Praktikum in Software Engineering

Prof. O. Nierstrasz

Sommersemester 2000

1. Praktikum — Software Engineering

Lecturer: Prof. Oscar Nierstrasz

Office: Schützenmattstr. 14/103
Tel. 631.4618
Email: Oscar.Nierstrasz@iam.unibe.ch

Assistants: Matthias Rieger, Mathis Kretz, Nicolas Wrobel

WWW: `www.iam.unibe.ch/~scg/Teaching/PSE`

- Requirements documents
- PSE wiki (electronic bulletin board)
- Entry point to team pages
- Pointers to technical documentation

Overview

- Goals of this workshop
- Project overview
- Schedule: *milestones, deliverables*
- Analysis and Design documents: *guidelines*
- Prototyping: *requirements validation and iterative development*
- Testing: *coverage and regression tests*
- Teamwork: *roles and responsibilities*
- Tools: *UML, rcs, make, SNiFF+, ...*

Goals of this Workshop ...

Methodological skills

- Practising Requirements Collection and Specification
- Practising Responsibility-Driven Design
- Evaluating Implementation Strategies
- Prototyping

Practical skills

- Working with open requirements (setting scope ...)
- Developing a complete product (documentation, installation ...)
- Teamwork (division of labour, planning, collaboration ...)

Technical skills

- UML
- Programming tools
- Testing

The Customer — AIESEC

www.cx.unibe.ch/aiesec:

Offers training and international exchange opportunities for students
Liaison with business

Organizes yearly Kontaktgespräch for graduating diploma students (May 25, 2000)

- Students fill out (paper) forms with profile
- AIESEC forward profiles to relevant companies
- Companies get back in touch with students

Want to replace paper form system with an *on-line job fair*

Project Characteristics

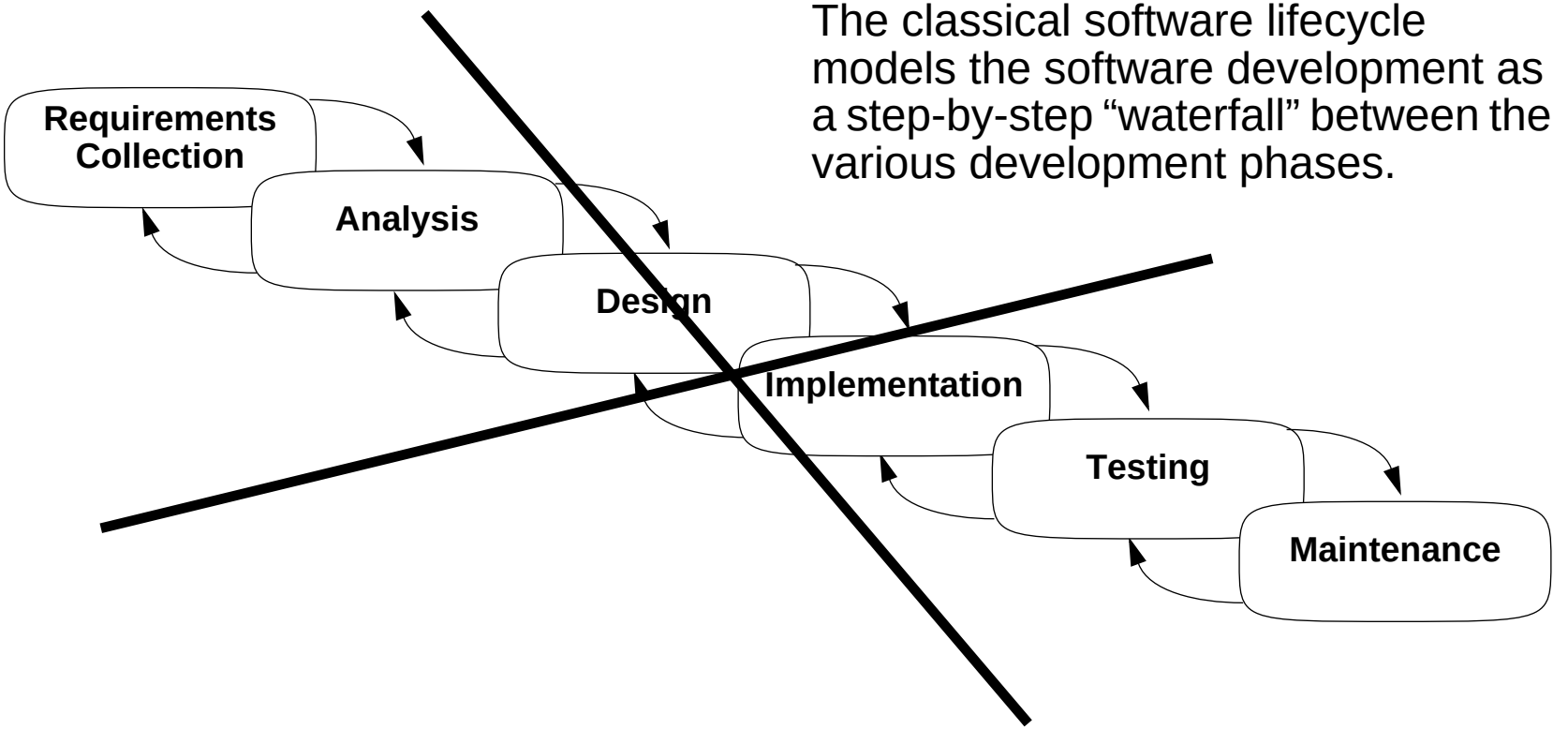
Several characteristics of “real” projects:

