

# Software Engineering I (02161)

Week 1

Assoc. Prof. Hubert Baumeister

Informatics and Mathematical Modelling  
Technical University of Denmark

Spring 2010

# Who are we?

- 110 students with different backgrounds
  - Bachelor Softwaretek.: 66
  - Bachelor It og Kom.: 39
  - Other: 5
- Teacher
  - Hubert Baumeister, Assoc. Prof. at DTU Informatik (hub@imm.dtu.dk; office 322.010)
  - 3 Teachingassistants

# Course activities

- Lectures every Monday 13:00 — approx 14:45 (Lecture plan is on the course Web page)
- Exercises in the E-databar (341.003+015)
  - Teaching assistants will be present : 15:00 — 17:00
  - Expected work at home: **5 hours** (lecture preparation; exercises, ...)
  - No hand-in of the solutions, but show the solutions to the TA or me for feedback
- Examination
  - Last 5 weeks: exam project in groups (2—4)
    - Software + **Report**
  - **no** written examination

# Course material

- Course Web page:

<http://www.imm.dtu.dk/courses/02161> contains

- practical information: (e.g. lecture plan)
- Course material (e.g. slides, exercises, notes)
- Check the course Web page regularly

- CampusNet: Is being used to send messages;

- make sure that you receive all messages from CampusNet

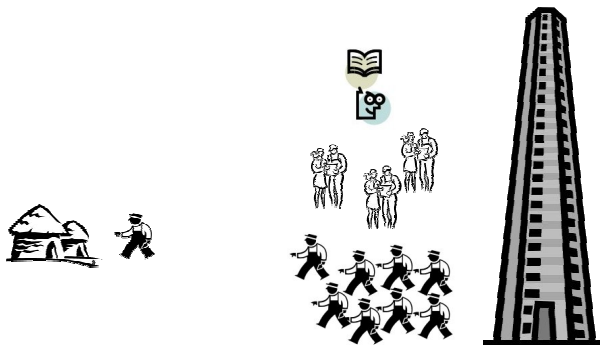
- Books:

- Textbook: Software Engineering 9 from Ian Sommerville available at Polyteknisk Boghandel
- Supplementary literature on the course Web page

- Course Language

- The course language is Danish; slides, notes, and other material mostly in English
- If everybody agrees to that, it can be given in English

# Software Development Problem: Building large software



- You learn **techniques** for building **skyscrapers**
- But the **projects** you are doing are still only **little huts**

# What is software?

Software is everywhere: PC, phones, electronic devices (e.g. TV, washing machine, ...), computing centre, Web server, ...

## Software

Not only the computer program(s) but also

- Documentation (User–, System–)
  - Configuration files, ...
- 
- Types of software
    - Mass production (like Windows, Linux, OpenOffice, SharePoint, SAP ...): The maker of the software owns the system specification
    - Customised software: The customer owns the system specification
    - Mixture: Customised software based on mass production software (e.g. workflow management systems based on SharePoint, ...)

# Software attributes

- Maintainability
  - Can be evolved through several releases and changes in requirements and usage
- Dependability and security
  - Includes: reliability (robustness), security, and safety
- Efficiency
  - Don't waste system resources such as memory or processor cycles
  - Responsiveness, processing time, memory utilisation
- Acceptability
  - To the user of the system
  - understandable, usable, and compatible with the other systems the user uses

# Software Engineering

## Software Engineering Definition (Sommerville 2010)

Software engineering is an **engineering discipline** that is concerned with **all aspects** of **software production** from the early stages of system specification through to maintaining the system after it has gone into use.

- An **engineer**
  - applies appropriate theories, methods, and tools
  - Works within a set of constraints: producing the required **quality** within the **schedule** and **budget**
    - **make things work**
- **All aspects** of software production: Not only writing the software but also
  - Software project management and creation of tools, methods and theories



# Related disciplines

- Software Engineering uses Computer Science (theories and methods that underlie computers and software systems)
  - e.g. formal languages (e.g. parsing, compilation), computational theory and complexity theory, datastructures, ...
  - Knowledge of Computer Science is essential for a software engineer
- System engineering (theories and methods for the development and evolution of complex systems)
  - e.g. physical systems, business processes, ...
  - Software is becoming more and more and important part of todays system

# Software Engineering diversity

- Types of application (excerpt)

