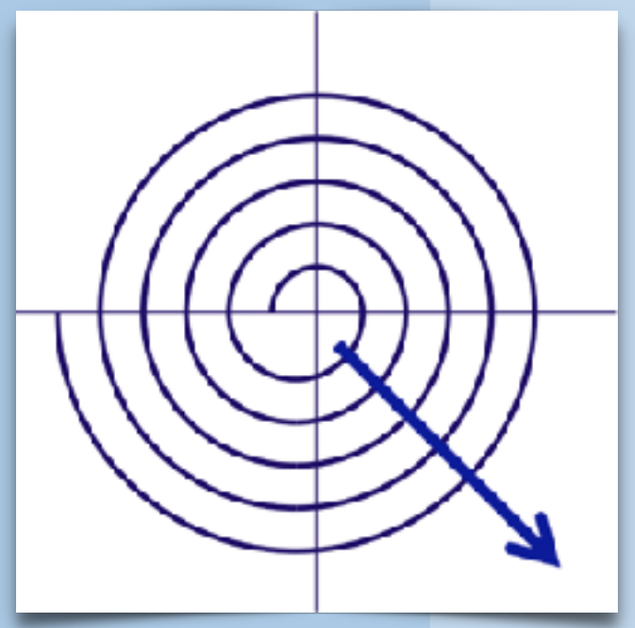
UNIVERSITÄT
BERN

Introduction to Software Engineering (ESE : Einführung in SE)

Prof. O. Nierstrasz



Selected material courtesy of Prof. Serge Demeyer, U. Antwerp

ESE — Introduction

Lecturers	Prof. Oscar Nierstrasz, Dr. Mohammad Ghafari
Assistants	Nitish Patkar, Manuel Leuenberger Silas Berger, Patrick Hodel
Lectures	Engehaldenstrasse 8, 001, Wednesdays @ 14h15-16h00
Exercises	Engehaldenstrasse 8, 001 Wednesdays @ 16h00-17h00
WWW	scg.unibe.ch/teaching/ese

This is a note (a hidden slide). You will find some of these scattered around the PDF versions of the slides.

NB: some links to copyrighted materials are only accessible within the unibe.ch domain.

Roadmap

- Course Overview
- What is Software Engineering?
- The Iterative Development Lifecycle
- Software Development Activities
- Methods and Methodologies

Roadmap

- Course Overview
- What is Software Engineering?
- The Iterative Development Lifecycle
- Software Development Activities
- Methods and Methodologies

Principle Texts

- *Software Engineering*. Ian Sommerville. Addison-Wesley, 10th edition, 2015
- *Software Engineering: A Practitioner's Approach*. Roger S. Pressman. McGraw Hill; 8th edition, 2003.
- *Designing Object-Oriented Software*. Rebecca Wirfs-Brock and Brian Wilkerson and Lauren Wiener. Prentice Hall; 1990
- *UML Distilled*. Martin Fowler. Addison-Wesley; 3rd edition, 1999

Sommerville's book is the classic SE textbook, offering a good overview of all aspects. It is not prescriptive, however, so does not specific advice. Pressman offers a more personal view, based on experience. Wirfs-Brock's classic book offers good insights into the principles of object-oriented design. Fowler's short book gives excellent advice on how to use UML effectively.

Recommended Literature

- *eXtreme Programming Explained: Embrace Change.* Kent Beck. Addison-Wesley; 2nd edition, 2004
- *The Mythical Man-Month: Essays on Software Engineering.* Frederick P. Brooks. Addison-Wesley; 2nd edition,1995
- *Peopleware: Productive Projects and Teams.* Tom Demarco and Timothy R. Lister. Dorset House; 2nd edition, 1999
- *The Psychology of Computer Programming*. Gerald Weinberg. Dorset House; Silver Anniversary Edition, 1998

Beck's book gives a good introduction into the principles behind extreme programming and agile development. Brooks' classic book explains how and why large projects can fail. Peopleware explains how important the dynamics of interaction are to successful SE projects. Weinberg's classic was the first to study human interaction in software projects, and introduces the notion of "egoless programming" (which is arguably a key component of agile methods today).

Course schedule (tentative)

Week	***Date***	***Lesson***
1	19-Sep-18	Introduction: The Software Lifecycle
2	26-Sep-18	Requirements Collection
3	3-Oct-18	*Guest lecture: Agile Methods (Pietari Kettunen)*
4	10-Oct-18	Responsibility-Driven Design
5	17-Oct-18	Modeling Objects and Classes
6	24-Oct-18	Modeling Behaviour
7	31-Oct-18	Software Security
8	7-Nov-18	*Guest lecture: Software Testing (Manuel Oriol)*
9	14-Nov-18	User Interface Design
10	21-Nov-18	Software Quality
11	28-Nov-18	Software Metrics
12	5-Dec-18	*Guest lecture: Project Management (Jan Hornwall)*
13	12-Dec-18	Software Architecture + *Guest lecture: Software Architecture in practice (Erwann Wernli)*
14	19-Dec-18	*Guest lecture: SE in practice (Peter Gfader)*
15	10-Jan-19	Final Exam: ExWi A6 @ 10h00-12h00

Roadmap

- Course Overview
- **What is Software Engineering?**
- The Iterative Development Lifecycle
- Software Development Activities
- Methods and Methodologies

Why Software Engineering?

