

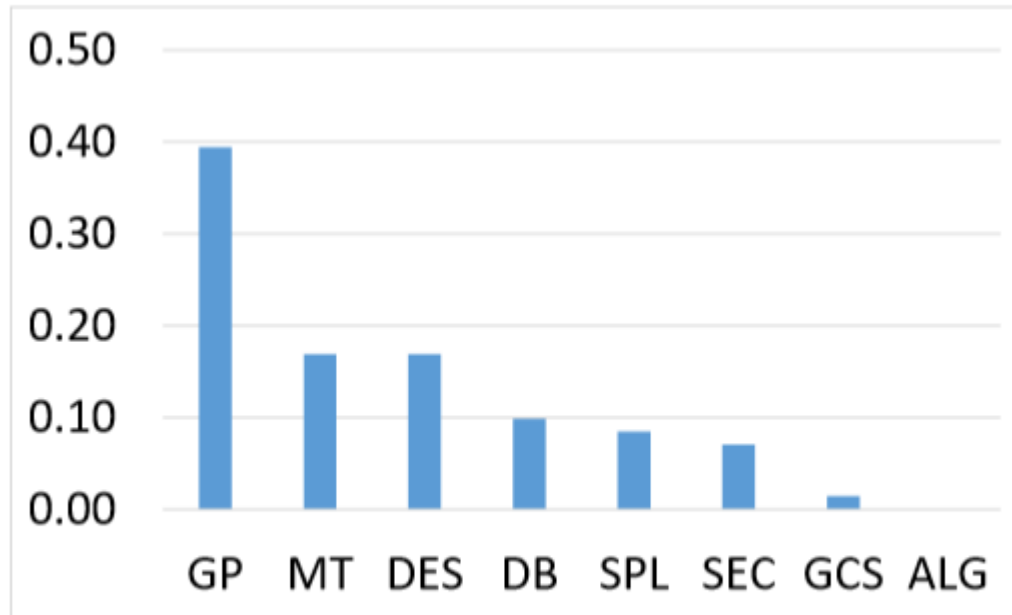
# **Code refactoring and review**

# Outline

- Code refactoring principles
  - Why refactoring
  - Bad smells in code
  - General refactoring principles and techniques
- Refactoring of Python code
- Code inspection and review
  - Testing vs. inspection
  - Reading techniques

# **Code refactoring principles**

# Readable and maintainable code is desired by industry



For software development (S), general programming knowledge (GP) and maintenance (MT) are the top two items.

**GP:** Clean and readable code with good documentation

**MT:** experience, especially with maintaining code

What competence do software companies want

Tor Stålhane, IDI, NTNU

Guttorm Sindre, IDI, NTNU

Extended version of conference paper "Hva vil programvareindustrien ha?", MNT-konferansen 2019

Stålhane, Deraas, Sindre and Abrahamsson:

# Bad smells in code (code smells)

- Code smells are any violation of fundamental design principles that decrease the overall quality of the code.
- Not bugs or errors
- Can certainly add to the chance of bugs and failures down the line.

# Code smells categories

- **Duplicated Code**
- **Long Method**
- **Large Class**
- **Long Parameter List**
- **Switch Statements**
- **Lazy Element**
- **Speculative Generality**
- **Temporary Field**
- **Divergent Change**
- **Shotgun Surgery**
- **Message Chains**
- **Middle Man**
- **Feature Envy**
- **Data Clumps**
- **Primitive Obsession**
- **Data Class**
- **Refused Bequest**

# Cod refactoring goals and properties

- Change the internal structure without changing external behavior
- Eliminate code smells for
  - Readability
  - Consistency
  - Maintainability
- Properties
  - Preserve correctness
  - One step at a time
  - Frequent testing

# Code refactoring steps

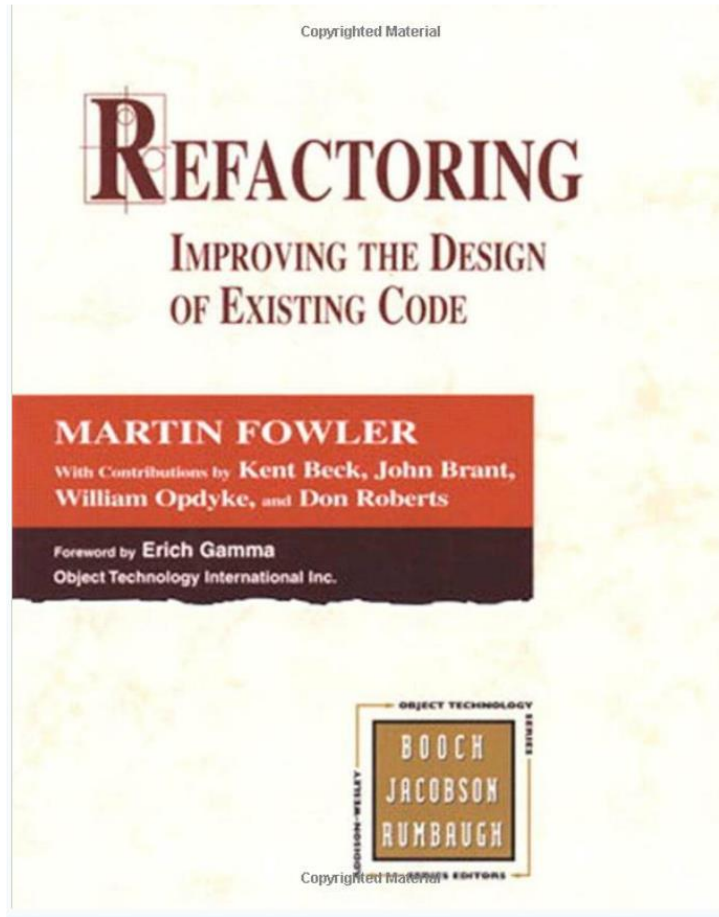
- Designing solid tests for the section to be refactored
- Reviewing the code to identify bad smells of code
- Introducing refactoring and running tests (One step at a time)