- Stand-alone application (e.g. Word, Excel)
- Interactive transaction-based applications (e.g. flight booking)
- Embedded control systems (e.g., control software the Metro, mobile phones)
- Batch processing systems (e.g. salary payment systems, tax systems)
- Entertainment systems (e.g. Games)
- System for modelling and simulation (e.g. weather forecasts)
- Data collection and analysing software (e.g. physical data collection via sensors, but also data-mining Google searches)
- System of systems (e.g. cloud, system of interacting software systems)

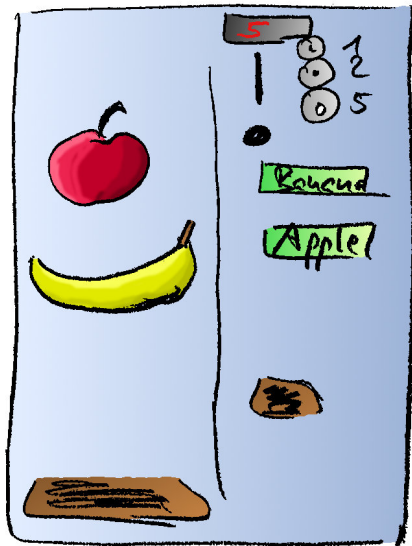
→ Not one tool, method, or theory

- Though there are general principles applicable to all domains

# Basic Activities in Software Development

- **Understand** and **document** what kind of the software the **customer** wants
  - **Requirements Engineering**
- Determine **how** the software is to be built
  - **Design**
- **Build** the software
  - **Implementation**
- **Validate** that the software solves the customers problem
  - **Testing**
  - **Verification**
  - **Evaluation**: e.g. User friendliness
- Each of these have their own methods, tools, and theories

# Example Vending Machine



The task is to implement a control software for a vending machine. The control software has to react on the insertion of coins and selecting a fruit. One should also be able to see how much money the user currently has inserted and how much of a given type of fruit is left. In addition, the owner of the vending machine should be able to see how much money currently is in the machine.

# Vending Machine: Requirements documentation

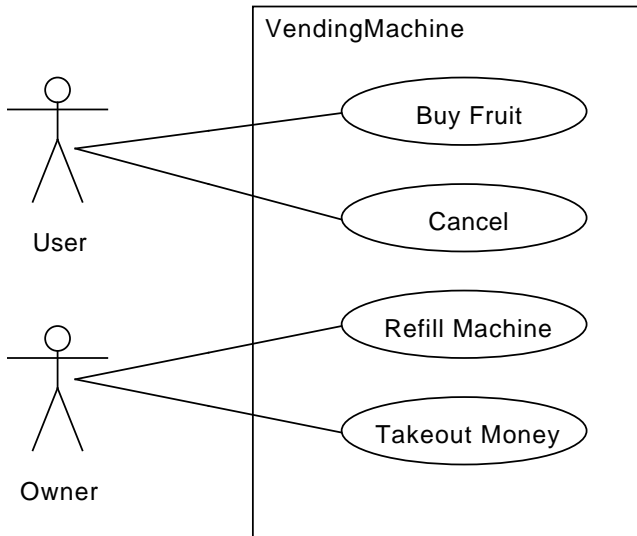
- **Understand** and **document** what kind of the software the **customer** wants
  - Glossary
  - Use case diagram

# Glossary

- Vending machine: The vending machine allows users to **buy fruit**.
- User: The user of the **vending machine** buys fruit by inserting coins into the machine.
- Owner: The owner owns the **vending machine**. He is required to refill the machine and can remove the money from the machine.
- Display: The display shows how much money the **user** has inserted.
- Buy fruit: Buy fruit is the process, by which the user inputs coins into the vending machine and selects a fruit by pressing a button. If enough coins have been provided the selected fruit is dispensed.
- Cancel: The **user** can cancel the process by pressing the button cancel. In this case the coins he has inserted will be returned.

...

# Use case diagram



# Use Case: Buy Fruit

**name:** Buy fruit

**description:** Entering coins and buying a fruit

**actor:** user

**main scenario:**

1. Input coins until the price for the fruit to be selected is reached
2. Select a fruit
3. Vending machine dispenses fruit

**alternative scenarios:**

- a1. User inputs more coins than necessary
- a2. select a fruit
- a3. Vending machine dispenses fruit
- a4. Vending machine returns excessive coins

...



