*Softwaretechnik / Software-Engineering*

# *Lecture 18: Runtime Verification, Review & Wrapup*

*2016-07-18*

Prof. Dr. Andreas Podelski, **Dr. Bernd Westphal**

Albert-Ludwigs-Universität Freiburg, Germany

# Topic Area Code Quality Assurance: Content

VL 15
- Introduction and Vocabulary

VL 16
- Limits of Software Testing
- Glass-Box Testing
  - Statement-, branch-, term-coverage.

⋮
- Other Approaches
  - Model-based testing,
  - Runtime verification.

VL 17
- Software quality assurance in a larger scope.

⋮
- Program Verification
  - partial and total correctness,
  - Proof System PD.

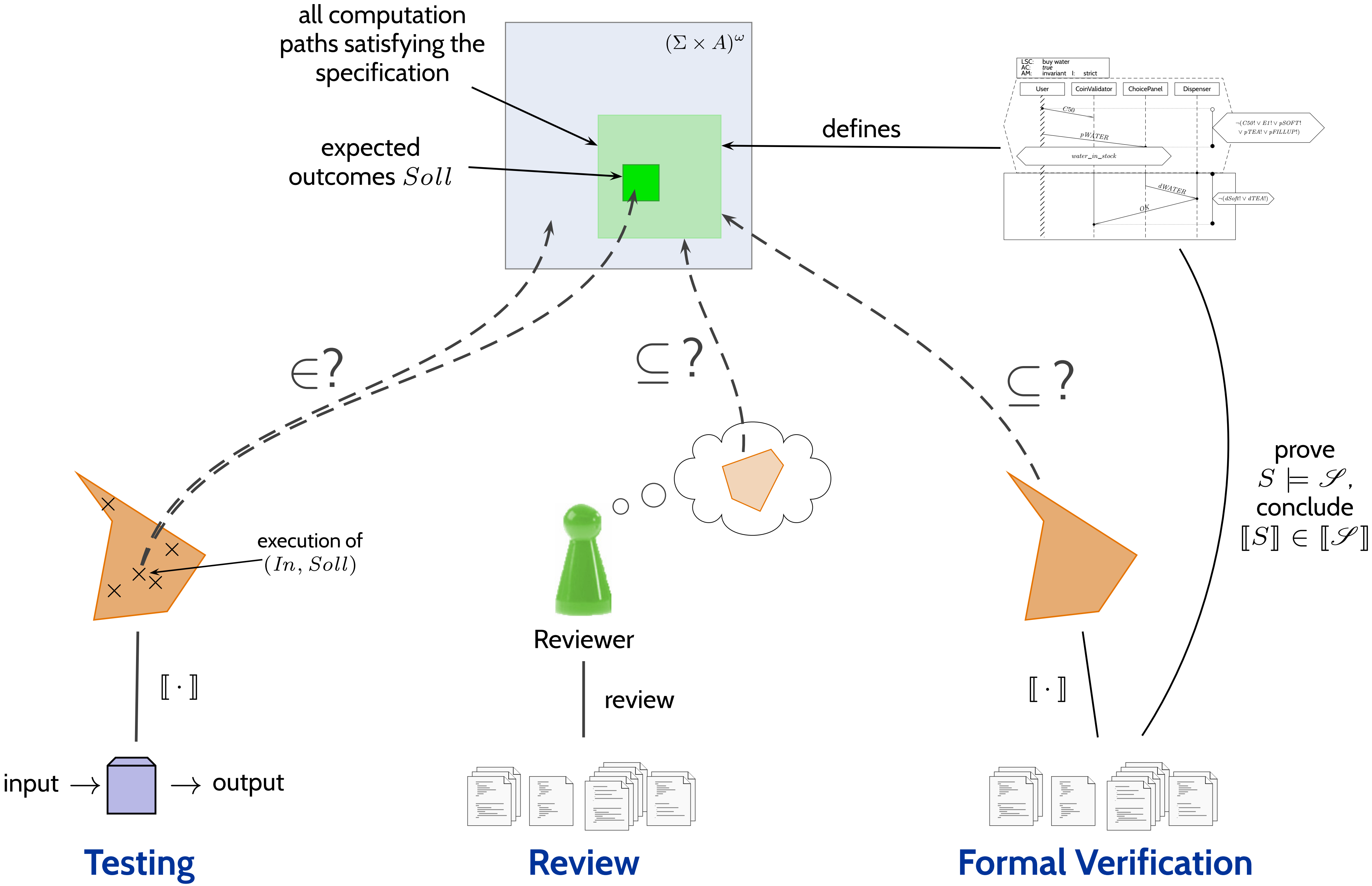
VL 18
- Runtime Verification

⋮
- Review
- Code QA: Discussion

# Content

- **Runtime-Verification**
  - Idea
  - Assertions
  - LSC-Observers
- **Reviews**
  - **Roles** and **artefacts**
  - Review **procedure**
  - Stronger and weaker **variants**
- **Do's and Don'ts** in Code QA
- **Code QA Techniques** Revisited
  - **Test**
  - **Runtime-Verification**
  - **Review**
  - **Static Checking**
  - **Formal Verification**
- **Dependability**

# *Recall: Three Basic Directions*



# *Run-Time Verification*

# *Run-Time Verification: Idea*

Software $S$

- Assume, there is a function $f$ in software $S$ with the following specification:
  - **pre-condition**: $p$, **post-condition**: $q$.
- Computation paths of $S$ may look like this:

$$\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots \xrightarrow{\alpha_{n-1}} \sigma_n \xrightarrow{call\ f} \sigma_{n+1} \cdots \sigma_m \xrightarrow{f\ returns} \sigma_{m+1} \cdots$$

- Assume there are functions $check_p$ and $check_q$,
  which **check** whether $p$ and $q$ hold at the current program state,
  and which **do not modify the program state** (except for program counter).
- **Idea**: create software $S'$ by

  (i) extending $S$ by implementations of $check_p$ and $check_q$,

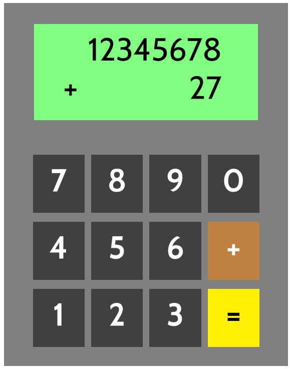
  (ii) call $check_p$ right after entering $f$,

  (iii) call $check_q$ right before returning from $f$.
- For $S'$, obtain computation paths like:

$$\sigma_0 \xrightarrow{\alpha_1} \sigma_1 \xrightarrow{\alpha_2} \sigma_2 \cdots \xrightarrow{\alpha_{n-1}} \sigma_n \xrightarrow{call\ f} \sigma_{n+1} \xrightarrow{check_p} \sigma'_{n+1} \cdots \sigma_m \xrightarrow{check_q} \sigma'_m \xrightarrow{f\ returns} \sigma_{m+1} \cdots$$

- If $check_p$ and $check_q$ notify us of violations of $p$ or $q$,
  then we are **notified of $f$ violating its specification** when running $S'$ (= at run-time).

