

Qualidade de Software (14450)

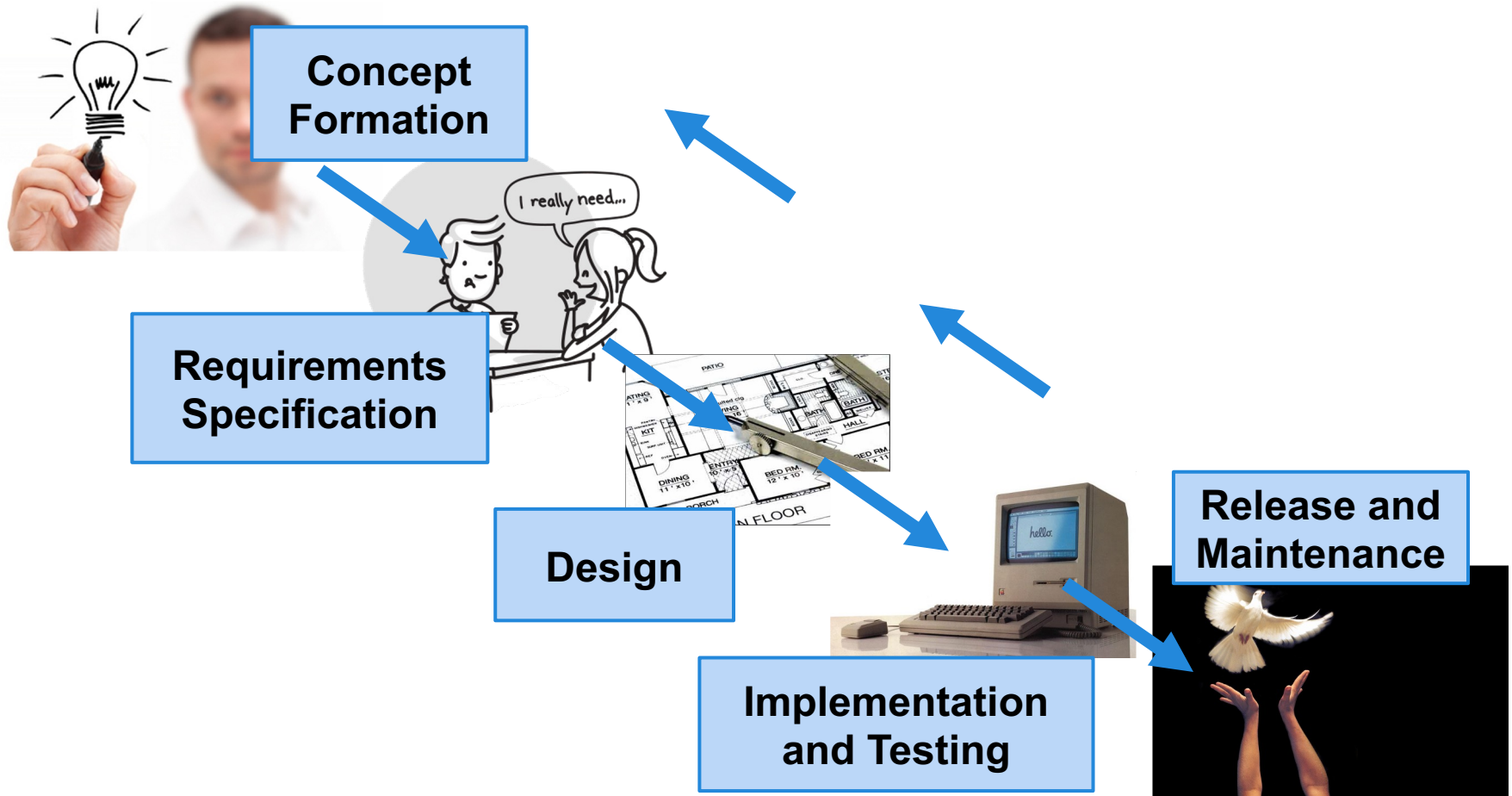
Refactoring

(adapted from lecture notes of the “DIT 635 - Software Quality and Testing” unit,
delivered by Professor Gregory Gay, at the Chalmers and the University of Gothenburg, 2022)