- Open, changing requirements
- Replace an existing, manual system with an automatic solution
- Web based interface
- Persistency must be provided by a relational DB
- Solution will be built using standard packages
- Project lifetime extends beyond the end of the course!

Non-issues:

- No legacy data
- No integration with existing applications or legacy software

The Classical Software Lifecycle



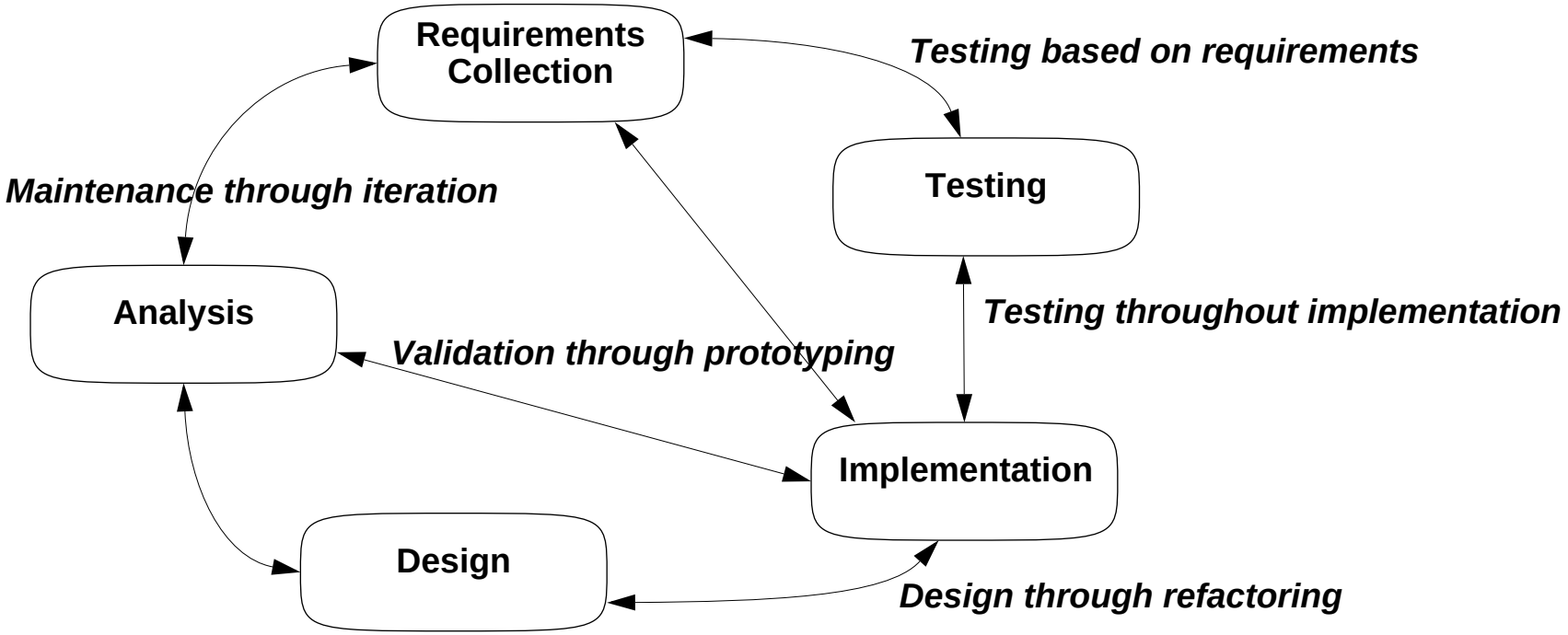
The classical software lifecycle models the software development as a step-by-step “waterfall” between the various development phases.

