Software Engineering (Part 4)

Robert M. Dondero, Ph.D.
Princeton University

Objectives

- We will cover these software engineering topics:
  - Requirements analysis
  - Design
  - Implementation
  - Debugging
  - Testing
  - Evaluation
  - Maintenance
  - Process models

Stages of SW dev

How to order the stages

You've tested your code to make sure it meets your expectations. What's next?

Agenda

- Requirements analysis
- Design
- Implementation
- Debugging
- Testing
- **Evaluation**
- Maintenance
- Process models

Evaluation

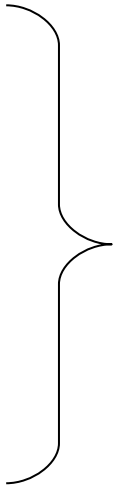
- **Testing**
  - Does the system meet **your (the programmer's)** expectations?
- ***Evaluation***
  - Does the system meet **the users'** expectations?
  - Does the system fulfill the needs of its users?

Evaluation

- Kinds of evaluation
  - By users
    - Actually, by software engineers in collaboration with users
  - By evaluation experts

Evaluation: Users

- Questionnaires
- **Interviews**
- Focus groups
- Direct observation



Recall requirements gathering techniques

Evaluation: Users

1. Recruit a set of users.
2. If necessary, compose a short written intro.
3. Compose a task sequence (maybe abstracted from use cases developed during design).
4. For each user:
   - 4.1. If necessary, give the user the short intro, ask the user to read it, and confirm that the user understands it.
   - 4.2. Give the user the task sequence.
   - 4.3. For each task:
     - 4.3.1. Ask the user to read the task and confirm that the user understands it.
     - 4.3.2. Ask the user to use your system to perform the task.
     - 4.3.3. Ask (force!!!) the user to talk aloud while performing the task.
5. Take copious notes.
6. Audio/video record?
7. Repeat for each kind of user.

Evaluation: Experts

Jakob Nielsen

Evaluation: Experts

- ***Heuristic Evaluation***
  - From Jakob Nielsen
  - For evaluating the **whole** system **generally**
  - Using these 10 heuristics…

Evaluation: Experts

- ***Heuristic Evaluation***
  - **(1) Visibility of system status**
    - **The system should always keep users informed** about what is going on, through appropriate feedback within reasonable time.

Nielsen, Jakob. *Usability Engineering.* Academic Press. 1994.

Evaluation: Experts

- ***Heuristic Evaluation***
  - **(2) Match between system and the real world**
    - **The system should speak the user's language**, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.

Nielsen, Jakob. *Usability Engineering.* Academic Press. 1994.

Evaluation: Experts

- ***Heuristic Evaluation***
  - **(3) User control and freedom**
    - **Users often choose system functions by mistake and will need a clearly marked "emergency exit"** to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.

Nielsen, Jakob. *Usability Engineering.* Academic Press. 1994.

Evaluation: Experts

- ***Heuristic Evaluation***
  - **(4) Consistency and standards**
    - Users should not have to wonder whether different words, situations, or actions mean the same thing. **Follow platform conventions.**

Nielsen, Jakob. *Usability Engineering.* Academic Press. 1994.

Evaluation: Experts

- ***Heuristic Evaluation***
  - **(5) Error prevention**
    - **Even better than good error messages is a careful design which prevents a problem from occurring** in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action.

Nielsen, Jakob. *Usability Engineering.* Academic Press. 1994.

Evaluation: Experts

- ***Heuristic Evaluation***
  - **(6) Recognition rather than recall**
    - Minimize the user's memory load by **making objects, actions, and options visible**. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.

Nielsen, Jakob. *Usability Engineering.* Academic Press. 1994.

Evaluation: Experts

- ***Heuristic Evaluation***
  - **(7) Flexibility and efficiency of use**
    - Accelerators—unseen by the novice user—may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. **Allow users to tailor frequent actions.**

Nielsen, Jakob. *Usability Engineering.* Academic Press. 1994.

Evaluation: Experts

- ***Heuristic Evaluation***
  - **(8) Aesthetic and minimalist design**
    - **Dialogues should not contain information which is irrelevant** or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.