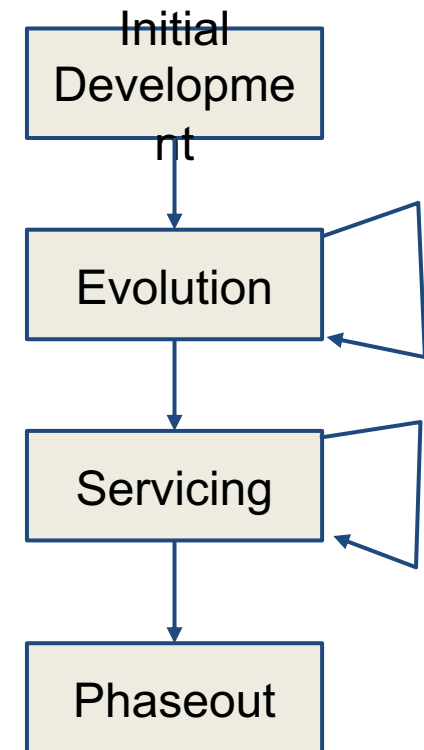
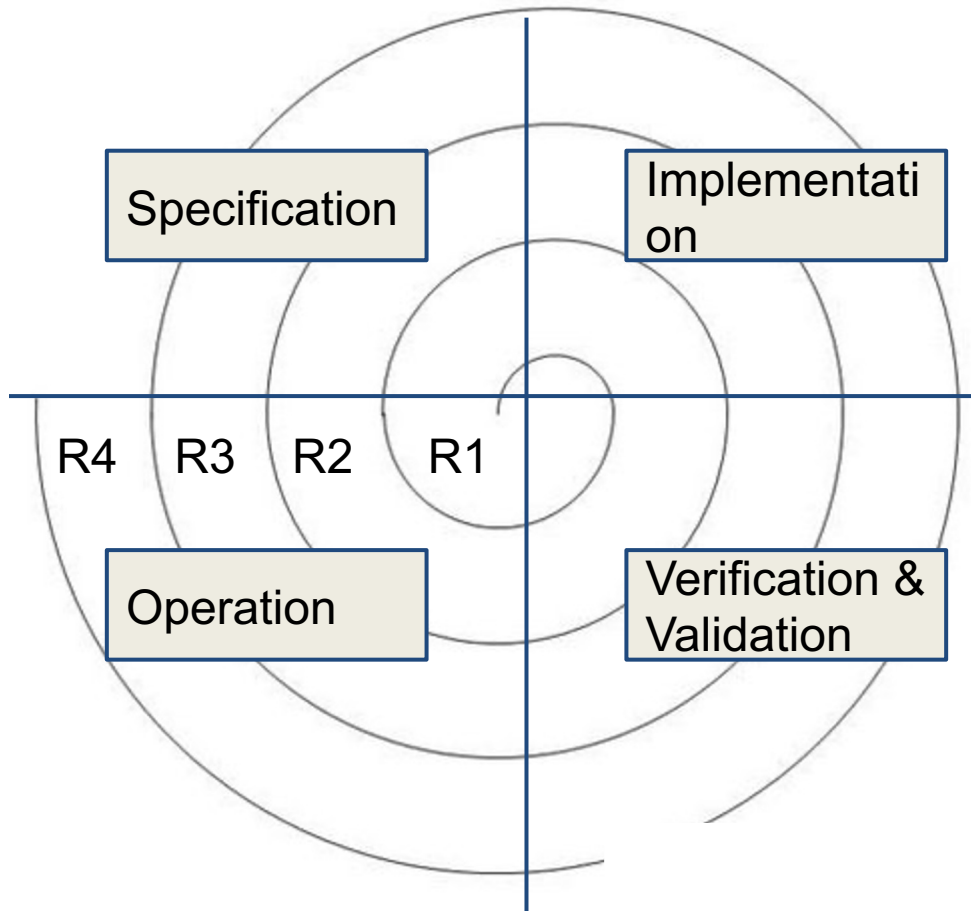
Today's Goals

- ✧ Cover the basics of refactoring
- ✧ Introduce the idea of “code smells”

The Software Lifecycle



The Real Lifecycle



Software Maintenance

- **Fault Repairs**
 - Changes made in order to correct coding, design, or requirements errors.
- **Environmental Adaptations**
 - Changes made to accommodate changes to the hardware, OS platform, or external systems.
- **Functionality Addition**
 - New features are added to the system to meet new user requirements.

