Software Engineering I (02161)

Week 2

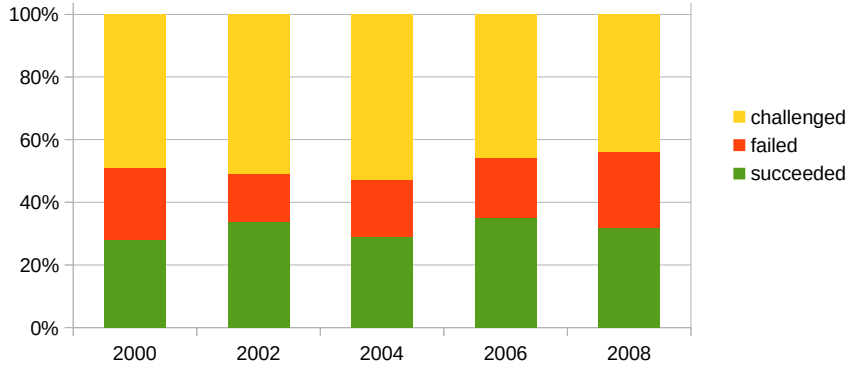
Assoc. Prof. Hubert Baumeister

DTU Compute
Technical University of Denmark

Spring 2018

Contents

Success rate for software projects 2000—2008



CHAOS Summary 2009 Report

- Succeeded: 32%
- Failed: 20%
- Challenged: 48% (over time, over budget, . . .)

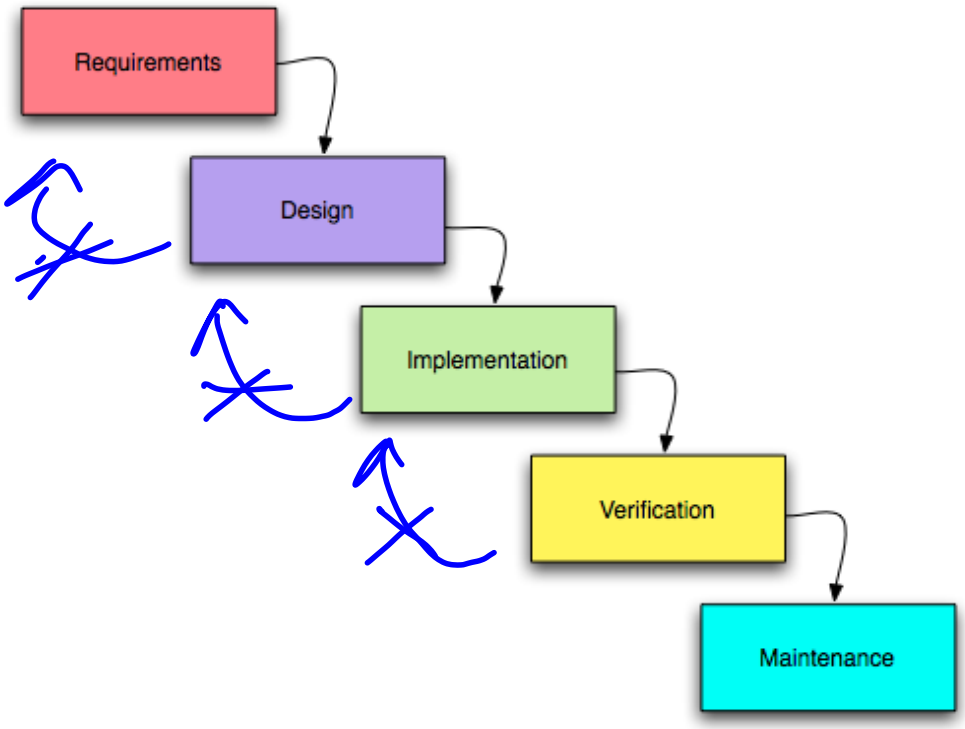
- Challenges of Software Development
  - On **time**
  - In **budget**
  - No **defects**
  - Customer **satisfaction**
- Type of projects
  - s-type, p-type, e-type

Activities in Software Development

- *Understand* and *document* what the customer wants: **Requirements Engineering**
- *How* to build the software: **Design**
- *Build* the software: **Implementation**
- *Validate*: **Testing**, **Verification**, **Evaluation**

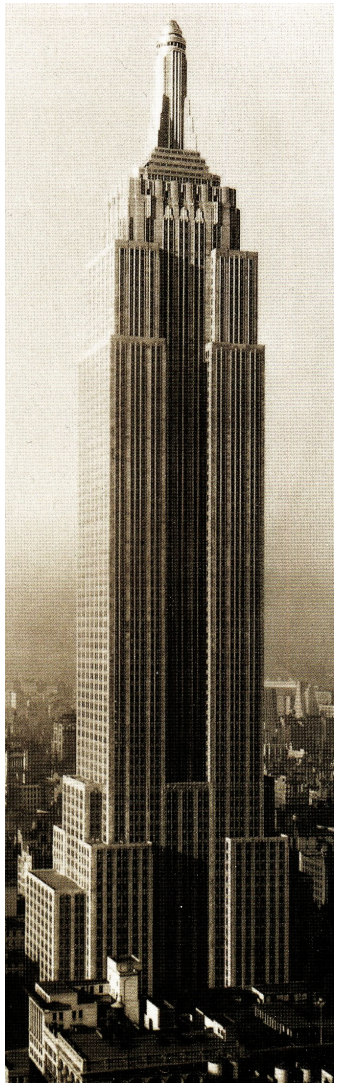
→ **Waterfall**

Waterfall process

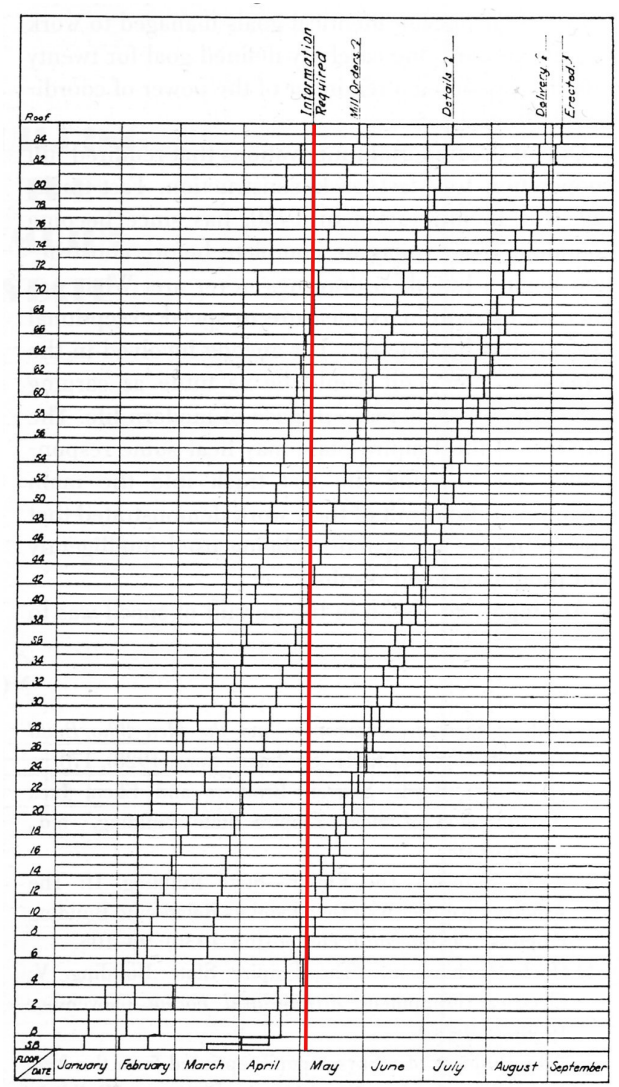


- 1970: Winston W. Royce how **not** to develop software
- 1985: Waterfall was required by the United States Department of Defence