A naive view:

Problem Specification —coding→ Final Program

But ...

- Where did the *specification* come from?
- How do you know the specification corresponds to the *user's needs*?
- How did you decide how to *structure* your program?
- How do you know the program actually *meets the specification*?
- How do you know your program will always *work correctly*?
- What do you do if the *users' needs change*?
- How do you *divide tasks up* if you have more than a one-person team?

What is Software Engineering? (I)

Some Definitions and Issues

"state of the art of developing quality software on time and within budget"

- Trade-off between perfection and physical constraints
  - —SE has to deal with real-world issues
- State of the art!
  - —Community decides on "best practice" + life-long education

Engineering as a discipline is concerned with *best practices* for developing *physical products* within *budget*, on *time*, and fulfilling certain *quality* requirements. Engineering is therefore different from science. Engineering practices cover not only technological aspects of the products being built, but also such diverse aspects such as planning, process management, and quality standards.

Software Engineering attempts to apply similar principles to the production of software products, though software is inherently not physical.

What is Software Engineering? (II)

"multi-person construction of multi-version software"

— Parnas

- Team-work
  - —Scale issue ("program well" is not enough) + Communication Issue
- Successful software systems must evolve or perish
  - —Change is the norm, not the exception

What is Software Engineering? (III)

"software engineering is <u>different</u> from other engineering disciplines"

— Sommerville

- Not constrained by physical laws
  - —limit = human mind

- It is constrained by political forces
  - —balancing stake-holders

Sommerville emphasizes the fact that SE is different from other kinds of Engineering since the product is not constrained by any physical laws. As a simple example, software can be copied essentially for free, which is not true for any other engineering discipline.

Roadmap

- Course Overview
- What is Software Engineering?
- **The Iterative Development Lifecycle**
- Software Development Activities
- Methods and Methodologies

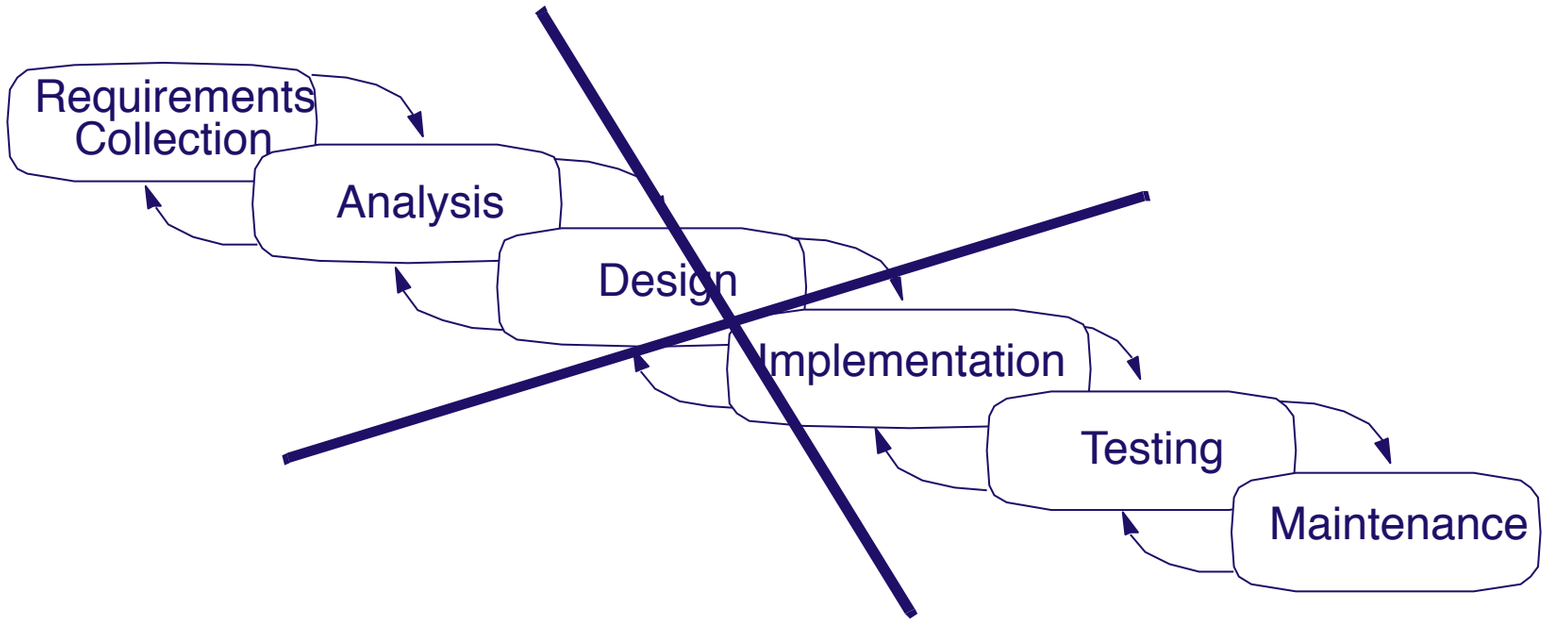
Software Development Activities

Requirements Collection	Establish customer's needs
Analysis	Model and specify the requirements ("what")
Design	Model and specify a solution ("how")
Implementation	Construct a solution in software
Testing	Validate the solution against the requirements
Maintenance	Repair defects and adapt the solution to new requirements

