Module 4

Done Right - Reviews, Testing, and Metrics for Software Improvement

Module Overview (Week 6 – Week 10)

- Topic 4.1 Software Reviews (Week 6)
  - The purposes and benefits of software reviews
  - Family of review methods: walkthrough, technical review, inspection
  - Requirement technical reviews and repair
  - Code review: pair programming, code smells and refactoring
- Topic 4.2 Software Testing (Week 7 – Week 9)
  - Fundamentals of software testing: challenges, types, levels
  - Test driven development
  - Black-box testing: boundary value testing, equivalence class testing, decision table testing
  - White-box testing: program dependence graph, DD-path testing and coverage criteria
- Topic 4.3 Software Metrics (Week 10)
  - Key issues in monitoring
  - Goal-Quality-Metric and desirable properties of metrics
  - Defect analysis

Learning Outcomes

- Identify techniques used in software review
- Apply pair programming, refactoring, and test-driven development
- Create effective plans for software testing
- Generate metrics for monitoring software produce quality

Module 4 – Topic 4.1

Software Reviews

Topic Outline

- Why reviews? Who benefits?
  - Software reviews are used for many purposes today
- Family of review techniques
  - Walkthrough, technical review, inspection
  - Requirement technical review and repair
- Code reviews
  - Pair programming
  - Code smells and refactoring

Why Reviews? Who benefits?

Cost-Benefit Analysis of Software Reviews

- Finding defects early and reducing rework can impact the overall cost of a project

Fagan reported that IBM inspections found 90% of all defects for a 9% reduction in average project cost

Jet Propulsion Laboratory study: average two hour inspection exposed four major and fourteen minor faults; savings estimated at $25,000 per inspection

Aetna insurance company: formal technical review found 82% of errors, 25% cost reduction

Johnson estimates that rework accounts for 44% of development cost

Cost of Defects

- What is the impact of the annual cost of software defects in the US?

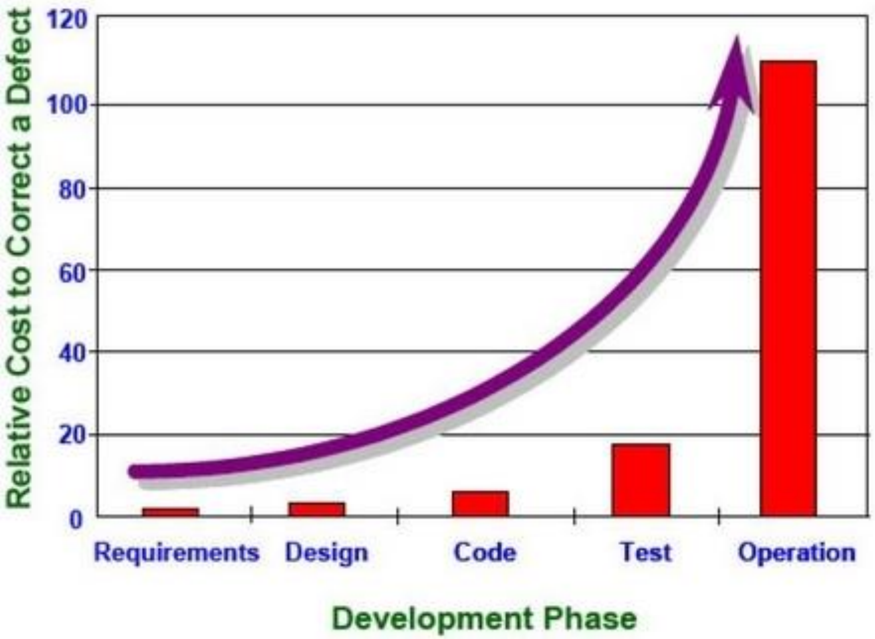
$59 billion

- Estimated that $22 billion could be avoided by introducing a defect detection infrastructure

Source: NIST, The Economic Impact of Inadequate Infrastructure for Software Testing, May 2002

If We Test, Why Still Needs Reviews?