Example: Empire State Steel Construction



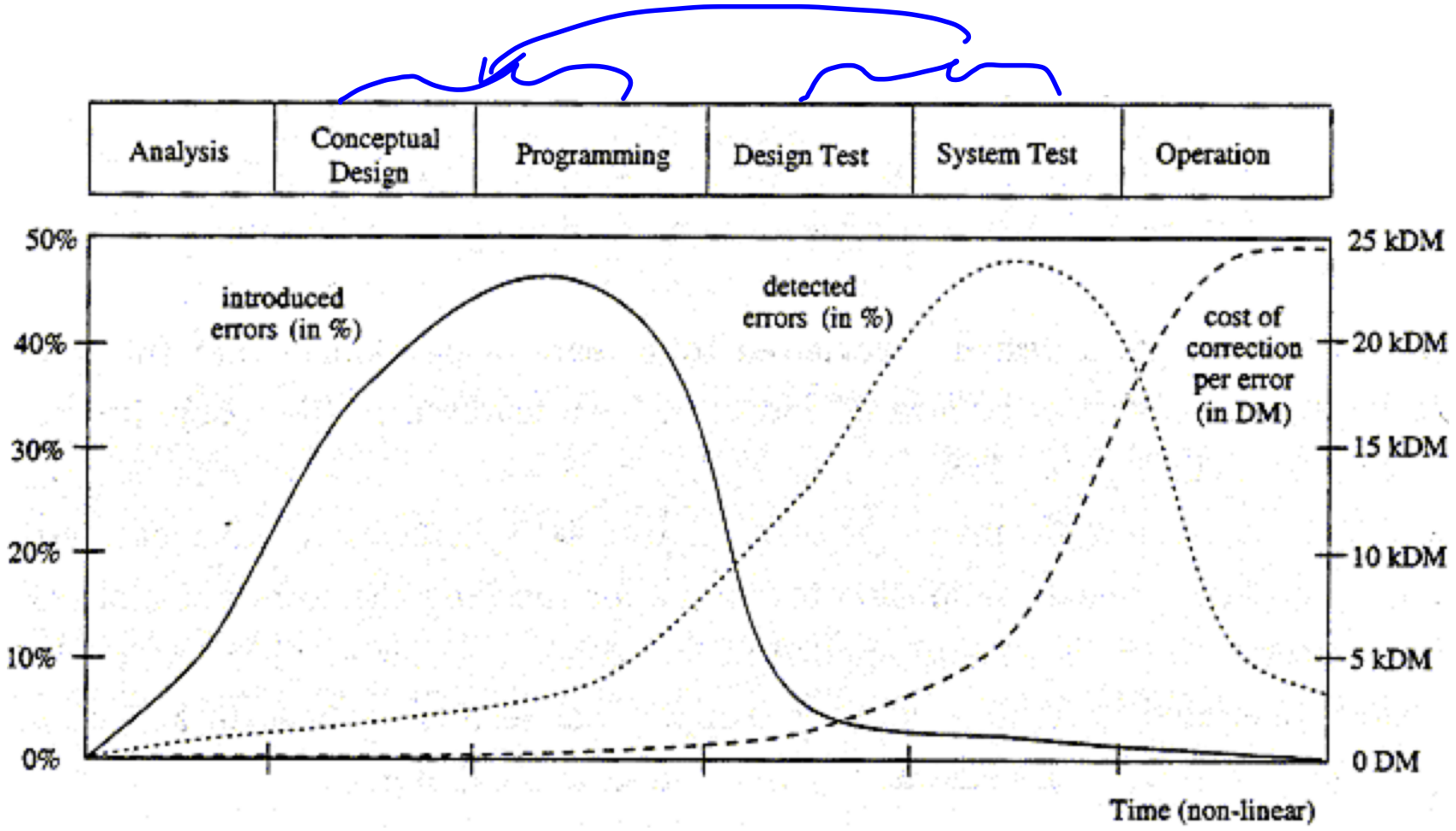
From The Empire State Building by John Tauranac



From Building the Empire State by Willis, 1998

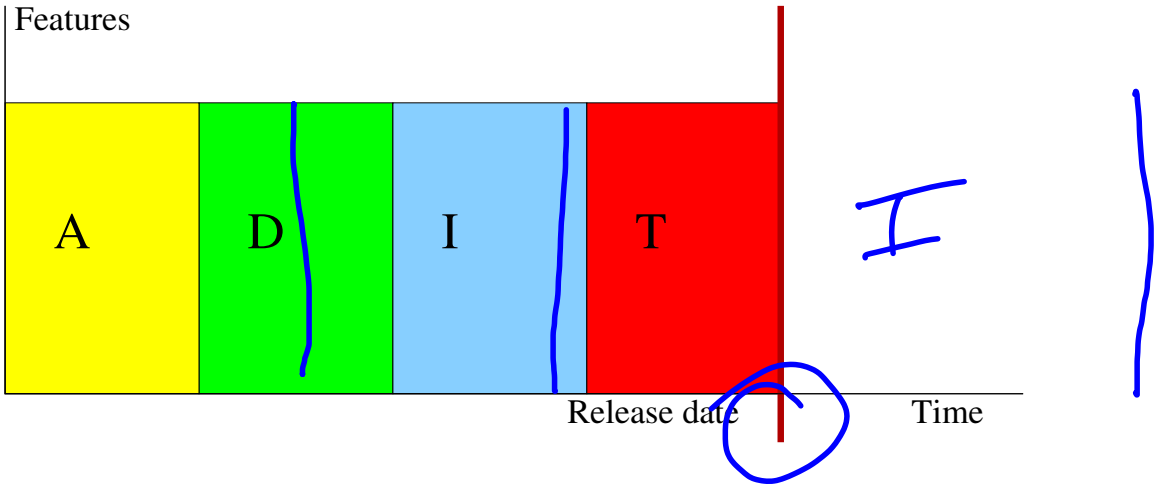
- Kept the budget
- Was finished **before** deadline
- Built in *21 month* (from conception to finished building) (1931)
  - → Basic design in 4 weeks
- *Fast-track* construction
  - → **Begin the construction before the design is complete**
  - → create a flow

