Softwaretechnik / Software-Engineering

Lecture 1: Introduction

2018-04-16

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany

Content

- Software, Engineering, Software Engineering
- Successful Software Development
  - working definition: success
  - unsuccessful software development exists
  - common reasons for non-success
- Course
  - Content
    - topic areas
    - structure of topic areas
    - emphasis: formal methods
    - relation to other courses
    - literature
  - Organisation
    - lectures
    - tutorials
    - exam
Software, Engineering, Software Engineering

Softwaretechnik / Software-Engineering

Lecture 1: Introduction

2018-04-16

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany
Software – Computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system.

See also: application software; support software; system software.

Contrast with: hardware.

IEEE 610.12 (1990)

Software –

1. all or part of the programs, procedures, rules, and associated documentation of an information processing system. […]
2. see 610.12
3. program or set of programs used to run a computer. […]

NOTE: includes firmware, documentation, data, and execution control statements.

IEEE 24765 (2010)
**Engineering vs. Non-Engineering**

<table>
<thead>
<tr>
<th></th>
<th>Workshop (technical product)</th>
<th>Studio (artwork)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mental prerequisite</strong></td>
<td>the existing and available technical know-how</td>
<td>artist's inspiration, among others</td>
</tr>
<tr>
<td><strong>Deadlines</strong></td>
<td>can usually be planned with sufficient precision</td>
<td>cannot be planned due to dependency on artist's inspiration</td>
</tr>
<tr>
<td><strong>Price</strong></td>
<td>oriented on cost, thus calculable</td>
<td>determined by market value, not by cost</td>
</tr>
<tr>
<td><strong>Norms and standards</strong></td>
<td>exist, are known, and are usually respected</td>
<td>are rare and, if known, not respected</td>
</tr>
<tr>
<td><strong>Evaluation and comparison</strong></td>
<td>can be conducted using objective, quantified criteria</td>
<td>is only possible subjectively, results are disputed</td>
</tr>
<tr>
<td><strong>Author</strong></td>
<td>remains anonymous, often lacks emotional ties to the product</td>
<td>considers the artwork as part of him/herself</td>
</tr>
<tr>
<td><strong>Warranty and liability</strong></td>
<td>are clearly regulated, cannot be excluded</td>
<td>are not defined and in practice hardly enforceable</td>
</tr>
</tbody>
</table>

(Ludewig and Lichter, 2013)
Software Engineering

(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

(2) The study of approaches as in (1).

IEEE 610.12 (1990)

Software Engineering –

1. the systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software.

2. see IEEE 610.12 (1)


Software Engineering –

Multi-person Development of Multi-version Programs.

D. L. Parnas (2011)

Software Engineering – the establishment and use of sound engineering principles to obtain economically software that is reliable and works efficiently on real machines.

F. L. Bauer (1971)
Software Engineering – (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1). IEEE 610.12 (1990)

Software Engineering – the systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software. ISO/IEC/IEEE 24764 (2010)


Software Engineering – the establishment and use of sound engineering principles to obtain economically software that is reliable and works efficiently on real machines. F. L. Bauer (1971)

Software Engineering in the Academy

Bertrand Meyer
Interactive Software Engineering

Institutions that teach software are responsible for producing professionals who will build and maintain systems to the satisfaction of their beneficiaries. This article presents some ideas on how best to honor this responsibility.

Software

here is no universally accepted definition of software engineering.

Institutions that teach software are responsible for producing professionals who will build and maintain systems to the satisfaction of their beneficiaries. This article presents some ideas on how best to honor this responsibility.

I won’t settle on any of these definitions; rather, I’d like to accept that they are all in some way valid and retain all the views of software they encompass. In April 2000 IEEE 610.12 was made obsolete in favor of ISO/IEC/IEEE 24764, which in turn will be discarded in favor of a new standard yet to be defined. The landscape of software engineering is rapidly changing. I declare my ignorance on the subject. There is no generally accepted definition of software engineering.

