Designing Object-Oriented Software

Jouni Smed

2006

Course Syllabus

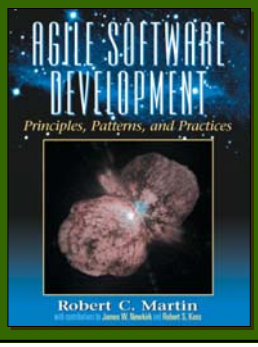
- Credits: 5 cp (3 cu)
- Prerequisites:
  - Ohjelmointi II [Programming II]
  - object-oriented programming
- Teaching methods: lectures
  - Tuesdays 10–12, Etäluokka
  - from March 7 to April 25
- Assessment: examination only
- Course web page:
  http://staff.cs.utu.fi/staff/jouni.smed/doos/

Examinations

- Tentative examination dates
  - May, 2006
  - June, 2006
  - September, 2006
- Check the exact times and places at http://www.it.utu.fi/opetus/tentit/
- If you are not a student of University of Turku, you must register to receive the credits
- Remember to enrol in time!

Textbook for the Course

- Martin, Robert C.: *Agile Software Development: Principles, Patterns, and Practices*, Prentice-Hall, 2003,
  ISBN: 0-13-597444-5
- This course will rely heavily on the textbook, which is why obtaining the book is necessary!
- You can use the textbook in the examination



Outline of the Course

Lecture	Book	Topics
1.	—	—
2.	Section 1	Planning, testing, refactoring
3.	Section 2	SRP, OCP, LSP, DIP, ISP
4.	Section 3	Design patterns 1
5.	Section 4	Package design
6.	Section 5	Design patterns 2
7.	Section 6	Design patterns 3

Programming in the 1940s and 1950s

- Programming = instructing the machine operations
  - machine language or assembler
  - machine-oriented programming
- Far from the way programmers (humans) think
- The problem domain remained close to the machine world
  - making calculations, sorting data etc.
- Main goals of design
  - enable the programmer to write the software
  - focus on the design of algorithms and data structures.
- Implementing the design was difficult

Programming in the 1960s and 1970s

- 'High-level' programming languages
  - Algol, Fortran, Pascal, Ada, C, …
  - problem-oriented programming
- Innovations
  - reducing machine dependency: portability
  - managing complex and large problems: structured programming, modular programming and information hiding
- Main goals of design
  - enable an automatic transformation of an analysis model to a program structure
  - managing the work of many simultaneous programmers
  - keeping the system maintainable

Programming in the 1960s and 1970s (cont'd)

- Programming comprised
  - creating and manipulating complex data structures and algorithms
  - realizing subroutines
  - dealing with the physical organization of the software
- The problem domain took a big leap towards the real world
  - commercial information systems, traffic, science…
- This was the era of waterfall way of building systems, which led to the so-called software crisis

Programming in the 1980s and 1990s

- Rise of the object-oriented (OO) paradigm
  - real-world concepts are directly supported with programming language constructs (e.g. classes)
- Detailed design
  - how to implement an analysis model with OO mechanisms?
  - class-level reuse, specifying the programming task for a programmer, understandability and maintainability of the source code
- Architecture design
  - how to create, describe and manage the big picture of a system
  - maintainability of the system, manageability during design (multiple team development), handling non-functional properties of the system, high-level reuse, adaptability of the system…
- Programming was still seen as realizing the design

Programming Now

- The new software crisis: the bloat of design work
  - expensive
  - opposes maintainability and adaptability
- Iterative way of building large systems changes the role of design
  - the design is created piece by piece as the understanding of the problem grows
- Programming and detailed design are unifying
  - high-level constructs allow to express the design in the code
  - design patterns allow even to express the architecture design
- The role of the programmer is rising
  - operates at the design level
  - understands the profound object-oriented principles

Three Perspectives on Software Development

- Conceptual
  - represents the concepts in the problem-domain
  - objects defined in the terms of responsibilities
- Specification
  - focuses on the software at the level of interfaces (not the implementation)
  - how the modules are connected
- Implementation
  - looks inside the modules, the code
  - probably the most often used perspective (should not be)
  - objects are seen as encapsulating data and providing access to services

Design

Programming

Traditional Engineering…

- The design is expressed in the blueprints
- Engineers try to make absolutely sure that
  - the design is correct
  - requirements are met
  - the product will function as specified
- Why?
  - building the product is expensive
  - construction cannot be undone (or it is expensive to do so)
- Construction is done by different people → The blueprints must contain all information needed for construction

…and Software Engineering

- Source code is the blueprints
- Compiler does the actual construction work
- Since construction is virtually costless it can be redone over and over again!
  - traditional engineering ≠ software engineering

What Makes Software Systems Complicated?