The waterfall model is unrealistic for many reasons, especially:

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

Iterative Development

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



Schedule

		Meeting / Deliverable	*Homework/Consultation*
1	2000-03-29	Introduction; Presentation of client and project; Team forming	Study project descriptions; prepare draft requirements specification; prepare interview questions; study software doc
2	04-05	Requirements interview with AIESEC	Specify E-R diagram; define workplan; start developing experimental prototypes (architecture, other risks)
3	04-12	Deliver E-R diagram; deliver overall workplan;	Specify requirements; implement prototypes
4	04-19	Present and exchange requirements spec; deliver component development proposal	Review requirement specs (other team); define DB schema
5	04-26	Present requirement spec review; deliver DB spec	Specify website structure; specify coarse design (system architecture), implement DB spec

		Meeting / Deliverable	*Homework/Consultation*
6	05-03	Present website spec; exchange architecture spec	Review architecture spec (other team); specify detailed design; start implementation and testing
7	05-10	Present architecture review; deliver detailed design spec	Implementation and testing ...
8	05-17	Lab Session: Demo of first product	Quality review (other team's product)
9	05-24	*To be defined* ...	Product revision ...
10	05-31	*To be defined* ...	Review code (other team) (planned)
11	06-07	Present code review (planned)	Product revision ...
12	06-14	Final product delivery (planned)	Review final product (other team)
13	06-21	*To be defined* ...	Revise final product
14	06-28	Deliver revised final product; Feedback and Testat	

Shaded rows mark sessions which will be attended by the client.
This schedule will be revised as the workshop progresses ...

Evaluation

Every Team must:

- provide deliverables of acceptable quality
  - all deliverables will be reviewed by another team (we will review the reviewers!)
  - unacceptable deliverables must be revised

Every Team Member must:

- assume responsibility for and present at least one deliverable
- contribute “fairly” to the team effort

Deliverables

Group web pages

- each group will have a group account for development
- all deliverables (documentation, demos and source code) must be accessible from the group's web page
- keep deliverables up-to-date as the project progresses
  - ☞ each version and revision of a deliverable must be accessible
- every deliverable will be reviewed by another team
  - ☞ write documentation and code to be read by others!

Log your activities

- Keep minutes of all meetings
  - ☞ Date & time; participants
  - ☞ Decisions and actions *with deadlines*

- Log all effort
  - ☞ Use simple metrics
  - – I spent 80 minutes debugging 15 short methods
  - – We spent 120 minutes filling out 12 CRC cards

- Estimate cost of each task
  - ☞ Iteratively improve your estimates
  - – We'll need about 8 CRC cards, so we'll probably need 80 minutes

Email all logs to designated assistant.

Workplans

Planning and Cost Estimation

- prepare a workplan with delivery times and costs for each deliverable
- break down each task into subtasks whose cost you can estimate
- detailed cost estimates must be made by the team member who accepts responsibility for it
- revise and refine your plan as the project progresses
- keep precise logs of how much time you *actually spend* on every subtask
  - ☞ try to improve your estimates as the project progresses

  NB: goal is to improve your estimates, not to evaluate productivity!

Use conventional means to define workplans (e.g., Gantt charts)

Requirements Collection and Analysis

Requirements Specification

- Describe *what* is required, not *how* it will be implemented — don't confuse requirements specification and design!
- Formalize scenarios, domain objects, functional and non-functional requirements
  - Use UML to formalize your models; use text to explain them
  - Specify enough detail so that *someone else* could design a solution
- Identify the risks and trade-offs
  - what open questions must be answered before you can start implementing a solution? (what prototyping is needed?)
- Identify priorities
  - what are the minimal requirements for a first product?
- Keep specification up-to-date as requirements are refined

Prototyping

Prototyping is an essential activity carried out during *all phases* of the software process.