Nielsen, Jakob. *Usability Engineering.* Academic Press. 1994.

Evaluation: Experts

- ***Heuristic Evaluation***
  - **(9) Help users recognize, diagnose, and recover from errors**
    - **Error messages should be expressed in plain language** (no codes), precisely indicate the problem, and constructively suggest a solution.

Nielsen, Jakob. *Usability Engineering.* Academic Press. 1994.

Evaluation: Experts

- ***Heuristic Evaluation***
  - **(10) Help and documentation**
    - Even though it is better if the system can be used without documentation, it may be necessary to **provide help and documentation**. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

Nielsen, Jakob. *Usability Engineering.* Academic Press. 1994.

Evaluation: Experts

- For more info on heuristic evaluation:
  - Wikipedia article: https://en.wikipedia.org/wiki/Heuristic_evaluation
  - Helen Sharp, Jenny Preece, Yvonne Rogers. *Interaction Design: Beyond Human-Computer Interaction*.
  - Nielsen, Jakob. *Usability Engineering*.

Evaluation: Experts

- ***Cognitive Walkthrough***
  - From Cathleen Wharton, Jakob Nielsen
  - For evaluating **part** of the system in **detail**

Repeatedly:
Will the correct action be sufficiently evident to the user?
Will the user know what to do to achieve the task?
Will the user notice that the correct action is available?
Can users see the button or menu item that they should use for the next action?
Will the user associate and interpret the response from the action correctly?
Will users know from the feedback that they have made the correct or incorrect choice of action?

Yvonne Rogers, Helen Sharp, Jenny Preece. *Interaction Design: Beyond Human-Computer Interaction (3rd Edition)*. Wiley, 2011.

So the system is finished. Or is it?

Agenda

- Requirements analysis
- Design
- Implementation
- Debugging
- Testing
- Evaluation
- **Maintenance**
- Process models

Maintenance

- ***Maintenance***
  - How can I ensure that the system continues to fulfill the users' needs through time?
  - Better term: ***continuance***

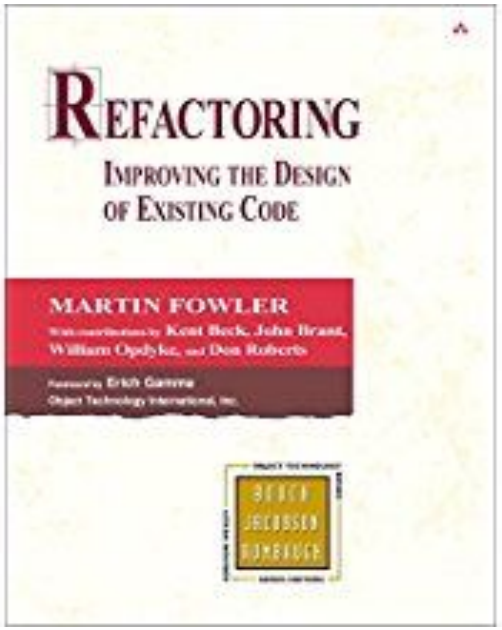
Maintenance

Rod Stephens.
Beginning Software Engineering.
Wiley. 2015

- ***Perfective*** maintenance
  - Add new features, improve (performance of) existing features
- ***Adaptive*** maintenance
  - Modify the system to meet changes in its environment
- ***Corrective*** maintenance
  - Fix bugs
- ***Preventive*** maintenance
  - **Refactor code** to make it more maintainable

Maintenance: Refactoring

Martin Fowler



2000

Maintenance: Refactoring

- ***Bad smells*** in code

Martin Fowler.
Refactoring: Improving the Design of Existing Code.
Addison-Wesley. New York. 2000.