NB: these are ongoing activities, not sequential phases!

The Classical Software Lifecycle

The classical software lifecycle models the software development as a step-by-step "waterfall" between the various development phases.



The waterfall model is unrealistic for many reasons:

- requirements must be *frozen too early* in the life-cycle
- requirements are *validated too late*

The “waterfall model” has frequently been put forward as an ideal design process, consisting of several sequential phases leading to a final product. This model is known not to work in practice since it does not accommodate sufficient feedback between the phases.

Interestingly, the waterfall model was first formally described in a paper by Royce (1970) as an example of a naive process that does not work. Curiously it was later adopted by many organisations as an ideal model to strive for.

https://en.wikipedia.org/wiki/Waterfall_model

Problems with the Waterfall Lifecycle

1. “Real projects rarely follow the sequential flow that the model proposes. *Iteration* always occurs and creates problems in the application of the paradigm”

2. “It is often *difficult* for the customer *to state all requirements* explicitly. The classic life cycle requires this and has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.”

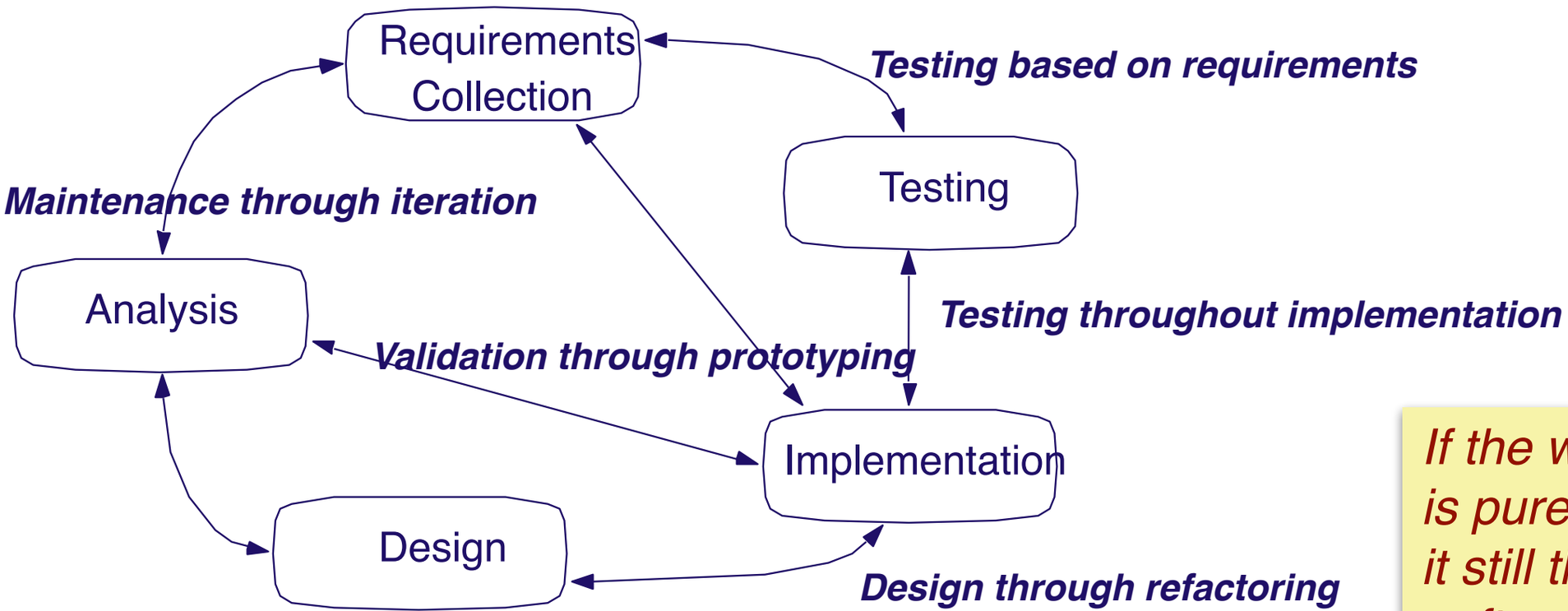
3. “The customer must have patience. A *working version* of the program(s) will not be available until *late in the project* timespan. A major blunder, if undetected until the working program is reviewed, can be disastrous.”