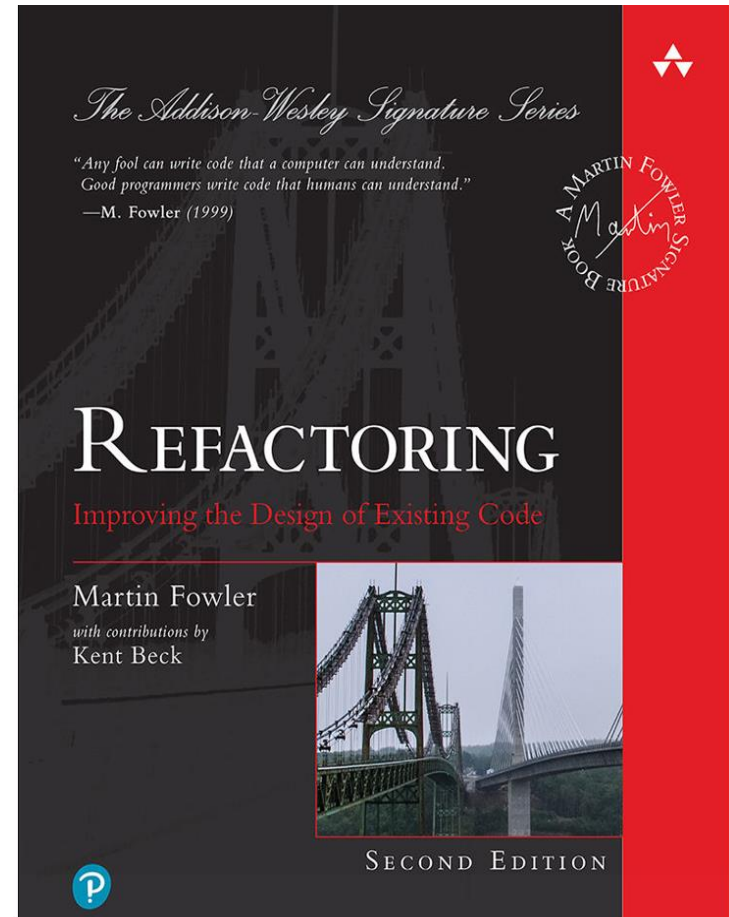
# Refactoring risks and countermeasures

- Refactoring is regarded as an overhead activity
  - Introducing failures with refactoring
  - Outdated comments and documents
- 
- Balance cost and benefits
  - Better to start from day one! Not at the end of the project/delivery
  - Should have sufficient and efficient regression tests

# Relevant books



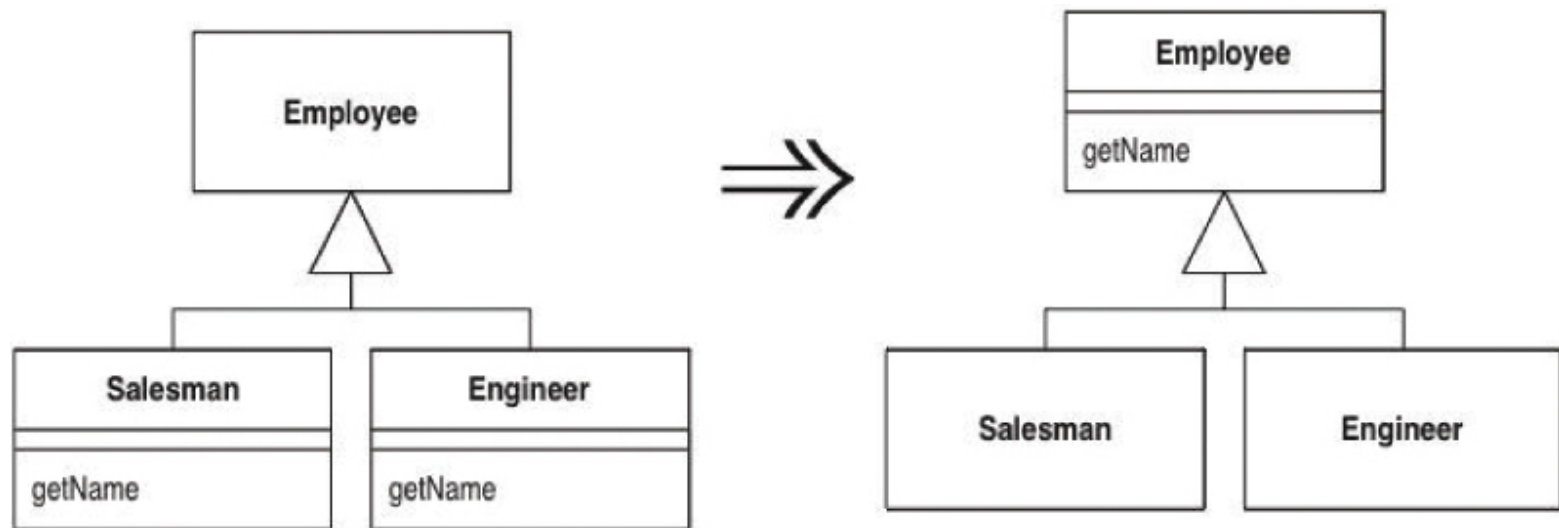
Old edition, available online at NTNU lib.



New edition

# Remove Duplicated Code – Pull up method (Don't repeat yourself (DRY))

- You have methods with identical results on subclasses.



# Remove Duplicated code - Substitute Algorithm

- You want to replace an algorithm with one that is clearer.

```
String foundPerson(String[] people){  
    for (int i = 0; i < people.length; i++) {  
        if (people[i].equals ("Don"))  
            { return "Don"; }  
  
        if (people[i].equals ("John"))  
            { return "John"; }  
  
        if (people[i].equals ("Kent"))  
            { return "Kent"; }  
    }  
    return "";  
}
```



```
String foundPerson(String[] people){  
    List candidates = Arrays.asList(new  
String[] {"Don", "John", "Kent"});  
  
    for (int i=0; i<people.length; i++)  
        if (candidates.contains(people[i]))  
            return people[i];  
  
    return "";  
}
```

# Reduce size – shorten method/class

- **Long Method**
  - E.g., Extract method
- **Large Class**
  - E.g., Extract class, subclass, interface

# Extract method example

If you have to spend effort looking at a fragment of code and figure out what it is doing, then you should extract it into a function/method and name it after “*what*.”

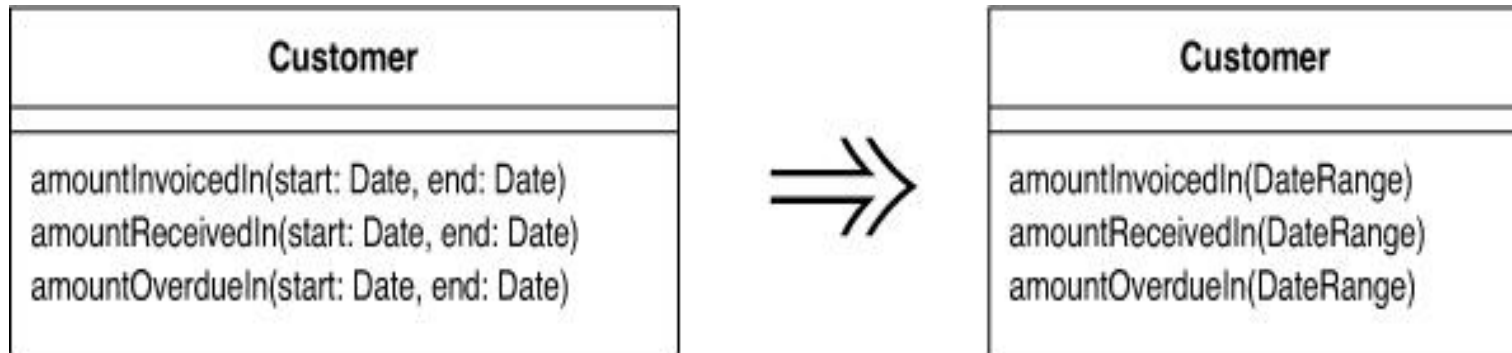
```
void printOwing()  
{  
    printBanner(); //print details  
    System.out.println ("name:  
        " + _name);  
    System.out.println ("amount  
        " + getOutstanding());  
}
```



