several parts
- Shotgun surgery → A single change requires changes in severall classes
- Parallel Inheritance Hierarchies

https://sourcemaking.com/refactoring/smells

Shotgun surgery

When changes are all over the place, they are hard to find and it’s easy to miss an important change





```
public class Account {

        private String type;
        private String accountNumber;
        private int amount;

        public Account(String type,String accountNumber,int amount)
        {
                this.amount=amount;
                this.type=type;
                this.accountNumber=accountNumber;
        }


        public void debit(int debit) throws Exception
        {
                if(amount <= 500)
                {
                        throw new Exception("Mininum balance shuold be over 500");
                }

                amount = amount-debit;
                System.out.println("Now amount is" + amount);

        }

        public void transfer(Account from,Account to,int cerditAmount) throws Exception
        {
                if(from.amount <= 500)
                {
                        throw new Exception("Mininum balance shuold be over 500");
                }

                to.amount = amount+cerditAmount;

        }

}
```

The problem occurs when we add another criterion in validation logic that is if account type is **personal and balance is over 500** then we can perform above operations

http://javaonfly.blogspot.ca/2016/09/code-smell-and-shotgun-surgery.html

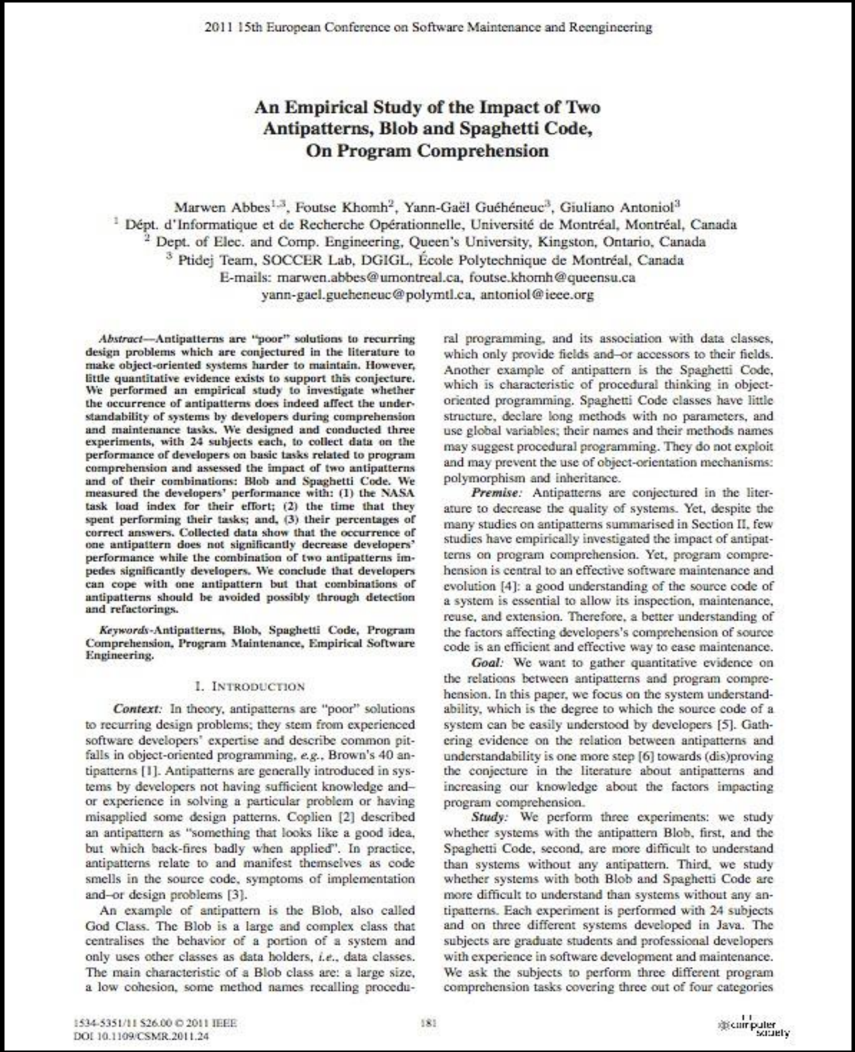
```java
public class AcountRefactored {

        private String type;
        private String accountNumber;
        private int amount;



        public AcountRefactored(String type,String accountNumber,int amount)
        {
                this.amount=amount;
                this.type=type;
                this.accountNumber=accountNumber;
        }

        private boolean isAccountUnderflow()
        {
                if(amount <= 500)
                {
                        return true;
                }
                return false;

        }


        public void debit(int debit) throws Exception
        {
                if(isAccountUnderflow())
                {
                        throw new Exception("Mininum balance shuold be over 500");
                }

                amount = amount-debit;
                System.out.println("Now amount is" + amount);

        }

        public void transfer(AcountRefactored from,AcountRefactored to,int cerditAmount) throwsException
        {
                if(isAccountUnderflow())
                {
                        throw new Exception("Mininum balance shuold be over 500");
                }

                to.amount = amount+cerditAmount;

        }
```

