SOEN6461: Software Design Methodologies

Zeinab (Azadeh) Kermansaravi

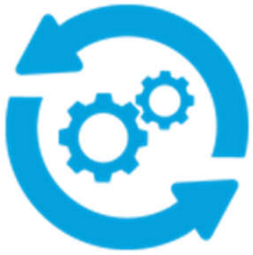
Smells: Anti-patterns and Code smells

Contributions: Yann-Gaël Guéhéneuc, Foutse Khomh, , Diana El-Masri, Fàbio Petrillo, Zéphryin Soh and Naouel Moha

Concordia University
GINA CODY SCHOOL OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science & Software Engineering

Development Team

Evolution



Code Smells

Design Smells

≠

Design patterns

Quality

Lexical Smells,

…

≠

Name conventions

“**qual·i·ty** *noun* \ˈkwä-lə-tē\

- how good or bad something is
- a characteristic or feature that someone or something has: something that can be noticed as a part of a person or thing
- a high level of value or excellence”

—Merriam-Webster, 2013

Software Quality

In the context of software engineering, software quality measures how well software is designed (quality of design), and how well the software conforms to that design (quality of conformance).

Software Quality

- Division of software quality according to ISO/IEC 9126:2001, 25000:2005…
  - Process quality
  - Product quality
  - Quality in use

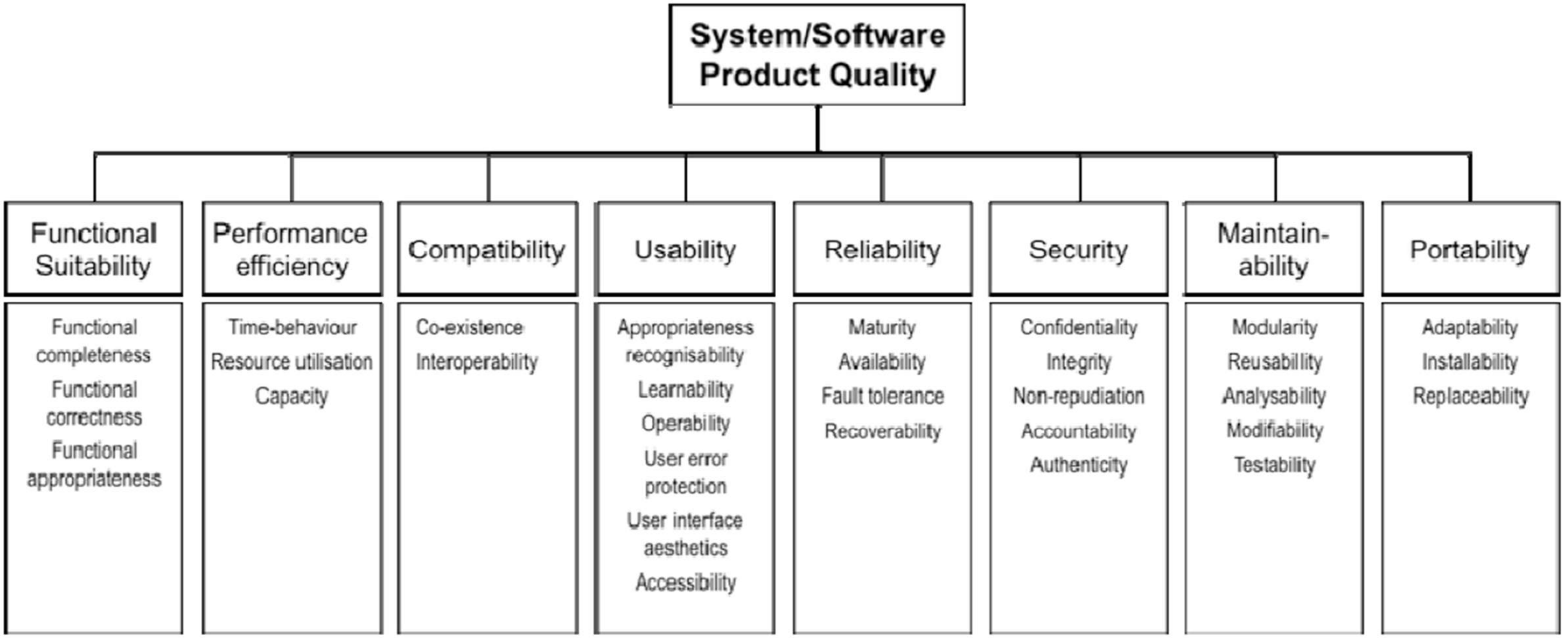
http://www.sqa.net/iso9126.html

Software Quality

- Division of software quality according to ISO/IEC 9126:2001, 25000:2005…
  - Process quality
  - **Product quality**
  - Quality in use

http://www.sqa.net/iso9126.html

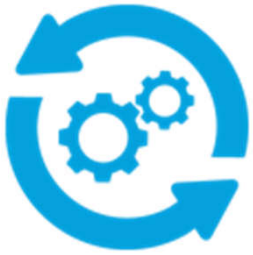
Software Quality



http://romainvv.ddns.net/iso-25010/

Development Team

Evolution



Code Smells

Design Smells

≠

Design patterns

Quality

Lexical Smells,

…

≠

Name conventions

Smells

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)

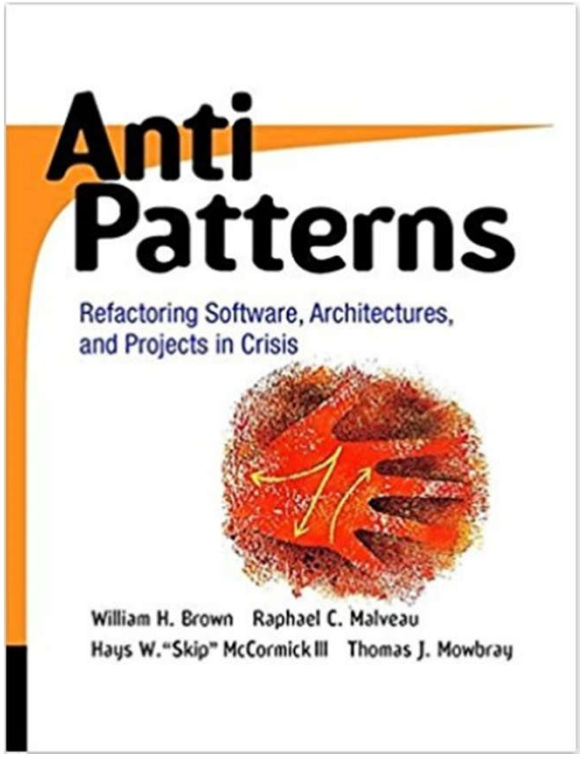
✓ Poor practice in the naming, documentation, … in the implementation of an entity.

- Anti-patterns (High Level (global) problems)

✓ Poor design solutions to recurring design problems

Anti-patterns

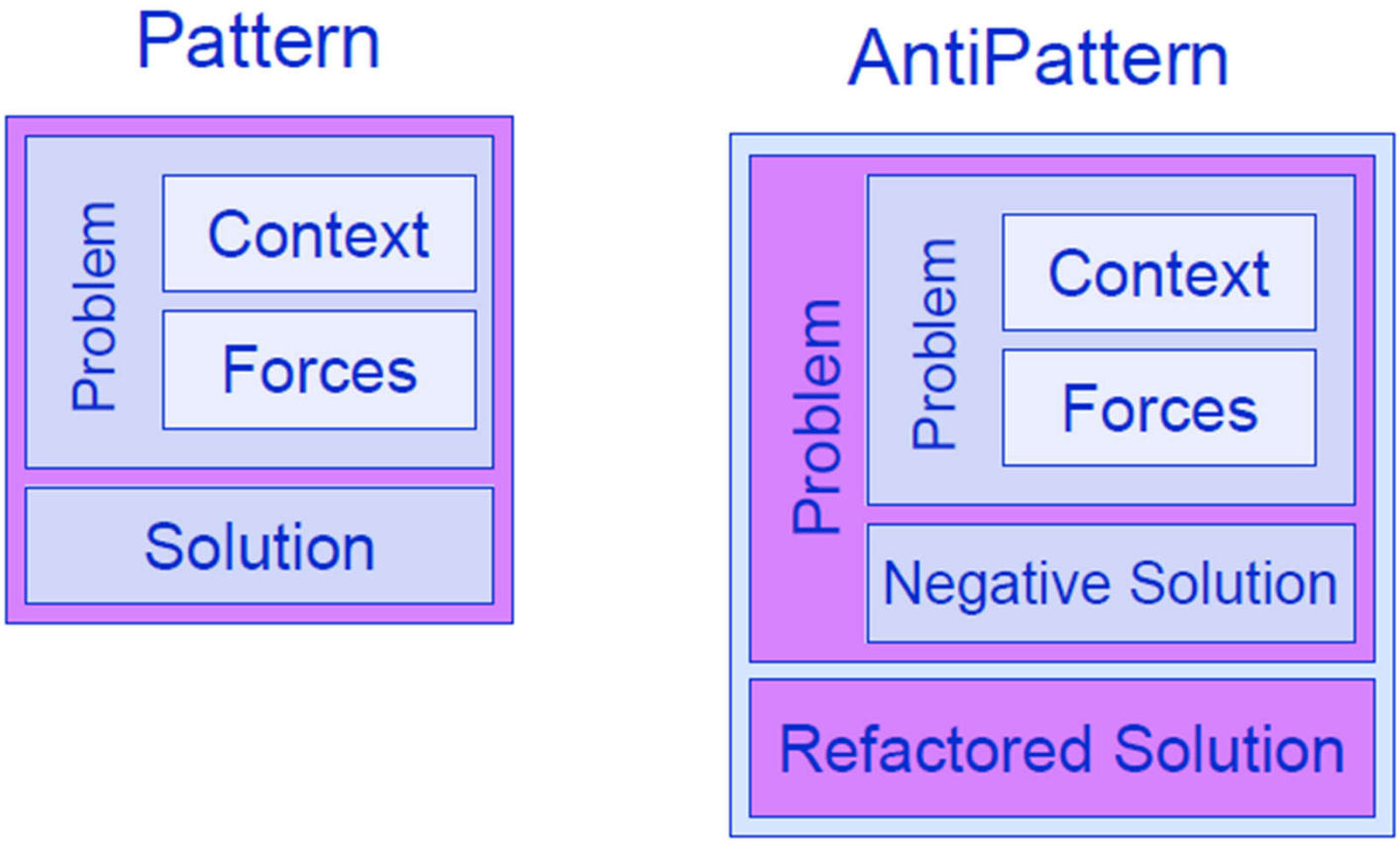
- **Anti-patterns** are “poor” solutions to recurring design and implementation problems.
- Impact program comprehension, software evolution and maintenance activities.
- Important to detect them early in software development process, to reduce the maintenance costs.



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.



Anti-patterns Classifications

- **Software Development Anti-Patterns**
- Software Architecture Anti-Patterns
- Software Project Management Anti-Patterns