- Defects are much more expensive to fix the later they are discovered
  - Upstream defect removal is 10-100 times cheaper



For example, Bell-Northern Research:

- Inspection cost: 1 hour per defect.
- Testing cost: 2-4 hours per defect.
- Post-release cost: 33 hours per defect

Software Reviews Have Extra Benefits

- Reviews may find errors not possible through testing
  - e.g., conflicting user stories, ambiguous requirements
- Reviews are training
  - disseminate domain knowledge, development skills, corporate standards
- Reviews can assess and improve the quality of
  - Work product
  - Software development process
  - Review process

Family of review techniques

Reviews are technical, not management

Summary of Review Techniques

Method	Typical Goals	Typical Attributes
Walkthroughs	Minimal overhead Developer training Quick turnaround	Informal process Little/no preparation No measurement
Technical reviews	Well-defined requirement Robust design Improvement to code Training	Formal process Author presentation Discussion oriented Review data collected
Inspections	Detect and remove all defects efficiently and effectively	Formal process Checklists Measurements

Source: Johnson P.M. (1996) Introduction to Formal Technical Reviews

Software Walkthroughs

- Involves an author informally showing his/her developed work product(s) to a small audience of peers
  - e.g., a developer gives a small audience a tour of his/her code
- Peers question and comment on the various artefacts as they are presented with the intent to identify defects
  - Mostly involves no prior preparation by the audience
- Usually involves minimal documentation of either the process or any issues that arise
  - There could be minutes indicating actions
  - Defect tracking is inconsistent - sometimes non-existent

- Pair programming can be considered as an extreme style of walkthroughs
  - Instead of periodically reviewing code, code is reviewed all the time by the other person in the pair

Quick Question

During a software walkthrough, a software developer is required to produce which of these work products?

A. Code review summary
B. Requirements document
C. Meeting minutes
D. Nothing

Software Technical review

- Discussed oriented, involving a formal process to improve a software product at a technical level
  - Artefact(s) to be reviewed is/are distributed to a group of the author's peers for (usually) constructive criticism
  - All feedback is collected and collated before some predefined deadline
  - A meeting is held once the collated material is ready
  - The meeting is strictly moderated for delivery of results of review, and answering to criticisms or identified defects
- At the end of a technical review, a certain goal should have been met
  - Forming well-defined requirements
  - Creating more robust designs
  - Making technical improvement to the code, e.g., refactorings to remove code smells and improve the design of existing code
- Significant amounts of data can be collected for the software product's improvement

Requirement Technical Review & Repair

Requirement Technical Review & Repair	
Review stage	• Each reviewer examines the requirements to locate as many defect as possible • Each reviewer classifies their findings into major, moderate, or minor levels of severity
Discussion meeting	• There should be reviewers, the author, a moderator (scrum master) and a recorder • The meeting should be time-boxed; keep discussions on minor issues as minimal as possible • The collected findings will be discussed and potentially reclassified
Repair stage	• The author of the requirements will take the feedback from the review and make adjustments to the requirements • They're not required to address all the suggested changes. Some of the issues raised may not be valid or have already been addressed