# Run-Time Verification: Example



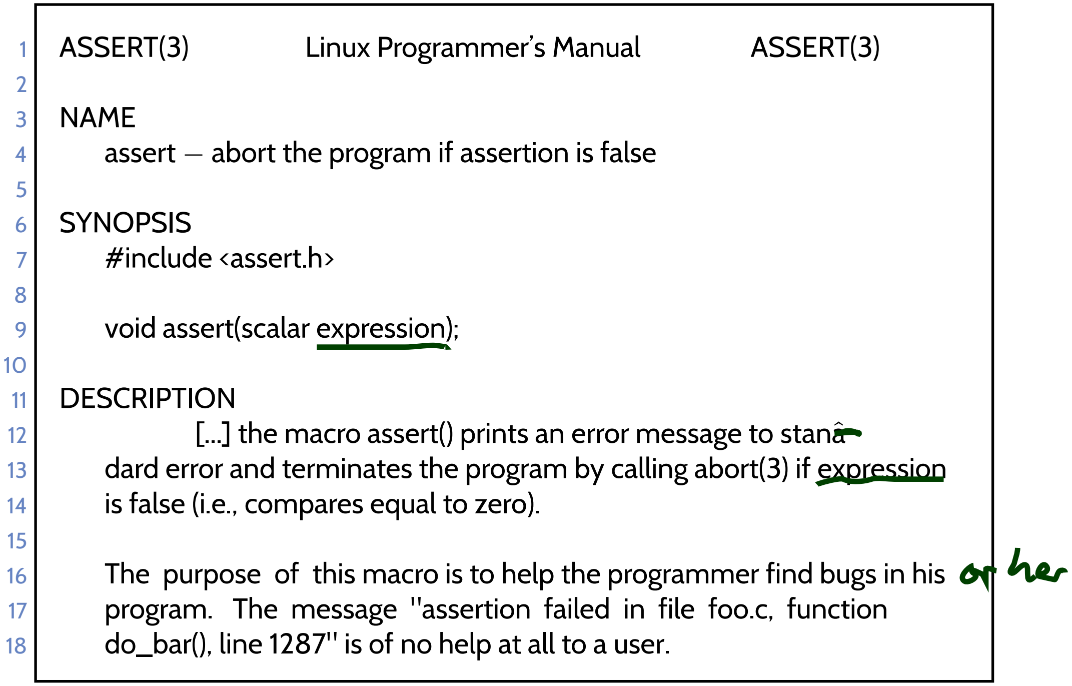
```
int main() {

  while (true) {
    int x = read_number();
    int y = read_number();

    int sum = add( x, y );

    verify_sum( x, y, sum );

    display(sum);
  }
}
```

```
void verify_sum( int x, int y,
                 int sum )
{
  if (sum != (x+y)
      || (x + y > 99999999
          && !(sum < 0)))
  {
    fprintf( stderr,
      "verify_sum:␣error\n" );
    abort();
  }
}
```

# *A Very Useful Special Case: Assertions*

- Maybe the simplest instance of runtime verification: **Assertions**.
- Available in standard libraries of many programming languages (C, C++, Java, ...).

- For example, the C standard library manual reads:



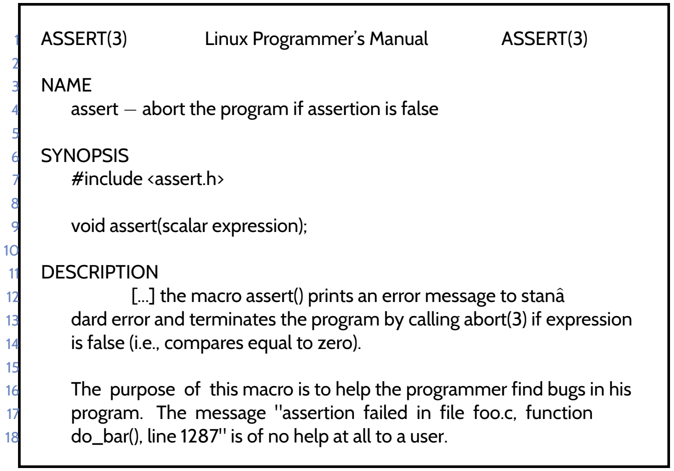
```
ASSERT(3)           Linux Programmer's Manual           ASSERT(3)

NAME
      assert — abort the program if assertion is false

SYNOPSIS
      #include <assert.h>

      void assert(scalar expression);

DESCRIPTION
                [...] the macro assert() prints an error message to stan-
      dard error and terminates the program by calling abort(3) if expression
      is false (i.e., compares equal to zero).

      The purpose of this macro is to help the programmer find bugs in his or her
      program.  The  message  "assertion  failed  in  file  foo.c,  function
      do_bar(), line 1287" is of no help at all to a user.
```

- In C code, `assert` can be **disabled** in **production code** (`-D NDEBUG`).

# Assertions At Work

- The abstract $f$-example from **run-time verification**:

```
void f( ... ) {
  assert( p );
  ...
  assert( q );
}
```



ASSERT(3) Linux Programmer's Manual ASSERT(3)

NAME
assert – abort the program if assertion is false

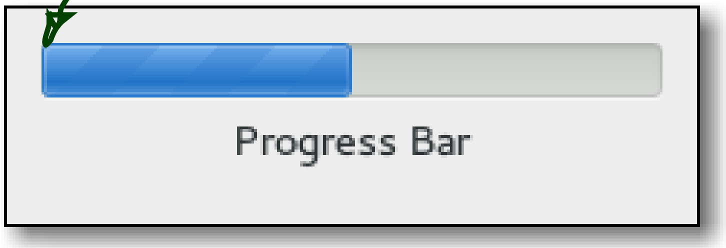
SYNOPSIS
#include <assert.h>

void assert(scalar expression);

DESCRIPTION
[...] the macro assert() prints an error message to stanâ
dard error and terminates the program by calling abort(3) if expression is false (i.e., compares equal to zero).

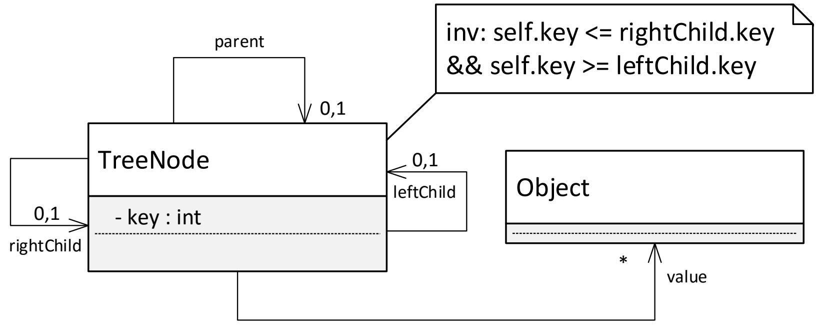
The purpose of this macro is to help the programmer find bugs in his program. The message "assertion failed in file foo.c, function do_bar(), line 1287" is of no help at all to a user.

- Compute the width of a progress bar: window_left



```

int progress_bar_width( int progress, int window_left, int window_right )
{
  assert(window_left <= window_right ); /* pre-condition */
  ...
  /* treat special cases 0 and 100 */
  ...
  assert( 0 < progress && progress < 100); // extremal cases already treated
  ...
  assert( window_left <= r && r <= window_right ); /* post-condition */
  return r;
}
```

# Assertions At Work II



- Recall the **structure model** with Proto-OCL constraint from Exercise Sheet 4.
- Assume, we add a method `set_key()` to class **TreeNode**:

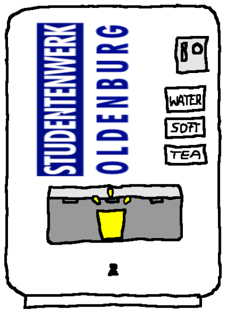
```
class TreeNode {

  private int key;
  TreeNode parent, leftChild, rightChild;

  public int get_key() { return key; }

  public void set_key( int new_key ) {
    key = new_key;
  }
}
```

- We can **check consistency** with the Proto-OCL constraint at runtime by using assertions:

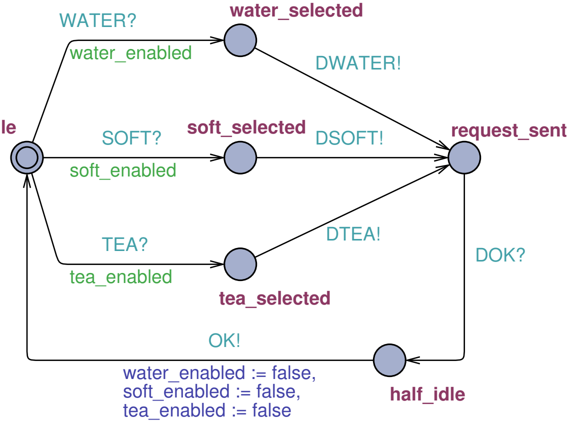
```
public void set_key( int new_key ) {
  assert ( parent == null || parent.get_key() <= new_key  ); ?
  assert ( leftChild == null || new_key <= leftChild.get_key() );
  assert ( rightChild == null || new_key <= rightChild.get_key() );

  key = new_key;
}
```