Problem in Software Engineering

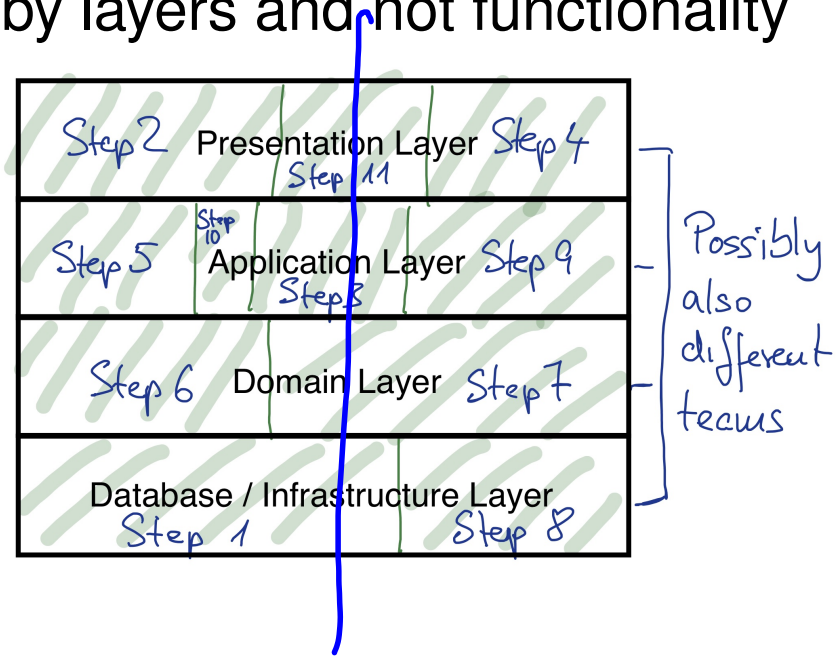


- Liggesmeyer 1998

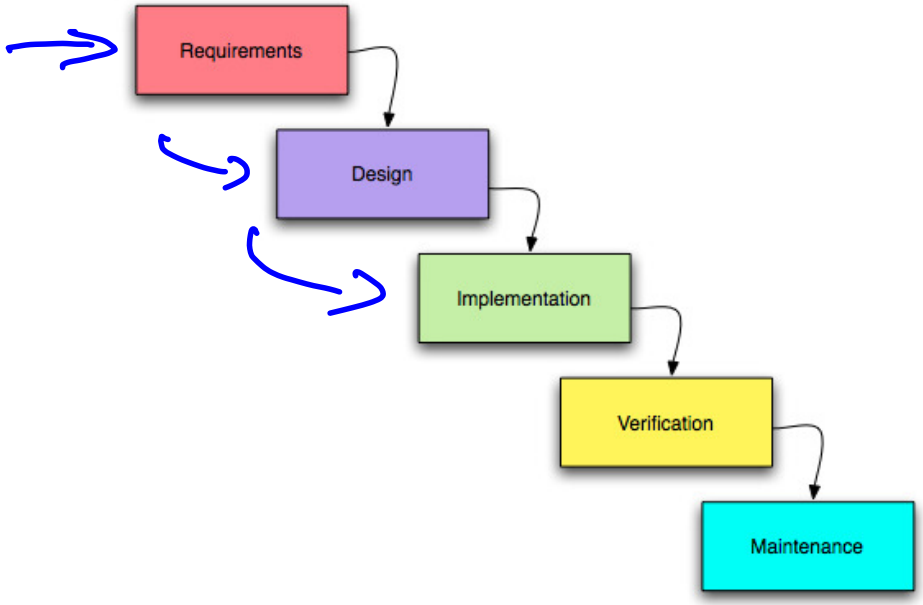
Delays in waterfall processes



Implementation by layers and not functionality



Costs of changing requirements: Waterfall

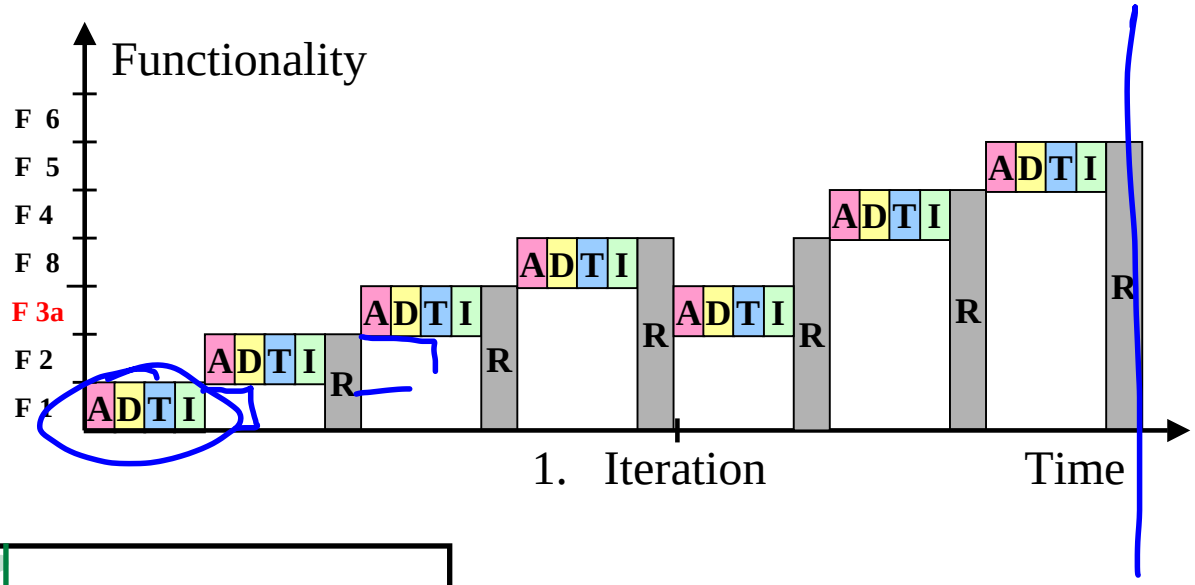


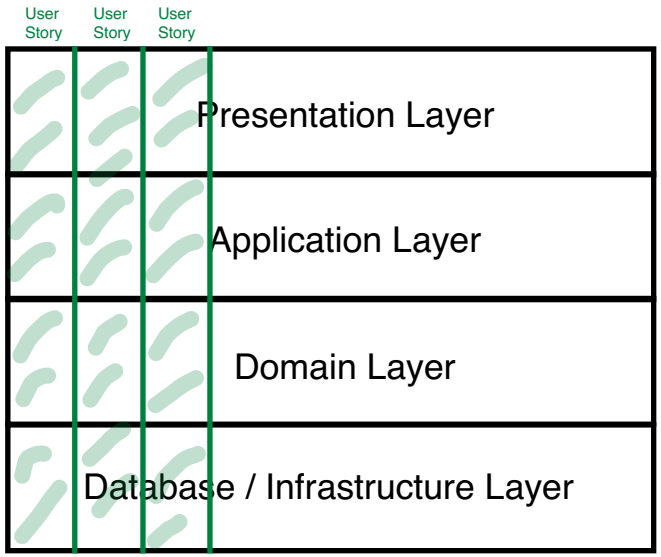
- Changed / new requirements change the design and implementation
  - Cost of change not predictable
  - → Avoid changing/new requirements if possible

→ Good for s-type projects, not applicable to p-type and e-type projects

Agile Software Development Methods (1999)

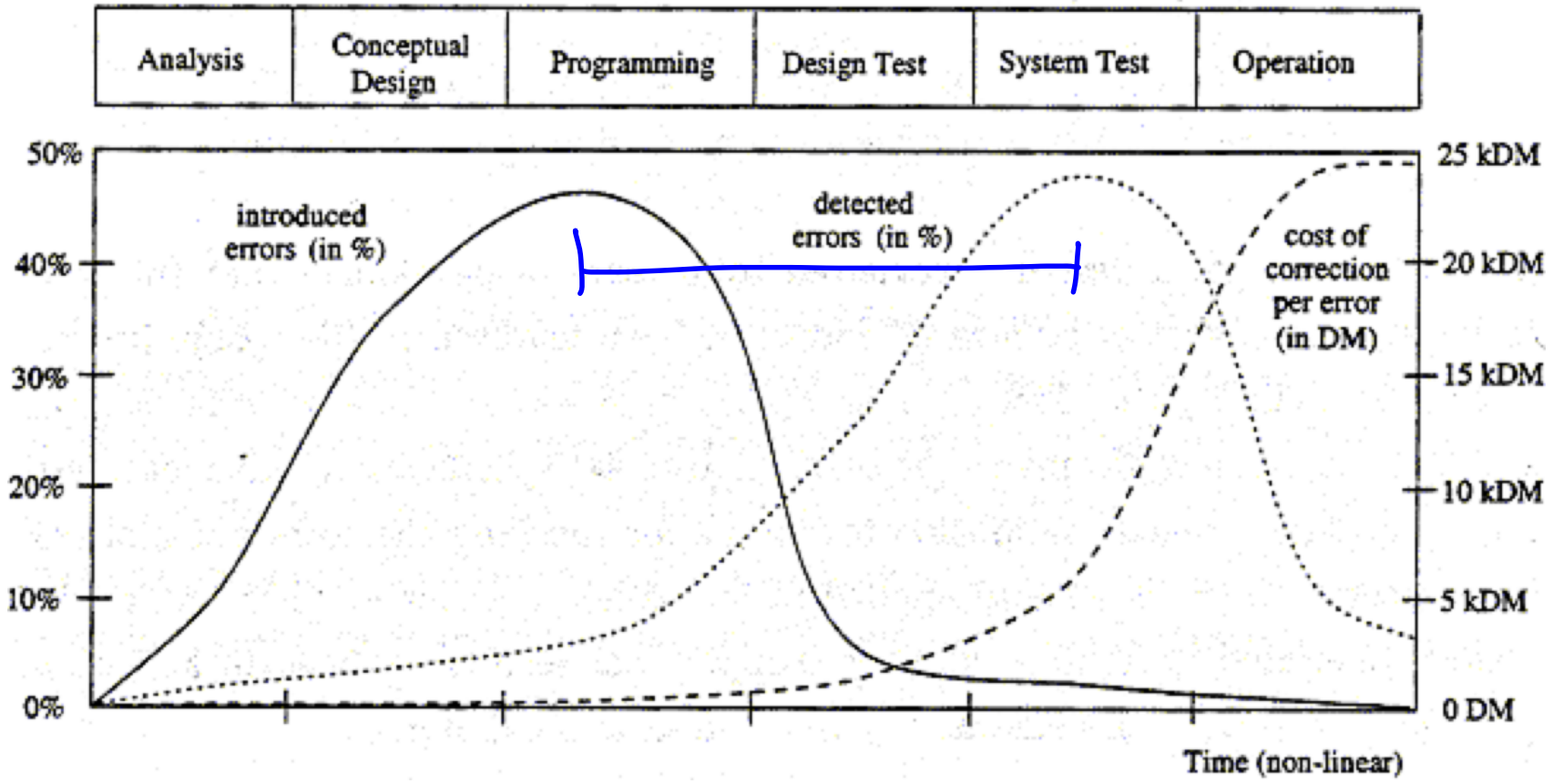
- Extreme Programming (XP) (1999), Scrum (1995–2001), Lean Software Development (2003), . . .
- Kanban (2010): not a method; *tool* to improve processes





