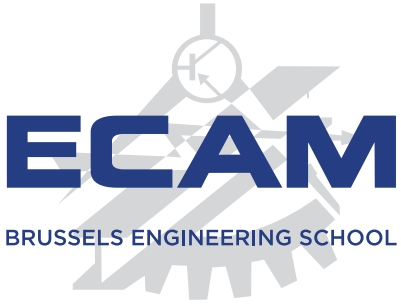


I402A Software Architecture
and Quality Assessment

Session 3

Bad Smells Elimination

Sébastien Combéfis

Fall 2019

Objectives

- Definition of a software pattern

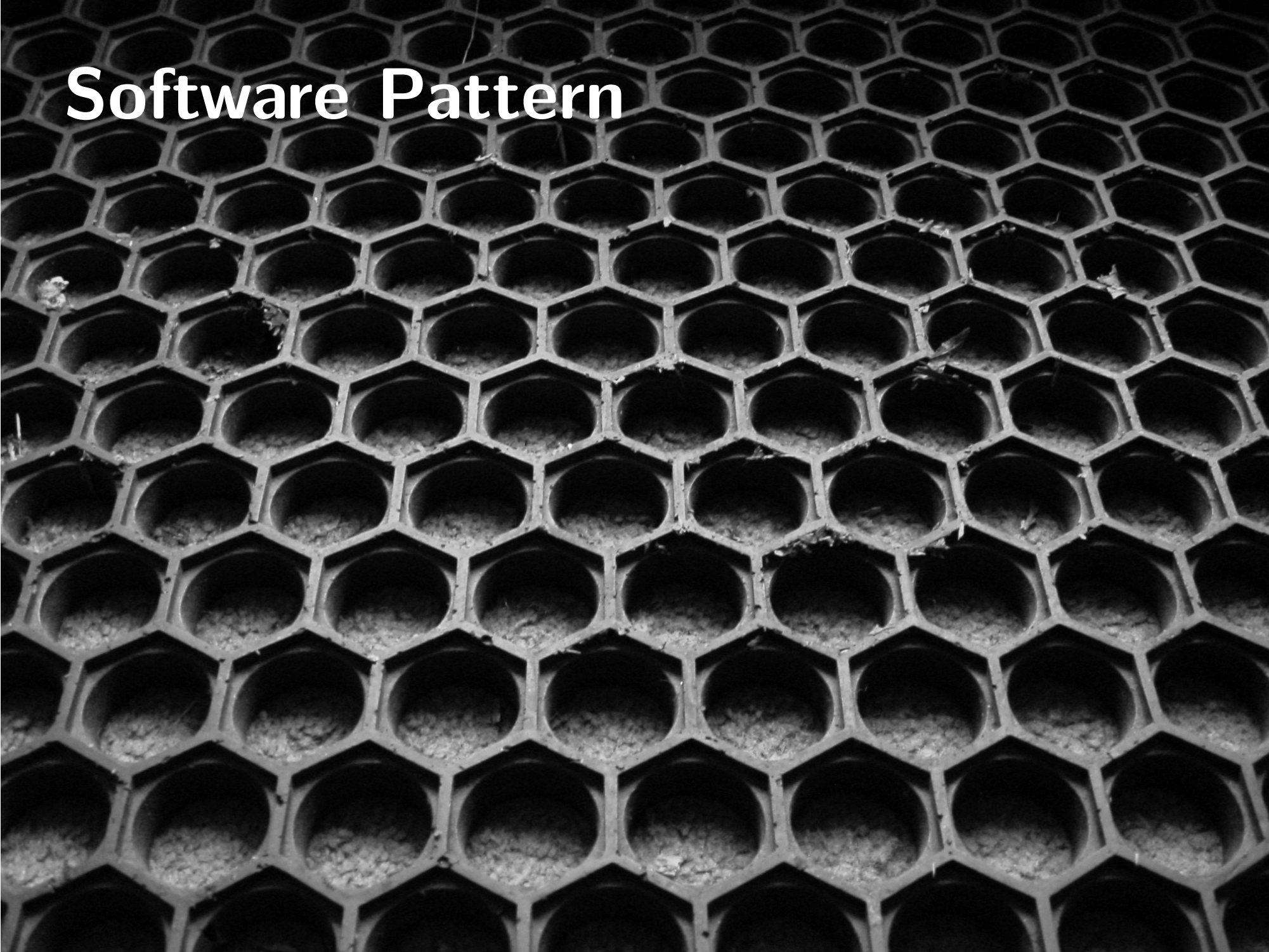
  Issues with over- and under-engineering