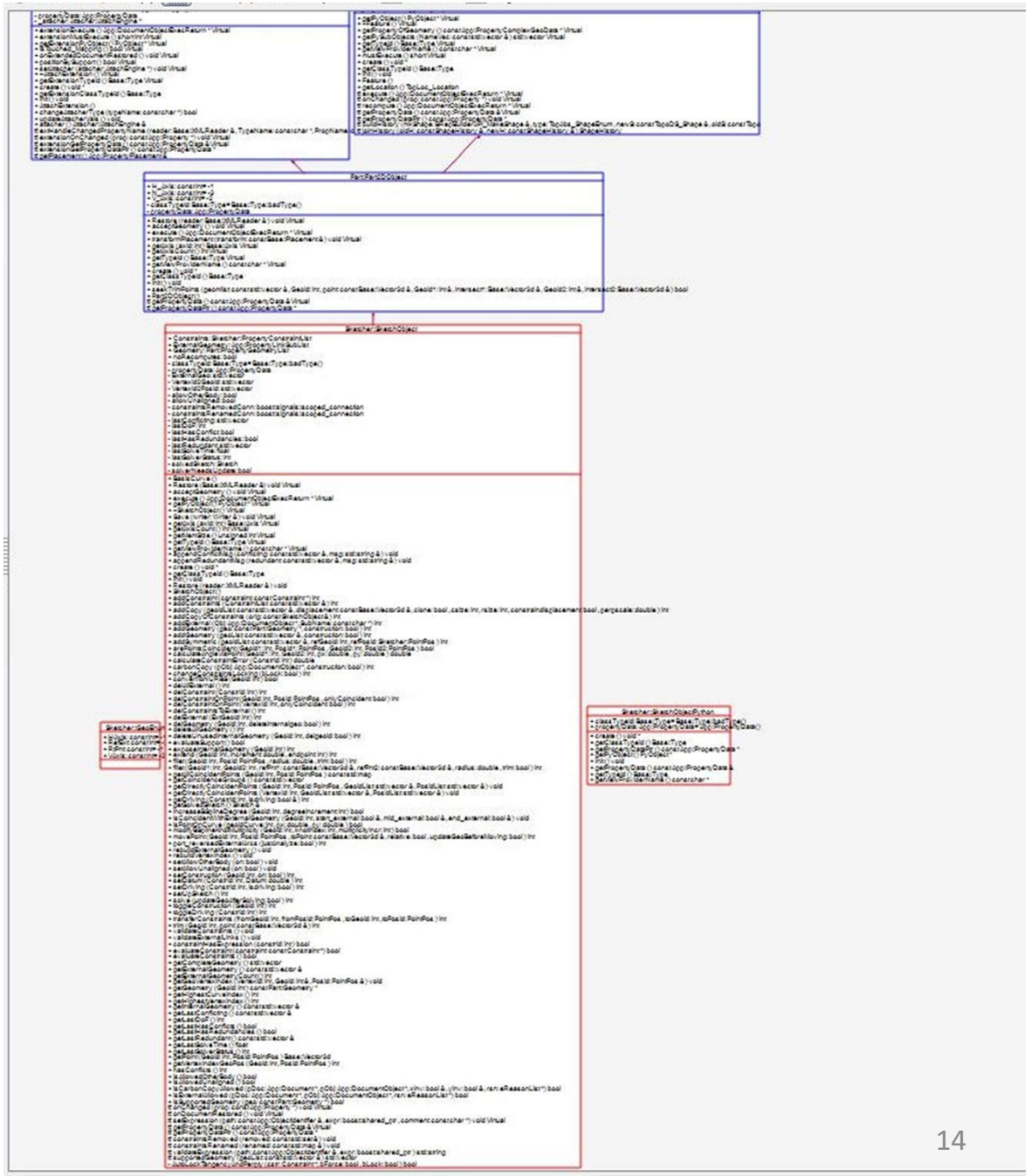
Anti-patterns Classifications

- Software Development Anti-Patterns

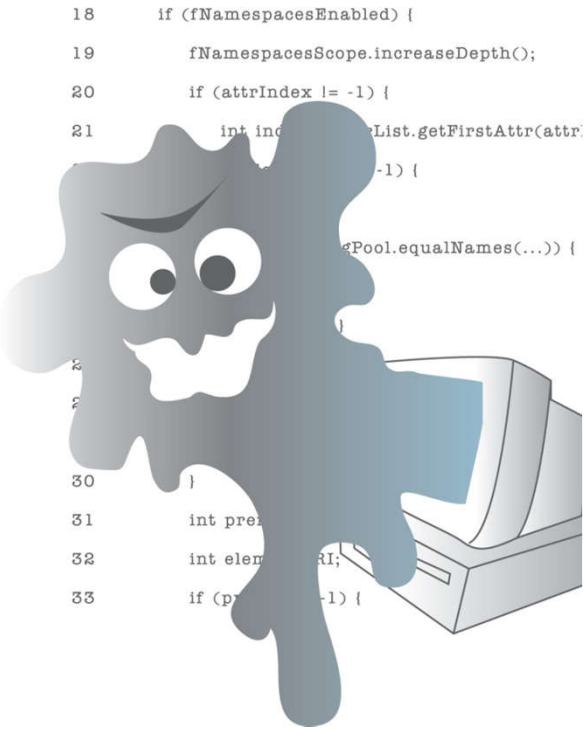
- **The Blob**
- Continuous Obsolescence
- Lava Flow
- Ambiguous Viewpoint
- Functional Decomposition
- Poltergeists
- Boat Anchor
- Golden Hammer
- Dead End
- **Spaghetti Code**
- Input Kludge
- Walking through a Minefield
- Cut–and–Paste Programming
- Mushroom Management

Blob (God Class)

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

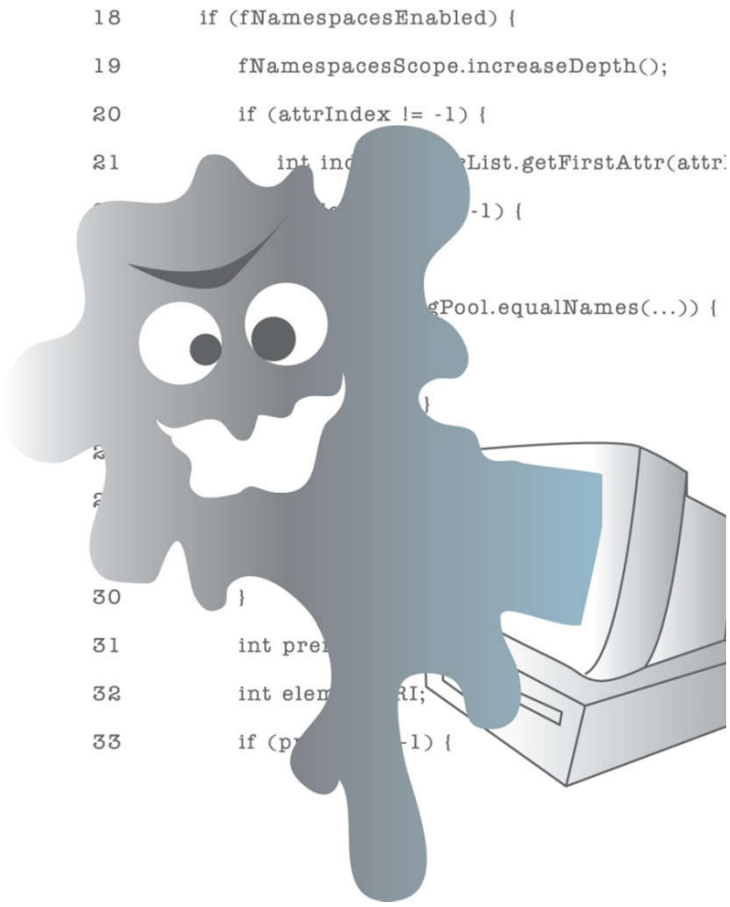


- Blob (God Class)
  - Symptoms:
    - Large controller class
    - Many fields and methods with a low cohesion*
    - Lack of OO design.
    - Procedural-style than object oriented architectures.



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

- Blob (God Class)
  - Consequences:
    - Lost OO advantage
    - Too complex to reuse or test.
    - Expensive to load

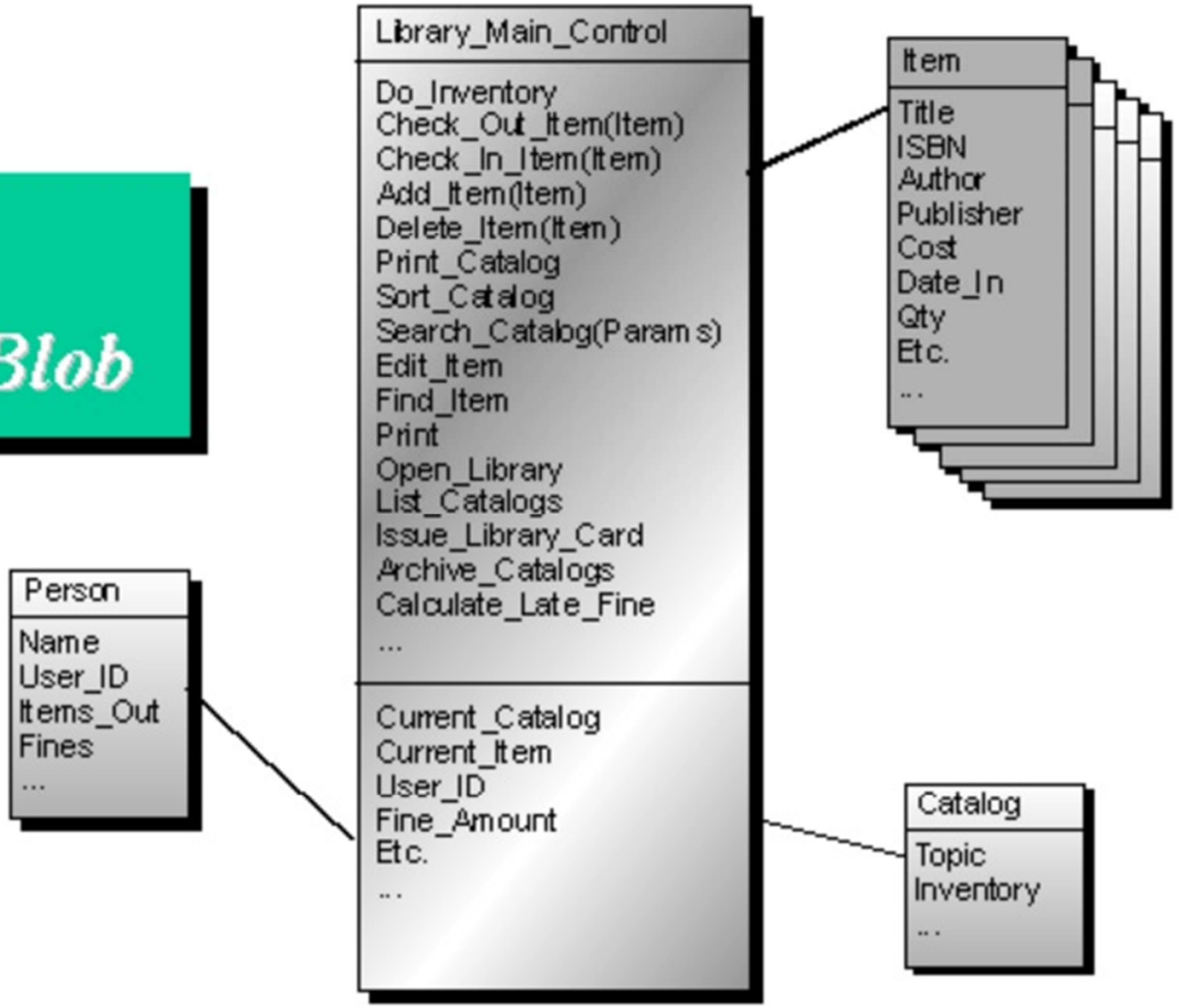


https://www.slideshare.net/Ptidej/cascon06-tooldemoppt

Blob (God Class)

Example:

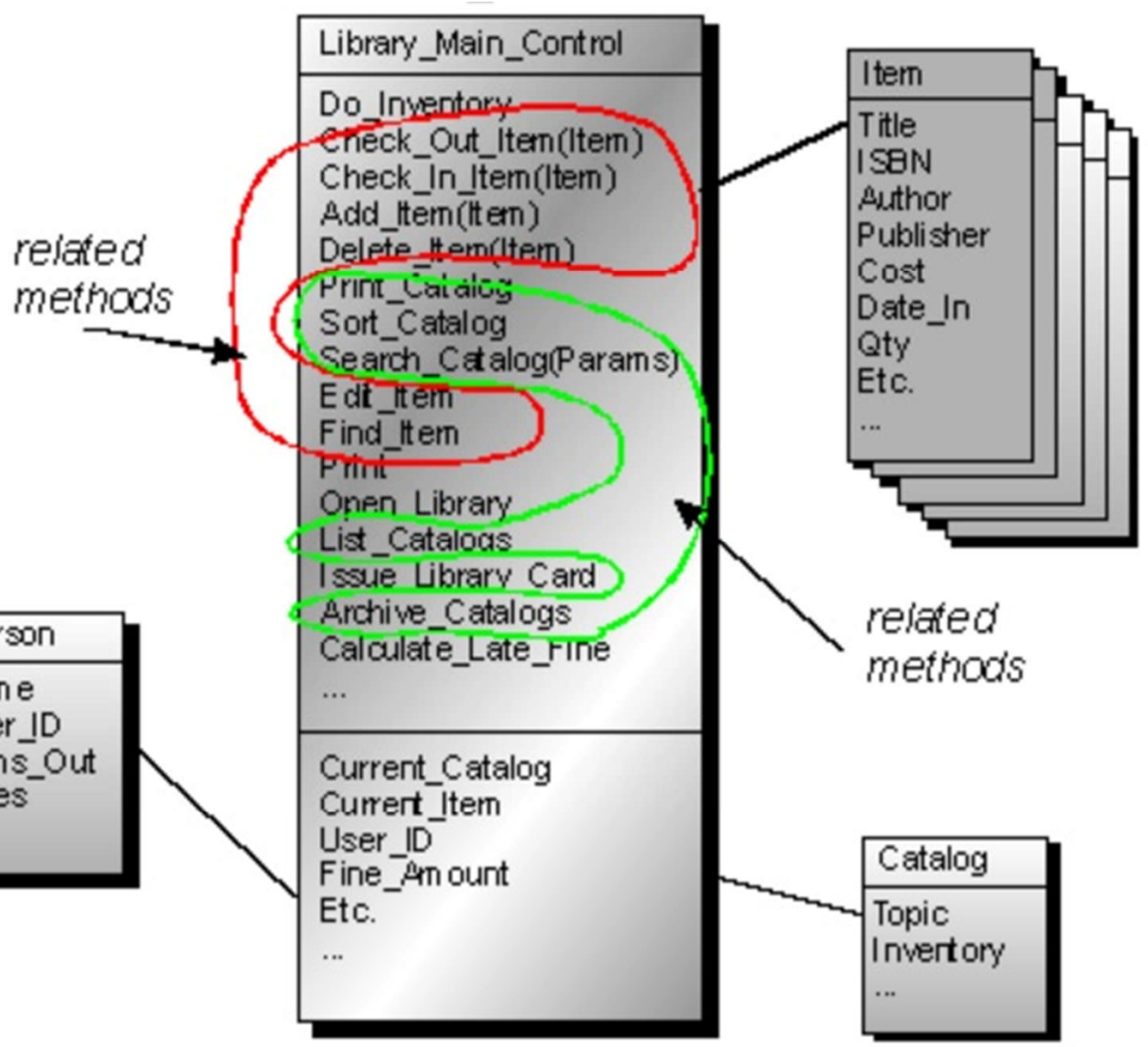
The Library Blob



http://antipatterns.com/briefing/sld025.htm

Refactoring…

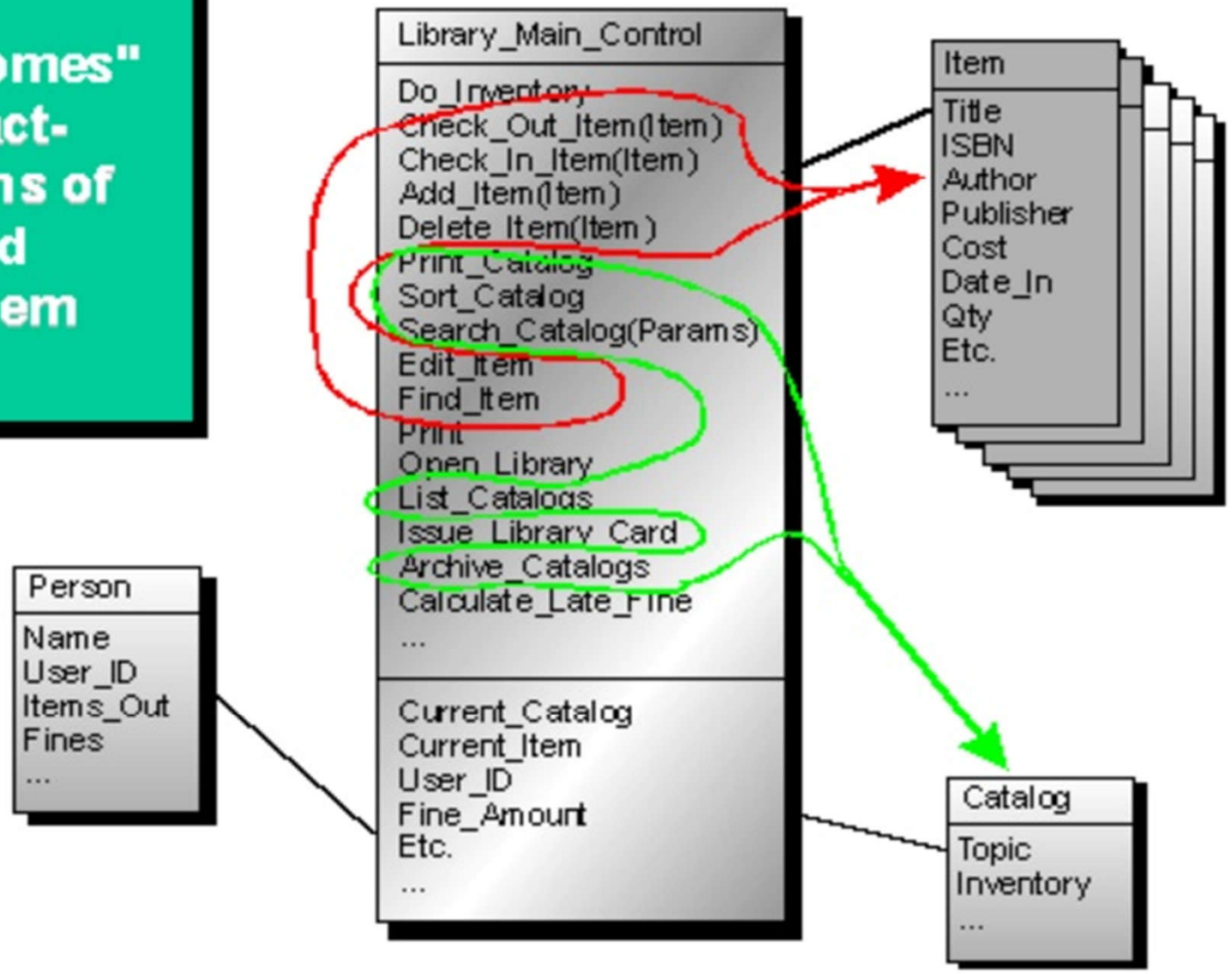
Step 1:
Identify or categorize related attributes and operations according to contracts.



http://antipatterns.com/briefing/sld026.htm

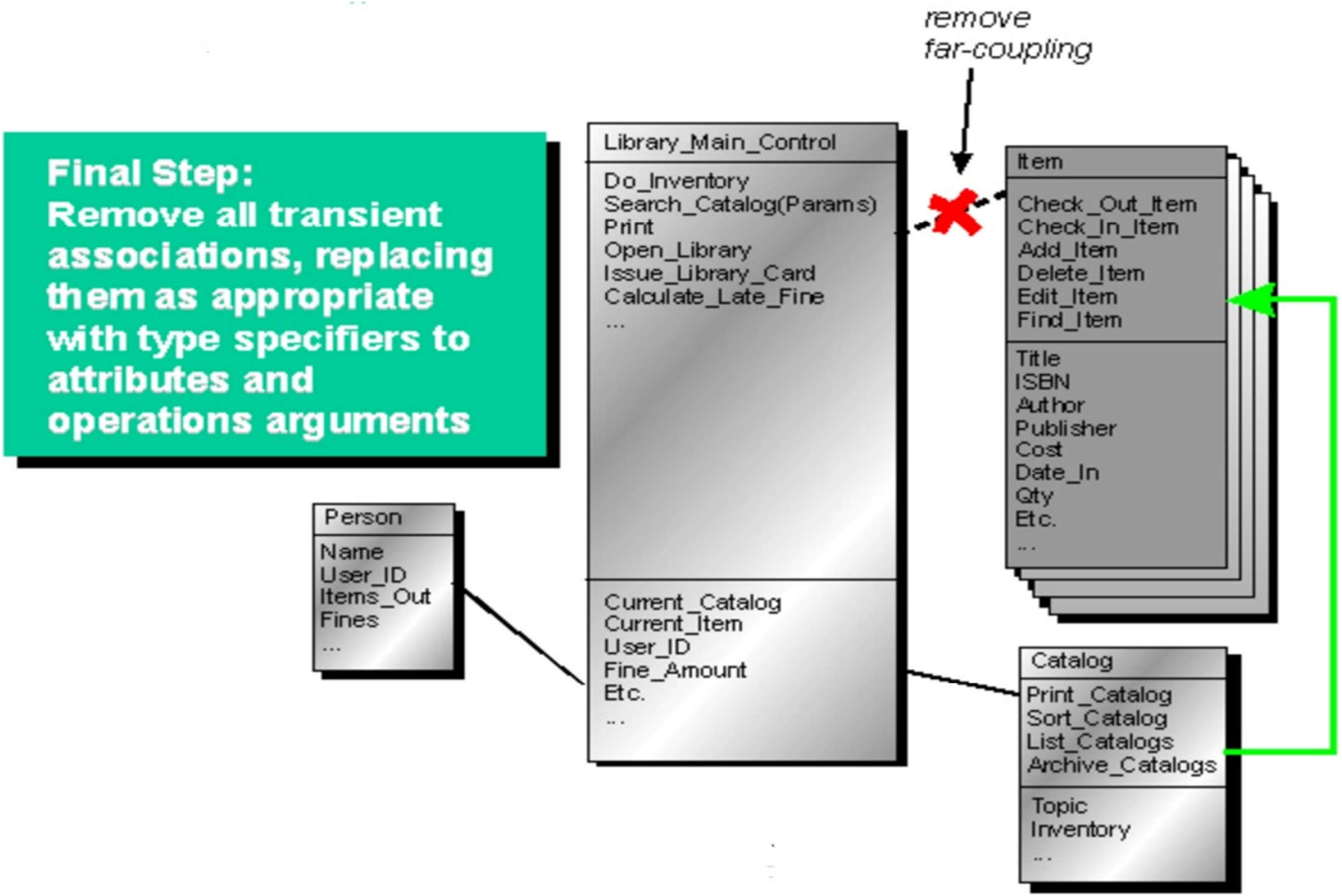
Refactoring…

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



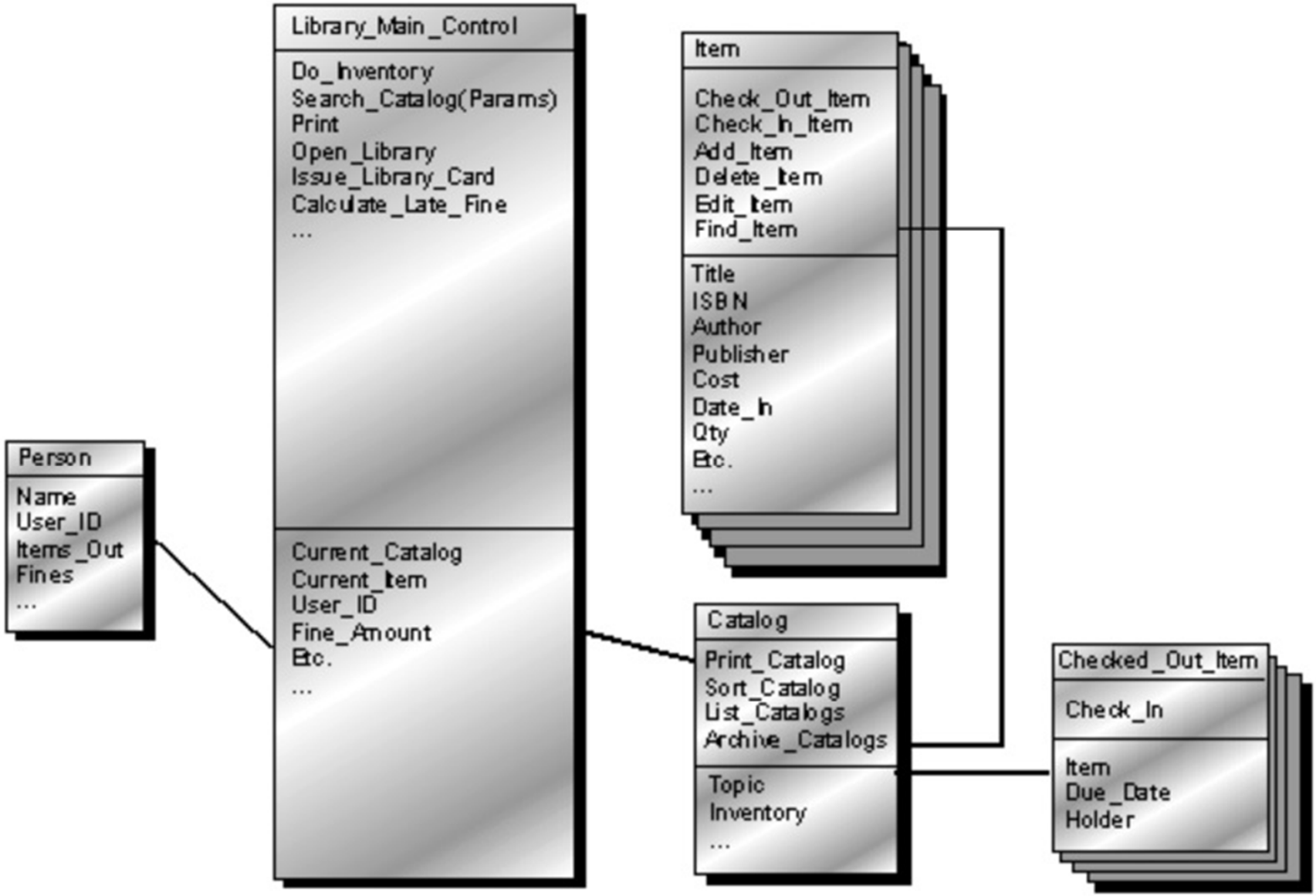
http://antipatterns.com/briefing/sld027.htm

Refactoring…



http://antipatterns.com/briefing/sld028.htm

Refactored



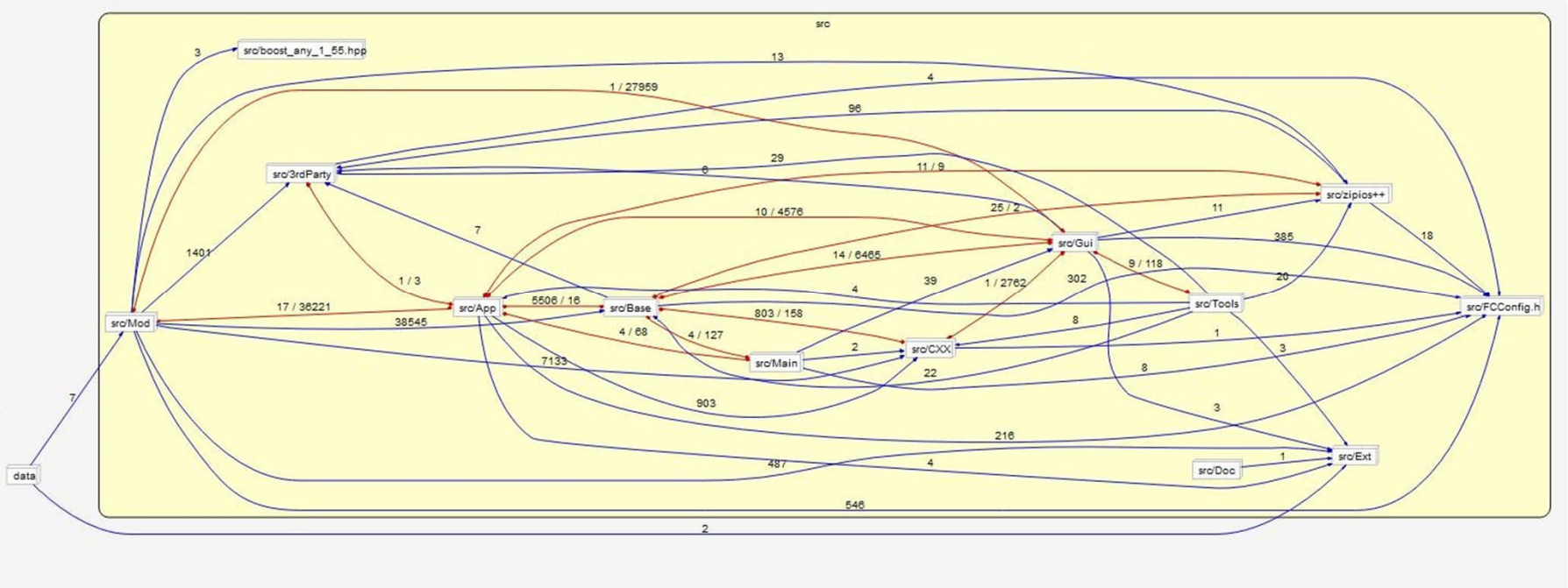
http://antipatterns.com/briefing/sld029.htm

Spaghetti code

- Ring project
- 233,492 lines of code

Spaghetti code

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

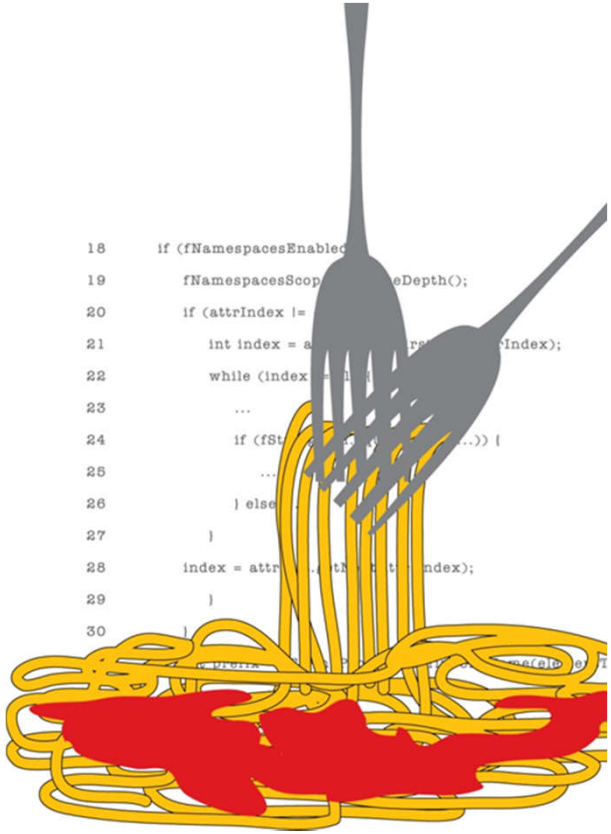


Spaghetti code

In spaghetti code, the relations between the pieces of code are so tangled that it is nearly impossible to add or change something without unpredictably breaking something somewhere else.

