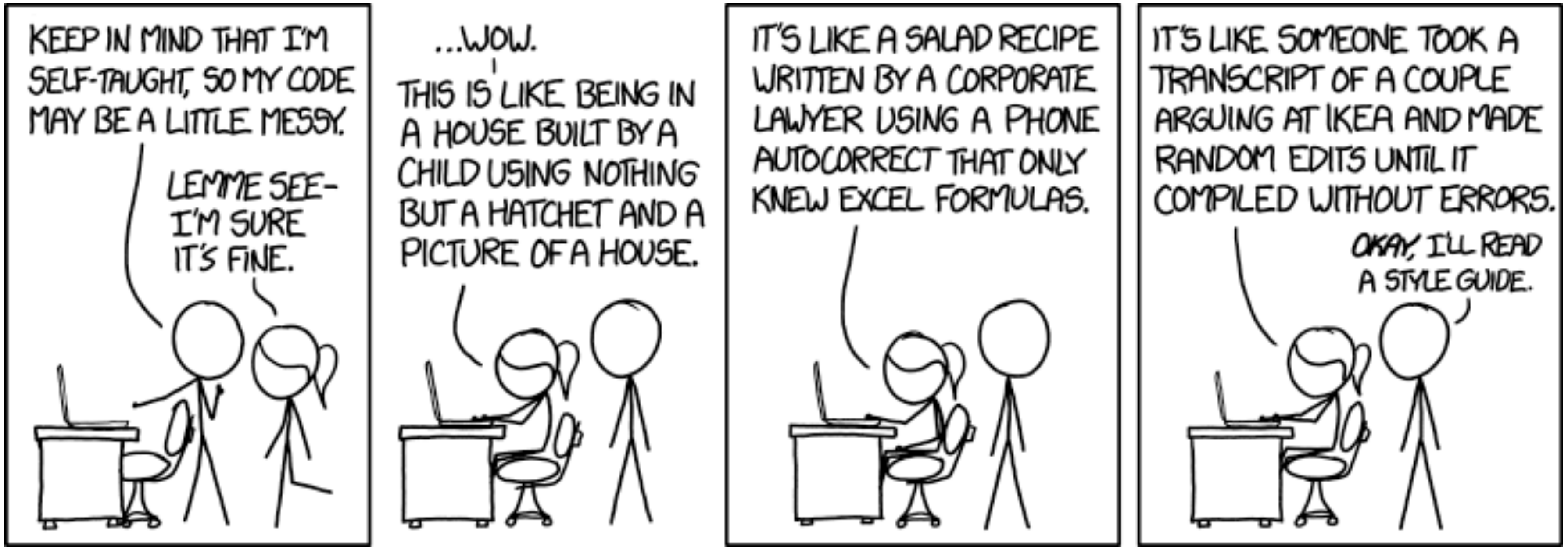

KEEP IN MIND THAT I'M SELF-TAUGHT, SO MY CODE MAY BE A LITTLE MESSY.
LEMME SEE— I'M SURE IT'S FINE.
...WOW.
THIS IS LIKE BEING IN A HOUSE BUILT BY A CHILD USING NOTHING BUT A HATCHET AND A PICTURE OF A HOUSE.
IT'S LIKE A SALAD RECIPE WRITTEN BY A CORPORATE LAWYER USING A PHONE AUTOCORRECT THAT ONLY KNEW EXCEL FORMULAS.
IT'S LIKE SOMEONE TOOK A TRANSCRIPT OF A COUPLE ARGUING AT IKEA AND MADE RANDOM EDITS UNTIL IT COMPILED WITHOUT ERRORS.
OKAY, I'LL READ A STYLE GUIDE.