Duplicated code
Long method
Long parameter list
Divergent change
Shotgun surgery
Feature envy
Data clumps
Primitive obsession
Switch statements
Parallel inheritance hierarchies
Lazy class

Speculative generality
Temporary field
Message chains
Middle man
Inappropriate intimacy
Alternative classes with diff interfaces
Incomplete library class
Data class
Refused bequest
Comments

Maintenance: Refactoring

1. Composing methods (9)
- Extract method
- Inline method
- Inline temp
- Replace temp with query
- Introduce explaining variable
- Split temporary variable
- Remove assignments to parameters
- Replace method with method object
- Substitute algorithm

2. Moving features between objects (8)
- Move method
- Move field
- Extract class
- Inline class
- Hide delegate
- Remove middle man
- Introduce foreign method
- Introduce local extension

Martin Fowler.
Refactoring: Improving the Design of Existing Code.
Addison-Wesley. New York. 2000.

Maintenance: Refactoring

3. Organizing data (16)
- Self encapsulate field
- 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
- Replace magic number with symbolic constant
- Encapsulate field
- Encapsulate collection
- Replace record with data class
- Replace record with class data
- **Replace type code with subclasses**
- Replace type code with state/strategy
- Replace subclass with fields

4. Simplifying conditional expressions (8)
- Decompose conditional
- Consolidate conditional expression
- Consolidate duplicate conditional fragments
- Remove control flag
- Replace nested conditional with guard clauses
- Replace conditional with polymorphism
- Introduce null object
- Introduce assertion

Martin Fowler.
Refactoring: Improving the Design of Existing Code.
Addison-Wesley. New York. 2000.

Maintenance: Refactoring

5. Making method calls simpler (15)
- Rename method
- Add parameter
- 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

6. Dealing with generalization (12)
- Pull up field
- Pull up method
- Pull up constructor body
- Push down method
- Push down field
- Extract subclass
- Extract superclass
- Extract Interface
- Collapse hierarchy
- Form template method
- Replace inheritance with delegation
- Replace delegation with inheritance

Martin Fowler.
Refactoring: Improving the Design of Existing Code.
Addison-Wesley. New York. 2000.

Maintenance: Refactoring

7. Big refactorings (4)
Tease apart inheritance
Convert procedural design to objects
Separate domain from presentation
Extract hierarchy

Total: 72

Martin Fowler.
Refactoring: Improving the Design of Existing Code.
Addison-Wesley. New York. 2000.

Maintenance: Refactoring

- ***Replace Type Code with Subclasses***
  - You have an immutable type code that affects the behavior of a class
  - Replace the type code with subclasses

Martin Fowler.
Refactoring: Improving the Design of Existing Code.
Addison-Wesley. New York. 2000.

Maintenance: Refactoring

- ***Replace Type Code with Subclasses***

Before

```
public class Shape
{
   private static final int RECTANGLE = 0;
   private static final int SQUARE = 1;
   private int shapeType;
   …
   public void move()
   {
      switch (shapeType)
      {  case RECTANGLE:
            …
            break;
         case SQUARE:
            …
            break;
       }
   }
}
```

Martin Fowler.
Refactoring: Improving the Design of Existing Code.
Addison-Wesley. New York. 2000.

Maintenance: Refactoring

- ***Replace Type Code with Subclasses***

After

```
public abstract class Shape
{
   public abstract void move();
}
public class Rectangle extends Shape
{
   public void move { … }
}
public class Square extends Rectangle
{
   public void move { … }
}
```

Martin Fowler.
Refactoring: Improving the Design of Existing Code.
Addison-Wesley. New York. 2000.

Maintenance: Refactoring

Smell	Common Refactorings
Alternative classes with diff interfaces	Rename method, move method
Comments	Extract method, introduce assertion
Data class	Move method, encapsulate field, encapsulate collection
Data clumps	Extract class, introduce parameter object, preserve whole object
Divergent change	Extract class
Duplicated code	Extract method, extract class, pull-up method, form template method
Feature envy	Move method, move field, extract method
Inappropriate intimacy	Move method, move field, change bidirectional association to unidirectional, replace inheritance with delegation, hide delegate

Martin Fowler.
Refactoring: Improving the Design of Existing Code.
Addison-Wesley. New York. 2000.

