Software Engineering I (02161)

Week 3

Assoc. Prof. Hubert Baumeister

DTU Compute
Technical University of Denmark

Spring 2017

Contents

Booleans


```
if ("adminadmin".equals(password)) {
   adminLoggedIn = true;
} else {
   adminLoggedIn = false;
}
```

Booleans

```
if ("adminadmin".equals(password)) {
    adminLoggedIn = true;
} else {
    adminLoggedIn = false;
}
```

Don't use conditionals to set a boolean variable

- Better

```
adminLoggedIn = "adminadmin".equals(password);
```

Booleans

```
if (!adminLoggedIn == false) {
   throw new OperationNotAllowedException();
} else {
   if ( adminLoggedIn == true ) books.add(book);
}
```

Booleans

```
if ( adminLoggedIn == false) {
   throw new OperationNotAllowedException();
} else {
   if ( adminLoggedIn == true ) books.add(book);
}
```

Use boolean variables directly; don't compare boolean variables with true or false

- Better

```
if ( !adminLoggedIn ) {
   throw new OperationNotAllowedException();
} else {
   books.add(book);
}
```

or

```
if ( !adminLoggedIn ) {
   throw new OperationNotAllowedException();
}
books.add(book);
```

Use constants instead of literals

```
public boolean login(String password) {
   adminLoggedIn = "adminadmin".equals(password);
   ...
}
```

String

DRY

Use constants instead of literals

```
public boolean login(String password) {
   adminLoggedIn = "adminadmin".equals(password);
   ...
}

static final String ADMIN_PASSWORD = "adminadmin";
...
public boolean login(String password) {
   adminLoggedIn = ADMIN_PASSWORD.equals(password);
   ...
}
```

- Put the constant in the class where it belongs conceptually
- Gives the constant a meaning: ADMIN_PASSWORD vs ”adminadmin”, MAX_NUMBER_OF_LOANED_BOOKS vs 5
- Don’t repeat yourself (DRY): avoids several occurences of the same constant, e.g. 5
- Naming convention: All uppercase with underscore separating words (inherited from C)

Delegate Responsibility

- Original

```
public List<Book> search(String string) {
    List<Book> booksFound = new ArrayList<Book>();
    for (Book book : books) {
        if (book.getSignature().contains(string) ||
            book.getTitle().contains(string) ||
            book.getAuthor().contains(string)) {
            booksFound.add(book);
        }
    }
    return booksFound;
}
```

→ contains(Book book, String string)

create a method in class LibraryApp

better: create contains(String string) in class Book.

Delegate Responsibility

- LibraryApp delegates *contains* functionality to class book

```
public List<Book> search(String string) {
   List<Book> booksFound = new ArrayList<Book>();
   for (Book book : books) {
      if (book.contains(string)) {
         booksFound.add(book);
      }
   }
   return booksFound;
}
```

- In class Book

```
public boolean contains(String string) {
   return signature.contains(string) ||
          title.contains(string) ||
          author.contains(string)
}
```

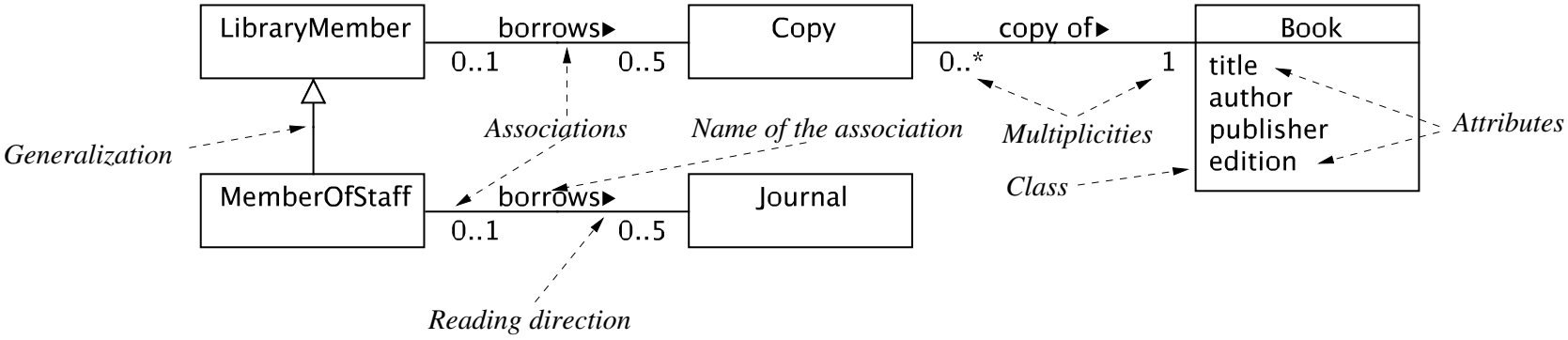
Advantages:

- Separation of concerns: LibraryApp is searching, Book is providing matching criteria
- Matching criteria can be changed without affecting the search logic