http://javaonfly.blogspot.ca/2016/09/code-smell-and-shotgun-surgery.html

Negative Impact of Bad Smells

Bad Smells hinder code comprehensibility

[Abbes et al. CSMR 2011]



2011 15th European Conference on Software Maintenance and Reengineering

An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, On Program Comprehension

Marwen Abbes[1,3], Foutse Khomh[2], Yann-Gaël Guéhéneuc[3], Giuliano Antoniol[3]
[1] Dépt. d'Informatique et de Recherche Opérationnelle, Université de Montréal, Montréal, Canada
[2] Dept. of Elec. and Comp. Engineering, Queen's University, Kingston, Ontario, Canada
[3] Ptidej Team, SOCCER Lab, DGIGL, École Polytechnique de Montréal, Canada
E-mails: marwen.abbes@umontreal.ca, foutse.khomh@queensu.ca
yann-gael.gueheneuc@polymtl.ca, antoniol@ieee.org

***Abstract*—Antipatterns are "poor" solutions to recurring design problems which are conjectured in the literature to make object-oriented systems harder to maintain. However, little quantitative evidence exists to support this conjecture. We performed an empirical study to investigate whether the occurrence of antipatterns does indeed affect the understandability of systems by developers during comprehension and maintenance tasks. We designed and conducted three experiments, with 24 subjects each, to collect data on the performance of developers on basic tasks related to program comprehension and assessed the impact of two antipatterns and of their combinations: Blob and Spaghetti Code. We measured the developers' performance with: (1) the NASA task load index for their effort; (2) the time that they spent performing their tasks; and, (3) their percentages of correct answers. Collected data show that the occurrence of one antipattern does not significantly decrease developers' performance while the combination of two antipatterns impedes significantly developers. We conclude that developers can cope with one antipattern but that combinations of antipatterns should be avoided possibly through detection and refactorings.**

***Keywords*-Antipatterns, Blob, Spaghetti Code, Program Comprehension, Program Maintenance, Empirical Software Engineering.**

I. Introduction

Context: In theory, antipatterns are "poor" solutions to recurring design problems; they stem from experienced software developers' expertise and describe common pitfalls in object-oriented programming, *e.g.*, Brown's 40 antipatterns [1]. Antipatterns are generally introduced in systems by developers not having sufficient knowledge and–or experience in solving a particular problem or having misapplied some design patterns. Coplien [2] described an antipattern as "something that looks like a good idea, but which back-fires badly when applied". In practice, antipatterns relate to and manifest themselves as code smells in the source code, symptoms of implementation and–or design problems [3].

An example of antipattern is the Blob, also called God Class. The Blob is a large and complex class that centralises the behavior of a portion of a system and only uses other classes as data holders, *i.e.*, data classes. The main characteristic of a Blob class are: a large size, a low cohesion, some method names recalling procedural programming, and its association with data classes, which only provide fields and–or accessors to their fields. Another example of antipattern is the Spaghetti Code, which is characteristic of procedural thinking in object-oriented programming. Spaghetti Code classes have little structure, declare long methods with no parameters, and use global variables; their names and their methods names may suggest procedural programming. They do not exploit and may prevent the use of object-orientation mechanisms: polymorphism and inheritance.