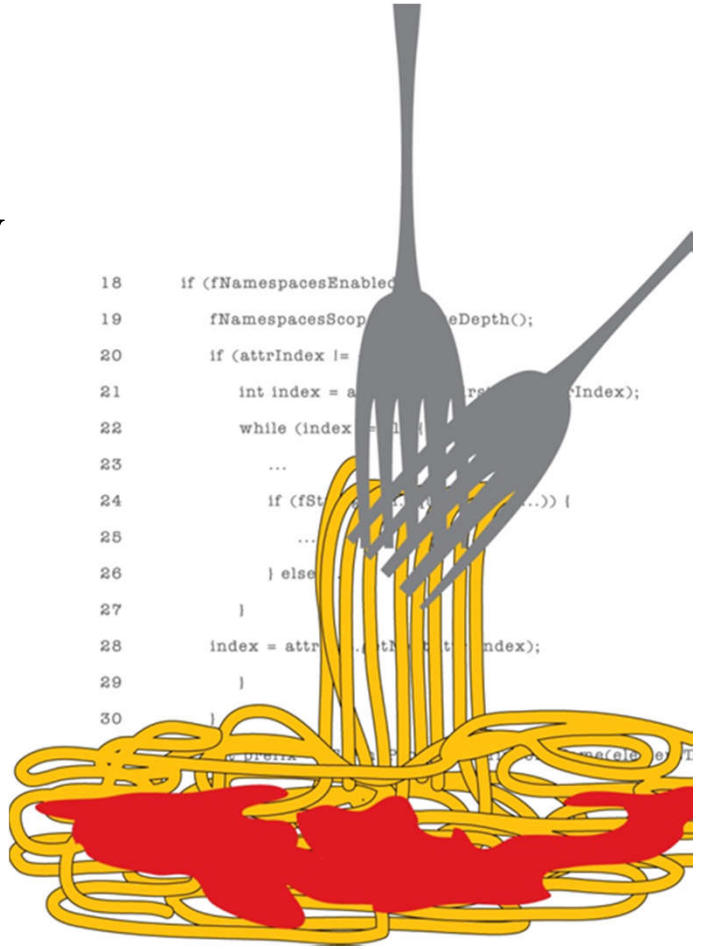
http://joostdevblog.blogspot.ca/2012/05/programming-with-pasta.html

- 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



https://www.slideshare.net/Ptidej/cascon06-tooldemoppt

- 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.



Spaghetti Code

```
function _menu_site_is_offline($check_only = FALSE) {
  // Check if site is in maintenance mode.
  if (variable_get('maintenance_mode', 0)) {
    if (user_access('access site in maintenance mode')) {
      // Ensure that the maintenance mode message is displaye
      // (allowing for page redirects) and specifically suppr
      // the maintenance mode settings page.
      if (!$check_only && $_GET['q'] != 'admin/config/develop
        if (user_access('administer site configuration')) {
          drupal_set_message(t('Operating in maintenance mode
        }
        else {
          drupal_set_message(t('Operating in maintenance mode
        }
      }
    }
    else {
      return TRUE;
    }
  }
```

https://www.slideshare.net/ygerasimov/clean-code-and-refactoring

Spaghetti Code

Refactoring…

```
function _menu_site_is_offline($check_only = FALSE) {
  // Check if site is in maintenance mode.
  if (!variable_get('maintenance_mode', 0)) {
    return FALSE;
  }
  if (!user_access('access site in maintenance mode')) {
    return TRUE;
  }
  // Ensure that the maintenance mode message is displayed
  // (allowing for page redirects) and specifically suppre
  // the maintenance mode settings page.
  if (!$check_only && $_GET['q'] != 'admin/config/developm
    if (user_access('administer site configuration')) {
      drupal_set_message(t('Operating in maintenance mode.
    }
    else {
      drupal_set_message(t('Operating in maintenance mode.
    }
  }
  return FALSE;
}
```

Code Smells

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

Payoff

- ✓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?

Function: Simple 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?

A simple 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

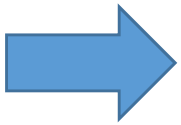
Another Example

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

Is this a good practice?

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 logic to handle the elements is written once for all

DO NOT EVER COPY AND PASTE CODE

GOAL

Readable, maintainable and extendable code

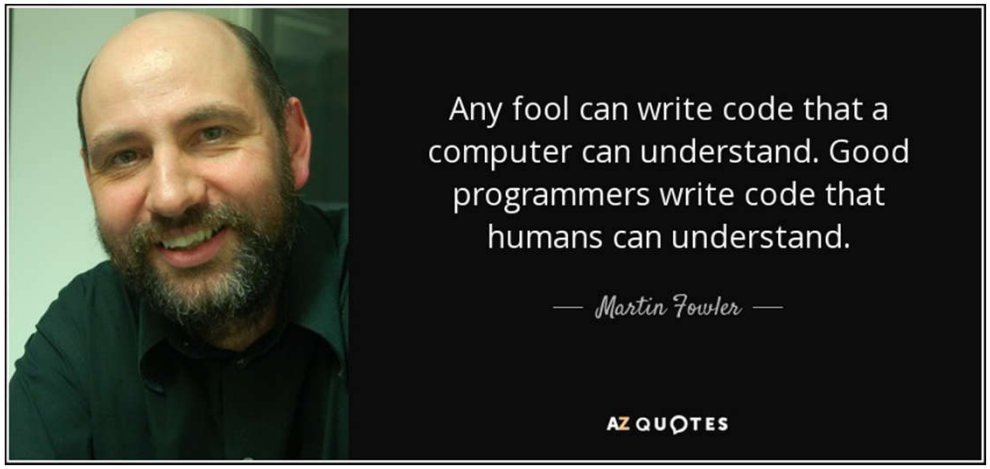
Code Smells

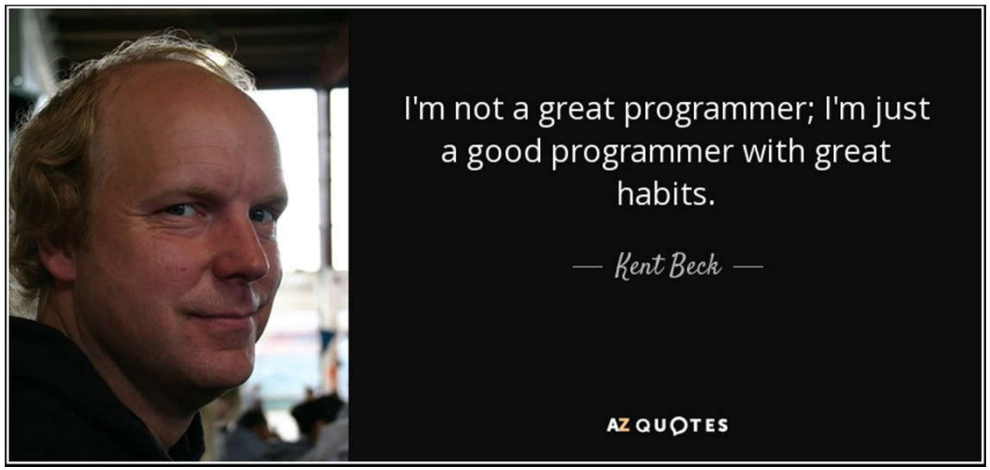
- A **code smell** is a flag that something has gone wrong somewhere in your code. Use the smell to track down the problem.

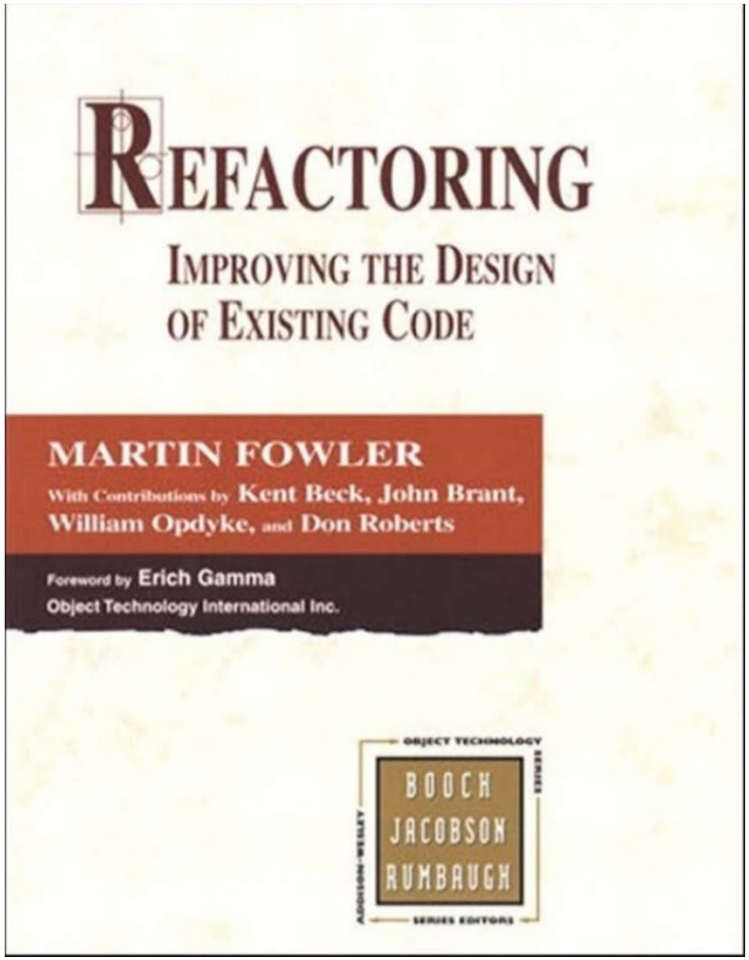
➢ **They are symptoms of poor design or implementation choices [Martin Fowler]**

http://wiki.c2.com/?CodeSmell

Code Smells


Any fool can write code that a computer can understand. Good programmers write code that humans can understand.
— Martin Fowler —
AZ QUOTES


I'm not a great programmer; I'm just a good programmer with great habits.
— Kent Beck —
AZ QUOTES


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
ADDISON-WESLEY
SERIES EDITORS

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;
    }
}
```

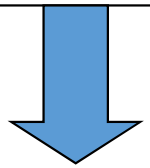
```
public Class Car{
    private Color color;

    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)
{       }
```

```
 *            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);
    }
}
```

Problem:

Hard to understand , change, and reuse

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);
    }
}
```

https://sourcemaking.com/refactoring/smells/long-parameter-list

Payoff

- more flexible thanks to use of objects instead of primitives.
- 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.

Object-Orientation Abusers

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

- Switch Statements → **Should use Polymorphism**
- Alternative Classes with Different Interfaces → **Poor class hierarchy**
- Refused Bequest → **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

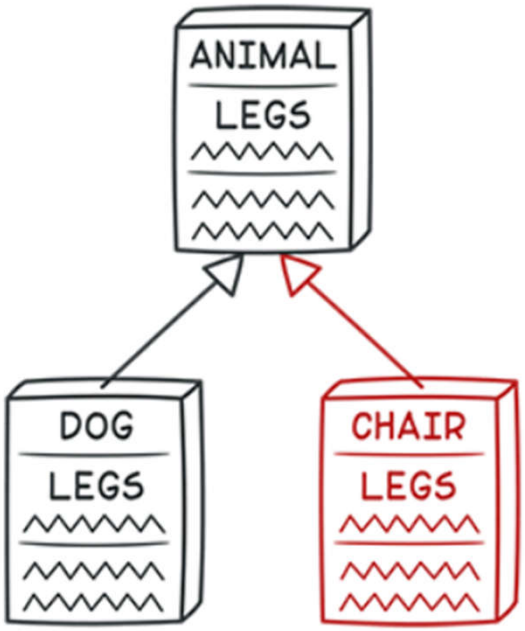
Subclass doesn't use superclass methods and attributes

```
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();
    }
}
```

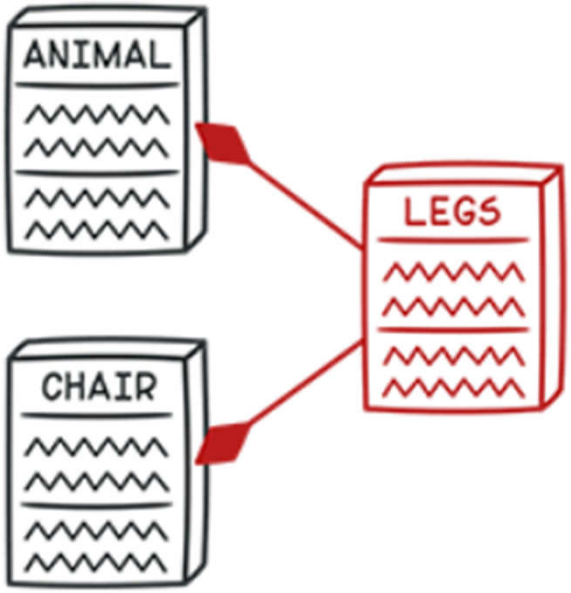
Engineer does not use quota. It should be pushed down to Salesman

Refused Bequest

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



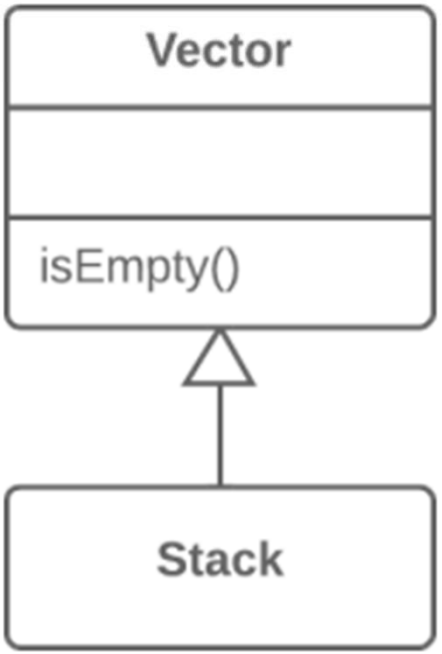
Delegation (has a…)



Refused Bequest

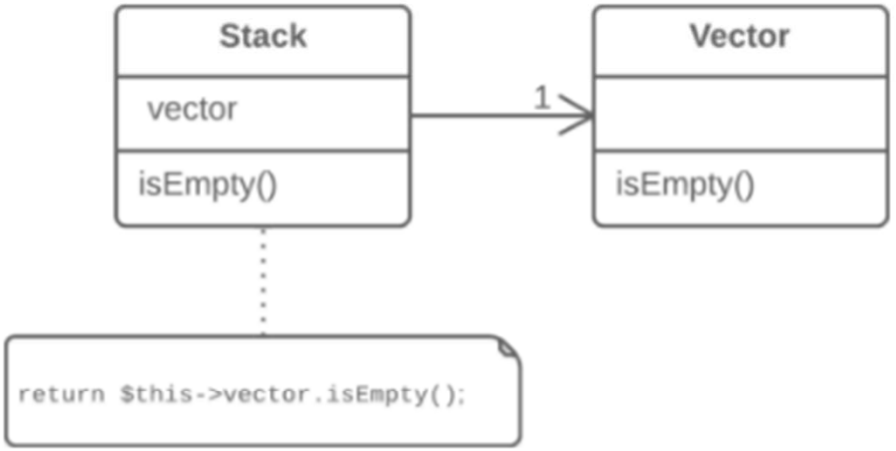
Problem

You have a subclass that uses only a portion of the methods of its superclass (or it's not possible to inherit superclass data).