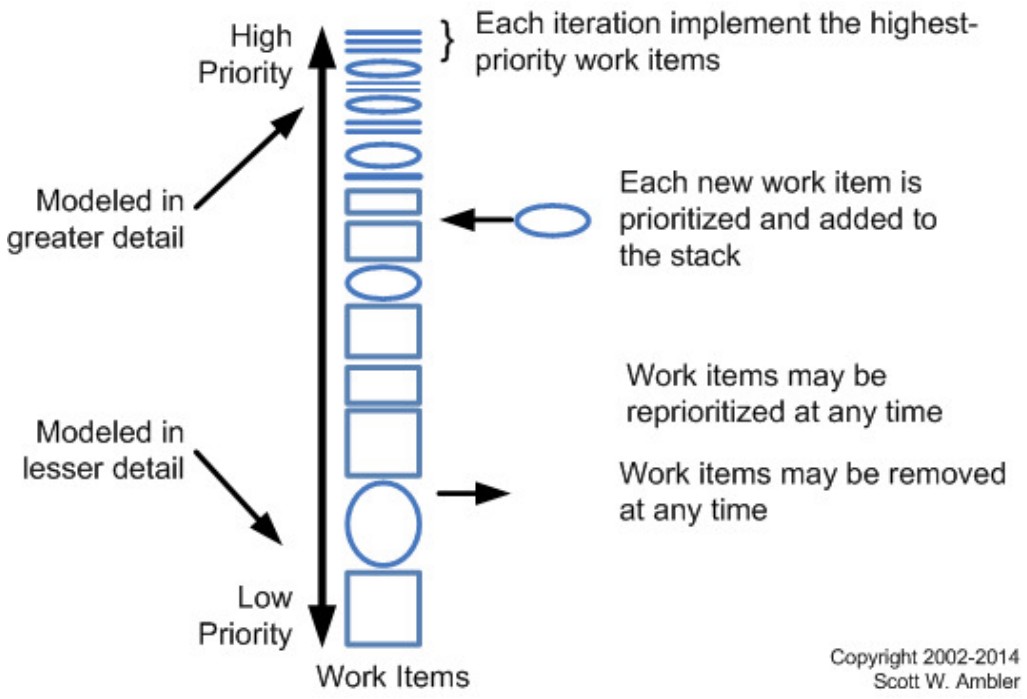
- Highest priority user story first
- If delayed: important features are implemented

Problem in Software Engineering (Recap)



- Liggesmeyer 1998

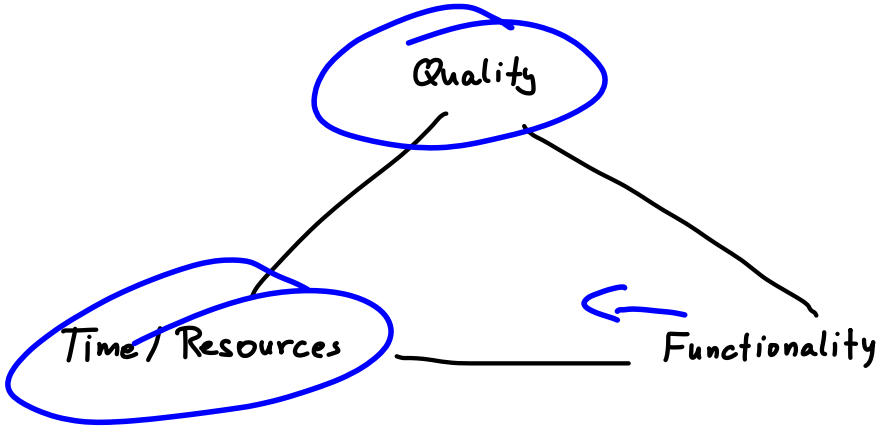
Changing Requirements: Agile Methods

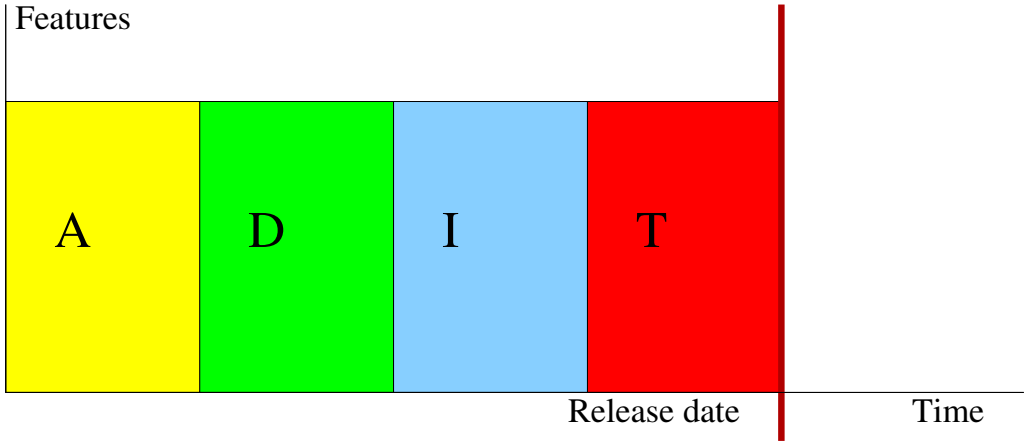


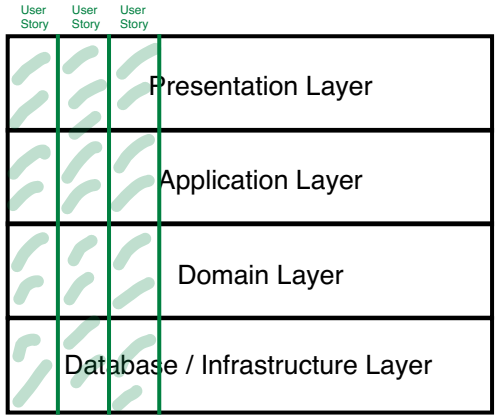
Scott Ambler 2003–2014 `http://www.agilemodeling.com/artifacts/userStory.htm`

- Cost of change
  - New / changed requirements not done yet: zero costs
  - Changed requirements already done: the cost of a requirement that can not be implemented