# Vending Machine: Specify success criteria

- Prepare for the validation
  - Create **tests** together with the customer that show when system fulfils the customers requirements

# Functional Test for Buy Fruit Use Case: JUnit Tests

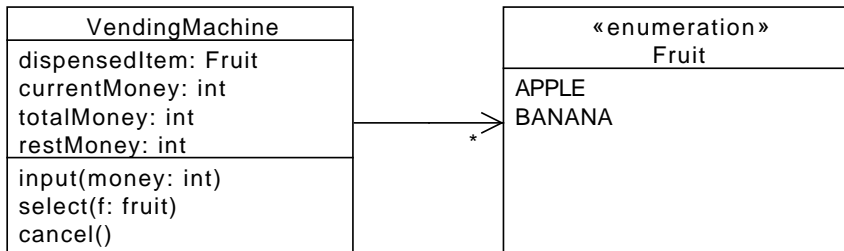
```
@Test
public void testBuyFruitExactMoney() {
    VendingMachine m = new VendingMachine(10, 10);
    m.input(1);
    m.input(2);
    assertEquals(3, m.getCurrentMoney());
    m.selectFruit(Fruit.APPLE);
    assertEquals(Fruit.APPLE, m.getDispensedItem());
}
```

```
@Test
public void testBuyFruitOverpaid() {
    VendingMachine m = new VendingMachine(10, 10);
    m.input(5);
    assertEquals(5, m.getCurrentMoney());
    m.selectFruit(Fruit.APPLE);
    assertEquals(Fruit.APPLE, m.getDispensedItem());
    assertEquals(2, m.getRest());
}
```

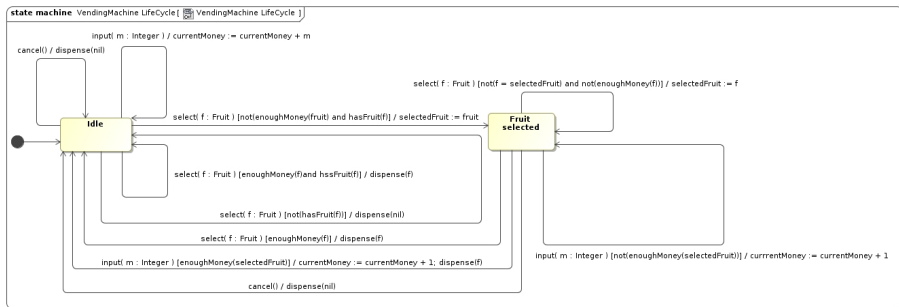
# Vending Machine: Design and implementation

- Determine **how** the software is to be built
  - Class diagrams to show the structure of the system
  - State machines to show how the system behaves
- **Build** the software
  - Implement the state machine using the state design pattern