Software Maintenance Effort

- Maintenance costs more than the initial development.
 - 2/3rds of budget goes to maintenance on average.
 - Up to four times the development cost to maintain critical systems.
- General breakdown:
 - 65% of effort goes to functionality addition
 - 18% to environmental adaptation
 - 17% to fault repair

Maintenance is Hard

It is harder to maintain than to write new code.

- Must understand code written by another developer, or code that you wrote long ago.
- Creates a “house of cards” effect.
- Developers tend to prioritize new development.

Smooth maintenance requires planning and design that supports maintainability.

The Laws of Software Evolution

- Maintenance is an inevitable process.
 - Requirements change as the environment changes.
 - Changing the software causes environmental changes, which leads to more requirement changes.
- As changes occur, the structure degrades.
 - When changes are made, the structure becomes more complex.
 - To prevent this, resources must go into *preventative maintenance* - refactoring to preserve and simplify the structure without adding to functionality.

The Laws of Software Evolution

- The amount of change in each release is approximately constant.
 - The more functionality introduced, the more faults.
 - A large functionality patch tends to be followed by a patch that fixes faults without adding additional functionality. Small functionality changes do not require a fault-correcting patch.
- Functionality must continually increase to maintain user satisfaction.

The Laws of Software Evolution

- The quality of the system will decline unless updated to work with changing environment.
- To improve quality, evolution must be treated as a feedback system.
 - Stakeholders must be continually involved in evolution, and changes should be influenced by their needs.

Refactoring

- Process of revising the code or design to improve its structure, reduce complexity, or otherwise accommodate change.
- When refactoring, you do not add functionality.
- Continuous process of improvement throughout the evolution of the system.

Why Refactor?

Why fix what isn't broken?

- Components have three purposes:
 - To perform a service.
 - To allow change.
 - To be understood by developers reading it.
- If it does not do any of these, it is “broken”.
- Enables change and improves understandability.

Refactoring is an Iterative Process

- Refactoring should take place as an iterative cycle of small transformations.
 - Choose a small part of the system, redesign it, and make sure it still works.
 - Choose a new section of the system and refactor it.
- Refactoring requires unit tests.
 - Make sure the code works before and after.

Choosing What to Refactor

- Refactor any piece of the system that:
 - Seems to work,
 - But isn't well designed,
 - And now needs new functionality.
- There are stereotypical situations that indicate the need for refactoring.
 - These are called “**bad smells**”.

Code Smells

- **Code is duplicated** in multiple places.
- A method is **too long**.
- Conditional statements control behavior based on an **object type**.
- Groups of data **attributes are duplicated**.
- A class has **poor cohesion** or **high coupling**.
- A method has **too many parameters**.
- **Speculative generality** - adding functionality that “we might need someday.”

More Code Smells

- Changes must be made in **several places**.
- **Poor encapsulation** of data that should be private.
- If a **weak subclass** does not use inherited functionality.
- If a class contains **unused code**.
- If a class contains **potentially unused attributes** that are only set in particular circumstances.
- There are data classes containing only attributes, getters, and setters, but nothing else - **objects should encapsulate data and behaviors**.
 - Unless that data is used by multiple classes.