Resource Triangle

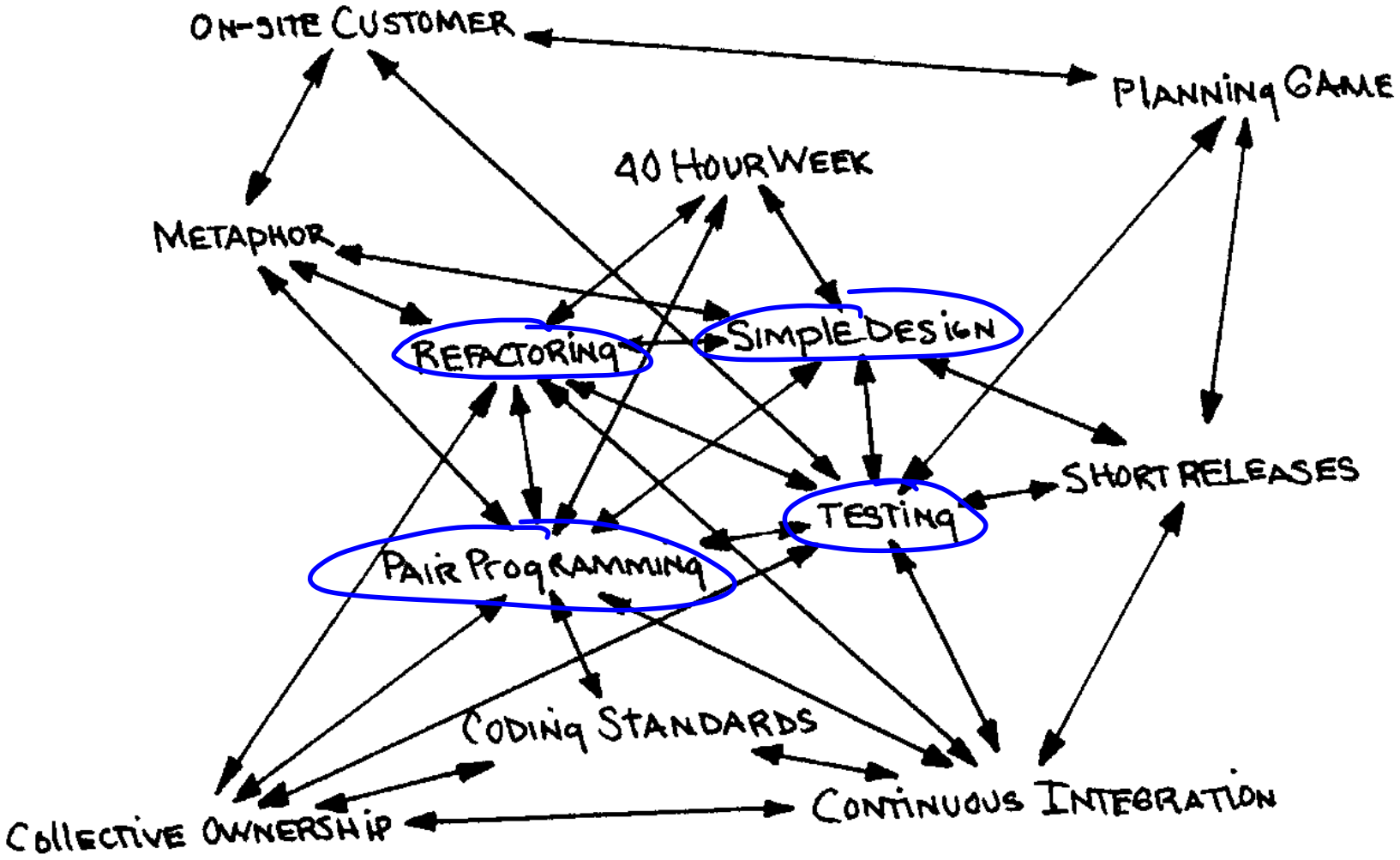






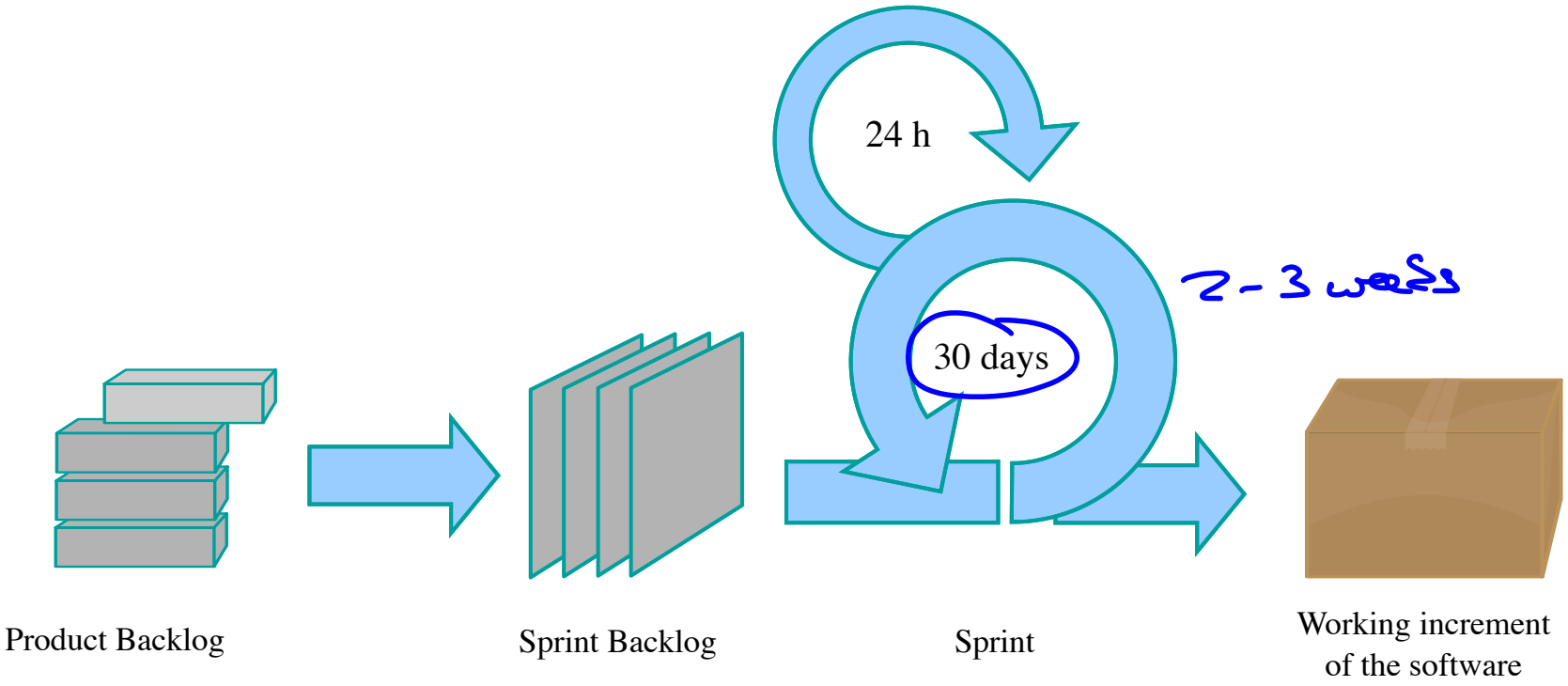
eXtreme Programming (XP)

- Kent Beck 1999
- 12 Practices



Kent Beck, Extreme Programming 1st ed.