# High-level Class diagram



# Application logic as state machine

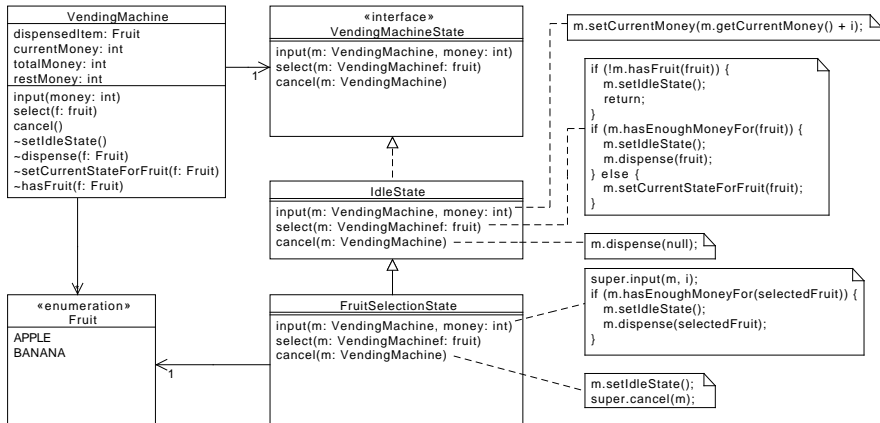


# Alternative representation of the application logic

event	guard	state	state	state
		<i>Idle (I)</i>	<i>Banana selected and not enough money (B)</i>	<i>Apple selected and not enough money (A)</i>
<i>banana</i>	<i>enough money for banana</i>	dispense banana and rest money-> I	dispense banana and rest money-> I	dispense banana and rest money-> I
<i>banana</i>	<i>not enough money for banana</i>	-> B	-> B	-> B
<i>banana</i>	<i>no bananas available</i>	-> I	-> I	-> I
<i>apple</i>	<i>enough money for apple</i>	dispense apple and rest money -> I	dispense apple and rest money -> I	dispense apple and rest money -> I
<i>apple</i>	<i>not enough money for apple</i>	-> A	-> A	-> A
<i>apple</i>	<i>no apples available</i>	-> I	-> I	-> I
<i>money</i>	<i>enough money for banana</i>	add money to current money	dispense banana and rest money-> I	add money to current money
<i>money</i>	<i>enough money for apple</i>	add money to current money	add money to current money	dispense apple and rest money -> I
<i>money</i>	<i>not enough money for neither banana nor apple</i>	add money to current money	add money to current money	add money to current money
<i>cancel</i>		return current money -> I	return current money -> I	return current money -> I

# Design of the system as class diagram

Uses the state design pattern



# Learning objectives of the course and approach to teaching

## • Learning objectives

- To have an overview over the field software engineering and what is required in software engineering besides programming
- To be able to take part in bigger software development projects
- To be able to communicate with other software designers about requirements, architecture, design
- To be able to conduct a **smaller** project from an **informal** and **open description** of the problem

## • Approach to teaching

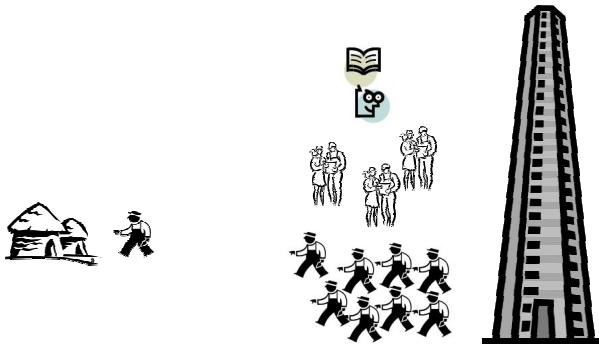
- Providing a general overview of what makes up software engineering
- Teach a concrete method of doing a project (i.e. agile software development with test-driven development)



# Course content

0. Introduction
1. Software Development Process
2. Requirements Engineering
3. Software Testing
4. System Modelling (mainly based on UML)
5. Architecture (e.g layered architecture)
6. Design (among others Design Patterns)

# Software Development Challenges



- Challenges of Software Development

- On **time**
- In **budget**
- No **defects**
- Customer **satisfaction**

# Software Development Process

- Activities in Software Development

- **Understand** and **document** what kind of the software the **customer** wants
  - **Requirements Analysis**
- Determine **how** the software is to be built
  - **Design**
- **Build** the software
  - **Implementation**
- **Validate** that the software solves the customers problem
  - **Testing**
  - **Verification**
  - **Evaluation**: e.g. User friendliness

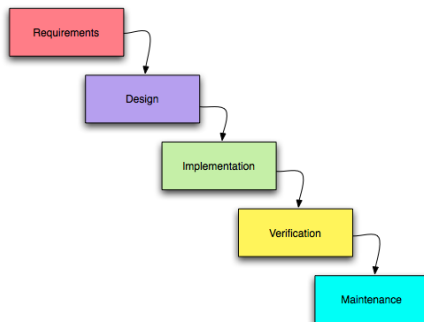
- Each of the steps has its associated set of **techniques**

- However, the techniques can be applied in different orders

- Different **software development processes**

- e.g. **Waterfall** and **Iterative processes** (e.g. Rational Unified Process (RUP), agile processes: Extreme Programming (XP), SCRUM, Feature Driven Development, **Lean** ...)

