Software Engineering I (02161)

Week 10

Assoc. Prof. Hubert Baumeister

Informatics and Mathematical Modelling
Technical University of Denmark

Spring 2011

Recap

- Design by contract (DbC)
  - Contract between client of a method and the method:
    - Client ensures precondition, method ensures postcondition
  - Implementation of DbC in Java: using the assert statement (for precondition, postcondition, and class invariant)
    - Usually assertion checking is enabled during development but not in the field (default is that assertion checking is disabled)
  - Defensive programming: The method does not rely on the client to provide correct data
    - e.g. with public library functions
- Design Patterns (II)
  - Visitor, Facade, Strategy / Policy, Decorator, Adapter / Wrapper
  - Places to find more patterns:
    - Wikipedia `http://en.wikipedia.org/wiki/Design_pattern_(computer_science)`
    - Portland Pattern repository `http://c2.com/cgi/wiki?PeopleProjectsAndPatterns` (since 1995)
    - Wikipedia `http://en.wikipedia.org/wiki/Category:Software_design_patterns`

Goal of project management

- Project management is important
  - Good project management does not guarantee success but
  - Bad project management usually results in project failure
- Goals include
  - Deliver the software to the customer at the agreed time
  - Keep overall costs within the budget
  - Deliver software that meets the customer's expectations
  - Maintain a happy and well-functioning development team

Project management for software engineering

Project management in software engineering is similar to that of other engineering disciplines.
However, there are also differences coming from the nature of the projects

1. The product is intangible
   - With construction work, one can see and feel the thing being built
2. Large software projects are often 'one-off' projects
   - Technology changes fast
   - Use of technologies changes fast
   - Experiences from past projects are not necessarily valid anymore
3. Software processes are variable and organisation-specific
   - There exist well-defined processes for different types of engineering tasks

Tasks of a project manager (I)

The tasks of a project manager will most likely include the following tasks

- Project Planning
  - Planning, estimating, and scheduling the work and assigning people to tasks
  - Monitor the progress
- Reporting
  - Report about the status of the project to customers and upper management
  - Reports exists on different level of abstraction
    a) detailed project reports
    b) concise, coherent documents presenting the critical information

Tasks of a project manager (II)

- Risk Management
  - Anticipate and manage the risks in a project
- People Management
  - The project manager is responsible for the development team
  - He has to choose the right people and provide effective means of working together
- Proposal Writing
  - Writing a proposal for winning a contract or funding is often the first step
  - Contains objectives and means how these objectives will be achieved
  - Contains Cost and schedule estimates
  - Critical task in the survival of software companies

Risk Management

Def. Risk

Something that one would prefer to have happened. Risks threaten the project, the software being developed, or the organisation.