Common Refactorings

(more at <http://www.refactoring.com>)

Composing Methods

- Extract Method
- Inline Method; Inline Temp
- Introduce Explaining Variable
- Split Temporary Variable
- Remove Assignments to Parameters
- Substitute Algorithm

Moving Features Between Objects

- Move Method; Move Field
- Extract Class
- Inline Class
- Hide Delegate
- Remove Middleman
- Introduce Foreign Method

Organizing Data

- Replace Data Value with Object
- Change Value to Reference; Change Reference to Value
- Replace Array with Object
- Duplicate Observed Data
- Change Unidirectional Association to Bidirectional
- Change Bidirectional Association to Unidirectional

Simplifying Conditional Expressions

- Decompose Conditional
- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Replace Conditional with Polymorphism
- Introduce Null Object
- Introduce Assertion

Making Method Calls Simpler

- Rename Method
- Add/Remove Parameter
- Separate Query from Modifier
- Parameterize Method
- Replace Parameter with Explicit Methods
- Preserve Whole Object
- Replace Parameter with Method
- Introduce Parameter Object
- Remove Setting Method
- Hide Method
- Replace Constructor with Factory Method
- Encapsulate Downcast
- Replace Error Code with Exception
- Replace Exception with Test

Dealing with Generalization

- Pull Up Field; Method; Constructor Body
- Push Down Method; Push Down Field
- Extract Subclass; Extract Superclass; Interface
- Collapse Hierarchy
- Form Template Method
- Replace Inheritance with Delegation (or vice versa)

Big Refactorings

- Nature of the Game
- Tease Apart Inheritance
- Convert Procedural Design to Objects
- Separate Domain from Presentation
- Extract Hierarchy

Refactorings - Composing Methods

- If you have a complex code fragment that can exist independently, **extract it into its own method**.
- If you have a method that is extremely simple, **inline** it into locations where it is used.
- If you assign values to a temporary variable more than once, **split it into additional temporary variables**.
- If assignments are made to parameter variables in a method, instead **assign to a temporary variable**.
- If an algorithm is hard to understand, **swap it for a version that is clearer**.

Refactorings - Moving Features Between Objects

- If a method or field is used more by a calling class than the class it is placed in, **move** it.
- If a class is doing more work than it should (or has low cohesion), **extract** a subset of related methods into a new class.
- If a class is doing too little, **combine** it with another.
- If a class delegates too many calls to a middleman class, **get rid of the middleman** and call the client directly.
- If an imported class needs an additional method, but you can't modify it directly, **create a method in the client class with the imported object as a parameter.**

Refactorings - Conditional Expressions & Data

- If your conditional statements are too complex, **extract methods from the if, then, and else conditions.**
- If you have a sequence of conditional tests with the same result or repeated conditions in each branch, **consolidate them into fewer conditional statements.**
- If you have conditional statements to choose behavior based on object type, instead **use polymorphism.**
- If you have an attribute that needs additional data or operations, turn it into a new type of **data object.**
- If certain array values have special meaning, use **a class to store items instead.**

Refactorings - Simplifying Method Calls and Generalization

- If a method both returns a value and changes the state of a passed object, split into two methods and **separate the query from the modifier**.
- If several methods do similar things - differentiated by value - create one method that takes the **value as a parameter**.
- If two classes have the same attribute/method/constructor body, **pull it up into the parent**. If an item is only used by some subclasses, **push it into the children**.
- If a class has features only used situationally, **extract subclasses** for those situations.

Dangers of Refactoring

- Code that used to be well commented, well tested, and fully reviewed might not be any of these things after refactoring.
- You might have inserted faults into code that previously worked.
 - This is why unit tests are important. If the new code is broken, revert back to the old code.
- What if the new design is not better?

“I Don’t Have Time”

- Most common excuse for not refactoring.
- Refactoring incurs an up-front cost.
 - Developers don’t want to do it.
 - Neither do managers - they lose time and get “nothing” (no new features)
- Small companies (start-ups) avoid it.
 - “We can’t afford it.” “We don’t need it.”
- So do large companies.
 - “We’d rather add new features.”
 - “No one gets promoted for refactoring.”

“I Don’t Have Time”

- Refactoring is the key to effective evolution.
 - Enables rapid addition of new features, with fewer faults (up to a 500% ROI).
 - Good for programmer morale.
- Refactoring is an investment in a company’s prime asset - its code base.
- Many start-ups use cutting-edge tech and agile processes that evolve rapidly. So should the code.
- Some of the most successful companies (Google) reward and require refactoring.

