Construction of User Interfaces (SE/ComS 319)

Ali Jannesari

Jinu Susan Kabala

Department of Computer Science

Iowa State University, Spring 2021

INTRODUCTION TO TEST-DRIVEN DEVELOPMENT (TDD)

Outline

- Introduction to software process
  - Waterfall model
  - Agile process
- Test-driven development (TDD) – XP process model
  - XP development practices
  - Test-driven development (TDD)
  - Unit Testing Frameworks
    - Junit
    - Jest: JavaScript testing framework
  - XP/TDD mini project (Example)

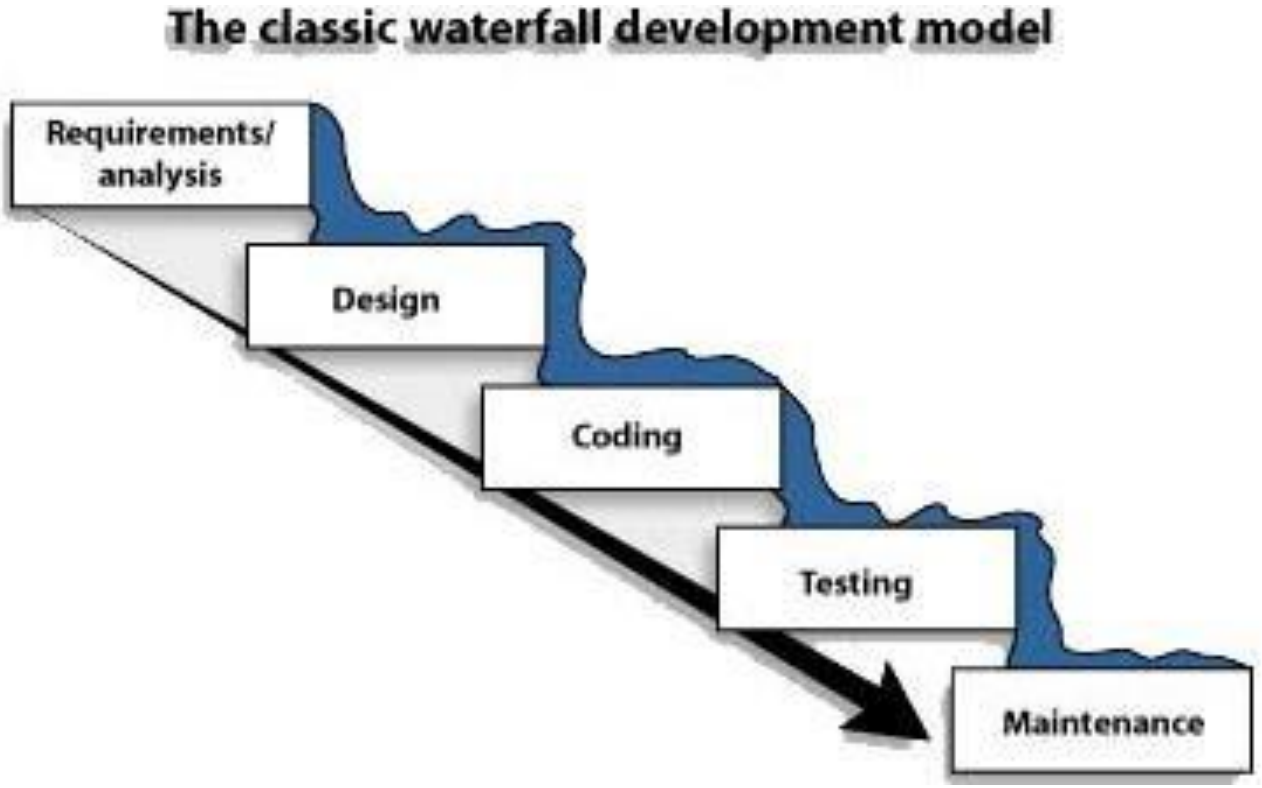
INTRODUCTION TO SOFTWARE PROCESS

Software engineering process models

- A sequence of activities that lead to the production of a software product
- There are many processes proposed
  - Waterfall
  - Incremental, evolutionary (Spiral), V-Model **(not covered)**
- Agile process model
  - Extreme programming (aka TDD)
  - Scrum **(not covered)**

Waterfall process model

- A sequence of separated phases
  - Activities in separate process phases



Waterfall model – phase 1

- Requirement & analysis
  - Where do we obtain the requirement?
    - Interview with customer
    - Questionnaires
    - On-site observations
  - Should we modify or refine the requirements?
    - What should we consider?
- Output
  - Requirement documents
  - **Actors** and **use case diagrams**

Waterfall model – phase 2

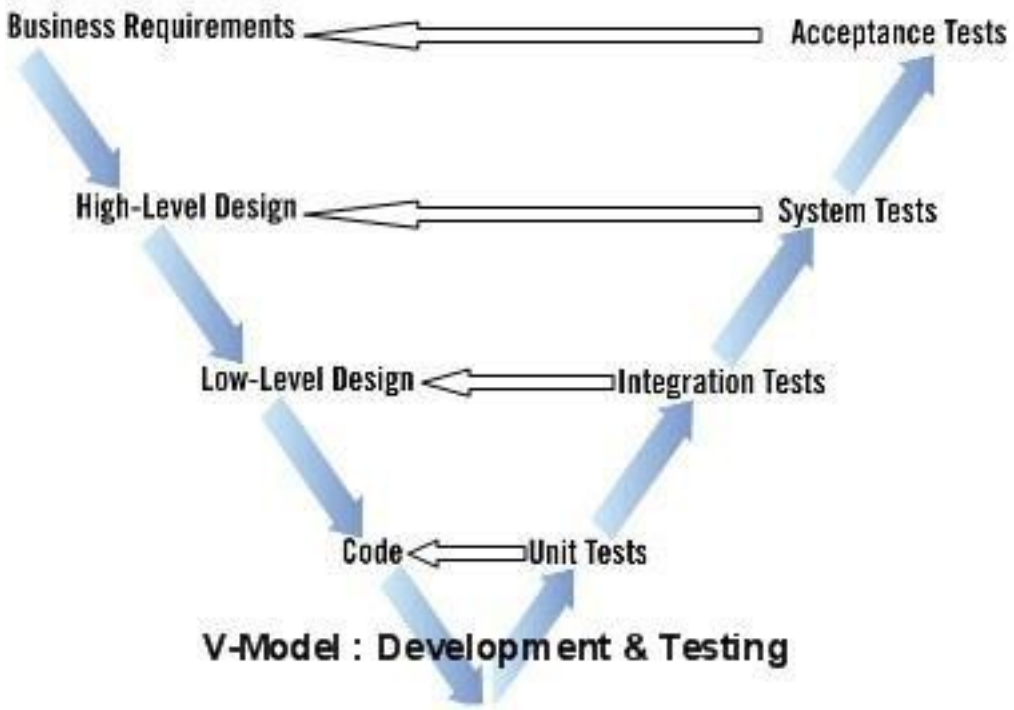
- Design
  - What need to be designed?
    - User interfaces (GUI)
    - Data structure (component design)
    - Module, API interface (architecture design/architectural styles)
- Output
  - Design document
  - **Class diagram**
  - Component diagrams
  - **GUIs**, etc.

Waterfall model – phase 3

- Implementation
  - Programming language, tools, frameworks/IDEs
  - Platforms, hardware, etc.
- Output
  - **Source code**

Waterfall model – phase 4

- Testing
  - **Unit (module) test**
  - Integration test
  - System test
  - Acceptance test
- Output
  - **Test results**
  - Source code



Waterfall model – phase 5

- Maintenance
  - New changes
  - New bugs, etc.
  - Regression tests
    - Verifies if the changed software still performs the same way
- Ratio of cost among phases
  - Maintenance can cost up to 2/3 of the total cost!

Problems with waterfall model

- Difficult to handle changes (not in model, high cost)
- Error fixing expensive
- Hard to estimate time
- Takes long time to deliver

- **Solution?**

AGILE PROCESS

Agile process

- Why Agile?
- Example
  - Airplane's control system needs 10 years to develop
    - Too much document
    - Too late code delivery
    - Not easy to deal with changes
    - Too much bureaucracy
    - Hard to finalize design without implementation
    - Hard to estimate time before design & imp.
    - Hard to finish planning (prioritize) without estimating time
- We need planning, planning, planning! □ planning game

Agile process

- Introduced in 2001
- The Agile manifesto
  - http://agilemanifesto.org/
- Better ways of developing software
- Core values:
  - **Individuals and interactions** over processes and tools
  - **Working software** over comprehensive documentation
  - **Customer collaboration** over contract negotiation
  - **Responding to change** over following a plan
- Culture of the whole team with shared responsibility and accountability
- Agile process model: XP (eXtreme Programming) aka **TDD**

XP process model (aka TDD)

- Intended to improve software quality and responsiveness to changing customer requirements
- **Each release** (iteration, weekly cycle) is **2 weeks** (aka **Sprint**)
- For each release:
  - Review & planning
  - Design (simple)
  - Implementation (**Test Driven Development – TDD**)
  - Following 12 key XP practices
    - Detailed design activity with multiple tight feedback loops
    - Effective implementation through testing and refactoring (continuously)
    - **In this course, we focus only on few development practices!**

12 key practices of XP process

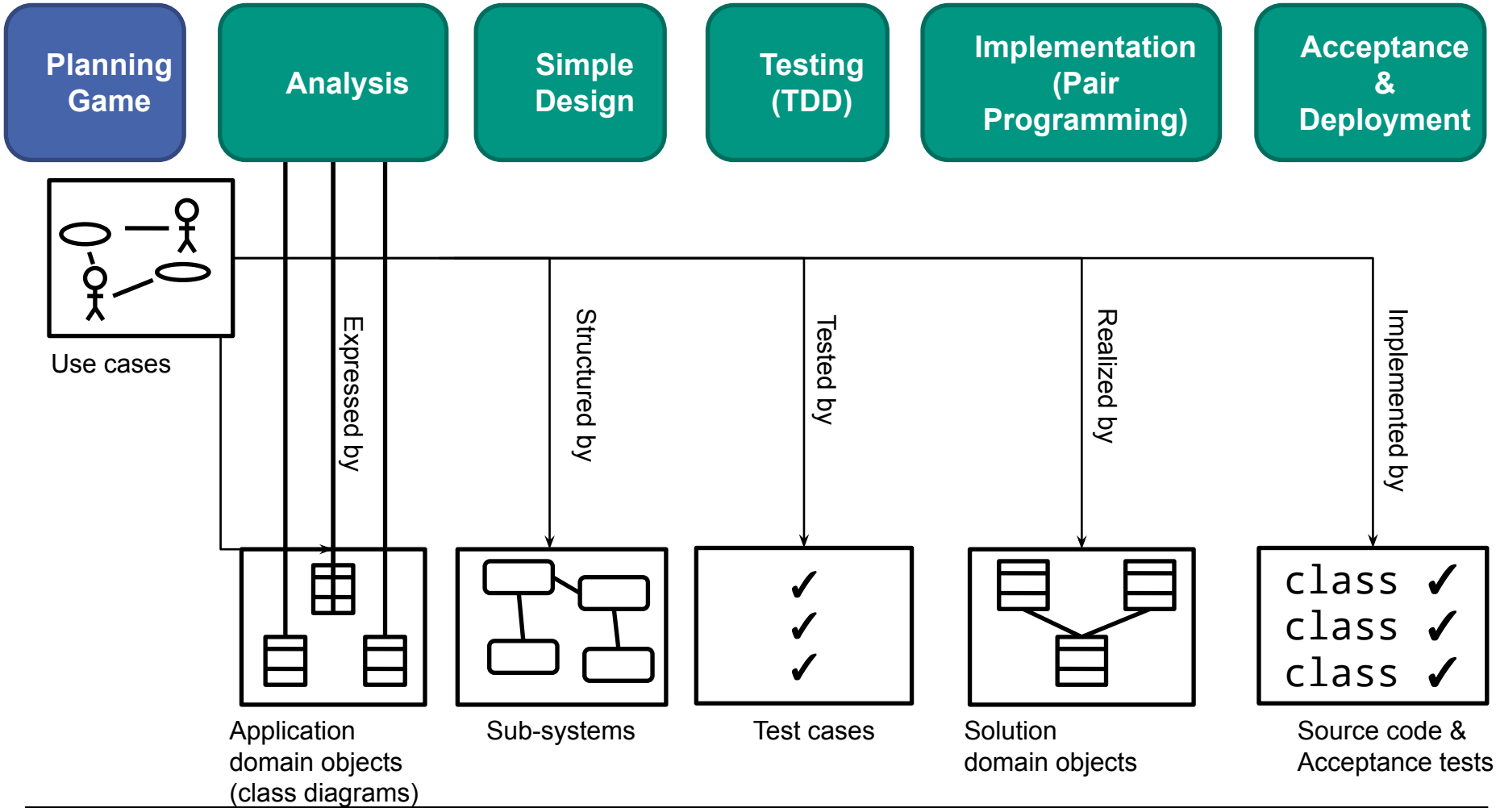
- **Planning game**
- **Small releases** (every 2 weeks)
- Metaphor
- **Simple design**
- **Testing (customer tests & Test-Driven Development – TDD)**
- Refactoring
- Pair programming
- Collective code ownership
- Continuous integration
- 40-hour week
- On-site customer
- Coding standards

TEST-DRIVEN DEVELOPMENT (TDD)

Implementation – Test-driven development (TDD)

- **TDD (test-driven development)**
  - Unit tests
  - Test suite
  - Regression testing & continuous integration
- Teams practice TDD by working in short cycles of **adding a test, and then making it work**
- Easy to produce code with 100 percent test coverage
- Each time a pair releases code to the repository, every test must run correctly

How to apply TDD?