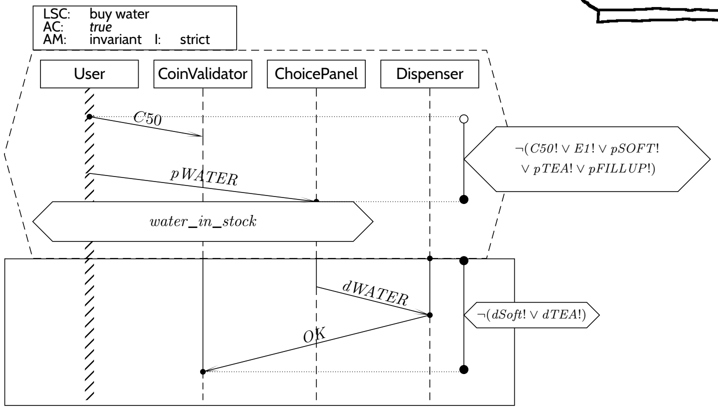
- Use `java -ea ...` to **enable assertion checking** (disabled by default).
  (cf. `https://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html`)

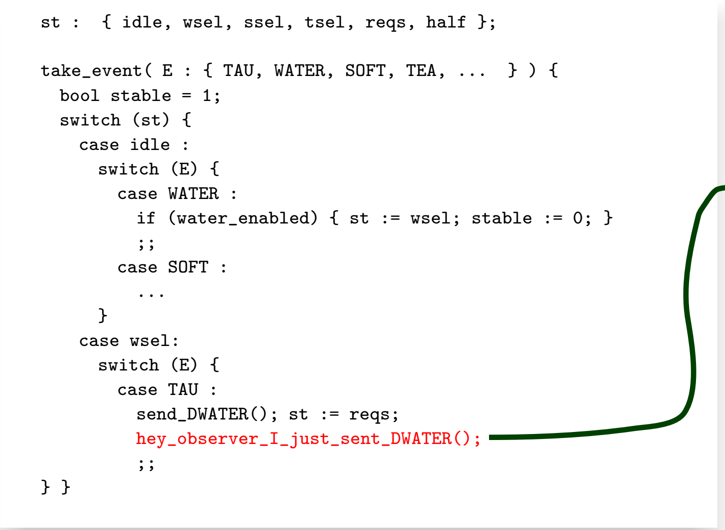
# *More Complex Run-Time Verification: LSC Observers*



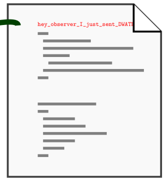
**ChoicePanel**:

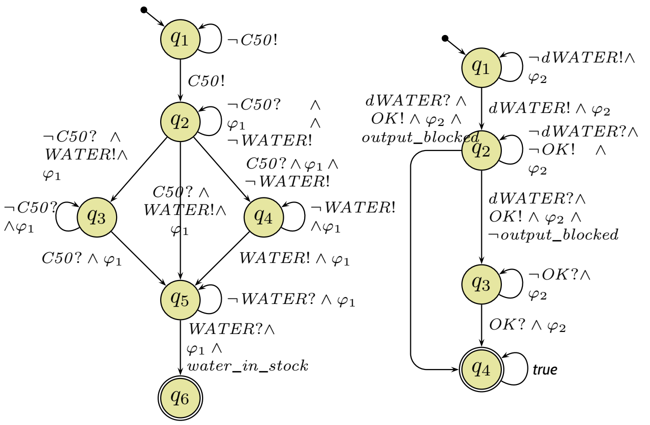






```
st :  { idle, wsel, ssel, tsel, reqs, half };

take_event( E : { TAU, WATER, SOFT, TEA, ...  } ) {
  bool stable = 1;
  switch (st) {
    case idle :
      switch (E) {
        case WATER :
          if (water_enabled) { st := wsel; stable := 0; }
          ;;
        case SOFT :
          ...
      }
    case wsel:
      switch (E) {
        case TAU :
          send_DWATER(); st := reqs;
          hey_observer_I_just_sent_DWATER();
          ;;
} }
```




$q_1$
$\neg C50!$
$C50!$
$q_2$
$\neg C50? \wedge \varphi_1 \wedge \neg WATER!$
$\neg C50? \wedge WATER! \wedge \varphi_1$
$C50? \wedge \varphi_1 \wedge \neg WATER!$
$C50? \wedge WATER! \wedge \varphi_1$
$q_3$
$\neg C50? \wedge \varphi_1$
$q_4$
$\neg WATER! \wedge \varphi_1$
$C50? \wedge \varphi_1$
$WATER! \wedge \varphi_1$
$q_5$
$\neg WATER? \wedge \varphi_1$
$WATER? \wedge \varphi_1 \wedge water\_in\_stock$
$q_6$
$q_1$
$\neg dWATER! \wedge \varphi_2$
$dWATER! \wedge \varphi_2$
$dWATER? \wedge OK! \wedge \varphi_2 \wedge output\_blocked$
$q_2$
$\neg dWATER? \wedge \neg OK! \wedge \varphi_2$
$dWATER? \wedge OK! \wedge \varphi_2 \wedge \neg output\_blocked$
$q_3$
$\neg OK? \wedge \varphi_2$
$OK? \wedge \varphi_2$
$q_4$
true

# Run-Time Verification: Discussion

- **Experience**:

  During development, **assertions** for pre/post conditions and intermediate invariants are an **extremely powerful** tool with a **very attractive gain/effort ratio** (low effort, high gain).

  - Assertions effectively work as **safe-guard against unexpected use** of functions and **regression**, e.g. during later maintenance or efficiency improvement.
  - Can serve as **formal** (support of) **documentation**:

    "Dear reader, at this point in the program, I expect condition *expr* to hold, because…".

- **Development- vs. Release Versions**:

  - Common practice:
    - development version **with** run-time verification enabled (cf. `assert(3)`),
    - release version **without** run-time verification.

    If run-time verification is enabled in a release version,

    - software should **terminate as gracefully as possible** (e.g. try to save data),
    - **save information from assertion failure** if possible for future analysis.

    **Risk**: with bad luck, the software only behaves well **because of** the run-time verification code…

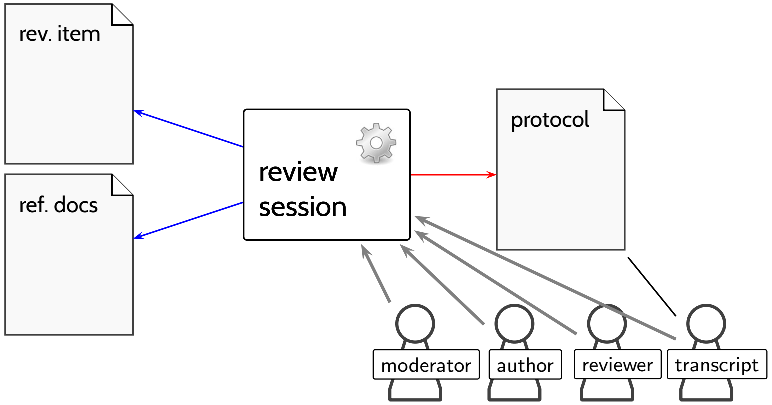
    Then disabling run-time verification "breaks" the software. Yet very complex run-time verification may significantly slow down the software, so needs to be disabled…

# *Content*

- **Runtime-Verification**
  - Idea
  - Assertions
  - LSC-Observers
- **Reviews**
  - **Roles** and **artefacts**
  - Review **procedure**
  - Stronger and weaker **variants**
- **Do's and Don'ts** in Code QA
- **Code QA Techniques** Revisited
  - **Test**
  - **Runtime-Verification**
  - **Review**
  - **Static Checking**
  - **Formal Verification**
- **Dependability**

# *Review*

# Reviews



- **Input to Review Session**:
  - **Review item**: can be every closed, human-readable part of software (documentation, module, test data, installation manual, etc.)

    **Social aspect**: it is an **artefact** which is examined, not the **human** (who created it).
  - **Reference documents**: need to enable an assessment

    (requirements specification, guidelines (e.g. coding conventions), catalogue of questions ("all variables initialised?"), etc.)

- **Roles**:

  **Moderator**: leads session, responsible for properly conducted procedure.

  **Author**: (representative of the) creator(s) of the artefact under review; is present to listen to the discussions; can answer questions; does not speak up if not asked.

  **Reviewer(s)**: person who is able to judge the artefact under review; maybe different reviewers for different aspects (programming, tool usage, etc.), at best experienced in detecting inconsistencies or incompleteness.

  **Transcript Writer**: keeps minutes of review session, can be assumed by author.

- The **review team** consists of everybody but the author(s).