There is no universally accepted definition of software engineering. For some, software engineering is just a glorified name for programming. If you are a programmer, you might put “software engineer” on your business card but never “programmer.” Others have higher expectations. A textbook definition of the term might read something like this: “the body of methods, tools, and techniques intended to produce quality software.”

Rather than just emphasizing quality, we could distinguish software engineering from programming by its industrial nature, leading to another definition: “the development of possibly large systems intended for use in production environments, with possibly many changes, over possibly many years, and worked on by possibly many people, and providing a management and maintenance model.”

David Parnas, a pioneer in the field, emphasizes the “engineering” part and advocates a software engineering education based on traditional engineering and rational thinking. Even more generally on how to instill software engineering concerns into an entire software curriculum.

Softwar Professionals

Software professionals are responsible for producing software professionals who will build and maintain systems to the satisfaction of their beneficiaries. This article presents some ideas on how best to honor this responsibility.

SOFTWARE PROFESSIONALS

Judging by the employment situation, current and future graduates can be happy with their studies. The Information Technology Association of America estimated in April 2000 that 850,000 IT jobs would go unfilled in the next 12 months. The dearness of qualified personnel is just as perceptible in Europe and Australia. Salaries are excellent. Project leaders wake up at night worrying about headhunters luring away some of their best developers or producing the same dilemma internally.

I won’t settle on any of these definitions; rather, I’d like to accept that they are all in some way valid and retain all the views of software they encompass. In April 2000 IEEE 610.12 was made obsolete in favor of ISO/IEC/IEEE 24764, which in turn will be discarded in favor of a new standard yet to be defined. The landscape of software engineering is rapidly changing. I declare my ignorance on the subject. There is no generally accepted definition of software engineering.
Software Engineering –
(1) The application of a systematic, disciplined, quantifiable approach to the
development, operation, and maintenance of software; that is, the applica-
tion of engineering to software.
(2) The study of approaches as in (1).

IEEE 610.12 (1990)

Software Engineering – the establishment and use of sound engineering
principles to obtain economically software that is reliable and works effi-
ciently on real machines.

F. L. Bauer (1971)

Successful Software Development
A software development project is **successful** if and only if developer, customer, and user are happy with the result at the end of the project.
### A Closer Look

**Successful:**

**Time $t$:**
- Customer (software) contract
- Developer (software) contract

**Time $t' > t$:**
- Developer (software) delivery
- Customer (software) delivery

**Unsuccessful:**

**Time $t$:**
- Customer (software) contract
- Developer (software) contract

**Time $t' > t$:**
- Developer (software) delivery
- Customer (software) delivery

What might’ve gone wrong?
In Other Words

All engineering disciplines face the same questions:

- How to describe requirements / avoid misunderstandings with the customer?
- How to describe design ideas / avoid misunderstandings with the implementers?
- How to ensure that the product is built right / that the right product is built?
  (→ How to measure the quality of the product?)
- How to schedule activities properly?

At best: are there procedures which promise to systematically avoid certain mistakes or costs?

This course is about Software Engineering, so we should discuss:

- How to describe requirements on software precisely?
- How to describe design ideas for software precisely?
- How to ensure that software is built right?
  (→ How to measure the quality of software?)
- How to schedule software development activities properly?

What are procedures to systematically avoid certain mistakes or costs in software development?
Example: Nightly Builds

Scenario:
- Program $P$ compiles successfully at time $t$.
- Programmers work for duration $d$ on $P$, yielding program $P'$ at time $t + d$.
- $P'$ does not compile at time $t + d$.

→ The reason for not compiling any more must be among the changes during $d$.

Experience:
- If $d$ is large, it can be very difficult (and time consuming) to identify the cause.

Proposal: “Nightly Builds”
- Set up a procedure, which (at best: automatically) tries to compile the current state of the development each day over night.
- Promise: with “nightly builds”, $d$ is effectively limited to be smaller or equal to one day, so the number of possible causes for not compiling should be manageable.

→ Software Engineering as a defensive discipline (measures against failures and “catastrophes”):
  - if program $P$ always compiles, the effort for “nightly builds” was strictly speaking wasted.
  - if a compilation issue occurs during the project, the caused damage is bounded.