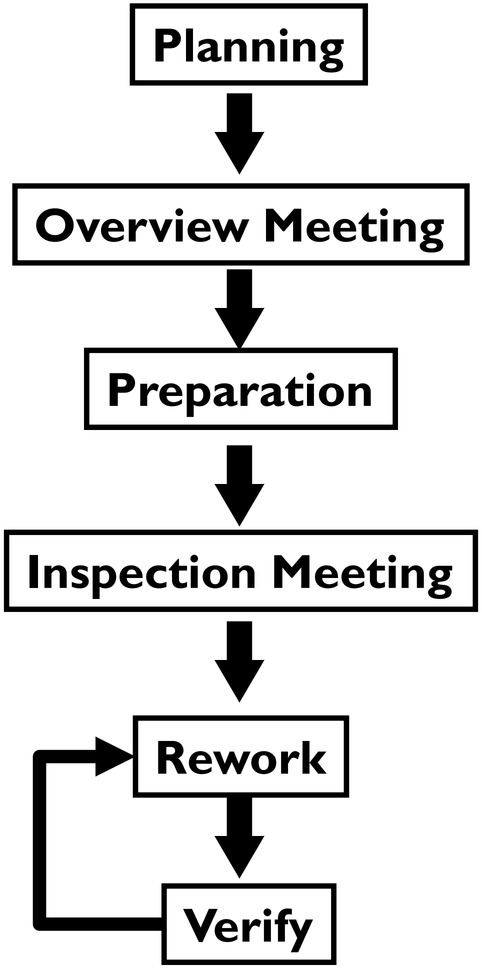
Quick Question

You're performing the review stage for the requirements of another team. You found an issue that one of their requirements was not feasible within their budget, timeline, and resources. You think that they would have to rethink their entire product planning.

What level of severity would you rate this issue?

A. Major
B. Moderate
C. Minor

Inspection (a.k.a Formal Technical Review)



Planning

- The **moderator** makes themselves familiar with the work product that is being reviewed, and any primary areas of concern to the author

Overview Meeting

- All the people filling roles in the review come together and are briefed on the product, process and goals by the moderator

Preparation

- The **inspectors** review the product for defects and make notes, which can take anywhere from a few minutes to a few days

Inspection Meeting

- The **reader** reads through the document in small chunks. The inspectors then make comments on the defects. The **recorder** makes sure to document all of it. The **author** can ask questions and seek clarification, but discussion is not encouraged.

Rework

- The author takes the feedback that they received from the inspectors and makes the necessary changes

Verify

- Insure that changes have been completed, are appropriate, and adjust the defects found in the inspection meeting

Inspection: Do’s and Don’ts

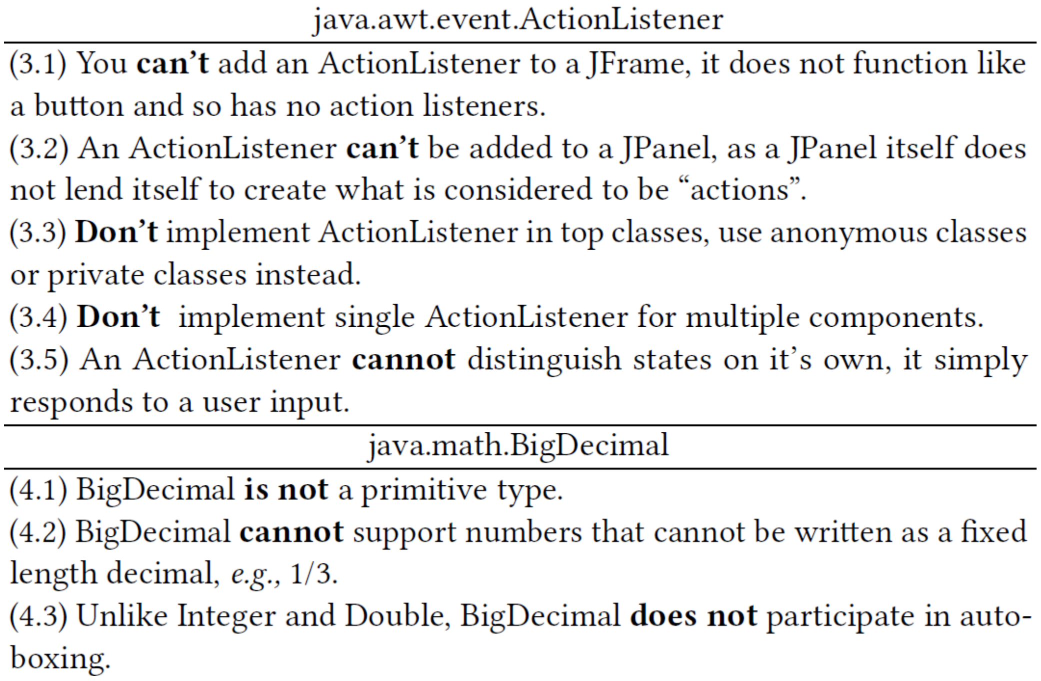
- Don’t author bash!
  - Focus on the defects not vilification of those who produced them
- Do provide adequate preparation time
  - Good preparation helps expose more defects
- Do involve only those who are absolutely necessary in inspection sessions
  - Too many people reduces effectiveness and adds to cost

