# Programming Engineering

Course 12 – 23 May

# Outline

- Previous courses…
  - Testing
  - SOLID Refactoring
- Program Quality
- Metrics
- Copyright

# Degradation of code

- **Rigidity** – the tendency for software to be difficult to change, even in simple ways
- **Fragility** – the tendency of the software to break in many places every time it is changed
- **Immobility** – the inability to reuse software from other projects or from parts of the same project
- **Viscosity** – it is easy to do the wrong thing, but hard to do the right thing

# SOLID Refactoring

- Cohesion
  - The Release Reuse Equivalency Principle
  - The Common Closure Principle
  - The Common Reuse Principle
- Coupling
  - Acyclic Dependencies Principle
  - The Stable Dependencies Principle
  - The Stable Abstractions Principle

# Assessing quality

- How do we measure the quality of an item?
  - **Construction quality** (how well it is built, whether the raw material has flaws, etc ...)
  - **Design quality** (comfort, elegance...)
  - A combination between quality of design and construction (sturdiness...)

- In general, we can say that chair A is better than chair B regarding some particular aspect, but it is usually difficult to say by how much.

# Software Quality (1)

- We do not assess construction quality (=> unique among engineering applications)
- All quality attributes refer to design.
- Esthetic qualities:
  - Software is mostly invisible, and esthetics only matter for the visible elements
  - Apart from the GUI, observable aspects software are:
    - Notations for design and writing of code
    - Behavior of software when interacting with other entities.

# Software Quality (2)

- When discussing software quality we must:
  - **Define those attributes of quality** that are of interest;
  - Determine a way of **measuring those attributes**;
  - Find a way of **representing design**;
  - **Write specifications** that will guide developers (following and implementing design qualities).

# Source Code

- Code that implements a given design is a representation of that design.
- Performing quality assurance after writing code is expensive and possibly useless.
- Usually, only the manner in which the code is written is taken into account (coding style, design patterns, adaptability, maintenance, reuse (coupling, cohesion), security)

# Software Quality (3)

- Measures the appropriateness of software to the environment it is used in.

- Variuos aspecte taken into account are:
  - The software is running;
  - The software performs according to specifications;
  - The software is safe;
  - The software can be adapted as requirements change.

- All measurements regarding quality are relative!