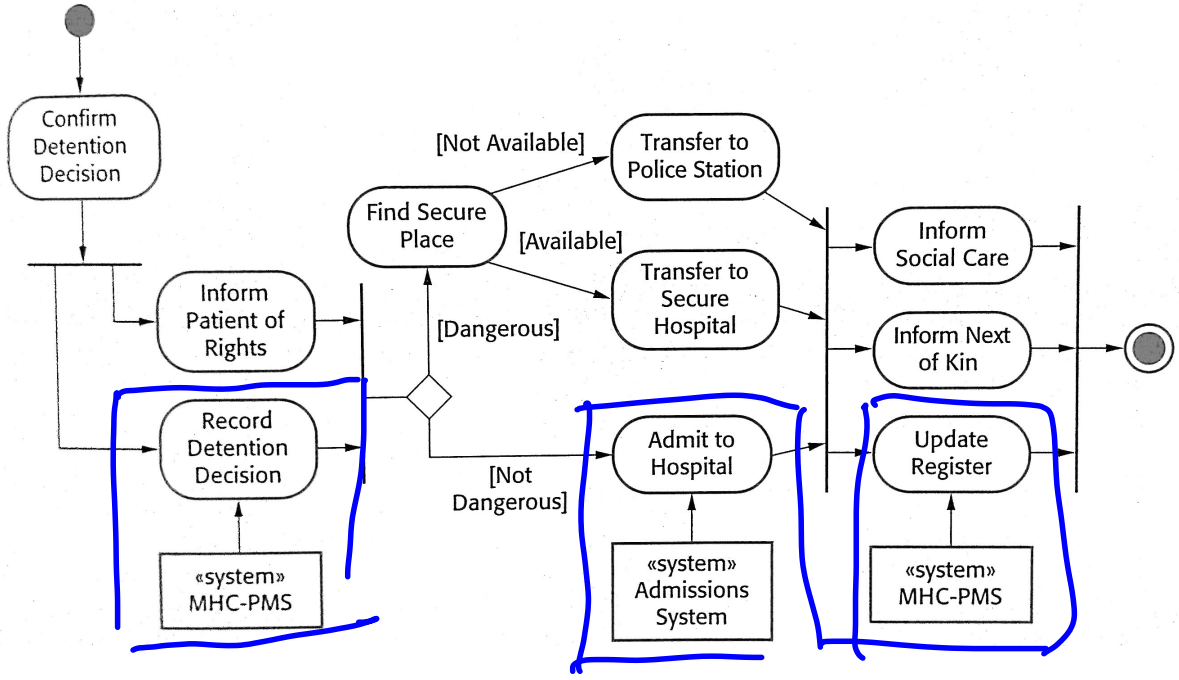
Contents

Recap

- Domain model

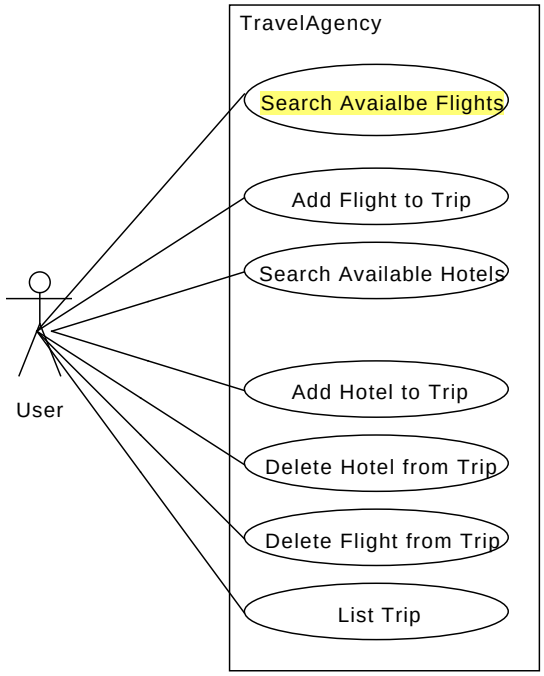


- Business Process



Use Case Recap

Use case diagram



Detailed use case

name: Search Available Flights

description: . . .

actor: User

main scenario:

1. The user provides information about the city to travel to and the arrival and departure dates
2. The system provides a list of available flights with prices and booking number

alternative scenario:

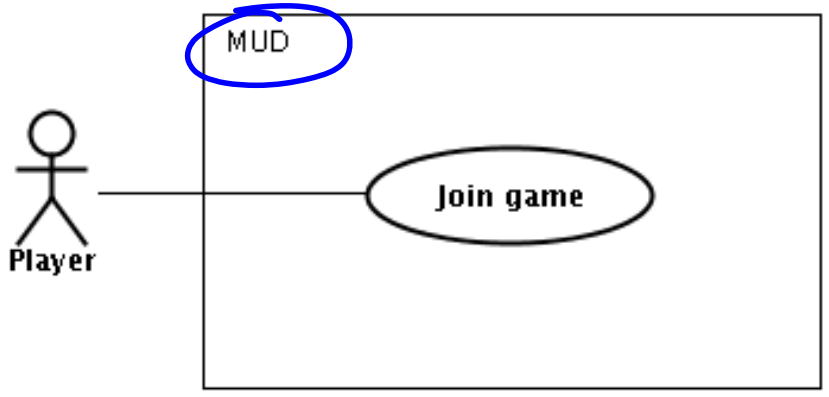
1a. The input data is not correct (see below)

   2. The system notifies the user of that fact and terminates and starts the use case from the beginning

. . .

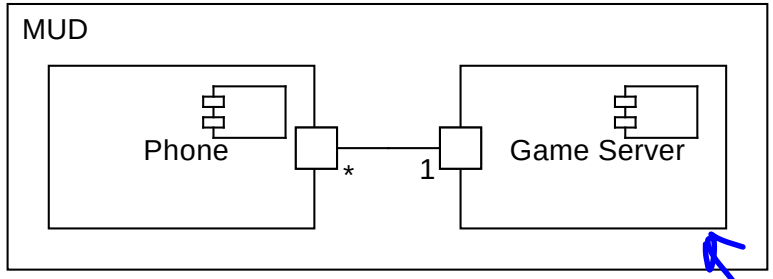
Use case refinement

- System boundary is important
- Deriving requirements of subsystems
- Example: Mobile Multi-User Dungeon (MUD) Game