Risk categories

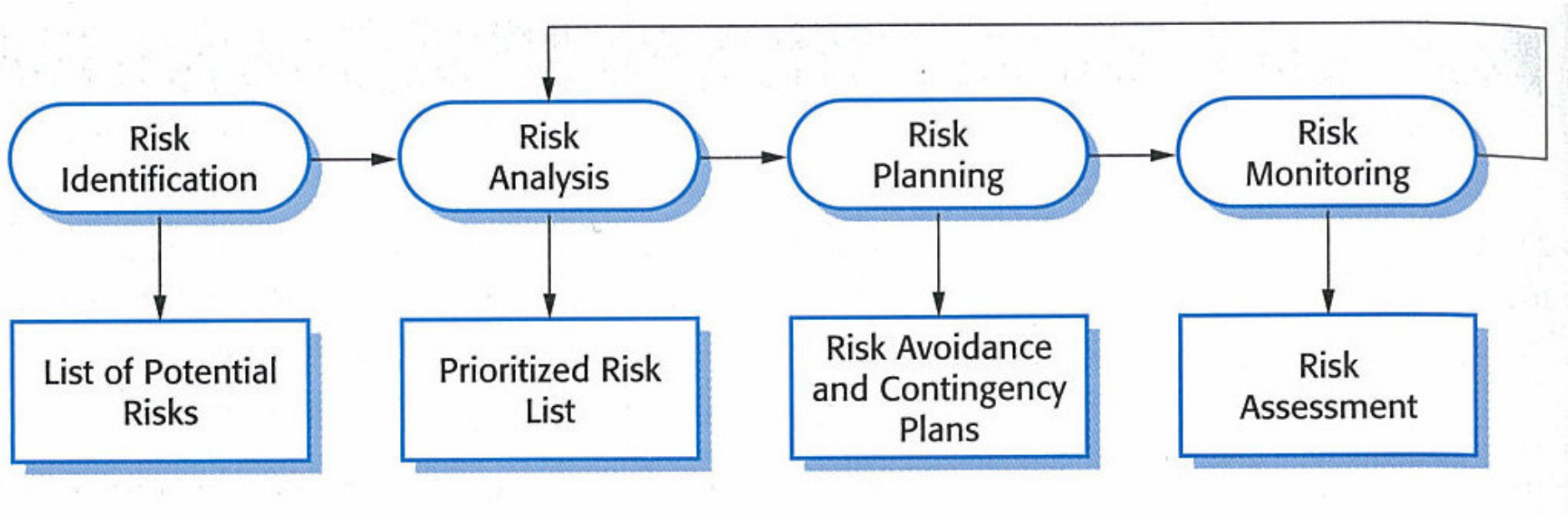
- Project risks
  - Risks affecting the project schedule or resources; e.g. loss of an experienced designer
- Product risks
  - Risks affecting the quality and performance of the product; e.g. a purchased component may not perform as expected
- Business risks
  - Risks affecting the organisation for which the software is developed or their customer; e.g. a competitor can introduce a new competing product

Examples of risks

Risk	Affects	Description
Staff turnover	Project	Experienced staff will leave the project before it is finished.
Management change	Project	There will be a change of organizational management with different priorities.
Hardware unavailability	Project	Hardware that is essential for the project will not be delivered on schedule.
Requirements change	Project and product	There will be a larger number of changes to the requirements than anticipated.
Specification delays	Project and product	Specifications of essential interfaces are not available on schedule.
Size underestimate	Project and product	The size of the system has been underestimated.
CASE tool underperformance	Product	CASE tools, which support the project, do not perform as anticipated.
Technology change	Business	The underlying technology on which the system is built is superseded by new technology.
Product competition	Business	A competitive product is marketed before the system is completed.

Software Engineering–9, Ian Sommerville, Pearson, 2010

Risk management process



Software Engineering–9, Ian Sommerville, Pearson, 2010

Risk identification

The following list of types (with examples) can be used as a checklist to see what risks there are

Risk type	Possible risks
Technology	The database used in the system cannot process as many transactions per second as expected. (1) Reusable software components contain defects that mean they cannot be reused as planned. (2)
People	It is impossible to recruit staff with the skills required. (3) Key staff are ill and unavailable at critical times. (4) Required training for staff is not available. (5)
Organizational	The organization is restructured so that different management are responsible for the project. (6) Organizational financial problems force reductions in the project budget. (7)
Tools	The code generated by software code generation tools is inefficient. (8) Software tools cannot work together in an integrated way. (9)
Requirements	Changes to requirements that require major design rework are proposed. (10) Customers fail to understand the impact of requirements changes. (11)
Estimation	The time required to develop the software is underestimated. (12) The rate of defect repair is underestimated. (13) The size of the software is underestimated. (14)

Software Engineering–9, Ian Sommerville, Pearson, 2010

Risk analysis

- After identification, risks need to be assessed
  a) How probable is the risks?
     - e.g. very low ($<$ 10%), low (10% – 25%), moderate (25% – 50%), high (50% – 75%), or very high ($>$ 75%)
  b) What is the effect if the risk scenario occurs?
     - e.g. catastrophic, serious, tolerable, insignificant