# *Review Procedure Over Time*

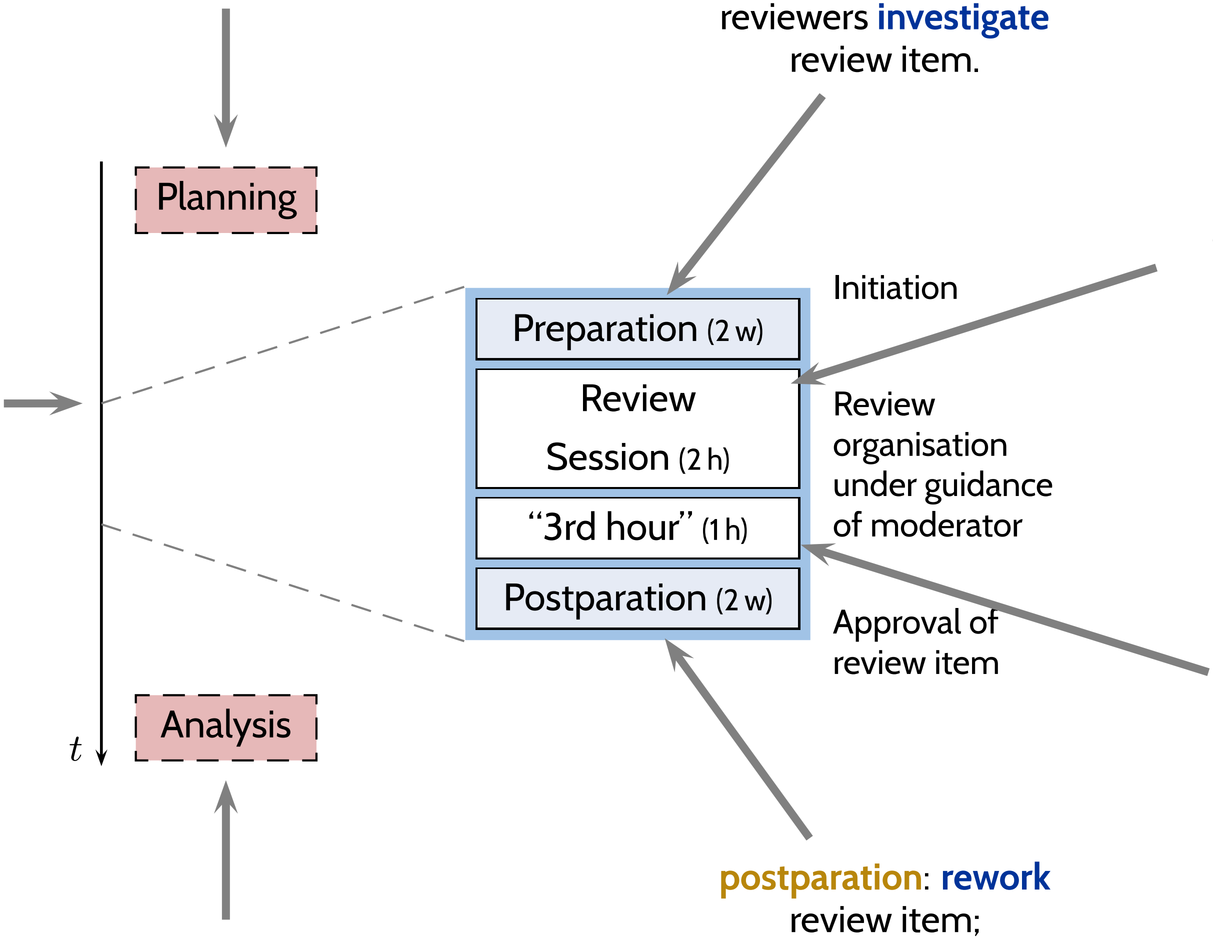
planning: reviews need **time** in the project plan.

preparation: reviewers **investigate** review item.

review session: reviewers **report**, evaluate, and document issues; **resolve** open questions.

a review is triggered, e.g., by a submission to the revision control system:

the moderator **invites** (include review item in invitation), and states **review missions**.



“3rd hour”: time for **informal chat**, reviewers may **state proposals** for solutions or improvements.

analysis: **improve** development and review process.

postparation: **rework** review item; responsibility of the author(s).

- Reviewers **re-assess** reworked review item (until **approval** is declared).

# Review Rules (Ludewig and Lichter, 2013)

(i) The moderator organises the review, issues invitations, supervises the review session.

(ii) The moderator may terminate the review if conduction is not possible, e.g., due to inputs, preparation, or people missing.

(iii) The review session is limited to 2 hours.
If needed: organise more sessions.

(iv) The review item is under review,
not the author(s).
Reviewers choose their words accordingly.
Authors neither defend themselves nor the review item.

(v) Roles are not mixed up, e.g., the moderator does not act as reviewer.
(Exception: author may write transcript.)

(vi) Style issues (outside fixed conventions) are not discussed.

(vii) The review team is not supposed to develop solutions.
Issues are not noted down in form of tasks for the author(s).

(viii) Each reviewer gets the opportunity to present her/his findings appropriately.

(ix) Reviewers need to reach consensus on issues, consensus is noted down.

(x) Issues are classified as:

- critical (review unusable for purpose),
- major (usability severely affected),
- minor (usability hardly affected),
- good (no problem).

(xi) The review team declares:

- accept without changes,
- accept with changes,
- do not accept.

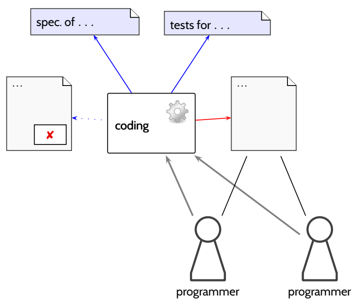
(xii) The protocol is signed by all participants.

# *Stronger and Weaker Review Variants*

- **Design and Code Inspection** (Fagan, 1976, 1986)
  - deluxe variant of review,
  - **approx. 50% more time, approx. 50% more errors found.**
- **Review**
- **Structured Walkthrough**
  - simple variant of review:
    - **developer** moderates walkthrough-session,
    - **developer** presents artefact(s),
    - **reviewer** poses (prepared or spontaneous) questions,
    - issues are noted down,
  - **Variation point**: do reviewers see the artefact before the session?
  - **less effort, less effective**.

  → **disadvantages**: unclear reponsibilities; "salesman"-**developer** may trick **reviewers**.

**XP's pair programming**
("on-the-fly review"?)



- **Comment** ('Stellungnahme')
  - **colleague(s)** of **developer** read artefacts,
  - **developer** considers feedback.

  → **advantage**: low organisational effort;
  → **disadvantages**: choice of colleagues may be biased; no protocol;
  consideration of comments at discretion of developer.
- **Careful Reading** ('Durchsicht')
  - done by **developer**,
  - **recommendation**: "away from screen" (use print-out or different device and situation)

# *Some Final, General Guidelines*

# *Do's and Don'ts in Code Quality Assurance*

**Avoid** using special **examination versions** for examination.
(Test-harness, stubs, etc. **may have errors** which may cause false positives and (!) negatives.)

**Avoid** to **stop examination** when the first error is detected.

**Clear**: Examination should be aborted if the examined program is not executable at all.

**Do not modify** the artefact under examination **during** examinatin.

- otherwise, it is **unclear what exactly** has been examined ("moving target"), (examination results need to be uniquely traceable to one artefact version.)
- fundamental flaws are sometimes **easier to detect** with a **complete picture** of unsuccessful/successful tests,
- **changes are particularly error-prone**, should not happen "en passant" in examination,
- fixing flaws during examination may cause them to **go uncounted** in the **statistics** (which we need for all kinds of estimation),
- roles **developer** and **examinor** are different anyway: an **examinor** fixing flaws would **violate the role assignment**.