- Detection and elimination of code smells
  - Definition and categorisation of bad smells
  - Techniques to refactor code to patterns

- Anti-patterns detection and code improvement

  Main anti-patterns of what not to do

Software Pattern



Software Pattern

- Transmission of many useful design ideas

  Learn and understand a maximum number of patterns

- Bring a lot of advantages for the development
  - Much more flexible frameworks
  - More robust and scalable computer systems

- Can lead to software over-engineering

  It is not a good practice to apply patterns at all costs

Over-Engineering

- Code too flexible or sophisticated than it should
  - Wanting to anticipate future requirements too much
  - Wanting to follow a specific/precise programming model
  - Wanting to do more things than requested
- Code refactoring to and from design patterns
  - No longer use the pattern initially in the analysis phase
  - But use it and evolve it during the refactoring
- An over-engineered code is very difficult to maintain

  Also complicated to get into the code for new developers

Under-Engineering

- Generally much more under-engineered code

  Lead to software with a very bad design

- Several possible reasons
  - Not enough time to refactor code
  - No or little knowledge of software design
  - Pressures to quickly produce/modify a software
  - Participation in too many projects at the same time

- Serious consequences and deterioration of a bad software

  Becomes expensive and difficult, if not impossible, to maintain

Big Ball of Mud

- A big ball of mud is a bad code not structured

  Concept introduced by Brian Foote and Joseph Yoder in 1997

*"A Big Ball of Mud is a **haphazardly structured**, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle. These systems show unmistakable signs of **unregulated growth**, and repeated, expedient repair. [...] Only those who are unconcerned about architecture, and, perhaps, are comfortable with the inertia of the day-to-day chore of patching the holes in these failing dikes, are content to work on such systems."*

Test-Driven Development (TDD)

- Development driven by tests and continuous refactoring

  Two excellent practices from XP to improve quality

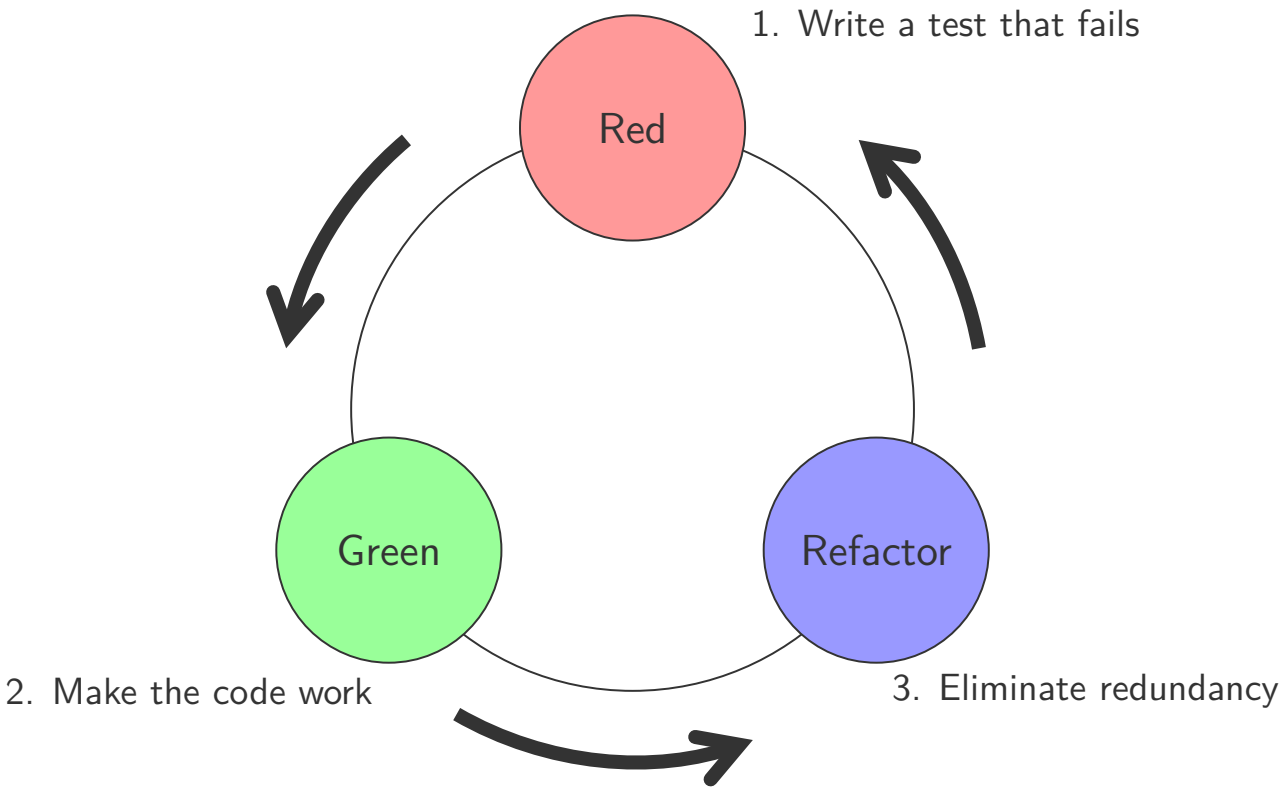
- Programming becomes a permanent dialogue