# Aspects of Software Quality

- Safety


- Efficiency

- Maintenance

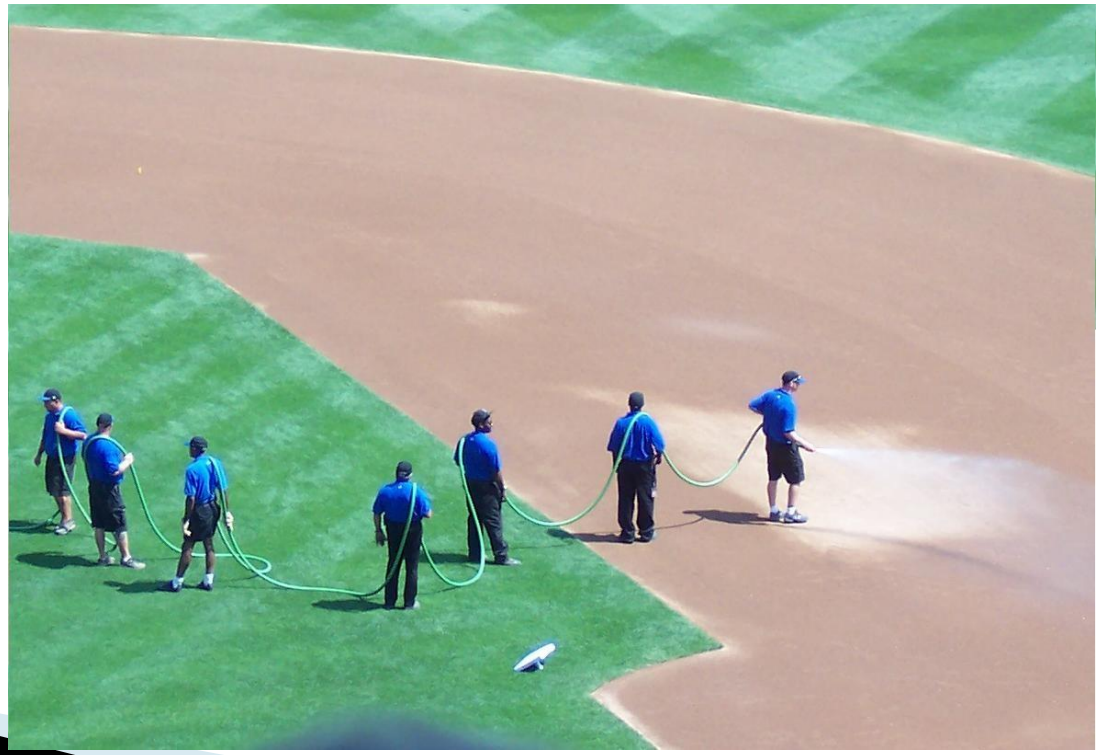

- Usability

# Safety

- Is the software **complete, correct and robust**?
  - **Completeness** – works for all possible inputs;
  - **Consistency** – always behaves as expected;
  - **Robustness** – behaves well in abnormal situations (eg. Lack of resources, lack of internet connection, etc.)

# Efficiency

- The software makes efficient use of available resources (CPU time, network connection, etc.)
- Efficiency is always less important than safety. It is easier to make safe software efficient than the reverse



# Maintenance


- How easily can the design be changed or adapted?

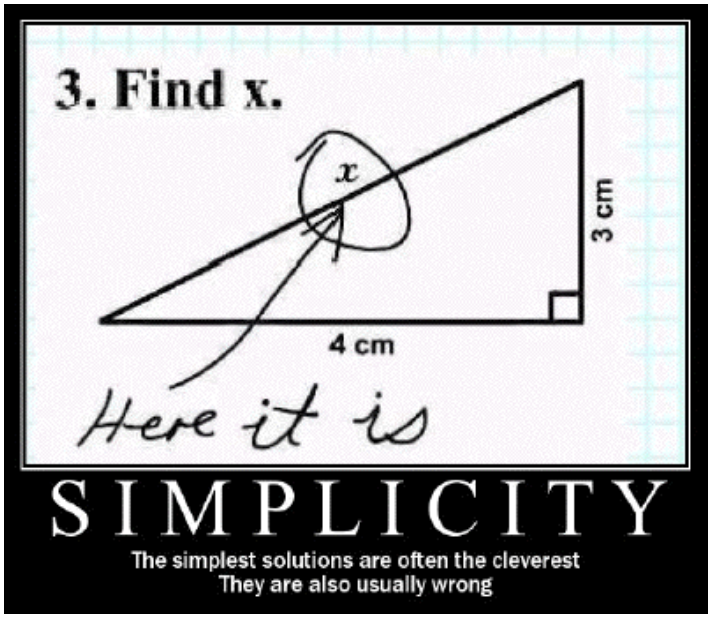
- Types of maintenance:
  - **Corrective**: error fixing;
  - **Perfective**: adding features that should have been part of the product;
  - **Adaptive**: updating software as requirements change.

# Usability

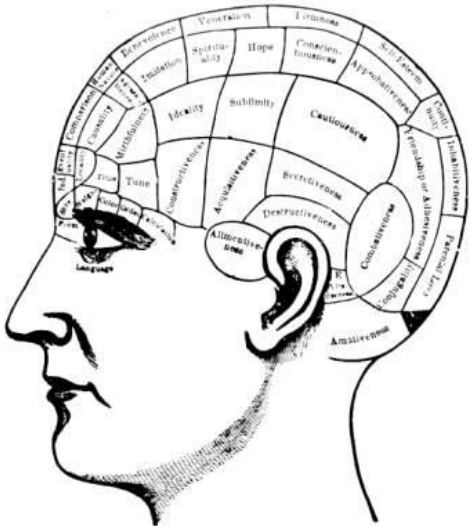
- How easily can the software be taught and used?