**Do not switch** (fine grained) between **examination** and **debugging**.

# Content

- **Runtime-Verification**
  - Idea
  - Assertions
  - LSC-Observers
- **Reviews**
  - **Roles** and **artefacts**
  - Review **procedure**
  - Stronger and weaker **variants**
- **Do's and Don'ts** in Code QA
- **Code QA Techniques** Revisited
  - **Test**
  - **Runtime-Verification**
  - **Review**
  - **Static Checking**
  - **Formal Verification**
- **Dependability**

# *Code Quality Assurance Techniques Revisited*

# *Techniques Revisited*

| | auto-matic | prove "can run" | toolchain considered | exhaus-tive | prove correct | partial results | entry cost |
|---|---|---|---|---|---|---|---|
| **Test** | (✔) | ✔ | ✔ | ✘ | ✘ | ✔ | ✔ |
| Runtime-Verification | | | | | | | |
| Review | | | | | | | |
| Static Checking | | | | | | | |
| Verification | | | | | | | |

**Strengths:**

- can be **fully automatic** (yet not easy for GUI programs);
- negative test **proves "program not completely broken"**, "can run" (or positive scenarios);
- **final product is examined**, thus toolchain and platform considered;
- one can stop at any time and take **partial results**;
- few, simple test cases are usually **easy to obtain**; ← "one test case per feature"
- provides **reproducible counter-examples** (good starting point for repair).

**Weaknesses:**

- (in most cases) **vastly incomplete**, thus no proofs of correctness;
- creating test cases for complex functions (or complex conditions) **can be difficult**;
- **maintenance** of many, complex test cases be **challenging**.
- executing many tests may need **substantial time** (but: can sometimes be run in parallel);

# *Techniques Revisited*

| | auto-matic | prove "can run" | toolchain considered | exhaus-tive | prove correct | partial results | entry cost |
|---|---|---|---|---|---|---|---|
| Test | (✔) | ✔ | ✔ | ✘ | ✘ | ✔ | ✔ |
| **Runtime-Verification** | ✔ | (✔) | ✔ | (✘) | ✘ | ✔ | (✔) |
| Review | | | | | | | |
| Static Checking | | | | | | | |
| Verification | | | | | | | |

**Strengths:**

- **fully automatic** (once observers are in place);
- **provides counter-example**;
- (nearly) **final product is examined**, thus toolchain and platform considered;
- one can stop at any time and take **partial results**;
- **`assert`-statements have a very good effort/effect ratio**.

**Weaknesses:**

- counter-examples **not necessarily reproducible**;
- may negatively affect **performance**;
- code is changed, program may only run **because of** the observers;
- completeness depends on usage,
  may also be **vastly incomplete**, so no correctness proofs;
- constructing observers for complex properties may be **difficult**,
  one needs to learn how to construct observers.

# *Techniques Revisited*

| | auto-matic | prove "can run" | toolchain considered | exhaus-tive | prove correct | partial results | entry cost |
|---|---|---|---|---|---|---|---|
| Test | (✔) | ✔ | ✔ | ✘ | ✘ | ✔ | ✔ |
| Runtime-Verification | ✔ | (✔) | ✔ | (✘) | ✘ | ✔ | (✔) |
| **Review** | ✘ | ✘ | ✘ | (✔) | (✔) | ✔ | (✔) |
| Static Checking | | | | | | | |
| Verification | | | | | | | |

**Strengths**:

- human readers can **understand the code**, may spot point errors;
- reported to be **highly effective**;
- one can stop at any time and take **partial results**;
- intermediate **entry costs**;
  **good effort/effect ratio achievable**.

**Weaknesses**:

- no **tool support**;
- no results on actual execution, **toolchain not reviewed**;
- human readers may **overlook** errors; usually not aiming at proofs.
- does (in general) **not provide counter-examples**,
  developers may deny existence of error.

# *Techniques Revisited*

| | automatic | prove “can run” | toolchain considered | exhaustive | prove correct | partial results | entry cost |
|---|---|---|---|---|---|---|---|
| Test | (✔) | ✔ | ✔ | ✘ | ✘ | ✔ | ✔ |
| Runtime-Verification | ✔ | (✔) | ✔ | (✘) | ✘ | ✔ | (✔) |
| Review | ✘ | ✘ | ✘ | (✔) | (✔) | ✔ | (✔) |
| **Static Checking** | ✔ | (✘) | ✘ | ✔ | (✔) | ✔ | (✘) |
| Verification | | | | | | | |

**Strengths**:

- there are (commercial), **fully automatic** tools (lint, Coverity, Polyspace, etc.);
- some tools are **complete** (relative to assumptions on language semantics, platform, etc.);
- can be **faster than testing**;
- one can stop at any time and take **partial results**.

**Weaknesses**:

- no results on actual execution, **toolchain not reviewed**;
- can be very **resource consuming** (if few false positives wanted), e.g., code may need to be “designed for static analysis”.
- many false positives can be very **annoying to developers** (if fast checks wanted);
- distinguish **false from true positives** can be challenging;
- **configuring the tools** (to limit false positives) can be challenging.

# *Techniques Revisited*

| | auto-<br>matic | prove "can run" | toolchain considered | exhaus-<br>tive | prove correct | partial results | entry cost |
|---|---|---|---|---|---|---|---|
| Test | (✔) | ✔ | ✔ | ✘ | ✘ | ✔ | ✔ |
| Runtime-Verification | ✔ | (✔) | ✔ | (✘) | ✘ | ✔ | (✔) |
| Review | ✘ | ✘ | ✘ | (✔) | (✔) | ✔ | (✔) |
| Static Checking | ✔ | (✘) | ✘ | ✔ | (✔) | ✔ | (✘) |
| **Verification** | (✔) | ✘ | ✘ | ✔ | ✔ | (✘) | ✘ |

**Strengths:**

- some **tool support** available (few commercial tools);
- **complete** (relative to assumptions on language semantics, platform, etc.);
- thus can provide **correctness proofs**;
- can prove correctness for **multiple language semantics and platforms** at a time;
- can be **more efficient than other techniques**.

**Weaknesses:**

- no results on actual execution, **toolchain not reviewed**;
- not many **intermediate results**: "half of a proof" may not allow any useful conclusions;
- **entry cost high**: significant training is useful to know how to deal with tool limitations;
- proving things is challenging: failing to find a proof does not allow any useful conclusion;
- **false negatives** (broken program "proved" correct) hard to detect.

# *Quality Assurance — Concluding Discussion*

# *Proposal: Dependability Cases (Jackson, 2009)*

- A **dependable** system is one you can **depend** on – that is, you can place your trust in it.

> "Developers [should] **express the critical properties**
> and **make an explicit argument** that the system satisfies them."

> **quality assurance** – (1) A planned and systematic pattern of all actions necessary to provide **adequate confidence** that an item or product conforms to established technical requirements.
> **IEEE 610.12 (1990)**

**Proposed Approach**:

- Identify the **critical requirements**, and determine what **level of confidence** is needed.

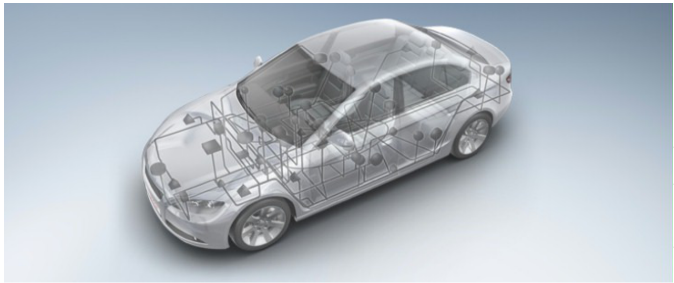
  Most systems do also have **non-critical** requirements.
- Construct a **dependability case**:
  - an argument, that the software, in concert with other components, establishes the critical properties.
- The case should be
  - **auditable**: can (easily) be evaluated by third-party certifier.
  - **complete**: no holes in the argument, any assumptions that are not justified should be noted (e.g. assumptions on compiler, on protocol obeyed by users, etc.)
  - **sound**: e.g. should not claim full correctness [...] based on nonexhaustive testing; should not make unwarranted assumptions on independence of component failures; etc.

# *Critical Systems*

Still, it seems like computer systems more or less **inevitably have errors**.

Then why…

Laurent ERRERA, CC BY-SA 2.0, commons.wikimedia.org/w/index.php?curid=29838567



Robert Bosch GmbH

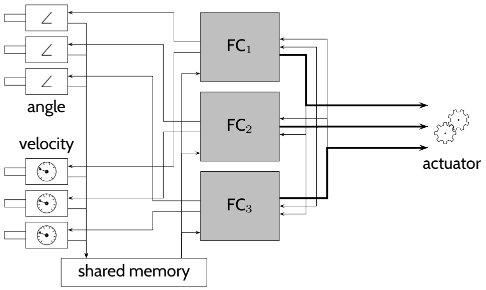
- … do modern planes fly at all?