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

# Reduce size – shorten parameter list

- **Long Parameter List**
  - E.g., Introduce Parameter Object



# Divergent Change

- **Code smell**
  - One module is often changed in different ways for **different reasons**.
  - Classes have more than distinct responsibilities that it **has more than one reason to change**
  - Violation of **single responsibility design** principle
- **Refactoring**
  - You identify everything that changes for a particular cause and put them all together



```
public class Account {  
    private int accountNumber;  
    private double balance = 0;  
  
    public Account(int accountNumber) {  
        this.accountNumber = accountNumber;  
    }  
  
    public int getAccountNumber() {  
        return accountNumber;  
    }  
  
    public double getBalance() {  
        return balance;  
    }  
  
    public void credit(double amount) {  
        balance += amount;  
    }  
  
    public void debit(double amount) {  
        balance -= amount;  
    }  
  
    public String toXml() {  
        return "<account><id>" + Integer.toString(getAccountNumber()) + "</id>" +  
            "<balance>" + Double.toString(getBalance()) + "</balance></account>";  
    }  
}
```

More than one responsibility, operation  
of a bank account + serializing the  
account

<https://www.slideshare.net/nadeembtech/code-craftsmanship>

```

public class Account {
    private int accountNumber;
    private double balance= 0;

    public Account(int accountNumber){
        this.accountNumber = accountNumber;
    }

    public int getAccountNumber() {
        return accountNumber;
    }

    public double getBalance() {
        return balance;
    }

    public void credit(double amount) {
        balance += amount;
    }

    public void debit(double amount) {
        balance -= amount;
    }
}

```

```

public class AccountXMLSerializer {

    public String toXml(Account account) {
        return "<account>" +
            "<id>" + Integer.toString(account.getAccountNumber()) + "</id>" +
            "<balance>" + Double.toString(account.getBalance()) + "</balance>" +
            "</account>";
    }
}

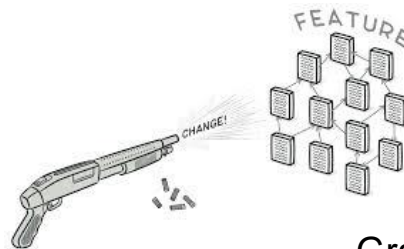
```

<https://www.slideshare.net/nadeembtech/code-craftsmanship>

# Shotgun Surgery

- **Code smell**

- A single change is made to multiple classes simultaneously
- When changes are all over the place, they are hard to find, and it's easy to miss important changes.



Graph from [refactoring.guru](http://refactoring.guru)

- **Refactoring**

- You put all the changes into a single class
- Once and only once (OAOO)

```

1 package com.example.codesmell;
2
3 public class Account {
4
5     private String type;
6     private String accountNumber;
7     private int amount;
8
9     public Account(String type,String accountNumber,int amount)
10    {
11        this.amount=amount;
12        this.type=type;
13        this.accountNumber=accountNumber;
14    }
15
16
17    public void debit(int debit) throws Exception
18    {
19        if(amount <= 500)
20        {
21            throw new Exception("Mininum balance shuold be over 500");
22        }
23
24        amount = amount-debit;
25        System.out.println("Now amount is" + amount);
26
27    }
28
29    public void transfer(Account from,Account to,int cerditAmount) throws Exception
30    {
31        if(from.amount <= 500)
32        {
33            throw new Exception("Mininum balance shuold be over 500");
34        }
35
36        to.amount = amount+cerditAmount;
37
38    }
39
40    public void sendWarningMessage()
41    {
42        if(amount <= 500)
43        {
44            System.out.println("amount should be over 500");
45        }
46    }
47

```

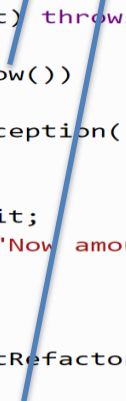
When we add another criterion in the validation logic: if the account type is personal and the balance is over 500

We have to make changes to all methods

<https://dzone.com/articles/code-smell-shot-surgery>

```
1 package com.example.codesmell;
2
3 public class AccountRefactored {
4
5     private String type;
6     private String accountNumber;
7     private int amount;
8
9
10
11     public AccountRefactored(String type,String accountNumber,int amount)
12     {
13         this.amount=amount;
14         this.type=type;
15         this.accountNumber=accountNumber;
16     }
17
18     private boolean isAccountUnderflow()
19     {
20         return amount<=500;
21     }
22 }
```

```
23
24
25 public void debit(int debit) throws Exception
26 {
27     if(isAccountUnderflow())
28     {
29         throw new Exception("Mininum balance shuold be over 500");
30     }
31
32     amount = amount-debit;
33     System.out.println("Now amount is" + amount);
34
35 }
36
37 public void transfer(AccountRefactored from,AccountRefactored to,int cerditAmount) throws Exception
38 {
39     if(isAccountUnderflow())
40     {
41         throw new Exception("Mininum balance shuold be over 500");
42     }
43
44     to.amount = amount+cerditAmount;
45
46 }
```



# Menti: code refactoring performed?

Go to [menti.com](https://menti.com), code 6344 0068

```
1 public class Account {
2     double principal, rate;    int daysActive, accountType;
3
4     public static final int STANDARD=0, BUDGET=1,
5         PREMIUM=2, PREMIUM_PLUS=3;
6 }
7
8 ...
9
10 public static double calculateFee(Account[] accounts)
11 {
12     double totalFee = 0.0;
13     Account account;
14     for (int i=0; i<accounts.length; i++) {
15         account=accounts[i];
16         if (account.accountType== Account.PREMIUM ||
17             account.accountType== Account.PREMIUM_PLUS)
18             totalFee+= .0125 * ( // 1.25% broker's fee
19                 account.principal* Math.pow(account.rate,
20                     (account.daysActive/365.25))
21                 -account.principal); // interest-principal
22     }
23
24     return totalFee;
25
26 }
27
```