Premise: Antipatterns are conjectured in the literature to decrease the quality of systems. Yet, despite the many studies on antipatterns summarised in Section II, few studies have empirically investigated the impact of antipatterns on program comprehension. Yet, program comprehension is central to an effective software maintenance and evolution [4]: a good understanding of the source code of a system is essential to allow its inspection, maintenance, reuse, and extension. Therefore, a better understanding of the factors affecting developers's comprehension of source code is an efficient and effective way to ease maintenance.

Goal: We want to gather quantitative evidence on the relations between antipatterns and program comprehension. In this paper, we focus on the system understandability, which is the degree to which the source code of a system can be easily understood by developers [5]. Gathering evidence on the relation between antipatterns and understandability is one more step [6] towards (dis)proving the conjecture in the literature about antipatterns and increasing our knowledge about the factors impacting program comprehension.

Study: We perform three experiments: we study whether systems with the antipattern Blob, first, and the Spaghetti Code, second, are more difficult to understand than systems without any antipattern. Third, we study whether systems with both Blob and Spaghetti Code are more difficult to understand than systems without any antipatterns. Each experiment is performed with 24 subjects and on three different systems developed in Java. The subjects are graduate students and professional developers with experience in software development and maintenance. We ask the subjects to perform three different program comprehension tasks covering three out of four categories

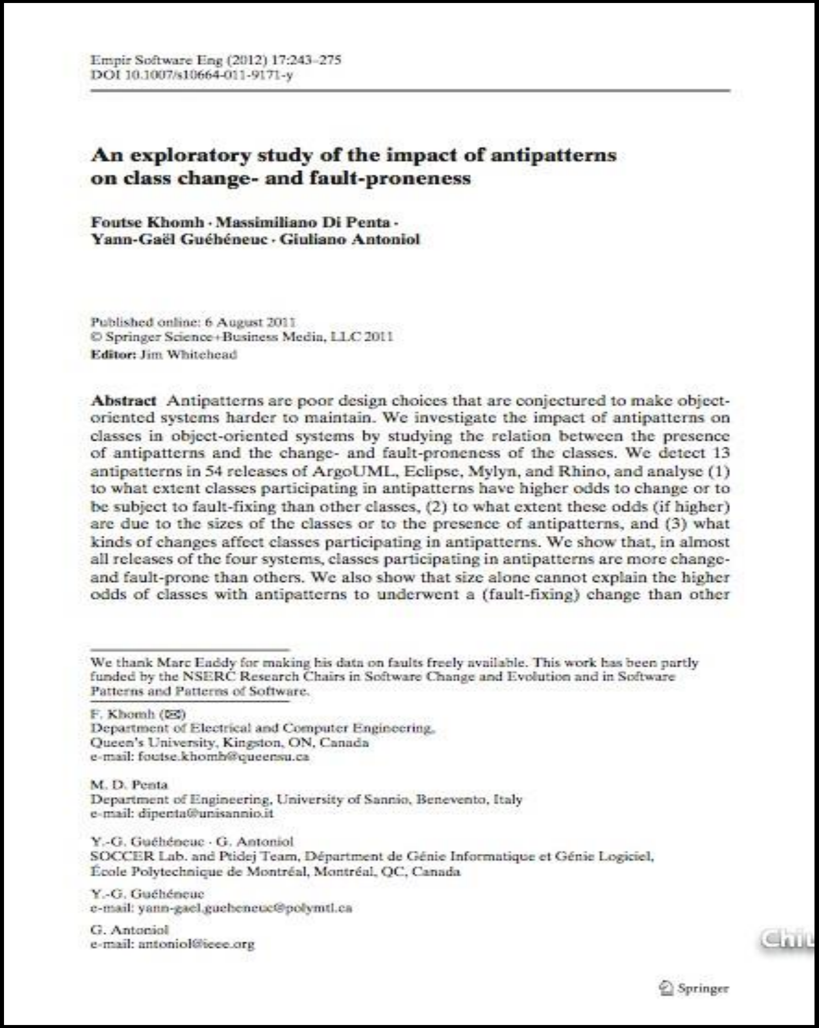
1534-5351/11 $26.00 © 2011 IEEE
DOI 10.1109/CSMR.2011.24

https://www.slideshare.net/fabiopalomba/icse15-smell-inducingchange

Negative Impact of Bad Smells

Bad Smells increase change- and fault-proneness

[Khomh et al. EMSE 2012]