Solution

Create a field and put a superclass object in it, delegate methods to the superclass object, and get rid of inheritance.



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

→ That isn't useful

- Dead Code
- Speculative Generality → Predicting the future
- Lazy class → 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;
```

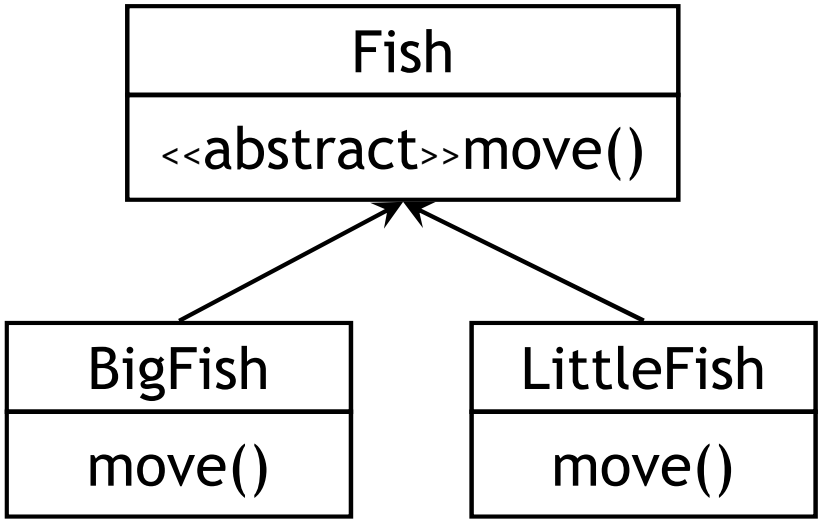
Refactor: Extract method

```
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 different classes

- 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:

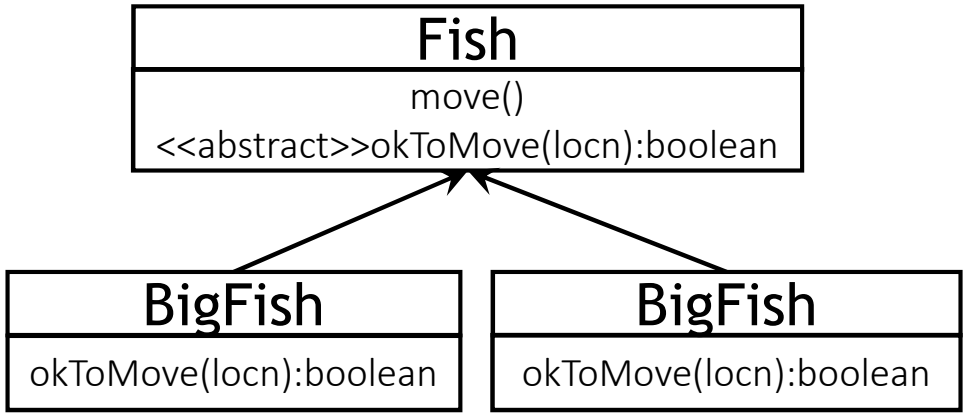
```
public void move() {
    choose a random direction;          // same for both
    find the location in that direction; // same for both
    check if it's ok to move there;      // different
    if it's ok, make the move;           // same for both
}
```

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



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.

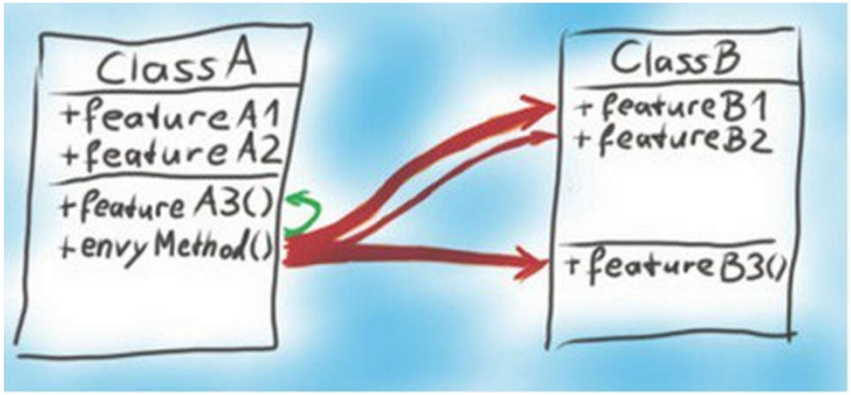
- Feature Envy → Misplaced responsibility
- Inappropriate Intimacy → Classes should know as little as possible about each other (↓ Cohesion)
- Middle Man → Too complex data access
- Message Chains → Too complex data access

https://sourcemaking.com/refactoring/smells

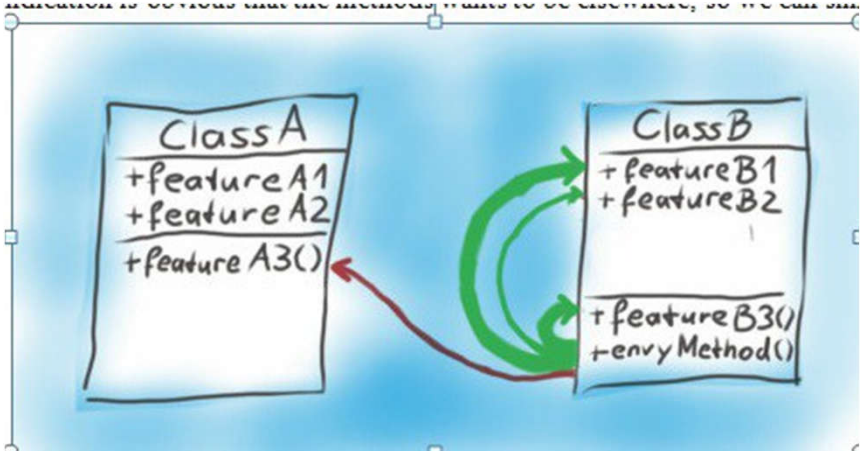
Feature Envy

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);
    }
 }
}
```

- *Refactor*ing 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()
```

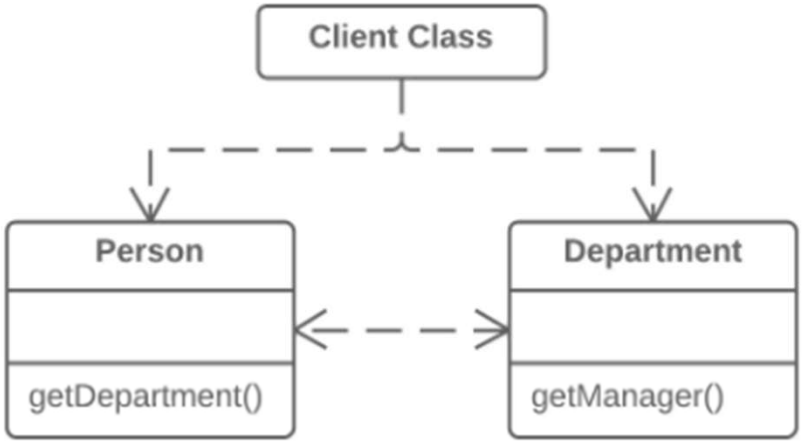
Law of Demeter: Each unit should only talk with friends

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

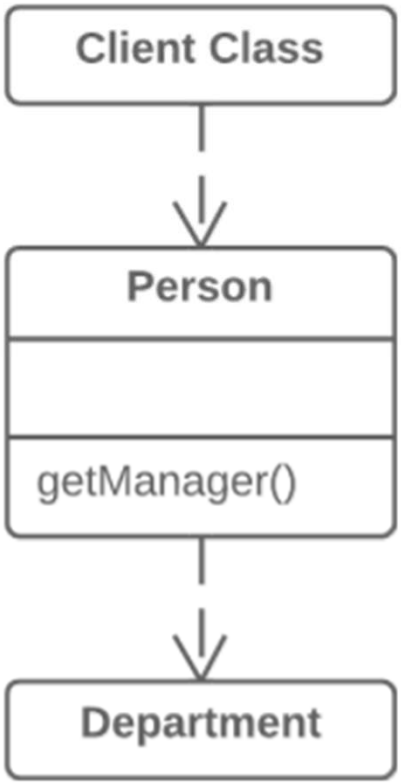
Message chains

- To refactor a message chain, use Hide Delegate.

Message chains



Refactor: Hide delegate



https://sourcemaking.com/refactoring/smells

Change preventers

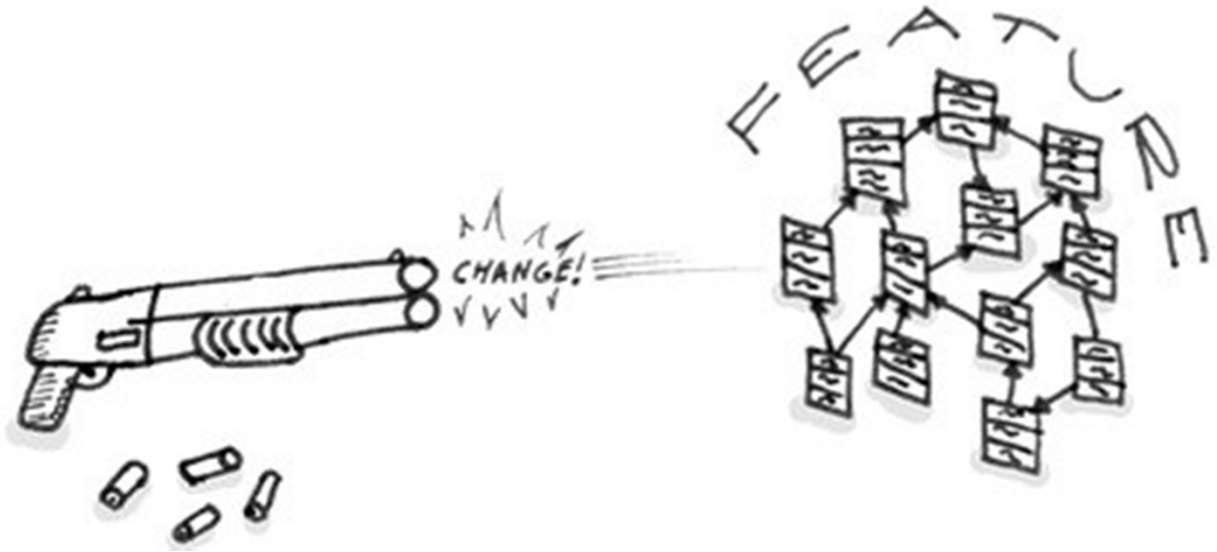
if you need to change something in one place in your code, you have to make many changes in other places too.
Program development becomes much more complicated and expensive as a result.