Code Review

Code Linting

- Code linting is the process of running a tool that will analyse code for potential errors

Emergent, Crowd-Scale Code Liniting



java.awt.event.ActionListener

(3.1) You **can't** add an ActionListener to a JFrame, it does not function like a button and so has no action listeners.

(3.2) An ActionListener **can't** be added to a JPanel, as a JPanel itself does not lend itself to create what is considered to be "actions".

(3.3) **Don't** implement ActionListener in top classes, use anonymous classes or private classes instead.

(3.4) **Don't** implement single ActionListener for multiple components.

(3.5) An ActionListener **cannot** distinguish states on it's own, it simply responds to a user input.

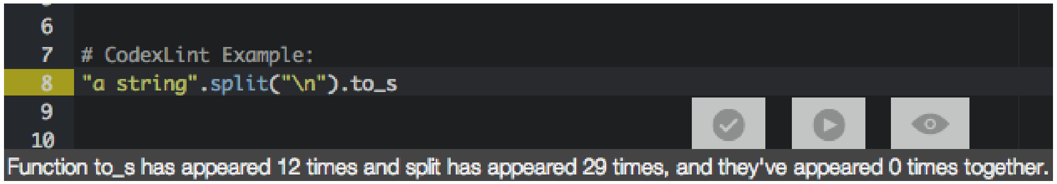
java.math.BigDecimal

(4.1) BigDecimal **is not** a primitive type.

(4.2) BigDecimal **cannot** support numbers that cannot be written as a fixed length decimal, *e.g.*, 1/3.

(4.3) Unlike Integer and Double, BigDecimal **does not** participate in auto-boxing.

To do or not to do: Distilling Crowdsourced Negative API Caveats by Xing et al. – FSE2017



```
# CodexLint Example:
"a string".split("\n").to_s
```

Function to_s has appeared 12 times and split has appeared 29 times, and they've appeared 0 times together.

Crowd-scale programming practices by Fast et al. – CHI2014

Code Reviews Go Beyond Simply Looking For Defects

- Defect-finding is still the main reason for code reviews, but in descending order of importance code reviews can include:
  - Code improvement, e.g., **refactorings**
  - Alternative solutions
  - Knowledge transfer
  - Team awareness
  - Improving development process
  - Sharing code ownership, e.g., **pair programming**
  - Avoiding build breaks
  - Track rationale
  - Team assessment

Source: Expectations, Outcomes, and Challenges of Modern Code Review by Bacchelli and Bird – ICSE 2013

Pair Programming

Recall Pair Programming

- This takes code review to the extreme
  - Two developers work side by side at one computer to develop code
- But personality conflicts will play a huge factor

Agile in Practice: Pair Programming by Agile Academy: https://www.youtube.com/watch?v=ET3Q6zNK3Io

You will practice pair programming in the Laboratory 1 in the Week 6.