Results of a risk analysis

Risk	Probability	Effects
Organizational financial problems force reductions in the project budget (7).	Low	Catastrophic
It is impossible to recruit staff with the skills required for the project (3).	High	Catastrophic
Key staff are ill at critical times in the project (4).	Moderate	Serious
Faults in reusable software components have to be repaired before these components are reused. (2).	Moderate	Serious
Changes to requirements that require major design rework are proposed (10).	Moderate	Serious
The organization is restructured so that different management are responsible for the project (6).	High	Serious
The database used in the system cannot process as many transactions per second as expected (1).	Moderate	Serious
The time required to develop the software is underestimated (12).	High	Serious
Software tools cannot be integrated (9).	High	Tolerable
Customers fail to understand the impact of requirements changes (11).	Moderate	Tolerable
Required training for staff is not available (5).	Moderate	Tolerable
The rate of defect repair is underestimated (13).	Moderate	Tolerable
The size of the software is underestimated (14).	High	Tolerable
Code generated by code generation tools is inefficient (8).	Moderate	Insignificant

Software Engineering–9, Ian Sommerville, Pearson, 2010

Risk planning

After having identified the risks, what can one do about them:

a) Avoidance strategies
   - Reducing the probability that the risk occurs (e.g. defective components)

b) Minimisation strategies
   - Reducing the impact of a risk (e.g. staff illness)

c) Contingency plans
   - Preparing for the worst: (e.g. Organisational financial problems)

Risk planning: Strategies

Risk	Strategy
Organizational financial problems	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business and presenting reasons why cuts to the project budget would not be cost-effective.
Recruitment problems	Alert customer to potential difficulties and the possibility of delays; investigate buying-in components.
Staff illness	Reorganize team so that there is more overlap of work and people therefore understand each other's jobs.
Defective components	Replace potentially defective components with bought-in components of known reliability.
Requirements changes	Derive traceability information to assess requirements change impact; maximize information hiding in the design.
Organizational restructuring	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business.
Database performance	Investigate the possibility of buying a higher-performance database.
Underestimated development time	Investigate buying-in components; investigate use of a program generator.

Software Engineering–9, Ian Sommerville, Pearson, 2010

Risks monitoring

- Risks can change over the duration of a project
  - e.g. the environment can change, system parts are developed, ...

→ Boehm recommends monitoring the top 10 risks (Sommerville: 5 – 15 depending on the project)

- Do a reassessment of these risk
- Report on risks in each management report

Example of risk indicators

Risk type	Potential indicators
Technology	Late delivery of hardware or support software; many reported technology problems.
People	Poor staff morale; poor relationships amongst team members; high staff turnover.
Organizational	Organizational gossip; lack of action by senior management.
Tools	Reluctance by team members to use tools; complaints about CASE tools; demands for higher-powered workstations.
Requirements	Many requirements change requests; customer complaints.
Estimation	Failure to meet agreed schedule; failure to clear reported defects.

Software Engineering–9, Ian Sommerville, Pearson, 2010

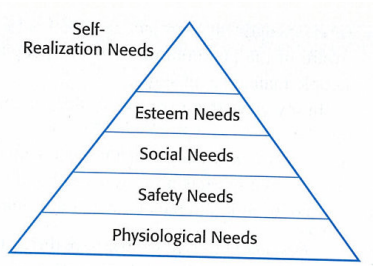
Managing people: critical factors

The critical factors in managing people are

- Consistency
  - Treat people in a project in a comparable way
- Respect
  - Respect differences in skills
  - Give all members the possibility to make a contribution
- Inclusion
  - Listen to people and take account of their proposals
  - All views are should be heard
- Honesty
  - Be honest with your staff regarding what is going good and what is going badly
  - Be honest with your own technical knowledge

Motivation

What motivates people?



- Physiological needs: hunger, thirst
- Safety needs: physical danger
- Social needs: need to communicate with other people
- Esteem needs: showing that their contribution matters
- Self realization needs: give people responsibilities and demanding (but not impossible) tasks; provide training programs to develop their skills

Software Engineering–9, Ian Sommerville, Pearson, 2010

Case study: motivation

- Observation:
  - Dorothy a hardware design expert comes late and the quality of work deteriorates and she does not talk with other members of the team
- Reason:
  - Dorothy is interested in the job but expected more hardware interfacing tasks to improve her hardware interfaceing skills; however, in the project she is doing more C programming
  - She feels that she cannot keep up with her hardware interfaceing skill and fears for her next project

Another source of motivation: Personality types

Personality types and motivation

- Task-oriented people
  - Are people who are motivated by the work they do and by the intellecutal challenge of software development
- self-oriented people
  - They are motivated by personal success and recognition. Might have longer term goals, such as career progression
- Interaction-oriented people
  - Are motivated by the presence and actions of their co-workers.