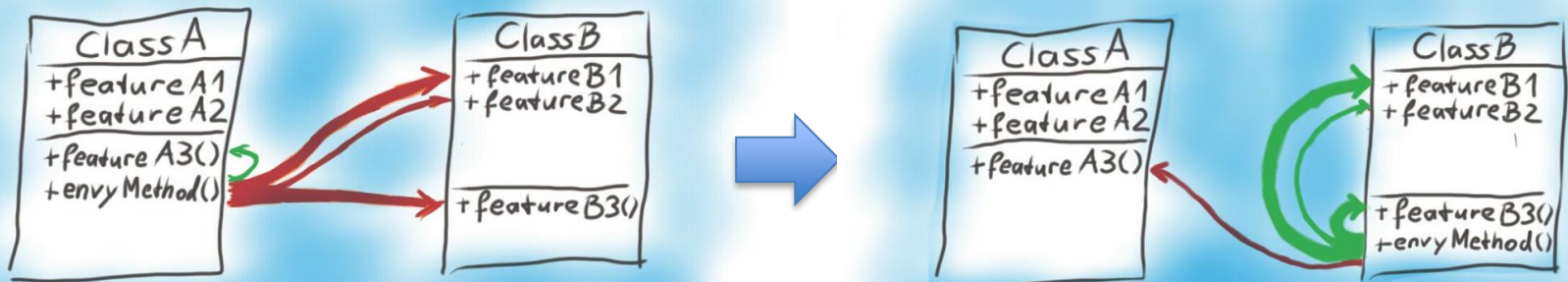


```
1 /** An individual account. Also see CorporateAccount. */
2
3 public class Account {
4
5     private double principal;
6     /** The yearly, compounded rate (at 365.25 days per year). */
7
8     private double rate; /** Days since last interest pay out */
9     private int daysActive;
10    private Type type;
11 }
12
13 /** The varieties of account our bank offers. */
14 public enum Type {STANDARD, BUDGET, PREMIUM, PREMIUM_PLUS}
15
16
17 /** Compute interest. */
18 public double interest() {
19     double years = daysActive / 365.25;
20     double compoundInterest = principal * Math.pow(rate, years);
21     return compoundInterest - principal;
22 }
23
24 /** Return true if this is a premium account. */
25 public boolean isPremium() {
26     return accountType == Type.PREMIUM ||
27         accountType == Type.PREMIUM_PLUS;
28 }
29
30 /** The portion of the interest that goes to the broker. */
31 public static final double BROKER_FEE_PERCENT = 0.0125;
32
33 /** Return the sum of the broker fees for all the given accounts. */
34 public static double calculateFee(Account accounts[]) {
35     double totalFee = 0.0;
36     for (Account account : accounts) {
37         if (account.isPremium()) {
38             totalFee += BROKER_FEE_PERCENT * account.interest();
39         }
40     }
41     return totalFee;
42 }
43 }
```

<http://www.ifi.uzh.ch/seal/teaching/courses/softwareWartung-FS16/CodeReviewExercise.pdf>\*

# Increase cohesion

- **Feature envy**
  - A function in one module spends more time communicating with functions or data inside another module than it does within its own module.
  - Move function to give it a dream home



<https://waog.wordpress.com/2014/08/25/code-smell-feature-envy/>

# Increase cohesion (Cont')

- **Data clumps**
  - Bunches of data often hang around together
  - Consolidate the data together, e.g., Introduce Parameter Object or Preserve Whole Object

```
int low = daysTempRange().getLow();  
int high = daysTempRange().getHigh();  
withinPlan = plan.withinRange(low, high);
```



```
withinPlan =  
plan.withinRange(daysTempRange());
```



# Primitive Obsession

- Primitive fields are basic built-in building blocks of a language, e.g., int, string, or constants
- Primitive Obsession is when the code relies too much on primitives and when uses primitive types to represent an object in a domain

```
class contactUs
{
    public function addressUsa()
    {
        $address = new Array();
        $address['streetNo'] = 2074;
        $address['streetName'] = 'JFK street';
        $address['zipCode'] = '507874';

        return $address['streetName']. ' '. $address['streetNo']. ', '. $address['zipCode'];
    }

    public function addressGermany()
    {
        $address = new Array();
        $address['streetNo'] = '25';
        $address['streetName'] = 'Frankfurter str.';
        $address['zipCode'] = '80256';

        return $address['streetName']. ' '. $address['streetNo']. ', '. $address['zipCode'];
    }

    public function hotLine(){
        return '+49 01687 000 000';
    }
}
```

The address is defined as an array.  
Every time we need the address we will  
have to hard code it

<https://codeburst.io/write-clean-code-and-get-rid-of-code-smells-aea271f30318>

```
class address{  
  
    private $streetNo;  
    private $streetName;  
    private $zipCode;  
  
    public function addressUsa()  
    {  
        $this->streetNo = 2074;  
        $this->streetName = 'JFK street';  
        $this->zipCode = '507874';  
  
        return $this->streetName. ' '. $this->streetNo. ', '. $this->zipCode;  
    }  
    public function addressGermany()  
    {  
        $this->streetNo = 25;  
        $this->streetName = 'Frankfurter str.';  
        $this->zipCode = '80256';  
  
        return $this->streetName. ' '. $this->streetNo. ', '. $this->zipCode;  
    }  
}
```

We create a new class called Address  
Every time we need to add/edit an  
address we hit the Address class

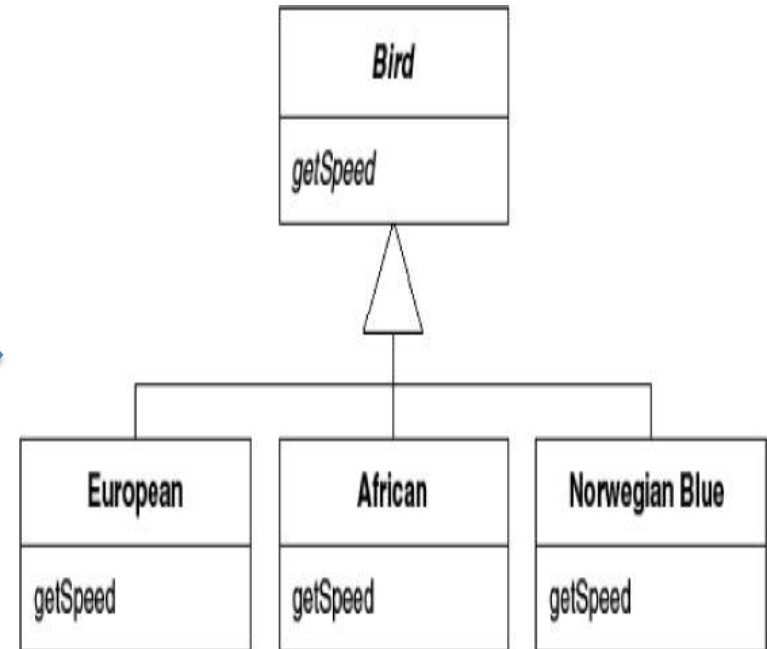
# Code smells categories

- **Duplicated Code**
  - **Long Method**
  - **Large Class**
  - **Long Parameter List**
- ← Decrease size
- **Divergent Change**
  - **Shotgun Surgery**
- ← Localize change
- **Feature Envy**
  - **Data Clumps**
  - **Primitive Obsession**
- ← Increase cohesion
- **Switch Statements**
  - **Lazy Element**
  - **Speculative Generality**
  - **Temporary Field**
  - **Message Chains**
  - **Middle Man**
- ← Simplify logic within and between objects
- **Data Class**
  - **Refused Bequest**
- ← Proper OO practice