Same holds for documentation: if no maintenance is ever needed, documentation effort may be wasted.

In Other Words

All engineering disciplines face the same questions:
- How to describe requirements / avoid misunderstandings with the customer?
- How to describe design ideas / avoid misunderstandings with the implementers?
- How to ensure that the product is built right / that the right product is built?
  (→ How to measure the quality of the product?)
- How to schedule activities properly?

At best: are there procedures which promise to systematically avoid certain mistakes or costs?

This course is about Software Engineering, so we should discuss:
- How to describe requirements on software precisely?
- How to describe design ideas for software precisely?
- How to ensure that software is built right?
  (→ How to measure the quality of software?)
- How to schedule software development activities properly?

What are procedures to systematically avoid certain mistakes or costs in software development?

Software Engineering is a young discipline: plenty of proposals for each question.
So the course will focus on the problems and discuss example proposals.
So the course will focus on the problems and discuss example proposals.
Course Content (Tentative)

| Introduction                  | L 1: 16.4, Mon          |
| Scales, Metrics, Costs,       | L 2: 19.4, Thu          |
| Development Process           | L 3: 23.4, Mon          |
| Requirements                  | L 4: 26.4, Thu          |
| Engineering                   | T 1: 30.4, Mon          |
|                              | L 5: 3.5, Thu           |
|                              | L 6: 7.5, Mon           |
|                              | L 7: 10.5, Thu          |
|                              | L 8: 14.5, Mon          |
|                              | T 2: 17.5, Thu          |
|                              | - 21.5, Mon             |
|                              | - 24.5, Thu             |
|                              | L 9: 28.5, Mon          |
|                              | L 10: 31.5, Thu         |
|                              | L 11: 4.6, Mon          |
|                              | L 12: 7.6, Thu          |
|                              | T 3: 11.6, Mon          |
| Arch. & Design, Software      | L 13: 14.6, Thu         |
|                              | L 14: 18.6, Mon         |
|                              | T 4: 21.6, Thu          |
| Modelling, Patterns, QA      | L 15: 25.6, Mon         |
|                              | L 16: 28.6, Thu         |
|                              | L 17: 2.7, Mon          |
|                              | T 5: 5.7, Thu           |
| (Testing, Formal Verification)| T 6: 9.7, Mon           |
| Wrap-Up                       | L 18: 12.7, Thu         |
|                              | L 19: 16.7, Mon         |
|                              | T 6: 19.7, Thu          |

Structure of Topic Areas

**Example:** Requirements Engineering

Vocabulary

- e.g. consistent, complete, tacit, etc.

Techniques

- informal
- semi-formal
- formal
**Excursion: Informal vs. Formal Techniques**

**Example:** Requirements Engineering, Airbag Controller

**Requirement:**

Whenever a crash is detected, the airbag has to be fired within 300 ms ($\pm \varepsilon$).

```
within means
≤ so 100 ms is okay, too
```

**Developer A**

within means between $300 - \varepsilon$ and $300 + \varepsilon$

---

**vs.**

- Fix observables: 
  - `crashdetected`: Time → {0, 1} and `fireairbag`: Time → {0, 1}
- Formalise requirement:
  ```
  \forall t, t' \in \text{Time} \left( \text{crashdetected}(t) \land \text{fireairbag}(t') \Rightarrow t' \in [t + 300 - \varepsilon, t + 300 + \varepsilon] \right)
  ```

→ no more misunderstandings, sometimes tools can objectively decide requirement satisfied yes/no.

---

**Sign says:**

No more misunderstandings; Sometimes tools can **objectively** decide things; → Karibu formal methods paradise.
**Structure of Topic Areas**

**Example: Requirements Engineering**

**Vocabulary**
- e.g. consistent, complete, tacit, etc.

**Techniques**
- informal
- semi-formal
- formal

- e.g. "Whenever a crash..."
- e.g. "Always, if ⟨crash⟩ at t..."
- e.g. "∀ t, t' ∈ Time • ..."