Teamwork

Project team sizes range from two to several hundred people

→ However, large teams are usually split into groups

- Optimal size: less than 10
- Putting together a group based on: technical skills, experience, and personalities
- More than a collection of people: cohesive group
  - own quality standards established by consensus
  - Individuels learn from and support each other
  - Knowledge is shared
  - Refactoring and continual improvement is encouraged

→ establish a sense of group identity

Group organisation

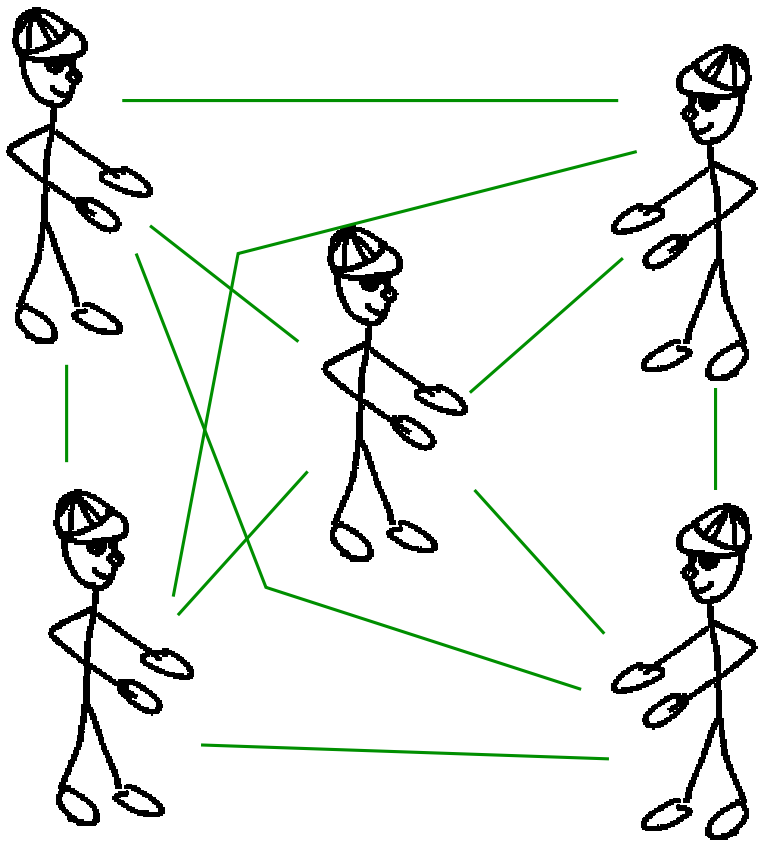
Issues to be addressed

- Is the project manager also technical lead (i.e. system architect)?
- Who will be involved in making critical technical decisions?
  - All of the team members (e.g. in XP projects); only the system architect