Maintenance: Refactoring

Smell	Common Refactorings
Primitive obsession	Replace data value with object, extract class, introduce parameter object, replace array with object, replace type code with class, replace type code with subclasses, replace type code with state/strategy
Refused bequest	Replace inheritance with delegation
Shotgun surgery	Move method, move field, inline class
Speculative generality	Collapse hierarchy, inline class, remove parameter, rename method
Switch statements	Replace conditional with polymorphism, **replace type code with subclasses**, replace type code with state/strategy, replace parameter with explicit methods, introduce null object
Temporary field	Extract class, introduce null object

Martin Fowler.
Refactoring: Improving the Design of Existing Code.
Addison-Wesley. New York. 2000.

Maintenance: Refactoring

Smell	Common Refactorings
Incomplete library class	Introduce foreign method, introduce local extension
Large class	Extract class, extract subclass, extract interface, replace data value with object
Lazy class	Inline class, collapse hierarchy
Long method	Extract method, replace temp with query, replace method with method object, decompose conditional
Long parameter list	Replace parameter with method, introduce parameter object, preserve whole object
Message chains	Hide delegate
Middle man	Remove middle man, inline method, replace delegation with inheritance
Parallel inheritance hierarchies	Move method, move field

Martin Fowler.
Refactoring: Improving the Design of Existing Code.
Addison-Wesley. New York. 2000.

How should you order those stages?