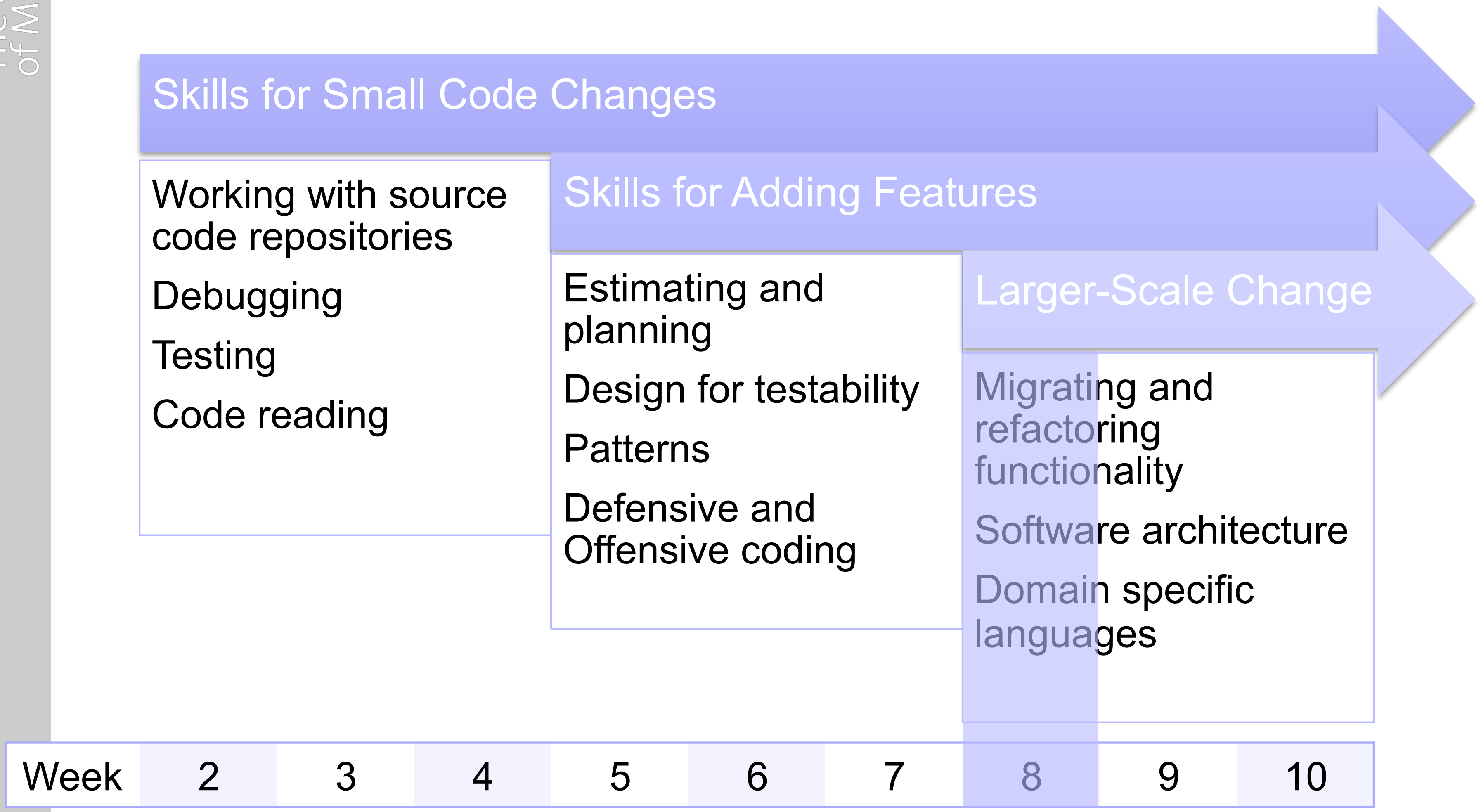
Refactoring and code migration

COMP23420: Software Engineering

Week 8

Caroline Jay and Robert Haines

Course Unit Roadmap (Weeks 2-10)



Skills for Small Code Changes

- Working with source code repositories
- Debugging
- Testing
- Code reading

Skills for Adding Features

- Estimating and planning
- Design for testability
- Patterns
- Defensive and Offensive coding

Larger-Scale Change

- Migrating and refactoring functionality
- Software architecture
- Domain specific languages

Week	2	3	4	5	6	7	8	9	10

Link to the Coursework

- We will discuss refactoring and introduce tools in Eclipse to help refactor code.
- We will also start thinking about how to approach the final coursework exercise.