Two Roles in Pair Programming

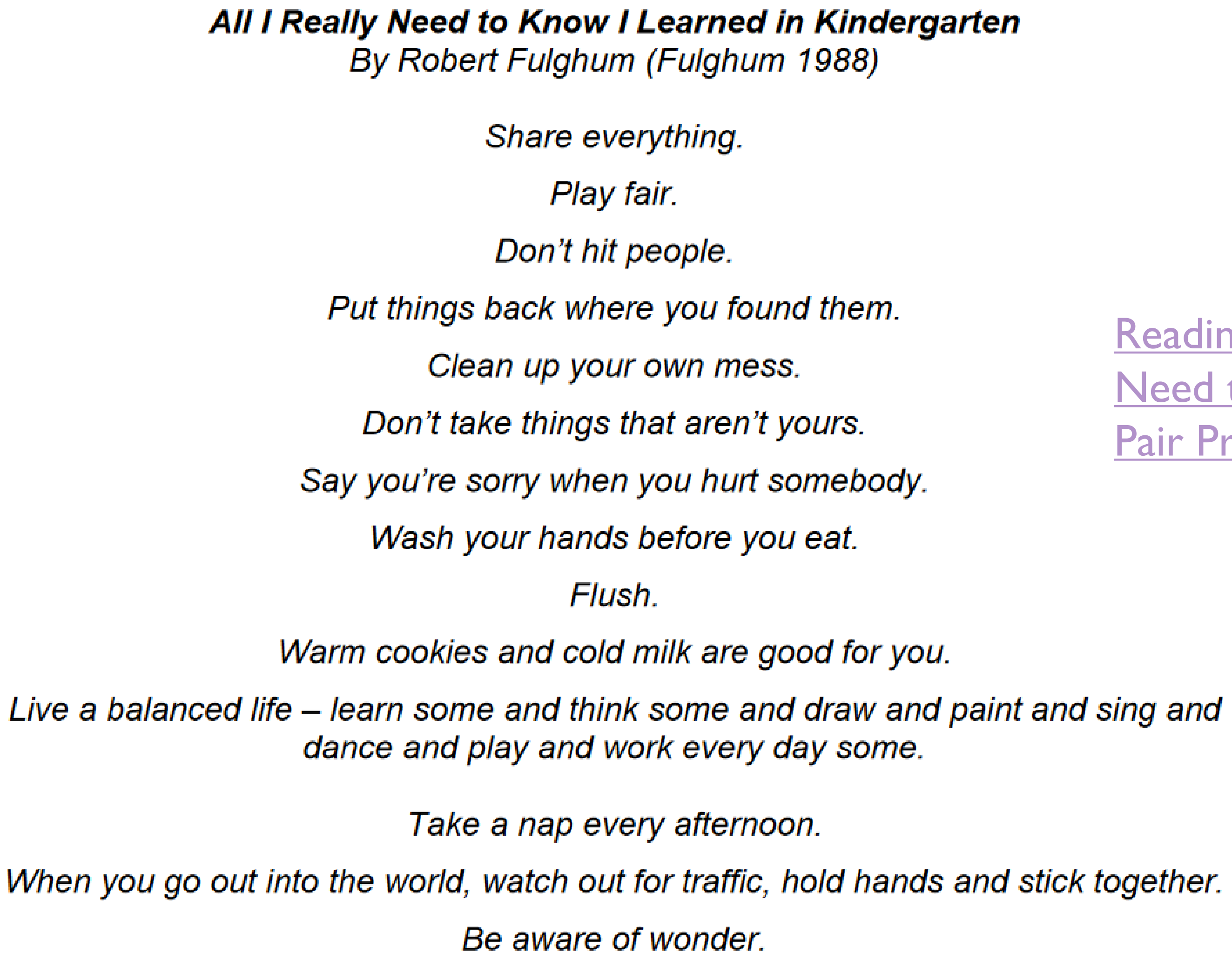
- The driver
  - responsible for typing, moving the mouse, etc.
  - be the only one entering the code
  - accepts feedback and direction from the navigator
- The navigator
  - responsible for reviewing the driver's work and catching mistakes (but not trivial ones like misspellings)
  - considers the code at a more strategic level
    - How will this fit the rest of the code?
    - Will it require changes elsewhere?
    - Could we design this program better?
    - How to test it?

How to Pair Program

- Every 15 minutes or so, the pair switches roles
  - Both partners spend an equal amount of time in each role

- It is not a divide-and-conquer approach
  - Do not beak down the task into smaller pieces, and each works on “her/his part” separately

All You Need to Do is to Follow What You Learned in Kindergarten ;-)



All I Really Need to Know I Learned in Kindergarten
By Robert Fulghum (Fulghum 1988)

Share everything.

Play fair.

Don't hit people.

Put things back where you found them.