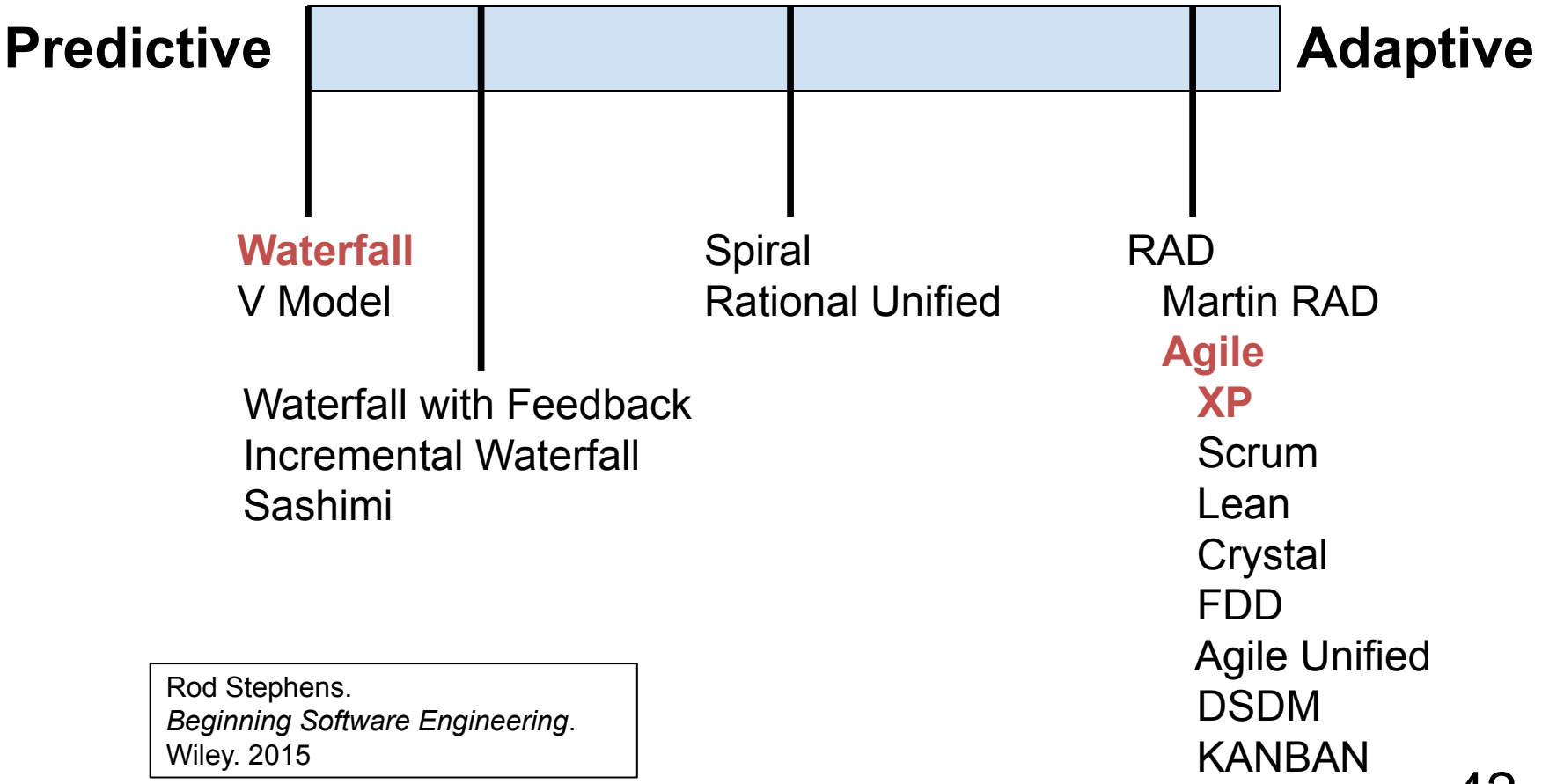
Agenda

- Requirements analysis
- Design
- Implementation
- Debugging
- Testing
- Evaluation
- Maintenance