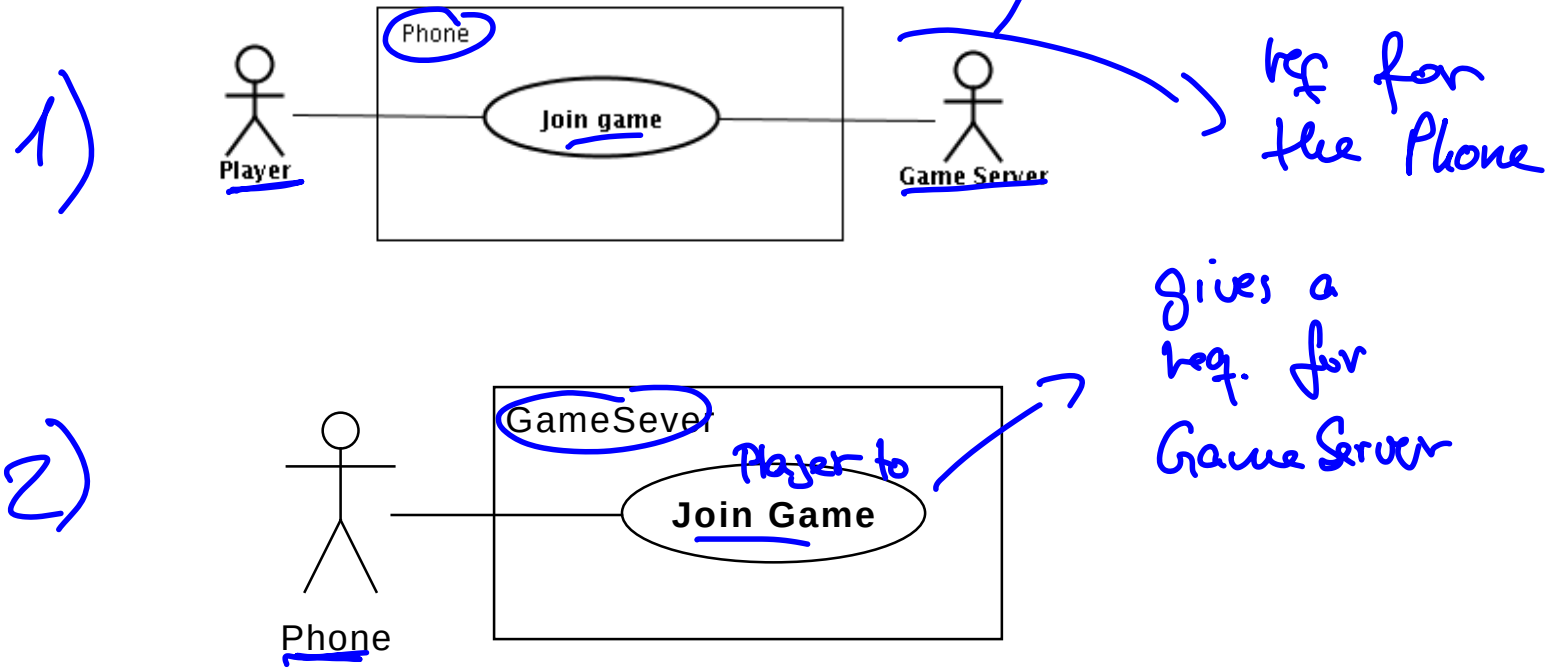


Use case refinement

- Decompose the system into subsystems



- Determine use cases for the subsystems



User stories

- Basic requirements documentation for agile processes
- Introduced with Extreme Programming: Simplifies use cases
- Contains a ”story” that the user tells about the use of the system
- Focus on features
  - ”As a customer, I want to book and plan a single flight from Copenhagen to Paris”.
  - Recommended, but not exclusive: ”As a <role>, I want <goal/desire> so that <benefit>”
- Documented by user story cards, i.e. index cards

Example of user stories

Each line is one user story:

- Students can purchase monthly parking passes online.
- Parking passes can be paid via credit cards.
- Parking passes can be paid via PayPal.
- Professors can input student marks.
- Students can obtain their current seminar schedule.
- Students can order official transcripts.
- Students can only enroll in seminars for which they have prerequisites.
- Transcripts will be available online via a standard browser.

=> one use case pay parking pass

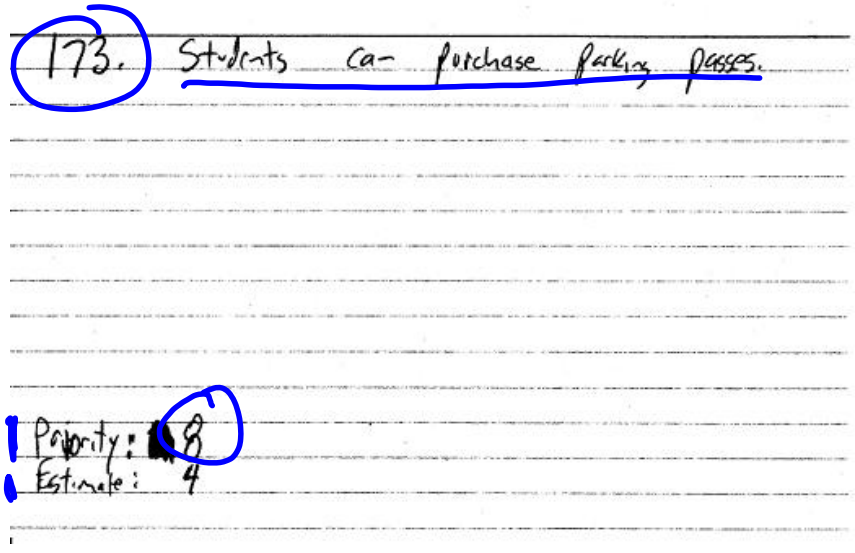
Scott Ambler 2003–2014 http://www.agilemodeling.com/artifacts/userStory.htm

Example of user story cards

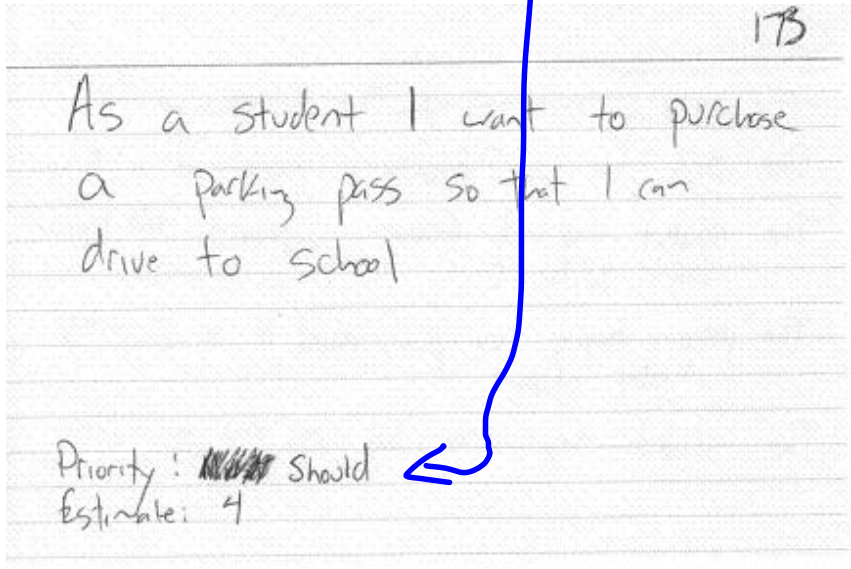
”Use the simplest tool possible”

→ index cards, post-its, . . .

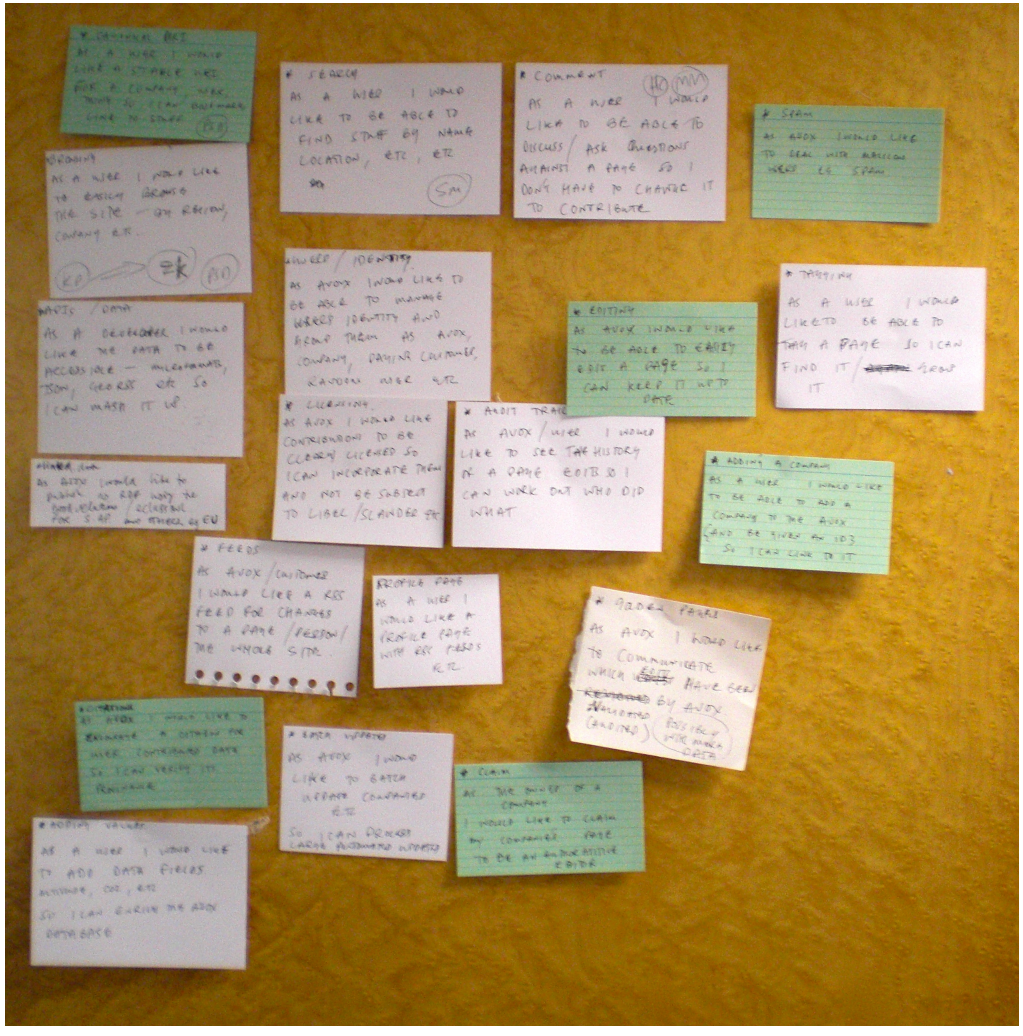
- electronically: e.g. Trello (`trello.com`)