— *Pressman, SE, p. 26*

NB: Early prototyping helps to alleviate the 2nd and 3d problems.

Iterative Development

In practice, development is always iterative, and *all* activities progress in parallel.



If the waterfall model is pure fiction, why is it still the dominant software process?

New requirements can be introduced at any point in the project. Implementation of prototypes may start before detailed design to explore requirements. Testing typically starts as implementation starts.

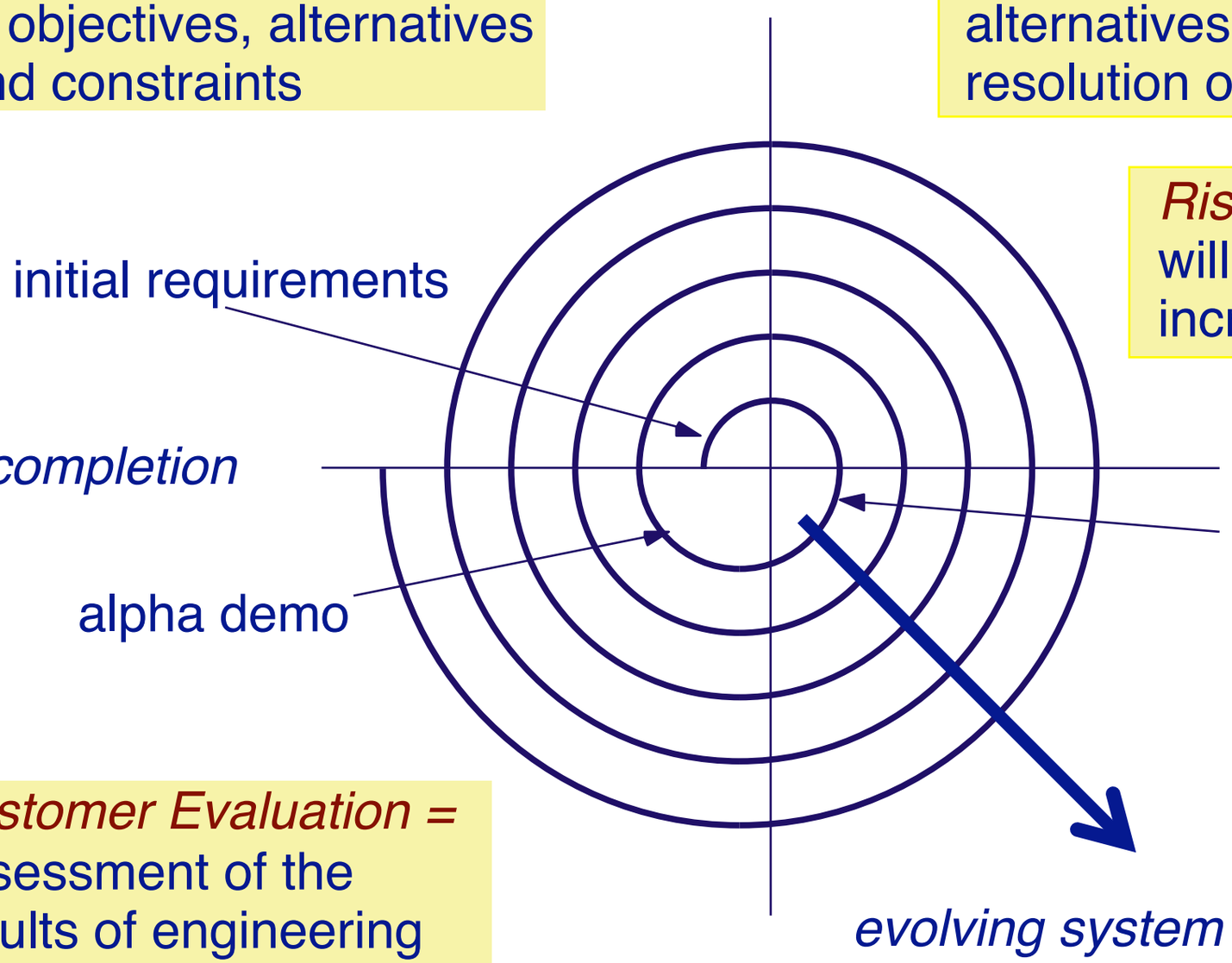
The reason waterfall is so popular is that managers like it: it gives the illusion that a project is proceeding according to a precise plan.

Boehm's Spiral Lifecycle

Planning = determination of objectives, alternatives and constraints

Risk Analysis = Analysis of alternatives and identification/ resolution of risks

Risk = something that will delay project or increase its cost



Customer Evaluation = Assessment of the results of engineering

Engineering = Development of the next level product

In reality, software projects typically follow an iterative and incremental *spiral* lifecycle. Each iteration consist of phases such as planning, risk assessment, engineering, and customer evaluation. With each iteration the system incrementally grows and evolves.