- 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



```java
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;
	}
```

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

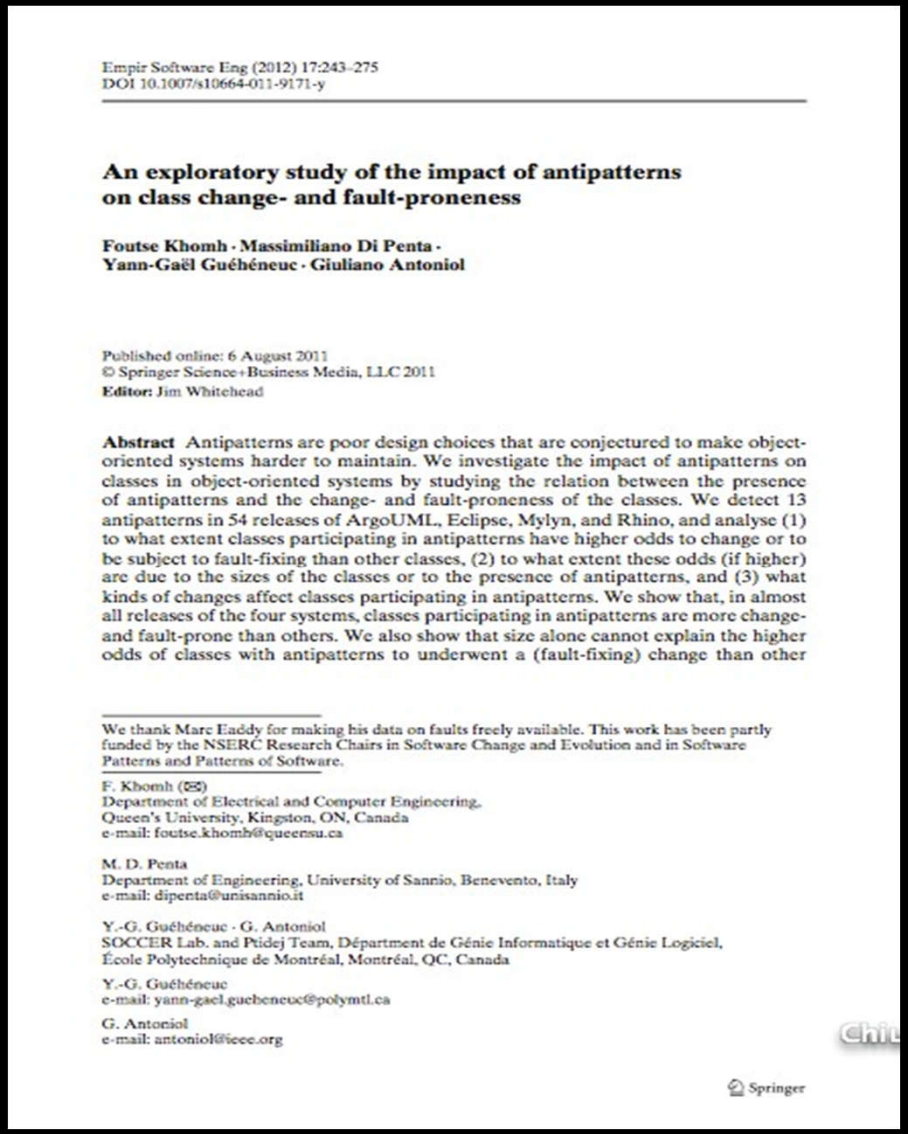
1534-5351/11 $26.00 © 2011 IEEE
DOI 10.1109/CSMR.2011.24

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

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, Département 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

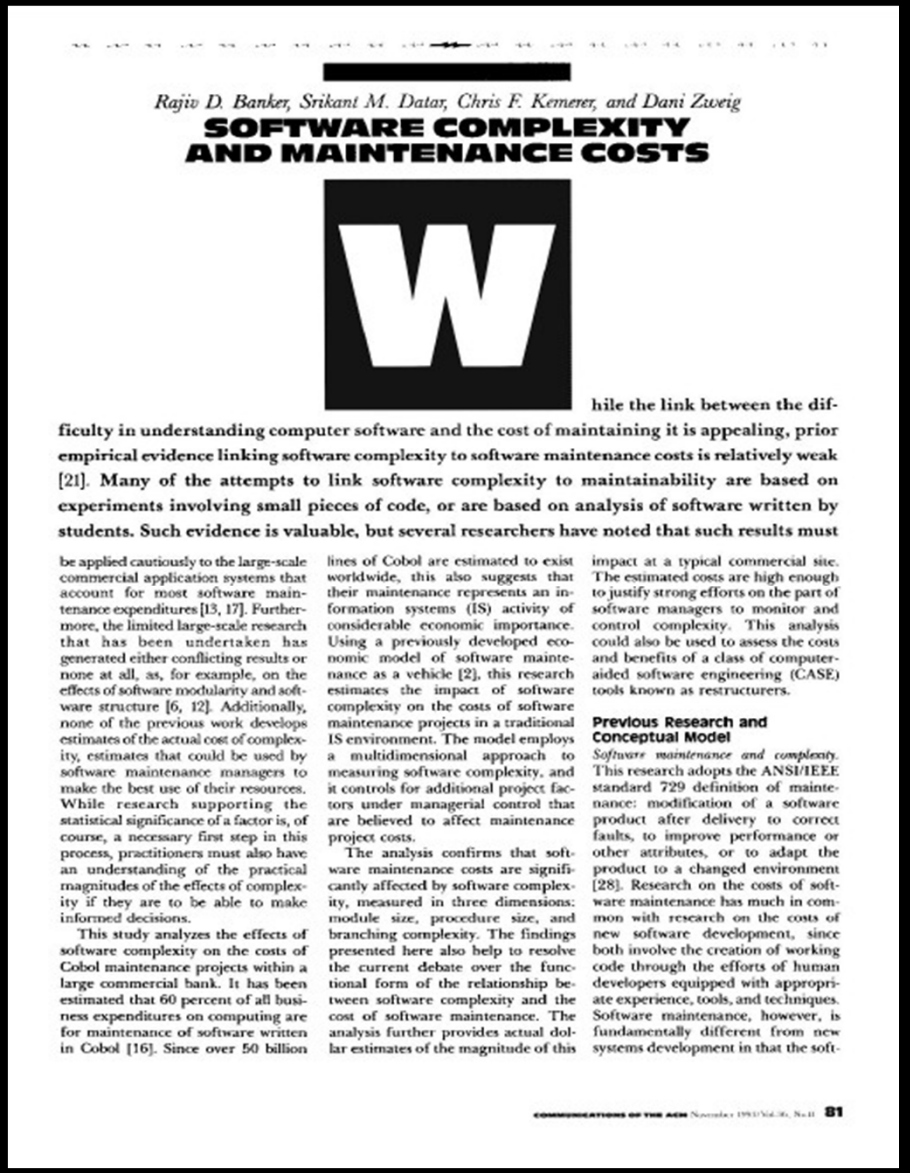
Springer

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]



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.

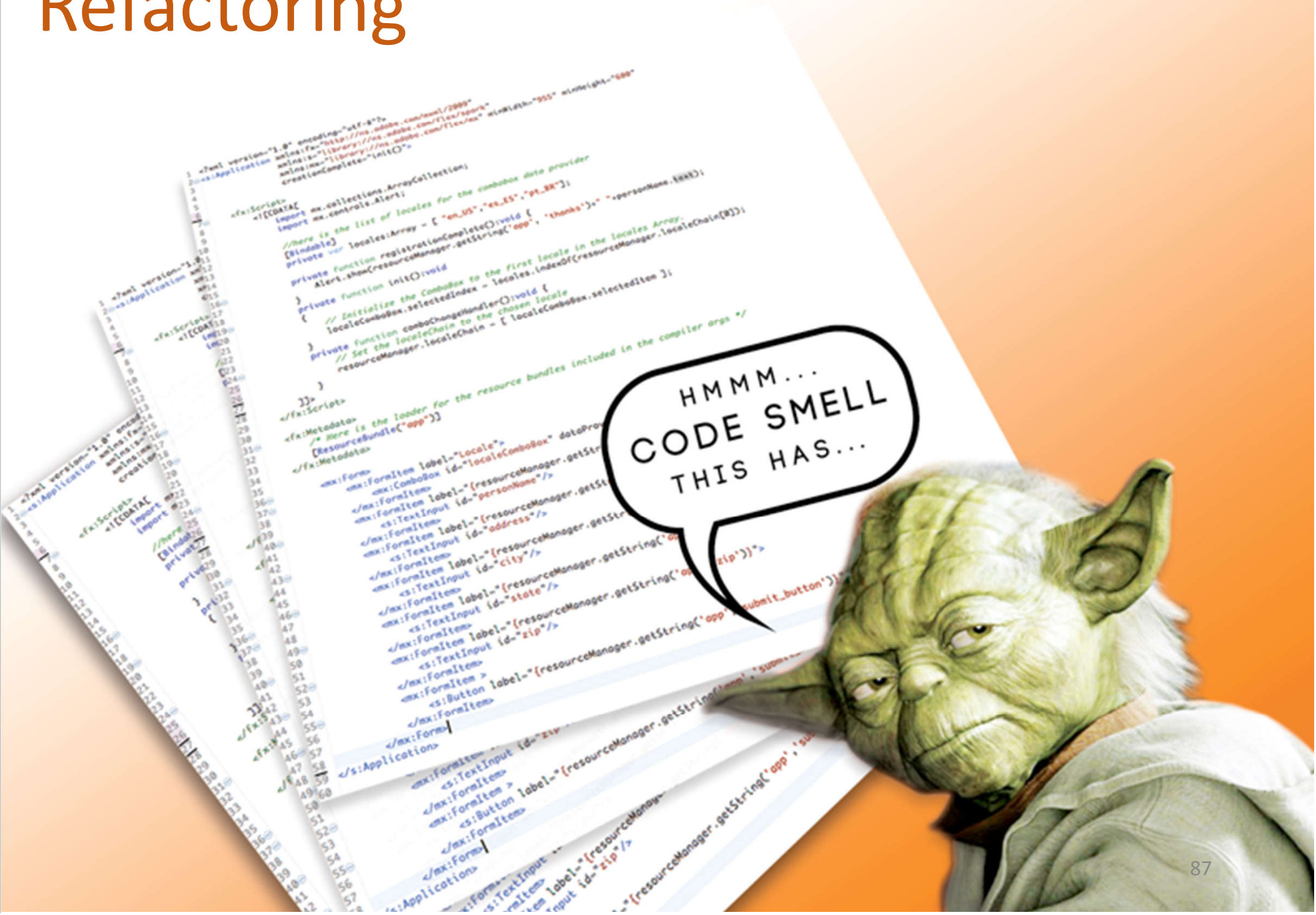
The analysis confirms that software maintenance costs are significantly affected by software complexity, measured in three dimensions: module size, procedure size, and branching complexity. The findings presented here also help to resolve the current debate over the functional form of the relationship 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

Refactoring



Bad Code Smells: Indications

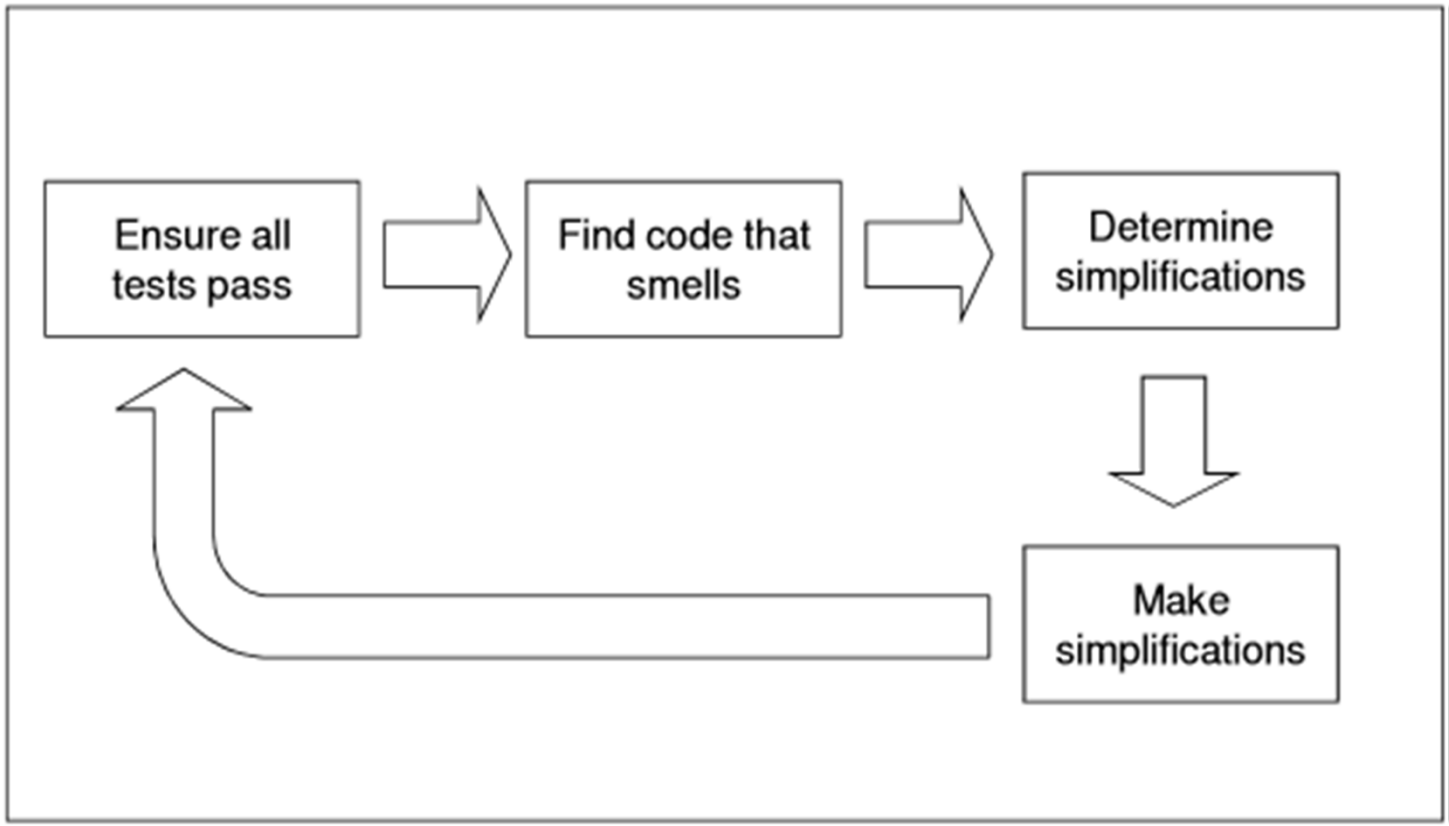
- Frequent failures
- Overly complex structure
- Very large components
- Excessive resource requirements
- Deficient documentation
- High personnel turnover
- Different technologies in one system

http://slideplayer.com/slide/7833453/ slides 88 to 90

Bad Code Smells: Solution

- Steps:
  - Program Comprehension (Understanding)
  - Refactoring
    - A set of transformations that are guaranteed to preserve the behavior while they can remove bad code smells
  - Refining

Bad Code Smells: Refactoring



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

Linguistic Anti-Patterns (LA)

Empir Software Eng (2016) 21:104–158
DOI 10.1007/s10664-014-9350-8

Linguistic antipatterns: what they are and how developers perceive them

Venera Arnaoudova · Massimiliano Di Penta · Giuliano Antoniol

Published online: 29 January 2015

Linguistic Anti-Patterns (LA)

- Represent recurring, poor naming and commenting choices
- Negatively affects software:
  - Understandability
  - Maintainability
  - Quality