Same card using ”As a <role>, I want <goal/desire> so that <benefit>” introduced by Mike Cohen and MoSCoW prioritization



Use the simplest tool possible



Paul Downey 2009 https://www.flickr.com/photos/psd/3731275681/in/photostream/

MoSCoW method for prioritizing requirements

Must have: Minimal usable subset to achieve the Minimal Vialble Product

Should have: Important requirments but not time critical, i.e. not relevant for the current delivery time frame

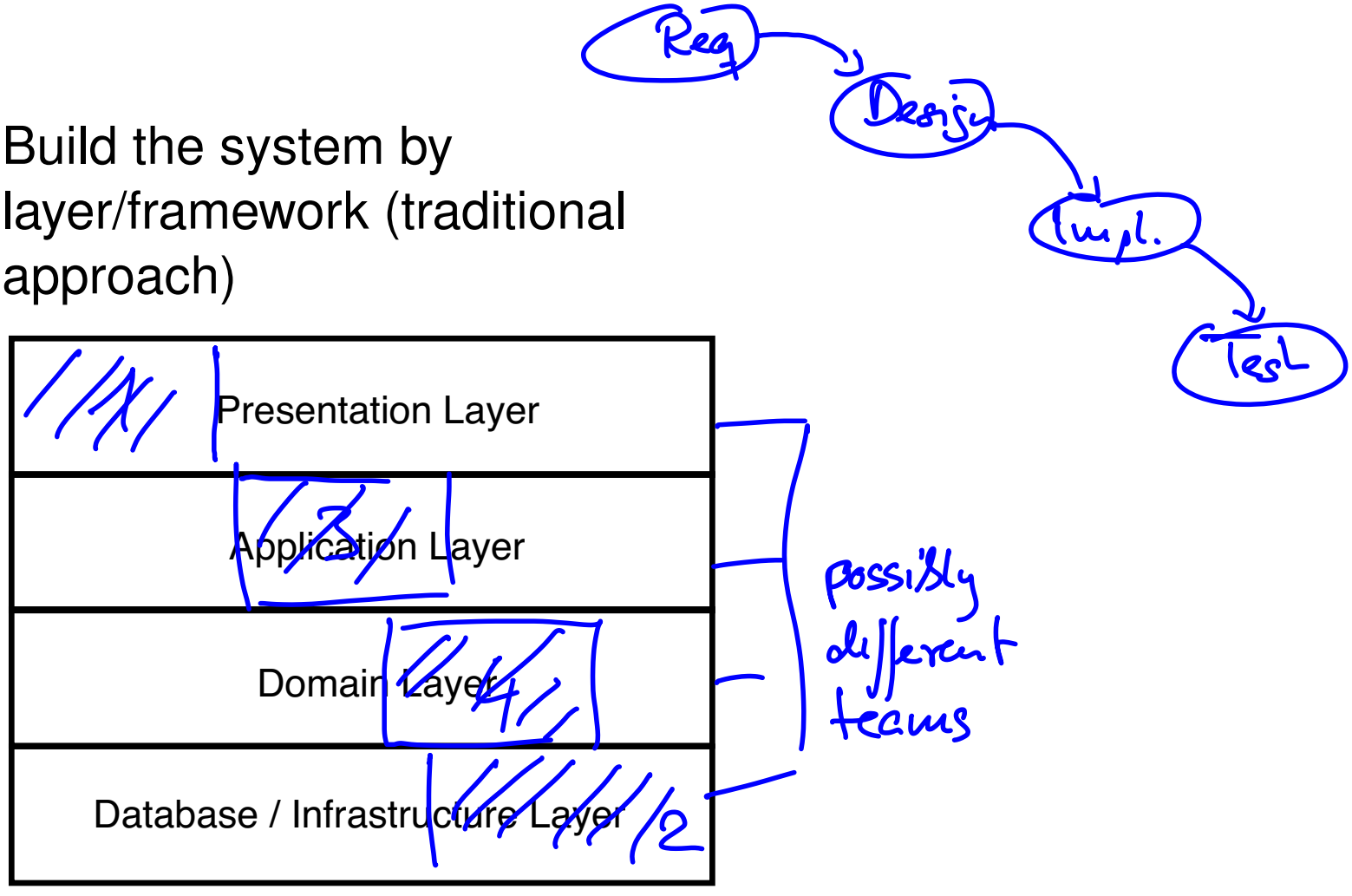
Could have: Desireable features; e.g. can improve usability

Won't have/Would like: Features explicitly excluded for the current delivery time frame

Wikipedia: `https://en.wikipedia.org/wiki/MoSCoW_method`

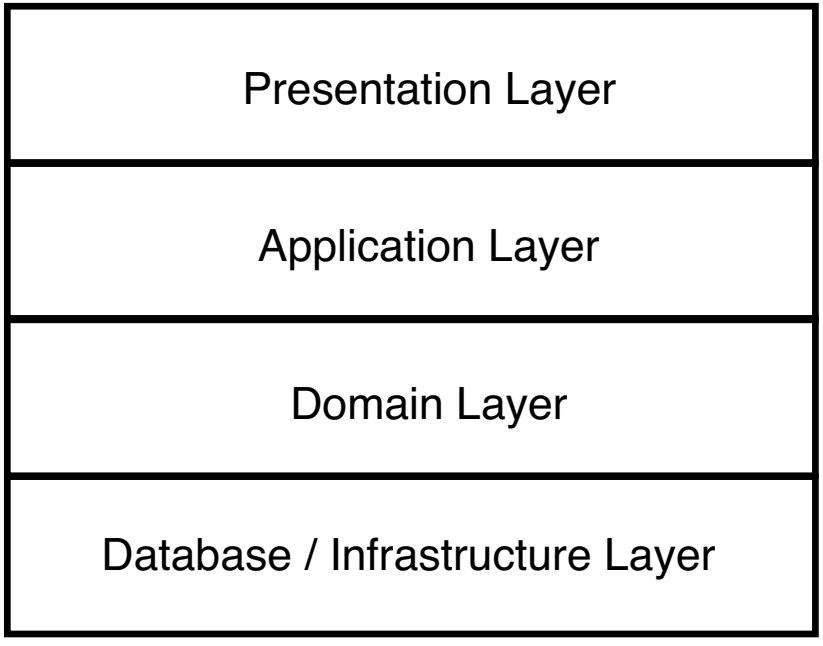
Two different ways of building the system