Clean up your own mess.

Don't take things that aren't yours.

Say you're sorry when you hurt somebody.

Wash your hands before you eat.

Flush.

Warm cookies and cold milk are good for you.

Live a balanced life – learn some and think some and draw and paint and sing and dance and play and work every day some.

Take a nap every afternoon.

When you go out into the world, watch out for traffic, hold hands and stick together.

Be aware of wonder.

Reading 4.1 All I Really Need to Know about Pair Programming

Code Smells and Refactorings

Why do We Need to Maintain Software?

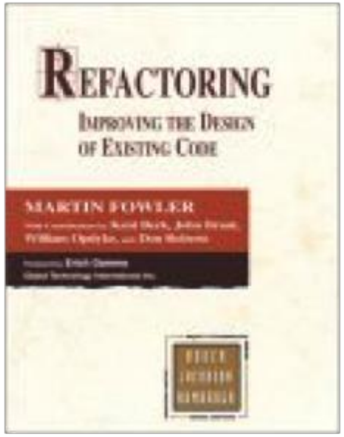
- Changing requirements
  - Changing a software system to satisfy new or revised requirements
- Fix bugs
  - Changing a software system to fix discovered bugs
- New environment
  - Changing a software system so that it can operate in a different environment (smart phones, web application, etc.)
- Refactoring
  - Restructuring code to improve its maintainability

Software Maintenance - Classification

- Perfective
  - Implementing new or revised requirements
- Corrective
  - Diagnosing and fixing errors
- Adaptive
  - Coping with a changed environment
- Preventive
  - Improving supportability (FURPS+), e.g., maintainability