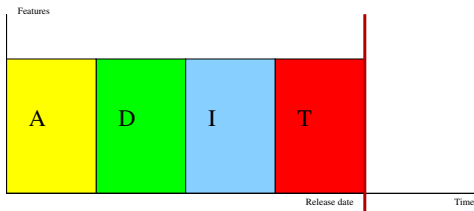
# Waterfall process



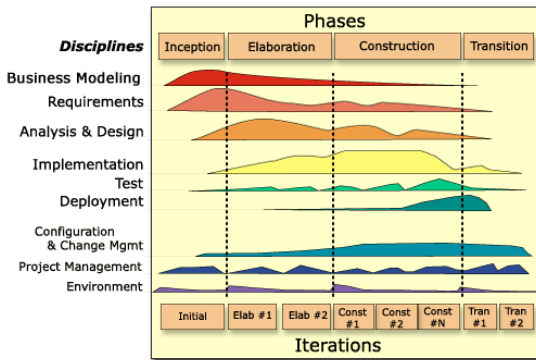
- **Strict** waterfall: An **activity** has to **terminate** (e.g. formally **approved**) **before** the next activity begins
  - No feedback possible from the later activities
  - Takes too long time for the system to be build, which does not allow the customer to give feedback

# Waterfall

- No **feedback** from the next activity possible until the next activity is approved
  - But: during the design of the system new questions come up on how to interpret the requirements; or during the implementation questions come up on how to interpret the design
- formal (costly) change process with approval → **waste** (c.f. Lean software development)
- Iterative development improves on that (depends on the number of iterations)
- **Delay** in one phase of the project delays the **whole** project



# Iterative Processes: E.g. Rational Unified Process



- Inception, Elaboration, Construction, Transition corresponds to Plan the project, understand the problem, build the solution, test the solution, maintain the solution
  - All activities occur throughout the project
  - After each iteration, the customer sees the product and gives feedback

# Agile Processes and Lean Software Development

- eXtreme Programming (XP), Scrum, Feature Driven Development (FDD), **Lean Software Development**
- Iterations getting shorter than with, e.g., RUP
- New set of techniques: Pair programming, customer on site, user stories, test-first and test-driven development, ...
- Focus on **marketable features**
  - **A feature of the software that is relevant for the customer**
  - User stories (XP), Backlog (Scrum), Features (FDD), ...
  - Corresponds **roughly** to use case scenarios
- Lean Software Development
  - Apply **values** and **principles** from **Lean Production** to Software Development

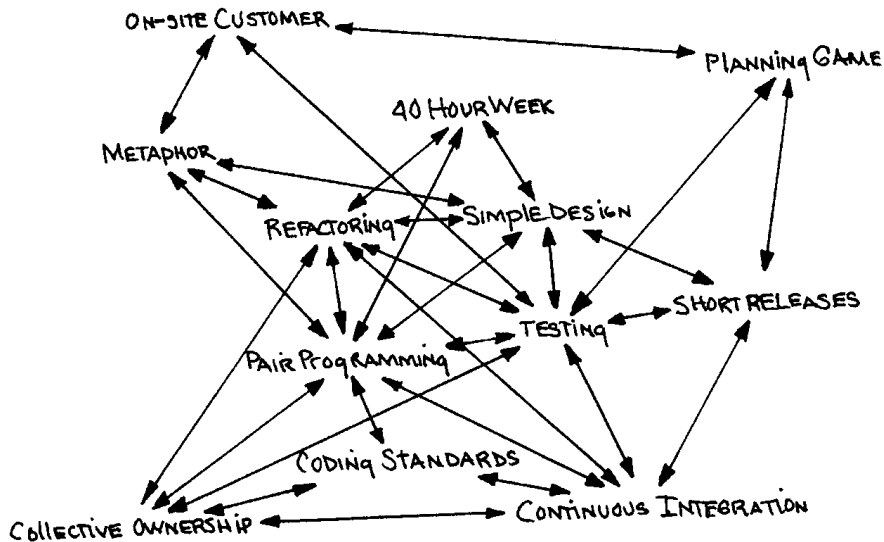


# eXtreme Programming (XP)

- Defined by Kent Beck, Ward Cunningham, and others while working at Chrysler on the C3 project (a payroll system) end of the 90's
  - Original project could not be finished
  - Kent Beck was hired as a consultant and defined eXtreme Programming to finish the project
- Name: take classical software engineering techniques and put them to the extreme
  - Tests are good: test always and make tests before writing the code
    - automatic tests and test-driven development
  - Code review is good: Do the code review while programming → pair programming