Ask	a question by writing a test
Respond	by writing a code that passes the test
Refine	answer by consolidating ideas, eliminating superfluity and clarifying ambiguities
Repeat	to maintain the dialogue by asking the next question

TDD Cycle

- TDD mantra and continuous refactoring by Kent Beck

  Lean programming style, iterative and disciplined




Code Smell

Kent Beck

- American software engineer born in 1961

  Computer science M.S. at the Oregon University

- Contributions to several domains
  - Extreme Programming (XP)
  - Signatory of the Agile Manifesto
  - Test-Driven Development
  - Software design patterns
  - JUnit that launched the xUnit

Bad Smell

- Bad smell (or *code smell*) popularised by Kent Beck

  These are not bugs, but design weaknesses

- Refactoring needed to improve its code

  Improving the quality while keeping the same features

- Some examples of bad smells
  - Duplicated code
  - Long method, long class
  - Too many parameters
  - Cyclomatic complexity

The Blob Example (1)

- A single class monopolises the whole system (God Class)
  - All the computation is done by a single class
  - Other classes only store data
- Break the concept *"one class = one feature"*

  And "one class = data and behaviour"

The Blob Example (2)

```
1  class Coord:
2     def __init__(self, x, y):
3        self.x = x
4        self.y = y
5
6  class Rectangle:
7     def __init__(self, lowerleft, width, height):
8        self.lowerleft = lowerleft
9        self.width = width
10       self.height = height
11
12 class Perso:
13    def __init__(self, width, height):
14       self.position = Coord(0, 0)
15       self.bb = Rect(self.position, width, height)
16
17    def move(self, dx, dy):
18       self.position.x += dx
19       self.position.y += dy
20
21    def x(self):
22       return self.position.x
23
24    def insideBB(self, x, y):
25       return 0 <= x - self.position.x <= self.bb.width
```

Duplicated Code Example (1)

- Duplicated or very similar code is present

  Statements, methods, classes, etc.

- Duplicated code is very hard to maintain

  Modification must be done at all duplicated locations

```
1  def printVerdict(score):
2      if score >= 10:
3          print('You obtained {} points, you succeeded!'.format(score))
4      else:
5          print('You obtained {} points, you failed!'.format(score))
6
7  if __name__ == "__main__":
8      try:
9          points = int(input('Enter your grade: '))
10         if 0 <= points <= 20:
11             printVerdict(points)
12         else:
13             raise ValueError
14     except ValueError:
15         print('Please enter an integer between 0 and 20.')
```

Duplicated Code Example (2)

- Factoring duplicated code into methods, classes, etc.

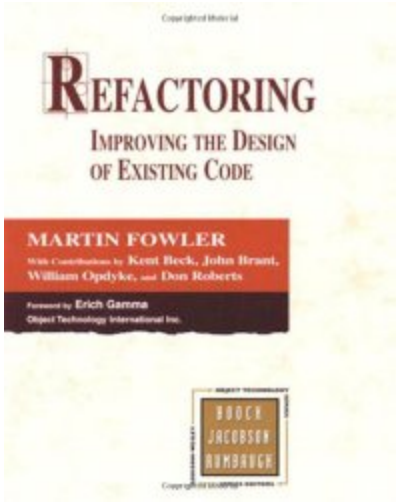
  Abstraction of common parts and definition of specific parts

```
1  def printVerdict(score):
2      verdict = 'succeeded'
3      if score < 10:
4          verdict = 'failed'
5      print('You obtained {} points, you {} !'.format(score, verdict))
6
7  if __name__ == "__main__":
8      try:
9          points = int(input('Enter your grade: '))
10         if 0 <= points <= 20:
11             printVerdict(points)
12         else:
13             raise ValueError
14     except ValueError:
15         print('Please enter an integer between 0 and 20.')
```