What is refactoring?

- “*Refactoring changes structure, not behaviour*”
- What do we mean by “behaviour”?
  - The function that is performed by the code
  - The interface that the code presents to the world
    - API
    - More obvious in object-oriented programming

Testing and refactoring

- "*Refactoring changes structure, not behaviour*"
  - Behaviour must be *exactly* the same after refactoring
- Good tests are essential for refactoring
  - Must start and finish in known good state
- Refactoring workflow:
  - Run tests (start from known good state)
  - Refactor
  - Run tests (finish in the same known good state)

Changing interfaces vs refactoring

- *“What if I split a complex method up into smaller private methods?”*
- This is a refactor; the interface is not changed
  - Run tests to ensure they pass
  - Refactor
  - Re-run entire test suite to check for regressions

Changing interfaces vs refactoring

- *“What if I combine similar functionality from two private methods into a single one?”*
- This is a refactor; the interface is not changed
  - Run tests to ensure they pass
  - Refactor
  - Re-run entire test suite to check for regressions

Changing interfaces vs refactoring

- *"What if I am removing a public method, and no longer need the test?"*
- Not a refactor; you are changing the interface
  - Run tests to ensure they pass
  - Remove the test
  - Remove the method
  - Re-run entire test suite to check for regressions

Changing interfaces vs refactoring

- "*What if I am moving a public method between two classes?*"

- Not a refactor; you are changing the interface
  - Run tests to ensure they pass
  - Change the test so that it tests the code in its new location
  - Move the code to the new class
  - Re-run entire test suite to check for regressions

How do I know when to refactor?

- Look out for

- Complexity
  - Assignments, branches, calls (ABC)
  - Cyclometric complexity
  - Consider refactoring high scoring methods
- Structural similarity
  - Consider refactoring similar code

- Don’t search by hand, use tools

Refactoring with your IDE (Eclipse)

- Modern IDEs have shortcuts to help refactoring
  - Use them!
  - Auto-update changed references
    - Across whole project
    - Across all files
    - Even in the unit tests
- For Eclipse, see: http://help.eclipse.org/

Simple tasks

- Rename…; Move…
  - Fields, local variables, types, packages, etc
- Change method signature…
  - Keep original as delegate
  - Deprecate original
- Encapsulate field…
  - Replaces all references to a field with getter and setter methods

Super-type/sub-type operations

- Use super-type where possible…
  - Replace occurrences of a type with one of its super-types where possible
- Pull up…
  - Move a field/method to a superclass
  - Declare the method abstract in the superclass
    - Methods only
- Push down…
  - Move a set of methods and fields from a class to its subclasses

Extracting structure

- Extract local variable…
  - Creates a new variable assigned to the current selection
  - Replaces the selection with a reference to the new variable
- Extract method…
  - Creates a new method containing the current selection
  - Replaces the selection with a reference to the new method
  - Useful for refactoring lengthy, cluttered, or overly-complex methods.

Extracting structure

- Extract superclass…
  – Extracts a common superclass from a set of sibling types
  – The selected sibling types become direct subclasses of the extracted superclass
  – Maybe re-run “Use super-type where possible…”
- Extract interface…
  – Creates a new interface with a set of methods
  – Makes the selected class implement the interface

Stendhal Project

- Team Coursework 3: Cross-Cutting Change for Non-Functional Requirements.

Understanding the problem

What is the goal?

- Make defining quests more declarative and simple

Why is it important?

- Extensibility
- Maintainability

Understanding the problem

Step 1: Look at the existing system

- What are the common features of collection and paper-chase style quests?
- What are the configurable features of collection and paper-chase style quests?
  - How might those features be configured (e.g. data type)?

Planning a solution

- Which approach should you use?
  - Extending the Java classes
  - Using XML
  - Using Groovy scripts

- How would you represent the common and configurable features in each case?
- Which one are you going to use?
  - Decision
  - Rationale

Next week

- In the team study sessions you will continue to work on the coursework and you will have your second mentoring session
- In the workshop we will learn about different styles of software architecture.