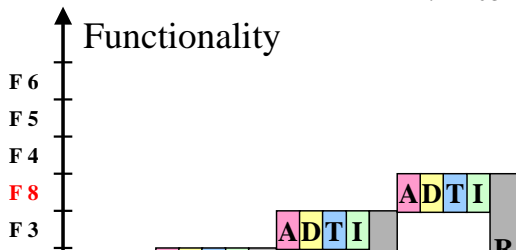
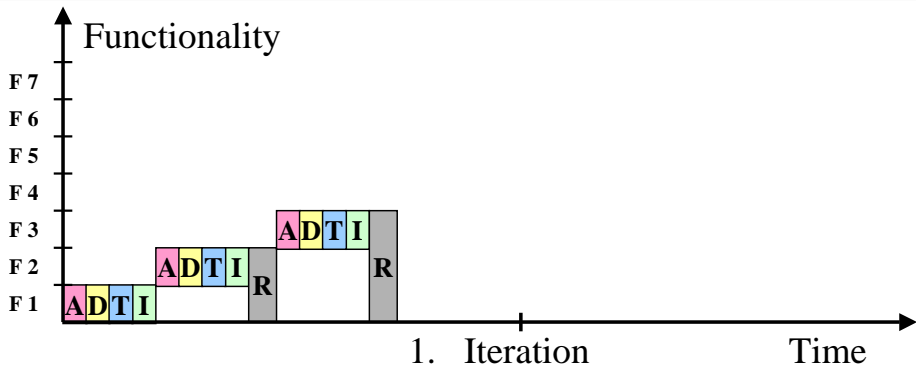
# eXtreme Programming practices



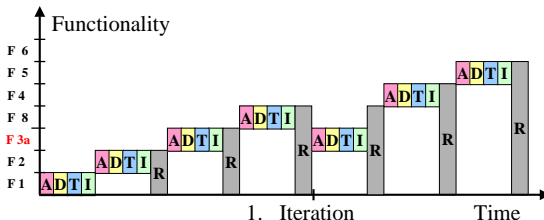
Kent Beck, eXtreme Programming, 1st edition



# eXtreme Programming



# eXtreme Programming



- Iterations are planned by the customer and the developer based on **user stories** (**stories about how the system is to be used**)
  - Customer decides on **priority**; Developer decides on **effort**: **planning game**
  - If the project is delayed, **low priority** features get not implemented
- Important step: **refactoring**: Improve the design of the system without changing its functionality
  - Automated tests** work as regression tests and ensure that existing functionality is not lost

# Lean Software Development

- Apply ideas from Lean production to software development
  - Main principle: Reduce the amount of waste (e.g. time spent that does not produce value for the customer)
  - E.g. for waste:
    - time needed to fix bugs;
    - time needed to change the system because it does not satisfies the customers requirements;
    - waiting time for someone to approve, e.g. the design e.t.c.

# Cycle time

- Solution: Increase feedback by reducing the **cycle time** (time it takes to go one time through the process)

$$\text{cycle\_time} = \frac{\text{number\_of\_features}}{\text{feature\_implementation\_rate}}$$

- E.g. Waterfall (A,D,I,T): 1000 features, 52 weeks;  
feature\_implementation\_rate = 19.23 features/week

$$\text{cycle\_time} = 1000 / 19.23 = 52 \text{ weeks}$$

→ Feedback regarding the **product** and the **process** after 52 week

- Reducing cycle time by reducing the number of features: Without changing the process, the cycle time for one feature becomes

$$\text{cycle\_time} = 1 / 19.23 = 2 \text{ hours}$$

→ Feedback regarding product feature **and process** only 2h

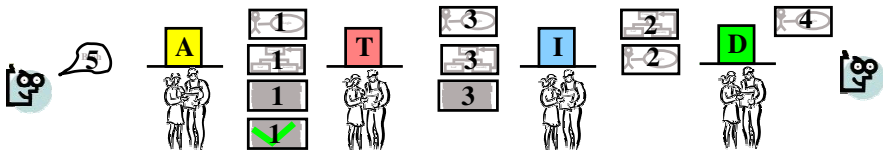
→ Time to finish the product is the same:  $1000 * 2h = 52 \text{ weeks}$ , **but** customer can provide **immediate feedback** on the product and **process improvements** become possible because the same process runs 1000 times instead of just once!!

→ This provides the possibility to **reduce** the 52 weeks time!!