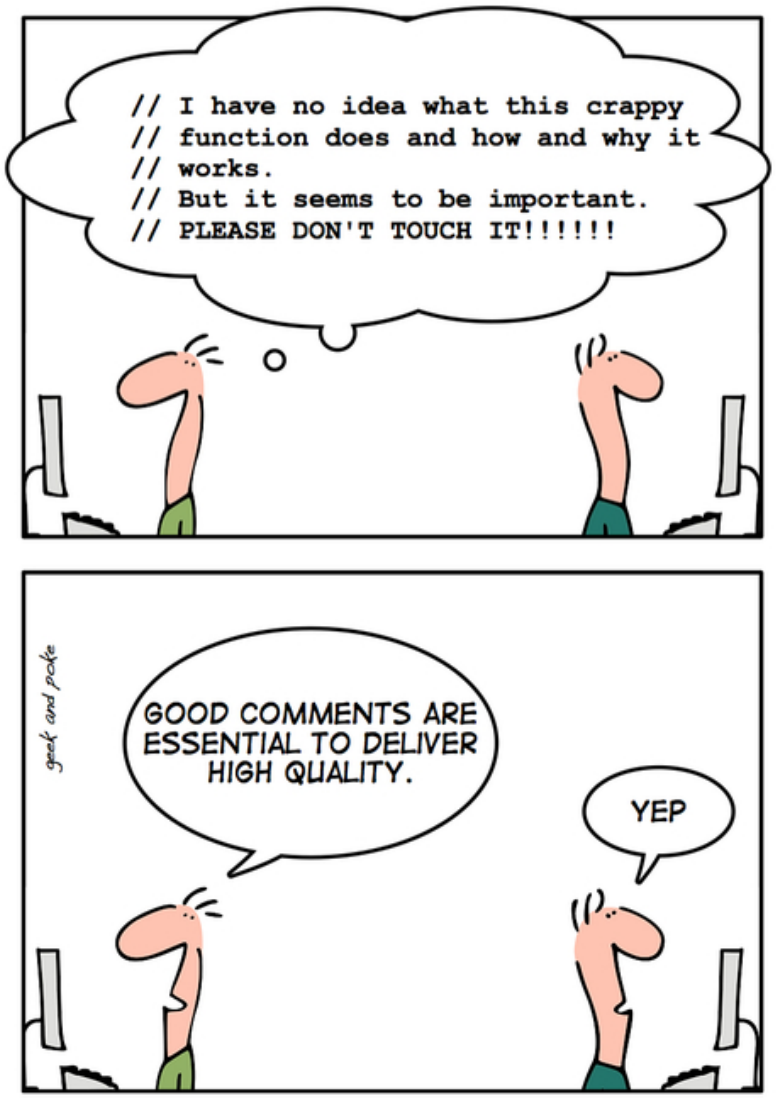
Code Smell

- The 22 code smells by Martin Fowler and Kent Beck

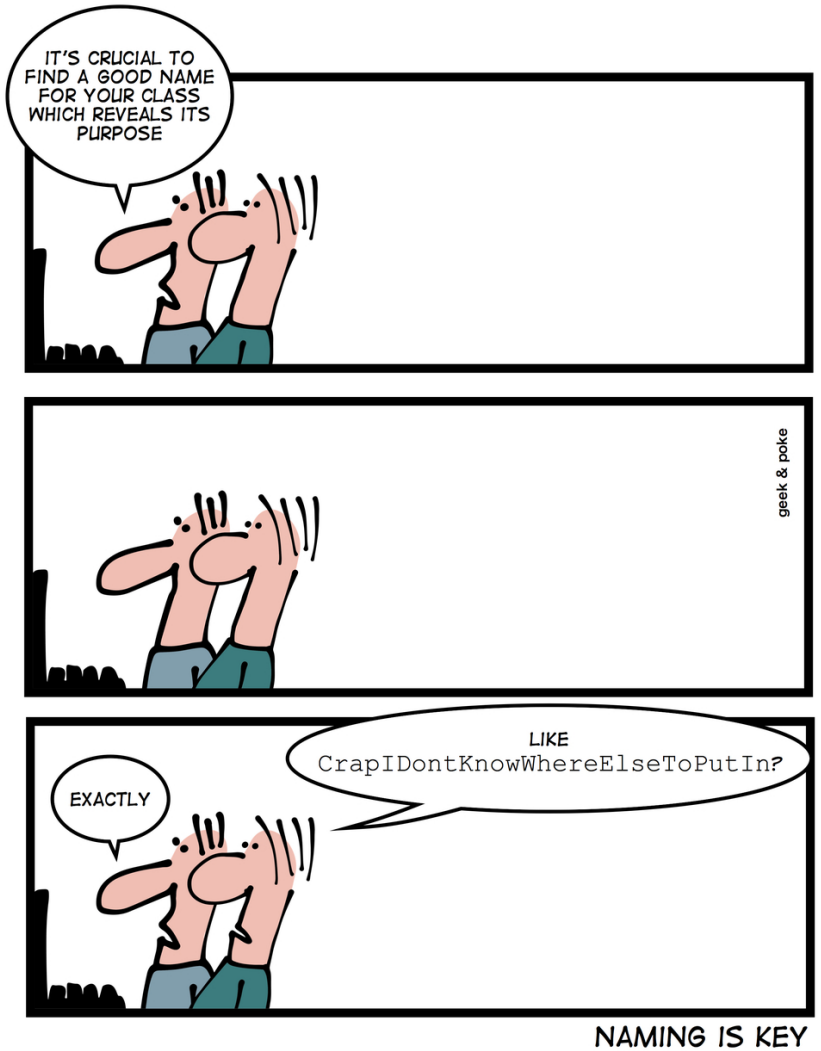
• Alternative classes with different interfaces	• Long method
• Comments	• Long parameter list
• Data class	• Message chains
• Data clumps	• Middle man
• Divergent change	• Parallel inheritance hierarchies
• Duplicated code	• Primitive obsession
• Feature envy	• Refused bequest
• Inappropriate intimacy	• Shotgun surgery
• Incomplete library class	• Speculative generality
• Large class	• Switch statements
• Lazy class	• Temporary field