# Measurable Attributes

- Simplicity




- Modularity



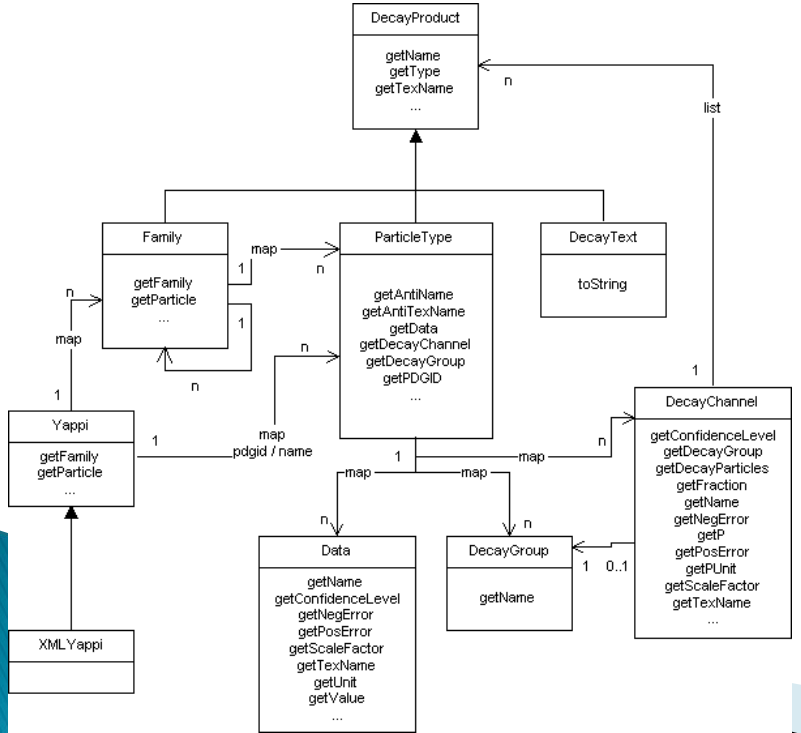
# Simplicity

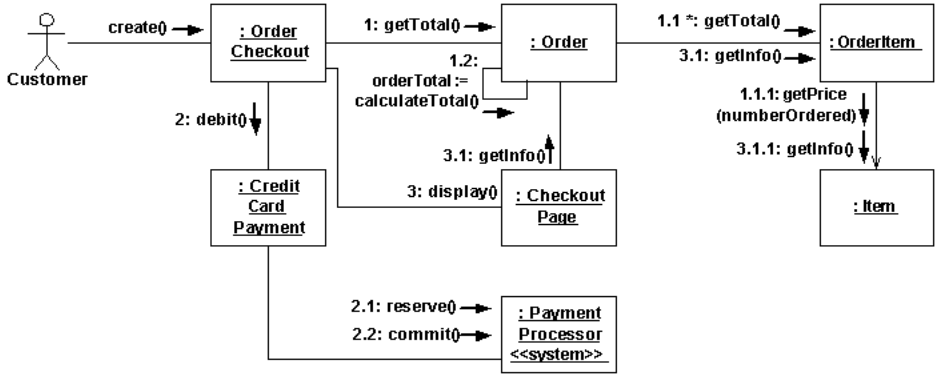
- The reverse of complexity.

- Aspects of complexity:
  - **Control flow**: counts all the possible execution paths for a program
  - **Information flow**: measures amount of data transmitted within the program
  - **Understanding**: counts the number of identifiers and operators

# Modularity

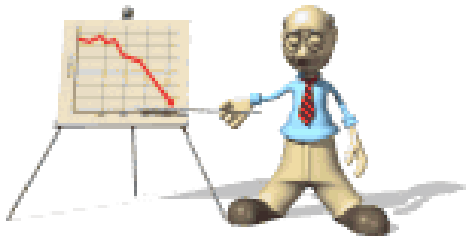
- Can be measured by examining :
  - **Cohesion**: how well the components of a module collaborate.
  - **Coupling**: interaction between modules