  (i) **very careful development**,

  (ii) **very thorough analysis**,

  (iii) **strong regulatory obligations**.

  **Plus**: classical engineering wisdom for high reliability, like **redundancy**.



(Mrugalla et al., 2005)

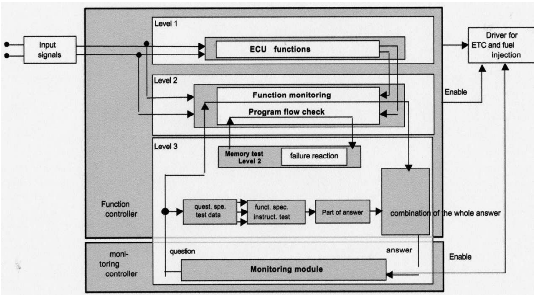
- … do modern cars drive at all?

  (i) **careful development**,

  (ii) **thorough analysis**,

  (iii) **regulatory obligations**.

  **Plus**: classical engineering wisdom for high reliability, like **monitoring**.



https://www.iav.com/sites/default/files/attachments/seite/ak-egas-v5-5-en-130705.pdf

# *Tell Them What You've Told Them...*

- **Runtime Verification**
  - (as the name suggests) checks properties at **program run-time**,
  - a good **pinch of `assert`'s** can be a valuable safe-guard against
    - **regressions**,
    - usage **outside specification**,
    - etc.

    and serve as **formal documentation** of assumptions.
- **Review** (structured examination of artefacts by humans)
  - (mild variant) advocated in the XP approach,
  - **not uncommon**:
    lead programmer reviews **all commits** from team members,
  - literature reports good effort/effect ratio achievable.
- All approaches to **code quality assurance** have their
  - **advantages** and **drawbacks**.
  - Which to use? It depends!
- **Dependability Cases**
  - an (auditable, complete, sound) argument,
    that a software has the **critical properties**.

# *References*

# References

Fagan, M. (1976). Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211.

Fagan, M. (1986). Advances in software inspections. *IEEE Transactions On Software Engineering*, 12(7):744–751.

IEEE (1990). *IEEE Standard Glossary of Software Engineering Terminology*. Std 610.12-1990.

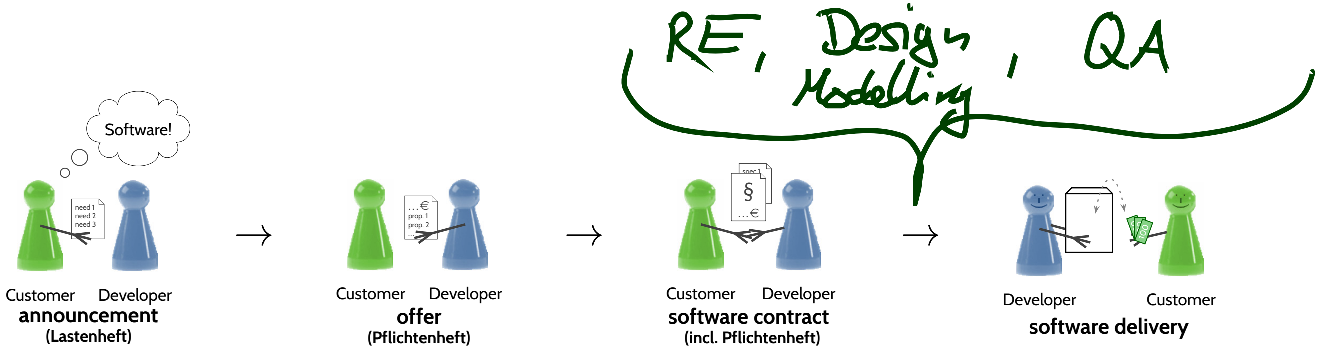
Jackson, D. (2009). A direct path to dependable software. *Comm. ACM*, 52(4).

Ludewig, J. and Lichter, H. (2013). *Software Engineering*. dpunkt.verlag, 3. edition.

Mrugalla, C., Robbe, O., Schinz, I., Toben, T., and Westphal, B. (2005). Formal verification of a sensor voting and monitoring UML model. In Siv Hilde Houmb, Jan Jürjens, R. F., editor, *Proceedings of the 4th International Workshop on Critical Systems Development Using Modeling Languages (CSDUML 2005)*, pages 37–51. Technische Universität München.

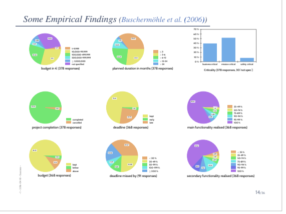
# *Looking Back:*
# *18 Lectures on Software Engineering*

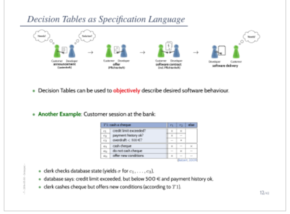
# Contents of the Course

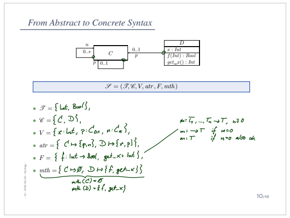


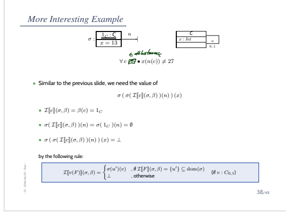
| | | |
|---|---|---|
| Introduction | L 1: | 18.4., Mon |
| Scales, Metrics, Costs | L 2: | 21.4., Thu |
| | L 3: | 25.4., Mon |
| | T 1: | 28.4., Thu |
| Development | L 4: | 2.5., Mon |
| | - | 5.5., Thu |
| Process | L 5: | 9.5., Mon |
| | L 6: | 12.5., Thu |
| | - | 16.5., Mon |
| | - | 19.5., Thu |
| | T 2: | 23.5., Mon |
| | - | 26.5., Thu |
| Requirements Engineering | L 7: | 30.5., Mon |
| | L 8: | 2.6., Thu |
| | L 9: | 6.6., Mon |
| | T 3: | 9.6., Thu |
| | L10: | 13.6., Mon |
| Architecture & Design | L 11: | 16.6., Thu |
| | L12: | 20.6., Mon |
| | T 4: | 23.6., Thu |
| Software Modelling | L13: | 27.6., Mon |
| | L14: | 30.6., Thu |
| | L15: | 4.7., Mon |
| | T 5: | 7.7., Thu |
| Quality Assurance (Testing, Formal Verification) | L16: | 11.7., Mon |
| | L17: | 14.7., Thu |
| | L18: | 18.7., Mon |
| Wrap-Up | L19: | 21.7., Thu |

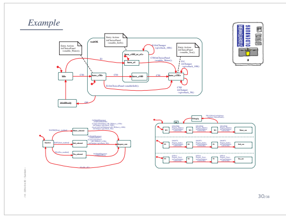
# What Did We Do?

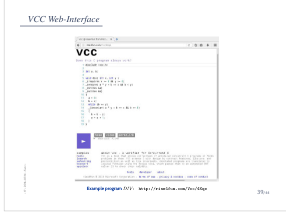

Some Empirical Findings (Buschermöhle et al. (2006))