Build the system by layer/framework (traditional approach)

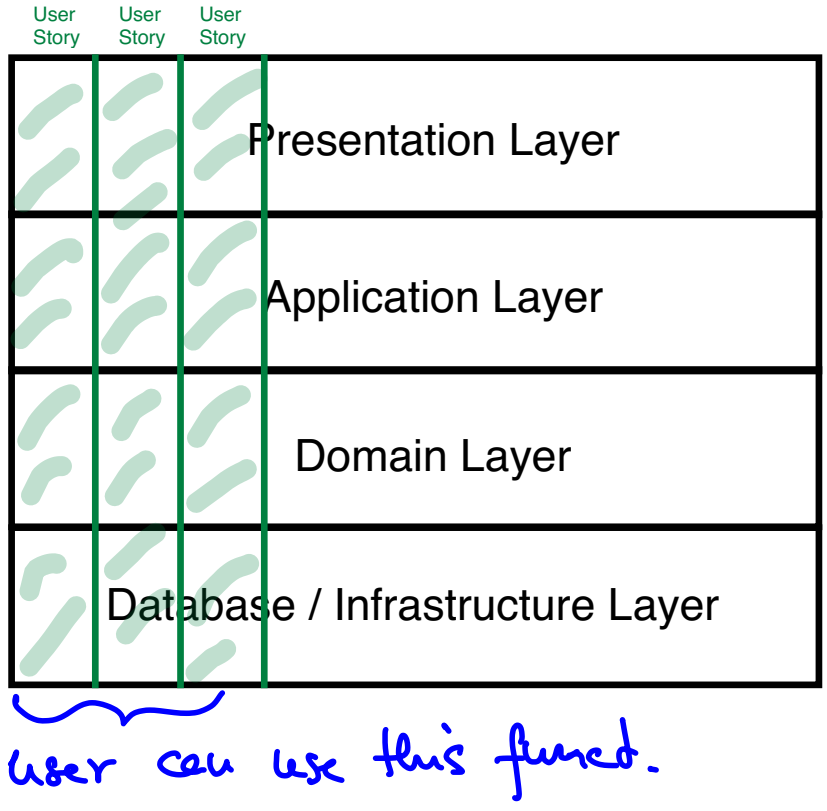


Two different ways of building the system

Build the system by layer/framework (traditional approach)



Build the system by functionality (Agile approach)



→ User story driven: After every implemented user story a functional system

Comparision: User Stories / Use Cases

User Case

- several abstract scenarios with one goal
- only functional requirements

Use Story

- one concrete scenario/feature
- Alternative scenarios of a use case are their own user story
- functional + non-functional requirement

  e.g. ”The search for a flight from Copenhagen to Paris shall take less than 5 seconds”

Comparision: User Stories / Use Cases

Use Case

- Advantage
  - Overview over the functionality of the system
- Disadvantage
  - Not so easy to do a use case driven development
  - E.g. Login use case

Use Story

- Advantage
  - Easy software development process: user story driven
- Disadvantage
  - Overview over the functionality is lost