Requirements validation

- Prototype a user interface as early as possible to validate your requirements specification.

Evaluating design decisions

- Prototype parts of your design to evaluate feasibility and usability of technical alternatives.
  - ☞ prototype to reduce risks!

Iterative development

- Integrate parts as early as possible to always have a running prototype of the target application that can be tested and demoed.

Design

Architecture

- Choose a simple architecture that can cope with all known requirements
  - what are the principal parts of the system and how do they communicate?
- Architecture will be heavily influenced by framework that is used
- Develop prototypes to test the architecture

Detailed design and implementation

- Iteratively apply responsibility-driven design
- Evaluate technical alternatives and document design decisions
  - keep it simple; add complexity only when necessary
  - prototype when trade-offs are unclear
- Refactor the design as the implementation evolves

Testing

Coverage

- Design tests that will exercise all required/implemented functionality
  - every time you add a feature, write a test for that feature!
- Check that all possible execution paths are tested
  - Apply both black-box and white-box testing

Regression

- Automate testing so that all tests can be carried out after any system change
- Set up tests so they can run in either
  - “verbose” mode (i.e., logging every interesting event), or in
  - “silent” mode (i.e., only reporting when and where tests have failed)

Component Development

Consider developing components wherever you identify a generally useful abstraction.

- Well-design components will be easier to adapt and reuse when the application evolves.
- A component may be either a piece of software (class, package) of a final application or a specialized tool to help in the development or testing of the application.

Consider marketing components to other teams:

- A team can deliver a (substantial) component set instead of a complete project if it finds *at least two* other client teams (subject to approval of the course instructors!)
- A component provider must supply (according to contract):
  - complete software with documentation
  - test drivers
  - maintenance support

Tools

Use (at least) the following tools (or equivalent ones)!

UML	Use UML to document all your models (esp. requirements specification and design).
SNiFF+	SNiFF+ integrates software development tools (compilers, version management etc.) and provides support for development of software in teams.
rcs	Use version control for all text documents (i.e., both source code and documentation).
make	Use make to automate compilation, installation, testing and cleanup.
javadoc	Automate generation of HTML documentation from source code.

Many other tools are available — use them!

If you aren't sure what tool you should use to get a job done, just ask!

Teamwork

Guidelines

- Break down and distribute work incrementally
  - always estimate cost when you accept a task
  - check and revise workplans and estimates as work progresses
- Use the team to your advantage
  - use role-playing with CRC cards to elaborate the design
  - distribute responsibilities according to skills
  - someone else should test your code
  - all code and documentation should be reviewed by someone else
- Program in pairs
  - code review as you program to increase quality

Roles and Responsibilities

Roles may be *fixed* or *floating*, but must always be assigned to some team member.

- Individual tasks should be *distributed* amongst team members

Sample roles and responsibilities

- Project Administrator
- Chief programmer/architect
- Backup programmer
- Tester/test case developer
- Toolsmith
- Component librarian
- Documentation editor

The *team* is responsible for all the deliverables.
Individuals assume responsibilities for specific subtasks.

Supporting roles

Customer

- answer questions about requirements ☞ *use the wiki!*
- accept/reject requirements specs
- evaluate prototypes, final system

System support

- system administration
- maintain installation of required software
- (limited) help for technical problems ☞ *use the wiki!*

Consultants

- meet regularly (at least twice weekly) with their teams
- oversee quality of work
- give advice concerning software process, technical solutions etc.
- crisis detection; trouble-shooting

Forming Teams

1. Identify your skills: *strong and weak points*
   - ☞ What skills would complement your own?

2. Round table: *20 seconds to present yourself*
   - ☞ What do you have to offer; who are you looking for?

3. Form teams of six: *look for suitable partners*
   - ☞ Seek complementary skills that cover responsibilities

4. Prepare your strategy and tactics:
   - ☞ What questions do you need to ask of the client?
   - ☞ What interactions do you anticipate with other teams?