Comment well your code!



Choose well your name!



Duplicated Code

- Two forms of duplicated code
  - Explicit duplication producing identical code
  - Subtle duplication with similar structure or process
- **Solutions**
  - Template Design Pattern
  - Chain Constructor
  - Extract Composite
  - ...

Example: Duplicated Code

```
public final class Contact
{
   private String firstName;
   private String lastName;
   private Address address;

   public Contact (String lastName)
   {
      this.lastName = lastName;
   }

   public Contact (String firstName, String lastName)
   {
      this.firstName = firstName;
      this.lastName = lastName;
   }

   public Contact (String firstName, String lastName, Address address)
   {
      this.firstName = firstName;
      this.lastName = lastName;
      this.address = address;
   }
}
```

Refactoring: Duplicated Code

```
public final class Contact
{
   private String firstName;
   private String lastName;
   private Address address;

   public Contact (String lastName)
   {
      this (null, lastName, null);
   }

   public Contact (String firstName, String lastName)
   {
      this (firstName, lastName, null);
   }

   public Contact (String firstName, String lastName, Address address)
   {
      this.firstName = firstName;
      this.lastName = lastName;
      this.address = address;
   }
}
```

Long Method

- Method whose body contains many statements
  - Does not have a single feature
  - Limit reusability
  - Difficult to understand
- **Solutions**
  - Compose Method
  - Extract Method
  - ...

Example: Long Method

```c
int main()
{
    int data[15];

    // Initialise
    int i;
    for (i = 1; i <= 15; i++) {
        data[i - 1] = 42;
    }

    // Display
    printf("[");
    int j;
    for (j = 0; j < 15; j++) {
        if (j == 14) {
            printf("%d", data[j]);
        } else {
            printf("%d, ", data[j]);
        }
    }
    printf("]\n");

    return 0;
}
```

Refactoring: Long Method

```
void fill_tab (int *data, int N, int value)
{
    int i;
    for (i = 0; i < N; i++) {
        data[i] = value;
    }
}

void print_tab (int *data, int N)
{
    printf("[%d", tab[0]);
    int j;
    for (j = 1; j < N; j++) {
        printf(", %d", data[j]);
    }
    printf("]\n");
}

int main()
{
    int N = 15;
    int data[N];
    fill_tab(data, N, 42);
    print_tab(data, N);
    return 0;
}
```

Large Class

- Big class... God Class
  - Too many instance variables
  - Too much responsibility
- **Solutions**
  - Extract Class
  - Extract Subclass
  - ...

Example: Large Class

```ruby
class Student
  def initialize(firstname, lastname, day, month, year, studyyear, courses)
    @firstname = firstname
    @lastname = lastname
    @day = day
    @month = month
    @year = year
    @studyyear = studyyear
    @courses = courses
  end

  def printname
    puts "#@firstname #@lastname"
  end

  def birthyear
    @year
  end

  def age
    2014 - @year
  end

  def isregistered(course)
    @courses.include? course
  end
end
```

Refactoring: Large Class (1)

```ruby
class Date
  def initialize(day, month, year)
    @day = day
    @month = month
    @year = year
  end

  attr_reader :year
end

class Person
  def initialize(firstname, lastname, birthdate)
    @firstname = firstname
    @lastname = lastname
    @birthdate = birthdate
  end

  def printname
    puts "#@firstname #@lastname"
  end

  def age
    2014 - @birthdate.year
  end
end
```