The Spiral Model was described by Barry Boehm in this 1988 paper:

http://scgresources.unibe.ch/Literature/ESE/Boeh88a-SpiralModel.pdf

(NB: these downloads are only accessible within the unite domain.)

Iterative Development

Plan to *iterate* your analysis, design and implementation.

— You won't get it right the first time, so *integrate, validate and test* as frequently as possible.

— "You should use iterative development only on projects that you want to succeed."

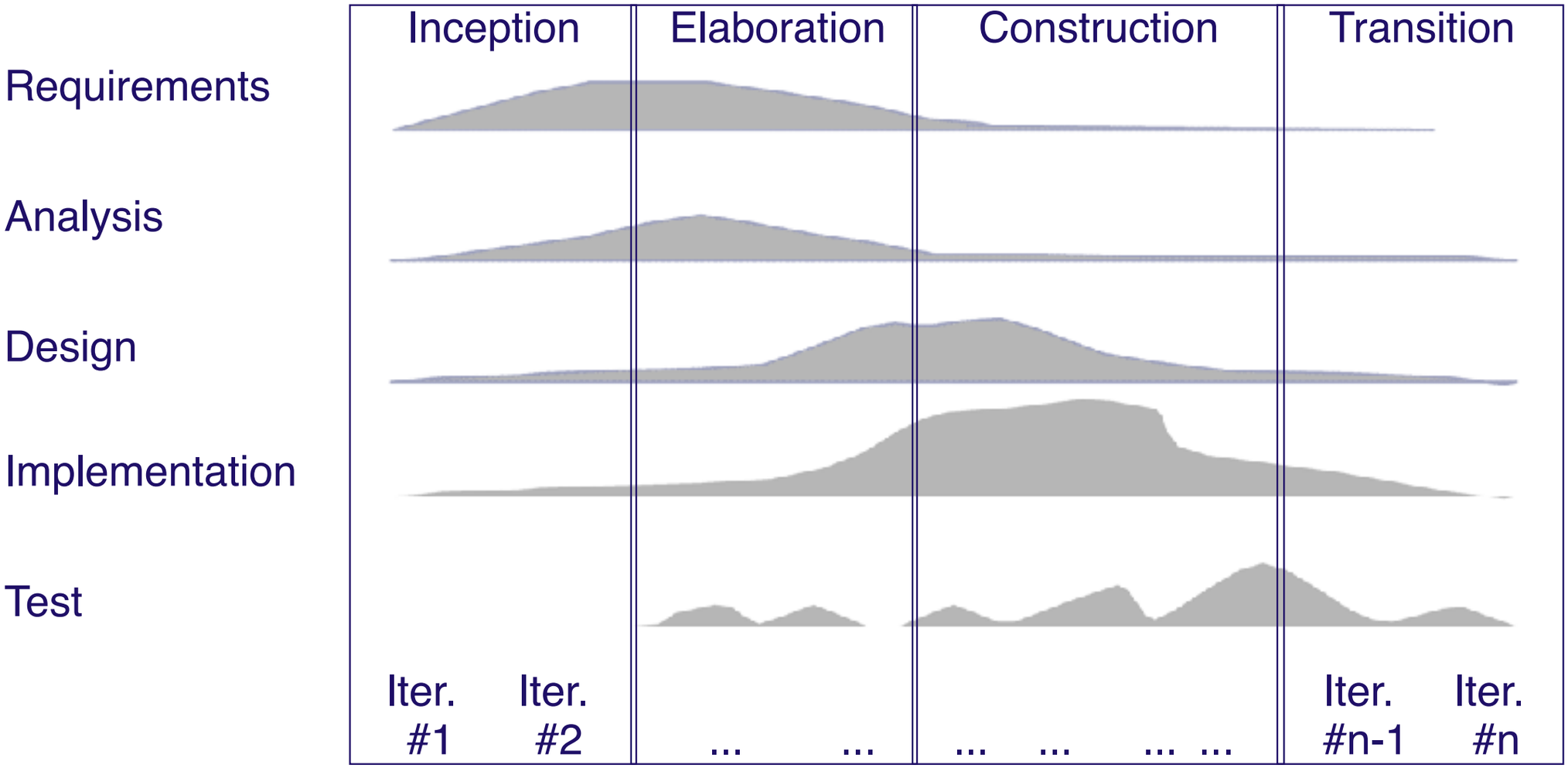
  – *Martin Fowler, UML Distilled (p 41)*

Incremental Development

Plan to *incrementally* develop (i.e., prototype) the system.

- If possible, always have a *running version of the system*, even if most functionality is yet to be implemented.
- *Integrate* new functionality as soon as possible.
- *Validate* incremental versions against user requirements.

The Unified Process



How do you plan the number of iterations?
How do you decide on completion?

The Rational Unified Process is a framework for an iterative development lifecycle proposed by Rational Software (the same company that first proposed the Unified Modeling Language).