# Why Use Metrics?

- We use metrics to
  - understand
  - control
  - predict



# What Should We Measure

- Size of software
- Complexity of software
- Robustness of software
- Amount of time required to develop some software
- Resource allocation for development
- Productivity of effort
- Development costs

# Estimation

- Intuitively, estimation seem subjective
  - To inexperienced persons, it looks like predicting the future
  - This is reinforced when estimation is incorrect and projects are delivered late
- Formal estimation processes
  - allows the project team to reach a consensus on the estimates
  - improve the accuracy

# Estimation

- Successful estimations take into account the following
  - Work Breakdown Structure (WBS) – what are the tasks that need to be performed to finish the product?
  - Assumptions – how to deal with incomplete information
  - Trust – if stakeholders and engineers trust each other, the estimate will be more accurate

# Work Breakdown Structure

- A list of tasks that, if completed, will produce the final product
  - Broken down by feature
  - By project phase (requirements tasks, design tasks, programming tasks, QA tasks, etc.)
  - Some combination of the two
  - Should reflect the way previous projects have been developed

# Work Breakdown Structure

- A project should be broken down into 10 – 20 tasks
  - Regardless of the size of the project
  - For large projects (e.g. an operating system), the tasks are large
  - For smaller projects, the tasks are correspondingly smaller
- Create an estimate for the cost of each task
  - Most accurate estimates are those that rely on prior experience
  - NO estimate is guaranteed to be accurate

# Assumptions

- At the beginning of the development team members do not have all the information
  - Assumptions are needed to fill in missing things
  - Assumptions can also be placeholders which will be corrected later
  - If an assumption is proven incorrect, the timeline of the project MUST be adjusted
- For effective estimates, assumptions need to be written down
  - If not, the team will need to have the same discussion again

# Assumptions and Trust

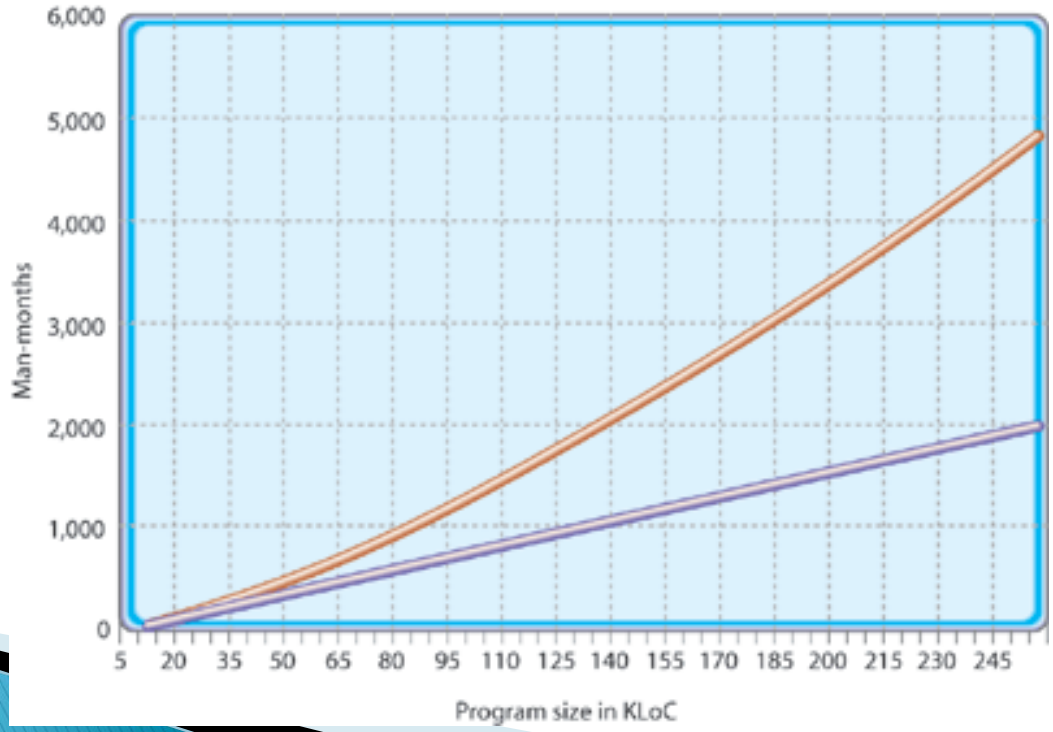
- Estimates can either be a source of trust or distrust between the project team and managers.
- Stakeholders need the project completed but usually do not have software engineering experience
- Project managers must take care to make the estimation process as open and honest as possible