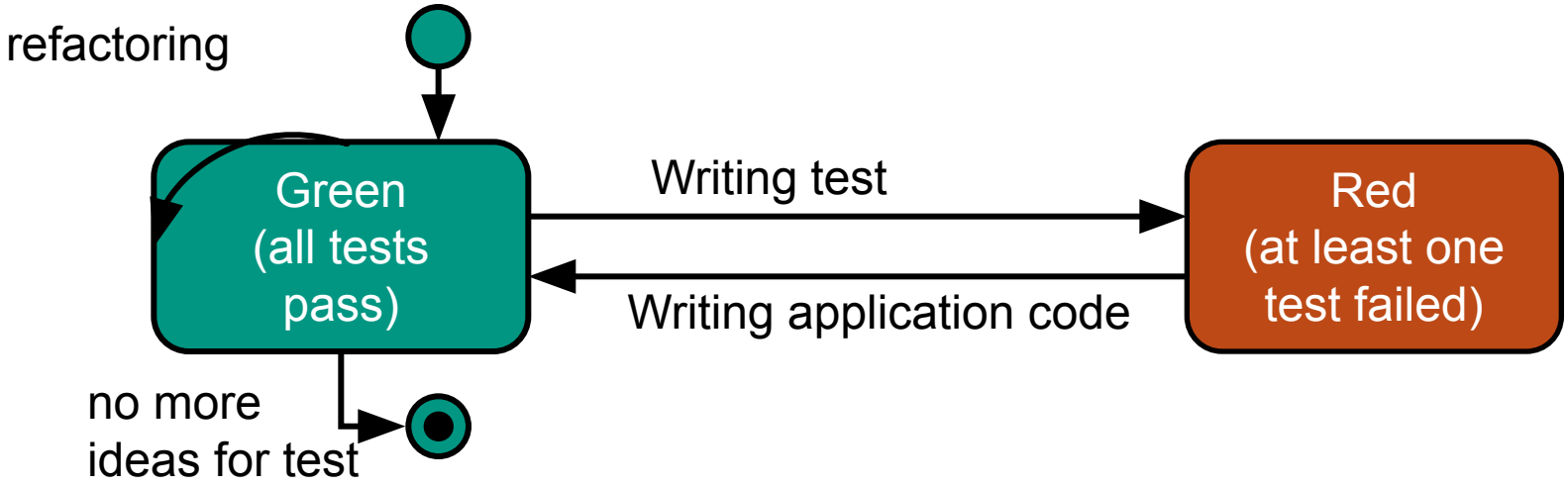


Test-driven development (TDD)

- Test / Code / Refactor Cycle: Motivate any behavioral change to the code through an automated test.
- Refactoring and simple design: Always put the code in the simple form.
- Continuous integration: Integrate the code as often as necessary.
- Pair Programming: Defeat the inner temptation.

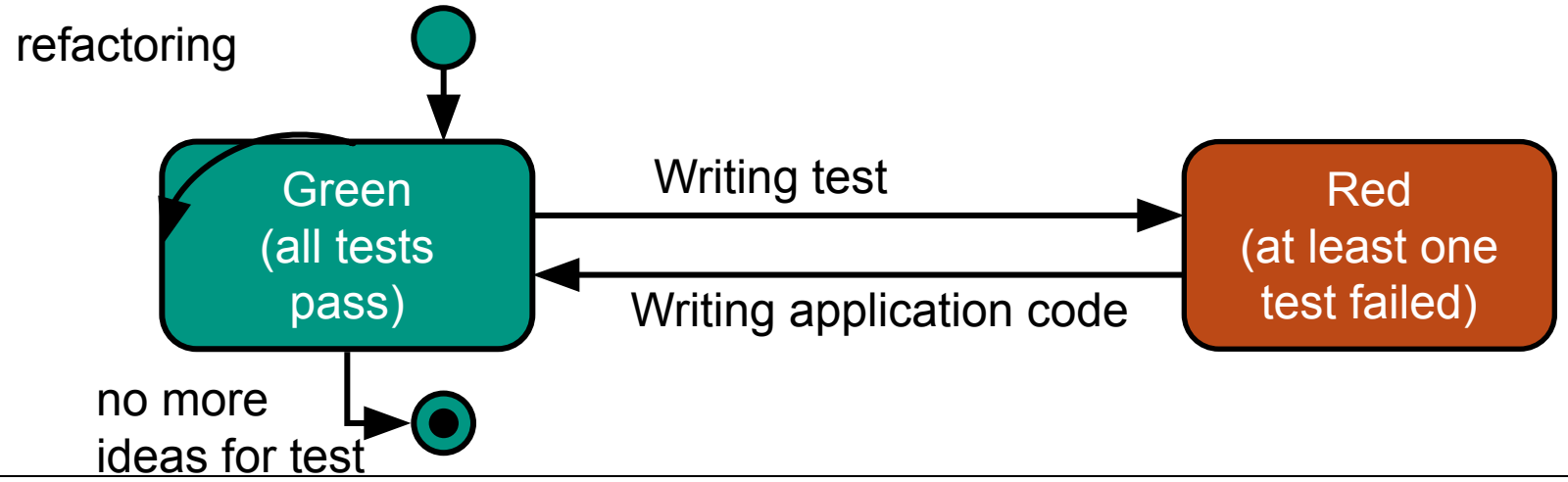
TDD Cycle

- State diagram idealized
- Test code before application code
- Small steps
- Incremental design



TDD Cycle – Test/Code/Refactor Cycle

- Green Red: Write a test that fails. If necessary, just write enough code that the test can be compiled.
- Red Green: Just write enough code for all tests to run successfully.
- Green Green: Eliminate duplication and other unnecessary code.
- Make it fail – make it work – make it better.



Test-driven development (TDD)

1. Write a failing test to prove code or functionality is missing from the end product.
2. Make the test pass by writing production code that meets the expectations of your test. The production code should be kept as simple as possible
3. Refactor your code.

Test-driven development (TDD)

- Refactoring means changing a piece of code without changing its functionality
  - Renaming, splitting large methods into smaller ones, removing duplicate code, …
- By seeing a test fails and then seeing it passes without changing the test, you're basically testing the test itself.

Unit Tests

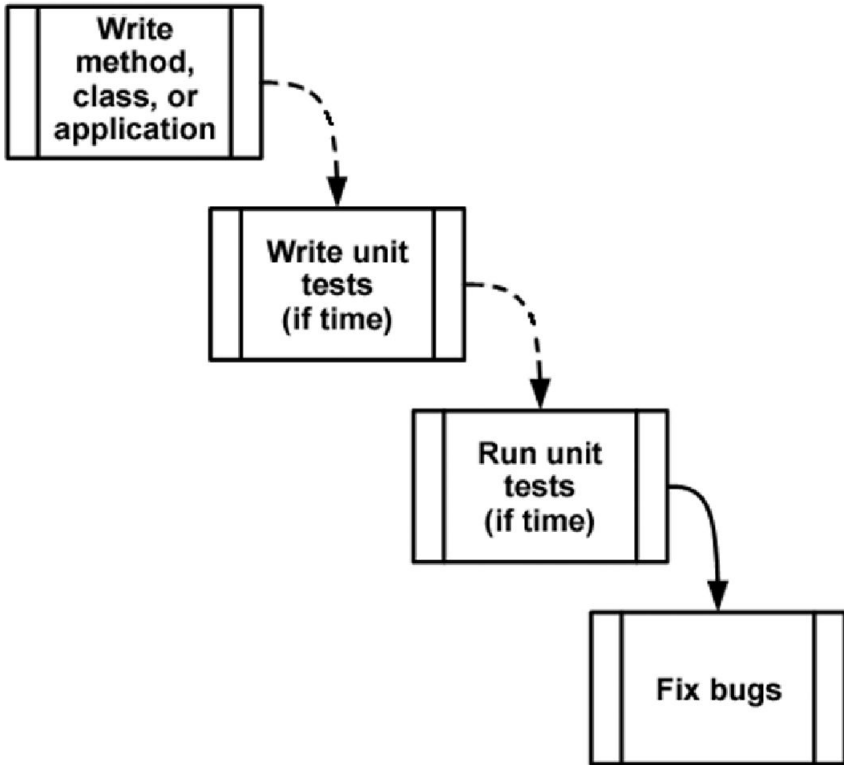
- Unit tests in XP (TDD) process…
- … are written by the developer himself/herself
- … give concrete feedback and security
- … enable safe changes
- … secure the existing functionality
- … must run at 100% with every code integration

Efficient testing

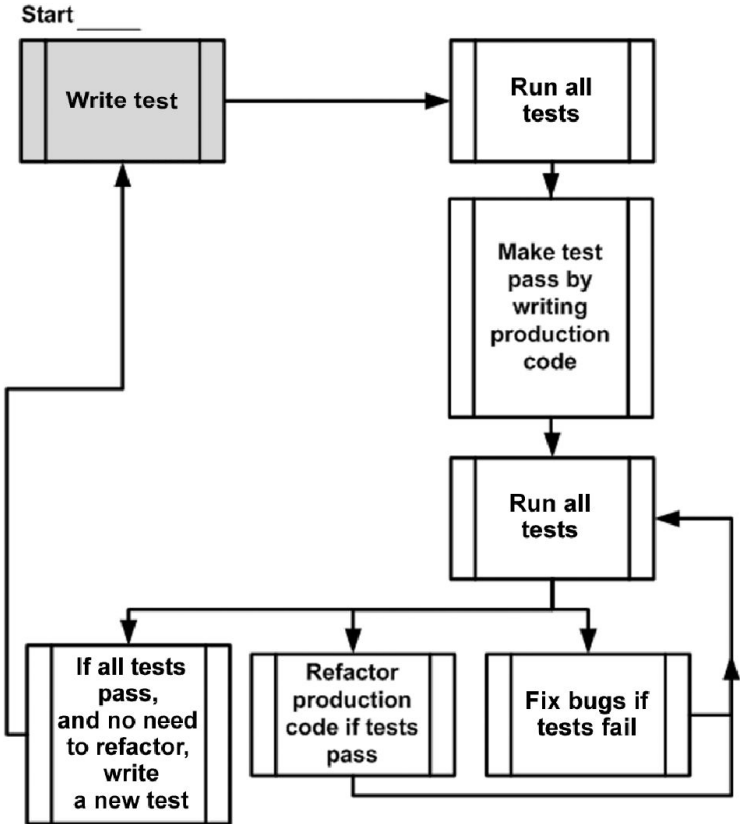
- Should be done as soon as possible for programming
- Is automated and thus repeatable
- Has to be fun
- Testing as often and as easy as compiling
- Finding mistakes, not proving faultlessness

TDD vs. conventional development and testing

- Conventional development and testing



- Test-driven development



Test-driven development

- Does help…
  - Design
  - Simplify production code
  - Code coverage
  - Understanding the problem at hand
  - Contract for development
  - You Always have some running code to show, Continuous delivery
  - Improves code quality
  - Incremental development

Does not help…

- Test Maintainability
- Test Readability

TDD

UNIT TESTING FRAMEWORKS

Unit Testing Frameworks

- **Junit**

  - Java Unit Testing Framework

- **Jest**

  - JavaScript Testing Framework

JUnit framework – Recap

- JUnit is Java framework for writing and running automatic unit tests
- Also available for many other programming languages (available in Eclipse)
- A JUnit *test* is a method contained in a class which is only used for testing. This is called a ***Test class***.
- To define that a certain method is a test method, annotate it with the **@Test** annotation
- You use an ***assert*** method, provided by JUnit to check an expected result versus the actual result!
  - These method calls are typically called *asserts* or *assert statements*

JUnit Test Class

- Contains (related) test cases in the form of methods.
- Holds references to the test objects to be tested
- The comparisons of target and actual values take place using assertions from the class **org.junit.Assert**.
- Defining test methods is done with annotations (**@Test**).
- Assert statements (methods):
  - **`assertEquals(Object shall, Object is)`**
  - **`assertTrue(boolean expression), etc`**

JUnit – Example

```
package demo;
import static org.junit.Assert.assertTrue;
import org.junit.Test;

public class BookLibraryTest {
  private BookLibrary lib;


  @Test public void bookIsInLibrary() {
    boolean b = lib.checkAvaiablity(“TestTitle");
    assertTrue(“TestTitle must be in the library.", b);
  }

}
```

Assertions are imported via **`import static`**

`@Test` defines a Test case.