The figure illustrates how the different SE activities (requirements collection etc.) continue to be carried out throughout the lifecycle of a project.

https://en.wikipedia.org/wiki/Rational_Unified_Process

Roadmap

- Course Overview
- What is Software Engineering?
- The Iterative Development Lifecycle
- **Software Development Activities**
- Methods and Methodologies

Requirements Collection

User requirements are often expressed *informally*:

—features

—usage scenarios

Although requirements may be documented in written form, they may be *incomplete*, *ambiguous*, or even *incorrect*.

It is rare for a SE project to start with pre-defined requirements. An important task is therefore to elicit the requirements from the customer. As we shall see, the concepts of *use cases* and *scenarios* are extremely useful to help bridge the gap between the users domain and the technical domain.

Changing requirements

Requirements *will* change!

—*inadequately captured* or expressed in the first place

—user and business *needs may change* during the project

Validation is needed *throughout* the software lifecycle, not only when the “final system” is delivered!

—build constant *feedback* into your project plan

—plan for *change*

—early *prototyping* [e.g., UI] can help clarify requirements

Requirements Analysis and Specification

Analysis is the process of specifying *what* a system will do.

—The intention is to provide a clear understanding of what the system is about and what its underlying concepts are.

The result of analysis is a *specification document*.

Does the requirements specification correspond to the users' actual needs?

Object-Oriented Analysis

An *object-oriented analysis* results in models of the system which describe:

- *classes* of objects that exist in the system
  - —*responsibilities* of those classes
- *relationships* between those classes
- *use cases* and *scenarios* describing
  - —*operations* that can be performed on the system
  - —allowable *sequences* of those operations

In this course we will focus on *object-oriented software engineering methods*. OO methods have proved to be extremely successful in the construction of large, complex software systems. One reason for this success is that OO offers developers various tools to *model domain concepts in code*, thus making it easier to bridge the gap between the application and the technical domains.

Prototyping (I)

A *prototype* is a software program developed to test, explore or validate a hypothesis, i.e. *to reduce risks.*

An *exploratory prototype*, also known as a *throwaway prototype*, is intended to *validate requirements* or *explore design choices*.

- —UI prototype — validate user requirements
- —rapid prototype — validate functional requirements
- —experimental prototype — validate technical feasibility

Prototyping (II)

An *evolutionary prototype* is intended to evolve in steps into a finished product.

> iteratively "grow" the application, *redesigning* and *refactoring* along the way

First do it,
then do it right,
then do it fast.

Design

Design is the process of specifying *how* the specified system behaviour will be realized from software components. The results are *architecture* and *detailed design documents*.

Object-oriented design delivers models that describe:

—how system operations are implemented by *interacting objects*

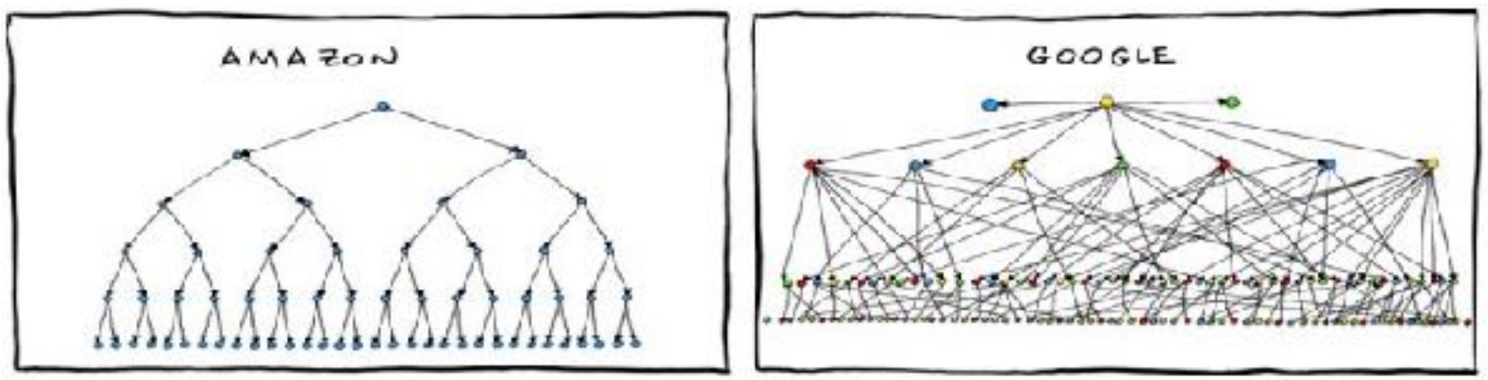
—how classes refer to one another and how they are related by *inheritance*

—*attributes* and *operations* associated to classes

Design is an iterative process, proceeding in parallel with implementation!

Conway's Law

"Organizations that design systems are constrained to produce designs that are copies of the communication structures of these organizations"



Famously IBM in the 1960s was organized as a many-layered hierarchy, and the software systems it produced also were designed as many-layered hierarchies.

If your software team splits responsibilities in a certain way, you can expect that the software will also be structured in the same way.