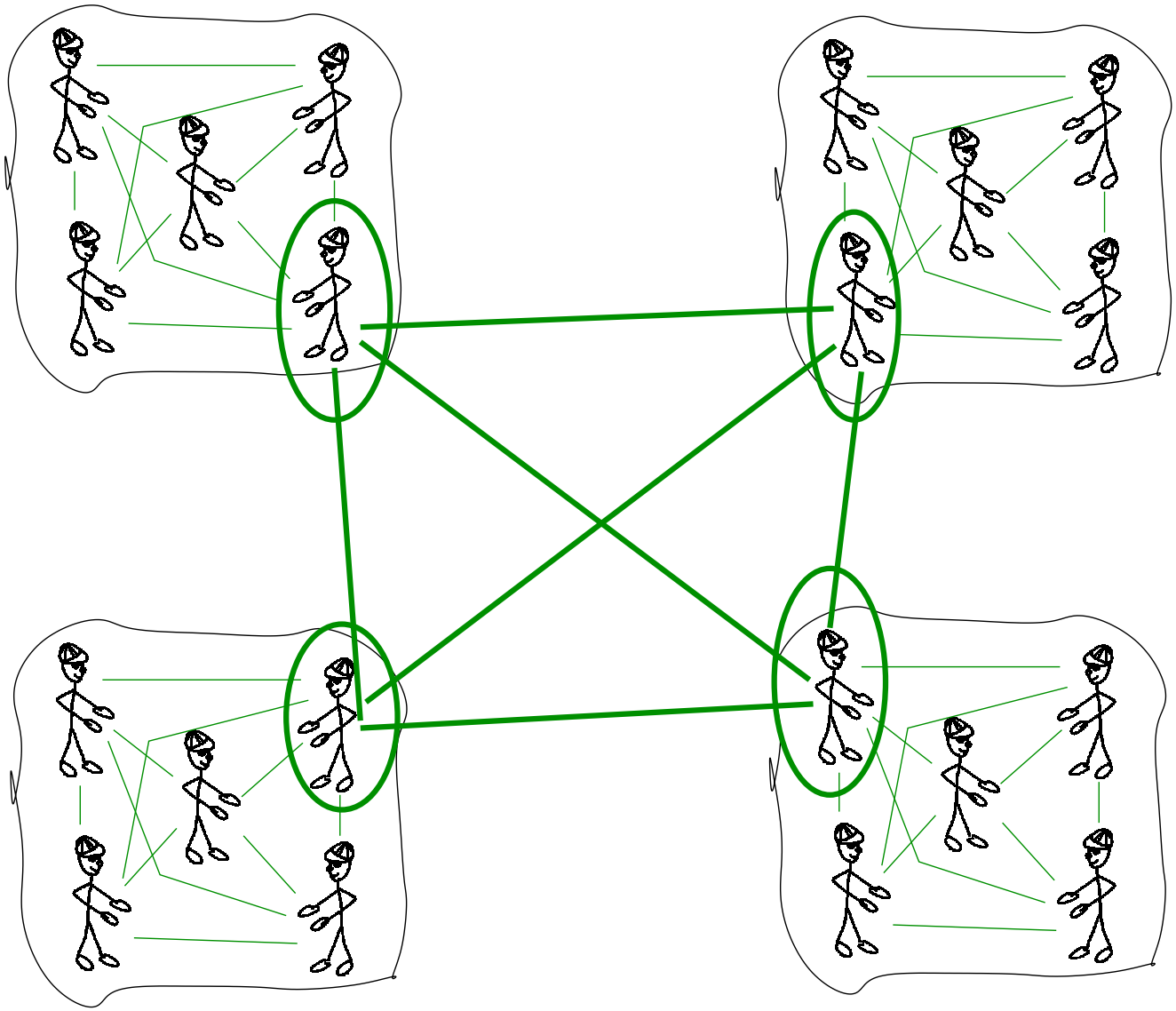
  → XP: all knowledge is spread through pair programming: thus everybody is informed to take part in critical technical decisions
- How will interactions with external stakeholders and senior company management be handled?
- How can groups integrate people who are not colocated?
- How can knowlege be shared across the group?

Group organization: Informal groups

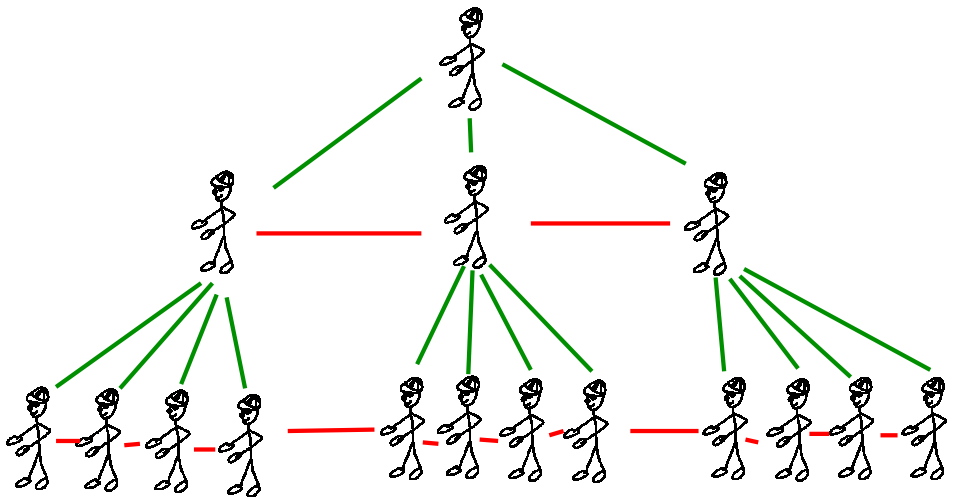
- Informal groups
  - Advocated by XP
  - Can of course still involve the concept of roles, e.g. coach, tracker, ...