# Simplify logic within object – Switch statement

- Replace Conditional with Polymorphism

```
double getSpeed() {  
    switch (_type) {  
        case EUROPEAN:  
            return getBaseSpeed();  
        case AFRICAN:  
            return getBaseSpeed() - getLoadFactor() ;  
        case NORWEGIAN_BLUE:  
            return getBaseSpeed(_voltage);  
        }  
        throw new RuntimeException ("error");  
    }
```



# Simplify logic within object – Switch statement (cont')

- Replace Parameter with Explicit Methods

```
void setValue (String name, int value) {  
  
    if (name.equals("height")) {  
        _height = value;  
        return;  
    }  
  
    if (name.equals("width")) {  
        _width = value;  
        return;  
    }  
  
    Assert.shouldNeverReachHere();  
}
```



```
void setHeight(int arg) {  
    _height = arg;  
}  
  
void setWidth (int arg) {  
    _width = arg;  
}
```

# Simplify logic within object – Lazy Element

- E.g., Lazy class: A class that isn't doing enough to pay for itself should be eliminated

```
class Person {  
    get officeAreaCode ()  
        { return this._telephoneNumber.areaCode;}  
    get officeNumber ()  
        {return this._telephoneNumber.number;}  
}
```

```
Class TelephoneNumber {  
    get areaCode () { return this._areaCode;}  
    get number () { return this._number;}  
}
```



```
Class Person {  
    get officeAreaCode ()  
        {return this._officeAreaCode;}  
  
    get officeNumber () {return  
        this._officeNumber;}  
}
```

# Simplify logic within object - Speculative Generality

- When you have code that **isn't actually needed today**.
- Such code often exist to support future behavior, which may or **may not be necessary in the future**
- Remove unnecessary delegation, unused parameters, dead code

```
Public class Customer {  
    private String name;  
    private String address;  
  
    Public Customer (String name, String add)  
    { this.name = name;  
      this.address = add;  
    }  
  
    String GetName () {  
        return name;  
    }  
}
```



# Simplify logic within object – Temporary Field

- Class has a variable which is only used in some situation
- Trying to understand why a variable is there when it doesn't seem to be used

```
class User
{
    private $id;
    private $name;
    private $priviledges;
    private $content;
    private $contactDetails;

    public function __construct()
    {
        $this->contactDetails = new UserContactDetails();
    }

    function notify($message)
    {
        $messageBody = 'Dear ' . $this->name;
        $messageBody .= '( ' . $this->contactDetails->getStreetNumber() . ' ' .
            $this->contactDetails->getStreetName() . ' ).';
        $messageBody .= $message;
        $notificationService->sendSMS($this->contactDetails->getPhoneNubmer(), $messageBody);
    }

    function delete()
    {
        systemDelete($this->id);
    }

    function update() {
        systemUpdate($this->id, $this->priviledges, $this->content);
    }
}
```

\$name and \$contactDetails are only used in the notify() method.

<https://codeburst.io/write-clean-code-and-get-rid-of-code-smells-aea271f30318>

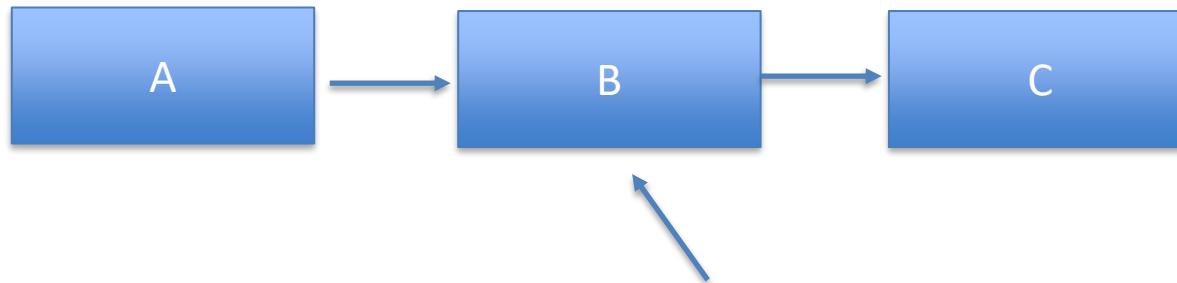
Why not pass them as a method parameters

```
function notify($userName, $contactDetails, $message) {  
  
    $messageBody = 'Dear ' . $userName;  
    $messageBody .= '( ' . $contactDetails->getStreetNumber() . ' ' .  
        $contactDetails->getStreetName() . ')';  
    $messageBody .= $message;  
  
    $notificationService->sendSMS($contactDetails->getPhoneNubmer(), $messageBody);  
}
```

<https://codeburst.io/write-clean-code-and-get-rid-of-code-smells-aea271f30318>

# Simplify logic between objects

- **Message Chains**
- **Middle Man**



Remove this intermediate object if not needed

Employee->EmployeeConfig->Config

Employee->Config

```
Class Employee{  
    public function getConfiguration(){  
        $this->employeeConfig->getConfiguration();  
    }  
}
```

```
Class EmployeeConfig{  
    public function getConfiguration(){  
        $this->config->getConfiguration();  
    }  
}
```

```
Class Config{  
    public function getConfiguration(){  
        $this->loadConfiguration();  
    }  
}
```



```
Class Employee{  
    public function getConfiguration(){  
        $this->config->getConfiguration();  
    }  
}
```

# Proper OO practice

- **Data Class**
  - Make a public field private and provide accessors

```
public String _name
```