Empir Software Eng (2012) 17:243–275
DOI 10.1007/s10664-011-9171-y

An exploratory study of the impact of antipatterns on class change- and fault-proneness

Foutse Khomh · Massimiliano Di Penta · Yann-Gaël Guéhéneuc · Giuliano Antoniol

Published online: 6 August 2011
© Springer Science+Business Media, LLC 2011
Editor: Jim Whitehead

Abstract Antipatterns are poor design choices that are conjectured to make object-oriented systems harder to maintain. We investigate the impact of antipatterns on classes in object-oriented systems by studying the relation between the presence of antipatterns and the change- and fault-proneness of the classes. We detect 13 antipatterns in 54 releases of ArgoUML, Eclipse, Mylyn, and Rhino, and analyse (1) to what extent classes participating in antipatterns have higher odds to change or to be subject to fault-fixing than other classes, (2) to what extent these odds (if higher) are due to the sizes of the classes or to the presence of antipatterns, and (3) what kinds of changes affect classes participating in antipatterns. We show that, in almost all releases of the four systems, classes participating in antipatterns are more change- and fault-prone than others. We also show that size alone cannot explain the higher odds of classes with antipatterns to underwent a (fault-fixing) change than other

We thank Marc Eaddy for making his data on faults freely available. This work has been partly funded by the NSERC Research Chairs in Software Change and Evolution and in Software Patterns and Patterns of Software.

F. Khomh (✉)
Department of Electrical and Computer Engineering,
Queen's University, Kingston, ON, Canada
e-mail: foutse.khomh@queensu.ca

M. D. Penta
Department of Engineering, University of Sannio, Benevento, Italy
e-mail: dipenta@unisannio.it

Y.-G. Guéhéneuc · G. Antoniol
SOCCER Lab. and Ptidej Team, Départment de Génie Informatique et Génie Logiciel,
École Polytechnique de Montréal, Montréal, QC, Canada