Group organization: Larger informal groups



Group organization: Hierarchical groups



- Hierarchical groups
  - Group leader on top of the hierarchy with formal authority
  - Decisions are made towards the top of the hierarchy and implemented by people lower down the hierarchy
  - Communications are primarily instructions from senior staff
  - Little upward communication

Group organization: Hierarchical groups

Hierarchical groups

- Advantage: Helps solve the $n(n-1)$ problem of communication with large teams
- Disadvantages:
  - Changes in the software require the involvment of several teams; a hierarchical structure with discussions and negotiations in the hierarchy can hinder this
  - Software technology changes fast: more junior people often know more about the technology than experienced staff. Top-down communication may prevent the sensior staff from learning about new and better possibilities. Junior staff might get frustrated

Group communication

Factors that influence the effectiveness and efficiency of communication

- Group size n: Remember that communication increases with $n(n-1)$
- Group structure:
  - Informally structured groups comminicate more effectively than peopls with a formal, hiearchical structure

  → In hierarchical structure the communication tends to flow up and down the hierarchy (in particular if the group size is large!)
- Group composition
- The physical work environment
- The available communication channels, e.g. face-to-face, e-mail messages, formal documents, telephone, meetings, Web 2.0 technologies such as social networking and wikis

  → important for teams that are distributed and where team members work at home

Refactoring

- Refactoring is to restructure the software without changing its functionality
- Goal: Improve the design of the software
- With agile software development this contributes to the design activity
- This is step is necessary to keep the software simple and adaptable
- Sufficient (automated) tests are prerequisite to refactoring
  - Without tests it is not ensured the refactoring does not destroy existing functionality → regression tests

Where we have already used refactoring

- Introducing the CD and Medium class led to a big refactoring
  - Which mainly has used the renaming of methods and classes refactoring
- Introducing the presentation layer
- Introducing the persistency layer

→ New requirements required us to change the structure of the program, so that it best reflects the new functionality

Refactoring: Improving the Design of Existing Code (1999)

- Fowler describes
  - a set of refactoring,
  - when to apply them, (based on code smells)
  - and how they are done (Mechanics)
    - proceed in small steps

    → each step should lead from a compilable program that passes the tests to a compilable program that passes the tests
- E.g. renameMethod, extractMethod, encapsulateField, encapsulateCollection, ...

→ For a complete list see `http://www.refactoring.com/catalog/index.html`

Refactoring: RenameMethod

- Motivation
  - Sometimes a method name does not express precisely what the method is doing
  - This can hinder the understanding of the code; thus give the method a more intention revealing name
- Mechanics
  1) Create a method with the new name
  2) Copy the old body into the new method
  3) In the old body replace the body by a call to the new method; compile and test
  4) Find all the references to the old method and replace it with the new name; compile and test
  5) Remove the old method; compile and test
- Note that Eclipse has nice support for this refactoring

→ Together with the support from Eclipse, this becomes a very powerful tool to improve ones code

Code smells

If it stinks, change it Refactoring, Martin Fowler, 1999

- Duplicate Code
- Long Method
- Large Class
- Long Parameter List
- Divergent Change
- Shotgun Surgery
- Feature Envy
- Data Clumps
- Primitive Obsession
- Switch Statements
- Parallel Inheritance
- Lazy Class
- Speculative Generalisation
- Temporary Field
- Message Chains
- MiddleMan
- Inappropriate Intimacy
- Alternative Classes With Different Interfaces
- Incomplete Library
- Data Class
- Refused Bequest
- Comments