Example: Login

Use case

name: Login

actor: User

main scenario

1 User logs in with username and password

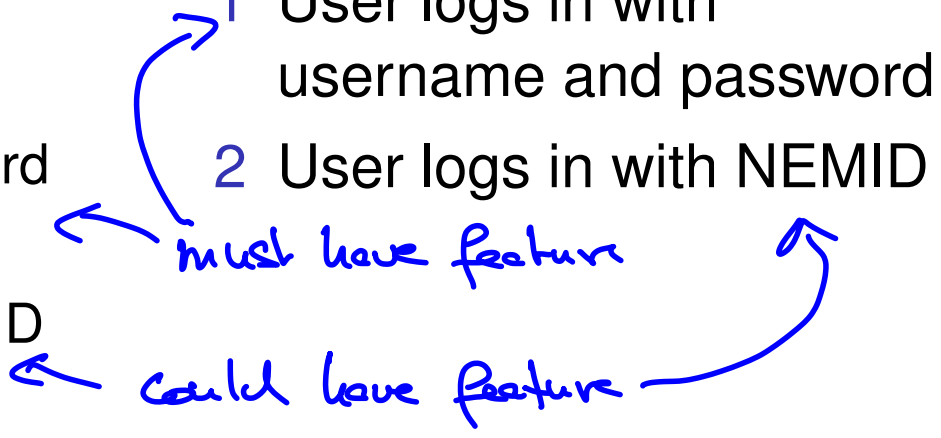
alternative scenario

1' User logs in with NEMID

User stories

1 User logs in with username and password

2 User logs in with NEMID



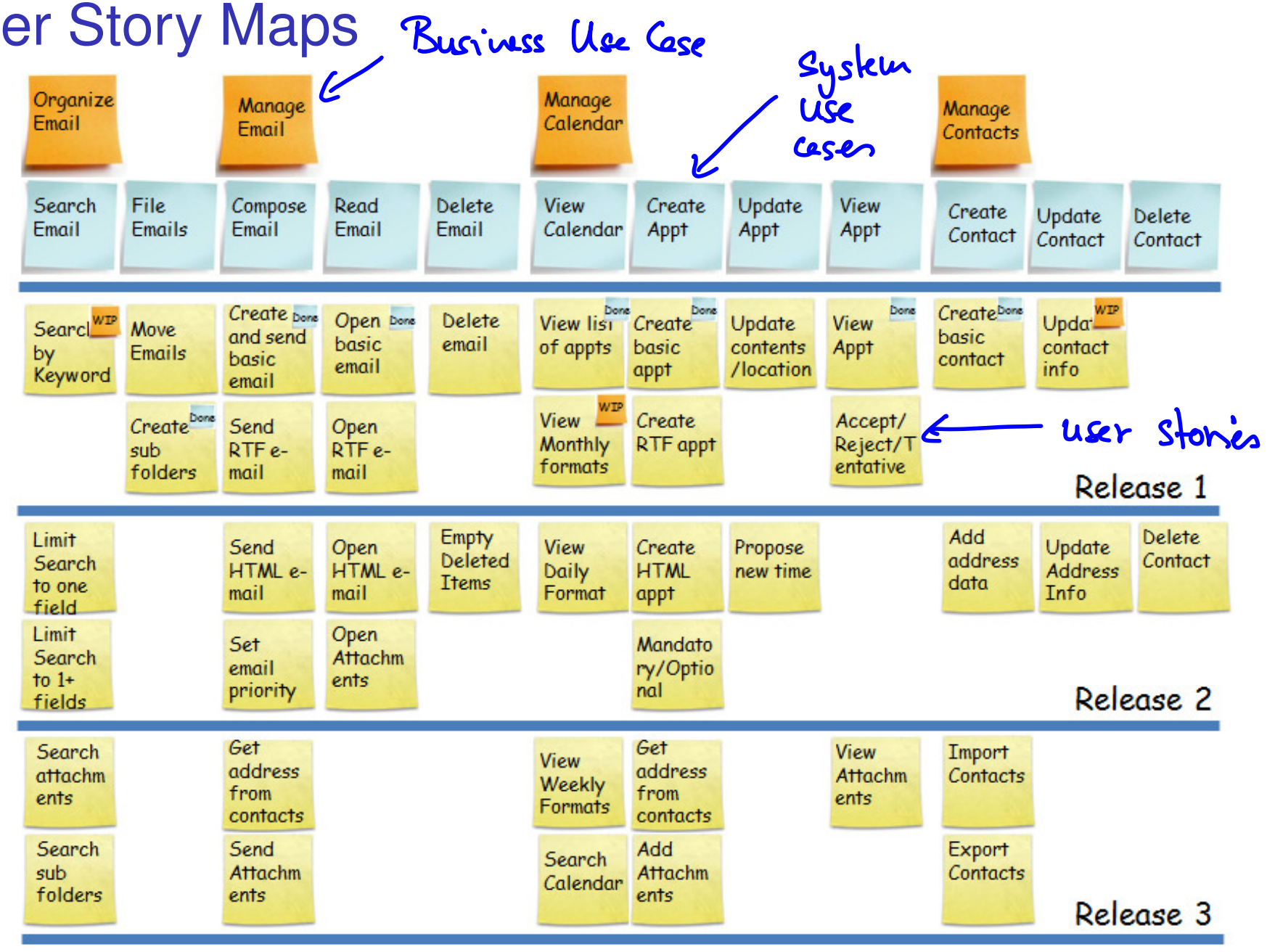
Combining Use Cases and User Stories

1. Use cases:
   - Gives an overview over the possible interactions

   → use case diagram
2. Derive user stories from use case scenarios (i.e. main- and alternative)
3. Implement the system driven by user stories
   - Note that different scenarios in use cases may have different priorities

     → Not necessary to implement all scenarios of a use case immediately

User Story Maps



Shrikant Vashishtha `http://www.agilebuddha.com/wp-content/uploads/2013/02/IMAG0144.png`

Problem: Changing Requirements

Requirements can change

- Feedback from designing the system, implementing it, and finally using it

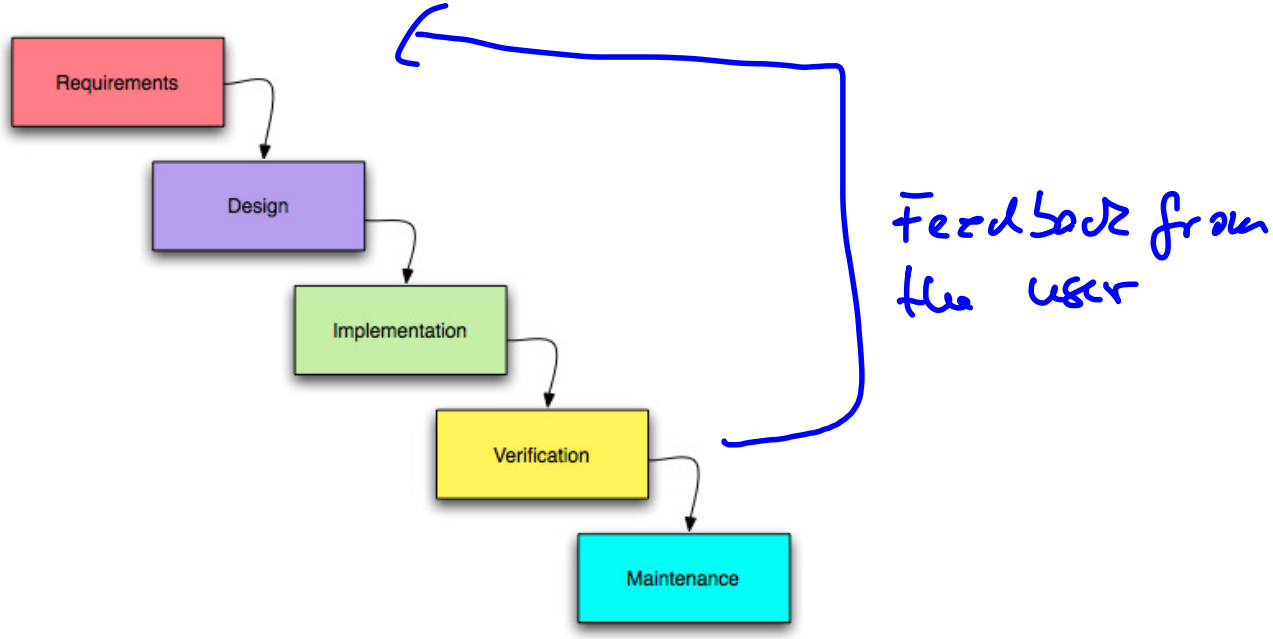
$\rightarrow$ clarification, changing, and new requirements

- The business case changes over time

Different type of software

- Standard Software: Similar systems done a 100 times
  - Requirements are stable
- New type of software. Experimental software. etc.
  - Requirements change a lot