“Conway’s Law” was described in this 1967 paper:

http://www.melconway.com/Home/Committees_Paper.html

Graphic from:

http://www.slideshare.net/RachelDavies/tech-talk-implicationsconwayslaw

Implementation and Testing

Implementation is the activity of *constructing* a software solution to the customer's requirements.

Testing is the process of *validating* that the solution meets the requirements.

—The result of implementation and testing is a *fully documented* and *validated* solution.

Design, Implementation and Testing

Design, implementation and testing are iterative activities

—The implementation does not "implement the design", but rather the design document *documents the implementation*!

> System tests reflect the requirements specification

> Testing and implementation go hand-in-hand

—Ideally, test case specification *precedes* design and implementation

Maintenance

Maintenance is the process of changing a system after it has been deployed.

- *Corrective maintenance*: identifying and repairing *defects*
- *Adaptive maintenance*: *adapting* the existing solution to new platforms
- *Perfective maintenance*: implementing *new requirements*

In a spiral lifecycle, everything after the delivery and deployment of the first prototype can be considered "maintenance"!

“Maintenance” is actually a terrible word to describe what happens after deployment. Is suggests that at a certain point a project is “done”, and then the only changes are bug fixes and patches.

Maintenance activities

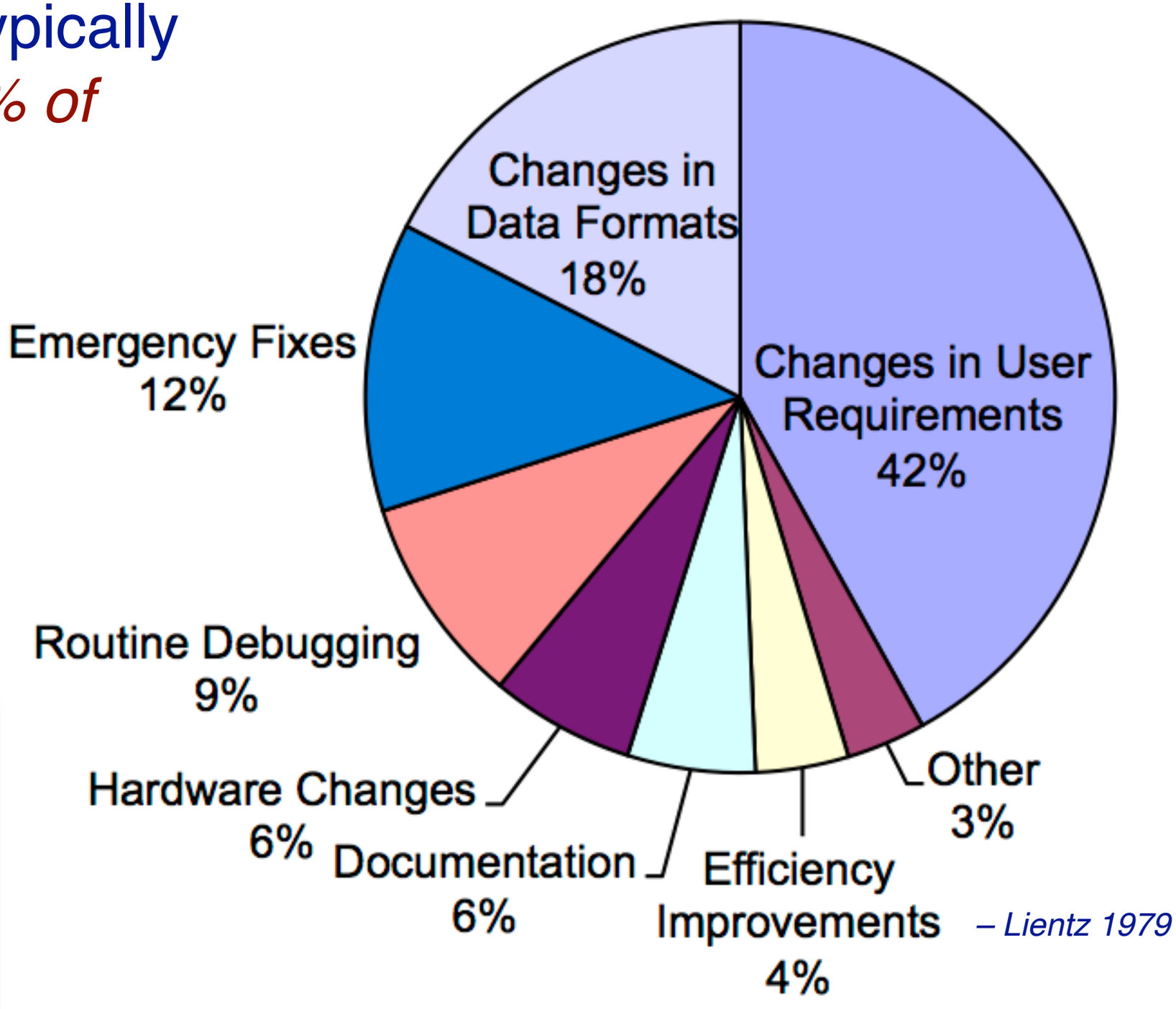
"Maintenance" entails:

- configuration and version management
- reengineering (redesigning and refactoring)
- updating all analysis, design and user documentation

Repeatable, automated tests enable evolution and refactoring

Maintenance costs

"Maintenance" typically accounts for *70% of software costs!*



– Lientz 1979

Means: most project costs concern continued development *after* deployment

So, in real projects most of the real development occurs after first deployment. In other words, in most projects “maintenance” is actually “continuous development”.