Test-driven development (TDD) – Example: bank.Account

```
public Account(String customer)
public String getCustomer()
public int getBalance()
public void deposit(int amount)
public void withdraw(int amount)
```

For **`deposit`**, **`withdraw`** only positive values are allowed, otherwise throw an exception **`IllegalArgumentException`**

We think about first test cases…

Create new (**Account**) for customers.

Make a (**deposit**).

Make a (**withdraw**).

Transfer between two accounts.

Forbid negative amounts.

We design a test that should fail first

```
public class AccountTest {
   @Test
 public void testCreateAccount() {
     Account a = new Account("Customer");
     assertEquals("Customer",
 a.getCustomer());
     assertEquals(0, a.getBalance());
   }
}
```

We are currently writing so much code that the test can be compiled

```
public class Account {
   public Account(String customer) {
   }
   public String getCustomer() {
      return null;
   }
   public int getBalance() {
      return 0;
   }
}
```

We check if the test fails

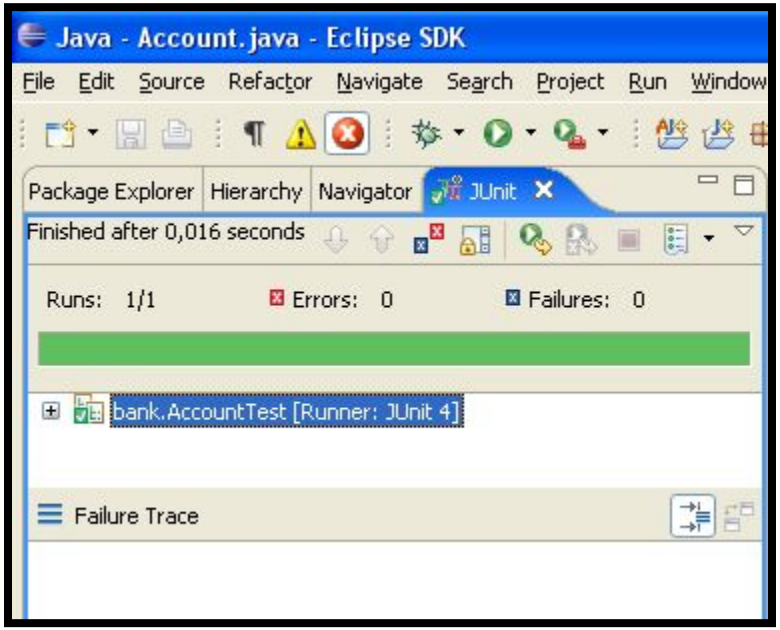
We are currently writing so much code that the test should be fulfilled!

```
public class Account {
   public Account(String customer) {
   }
   public String getCustomer() {
      return "Customer";
   }
   public int getBalance() {
      return 0;
   }
}
```

We do not write more code than the tests claim because it would be unspecified and unsecured.

 The goal is to reach the green, safe terrain as fast as possible.

We check if the test goes through

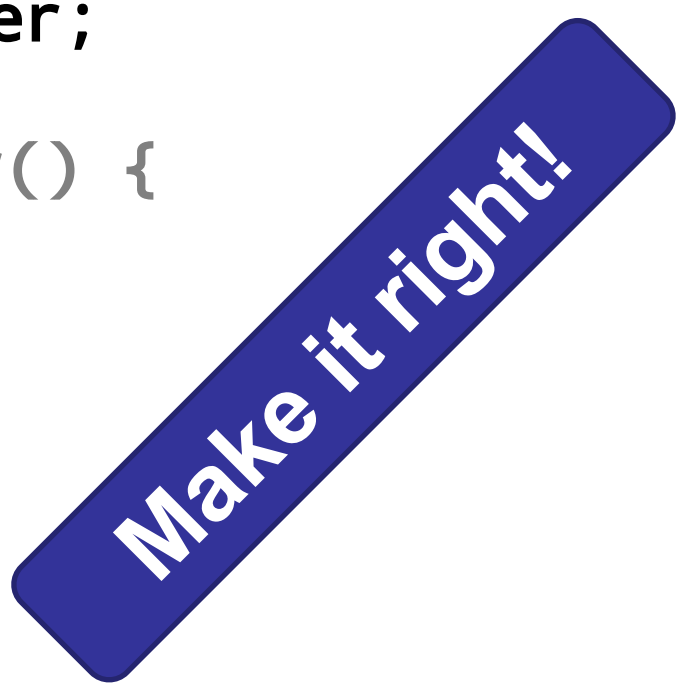


We remove duplication – but where is it?

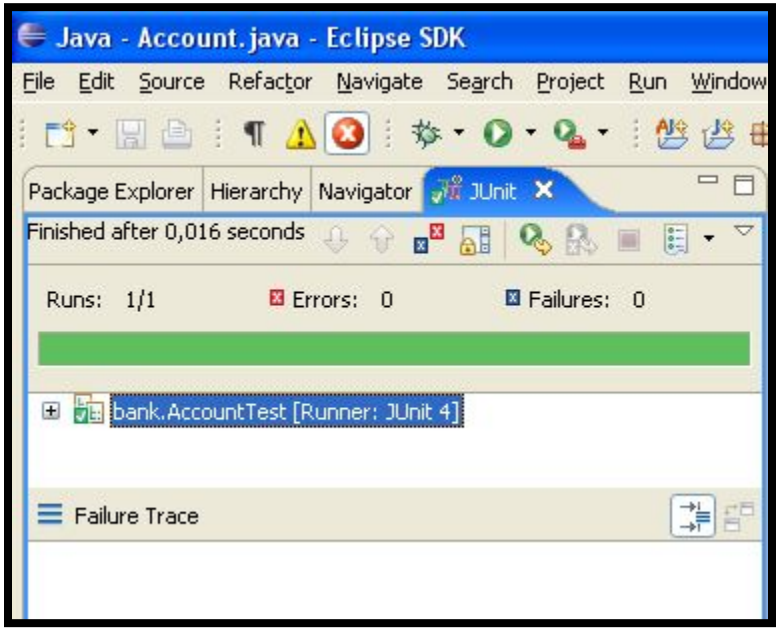
```
public class Account {
    public Account(String customer) {
    }
    public String getCustomer() {
        return "Customer";
    }
    public int getBalance() {
        return 0;
    }
}
public class AccountTest {
    @Test public void testCreateAccount() {
      Account a = new Account("Customer");
      assertEquals("Customer", a.getCustomer());
      assertEquals(0, a.getBalance());
    }
}
```

We remove duplication – Refactoring

```
public class Account {
   private String customer;
   public Account(String customer) {
      this.customer = customer;
   }
   public String getCustomer() {
      return customer;
   }
   public int getBalance() {
      return 0;
   }
}
```



We check if the test is still running



Tests and code in interplay

- The failing test decides which code to write next, to drive the development of the program logic.
- Based on the code we have written so far, we decide which test we will tackle next in order to further the development of the design.

Selection of the next test case

Create new (**Account**) for customers.

Make a (**deposit**).

Make a (**withdraw**).

Transfer between two accounts.

Forbid negative amounts.

Next Test: Deposit

```
public class AccountTest {
  […]
  @Test public void testDeposit() {
    Account a = new Account("Customer");
    a.deposit(100);
    assertEquals(100, a.getBalance());
    account.deposit(50);
    assertEquals(150, a.getBalance());
  }
}public class Account {
    […]
   private int balance = 0;

   public int getBalance() { return balance; }

    public void deposit(int amount) { balance += amount; }
 }
```

Jest Unit Testing Frameworks

- Jest: JavaScript Testing Framework maintained by Facebook
  works with projects using Babel, TypeScript, Node.js, React, Angular and Vue.js.
  - Also code coverage, mock functions
- Reference: https://jestjs.io
- Architecture: https://jestjs.io/docs/en/architecture (51 min video)
- Installation: npm install --save-dev jest

Jest Unit Test Example

File: sum.js

```
function sum(a, b) {
      return a + b;
}
module.exports = sum;
```

Function to add two numbers saved as sum.js

File: sum.test.js

```
const sum = require('./sum');

test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3);
});
test('adds 0 + 1 to equal 1', () => {
  expect(sum(0, 1)).toBe(0);
});
```

test indicates Unit test to test the sum function. Uses **expect** and **toBe** to test that two values were exactly identical

Running the example

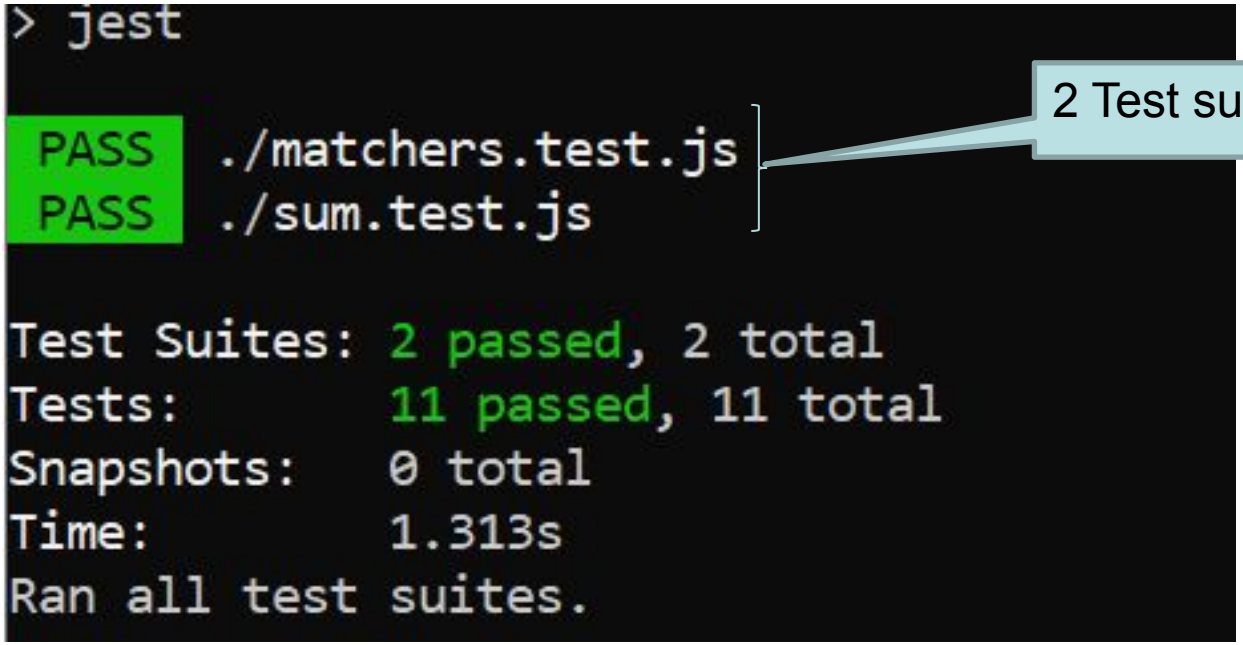
> npm run test

```
FAIL ./sum.test.js
  √ adds 1 + 2 to equal 3 (2ms)
  × adds 0 + 1 to equal 1 (2ms)

  ● adds 0 + 1 to equal 1

    expect(received).toBe(expected) // Object.is equality

    Expected: 0
    Received: 1

       6 |
       7 | test('adds 0 + 1 to equal 1', () => {
    >  8 |   expect(sum(0, 1)).toBe(0);
         |                     ^
       9 | });
      10 |

      at Object.<anonymous> (sum.test.js:8:21)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 passed, 2 total
Snapshots:   0 total
Time:        1.35s
Ran all test suites.
```

Test Fails: Sum(0,1) evaluates to **1**
But test expects exact value **0**

Test Suite & Test Cases

- Test case: Test method
- Test Suite: Collection of test cases



Matchers

Jest uses "**matchers**" to let you test values in different ways:

- toBe, toEqual
- toBeNull, toBeDefined, toBeUndefined, toBeTruthy, toBeFalsy
- toBeGreaterThan, toBeGreaterThanOrEqual, toBeLessThan, toBeLessThanOrEqual
- toBeCloseTo
- toMatch
- toContain
- toThrow

```
test('adding positive numbers is not zero', () => {
  for (let a = 1; a < 10; a++) {
    for (let b = 1; b < 10; b++) {
      expect(a + b).not.toBe(0);
    }
  }
});
```

- Use **not** to check the false condition e.g.: **not.toBe**

https://jestjs.io/docs/en/using-matchers

https://jestjs.io/docs/en/expect

Setup & Teardown

- Setup initial state and call cleanup in teardown
- Function called before and after each test is executed
- **beforeEach**, **afterEach**: same setup and teardown method called for each test
- **beforeAll**, **afterAll**: one-time setup and teardown called for all tests

```
beforeEach(() => {
  initializeCityDatabase();
});

afterEach(() => {
  clearCityDatabase();
});

test('city database has Vienna', () => {
  expect(isCity('Vienna')).toBeTruthy();
});

test('city database has San Juan', () => {
  expect(isCity('San Juan')).toBeTruthy();
});
```

https://jestjs.io/docs/en/setup-teardown

https://jestjs.io/docs/en/api

Test Driven-Development

PLANNING GAME, SIMPLE DESIGN & SMALL RELEASES

12 key practices of XP process

- **Planning game**
- **Small releases** (every 2 weeks)
- Metaphor
- **Simple design**
- **Testing (customer tests & Test-Driven Development – TDD)**
- Refactoring
- Pair programming
- Collective code ownership
- Continuous integration
- 40-hour week
- On-site customer
- Coding standards

Planning game

- No more freezing requirement
  - No more requirement document
- No more exact prediction
  - Predict what will be accomplished by the due date
  - Determine what to do next
- **Story cards** (user stories)
  - Customer presents required features … (in our project: team members or TA)
  - Developers estimate difficulty …
  - Revise regularly

Planning game (2)

- XP Release Planning
  - Customer describes required features
  - Programmers estimate difficulty
  - Imprecise but revised regularly
- XP Iteration Planning
  - **Two-week** iterations
  - Customer presents features required
  - Programmers break features down into tasks
  - Team members sign up for tasks
  - **Running software** at end of each iteration

Story

- A specific system behavior from the user's perspective
- No exact specification
- Basis for discussion
- **Implementable in an iteration**

How stories are produced?