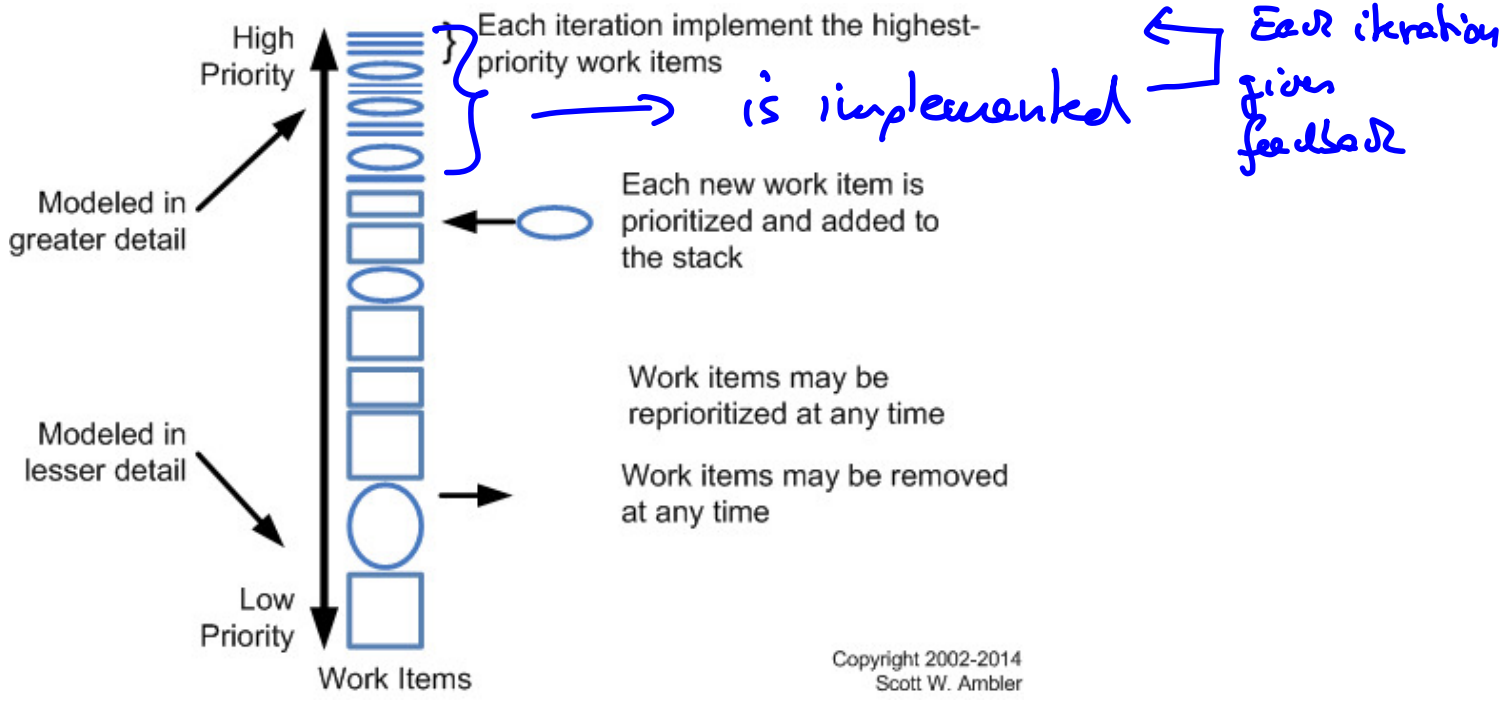
Requirements management: Waterfall



- Needs a defined requirement management process
  - E.g. Agreement of all stakeholders
- Changed / new requirements change the design and implementation
  - Cost of change not predictable

  → Avoid changing/new requirements if possible

Requirements management: 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 requiment that can not be implemented

Contents

Purpose of tests

- Goal: finding bugs

Edsger Dijkstra

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

- Types of bugs: requirement-, design-, implementation errors
- Types of testing:
  - validation testing
    - Does the software conform to the requirements?
    - **Have we built the right system?**
  - defect testing
    - Does the software has any unexpected behaviour (e.g. crashes)?
    - **Have we built the system right?**

Validation testing vs defect testing

Validation Test (Quality Assurance (QA))

- Start city is Copenhagen, destination city is Paris. The date is 1.3.2017. Check that the list of availabe flight contains SAS 1234 and AF 4245

Defect Test (QA and stress tests)

- Start city is Copenhagen, the name of the destination city contains the CrtI-L character.
- Check that the software reacts reasonable and does not crash

Types of tests

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

Acceptance Tests

- Tests defined by / with the help of the user
  - based on the requirements
- Traditionally
  - manual tests
  - by the customer
  - *after* the software is delivered
  - based on use cases / user stories
- Agile software development
  - automatic tests: JUnit, Behaviour Driven Development (BDD), Framework for Integrated Tests (Fit), . . .
  - created *before* the user story is implemented

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

test fails

Prerequisit: the password for the administrator is “adminadmin”

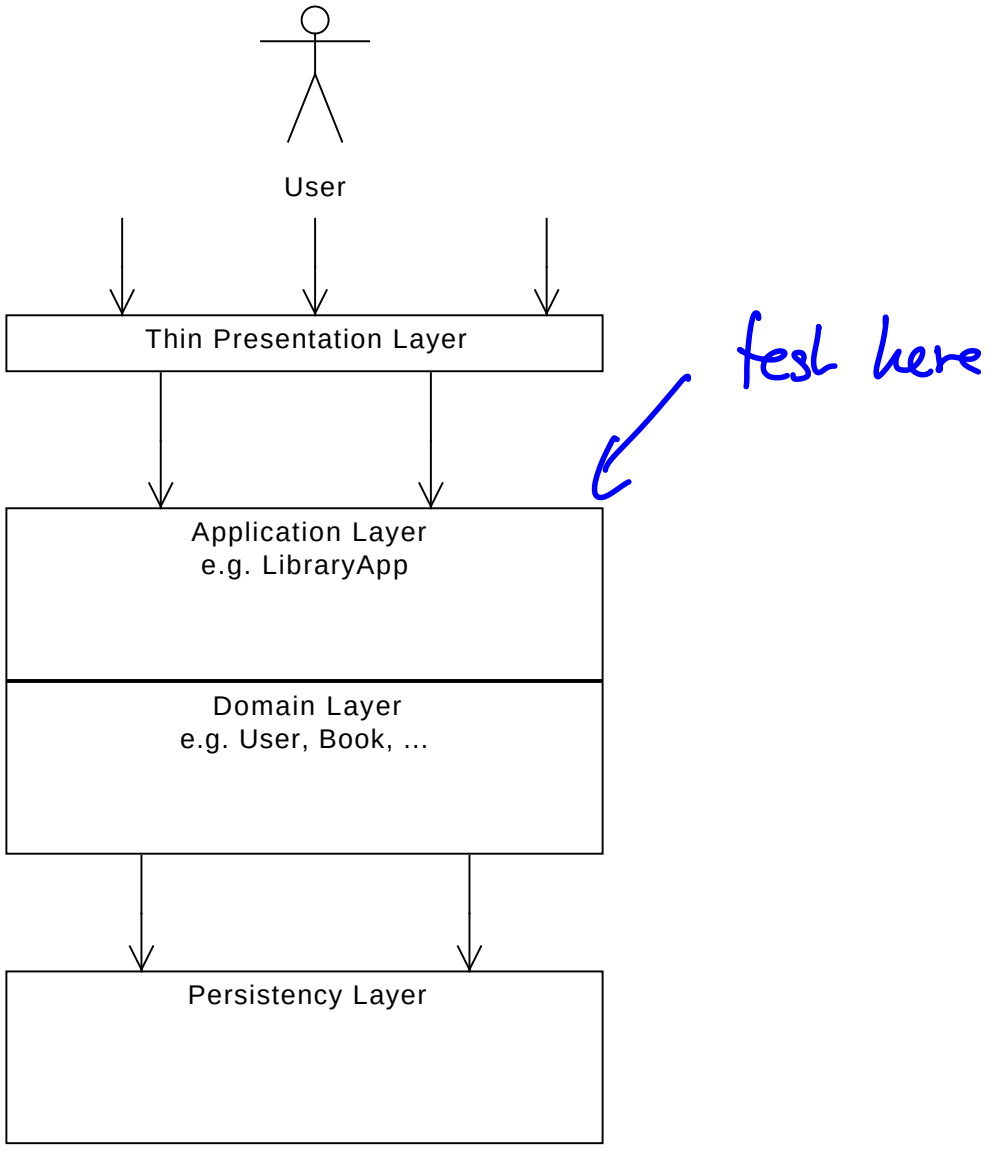
Input	Step	Expected Output	Fail	OK
	Startup system	“0) Exit” “1) Login as administrator”		
“1”	Enter choice	“password”		
“admin”	Enter string	“Password incorrect” “0) Exit” “1) Login as administrator”		