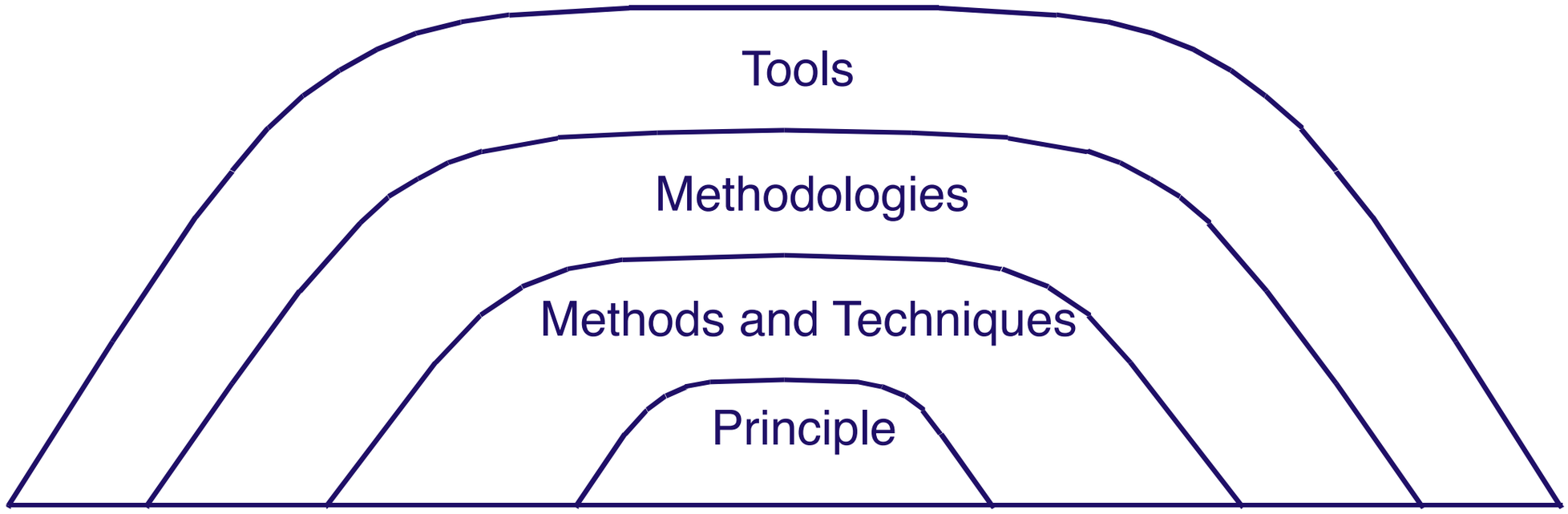
This is perhaps one of the most important points that distinguishes “Software Engineering” from “programming”. SE must cope with the fact that real software projects are long-lived, and they evolve considerable over their lifetime.

Roadmap

- Course Overview
- What is Software Engineering?
- The Iterative Development Lifecycle
- Software Development Activities
- **Methods and Methodologies**

Methods and Methodologies

Principle = general statement describing desirable properties
Method = general guidelines governing some activity
Technique = more technical and mechanical than method
Methodology = package of methods and techniques packaged



— *Ghezzi et al. 1991*

Object-Oriented Methods: a brief history

- **First generation:**
  - Adaptation of existing notations (ER diagrams, state diagrams ...): Booch, OMT, Shlaer and Mellor, ...
  - Specialized design techniques:
    - *CRC cards; responsibility-driven design; design by contract*
- **Second generation:**
  - Fusion: Booch + OMT + CRC + formal methods
- **Third generation:**
  - Unified Modeling Language:
    - *uniform notation: Booch + OMT + Use Cases + ...*
    - *various UML-based methods (e.g. Catalysis)*
- **Agile methods:**
  - Extreme Programming
  - Test-Driven Development
  - Scrum …

What you should know!

- How does Software Engineering differ from programming?
- Why is the “waterfall” model unrealistic?
- What is the difference between analysis and design?
- Why plan to iterate? Why develop incrementally?
- Why is programming only a small part of the cost of a “real” software project?
- What are the key advantages and disadvantages of object-oriented methods?

Can you answer these questions?

- What is the appeal of the “waterfall” model?
- Why do requirements change?
- How can you validate that an analysis model captures users’ real needs?
- When does analysis stop and design start?
- When can implementation start?
- What are good examples of Conway’s Law in action?