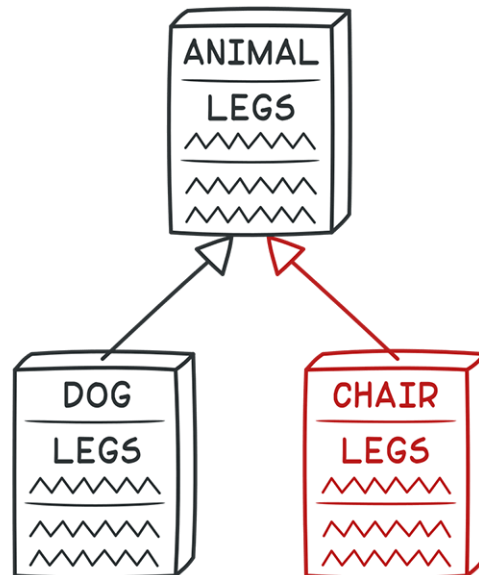


```
private String _name;  
public String getName() {return _name;}  
public void setName(String arg) {_name = arg;}
```

# Proper OO practice (Cont')

- **Refused Bequest**

- Only some of the inheritances are needed, i.e., hierarchy is wrong
- Inheritance makes no sense and the subclass really does have nothing in common with the superclass



<https://refactoring.guru/smells/refused-bequest>

# Comments in code

- Good to have, can be simplified
- **Best if code is self-explanatory**
  - If you need a comment to explain what a block of code does, **simply the code**
  - If the method is already extracted but you still need a comment to explain what it does, **rename the method**
  - If you need to state some rules about the required state of the system, **use Assertion**



# Python code refactoring

# Python code smell\* – code conventions

- Some are relevant to python code conventions\*\* , e.g.,
  - Function names should comply with a naming convention
  - Class names should comply with a naming convention
- Tool support
  - Pylint (Checking structure of the code is compliant with PEP-8)



\* <https://rules.sonarsource.com/python/type/Code%20Smell/RSPEC-1720>

\*\* <https://www.python.org/dev/peps/pep-0008/>

# PyLint demo



<https://www.youtube.com/watch?v=fFY5103p5-c>

# Python code smell\*

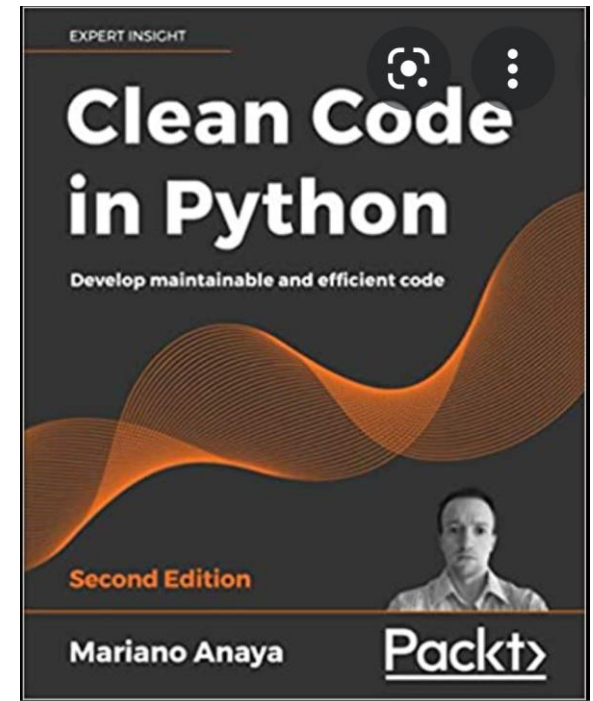
- Remove duplications
  - Redundant pairs of parentheses should be removed
  - Using features of Python, e.g., Decorator\*\*
- Reduce size
  - Functions, methods and lambdas should not have too many parameters
  - Lines should not be too long
- Speculative Generality, e.g.,
  - Nested blocks of code should not be left empty
  - Unused local variables should be removed

\* <https://rules.sonarsource.com/python/type/Code%20Smell/RSPEC-1720>

\*\*[https://www.youtube.com/watch?v=n\\_Y-\\_7R2KsY](https://www.youtube.com/watch?v=n_Y-_7R2KsY)

# Python code smell\* (cont')

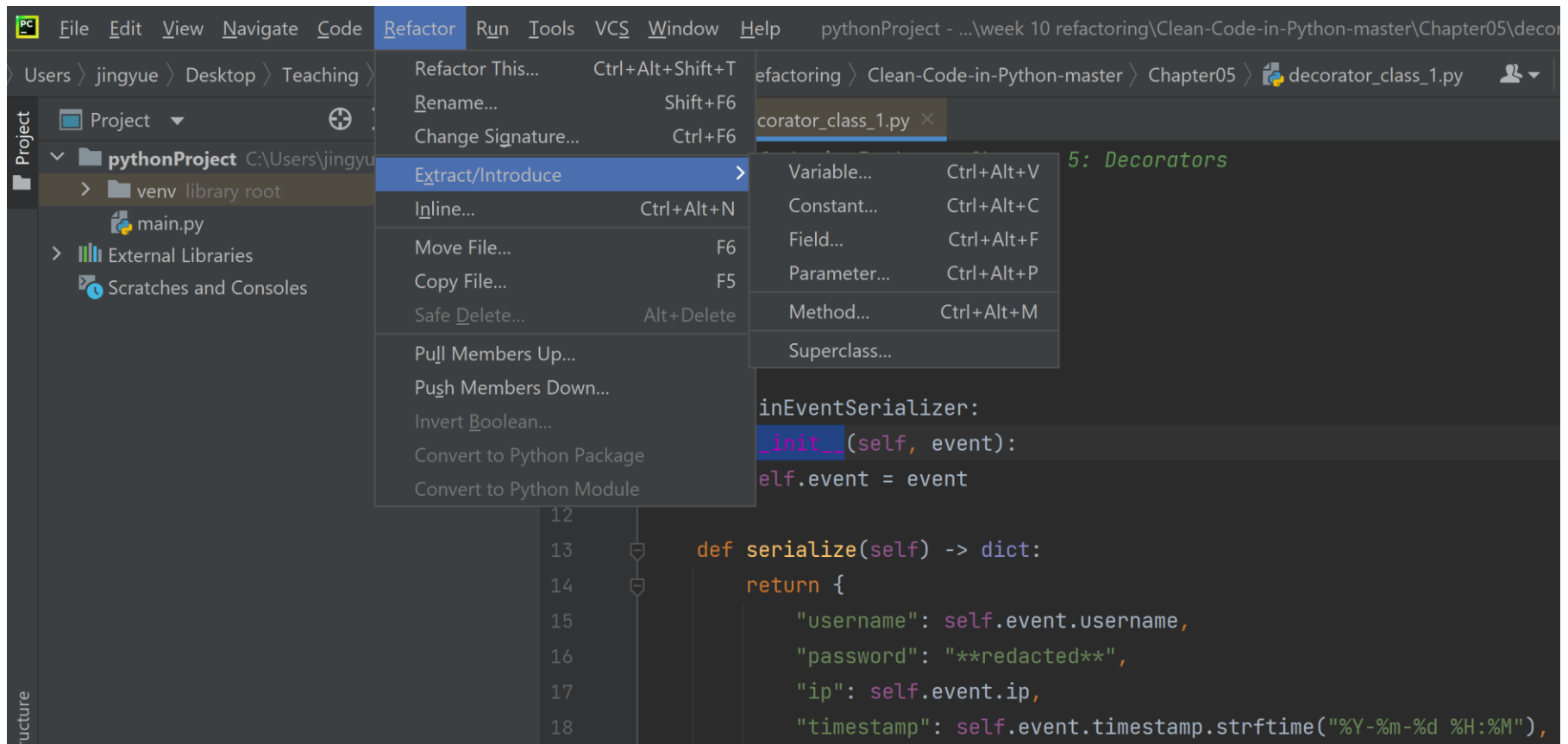
- Reduce complexity and simplify logics
  - Cognitive Complexity of functions should not be too high
  - Functions should not be too complex
  - Control flow statements "if", "for", "while", "try" and "with" should not be nested too deeply
  - Functions should not contain too many return statements