# Assumptions and Trust

- It is common for nontechnical people to assume that programmers pad their estimates
  - They have a “rule” by which they cut off a third or half of any estimate
  - This lack of trust causes engineers to automatically pad their estimates
- An important part of running successful software projects is reaching a common understanding between the engineers, managers, and stakeholders.

# Basic Metrics

- KLOC: Kilo Lines Of Code
- Effort, PM: Person – Month



# COCOMO 2

- Boehm 1995
- Takes into consideration high level development tools and techniques
  - Prototyping
  - Modular development
  - 4GL (fourth generation language)
- Allows for estimates from the very first stages of development

# COCOMO 2: Initial prototyping

- Effort required to create a prototype of the application
- Based on the Number of Object Points (NOP)
- Formula for computing effort:

$$PM = (1 - P_{reuse}) \frac{NOP}{PROD}$$

# NOP

- Investigate the screens and dialogs that are needed
  - Simple: 1
  - Complex: 2
  - Very complex: 3
- Reports that need generated
  - Simple: 2
  - Complex: 5
  - Very complex: 8
- Each lower level module (eg. 3GL): 10
- The sum of all of the above represents the NOP.

# COCOMO 2: after prototyping

- Estimate the total lines of code (ESLOC)
- Takes into account
  - Requirements instability
  - Possibilities of code reuse

# COCOMO 2: Influences on Costs

- Product attributes
  - Safety, module complexity, size of user manual, size of the required database, amount of reusable components
- Platform attributes
  - Constraints referring to execution time; platform volatility, memory constraints

# COCOMO 2: Influences on Costs (cont.)

- Personnel attributes
  - Analyst experience; developer experience; personnel continuity; knowledge of the domain of the problem to be solved with regards to analysts and developers; knowledge of the programming language and development tools

- Project attributes
  - Required tools; distance between development teams (eg. different countries) and quality of communication; Development plan compression

# The Planning Game

- A planning method from Extreme Programming (XP)
- A method used to manage the negotiation between the engineering team (Development) and the stakeholders (Business)
  - Treats the planning process as a game
  - playing pieces are “user stories” written on cards
  - the goal is to assign value to stories and put them into production over time

# The Planning Game

- Unlike other planning methodologies, it does not require a documented description of the scope of the project to be estimated
- The Planning Game combines
  - estimation
  - identifying the scope of the project
  - Identifying the tasks required to complete the software

# The Planning Game

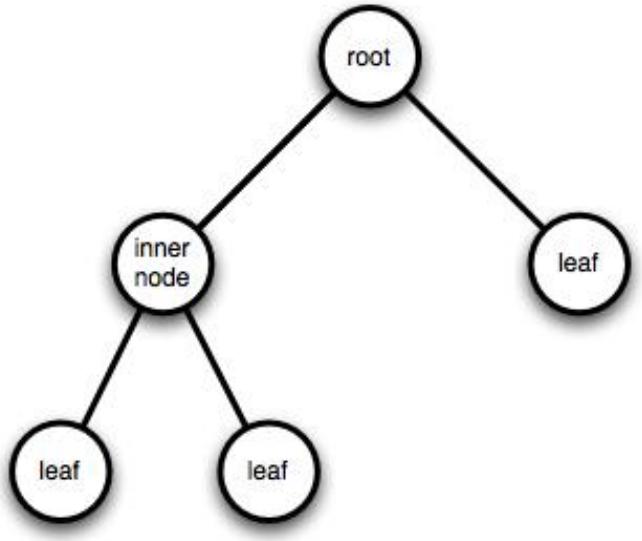
- The planning process is highly iterative. Each iterations looks like this:
  - **Scope** is established by having Development and Business work together to interactively write the stories.
  - Each **story** is given an estimate of 1, 2, or 3 weeks.
    - Larger stories are split into multiple iterations
  - Business is given an opportunity to steer the project between iterations.
  - The **estimates** are created by the programmers, based on the stories that are created.
  - **Commitments** are agreed upon