**In the course:**
- Use Cases
- Pattern Language
- Decision Tables
- Live Sequence Charts

**Course Content (Tentative)**

- **Introduction**
  - L 1: 16.4, Mon
- **Scales, Metrics, Costs,**
  - L 2: 19.4, Thu
  - L 3: 23.4, Mon
  - T 1: 26.4, Thu
- **Development Process Requirements**
  - L 4: 30.4, Mon
  - L 5: 3.5, Thu
  - L 6: 7.5, Mon
  - T 2: 10.5, Thu
- **Engineering**
  - L 7: 14.5, Mon
  - T 2: 17.5, Thu
  - T 3: 21.5, Mon
  - T 4: 24.5, Thu
- **Software Project Management**
  - L 8: 28.5, Mon
  - L 9: 31.5, Thu
  - L 10: 7.6, Mon
  - L 11: 14.6, Thu
  - L 12: 18.6, Thu
  - T 4: 21.6, Thu
- **Arch. & Design, Software**
  - L 13: 12.7, Thu
  - L 14: 17.7, Thu
  - L 15: 19.7, Thu
  - L 16: 25.7, Mon
  - L 17: 29.7, Mon
  - T 5: 31.7, Thu
  - L 18: 5.7, Mon
  - L 19: 9.7, Mon
  - T 6: 12.7, Thu
  - T 7: 16.7, Mon
  - T 8: 19.7, Thu
Content

- Software Engineering, Software Engineering
- Successful Software Development
  - working definition: success
  - unsuccessful software development exists
  - common reasons for non-success

Course

- Content
  - topic areas
  - structure of topic areas
  - emphasis: formal methods
  - relation to other courses
  - literature

- Organisation
  - lectures
  - tutorials
  - exam

Course Software-Engineering vs. Other Courses
**Agreement** between ‘Fachschaft’ and the chair for software engineering: strong(er) coupling between both courses.

---

**Content**

- **Software, Engineering, Software Engineering**
- **Successful Software Development**
  - working definition: success
  - unsuccessful software development exists
  - common reasons for non-success
- **Course**
  - **Content**
    - topic areas
    - structure of topic areas
    - emphasis: formal methods
    - relation to other courses
    - literature
  - **Organisation**
    - lectures
    - tutorials
    - exam
Any Questions So Far?

…more on the course homepage.
Course: Organisation
Organisation: Lectures

- **Homepage:** [http://svt.informatik.uni-freiburg.de/teaching/SS2018/swtv1](http://svt.informatik.uni-freiburg.de/teaching/SS2018/swtv1)
- **Course language:** English (since we are in an even year)
- **Script/Media:**
  - slides without annotations on homepage with beginning of lecture the latest
  - slides with annotations on homepage typically soon after the lecture
  - recording on ILIAS (stream and download) with max. 2 days delay (cf. link on homepage)
- **Schedule:** topic areas à three 90 min. lectures, one 90 min. tutorial (with exceptions)
- **Interaction:** absence often moaned; but it takes two, so please ask/comment immediately.
- **Questions/comments:**
  - "online": ask immediately or in the break
  - "offline":
    1. try to solve yourself
    2. discuss with colleagues
  - a) Exercises: ILIAS (group) forum, contact tutor
  - b) Everything else: contact lecturer (cf. homepage)
    or just drop by: Building 52, Room 00-020

- **Break:** we'll have a 5-10 min. break
  in the middle of each lecture (from now on), unless a majority objects now.

### Schedule/Submission:
- exercises online [homepage] and ILIAS with first lecture of a block.
- early submission 24h before tutorial...
  (usually Wednesday, 12:00, local time).
- regular submission right before tutorial
  (usually Thursday, 12:00, local time).

- please submit electronically via ILIAS; paper submissions are tolerated
- should work in teams of 2-3 people, clearly give names on submission

### Grading system: “most complicated grading system ever”
- Admission points (good-will rating, upper bound)
  ("reasonable grading given student's knowledge before tutorial")
- Exam-like points (evil rating, lower bound)
  ("reasonable grading given student's knowledge after tutorial")

10% bonus for early submission.