\* <https://rules.sonarsource.com/python/type/Code%20Smell/RSPEC-1720>

# Python code smell – tool support

E.g., PyCharm



# **Inspection and code review**

# What is inspection?

- Visual examination of software product
- Identify software anomalies
  - Errors
  - Code smells
  - Deviations from specifications
  - Deviations from standards
    - E.g., Java code conventions  
[www.oracle.com/technetwork/java/codeconventions-150003.pdf](http://www.oracle.com/technetwork/java/codeconventions-150003.pdf)



# Why do we need inspection?

- Many software artifacts cannot be verified by running tests, e.g.,
  - Requirement specification
  - Design
  - Pseudocode
  - User manual
- Inspection reduces defect rates
- Inspection complements testing
- Inspection identifies code smells
- Inspection provides additional benefits

# Inspection finds types of defects different from testing

Number of different types of defects detected by testing vs. inspection <sup>[1]</sup>

Some defect types	Testing	Inspection
Uninitialized variables	1	4
Illegal behavior, e.g., division by zero	49	20
Incorrectly formulated branch conditions	2	13
Missing branches, including both conditionals and their embedded statements	4	10

# Inspection reduces defect rates

Removal Step	Lowest Rate	Modal Rate	Highest Rate
Informal design reviews	25%	35%	40%
Formal design inspections	45%	55%	65%
Informal code reviews	20%	25%	35%
Formal code inspections	45%	60%	70%
Modeling or prototyping	35%	65%	80%
Personal desk-checking of code	20%	40%	60%
Unit test	15%	30%	50%
New function (component) test	20%	30%	35%
Integration test	25%	35%	40%
Regression test	15%	25%	30%
System test	25%	40%	55%
Low-volume beta test (<10 sites)	25%	35%	40%
High-volume beta test (>1,000 sites)	60%	75%	85%

Source: Adapted from *Programming Productivity* (Jones 1986a), "Software Defect-Removal Efficiency" (Jones 1996), and "What We Have Learned About Fighting Defects" (Shull et al. 2002).

# Testing vs. Inspection

	Testing	Inspection
Pros	<ul style="list-style-type: none"><li>• Can, at least partly, be automated</li><li>• Can consistently repeat several actions</li><li>• Fast and high volume</li></ul>	<ul style="list-style-type: none"><li>• Can be used on all types of documents, not just code</li><li>• Can see the complete picture, not only a spot check</li><li>• Can use all information in the team</li><li>• Can be innovative and inductive</li></ul>
Cons	<ul style="list-style-type: none"><li>• Is only a spot check</li><li>• Can only be used for code</li><li>• May need several stubs and drivers</li></ul>	<ul style="list-style-type: none"><li>• Is difficult to use on complex, dynamic situations</li><li>• Unreliable (people can get tired)</li><li>• Slow and low volume</li></ul>

# Inspection provides additional benefits [2]

- Code and change understanding
- Knowledge transfer
- Increased team awareness
- Creation of alternative solutions to problems

# Code review at Google [3]

- "All code that gets submitted needs to be reviewed by at least one other person, and either the code writer or the reviewer needs to have readability in that language. Most people use Mondrian to do code reviews, and obviously, we spend a good chunk of our time reviewing code."

--Amanda Camp, Software Engineer, Google

# Code reviews in OSS projects [4]

- Asynchronous and distributed review
  - Lightweight with tool support
  - Not limited to the existence of defects
  - Encourage discussing the best solution
- Frequent review
  - Most peer review happens **within hours** of completing a change
- Incremental review
  - Reviews should be of changes that are small, independent, and complete

*The potential system benefit of accepting code that hasn't been discussed by a group of experts doesn't outweigh the risks. (Linus Torvalds)*

# Tools to support inspection

- Google's Mondrian
- Facebook's Phabricator
- GitHub
- ...



# GitHub review tool

# Software reading techniques

- Unstructured: Ad hoc reading
- Semi-structured: checklist-based reading
- Structured or systematic reading
  - Defect-based reading
  - Perspective-based reading

# A sample (partial) checklist for requirement review [4]

1. Requirements specifications shall be **testable**.
2. Requirements specifications shall not **conflict** with other requirements specifications.
3. Conditional requirements specifications shall cover **all cases**.
4. Numerical values in requirements specifications **shall include physical units** if applicable.

# A sample (partial) checklist for code review [4]

1. Have resources (e.g., memory, file descriptor, database connection) properly been freed?
2. Are shared variables protected/thread-safe?
3. Is logging implemented?
4. Are comments updated and consistent with the code?
5. Is data unnecessarily copied, saved, or reloaded?
6. Is the number of cores checked before spawning threads?

# Checklist-based reading with active guidance

- Use a tailored checklist
- Guide reader how to use the checklist
  - Where to find?
  - To find what?
  - How to detect?