Scrum



Wikipedia

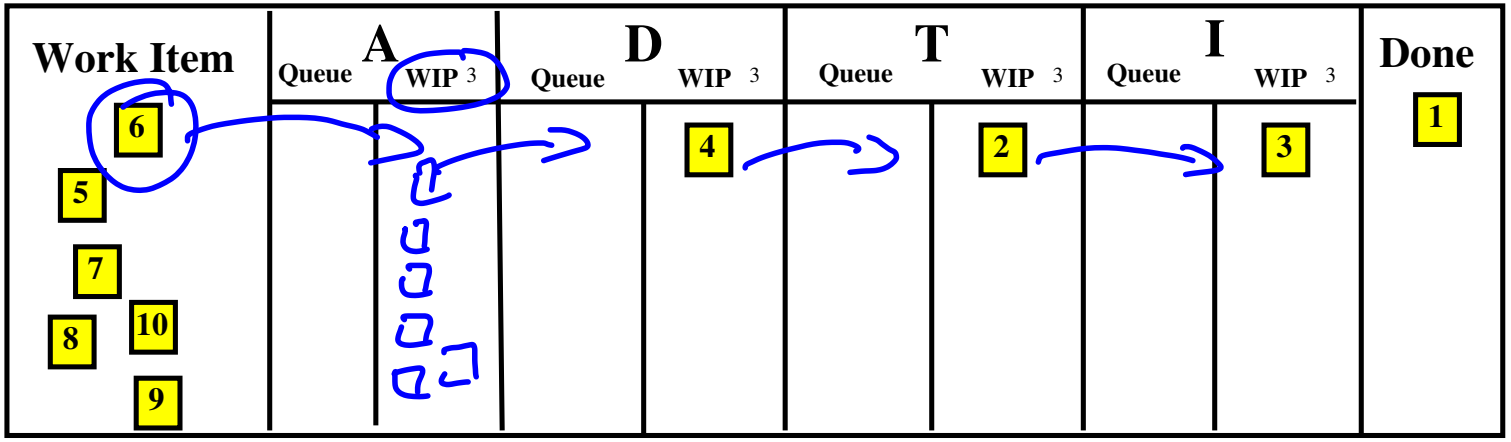
- Robert Martin (Uncle Bob) about "The Land that Scrum Forgot"
  `http://www.youtube.com/watch?v=hG4LH6P8Syk`
  → History about agile methods, the agile manifesto, and Scrum and its relationshop to XP

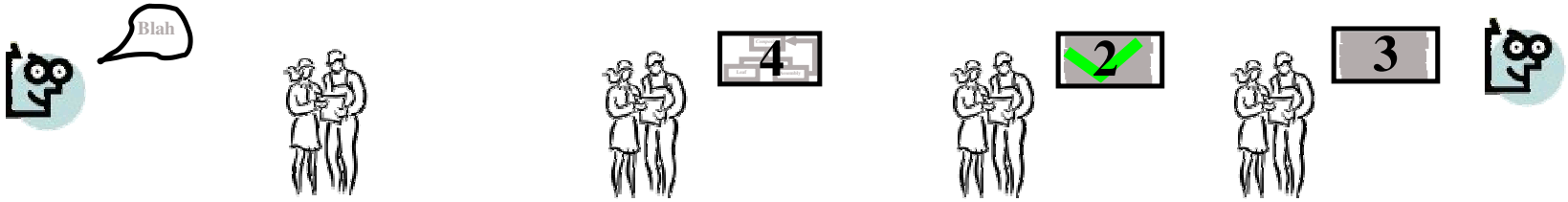
Lean Software Development

- Lean Production:
  - Value for the customer
  - Reduce the amount of *waste* in the production process
  - Generate *flow*
- Waste: resources used which do not produce value for the customer
  - time needed to fix bugs
  - time to change the system because it does not fit the customers requirements
  - time waiting for approval
  - ...

Generating flow using Pull and Kanban

WIP = Work in Progress Limit





Flow through Pull with Kanban

- Process controlling: local rules
- Load balancing: Kanban cards and *Work in Progress (WIP) limits*
- Integration in other processes

Figure from David Anderson `www.agilemanagement.net`

Online Kanban Tool: Trello

- `www.trello.com`: Electronic Kanban board useful for your project
- Kanban board for the exercise `https://trello.com/b/w3Dal5rF`
- Another `https://trello.com/b/4wddd1zf/kanban-workflow`

Contents

Purpose of tests

Goal: finding bugs

Edsger Dijkstra

”Tests can show the presence of bugs, but not their absence.”

$\rightarrow$ proof of program correctness

Types of tests

1. Developer tests
   a) Unit tests (single classes and methods)
   b) Component tests (single components = cooperating classes)
   c) System tests / Integration tests (cooperating components)
2. Release tests
   a) Scenario based testing
   b) Performance testing
3. User tests
   a) Acceptance tests

Acceptance Tests

- Tests defined by / with the help of the user
  - based on the requirements
- Traditionally
  - manual tests
  - *after* the software is delivered
- Agile software development
  - automatic tests: JUnit, Cucumber, . . .
  - created *before* the user story / use case scenario is implemented
  - developed with the customer

Example of acceptance tests

- Use case

  name: Login Admin
  actor: Admin
  precondition: Admin is not logged in
  main scenario

  1. Admin enters password
  2. System responds true

  alternative scenarios:

  1a. Admin enters wrong password
  1b. The system reports that the password is wrong and the use case starts from the beginning

  postcondition: Admin is logged in

Manual tests

Successful login

Prerequisit: the password for the administrator is “adminadmin”

Input	Step	Expected Output	Fail	OK
	Startup system	“0) Exit” “1) Login as administrator”		✓
“1”	Enter choice	“password”	✓	✓
“adminadmin”	Enter string	“logged in”		

Failed login

Prerequisit: the password for the administrator is “adminadmin”

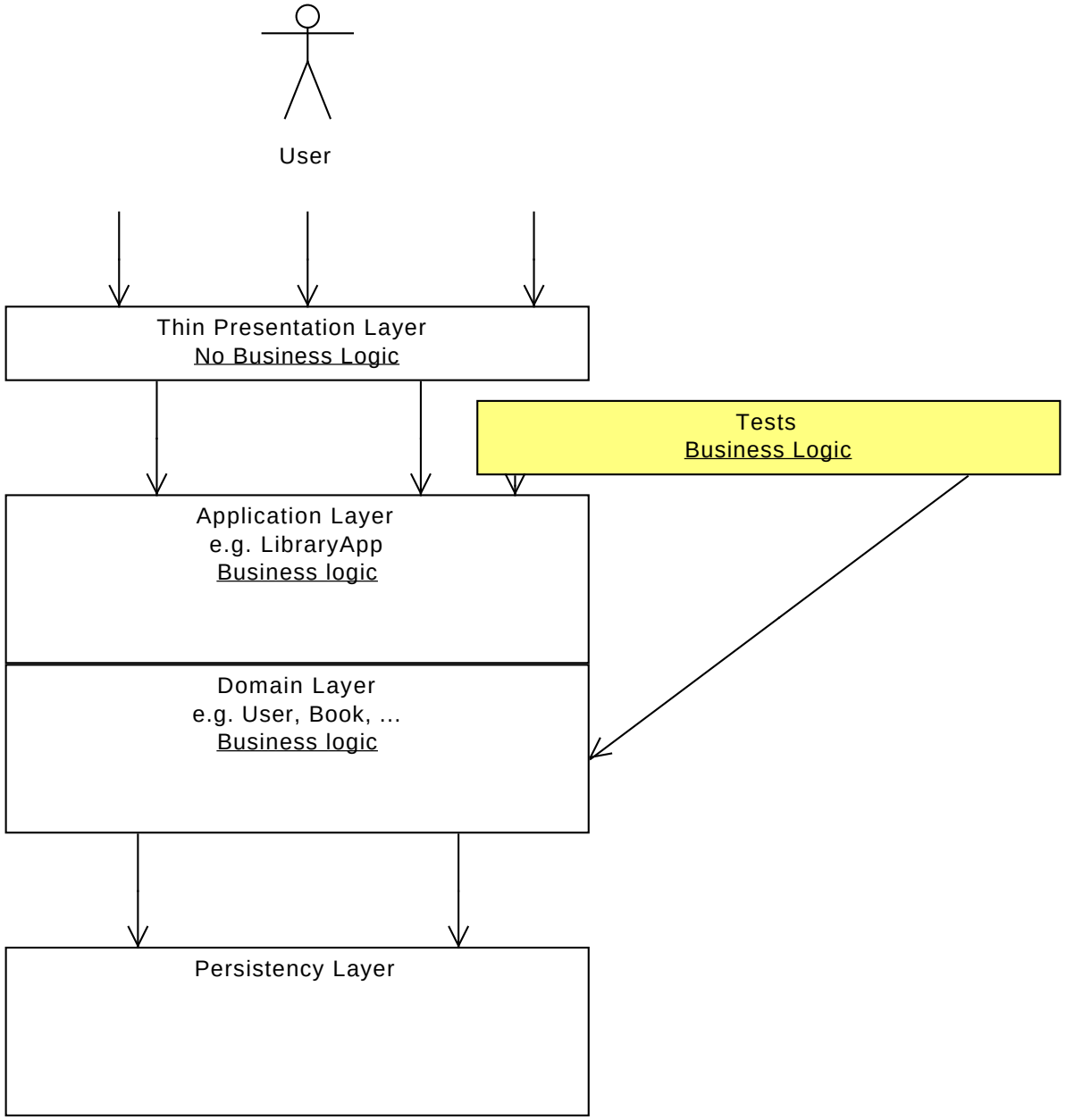
Input	Step	Expected Output	Fail	OK
	Startup system	“0) Exit” “1) Login as administrator”		✓
“1”	Enter choice	“password”		✓
“admin”	Enter string	“Password incorrect” “0) Exit” “1) Login as administrator”		✓