Refactoring is a Preventive Maintenance Activity

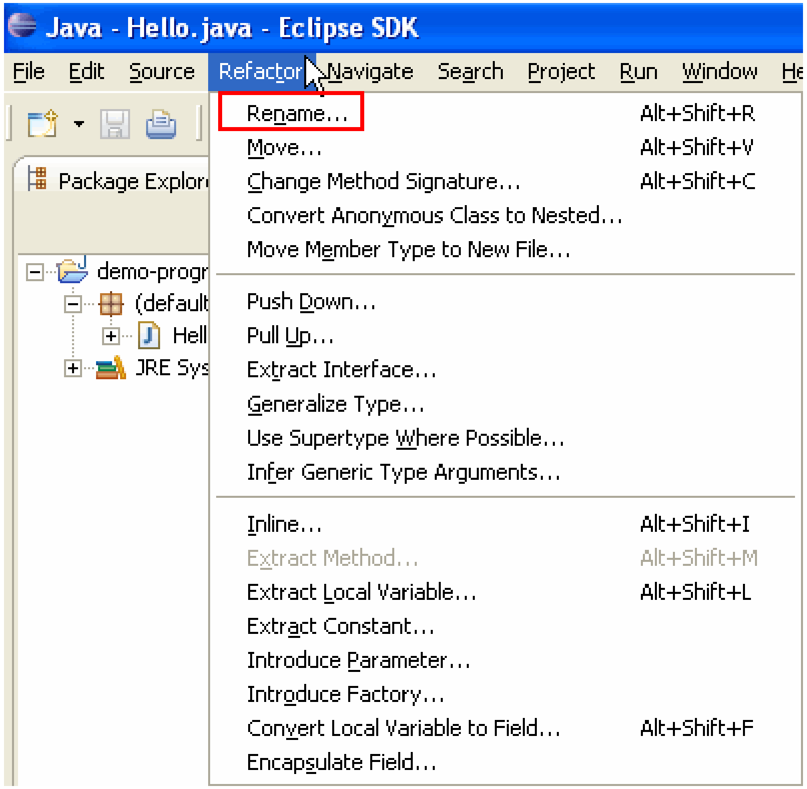
- Martin Fowler, 1999



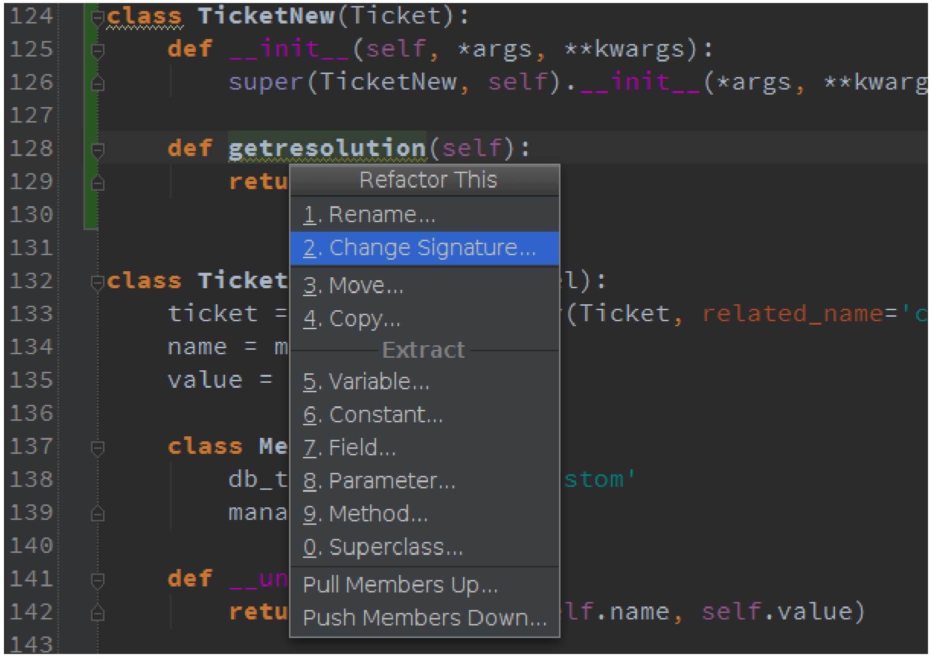
Change a software system so that the external behavior does not change but the internal structure is improved

You will practice refactoring and test-driven development in the Laboratory 2 in Week 9.

Refactorings Support in Modern IDEs


Java - Hello.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window He
Package Explor
demo-progr
(default
Hell
JRE Sys
Rename... Alt+Shift+R
Move... Alt+Shift+V
Change Method Signature... Alt+Shift+C
Convert Anonymous Class to Nested...
Move Member Type to New File...
Push Down...
Pull Up...
Extract Interface...
Generalize Type...
Use Supertype Where Possible...
Infer Generic Type Arguments...
Inline... Alt+Shift+I
Extract Method... Alt+Shift+M
Extract Local Variable... Alt+Shift+L
Extract Constant...
Introduce Parameter...
Introduce Factory...
Convert Local Variable to Field... Alt+Shift+F
Encapsulate Field...

We prepare the refactoring demos using the Eclipse IDE for the refactorings that are discussed in the lecture.



```
class TicketNew(Ticket):
    def __init__(self, *args, **kwargs):
        super(TicketNew, self).__init__(*args, **kwarg

    def getresolution(self):
        retu

class Ticket                    l):
    ticket =                    (Ticket, related_name='c
    name = m
    value =

    class Me
        db_t                    stom'
        mana

    def __un
        retu                    lf.name, self.value)
```

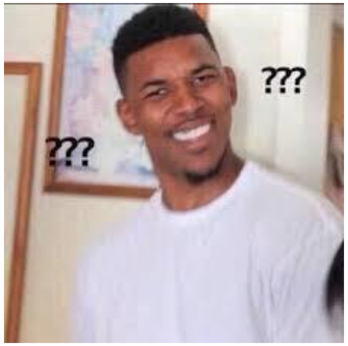
When (Code Smells) and How (Refactorings)

“If it stinks, change it”

Improve existing code in small steps

Is Guessing Fun?