Refactoring: Large Class (2)

```
class Student < Person
  def initialize(firstname, lastname, birthdate, studyyear, courses)
    super(firstname, lastname, birthdate)
    @studyyear = studyyear
    @courses = courses
  end

  attr_reader :courseslist

  def isregistered(course)
    @courses.include? course
  end
end
```


Refactoring

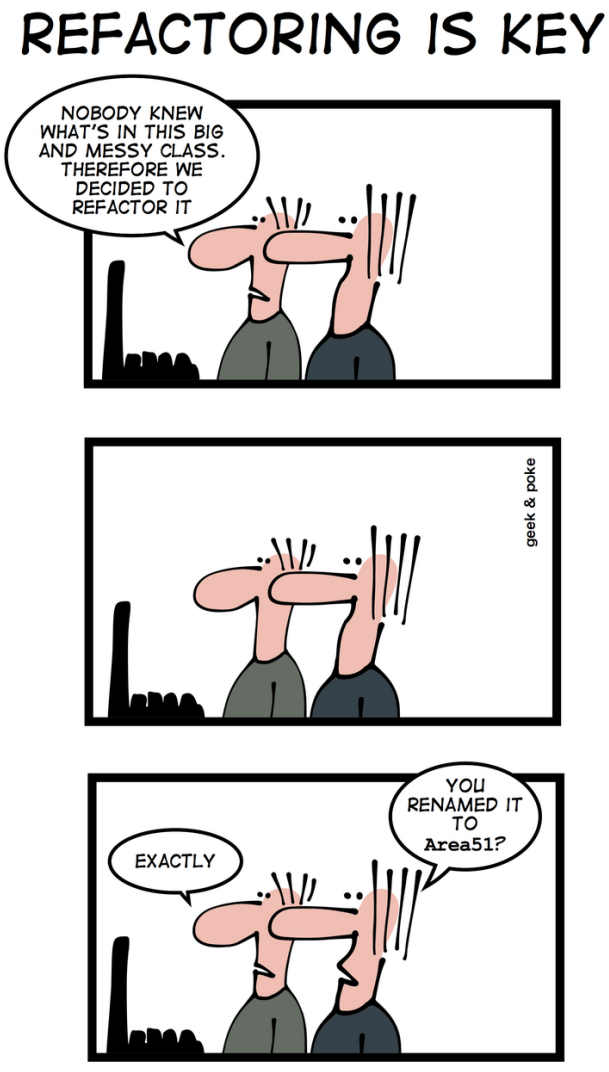
Refactoring (1)

- Code transformation that preserves its behaviour

  *"A **change** made to the internal structure of software to make it easier to understand and cheaper to modify **without changing** its observable behaviour" —Martin Fowler*

- **Goals**
  - Make it easier to add new code
  - Improve the design of existing code
  - Better understand a code
  - Make the code less boring

Refactoring (2)



Refactoring (3)

- Only refactor functional and tested code

  Test suites to ensure that refactoring has not changed anything

- Design problems come from code that is...
  - ...duplicated
  - ...unclear
  - ...complex
  - ...

Keeping it clean!

- Important not to let a code become dirty

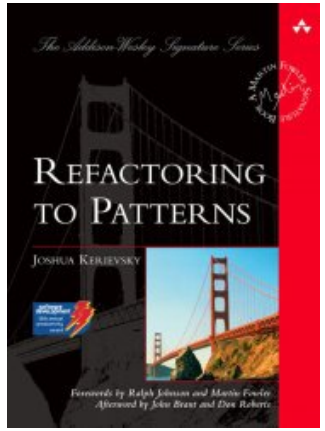
  Regular refactoring help to obtain a quality code

"Keeping code clean is a lot like keeping a room clean. Once your room becomes a mess, it becomes harder to clean. The worse the mess becomes, the less you want to clean it."