→ **Generate flow**

# Generating flow using Pull and Kanban

Work Item	A		D		I		T		Done
	Queue	WIP	Queue	WIP	Queue	WIP	Queue	WIP	
		5		4		2		3	1



# Software Engineering: Flow through Pull with Kanban

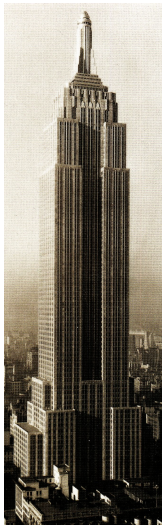


- Process controlling: local rules
- Load balancing: Kanban cards and **Work in Progress (WIP) limits**

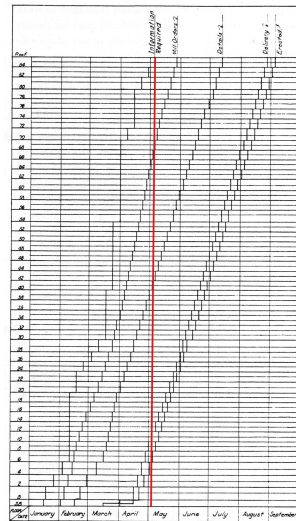
Figure from David Anderson [www.agilemanagement.net](http://www.agilemanagement.net)



# Example: Empire State Steel Construction



From *The Empire State Building* by John Tauranac



- 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




# Summary Software Development Process

- There are several types of basic process models
  - Waterfall, Iterative Development (e.g. RUP), agile processes (eXtreme Programming, Lean software development)
- Each of these process models needs to be adapted to the real situation in a company
- Some process models also depend on the culture of the company; e.g. XP does not work good in an environment with a strict process/document mindset
- While the usual saying is, different types of project need different types of projects and that agile software development is not suitable for everything, my experience shows that agile software development can be applied to a lot more project types then one thinks
  - The reason why one thinks it does not work is, because the decision makes have not tried it
- Elements from agile processes are integrated into the exercises of this course

# Exercise

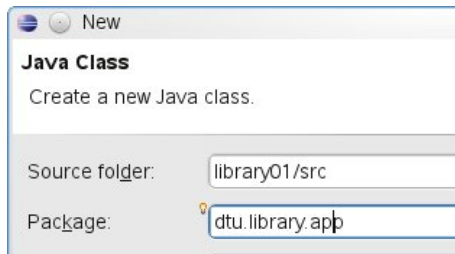
- The basic exercises for the next 7 weeks are related to the implementation of a library software
- Each week will have tasks related to this implementation
- The development will be based on agile software development principles
  - User-story driven: The development is done based on user stories that are implemented one by one
  - Test-driven: Each user-story is implemented test-driven
    - We use JUnit as the test framework
    - First the tests are provided by me, later you have to write your own tests
    - You need to uncomment each test one by one (and not all at once)
    - Use Eclipse code hints to implement missing classes/interfaces/methods

# Eclipse code hint

- Eclipse does not know the class `LibraryApp` and proposes to create it if one clicks on the light bulb .



- Make sure that the source folder ends with `src` and not `test`

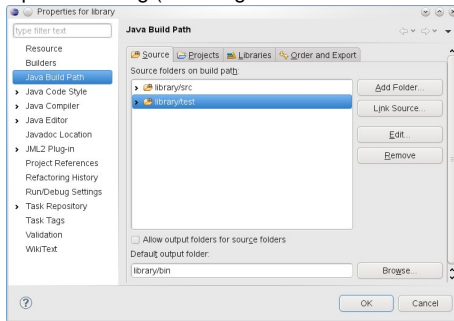


# JUnit

- JUnit is designed by Kent Beck in Erich Gamma to allow to write automated tests and execute them conveniently
- JUnit can be used standalone, but is usually integrated in the IDE (in our case Eclipse)
- We are going to use JUnit version 4.x which indicates tests to be run automatically using the `@org.junit.Test` annotation (or just `@Test` if `org.junit.Test` is imported)

# JUnit: Creating new Eclipse projects I

- With JUnit 4.x every class can have tests by just annotating the method with `@Test`
- However, I suggest to separate tests from the source code by putting them into their own source folder
  - This can be done either on creation time or by
  - Using the properties dialog (selecting Java Build Path and then Source)



- The advantage of doing so is that tests can be put in the same package as the class to test, and thus have more access rights than they would have if they would be in a separate package. And still tests are separated from production code.



# JUnit: Creating new Eclipse projects II

- In addition, the JUnit 4 libraries have to be available in the project. This can be done again in the properties dialog (selecting Java Build Path and then Libraries)