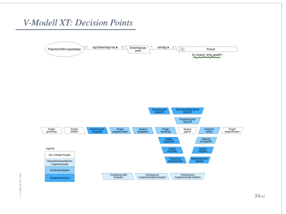

Decision Tables as Specification Language

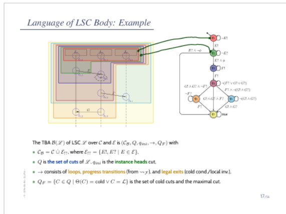

From Abstract to Concrete Syntax

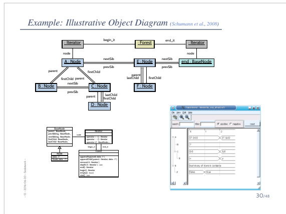

More Interesting Example

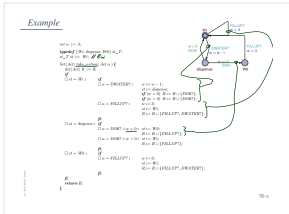

Example

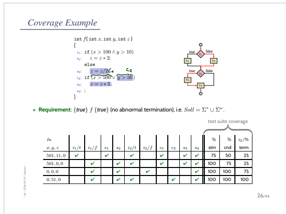

VCC Web-Interface

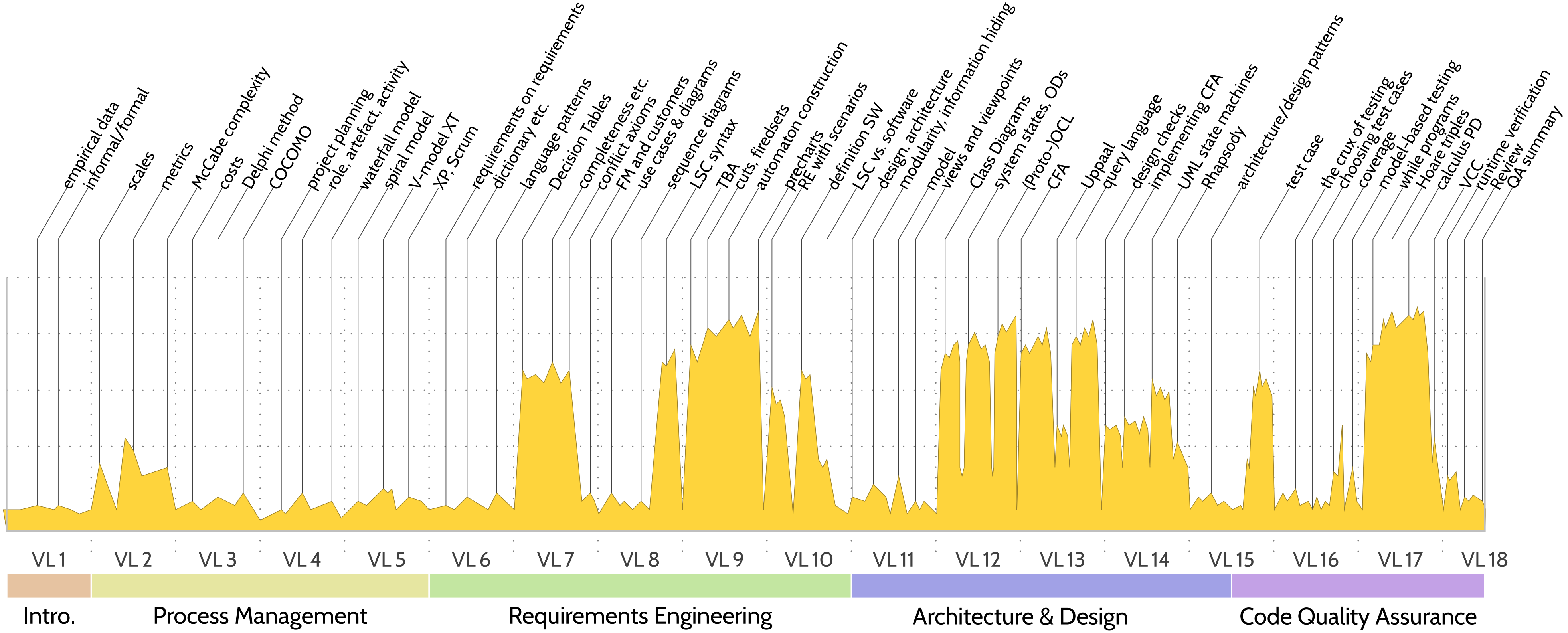

V-Modell XT: Decision Points


Language of LSC Body: Example


Example: Illustrative Object Diagram (Schumann et al., 2008)


Example


Coverage Example



# *Expectations*

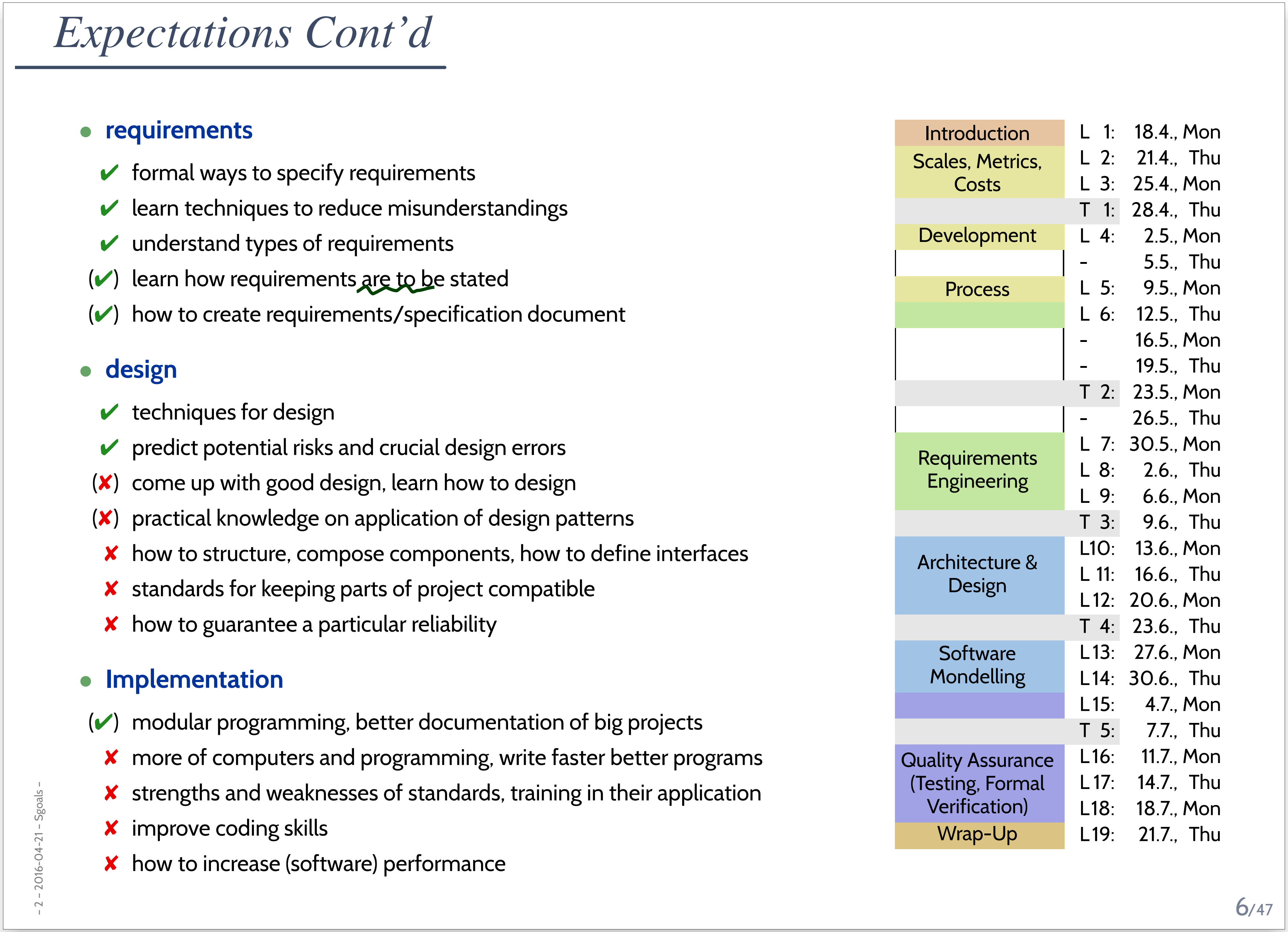
- none, because mandatory course
- **overall**
  - ✔ well-structured lectures
  - (✔) praxis oriented
  - ✘ practical knowledge about planning, designing and testing software
  - ✔ improve skills in scientific work
  - (✔) more about scientific methods
- **other courses**
  - ✘ more on how courses are linked together
  - ✘ skills we need to organise SoPra
  - ✔ maybe transfer knowledge in SoPra
- **“real world”**
  - ✔ vocabulary and methods in professional software development
  - ✔ learn how things work in a company, to easier integrate into teams, e.g., communication
- **kinds of software**
  - ✔ embedded systems and software
  - ✘ how to combine HW and SW parts

| | | |
|---|---|---|
| Introduction | L 1: | 18.4., Mon |
| Scales, Metrics, Costs | L 2: | 21.4., Thu |
| | L 3: | 25.4., Mon |
| | T 1: | 28.4., Thu |
| Development | L 4: | 2.5., Mon |
| | - | 5.5., Thu |
| Process | L 5: | 9.5., Mon |
| | L 6: | 12.5., Thu |
| | - | 16.5., Mon |
| | - | 19.5., Thu |
| | T 2: | 23.5., Mon |
| | - | 26.5., Thu |
| Requirements Engineering | L 7: | 30.5., Mon |
| | L 8: | 2.6., Thu |
| | L 9: | 6.6., Mon |
| | T 3: | 9.6., Thu |
| Architecture & Design | L10: | 13.6., Mon |
| | L 11: | 16.6., Thu |
| | L12: | 20.6., Mon |
| | T 4: | 23.6., Thu |
| Software Mondelling | L13: | 27.6., Mon |
| | L14: | 30.6., Thu |
| | L15: | 4.7., Mon |
| | T 5: | 7.7., Thu |
| Quality Assurance (Testing, Formal Verification) | L16: | 11.7., Mon |
| | L 17: | 14.7., Thu |
| | L18: | 18.7., Mon |
| Wrap-Up | L19: | 21.7., Thu |