- **Process models**

Process Models

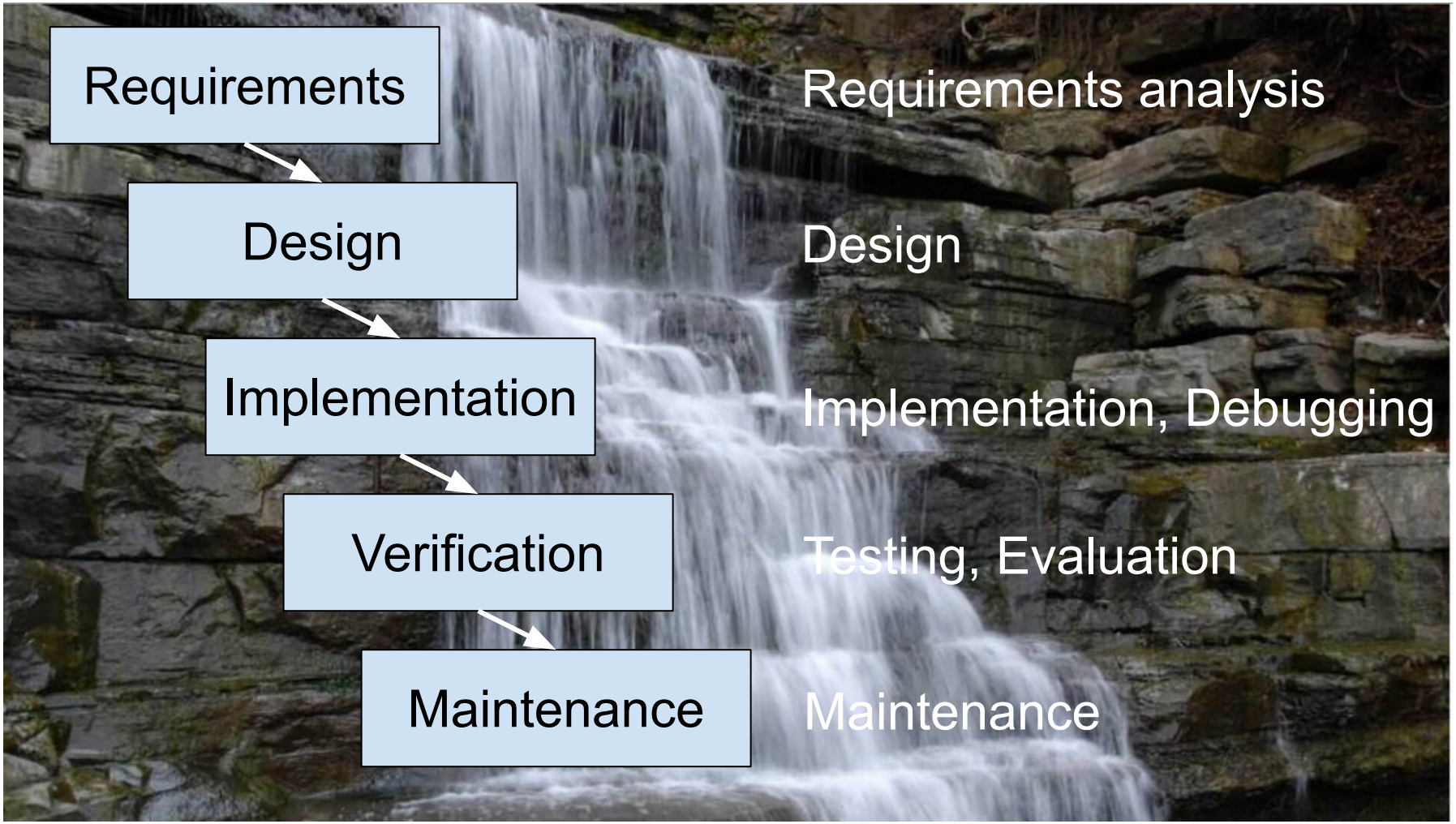
- ***Process models***
  - How should you order those stages?
  - (And more)