Refactor to Patterns

- The 27 refactoring techniques of Kerievski

- Chain constructors
- Compose Method
- Encapsulate Classes with Factory
- Encapsulate Composite with Builder
- Extract Adapter
- Extract Composite
- Extract Parameter
- Form Template Method
- Inline Singleton
- Introduce Null Object
- Introduce Polymorphic Creation with Factory Method
- Limit Instanciation with Singleton
- Move Accumulation to Collecting Parameter
- Move Accumulation to Visitor
- Move Creation Knowledge to Factory
- Move Embellishment to Decorator
- Replace Conditional Dispatcher with Command
- Replace Conditional Logic with Strategy
- Replace Constructors with Creation Methods
- Replace Hard-Coded Notifications with Observer
- Replace Implicit Language with Interpreter
- Replace Implicit Tree with Composite
- Replace One/Many Distinctions with Composite
- Replace State-Altering Conditionals with State
- Replace Type Code with Class
- Unify Interfaces
- Unify Interfaces with Adapter



Form Template Method

- Two similar methods in two subclasses

  Similar steps done and in the same order

- **Solution**
  - Generalise the method by extracting common parts
  - Put the method in superclass to make it a Template
  - Keep the specific parts in subclasses

*"**Template Methods** implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behaviour that can vary."*

Move Accumulation to Collecting Param.

- Big method that accumulates information

  Constructing data in a local variable

- **Solution**
  - Identify an object that can accumulate information
  - Decompose the big method in several methods
  - Pass the collector object as a parameter of these methods

Refactoring and Agile Development

- Short iterations: development, test, refactoring...

  Identify and work on small range contributions

- Refactoring should not affect existing features

  Everything that worked must continue to work

- Ideally, tests after each code transformation

  Combination of refactoring with TDD

Agile Development Example (1)

```
<html>
   <head><title>My Application</title></head>
   <body>
      <h1>Home</h1>
      <ul>
         <li><a href="page1.htm">Page 1</a></li>
         <li><a href="page2.htm">Page 2</a></li>
         <li><a href="page3.htm">Page 3</a></li>
      </ul>
   </body>
</html>
```

```
<html>
   <head><title>My Application</title></head>
   <body>
      <h1>Page 1</h1>
      <ul>
         <li><a href="page1-1.htm">Page 1</a></li>
         <li><a href="page1-2.htm">Page 2</a></li>
      </ul>
   </body>
</html>
```

Agile Development Example (2)

```
<html>
   <head><title>Home</title></head>
   <script type="text/javascript" src="jquery-1.11.1.min.js"></script>
   <script type="text/javascript" src="script.js"></script>
   <body>
      <ul id="menu1">
         <li><a href="page1.htm">Page 1</a></li>
         <li><a href="page2.htm">Page 2</a></li>
         <li><a href="page3.htm">Page 3</a></li>
      </ul>
      <ul id="menu2" style="display: none">
         <li><a href="page1-1.htm">Page 1</a></li>
         <li><a href="page1-2.htm">Page 2</a></li>
      </ul>
   </body>
</html>
```

```
$(function(){
   $('#menu1 li a:nth-child(1)').click (function(){
      $('#menu1').hide();
      $('#menu2').show();
   });
});
```

Agile Development Example (3)

```
<html>
   <head><title>Home</title></head>
   <script type="text/javascript" src="jquery-1.11.1.min.js"></script>
   <script type="text/javascript" src="script.js"></script>
   <body>
      <ul id="menu"></ul>
   </body>
</html>
```

```
{"data" : [{"id" : "menu1",
            "content" : [{"name": "Page 1", "url" : "page1.html"},
                         {"name": "Page 2", "url" : "page2.html"},
                         {"name": "Page 3", "url" : "page3.html"}]},
           {"id" : "menu2",
            "content" : [{"name": "Page 1", "url" : "page1-1.html"},
                         {"name": "Page 2", "url" : "page1-2.html"}]}
          ]}
```

```
$(function(){
   $.getJSON("data.json", function (data) {
      // ...
      $('#menu').html(content);
   });
});
```


Antipattern

Antipattern

- Unified vocabulary to identify design flaws

  Description of a generic solution to correct the flaw

- Several possible causes of appearance of an antipattern
  - Insufficient knowledge to solve a problem
  - Applying a pattern in a bad context
  - Code written in the rush without prior analysis

- Three categories of antipatterns

  Development, architecture and project management

Antipattern Categories

- Software development
  - Code structure and development
  - The Blob, Lava Flow, Functional Decomposition, Golden Hammer, Spaghetti Code, Cut-and-Paste Programming, etc.
- Software architecture
  - Software structure at the system/enterprise level
  - Jumble, Swiss Army Knife, The Grand Old Duke of York, etc.
- Software project management
  - Development process, communication, resources management
  - Analysis Paralysis, Death by Planning, The Feud, etc.

Golden Hammer

- Abuse of the same solution or product

  The developer has a high level of competences of a solution

- Inappropriate or irrelevant application of a solution

  Loss of performances, reduced maintainability, etc.