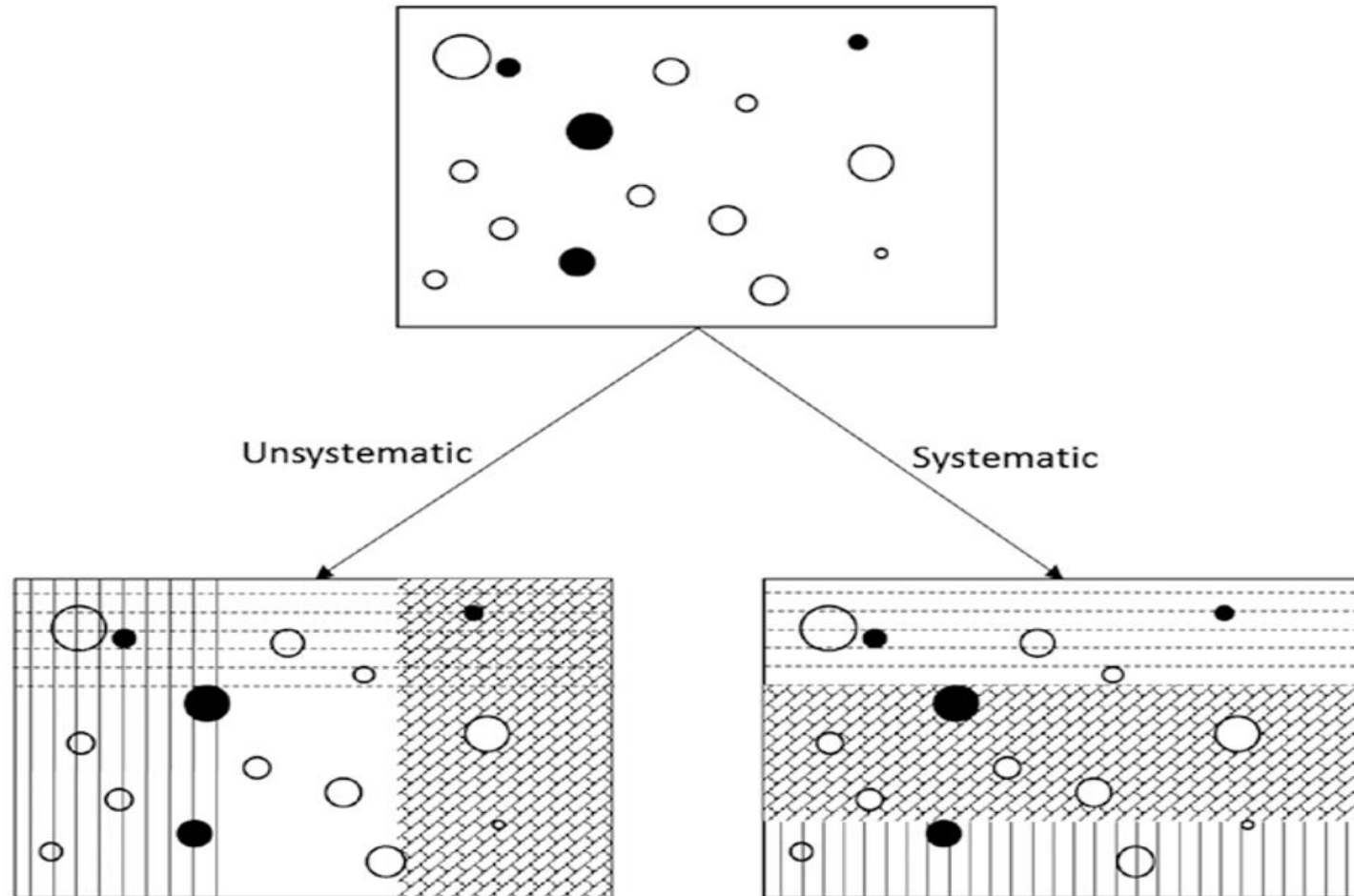
# An example of a checklist with active guidance [4]

			Where	What	How
Section	Feature	Checklist item			
For each class	Inheritance	Q1. Is all inheritance required by the design implemented in the class?			
		Q2. Is the inheritance appropriate?			
	Constructor	Q3. Are all instance variables initialized with meaningful values?			
		Q4. If a call to base class is required in the constructor, is it present?			
For each method	Data referencing	Q5. Are all parameters used within a method?			
		Q6. Are the correct class constants used?			
		Q7. Are indices of data structures operating within the correct boundaries?			

# Best practice of checklists

- Should be periodically revised
- Should be concise and fit on one page
- Should not be too general
- Should not be used for conventions which are better checked or enforced with software tools

# Unsystematic vs. Systematic reading





# Defect-based reading

- Each reader is given specific responsibility to discover a particular class of defects
- Specific responsibility from defect taxonomy
  - Wrong or missing assignment
  - Wrong or missing data validation
  - Wrong timing or sequencing
  - Interface problems
  - Etc.

# Perspective-based reading

- Each reader reviews software artifact from a particular perspective
- Then combine results from different perspectives
- Perspective examples
  - Novice user and expert user (GUI)
  - User, designer, tester (Requirements)

# Template for perspective-based reading

- Introduction
  - Explain stakeholder's interests
- Instruction
  - How to read and extract information
- Questions
  - For reviewers to answer by following the instruction

# Novice use usability reading guideline [5]

- Introduction
  - You review the usability of the software to make PPT from the perspective of users who are new to use software to make slides.
- Instruction 1
  - Scan through menu, sub-menu, help info. in GUI
- Questions
  - Are there instructions or online help?
  - Are items in menu, sub-menu unambiguous in meaning?

# Expert use usability reading guideline

- Introduction
  - You review the usability of the software to make PPT from the perspective of an expert who wants to use the software for making slides quickly.
- Instruction
  - Execute actions through finding and using shortcuts.
- Questions related to the instruction
  - Is it easy to find the frequently used shortcuts?
  - Is there any online help to guide the usage of shortcuts?

# Benefits of perspective-based reading

- Focused
  - Particular coverage of part of the document
- Detailed
  - Instruction and questions
- Adaptable
  - To particular software artifacts
- Tailorable
  - To organizational and project setting

# Effective factors affecting inspection

[6]

- Individual expertise and training
  - Most important factor
- Amount of materials
  - Not to attempt to inspect too many materials in one cycle
  - E.g., less than 125 lines of code per hour
- Team size
  - 4 inspectors may not find significantly more defects than 2 inspectors
  - 4 or 2 usually find more defects than 1
- Process and tool support
  - With good tool support, inspection meeting may not be necessary

# Summary

- We have studied
  - Why and how to do code refactoring
  - Why and how to perform code review



# References

- [1] S.S. So et al. An empirical Evaluation of Six Methods to Detect Faults in Software, Software Testing, Verification, and Reliability, 12, 3, 2002, pp. 155-171.
- [2] A. Bacchelli and C. Bird: Expectations, outcomes, and challenges of modern code review. Proc. ICSE 2013.
- [3] CSE 403 course at University of Washington, available at <https://courses.cs.washington.edu/courses/cse403/11sp/lectures/lecture15-reviews.pdf>
- [4] Y. M. Zhu: Software reading techniques. Apress, 2016.
- [5] Z. Zhang, V. Basili, and B. Shneiderman: Perspective-based usability inspection: An empirical validation of efficacy. Journal of empirical software engineering, 4, 1999, pp. 43-69.
- [6] S. Kollanus and J. Koskinen: Survey of software inspection research, The Open Software Engineering Journal, 3, 2009, pp. 15-34.
- [7] D. Rombach et al. Impact of research on practice in the field of inspections, reviews, and walkthroughs, ACM SIGSOFT SE notes, 33, 6, 2008, pp. 26-35.