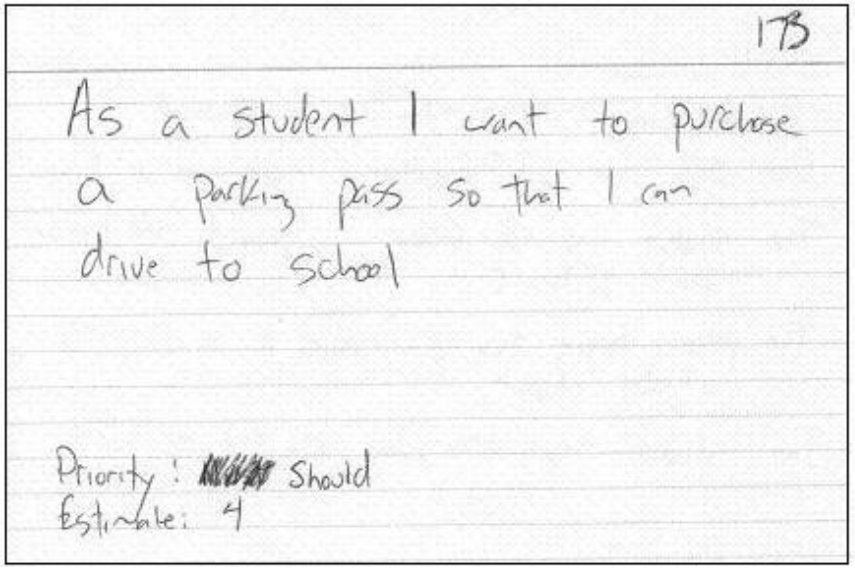
- Customer tells something about how to use the system
- Developers ask comprehension questions
- Customer writes story in his own words
- Over time, stories are discarded, rewritten, shared and merged

(**for our group project a TA or a team member can serve as a customer**)

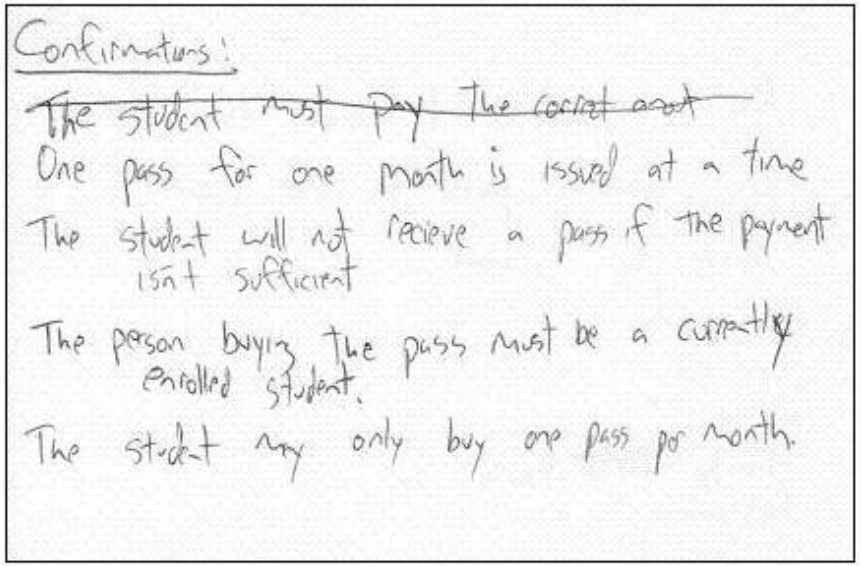
Story card – Example : Parking pass system (1)

- ID and Task Description
- Priority
- Estimation
- Confirmation

Front of Card



Back of Card



Source: http://www.agilemodeling.com/artifacts/userStory.htm

Story card – Example : Parking pass system (2)

Examples of user stories for a parking pass system:

- Students can purchase monthly parking passes online.
- Parking passes can be paid via credit cards.
- Parking passes can be paid via PayPal.

Story card – Example: Seminar management system

Example of user stories for a seminar management system:

- Professors can input student grades.
- 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.

How to end an iteration? – Small releases

- Team releases running, tested software by the end of every iteration (**two weeks**)
- Releases are small and functional
- The customer can evaluate or release to end users and provide feedback
- **Important**: The software is visible and given to the customer at the end of every iteration

(**for our group project a TA (instructor) serves as a customer and check the software at the end of every iteration!**)

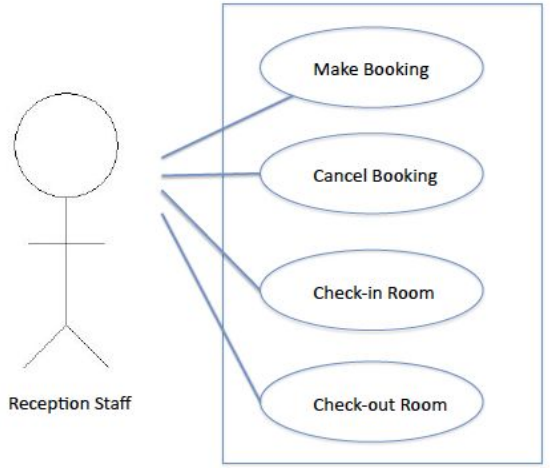
Simple design (1)

- Principle – KIS (keep it simple)
  - Build software to a simple design: Through programmer testing and design improvement, keep the software simple and the design suited to current functionality
  - Not a one-time thing nor an up-front thing
  - The requirements will change tomorrow, so only do what's needed to meet today's requirements
- Teams design and revise design through refactoring in the course of the project

Simple design (2)

- Output:
  - CRC (Class-Responsibility-Collaboration – **not covered in this class**)
  - **UML diagrams**: Divides structural and behavioral modelling from each other
  - We can model using UML use-case/class/component diagrams
    - Such translations will surely filter much of your misunderstandings and errors.

Example: Use Case diagram – Reception Scenario

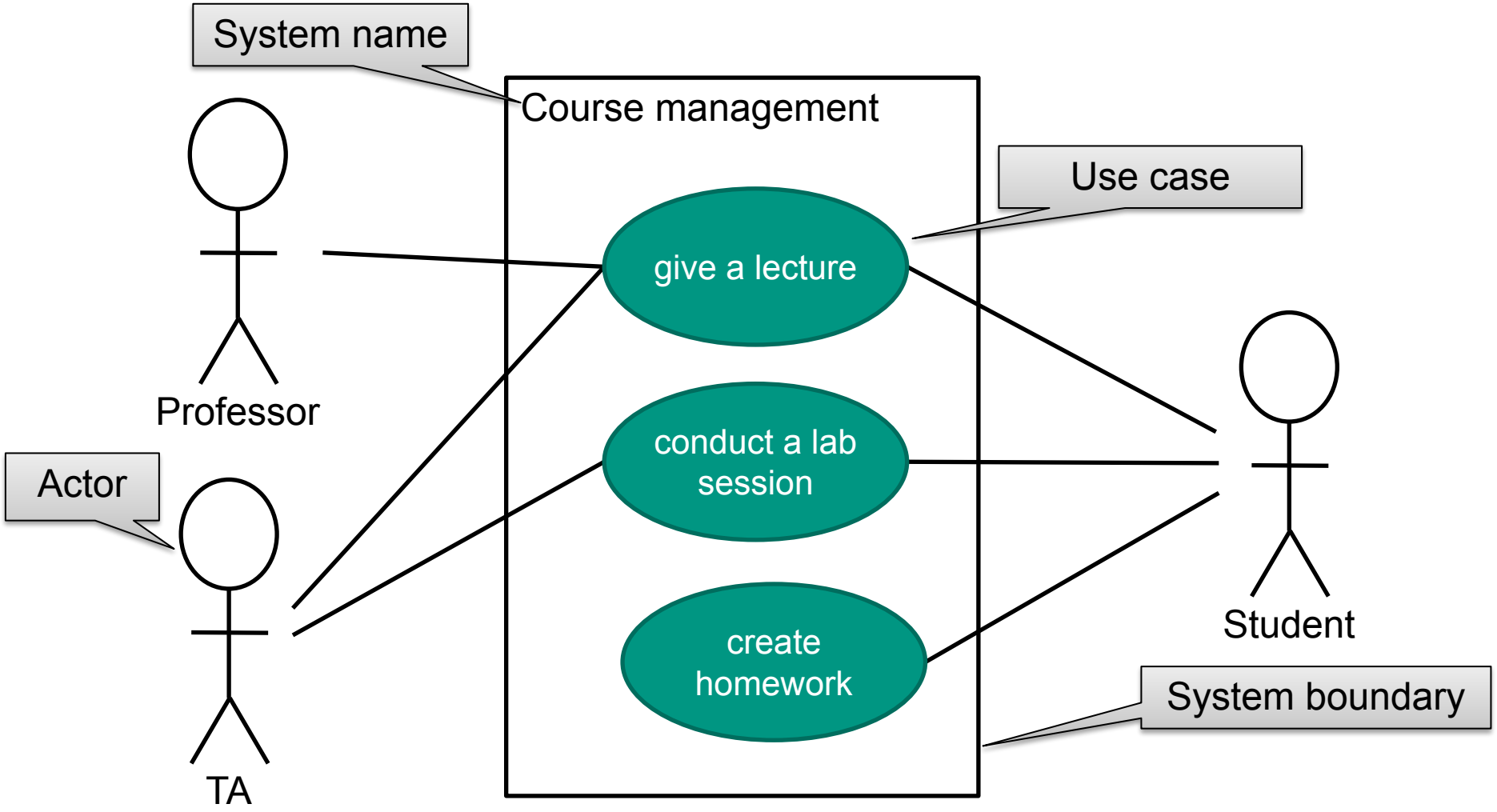


We focus only on **UML diagrams**!

Use case diagrams (UML) for simple design (system modeling)

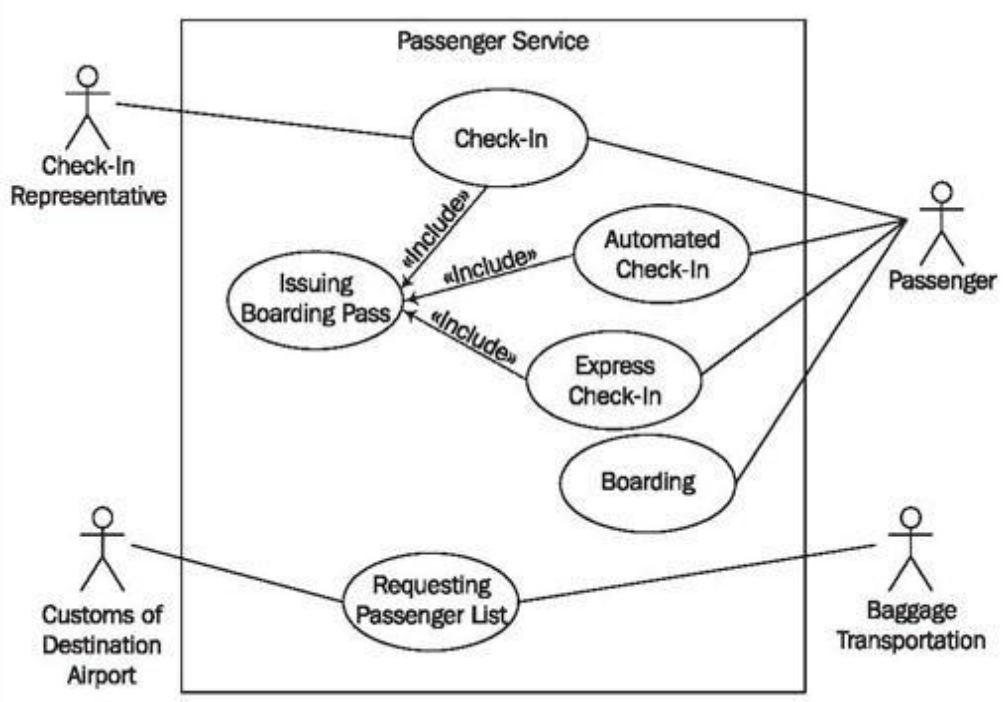
- Use case diagrams are used during the requirement engineering to represent the **externally visible behavior** of the system.
- An **actor** specifies a **role of a user or other system** that interacts with the system we are analyzing.
- A **use case** represents a **class of functions** offered by the system.
- A **use case model** is the set of all use cases that describe the **entire functionality** of the system.
- A use case diagram includes:
  - Actors, use cases, associations, system boundary