- **Solutions**
  - Change mentalities to discover new technologies
  - Experimentation of smaller projects

Spaghetti Code

- Big pile of code with very little structure

  The main developer is completely lost after a while

- Antipattern often present in non object oriented

  OO already imposes some minimal structure

- **Solutions**
  - Code cleanup and refactoring
  - Prevention

Swiss Army Knife

- Large and overly complex interface

  Can contain up to thousands of methods

- Very rich complex interface, but difficult to understand

  Complex documentation update and maintainability

- **Solutions**

  - Define interface usage profile
  - Convention explaining how to use the technology

The Grand Old Duke of York

- Dive into implementation without thinking about architecture

  Everyone Charges Up the Hill, Lack of Architecture Instinct Abstractionists versus Implementationists

- *"Experts report that only 1 in 5 software developers is able to define good abstractions On hearing this, a practicing software architect retorted, 'It's more like 1 out of 50'."*

- **Solutions**

  - Define specific roles in the development team
  - Distinguish component/application developers roles

Death by Planning

- Excessive planning of a development project

  With constant tracking and monitoring, and planning control

- *"We can't get started until we have a complete program plan."*, *"The plan is the only thing that will ensure our success."*, *"As long as we follow the plan and don't diverge from it, we will be successful."*, *"We have a plan; we just need to follow it!"*

- **Solutions**

  - Plan products delivery deadline (finished or component)
  - Plan advance approval phases (go/no go)

The Feud

- Personality conflicts between managers

  Do not know who is leading the boat anymore and who to follow

- Employees become as dark as managers

  Lack of productive communication, and cooperation, etc.

- **Solutions**
  - Friendly team meeting and out of the work context
  - Appointments with facilitators, psychologists, etc.

"There is no problem that a pizza party cannot solve." —Dr. Randall Oakes

References

- RDX, *Modern Software Over-Engineering Mistakes*, July 21, 2016.
  https://medium.com/@rdsubhas/10-modern-software-engineering-mistakes-bc67fbef4fc8
- Brian Foote, & Joseph Yode (1997). Big Ball of Mud. *Pattern Languages of Program Design 4*, chapter 29, 654–692.
- Dave Farinelli, *The Benefits of Test-driven Development*, October 11, 2018.
  https://devops.com/the-benefits-of-test-driven-development
- Martin Fowler, Kent Beck, John Brant, William Opdyke, & Don Roberts, *Refactoring: Improving the Design of Existing Code*, 2nd Edition, Addison-Wesley. (ISBN: 978-0-134-75759-9)
- Mohamed Aladdin, *Write clean code and get rid of code smells with real life examples*, August 13, 2018.
  https://codeburst.io/write-clean-code-and-get-rid-of-code-smells-aea271f30318
- Igor Vorobiov, *The 5 Greatest Signs that Your Code Smells Bad*, February 18, 2018.
  https://medium.com/@ivorobioff/5-greatest-signs-that-your-code-smells-bad-6ccc2839fb2f
- Sydney Stone, *Code Refactoring Best Practices: When (and When Not) to Do It*, September 27, 2018.
  https://www.altexsoft.com/blog/engineering/code-refactoring-best-practices-when-and-when-not-to-do-it
- Ihor Sokolyk, *10 Tips To Keep Your Code Clean*, December 5, 2017.
  https://medium.com/oril/10-tips-to-keep-your-code-more-clean-2fa9aafea1cf
- Joshua Kerievsky, *Refactoring to Patterns*, Addison-Wesley. (ISBN: 978-0-321-21335-8)
- Matthew Jones, *The Daily Software Anti-Pattern*, August 13, 2018.
  https://exceptionnotfound.net/the-daily-software-anti-pattern/

Credits

- Book pictures from Amazon.
- Andrew Kelsall, February 25, 2006, https://www.flickr.com/photos/andrewkelsall/4167041386.
- Tjarko Busink, November 22, 2014, https://www.flickr.com/photos/sjekkiebunzing/15684976879.
- Derbeth, October 18, 2007, https://en.wikipedia.org/wiki/File:Kent_Beck_no_Workshop_Mapping_XP.jpg.
- SourceMaking.com, https://sourcemaking.com/files/v2/content/antipatterns/blob-2x.png.
- Oliver Widder (Geek and Poke), http://geekandpoke.typepad.com/geekandpoke/images/2008/07/24/goodcomments.jpg.
- Oliver Widder (Geek and Poke), http://geek-and-poke.com/geekandpoke/2013/8/20/naming-is-key.
- webmove, June 25, 2006, https://www.flickr.com/photos/daniello/176252127.
- Oliver Widder (Geek and Poke), http://geek-and-poke.com/geekandpoke/2013/8/26/refactoring-is-key.
- Sébastien Combéfis, August 28, 2016, https://www.flickr.com/photos/157142794@N06/48775503356.