### Tutorial: Four groups (central assignment), hosted by tutor.
- Starting from discussion of the early submissions (anonymous).
  develop one good proposal together,
- tutorial notes provided via ILIAS.

Organisation: Exercises & Tutorials

### Schedule/Submission:
- exercises online [homepage] and ILIAS with first lecture of a block.
- early submission 24h before tutorial...
  (usually Wednesday, 12:00, local time).
- regular submission right before tutorial
  (usually Thursday, 12:00, local time).

- please submit electronically via ILIAS; paper submissions are tolerated
- should work in teams of 2-3 people, clearly give names on submission

### Grading system: “most complicated grading system ever”
- Admission points (good-will rating, upper bound)
  ("reasonable grading given student's knowledge before tutorial")
- Exam-like points (evil rating, lower bound)
  ("reasonable grading given student's knowledge after tutorial")

10% bonus for early submission.

### Tutorial: Four groups (central assignment), hosted by tutor.
- Starting from discussion of the early submissions (anonymous).
  develop one good proposal together,
- tutorial notes provided via ILIAS.
Organisation: Exam

- Exam Admission:

  Achieving 50% of the regular admission points of Exercise Sheets 0–3 and 50% of Exercise Sheets 4–6 is sufficient for admission to exam.

  10 regular admission points on sheets 0 and 1, and 20 regular admission points on exercise sheets 2–6 → 120 regular admission points for 100%.

  (plus plenty of admission bonus points in both blocks, O–3 and 4–6)

- Exam Form:
  - written exam
  - date, time, place: tba
  - permitted exam aids: one A4 paper (max. 21 x 29.7 x 1 mm) of notes, max. two sides inscribed
  - scores from the exercises do not contribute to the final grade.
  - example exam available on ILIAS

One Last Word on The Exercises...

- Every exercise task is a tiny little scientific work!
- Basic rule for high quality submissions:
  - rephrase the task in your own words,
  - state your solution,
  - convince yourself and your tutor of the correctness of your solution (at best: prove it).

Example:

**Task:** What is the length of the longest line inside the square with side length $a = 19.1$?

**Submission A:**

27

**Submission B:**

The length of the longest straight line fully inside the square with side length $a = 19.1$ is 27.01 (rounded).

The longest straight line inside the square is the diagonal. By Pythagoras, its length is $\sqrt{a^2 + a^2}$. Inserting $a = 19.1$ yields 27.01 (rounded).
Every exercise task is a tiny little scientific work!

Basic rule for high quality submissions:

- rephrase the task in your own words,
- state your solution,
- convince yourself and your tutor of the correctness of your solution (at best: prove it).

Example:

**Task**: What is the length of the longest line inside the square with side length $a = 19.1$?

**Submission A**:

The length of the longest straight line fully inside the square with side length $a = 19.1$ is $27.01$ (rounded).

**Submission B**:

The longest straight line inside the square is the diagonal. By Pythagoras, its length is $\sqrt{a^2 + a^2}$. Inserting $a = 19.1$ yields $27.01$ (rounded).

I have improved my skills in scientific problem solving.

- totally agree 〇 〇 〇 〇
- strongly disagree 〇 〇 〇

I have improved my skills in scientific problem solving.

- totally agree 〇 〇 〇 〇
- strongly disagree 〇 〇 〇
Tell Them What You’ve Told Them...

- **Basic vocabulary:**
  - software, engineering, software engineering,
  - customer, developer, user,
  - successful software development

  → **note:** some definitions are neither formal nor universally agreed

- **(Fun) fact:** software development is not always successful

- **Basic activities of (software) engineering:**
  - gather requirements,
  - design,
  - implementation,
  - quality assurance,
  - project management

  → motivates content of the course – for the case of software

- **Formal (vs. informal) methods**
  - avoid misunderstandings,
  - enable objective, tool-based assessment

  → **note:** still, humans are at the heart of software engineering.

- **Course content and organisation**

---

Any (More) Questions?
References