- Code smells
  - Single letter variables
  - Be abstract and vague
  - Magic number

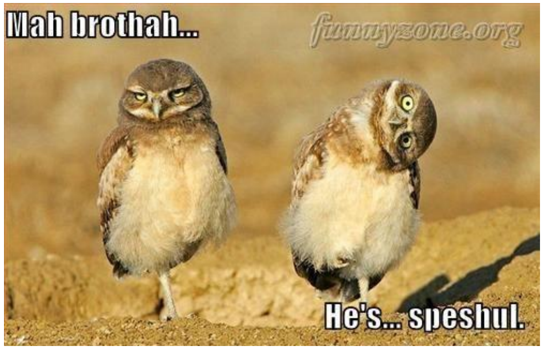


- Refactorings
  - Rename class, field, method, etc.
  - Replace magic number with symbolic constant (also know as Extract constant)

Is it as simple as it seems to be? Can you ensure the consistent update of all places?

Do You Have a Twin Brother?

- Code smells – code duplication
  - e.g., similar computation in several methods of the same class
  - e.g., similar computation in several sibling subclasses



- Refactorings
  - For duplicated methods in the same class, extract method to extract the duplication into a new method
  - For duplication in sibling subclasses, pull up method/field to the superclass, or if no superclass, extract superclass

How to deal with subclass specific behaviours?

Do You Inherit Too Much?

- Code smell
  - A subclass uses only part of a superclass, but inherits unwanted methods

Override all unwanted methods.
Problem solved?

- Refactoring
  - Replace inheritance with delegation – remove the subclassing, create a field for the superclass, adjust methods to delegate to the field

Try out the Java Swing program AddSomethingtoJButton: can you add a JPanel to a JButton?

Hint: JPanel is a subclass of javax.swing.JComponent which extends java.awt.JContainer

Do You Do Too Much?

- Code smells
  - Large class: a class ends up with too many responsibilities
  - Long method: a method ends up with too many responsibilities

- Refactorings
  - Apply Extract Class to factor out some related set of attributes and methods
  - Apply Extract Method to factor out some portion of method body

Do You Ask Too Much?

- Code smell
  - Feature envy: a method or part of a method seems more interested in the details of a class other than the one it is actually defined in
- Refactorings
  - If the whole method clearly wants to be elsewhere, move method
  - If only part of the method is envious, extract method and then move method

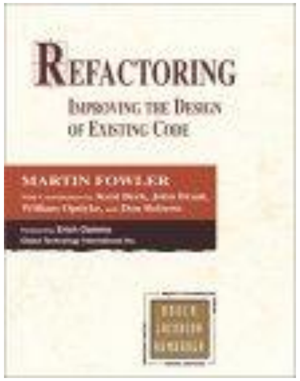
Refactoring + Unit Testing

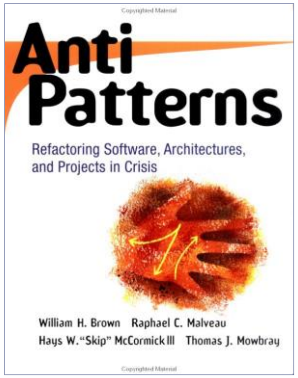
- Refactorings change working code, you run the risk of breaking it

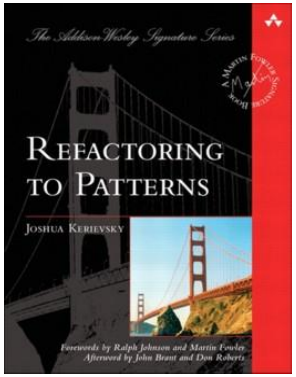
- Unit testing can save your neck
  - always "compile and test"

Refactoring Resources

- Refactoring
  - M. Fowler et al., 1999
- AntiPatterns
  - W.J. Brown et al., 1998
- Refactoring to Patterns
  - J. Kerievsky, 2005




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


The Addison-Wesley Signature Series
REFACTORING TO PATTERNS
JOSHUA KERIEVSKY
Forewords by Ralph Johnson and Martin Fowler
Afterword by John Brant and Don Roberts

- Refactoring Home Page
  - http://www.refactoring.com/
- How to Write Unmaintainable Code
  - https://github.com/Droogans/unmaintainable-code