Use case diagram – Example: Course management



Use case diagram – Example: Airport Passenger Service

- Airport Checking Scenario [Robertson, s., Mastering the Requirements Process]
- **Include** relationship represents functionality that is used by more than one use case



Use case text – Describing use case in text format

- Use case name
- Main **scenario**
  - **Steps**
- Extensions
  - Extension condition; steps

- Specify **what** to do, **not how** to do!
- Do not specify user interface
- Optional: priority, trigger, pre-condition, post-condition (guarantees), sub-use case

Use case text – Example

- **Name**:
  - Create homework
- **Participating actor:**
  - College student
- **Input condition:**
  - Student receives exercise sheet
  - Student is healthy
- **Output condition:**
  - Student makes solution
- **Flow of events**:
  - Student brings current exercise sheet
  - Student reads through the tasks
  - Student solves the task and enters it into the computer
  - Student prints the solution
  - Student submit the solution
- **Special requirements:**
  - No

Activity diagrams

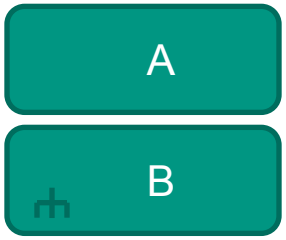
- An activity – multiple actions
  - Can be used **to describe a use case**
  - Can represent parallel relationship
- An activity diagram describes a procedure
  - Operational or business processes
  - Technical processes of workflows and use cases
  - Concrete algorithmic processes in programs
- Activity diagrams consist of
  - Action, object nodes and control nodes, as well
  - Object flows and control flows.

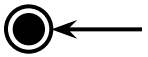
Activity diagram – Main components

- Main components
  - Start
  - Actions
  - Fork/Join
  - Decision/Merge
  - Flow
  - Final

Activity diagram symbols and elements (1)

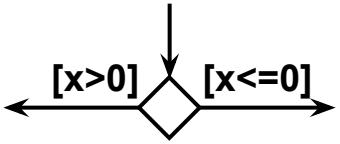
- Actions
  - Elementary action
  - Nested action
- Nodes
  - Starting node
    - Starting point of a process
  - End nodes
    - Ends all actions and control flows
  - Flow final
    - Ends a single object flow and control flow



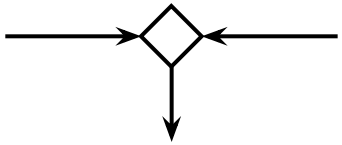


Activity diagram symbols and elements (2)

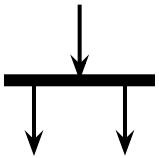
- Decision
  - Conditional branching



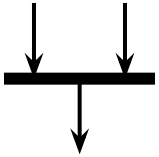
- Merging
  - "or" connecting



- Forking
  - Dividing a control flow

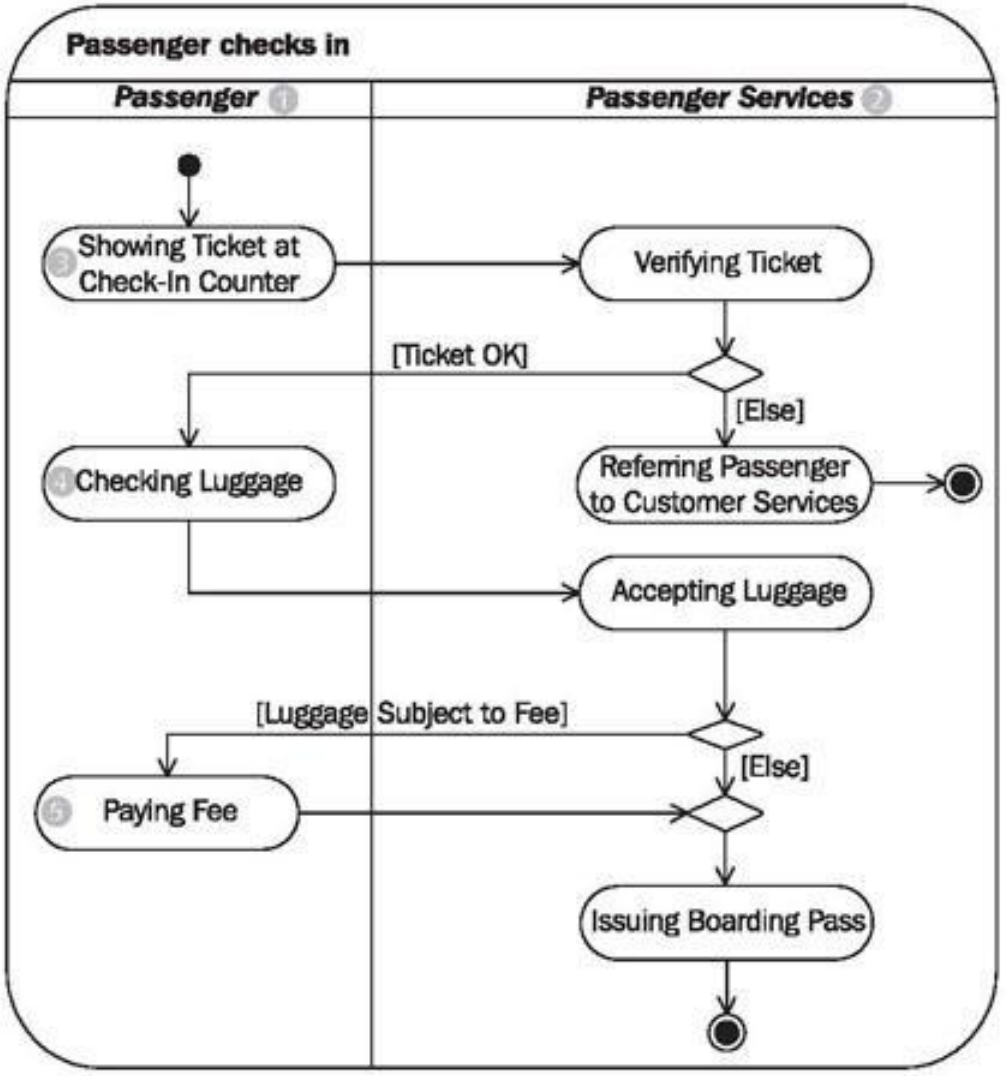


- Synchronization
  - ”and" joining



Activity diagram – Example: Airport Passenger Service

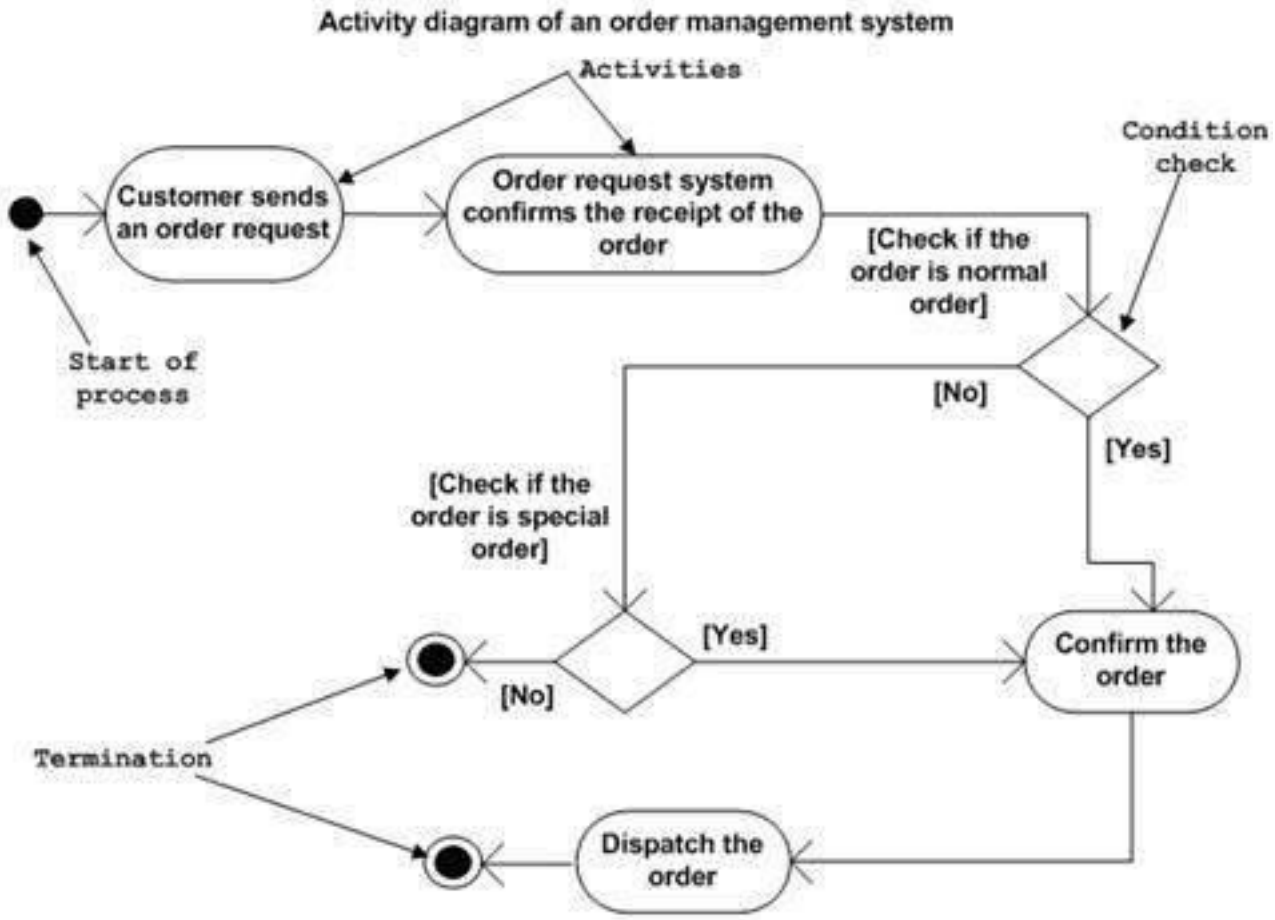
- Airport Checking Scenario [Robertson, s., Mastering the Requirements Process]
- Example with partitions:



Activity diagram – Example: Order management

- An activity diagram for order processing

Source: https://www.tutorialspoint.com/uml/

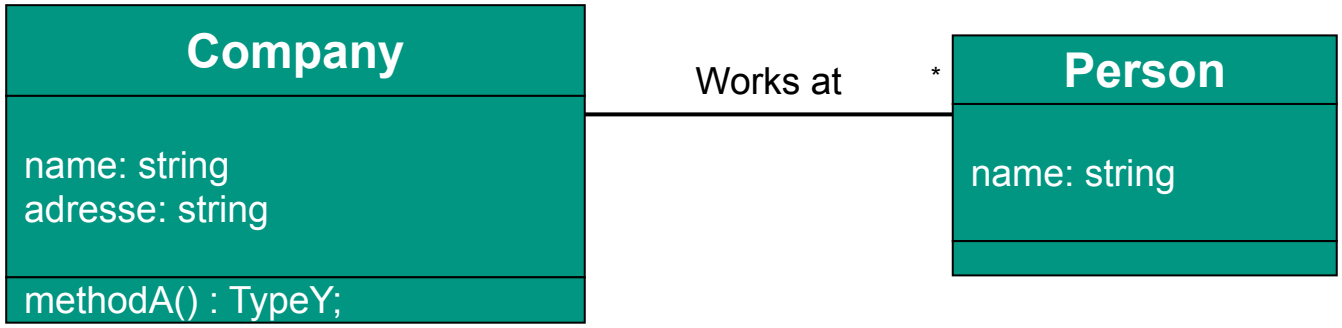


Class diagram (UML) – Designing your system

- Describes the types of objects in the system
- Describes the **static relationships** among them
- Basic components of class diagrams
  - Class name
  - Class properties
    - Attributes
    - Associations (could be bi-directional)
  - Class operations
    - Visibility name (parameter list): return-type {property-string}

Class diagram (UML) – Example

- Describes the types of objects in the system
- Describes the static relationships among them



Test Driven-Development

OTHER XP KEY PRACTICES

12 key practices of XP process

- **Planning game**
- **Small releases** (every 2 weeks)
- Metaphor
- **Simple design**
- **Testing (customer tests & Test-Driven Development – TDD)**
- Refactoring
- Pair programming
- Collective code ownership
- Continuous integration
- 40-hour week
- On-site customer
- Coding standards

Refactoring

- What is the problem of KIS?
  - Simplicity vs. generality
- Solution is Refactoring
  - Teams design and revise design through refactoring in the course of the project
- Continuous design improvement process by ‘refactoring’:
  - **Removal of duplication**
  - **Increase cohesion**
  - **Reduce coupling**
- Refactoring is supported by comprehensive testing – customer tests and **programmer tests**

Continuous integration

- Teams keep the system fully integrated at all times
- Daily, or multiple times a day builds
- Avoid '**integration hell**'
- Avoid code freezes
- **Integrate your function only if all unit tests run 100%!**

Implementation – Pair programming

- Code is built by two programmers, sitting side by side, at the same machine (pilot/co-pilot)
- All production code is therefore reviewed by at least one other programmer
- Research shows that pair programming produces better code in the same time as programmers working singly
- Pairing also communicates knowledge throughout the team

(if pair programming is not possible, ask your teammate to review your code!)