Manual vs. automated tests

- Manual tests should be avoided
  - Are expensive; can't be run often
- Automated tests
  - Are cheap; can be run often
- Robert Martin (Uncle Bob) in `http://www.youtube.com/watch?v=hG4LH6P8Syk`
  - manual tests are immoral from 36:35
  - how to test applications having a UI from 40:00
- How to do UI tests?

  → Solution: Test under the UI

Test under the UI



Automatic acceptance test using JUnit

Successful login

```
@Test
public void testLoginAdmin() {
   LibraryApp libApp = new LibraryApp();

   assertFalse(libApp.adminLoggedIn());

   boolean login = libApp.adminLogin("adminadmin");

   assertTrue(login);
   assertTrue(libApp.adminLoggedIn());
}
```

Failed login

```
@Test
public void testWrongPassword() {
   LibraryApp libApp = new LibraryApp();

   assertFalse(libApp.adminLoggedIn());

   boolean login = libApp.adminLogin("admin");

   assertFalse(login);
   assertFalse(libApp.adminLoggedIn());
}
```

Acceptance test as a Cucumber Feature

```
Feature: Admin login
   Description: The administrator logs into the library system
   Actor: Administrator

Scenario: Administrator can login
   Given that the administrator is not logged in
   And the password is "adminadmin"
   Then the administrator login succeeds
   And the adminstrator is logged in

Scenario: Administrator has the wrong password
   Given that the administrator is not logged in
   And the password is "wrong password"
   Then the administrator login fails
   And the administrator is not logged in
```

Step definitions (excerpt)

Reg Exp.

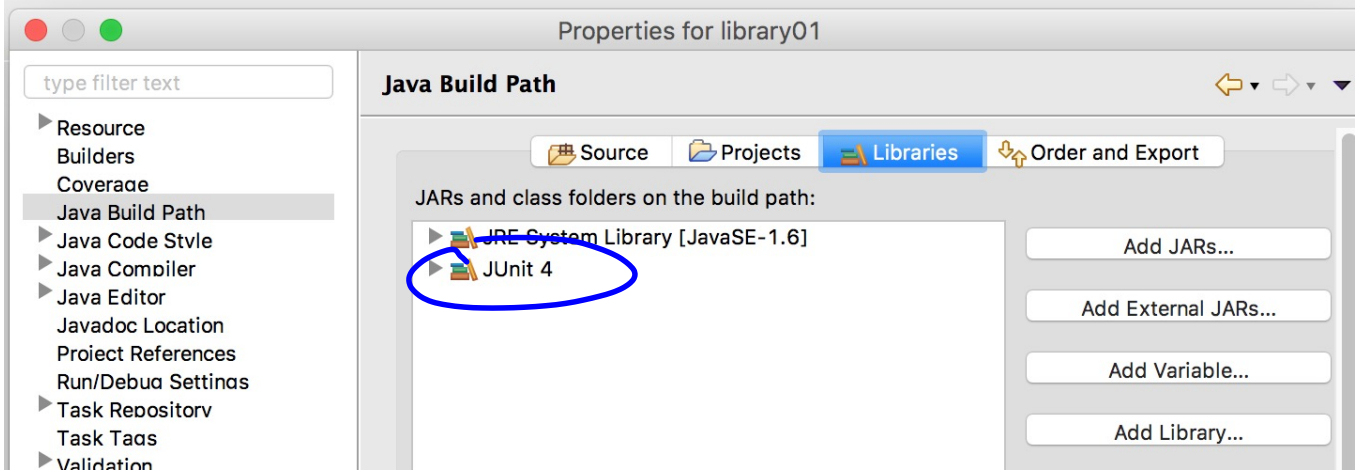
```
   @Given("^the password is \"([^\"]*)\"$")
   public void thePasswordIs(String password) throws Exception {
      this.password = password;
   }

   @Then("^the administrator login succeeds$")
   public void theAdministratorLoginSucceeds() throws Exception {
      assertTrue(libraryApp.adminLogin(password));
   }
```

JUnit

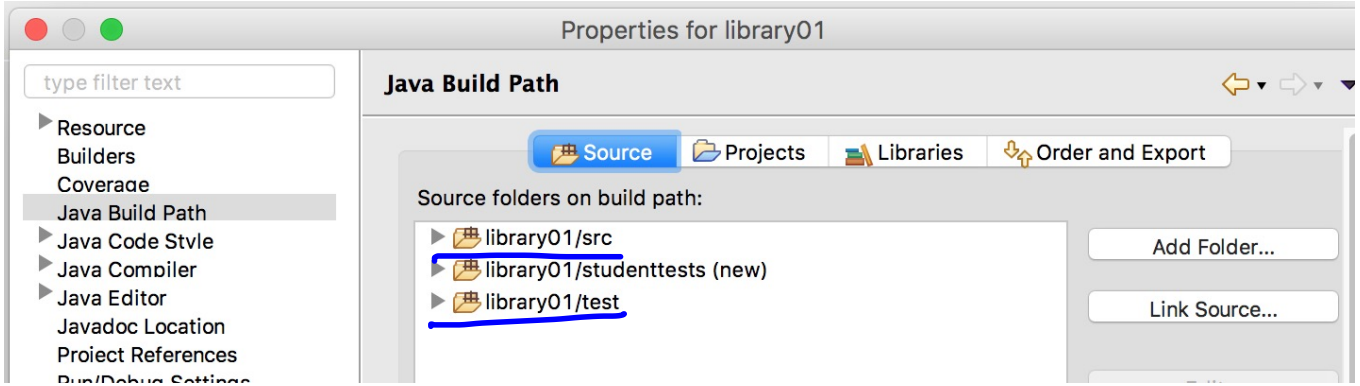
- Framework for automated tests in Java
- Kent Beck (Patterns, XP) and Erich Gamma (Design Patterns, Eclipse IDE)
- Unit-, component-, and *acceptance* tests
- `http://www.junit.org`
- *x*Unit

JUnit and Eclipse

- JUnit 4.x libraries



- New source directory for tests



JUnit *4.x* structure

```
import org.junit.Test;
import static org.junit.Assert.*;

public class UseCaseName {
   @Test
   public void scenarioName1() {..}
   @Test
   public void scenarioName2() throws Exception {..}
   ...
}
```

- *Independent* tests
- No try-catch blocks (exception: checking for exceptions)

JUnit *4.x* structure (Before and After)

```
...
public class UseCaseName {
   @After
   public void tearDown() {...}
   @Before
   public void setUp() {...}
   @Test
   public void scenario1() {..}
   @Test
   public void scenario2() {..}
   ...
}
```

JUnit assertions (also used with Cucumber)

General assertions

```
import static org.junit.Assert.*;

assertTrue(bexp)
assertTrue(msg,bexp)
```