Process Models



Rod Stephens.
Beginning Software Engineering.
Wiley. 2015

Process Models: Waterfall



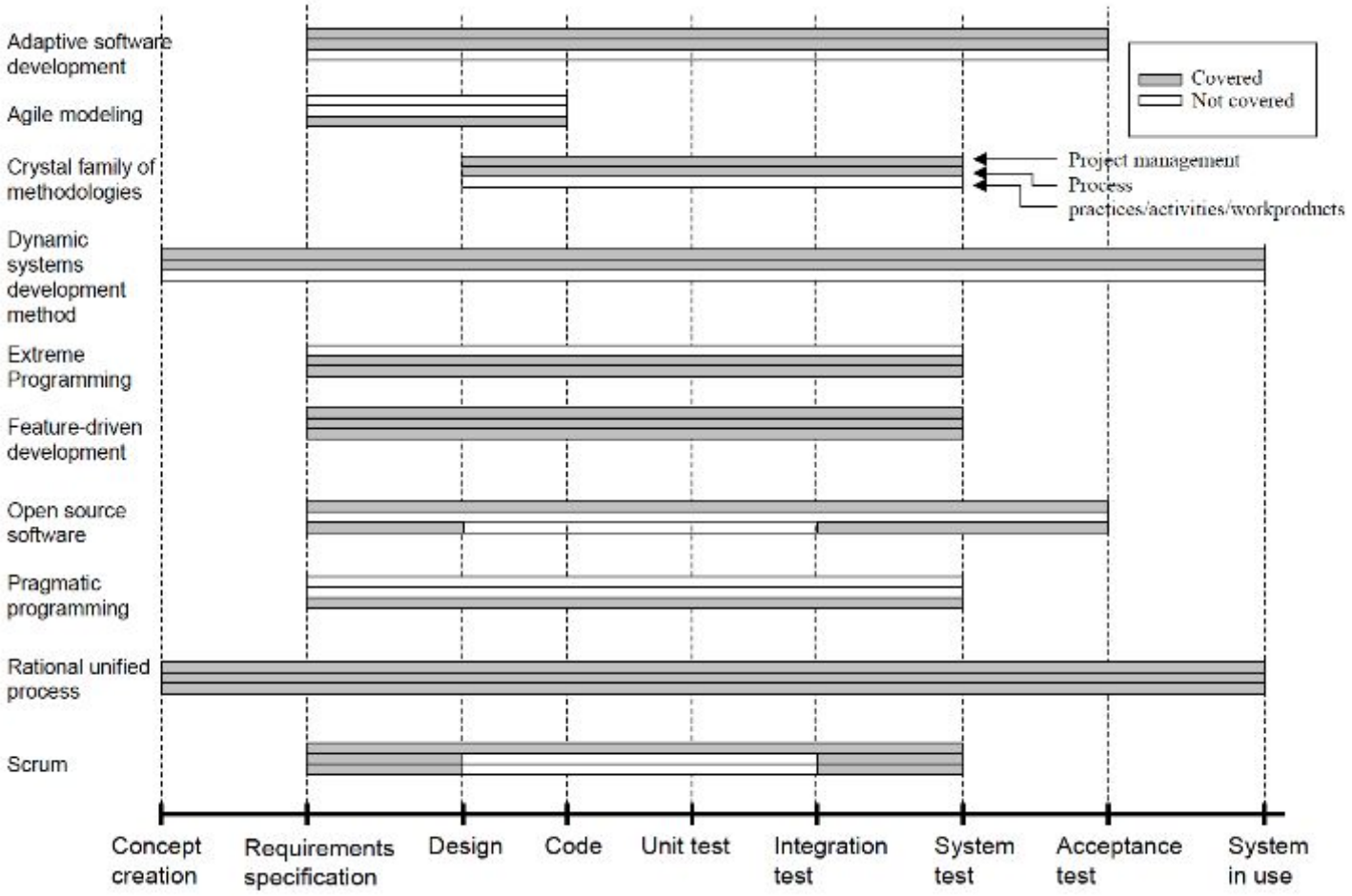
Process Models: Waterfall

- Completely predictive (non-adaptive)
  - From manufacturing industries
- Used by many early software dev projects
  - No other process models were known!
- Required by many funding agencies
  - Agency defines requirements
  - SW company does the rest, while agency monitors progress

Process Models: Waterfall

- Commentary
  - Perfect if all predictions are correct
  - It's **hardly ever** the case that all predictions are correct!

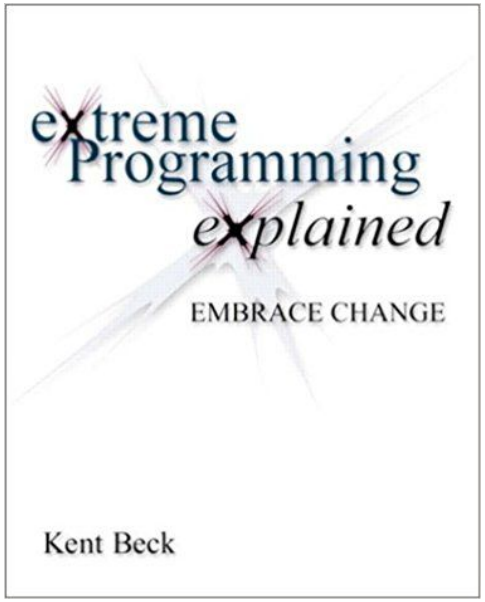
Process Models: Agile



Abrahamson P, Salo O, Ronkainen J, Warsta J (2002). *Agile Software Development Methods: Review and Analysis*. (Technical report). VTT. 478.