Collective code ownership

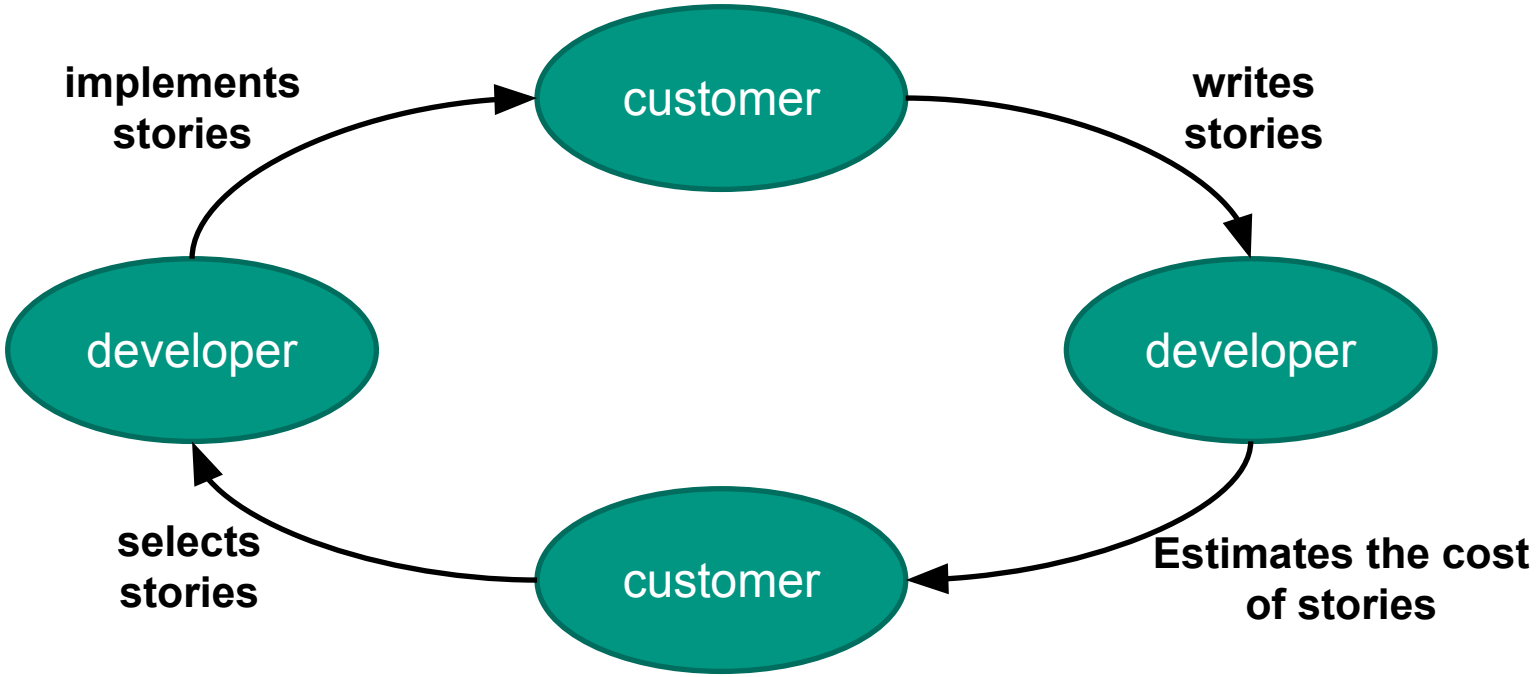
- Any pair of programmers can improve any code at any time
- No secure workspaces
- All code gets the benefit of many people’s attention
- Avoid duplication
- Programmer tests catch mistakes
- Pair with expert when working on unfamiliar code

Whole Team – Customer on-site

- All contributors to an XP/TDD project are one team
- Must include a business representative – the '**customer on-site**'
  - Provides requirements
  - Sets priorities
  - Steers project
- Team members are **programmers**, **testers**, **analysts**, coach, manager
  - Coach: supports the team in adhering to practices
  - manager: tracks the estimated and actual development effort
- Best XP teams have **no specialists!**

Our project team members are **programmers**, **testers**, **analysts**!

Interaction between customer and developer



If no customer can be on-site

- Determine local representative
- Customer comes to the planning meeting
- Customer is visited
- More frequent releases

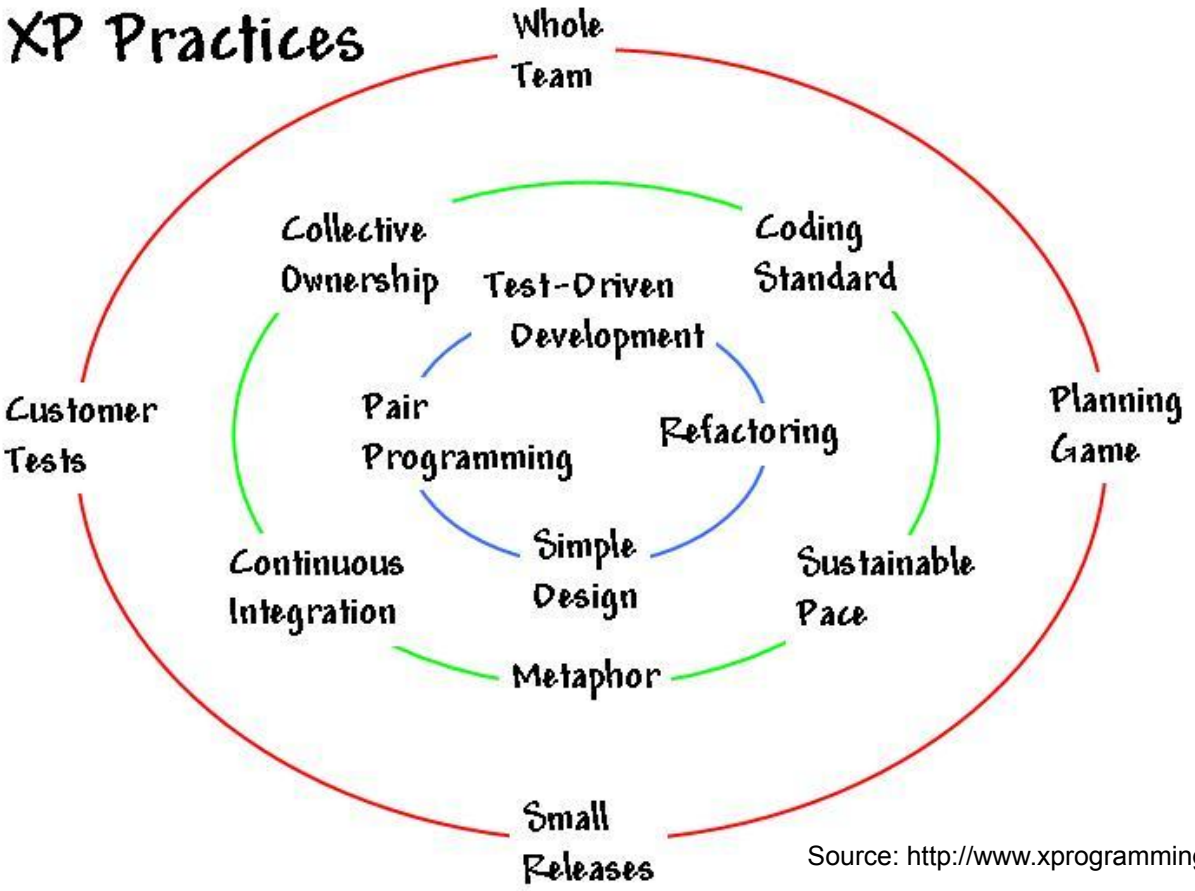
(for our group projects: a TA or a team member can serve as a customer)

Customer tests (acceptance tests)

- The customer defines one or more automated acceptance tests for a feature
- Team builds these tests to verify that a feature is implemented correctly
- Once the test runs, the team ensures that it keeps running correctly thereafter
- System always improves, never backslides

12 key practices

- Our focus on TDD/XP **development** practices:
  - **TDD**
  - **Simple Design**
  - **Small releases**
  - **Planning game**
  - Refactoring
  - Continuous integration



Source: http://www.xprogramming.com

The rules of XP

- XP/TDD describes 5 basic rules that are performed within the software development process:
  - **Planning**
  - Managing
  - **Designing**
  - **Coding**
  - **Testing**

XP rule – Planning

- User stories are written
- Release planning creates the release schedule
- Make frequent small releases
- The project is divided into iterations
- Iteration planning starts each iteration

XP rule – Managing

- Give the team a dedicated open workspace
- Set a sustainable pace
- A standup meeting starts each day
- The project velocity is measured
- Move people around
- Fix XP when it breaks

(**Not applicable for our group project**)

XP rule – Designing

- **Simplicity**
- Choose a system metaphor
  - Teams develop a common vision of the system
  - Everyone understands how the system works
- Use CRC cards or **UML diagrams** for design sessions
- Create **spike** solutions to reduce risk
  - **Spike**: A very simple program (experiment) to explore **potential solutions** and figure out answers to tough technical or design problems
- No functionality is added early
- Refactor whenever and wherever possible

XP rule – Coding

- The customer is always available
- Code must be written to agreed standards
- **Code the unit test first, and then production code**
- All production code is pair programmed (revised by another team member)
- Only one pair integrates code at a time
- Integrate often
- Set up a dedicated integration computer
- Use collective ownership

XP rule – Testing

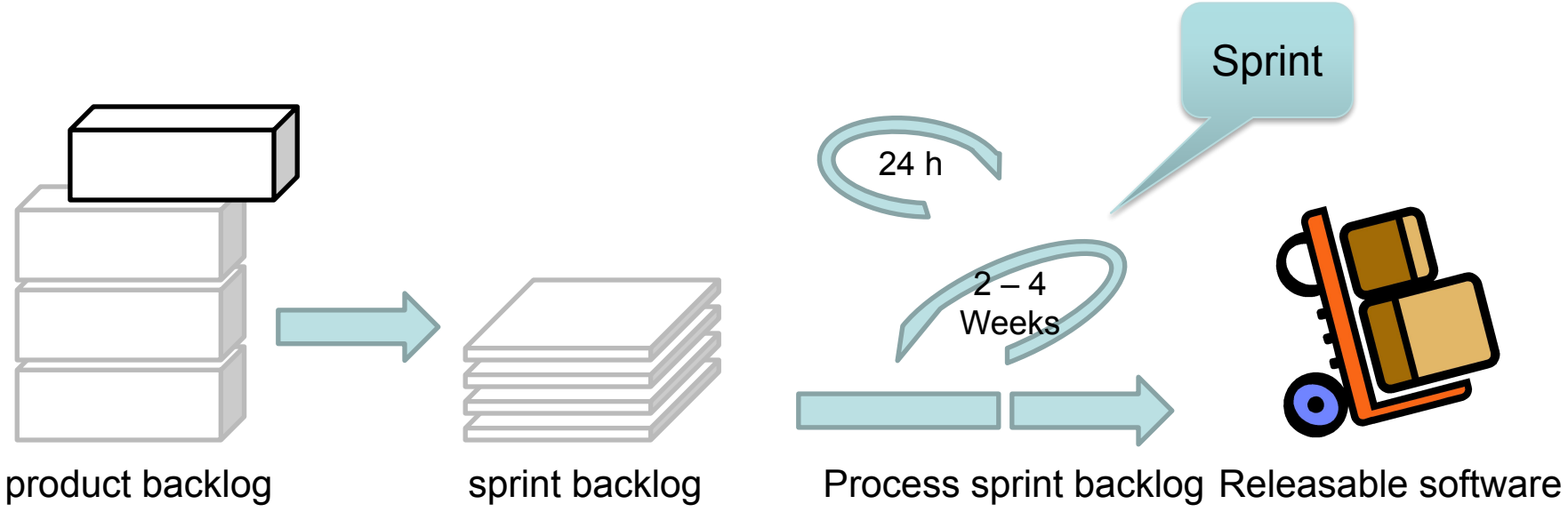
- All code must have unit tests
- All code must pass all unit tests before it can be released
- When a bug is found tests are created
- Acceptance tests are run often and the score is published

Typical XP artifacts

- **Story cards**
- Task cards
- Delivery plan
- Iteration plan
- **Code and tests**
- Acceptance tests

Scrum – Project management

- Scrum is a process model for agile project management (for this course we focus on agile **development practices – XP/TDD practices**)