Specialised assertions for readability

```
import static org.junit.Assert.assertThat;
import static org.hamcrest.CoreMatchers.*;

1. assertFalse(bexp) / assertThat(bexp,is(false))

2. fail()

3. assertEquals(exp,act) / assertThat(exp,is(equal(act)))

4. assertNull(obj) / assertThat(obj,is(nullValue()))

5. assertNotNull(obj) / assertThat(obj,is(not(nullValue())))
...
```

matcher

act

exp

Cucumber

- Behaviour-Driven Development: User Stories

```
Feature: Name of the feature
   Description ...

Scenario: Name
   Description ...
   Given an initial state
   And ...
   When an action happens
   And ...
   Then an assertion is true
   And ...
```

- Originally Ruby
- Gherkin: for scenarios
- Programming language (Java): Glue code
- More information: *The Cucumber for Java Book* available online through DTU library

Example: Add book

```
Feature: Add book
Description: A book is added to the library
Actors: Administrator

Scenario: Add a book successfully
  Given that the administrator is logged in
  And I have a book with title "Extreme Programming",
                 author "Kent Beck",
                 and signature "Beck99"
  When I add the book
  Then the book is added to the library
```

Example: Add book Step definitions

```
@Given("^that the administrator is logged in$")
public void thatTheAdministratorIsLoggedIn() throws Exception {
   assertTrue(libraryApp.adminLogin("adminadmin"));
}

@Given("^I have a book with title \"([^\"]*)\", author \"([^\"]*)\",
                  and signature \"([^\"]*)\"$")
public void iHaveABookWithTitleAuthorAndSignature(String title,
         String author, String signature) throws Exception {
   book = new Book(title,author,signature);
}

@When("^I add the book$")
public void iAddTheBook() throws Exception {
   try {
      libraryApp.addBook(book);
   } catch (OperationNotAllowedException e) {
      errorMessage = e.getMessage();
   }
}

@Then("^the book is added to the library$")
public void theBookWithTitleAuthorAndSignatureIsAddedToTheLibrary(
                  String title, String author, String signature)
throws Exception {
   assertTrue(libraryApp.getBooks().contains(book));
}
```

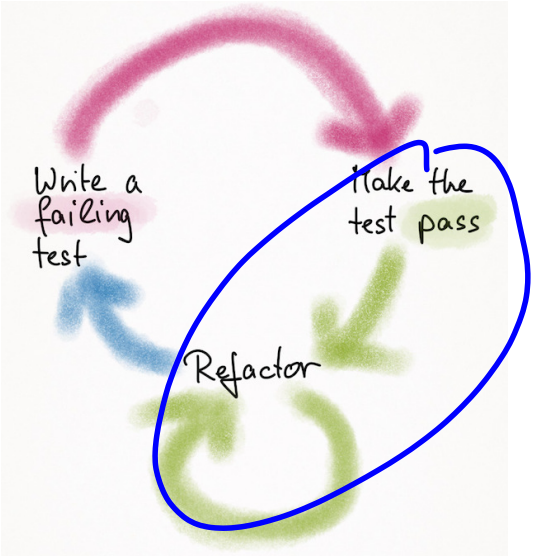
Contents

Test-Driven Development

- Test *before* the implementation
  - → API design
  - → Testable software
- Tests = expectations on software
- All kind of tests: unit-, component-, system tests

TDD cycle

- Repeat for functionality, bug, . . .



- Until: no more ideas for tests
- Important:
  - One failing test only
  - Simplicity: Only write that code that make the test pass, even if trivial
    - → add failing tests to force more code

TDD/BDD example: Borrow Book

Use Case = Cucumber Feature

```
Feature: Borrow Book
 Description: The user borrows a book
 Actors: User

Scenario: User borrows book
 Given a book in the library
 And a user is registered with the library
 When the user borrows the book
 Then the book is borrowed by the user

Scenario: User borrows book but has already more than 10 books
 Given the user has borrowed 10 books
 And a user is registered with the library
 And a book is in the library
 When the user borrows the book
 Then the book is not borrowed by the user
 And the user gets the error message "Can’t borrow more than 10 books"
```

Create the step definitions for the first scenario

```
@Given("^a book is in the library$")
public void aBookWithSignatureIsInTheLibrary() throws Exception {
    throw new PendingException();
}

@Given("^a user is registered with the library$")
public void aUserIsRegisteredWithTheLibrary() throws Exception {
    throw new PendingException();
}

@When("^the user borrows the book$")
public void theUserBorrowsTheBook() throws Exception {
    throw new PendingException();
}

@Then("^the book is borrowed by the user$")
public void theBookIsBorrwedByTheUser() throws Exception {
    throw new PendingException();
}
```

Implement the test logic

```
@Given("^a book is in the library$")
public void aBookWithSignatureIsInTheLibrary(String signature) throws
    book = new Book("Extreme Programming", "Kent Beck", "Beck99");
    libraryApp.adminLogin("adminadmin");
    libraryApp.addBook(book);
    libraryApp.adminLogout();
}

@Given("^a user is registered with the library$")
public void aUserIsRegisteredWithTheLibrary() throws Exception {
    user = helper.getUser();
    libraryApp.adminLogin("adminadmin");
    libraryApp.registerUser(user);
    libraryApp.adminLogout();
}

@When("^the user borrows the book$")
public void theUserBorrowsTheBook() throws Exception {
   helper.getUser().borrowBook(book);
}

@Then("^the book is borrowed by the user$")
public void theBookIsBorrwedByTheUser() throws Exception {
   assertThat(helper.getUser().getBorrowedBooks(),hasItem(book));
}
```

Implement the production code

```
public void borrowBook(Book book) {
   borrowedBooks.add(book);
}
```

Implement (create) a second scenario:

```
Feature: Borrow Book
 Description: The user borrows a book
 Actors: User

Scenario: User borrows book
 Given a book is in the library
 And a user is registered with the library
 When the user borrows the book
 Then the book is borrowed by the user

Scenario: User borrows book but has already more than 10 books
 Given the user has borrowed 10 books
 And a user is registered with the library
 And a book is in the library
 When the user borrows the book
 Then the book is not borrowed by the user
 And the user gets the error message "Can’t borrow more than 10 books"
```

Implement the missing steps

```
@Given("^the user has borrowed (\\d+) books$")
public void theUserHasBorrowedBooks(int arg1) throws Exception {
  List<Book> exampleBooks = getExampleBooks(10);
  addBooksToLibrary(exampleBooks);
  for (Book b : exampleBooks) {
   helper.getUser().borrowBook(b);
  }
}

@Then("^the book is not borrowed by the user$")
public void theBookIsNotBorrowedByTheUser() throws Exception {
  assertThat(helper.getUser().getBorrowedBooks(),not(hasItem(book)));
}

@Then("^the user gets the error message \"([^\"]*)\"$")
public void theUserGetsTheErrorMessage(String errorMessage)

  assertEquals(errorMessage, this.errorMessage.getErrorMessage());
}

@When("^the user borrows the book$")
public void theUserBorrowsTheBook() throws Exception {
  try {
   helper.getUser().borrowBook(book);
  } catch (TooManyBooksException e ) {
   errorMessage.setErrorMessage(e.getMessage());
  }
}
```

Implementation of the alternative scenario

```
public void borrowBook(Book book) throws TooManyBooksException
 if (borrowedBooks.size() >= 10) {
  throw new TooManyBooksException();
 }
 borrowedBooks.add(book);
}
```

Implement missing logic

- Add more scenarios
- Add JUnit tests

Another example with JUnit

- Creating a program to generate the n-th Fibonacci number

→ Codemanship's Test-driven Development in Java by Jason Gorman

  `http://youtu.be/nt2KKUSSJsY`

- Note: The video uses JUnitMax to run JUnit tests automatically whenever the test files change, which, it seems, is not available anymore.
- A tool with similar functionality but free is Infinitest (`https://infinitest.github.io`)

Exercise

- Trello Board: `https://trello.com/b/w3Dal5rF`