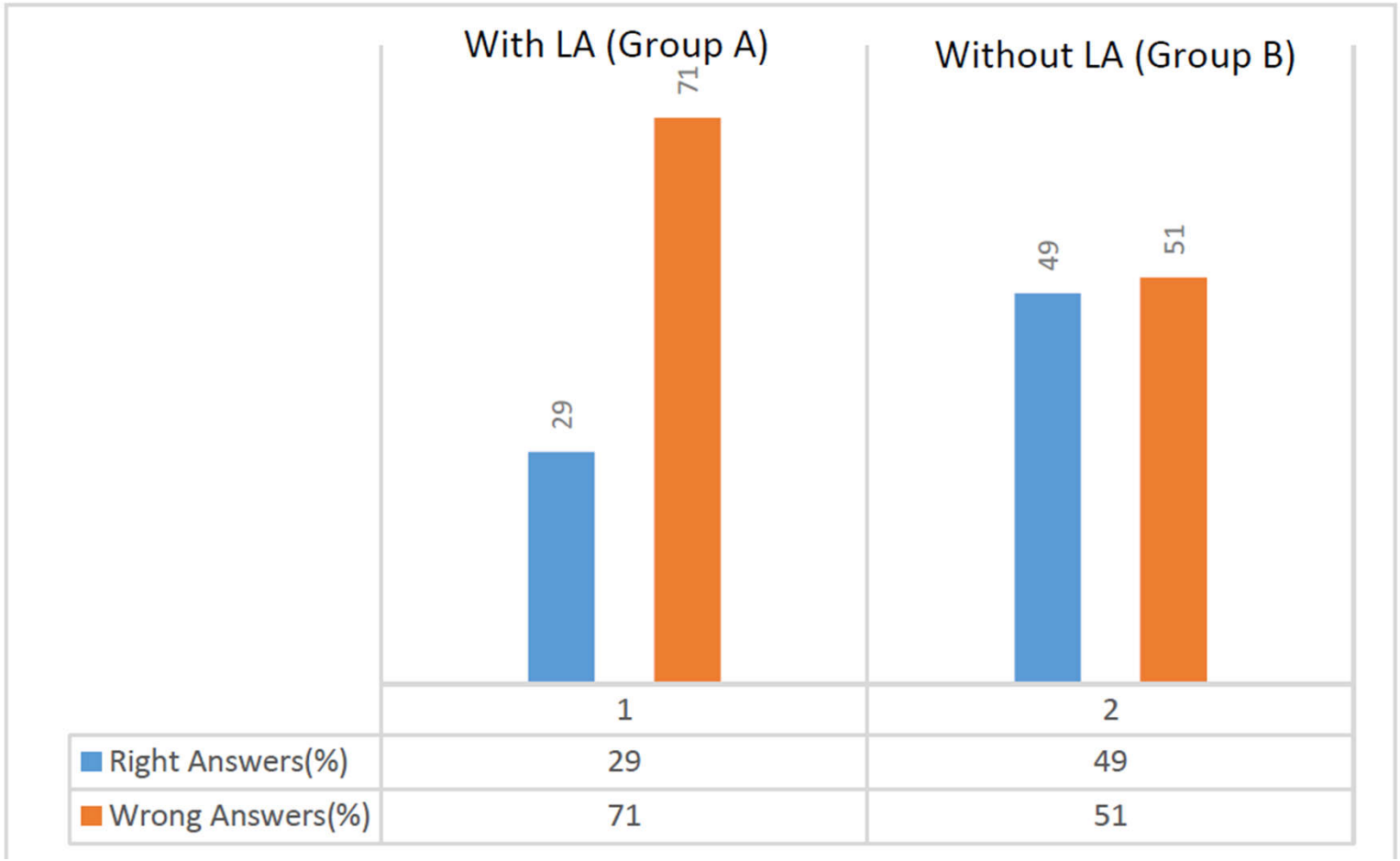
➢ Developers will

  ➢ Spend more time and effort when understanding these software artifacts

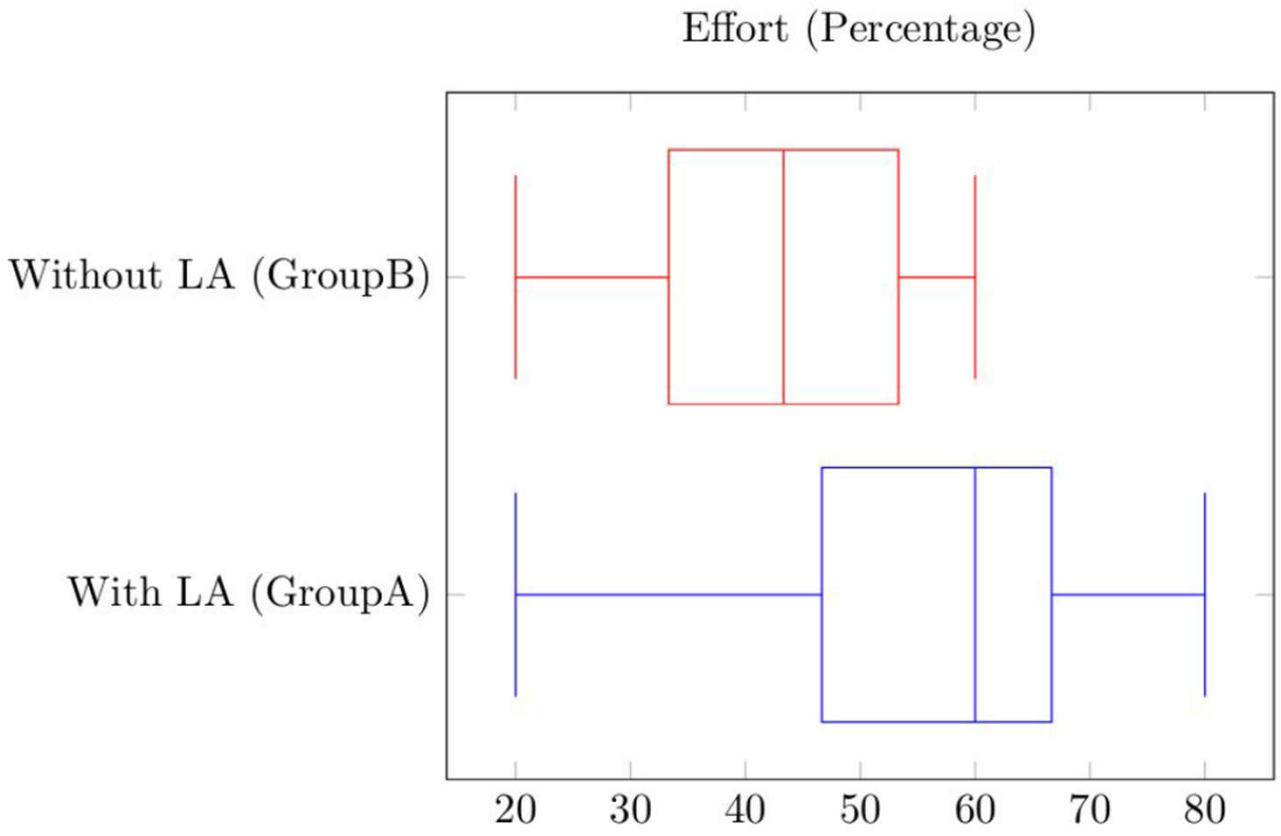
  ➢ Make wrong assumptions when they use them

!! This is clearly reflected by the results of the activity you did last week

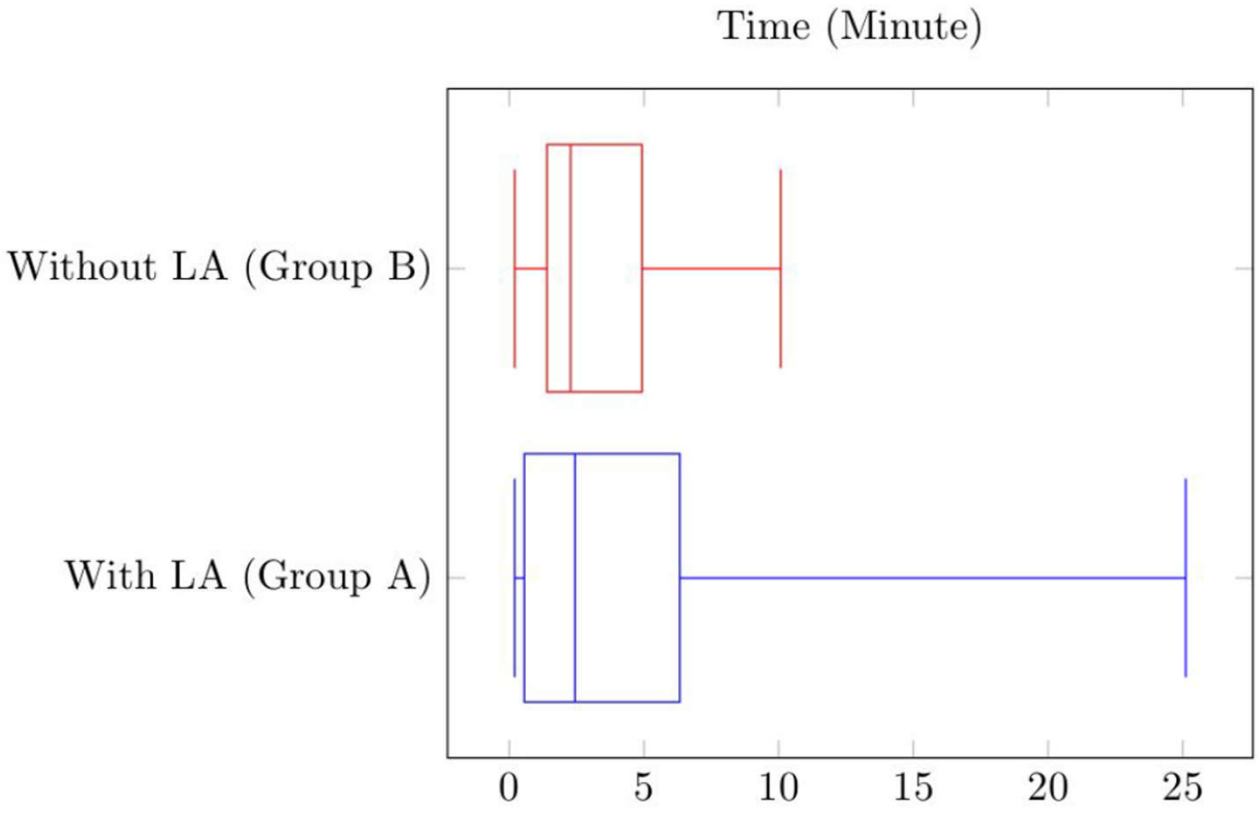
Results of Activity on LA - Correctness



Results of Activity (LA) - Effort (cognitive)



Results of Activity (LA) - Effort (time)



LA's catalogue

Method

- **Do more** than they say
  - A.1 - "Get" - more than an accessor
  - A.2 - "Is" returns more than a Boolean
  - A.3 - "Set" method returns
  - A.4 - Expecting but not getting a single instance
- **Do less** than they say
  - B.1 - Not implemented condition
  - B.2 - Validation method does not confirm
  - B.3 - "Get" method does not return
  - B.4 - Not answered question
  - B.5 - Transform method does not return
  - B.6 - Expecting but not getting a collection
- **Do** the **opposite** of what they say
  - C.1 - Method name and return type are opposite
  - C.2 - Method signature and comment are opposite

Attribute's

- **Name says more** than the entity contains
  - D.1 - Says one but contains many
- **Name says less** than the entity contains
  - D.2 - Name suggests Boolean but type does not
- **Name says the opposite** of what the entity contains
  - E.1 - Says many but contains one
  - F.1 - Attribute name and type are opposite
  - F.2 - Attribute signature and comment are opposite

Practice

Identify the LA contained in the following code snippet, then write the corresponding LA Identifier.

```
public int isValid() {
    final long currentTime = System.currentTimeMillis();
    if (currentTime <= this.expires) {
        // The delay has not passed yet - assuming source is valid.
        return SourceValidity.VALID;
    }

    // The delay has passed, prepare for the next interval.
    this.expires = currentTime + this.delay;

    return this.delegate.isValid();
}
```

A2. “Is” returns more than a Boolean

```
public int isValid() {
    final long currentTime = System.currentTimeMillis();
    if (currentTime <= this.expires) {
        // The delay has not passed yet - assuming source is valid.
        return SourceValidity.VALID;
    }

    // The delay has passed, prepare for the next interval.
    this.expires = currentTime + this.delay;

    return this.delegate.isValid();
}
```

A2. “Is” returns more than a Boolean

Rational:

- Having an “is” method does not return a Boolean, but returns more information is counterintuitive.

Consequences:

- Such problems will be detected at compile time (or even by the IDE
- Misleading naming can still cause misunderstanding from the maintainers’ side.