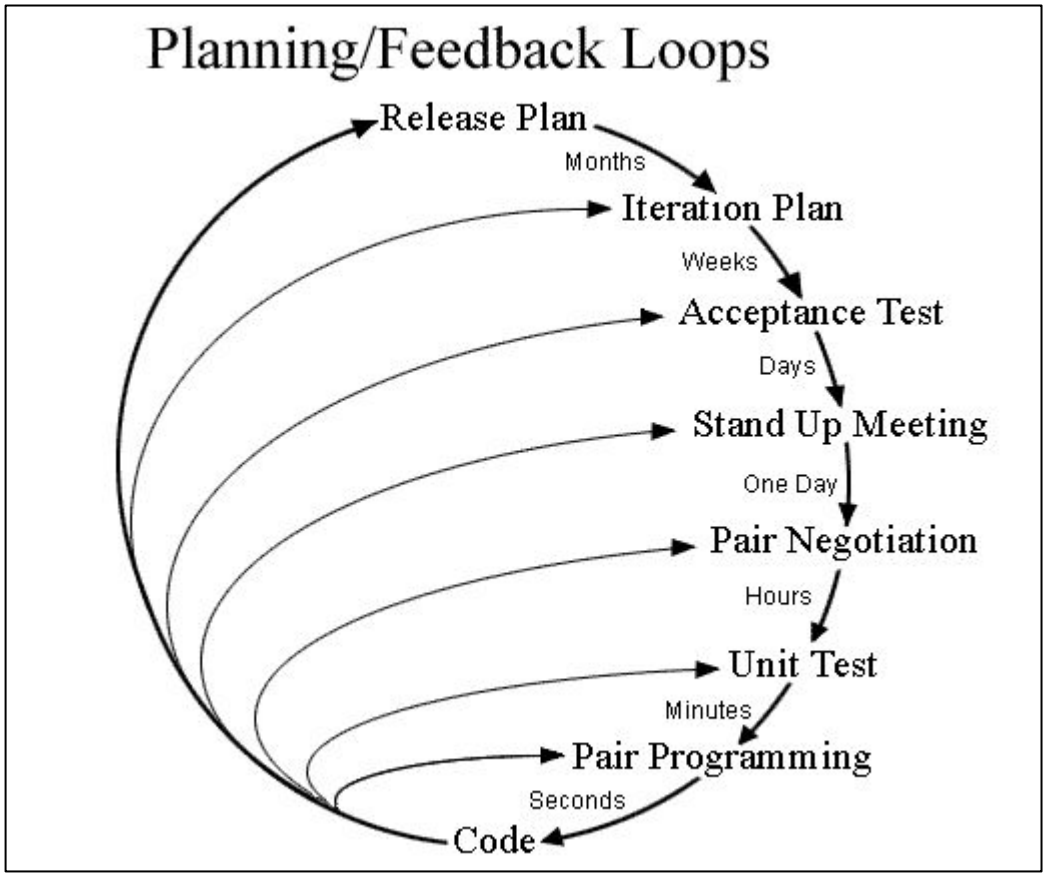
Process Models: Agile

Kent Beck



2000

Process Models: Agile



Requirements analysis

Evaluation

Design

Testing

Impl., Debugging

Maintenance

Diagram from Wikipedla *Extreme Programming* page

Process Models: Agile

- As adaptive (non-predictive) as possible
  - “Extremely” adaptive
  - “Embrace change”
- Essentially, code is the only artifact produced

Process Models: Agile

- The planning game
- **Small releases**
- Metaphor
- Simple design
- **Testing**
- **Refactoring**
- **Pair pgmming**
- Collective ownership
- Continuous integration
- 40-hour work week
- **On-site customer**
- Coding standards

Kent Beck.
Extreme Programming Explained: Embrace Change.
Addison-Wesley. New York. 2000.

Process Models: Agile

- Commentary
  - Appealing!
  - Too extreme?
    - An excuse for programmers to avoid some tasks that they find less fun?

Process Models

Predictive vs. Adaptive models:

Use Predictive When:	Use Adaptive When:
Developers are plan-oriented, adequately skilled, and have access to external knowledge	**Developers** are agile, highly skilled, collocated, and collaborative
Customers are not collocated	**Customers** are collocated
Requirements are knowable early and largely stable	**Requirements** are largely emergent and change rapidly
Team and **product** are large	**Team** and **product** are small
Primary objective is high assurance	**Primary objective** is rapid value

Boehm, B.
"Get Ready for the Agile Methods, With Care"
Computer 35 (1): 64-69.

Process Models: Commentary

- Every project is unique
  - Choose a process model that fits the project
  - Be willing to customize that process model

Process Models: Commentary

- Core points:
  - **Requirements**: First determine **who** the users are and **what** your system should do for them
    - **Involve the users!!!**
  - **Design**: Then determine **how** you want your system to work
  - **Implement**, **debug**, **test**: Then code, debug, and test your system
  - **Evaluate**: Then evaluate your system
    - **Involve the users!!!**
  - Iterate as often as you reasonably can

Summary

- We have covered these software engineering topics:
  - (1) Requirements analysis
  - (2) Design
  - (3) Implementation
  - (4) Debugging
  - (5) Testing
  - (6) Evaluation
  - (7) Maintenance
  - (8) Process models