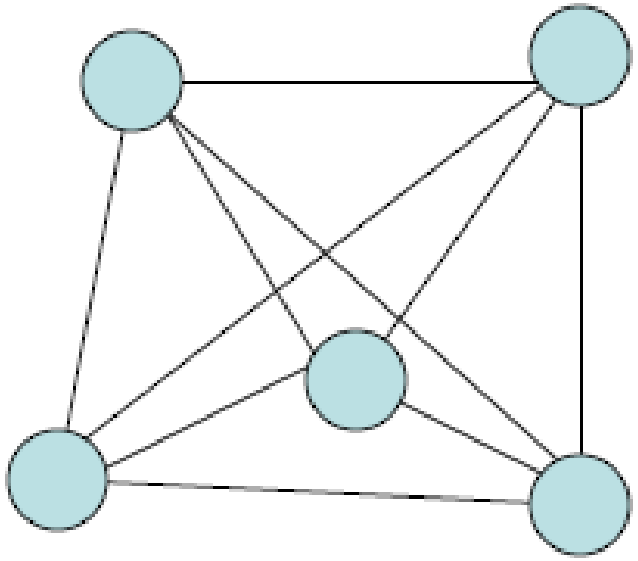
# Distributing Workforce Over Time

- 20 PM. Are the following correct?
  - 20 people working 1 month
  - 4 people working 5 months
  - 1 person working 20 months
- Individual productivity decreases as team size increases
  - Communication overhead
  - On adding new members, productivity decreases initially
- *Adding people to a team behind of schedule makes that project more behind schedule.* (Brooks' law)

# Distributing Workforce Over Time (2)

- For a team with P members, one can have between P-1 and P(P-1)/2 communication channels
- Each channel is a decrease in efficiency



# How Not to Plan and Estimate Costs

- We have 12 months to finish the job, so it will take 12 months.
- A competitor asked for $1.000.000. We wil ask for $900.000.
- The client budget is $500.000. That will be the exact cost of development.
- Development takes 1 year, but we say it will take 10 months. A delay of 2 months is not important...

# Problems with Metrics

- Lack of accuracy
- Employee pushback
- Use for other purposes than intended
- Animosity within the development team

# Copyright

- The rights enjoyed by authors with regards to their work;
- Copyright is the instrument of protection of authors and their work;
- *Copyright gives the creator of an original work exclusive right for a certain time period in relation to that work, including its* ***publication, distribution and adaptation****; after which time the work is said to enter the public domain.*

# Exclusive rights

- Several exclusive rights typically attach to the holder of a copyright:
  - to produce copies or reproductions of the work and to sell those copies (mechanical rights; including, sometimes, electronic copies: distribution rights)
  - to create derivative works (works that adapt the original work)
  - to perform or display the work publicly (performance rights)
  - to sell or assign these rights to others
  - to transmit or display by radio or video (broadcasting rights)

# Bibliography


- Extreme Programming Explained by Kent Beck (Addison Wesley, 2000)
- Applied Software Project Management, by Andrew Stellman and Jennifer Greene (O'Reilly, 2006)

# Links

- **Bug Life Cycle:** http://www.buzzle.com/editorials/4-6-2005-68177.asp, http://qastation.wordpress.com/2008/06/13/process-for-bug-life-cycle/
- **COCOMO:** http://en.wikipedia.org/wiki/COCOMO
- Curs 12, Ovidiu si Adriana Gheorghies: http://www.info.uaic.ro/~ogh/files/ip/curs-12.pdf