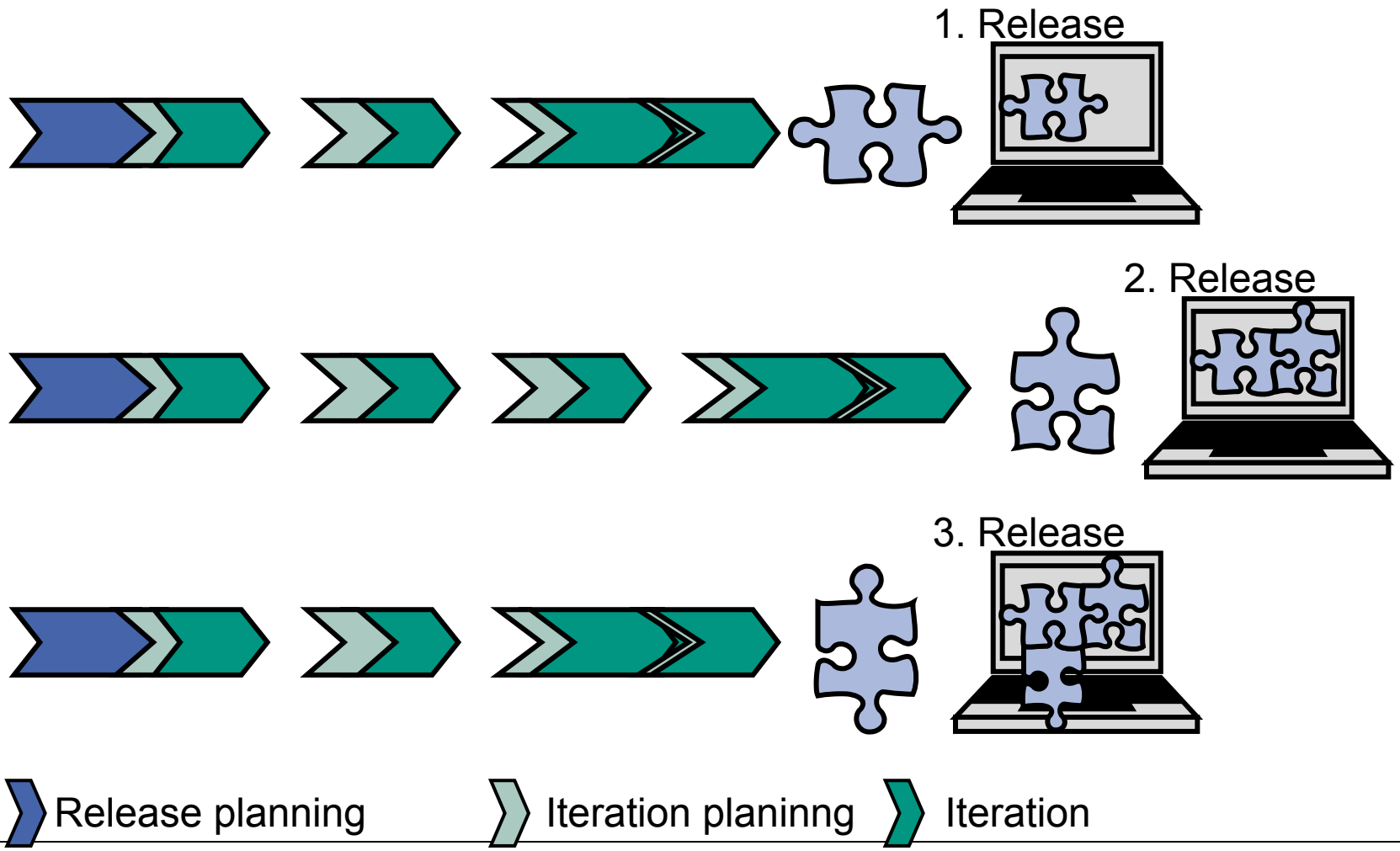
Did agile process solve the problems?

- Process adaptation! (as we did it for our group projects!)
- Customer involvement: difficult to find a customer who can become part of the XP/TDD team
- Architectural design: the incremental style of development can cause inappropriate architectural decisions at an early stage of the process
  - Not clear until many features have been implemented and refactoring could make the architecture very expensive
- Test complacency: easy to believe the system is properly tested (since it has many tests)
  - Because of the automated testing, there is a tendency to develop tests that are easy to automate rather than tests that are ‘good’ tests

Example

XP / TDD MINI PROJECT

Planning game – Overview



Planning game – Step 1: Writes stories

- Customer [a team member] writes stories on cards
- Developers estimate stories in points/scores (in the beginning: ideal effort)
- Developers determine speed in points/scores per planning period (X points/scores)
- Customer [team] selects stories for X points/scores (based on his/her priorities)

Planning game – Step 2: Estimate stories

- First iteration: estimate time
  - E.g. couple days
- More iterations: estimate difficulty
  - Compare with finished stories
  - Abstract unit of measurement (e.g. dots, points, ...)
- No estimate possible:
  - Sharing the story by the customer
  - Experiments ("spikes") as stories

Spikes

- "Never made anything comparable”
- Experiment
  - Once through the whole problem
  - Do not aim for a perfect solution
  - Stop as soon as possible

Planning game – Step 3: Determine the speed

- Speed = points/scores of all finished stories per iteration
- Speed changes
  - growing experience of the team increase the speed
  - Support of the finished subsystems decrease the speed

Planning game – Step 4: Selection of the stories

- Customer [team] selects stories
- Usual selection criteria:
  - Estimated **effort**
  - Assumed business value (how **important** is)
  - **Speed** of the team

A task is causing problems…

- Discuss problem in the team
  - New solutions
  - Change the order of tasks
  - Group couples again
- Inform customers
  - Delete stories from the iteration
  - Simplify stories

XP/TDD mini project: “Counter application”

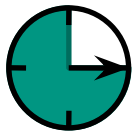
- Development of a small application in iterations of 15 minutes
- Procedure per iteration:
  - Estimation and selection of stories
  - Detailed iteration planning
  - Implementation (TDD)
  - Integration
  - Acceptance test (manual)
  - Determining the speed

XP/TDD mini project: Iteration 1

- Available time: 2 pairs x 15 minutes (= 6 points)
- Estimated effort:
  - Estimation in full 5 minutes
  - Initial definition of points: 5 minutes = 1 point

Story 1

- The counter reading is displayed (initial value 0) and can be increased by 1 with the command '+'.

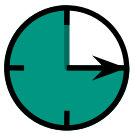
- estimated: =  

15 min 3 Points

Story 2

- The counter reading can be set directly to any positive value with a new command '='.

- estimated: =

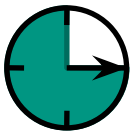


5 min

1 Point

Story 3

- An upper limit can be set for the counter reading.

- estimated: =  

10 min 2 Points

XP/TDD mini project: Iteration 1

- Available time: 2 x 15 minutes (= 6 points)
- Estimated effort:
  - Story 1: 15 min = 3 points
  - Story 2: 5 min = 1 point
  - Story 3: 10 min = 2 points
- All stories can be done!

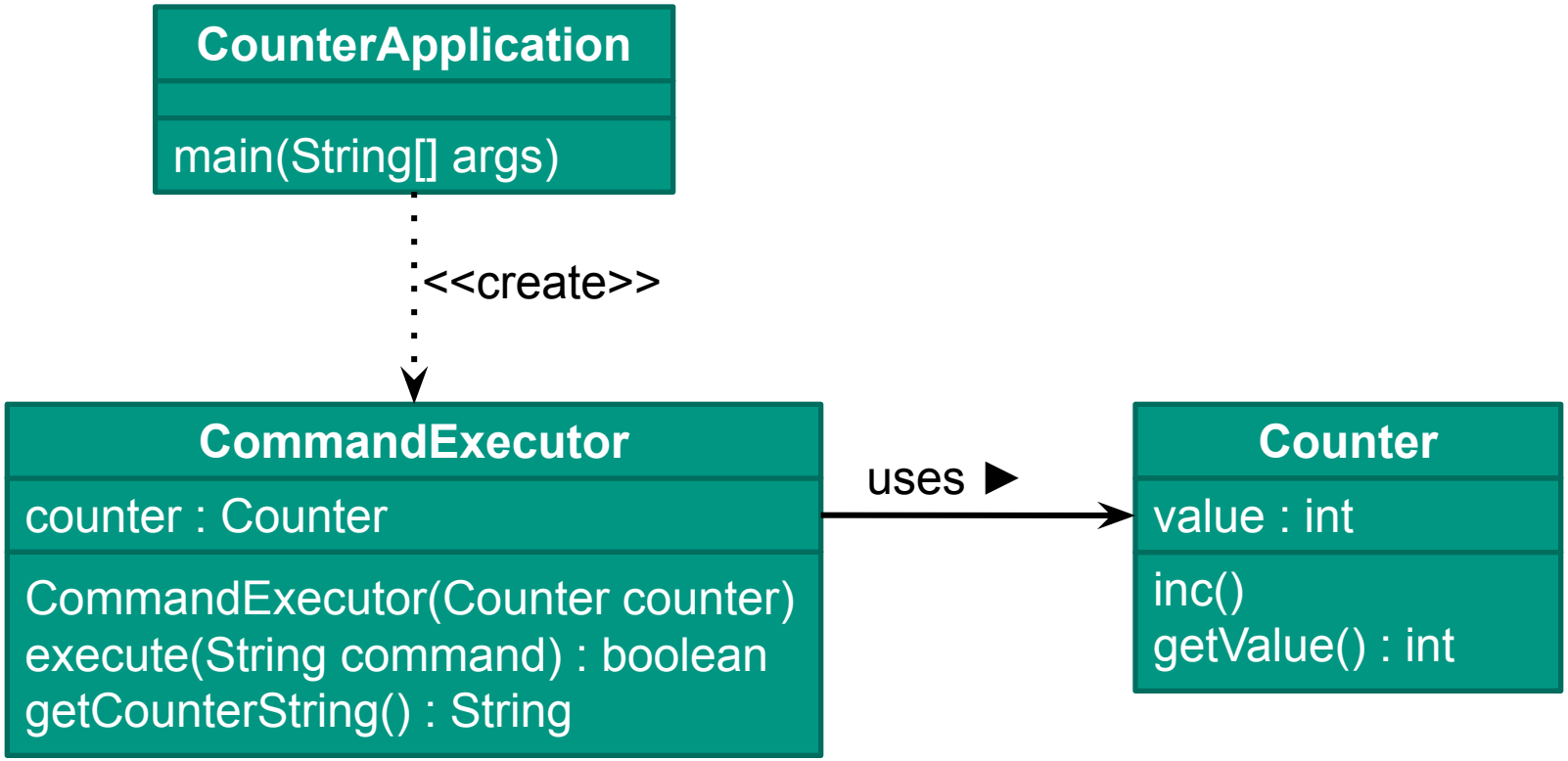
XP/TDD mini project: Iteration planning

- Break down stories into tasks
- Story 1:
  - Task 1: CommandExecutor
  - Task 2: Class Counter
  - Task 3: Linking Executor and Counter
  - Task 4: CommandApplication.main (..)
- Story 2:
  - ...

XP/TDD mini project: Implementation

- 2 pairs work on the tasks for Story 1
  - Pair 1 implements Task 1 in 10 min
  - Pair 2 implements Task 2 in 5 min
  - Pair 2 implements Task 4 in 5 min
  - Both pairs implement task 3 in negligible more than 5 min
- **Story 2 and 3 can not be fully implemented**

Implementation Story 1



XP/TDD mini project: Acceptance tests (AT)

- Small language for AT specification:
  - exec (x): execute command x
  - check (value): Check the counter value displayed
- Acceptance Test for Story 1:
  - check (0), exec (+), check (1), exec (+), exec (+), exec (+), check (4)

XP mini project: Evaluation Iteration 1

- Acceptance test for Story 1 successful
  - Actual completed stories: one
  - Sum of points of completed stories: 3 points
- Estimated speed: 6 points
- Actual speed: 3 points
- Accepted speed for iteration 2: **3 points**

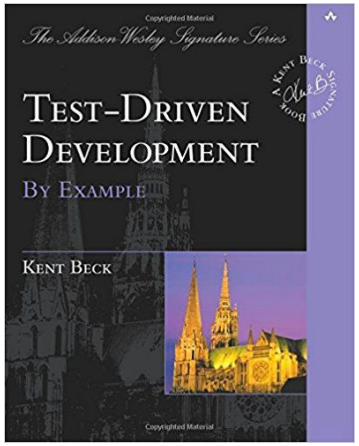
 iteration 2: 3 points for next Story Cards…

Summary

- Introduction to software process
  - Waterfall
  - Drawbacks of waterfall
- Agile process: Test-driven development (TDD)
  - XP practices and rules
  - Challenges, etc.
  - Test-driven development (TDD)
  - Jest: JavaScript testing framework
- XP /TDD mini project example

Literature – TDD

- Beck, K.: Test Driven Development: By Example, 1st Edition.
- https://www.guru99.com/test-driven-development.html
- https://www.tutorialspoint.com/software_testing_dictionary/test_driven_development.htm
- https://github.com/dwyl/learn-tdd
- JavaScript Testing Framework (Jest): https://jestjs.io
- Beck, K.: Extreme Programming explained, Addison-Wesley 1999
- Fowler, M.: Refactoring: Improving the Design of Existing Code, Addison-Wesley 1999
- Martin, Robert C. Clean code: a handbook of agile software craftsmanship. Pearson Education, 2009
- Martin, Robert C. Clean architecture: a craftsman's guide to software structure and design. Prentice Hall Press, 2017



Back-up

Outline

- Introduction to software process
  - Aspects of software engineering
  - Waterfall model, Incremental model, Evolutionary
  - Agile process
- XP process model – Test-driven development (TDD)
  - XP practices
  - Test-driven development (TDD)
  - XP mini project (Example)

Brief history of software engineering

- The pioneering era
  - No Software Engineering: no way to estimate software development time
- Starting 1960s
  - The Software Crisis 1965-1985
    - Therac-25 (radiation treatments to cancer patients): 1985-1987
    - Morris worm 1988
- 1985 – 2000
  - No silver bullet: OO, design patterns, formal methods, process
- 2000 – present
  - Agile software development process, Model-driven design, tools, Program synthesis, …

What are the aspects of software engineering?