MarriageAgency code

Method matchCustomer in class MarriageAgency

```
public ArrayList<Customer> matchCustomer(Customer customer){
  ArrayList<Customer> res = new ArrayList<Customer>();
  for (Customer potential : customers) {
    if (potential.getSex() != customer.getSex()) {
      int yearDiff = Math.abs(potential.getBirtYear()
                    -customer.getBirtYear());
      if (yearDiff  <= 10) {
        for (String interest : potential.getInterests()) {
          if (customer.getInterests().contains(interest)) {
            res.add(potential);
            break;
          } } } } }
  return res;
}
```

Bad Smell: Long methods

- Methods in object-oriented programs should be short
- They should do a lot of delegation
- Long methods are an indication that an object is doing too much work by themselves
  - → We already discussed this when we discussed about centralised- and decentralised control
- Solution
  - Use ExtractMethod to introduce delegation

Bad Smell: Comments

- Comments are good if they explain things that are not visible in the code, e.g. design decisions (why you wrote the code in this way)
- Sometimes comments try to hide badly written code; they act as a deodorant to cover the bad smells
- Solution
  - Remove the underlying bad design and see if the comments are still necessary in the refactored code
  - Use ExtractMethod or RenameMethod to let the code speak for itself

Refactoring: ExtractMethod

- Motivation:
  - For code fragments that can be grouped together introduce a new method
- Mechanics
  1) Create a new method with a name revealing the intention of the code
  2) Copy the code fragment into the new method

  → Complications: references to variables defined outside the scope of the code fragment have to passed as parameter

  → Sometimes other refactorings have to be done before ExtractMethod becomes possible (e.g. in our example to InlineTemp
- Note that Eclipse has nice support for this refactoring

→ Together with the support from Eclipse, this becomes a very powerful tool to improve ones code

Refactoring: InlineTemp

- Motivation:
  - A temp is assigned to once and comes in the way of further refactorings
- Mechanics
  1) Declare the temp as final and compile; Why is this step helpful?
  2) Find all references of temp and replace them with the right hand side of the expression; compile and test
  3) Remove the declaration of temp and the assignment to temp; compile and test
- Note: There exists also an opposite refactoring that introduces temps for expressions

→ A lot of refactorings have versions that do exactly the opposite

→ Refactorings should be used to avoid code smells

Code smell: A method is doing two things

- A method should usually do one thing and only one thing and that good.
- If a method tries do to too many things, try to split the method in several methods
- In our example hasOneInterestInCommon determines what it means to have one interest in common and also adds a potential customer to the list of matches

Code smell: Inappropriate Intimacy

- If a class uses to many features of another class this can be a sign that code and fields may need to move to other classes

→ In our example, class MariageAgency itself computes the matching conditions instead of the delegating this to the customer class

Refactoring: Move Method

- Motivation
  - A method is using or used by more features of another class than the class on which it is defined
  - Move that method to the other class
  - E.g. the methods hasOppositeSex, hasAppropriateAge, hasOneInterestInCommon use instance variables of the customer object; thus these should be methods of class customer
- Mechanics
  1) Declare the method in the target class
  2) Copy the body of the old method to the new method
  3) If necessary, pass on the original object
  4) Replace the body of the old method by the call to the new method; compile and test
  5) Replace all references to the old method with calls to the new method; compile and test
  6) Remove the old method; compile and test

Summary

- Refactoring improves the quality of the code by removing code smells
- Refactoring is applied to prepare the code for adding new functionality and improving the code of existing functionality (after, e.g., new functionality has been added)
  - → Refactoring plays an important role in good design
- Refactoring depends on sufficient test cases
- Refactoring itself are described through their mechanics
  - Small steps leading from correct (compilable and tests pass) to correct programs

→ Today, IDE's support the most important refactorings

Summary

- Project Management
  - Project Planning (in one of the previous lectures)
  - Risk Management
  - Team Management
- Refactoring