- Automatic test for the main scenario

Manual vs. automated tests

- Manual tests should be avoided
  - They are expensive (time and personal) to execute: Can't be run often
- Automated tests
  - Are cheap (time and personal) to execute: Can be run as soon something is changed in the system
    - → immediate feedback if a code change introduced a bug
    - → Regression tests
  - More difficult (but not impossible) when they include the UI
  - → Solution: Test under the UI
- 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

Testing under the UI



Automatic tests

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());
}
```

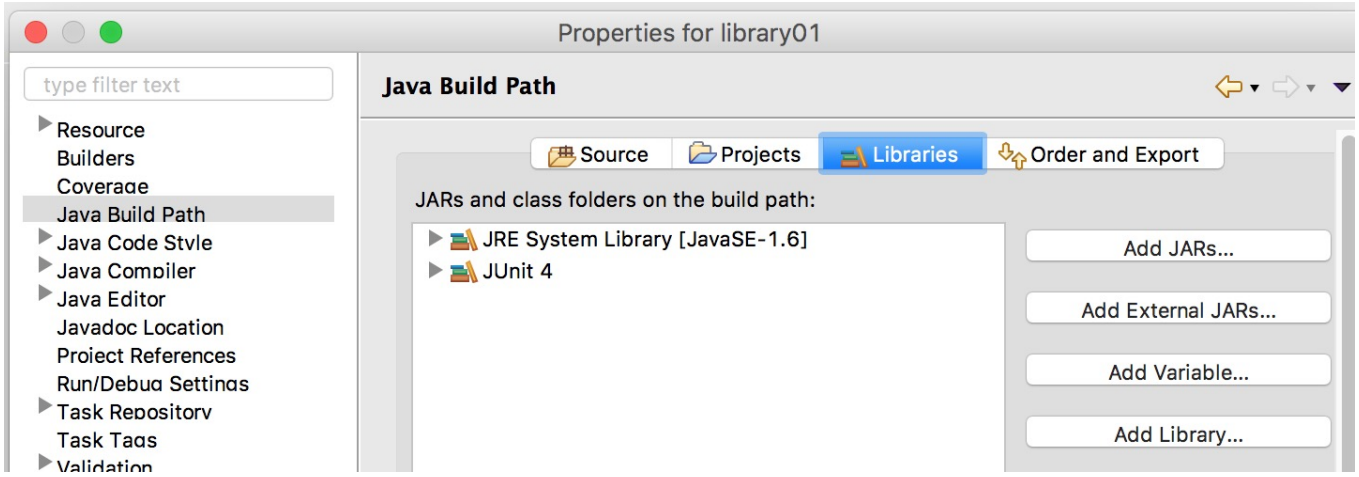
JUnit

- Framework for automated tests in Java
- Developed by Kent Beck and Erich Gamma
- Unit-, component-, and *acceptance* tests
- `http://www.junit.org`
- *x*Unit

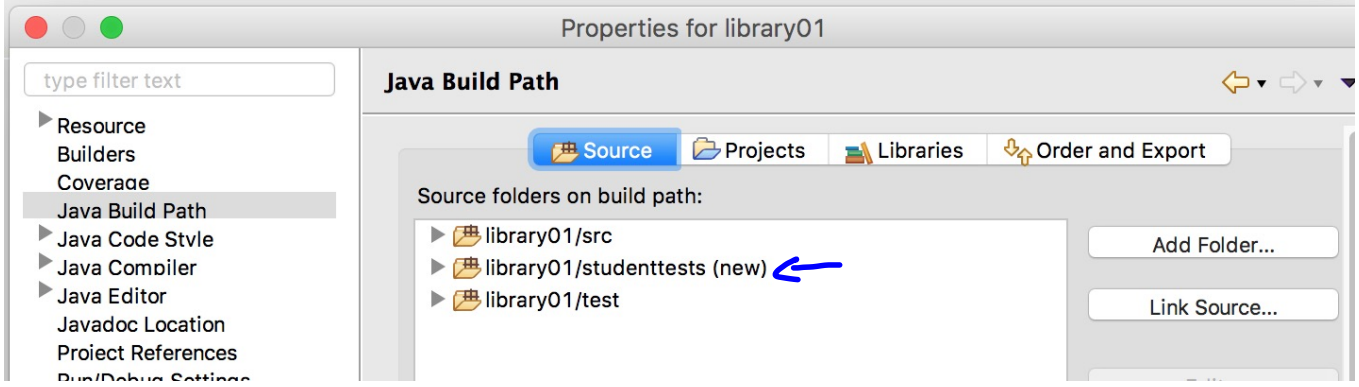
x Unit

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() {..}
    ...
}
```

setUp(); scenario1(); tearDown(); setUp() ....

Struture of test cases

- Test class = one use case
- Test method = one scenario
- Use inheritance to share sample data between use cases

```
public class SampleDataSetup {
    @Before()
    public void setUp() { .. }
    @After()
    public void tearDown { .. }
    ... }

public class TestBorrowBook extends SampleDataSetup {..}
```

JUnit assertions

General assertion

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

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

Specialised assertions for readability

1. `assertFalse(bexp)` = `assertTrue(!bexp)`
2. `fail()` = `assertTrue(false)`
3. `assertEquals(exp,act)` = `assertTrue(exp.equals(act))` "got act expected exp"
4. `assertNull(obj)` = `assertTrue(obj == null)`
5. `assertNotNull(obj)` = `assertFalse(obj == null)`

...

JUnit: testing for exceptions

- Test that method m() throws an exception MyException

```
@Test
public void testMThrowsException() {
    ...
    try {
        m();
        fail(); // If we reach here, then the test fails because
                // no exception was thrown
    } catch(MyException e) {
        // Do something to test that e has the correct values
    }
}
```

- Alternative

```
@Test(expected=MyException.class)
public void testMThrowsException() {..}
```

Contents

Summary

- Requirements
  - New use cases through system decomposition
  - User Stories vs Use Cases
    - Use Cases: Better overview of functionality; lets one think about alternative and error cases
    - User Stories: Simple scenarios, better for driving the software development
  - Changing Requirements: Requirments management
- Tests
  - Test to find bugs
  - Manual vs automated tests
  - Acceptance tests
  - JUnit

Exercises

- Homework for this week: continue with
  - `http://www2.imm.dtu.dk/courses/02161/2017/slides/exercise02.pdf`
- Still ongoing: programming exercises
  - `http://www2.imm.dtu.dk/courses/02161/2017/index2.html`