Y.-G. Guéhéneuc
e-mail: yann-gael.gueheneuc@polymtl.ca

G. Antoniol
e-mail: antoniol@ieee.org

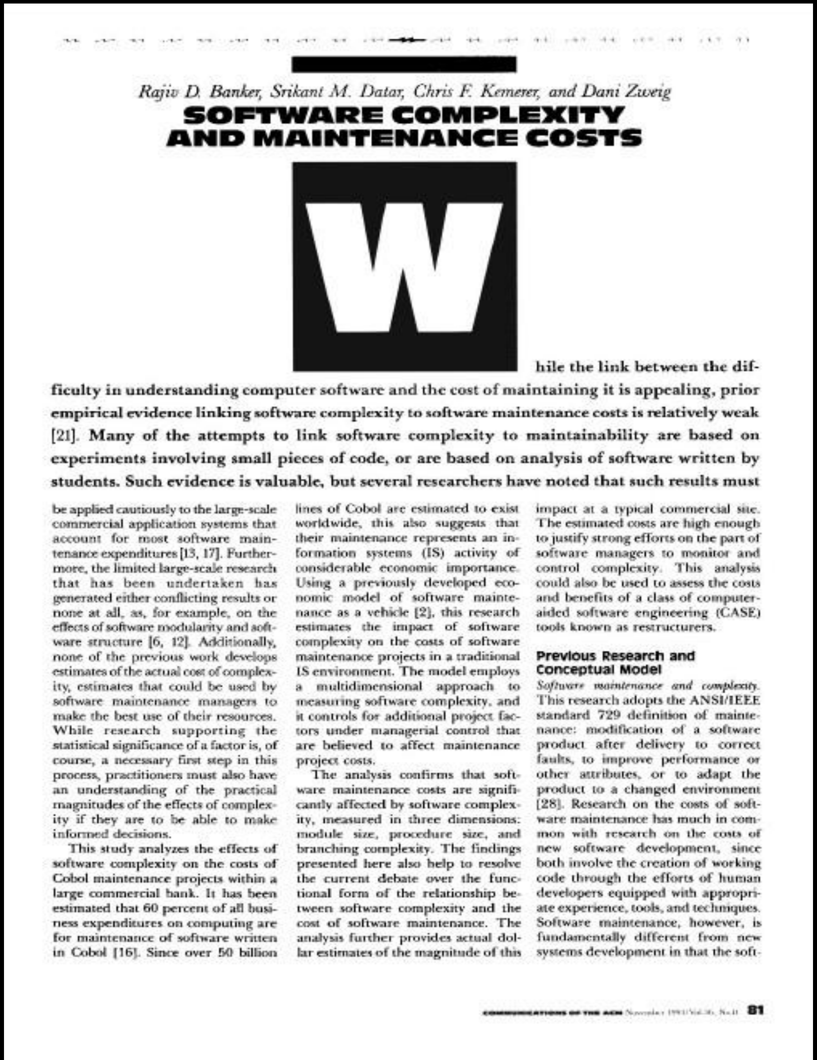
Springer

https://www.slideshare.net/fabiopalomba/icse15-smell-inducingchange

Negative Impact of Bad Smells

Bad Smells increase maintenance costs

[Banker et al. Communications of the ACM]



[redacted]

Rajiv D. Banker, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig

SOFTWARE COMPLEXITY AND MAINTENANCE COSTS

While the link between the difficulty in understanding computer software and the cost of maintaining it is appealing, prior empirical evidence linking software complexity to software maintenance costs is relatively weak [21]. Many of the attempts to link software complexity to maintainability are based on experiments involving small pieces of code, or are based on analysis of software written by students. Such evidence is valuable, but several researchers have noted that such results must be applied cautiously to the large-scale commercial application systems that account for most software maintenance expenditures [13, 17]. Furthermore, the limited large-scale research that has been undertaken has generated either conflicting results or none at all, as, for example, on the effects of software modularity and software structure [6, 12]. Additionally, none of the previous work develops estimates of the actual cost of complexity, estimates that could be used by software maintenance managers to make the best use of their resources. While research supporting the statistical significance of a factor is, of course, a necessary first step in this process, practitioners must also have an understanding of the practical magnitudes of the effects of complexity if they are to be able to make informed decisions.

This study analyzes the effects of software complexity on the costs of Cobol maintenance projects within a large commercial bank. It has been estimated that 60 percent of all business expenditures on computing are for maintenance of software written in Cobol [16]. Since over 50 billion lines of Cobol are estimated to exist worldwide, this also suggests that their maintenance represents an information systems (IS) activity of considerable economic importance. Using a previously developed economic model of software maintenance as a vehicle [2], this research estimates the impact of software complexity on the costs of software maintenance projects in a traditional IS environment. The model employs a multidimensional approach to measuring software complexity, and it controls for additional project factors under managerial control that are believed to affect maintenance project costs.

The analysis confirms that software maintenance costs are significantly affected by software complexity, measured in three dimensions: module size, procedure size, and branching complexity. The findings presented here also help to resolve the current debate over the functional form of the relationship between software complexity and the cost of software maintenance. The analysis further provides actual dollar estimates of the magnitude of this impact at a typical commercial site. The estimated costs are high enough to justify strong efforts on the part of software managers to monitor and control complexity. This analysis could also be used to assess the costs and benefits of a class of computer-aided software engineering (CASE) tools known as restructurers.

Previous Research and Conceptual Model

Software maintenance and complexity. This research adopts the ANSI/IEEE standard 729 definition of maintenance: modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment [28]. Research on the costs of software maintenance has much in common with research on the costs of new software development, since both involve the creation of working code through the efforts of human developers equipped with appropriate experience, tools, and techniques. Software maintenance, however, is fundamentally different from new systems development in that the soft-

https://www.slideshare.net/fabiopalomba/icse15-smell-inducingchange