• **Refactoring**:

- By renaming the method to “***Validity”***
- By documenting that *“The method check the source validity by considering correct time and the expired time.”*

Practice

Identify the LA contained in the following code snippet, then write the corresponding LA Identifier.

```
public Dimension setBreadth(Dimension target, int source) {
    if (orientation == VERTICAL)
        return new Dimension(source ,
            (int)target.getHeight()) ;
    else
        return new Dimension (
                (int)target.getWidth(), source ) ;
}
```

A3. “Set” method returns (do more)

```
public Dimension setBreadth(Dimension target, int source) {
    if (orientation == VERTICAL)
        return new Dimension(source ,
            (int)target.getHeight()) ;
    else
        return new Dimension (
                (int)target.getWidth(), source ) ;
}
```

Doesn't set anything!!!
Creates a new objects and returns it

A3. “Set” method returns (do more)

Rational:

- By convention, set methods do not return anything

! Valid exceptions: returning the modified attribute, or returning the object in which the method is defined to allow chaining method calls

Consequences:

- The developer uses the setter method without storing/checking its returned value:
  - Loose useful information
  - erroneous or unexpected behavior—not captured

Refactoring:

- By renaming the method to “***createDimensionWithBreadth”***
- By documenting that *“The method creates a Dimension and sets its breadth to the value of source.”*

Practice

Identify the LA contained in the following code snippet, then write the corresponding LA Identifier.

```
protected void getMethodBodies
(CompilationUnitDeclarationunit , int place){
    //fill the methods bodies in order
    // for the code to be generated
    if(unit.ignoreMethodBodies) {
        unit.ignoreFutherInvestigation = true ;
        return; // if initial diet parse did not
        // work , no need to dig into method bodies.
        }
    if(place < parseThreshold)
        return ; // work already done ...
    // real parse of the method ....
    parser.scanner.setSourceBuffer(
            unit.compilationResult.
            compliationUnit.getContents());
    if(unit.types != null) {
        for(int i = unit.types.length; --i >= 0;)
            unit.types[ i ].parseMethod(parser , unit) ;
        }
    }
```

B3. “Get” method does not return (do less)

Nothing is returned!! Where is the retrieved data stored and how to obtain it??

```
protected void getMethodBodies
(CompilationUnitDeclarationunit , int place){
    //fill the methods bodies in order
    // for the code to be generated
    if(unit.ignoreMethodBodies) {
        unit.ignoreFutherInvestigation = true ;
        return; // if initial diet parse did not
        // work , no need to dig into method bodies.
        }
    if(place < parseThreshold)
        return ; // work already done ...
    // real parse of the method ....
    parser.scanner.setSourceBuffer(
            unit.compilationResult.
            compliationUnit.getContents());
    if(unit.types != null) {
        for(int i = unit.types.length; --i >= 0;)
            unit.types[ i ].parseMethod(parser , unit) ;
        }
    }
```

B3. “Get” method does not return (do less)

- **Rational**:
  - Suggests that an object will be returned as a result of the method execution.
  - Returning void is **counterintuitive**
- **Consequences**:
  - The developer expects to be able to assign the method return value to a variable.
- **Refactoring**:
  - Renaming the method to “***fillMethodBodies”*** (parse/set)
  - Code modification
  - Adding documentation: *“The method parses the method bodies and stores the result in the parameter unit”.*

Practice

Identify the LA contained in the following code snippet, then write the corresponding LA Identifier.

```
public void isValid(
        Object[] selection, StatusInfores){
    // only single selection
        if(selection.length == 1
                && (selection[0] instanceof IFile))
            res.setOK();
        else res.setError( ” ” ) ; / / $NON-NLS-1$
        }
```

B4. Not answered question

```
public void isValid(
        Object[] selection, StatusInfores){
    // only single selection
        if(selection.length == 1
                && (selection[0] instanceof IFile))
            res.setOK();
        else res.setError( ” ” ) ; / / $NON-NLS-1$
        }
```

method = predicate but the return type is not Boolean.

B4. Not answered question

- **Rational**:
  - The method name is in the form of predicate whereas the return type is not Boolean.
- **Consequences**:
  - Like B3, the developer would even expect to use the method within a conditional control structure, which is however not possible
- **Refactoring**:
  - Adding documentation: *"The result of the validation and the validation message are stored in res"*.
  - Changing the return type to Boolean, by returning "true" when the selection is valid and "false" otherwise.

Practice

Identify the LA contained in the following code snippet, then write the corresponding LA Identifier.

```
/*  Saves the current enable/ disable state of
 * the given control and its descendents in the
 * returned object; the controls are all disabled.
 * @param w the control
 *  @return an object capturing the enable/disable
 *  state */
public static ControlEnabledState
disable(Control w){
    return new ControlEnableState(w);
}
```

C1. Method name and return type are opposite

```
/*  Saves the current enable/ disable state of
 * the given control and its descendents in the
 * returned object; the controls are all disabled.
 * @param w the control
 *  @return an object capturing the enable/disable
 *  state */
public static ControlEnabledState
disable(Control w){
    return new ControlEnableState(w);
}
```

C1. Method name and return type are opposite

- **Rational**:
  - The return type mush be consistent, i.e., not in contradiction, with the method's name
- **Consequences**:
  - The developers can make wrong assumptions on the returned value and this might not be discovered at compile time.
  - ! When the method returns a Boolean—the developer could negate (or not) the value where it should not be negated (or it should be)..
- **Refactoring**:
  - Rename the class **"ControlEnableState"** to **"ControlState"** to handle the case where the state is enabled but also where the state is disabled.
  - The inconsistency with method disable is resolved as it will be returning a ControlState.

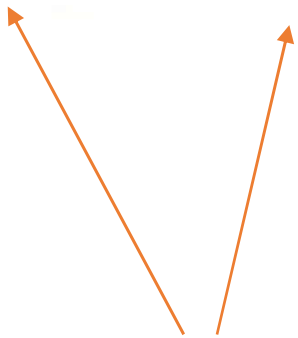
Practice

Identify the LA contained in the following code snippet, then write the corresponding LA Identifier.

```
protected static Category cat =
Category.getInstance(TableModelCritics.class);
//////////////////
// instance varables
Vector _target;
```

D1. Says one but contains many

```
protected static Category cat =
Category.getInstance(TableModelCritics.class);
////////////////
// instance varables
Vector _target;
```



! Don’t know whether the change impacts a one or multiple objects

D1. Says one but contains many

- **Rational**:
  - The name of an attribute and its type must be consistent
    - If the name suggests that a single instance the type must also do.

- **Consequences**:
  - Lack of understanding of the class state/associations.
  - When such attribute changes, one would know whether the change impacts a one or multiple objects.

- **Refactoring**:
  - Renaming the attribute to ***"targetCritics"*** or simply ***"critics"***

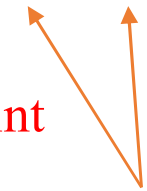
Practice

Identify the LA contained in the following code snippet, then write the corresponding LA Identifier.

```
/**
*    This class implements a navigation history
*
*    @author Curt Arnold
*    @since 0.9
*/
public class NavigationHistory {

    private List _history;
    private int _position;
    //
    //  tri-state boolean (-1 unevaluated, 0 false, 1 true)
    //
    int _isForwardEnabled = -1;
    int _isBackEnabled = -1;
```

D2. Name suggest Boolean but Type doesn't

```
/**
*     This class implements a navigation history
*
*     @author Curt Arnold
*     @since 0.9
*/
public class NavigationHistory {

    private List _history;
    private int _position;
    //
    //  tri-state boolean (-1 unevaluated, 0 false, 1 true)
    //
    int _isForwardEnabled = -1;
    int _isBackEnabled = -1;
```



int

True/False

! Not clear how to handle this attribute.

D2. Name suggest Boolean but Type doesn't

- **Rational**:
  - The name of an attribute and its type must be consistent
    - If the name suggests that a Boolean value is contained then the declared type must be indeed Boolean

- **Consequences**:
  - The developer would expect to be able to test the attribute in a control flow statement condition

- **Refactoring**:
  - The type of the int can be changed to boolean[]

Practice

Identify the LA contained in the following code snippet, then write the corresponding LA Identifier.

```
public SAXParserBase() { }

//////////////////////////////////////////////////////////////
// static variables

protected static  boolean       _dbg            = false;
protected static  boolean       _verbose        = false;

private   static  XMLElement    _elements[]     = new XMLElement[100];
private   static  int           _nElements      = 0;
private   static  XMLElement    _freeElements[] = new XMLElement[100];
private   static  int           _nFreeElements  = 0;
private   static  boolean       _stats          = true;
private   static  long          _parseTime      = 0;

//////////////////////////////////////////////////////////////
// instance variables

protected         boolean       _startElement   = false;
```

E1. Says many but contains one

```
public SAXParserBase() { }

//////////////////////////////////////////////////////////////////
// static variables

protected static  boolean       _dbg            = false;
protected static  boolean       _verbose        = false;

private   static  XMLElement    _elements[]     = new XMLElement[100];
private   static  int           _nElements      = 0;
private   static  XMLElement    _freeElements[] = new XMLElement[100];
private   static  int           _nFreeElements  = 0;
private   static  boolean       _stats          = true;
private   static  long          _parseTime      = 0;

//////////////////////////////////////////////////////////////////
// instance variables

protected         boolean       _startElement   = false;
```

single boolean

! contains statistics (collection)

E1. Says many but contains one

- **Rational**:
  - When the name suggests multiple objects the type must also.

- **Consequences**:
  - When such attribute changes,
    - Lack of understanding of the impact
    - Does the change impacts one or multiple objects?

- **Refactoring**:
  - Rename the attribute to *"**statisticsEnabled**"*

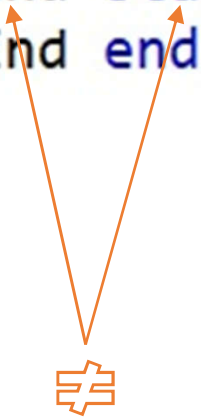
Practice

Identify the LA contained in the following code snippet, then write the corresponding LA Identifier.

```
public class ActionNavigability extends UMLAction {
    final public static int BIDIRECTIONAL = 0;
    final public static int STARTTOEND = 1;
    final public static int ENDTOSTART = 2;

    int nav = BIDIRECTIONAL;
    MAssociationEnd start = null;
    MAssociationEnd end = null;
```

F1. Attribute name and type are opposite

```
public class ActionNavigability extends UMLAction {
    final public static int BIDIRECTIONAL = 0;
    final public static int STARTTOEND = 1;
    final public static int ENDTOSTART = 2;

    int nav = BIDIRECTIONAL;
    MAssociationEnd start = null;
    MAssociationEnd end = null;
```



F1. Attribute name and type are opposite

- **Rational**:
  - The name and declaration type of an attribute are expected to be consistent with each other. Not contradict the other.

- **Consequences**:
  - Inducing wrong assumptions.
  - Could confuse the developer about the direction a data structure should be traversed.

- **Refactoring**:
  - Rename the class “MAssociationEnd” to *“**MAssociationExtremity**”*

Practice

Identify the LA contained in the following code snippet, then write the corresponding LA Identifier.

```
/**
 * Configuration default exclude pattern,
 * ie img/@src
 */
public final static String EXCLUDE_NAME_DEFAULT = "img/@src=";

/**
 * Configuration default exclude pattern,
 * ie .*\/@href|.*\/@action|frame/@src
 */
public final static String INCLUDE_NAME_DEFAULT = ".*/@href=|.*/@action=|frame/@src=";
```

F2. Attribute signature and comment are opposite

```
/**
 * Configuration default exclude pattern,
 * ie img/@src
 */
public final static String EXCLUDE_NAME_DEFAULT = "img/@src=";

/**
 * Configuration default exclude pattern,
 * ie .*\/@href|.*\/@action|frame/@src
 */
public final static String INCLUDE_NAME_DEFAULT = ".*/@href=|.*/@action=|frame/@src=";
```

F2. Attribute signature and comment are opposite

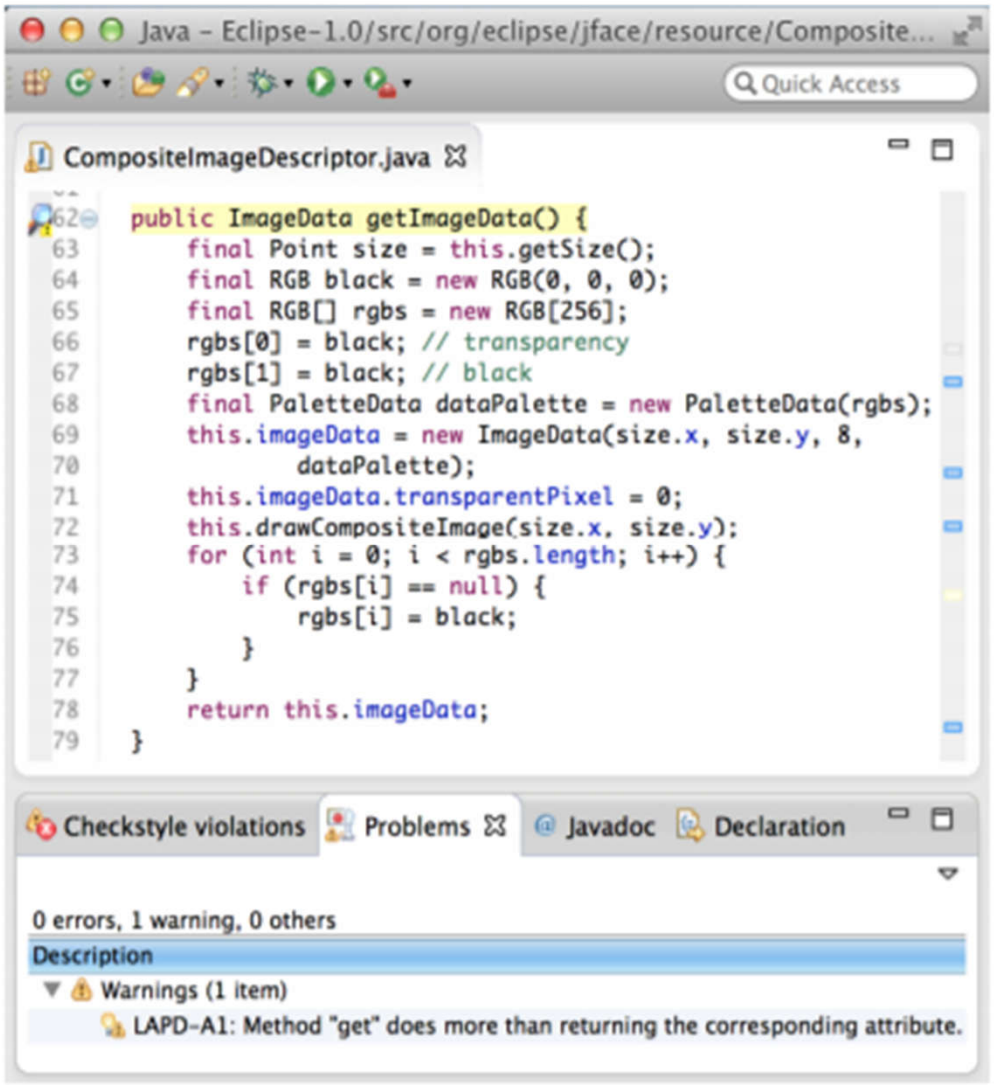
- **Rational**:
  - The comment of an attribute clarifies its intent and as such there must be no contradiction between the attribute's comments and declaration.

- **Consequences**:
  - Increasing comprehension effort as without a deep analysis of the source code, the developer might not clearly understand the role of the attribute.

- **Refactoring**:
  - Correcting the comment "default include pattern" being consistent with the name of the attribute_ i.e. INCLUDE_NAME_DEFAULT

Tools **LAPD**

- Linguistic Anti-Pattern Detector
- Java source code
- Integrated into Eclipse as part of the Eclipse Checkstyle Plugin5



http://www.veneraarnaoudova.ca/linguistic-anti-pattern-detector-lapd/