- Software engineering is the **technological** and **organizational** discipline for the **systematic development** and **maintenance** of software systems that fulfill specified **functional** and **non-functional** attributes
- Organizational aspects
- Aspects of software production
  - Gathering requirements
  - Design
  - Development
  - Testing & debugging
  - Maintenance

Organizational aspects

- **Planning**: how, where, when is what to do by whom; what does it cost?
- **Staffing**: finding qualified personnel
- **Work organization**: rules for cooperation (processes, tasks, responsibilities, relations, obligation to report)
- **Leadership**: motivation and communication of goals
- **Supervision**: Check if work progresses according to plan, requirement are met?

Aspects of software engineering

- **Systematic**
  - It is not tried or advised (trial and error)
  - Methods and tools are used purposefully and according to the state-of-the-art
- **Development and maintenance** of software systems
  - For having a successful software, maintenance is required after the development (corrections, adjustments, extensions).

Other aspects of software engineering

- **Functional** requirements (attributes) specify the functions of the software
- **Non-functional** requirements (attributes) – aka quality requirements – specify how well the software performs its functions
  - Such as reliable, how fast, how user-friendly, how secure
  - But also internal qualities such as changeability, degree of documentation,…

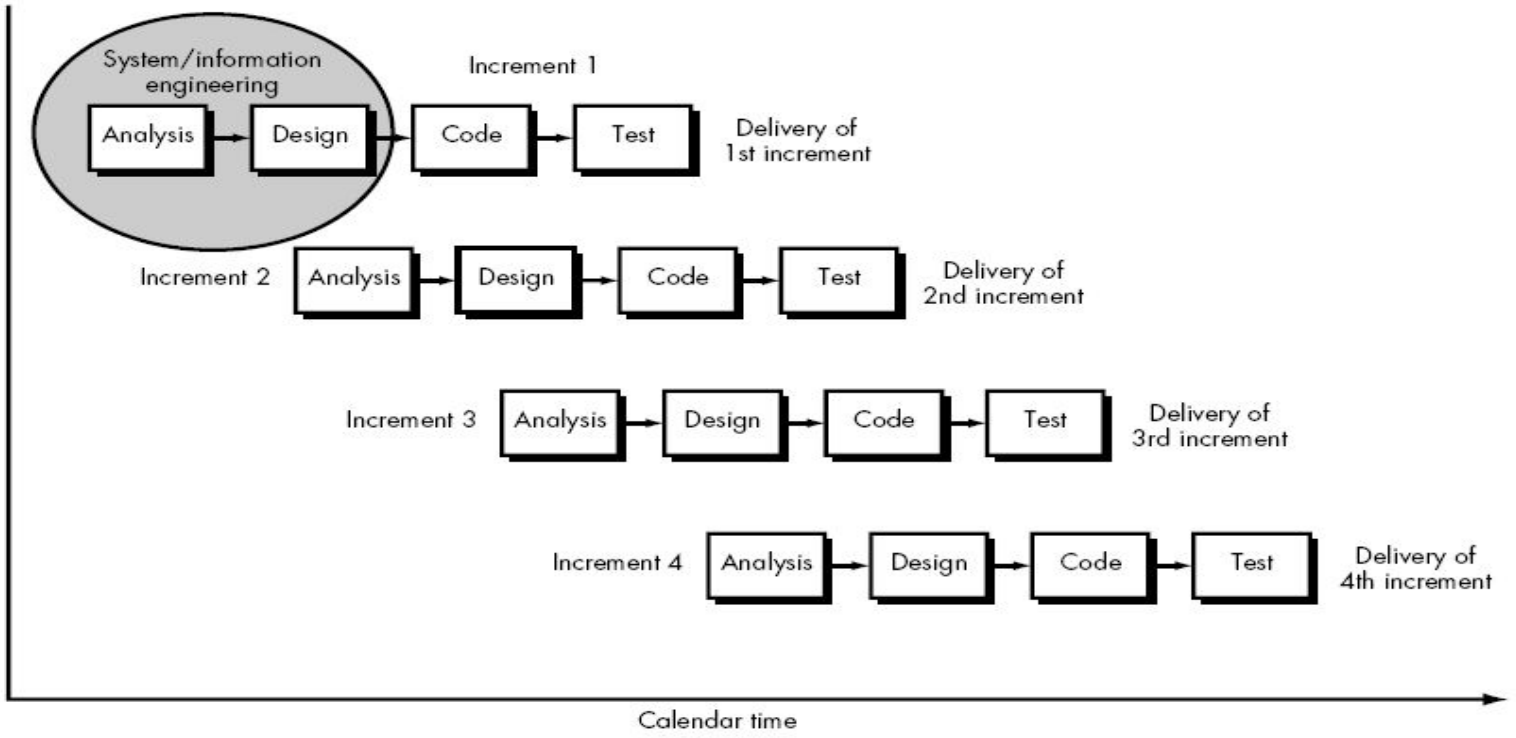
Other aspects of software engineering

- A social contract to software engineering: produce software
  - sufficient quantity
  - of satisfactory quality
  - on schedule
  - at competitive costs and within the budget

- Not necessary: in vain, perfection, immediately

Improving waterfall – Incremental model

- **Incremental process model:** “multi-waterfall” cycle
- Building different prototypes of different features, then merging them all into one over-arching design concept



Incremental model

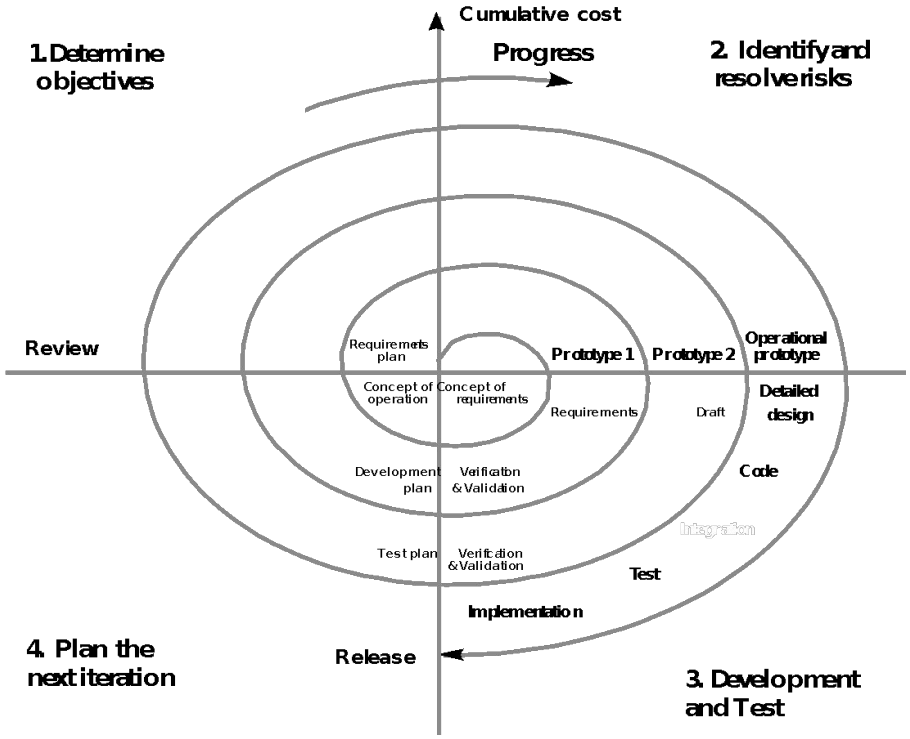
- Produce core products first: initially the software is released with the **basic requirement**
- Further refinements are produced in follow-up releases
- Deliver faster
- More flexible – less costly to change scope and requirements
- Higher cost than waterfall – needs a clear and complete definition of the whole system before it can be broken down and built incrementally
- Example:
  - Text editor: in the first increment the basic text edit e.g. notepad is delivered. Then in the second increment the auto spell check and the grammar feature is delivered

Handling changes better – Evolutionary process

- Requirements that are **well understood and defined** are used to build an initial functional prototype
- Gradually, more and more requirements come into focus and they are built around the original prototype
  - Planning
  - Risk analysis (design)
  - Engineering
  - Evaluation

Evolutionary process

- Spiral model
- Handle changes better
- High amount of risk analysis
- Good for large and mission-critical projects



Simple design (2)

- Output
  - CRC Card (Class-Responsibility-Collaboration)
    - Considers class structure and its behavior together
    - CRC is a collection of standard index cards used for design sessions
    - It is divided into three sections:
      - class name, responsibilities and collaborators
  - **UML diagrams**

Class Name	
Responsibilities	Collaborators

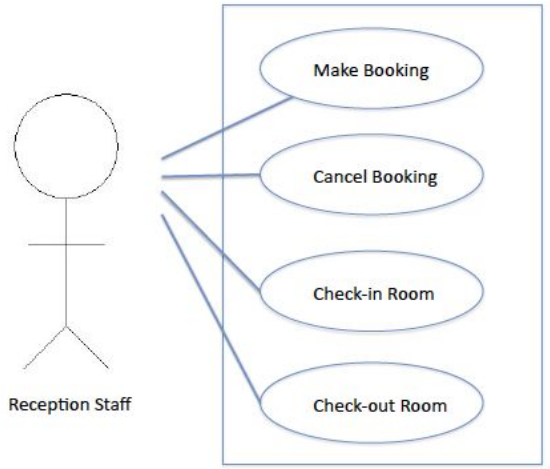
Example: Student CRC card

Student	
Student number Name Address Phone number Enroll in a seminar Drop a seminar Request transcripts	Seminar

Simple design (2)

- Output
  - CRC (Class-Responsibility-Collaboration – **not covered in this class**)
  - **UML diagrams**: Divides structural and behavioral modelling from each other
  - We can model using both UML use-case/class/component diagrams and CRC
    - Go to CRC and go on to UML diagrams
    - Such translations will surely filter much of your misunderstandings and errors.

Example: Use Case diagram – Reception Scenario



We focus only on **UML diagrams**!

Jest Unit Test Example

File: sum.js

```
function sum(a, b) {
      return a + b;
}
module.exports = sum;
```

Function to add two numbers

File: sum.test.js

```
const sum = require('./sum');

test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3);
});
test('adds 0 + 1 to equal 1', () => {
  expect(sum(0, 1)).toBe(0);
});
```

test indicates Unit test to test the sum function. Uses **expect** and **toBe** to test that two values were exactly identical*

Jest configuration file generated using **npm init -y**

File: package.json

```
{
  "scripts": {
    "test": "jest"
  }
}
```

* Use **test.only** instead of **test** to only execute one test case

Mock Functions

- Test code dependencies by erasing implementation
- Mock function calls, implementation, modules, return values
- Mock in testcase or manual mock
- All mock functions **.mock** property: data about how function called, return value, *this* value
- **Demo:** mock.test.js

https://jestjs.io/docs/en/mock-functions
http://xunitpatterns.com/Mocks,%20Fakes,%20Stubs%20and%20Dummies.html

Testing Asynchronous Code

- Testing asyn code eg: callbacks

https://jestjs.io/docs/en/asynchronous
https://jestjs.io/docs/en/tutorial-async

Snapshot Testing

- Make sure UI does not change unexpectedly
- Take snapshot of UI and compare with one on file

```
//compare DOM tree to match snapshot file created last time
import React from 'react';
import Link from '../Link.react';
import renderer from 'react-test-renderer';

it('renders correctly', () => {
  const tree = renderer
    .create(<Link page="http://www.facebook.com">Facebook</Link>)
    .toJSON();
  expect(tree).toMatchSnapshot();
});
```

https://jestjs.io/docs/en/snapshot-testing
https://github.com/facebook/jest/blob/master/examples/snapshot/__tests__/__snapshots__/link.react.test.js.snap
https://benmccormick.org/2016/09/19/testing-with-jest-snapshots-first-impressions/