- The problem being solved
  - finding and understanding the requirements
- The software itself
  - managing and understanding the software
  - finding a solution that meet the requirements
- The software organization
  - managing the employees working on the same system
- The software industry
  - constantly changing environments
  - market situation
  - platform technologies

Programming vs. Design?

- Complexity and volatile requirements
  → iterative approach with feedback
- Construction is cheap
  - no need to speculate by building models, prototypes, simulations
- Iteration can cover analysis-design-implementation-testing phases
  → the act of design in software engineering
- Programming is
  - constructing the software (i.e. programming)
  - designing the software
- Not designing the program but programming the design

Development

- Planning
- Testing
- Refactoring

Planning

- Initial exploration
  - developers and customers identify all *significant* user stories
  - estimate the cost of the stories
    - splitting and merging
    - velocity ← cost and priority of a story
- Release planning
  - a crude selection of stories to be implemented in the first release
    - business decisions
  - developers and customers agree on a date for the first release
    - typically 2–4 months in the future
- Iteration planning
  - developers and customers agree on the iteration length
    - typically 2 weeks

Iteration

- Start:
  - developers and customers get together
  - customers choose stories to be implemented
    - no more than velocity allows
    - cannot be changed once the iteration has begun
  - task planning: developers break the stories down to development tasks (implementable in 4–16 h)
- Halfway point:
  - the team meats and assesses the progress so far
- End:
  - iteration ends on the specified date (regardless whether the stories are done)
  - developers demonstrate the current running executable to the customers for evaluation
  - the velocity is updated

Testing

- Writing unit tests is an act of
  - design
  - documentation
  - verification
- Test driven development: design tests before you design the program
- Effects
  - backstop for further development: add and change without fear of breaking the existing software
  - different point of view: write from the vantage point of a caller of the program → interface and function of a program
  - forces to decouple the software
  - documentation: how to call a function? check the test!

Testing (cont'd)

- Unit tests
  - white-box tests that verify individual mechanisms
  - do not verify that the system works as a whole
  - documents the internals of a system
- Acceptance tests
  - black-box tests that verify that customer requirements are being met
  - written by people (customers, QA) who do not know the internal mechanisms of the system
  - documents the features of a system

Refactoring

- Practical definition: altering the source code systematically to improve its design
  - easier to understand
  - cheaper to modify
  - does not change its observable behavior
  - the goal is not better performance

Benefits of Refactoring

- Improves the design of the software
  - creates the design in the existing code
  - adjusts the design piece by piece to changing functionality
- Helps in finding bugs
  - clarifies the purpose of the code to a point where you simply cannot avoid seeing the bugs
- Helps in programming faster
  - without refactoring you start faster but lose speed after a while

When to Refactor?

- Rule of three
  - first time: just do it
  - second time: you may duplicate
  - third time: refactor
- Refactor when you add functionality
  - refactor the code that is going to change before you make the change to understand it deeply
  - if the change does not fit in easily, refactor the design to enable smooth addition of the new feature
- Refactor when you need to fix a bug
  - a bug indicates that the code is not easy to understand
- Refactor in a code review
  - if you are going through the code, why not go through the design as well
- Refactor all the time
  - it is an integral part of designing and programming

What Goes Wrong with Software?

- You start with a clear picture in your mind
- Then something goes wrong
  - changes and additions are harder and harder to make
  - you are forced to let go of the original design ideas
  - eventually even the simplest changes terrify you because of rippling unexpected effects, and you must redesign the whole software.
- You started with good intentions, so what went wrong?

Seven Deadly Sins (or Symptoms of Poor Design)

1. Rigidity
2. Fragility
3. Immobility
4. Viscosity
5. Needless complexity
6. Needless repetition
7. Opacity

Reading for the Next Week

- Section 2: Agile Design
  - Chapter 7: What Is Agile Design?
  - Chapter 8: The Single-Responsibility Principle
  - Chapter 9: The Open–Closed Principle
  - Chapter 10: The Liskov Substitution Principle
  - Chapter 11: The Depency-Inversion Principle
  - Chapter 12: The Interface-Segregation Principle