# *Expectations Cont'd*

- **software development**
  - ✔ understand how software development practically works
  - ✔ developing, maintaining software at bigger scale
  - ✔ aspects of software development
- **software project management**
  - ✔ learn what is important to plan
  - ✔ how to structure the process of a project
  - ✔ how to keep control of project, measure success
  - ✘ which projects need full-time project manager
  - ✘ which kind of documentation is really necessary
  - ✘ want to get better in leading a team; how to lead team of engineers
- **cost estimation**
  - ✔ how to estimate time and effort
  - (✘) formal methods for better planning of projects
  - ✘ tools which help planning
- **quality**
  - ✔ learn ways how to judge quality based on the requirements
  - ✔ avoid mistakes during software development
  - ✔ make better programs, or make programs more efficiently

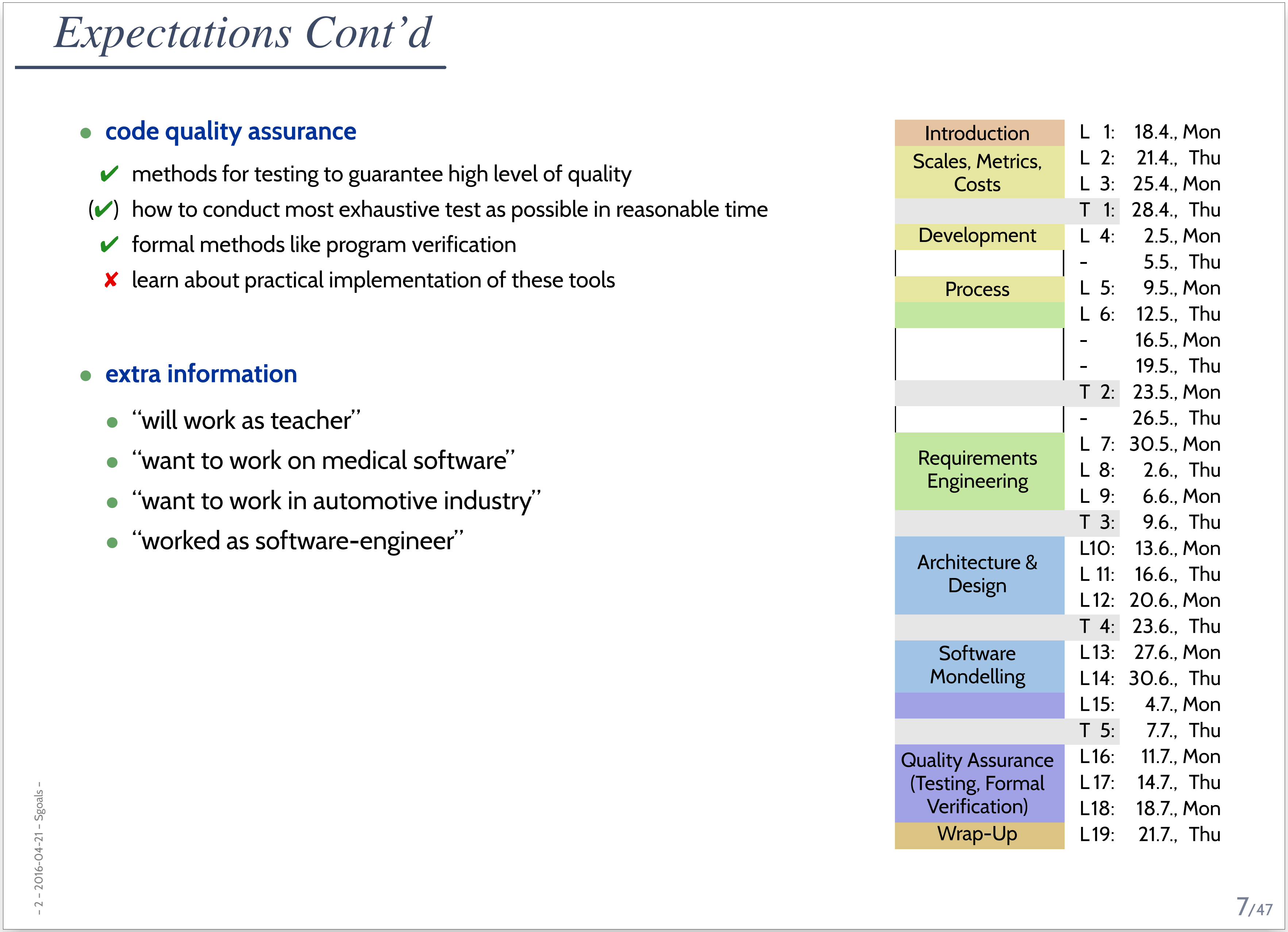
| | | |
|---|---|---|
| Introduction | L 1: | 18.4., Mon |
| Scales, Metrics, Costs | L 2: | 21.4., Thu |
| | L 3: | 25.4., Mon |
| | T 1: | 28.4., Thu |
| Development | L 4: | 2.5., Mon |
| | - | 5.5., Thu |
| Process | L 5: | 9.5., Mon |
| | L 6: | 12.5., Thu |
| | - | 16.5., Mon |
| | - | 19.5., Thu |
| | T 2: | 23.5., Mon |
| | - | 26.5., Thu |
| Requirements Engineering | L 7: | 30.5., Mon |
| | L 8: | 2.6., Thu |
| | L 9: | 6.6., Mon |
| | T 3: | 9.6., Thu |
| Architecture & Design | L10: | 13.6., Mon |
| | L 11: | 16.6., Thu |
| | L12: | 20.6., Mon |
| | T 4: | 23.6., Thu |
| Software Mondelling | L13: | 27.6., Mon |
| | L14: | 30.6., Thu |
| | L15: | 4.7., Mon |
| | T 5: | 7.7., Thu |
| Quality Assurance (Testing, Formal Verification) | L16: | 11.7., Mon |
| | L17: | 14.7., Thu |
| | L18: | 18.7., Mon |
| Wrap-Up | L19: | 21.7., Thu |



# Expectations Cont'd

- **requirements**
  - ✔ formal ways to specify requirements
  - ✔ learn techniques to reduce misunderstandings
  - ✔ understand types of requirements
  - (✔) learn how requirements are to be stated
  - (✔) how to create requirements/specification document
- **design**
  - ✔ techniques for design
  - ✔ predict potential risks and crucial design errors
  - (✘) come up with good design, learn how to design
  - (✘) practical knowledge on application of design patterns
  - ✘ how to structure, compose components, how to define interfaces
  - ✘ standards for keeping parts of project compatible
  - ✘ how to guarantee a particular reliability
- **Implementation**
  - (✔) modular programming, better documentation of big projects
  - ✘ more of computers and programming, write faster better programs
  - ✘ strengths and weaknesses of standards, training in their application
  - ✘ improve coding skills
  - ✘ how to increase (software) performance

| | | |
|---|---|---|
| Introduction | L 1: | 18.4., Mon |
| Scales, Metrics, Costs | L 2: | 21.4., Thu |
| | L 3: | 25.4., Mon |
| | T 1: | 28.4., Thu |
| Development | L 4: | 2.5., Mon |
| | - | 5.5., Thu |
| Process | L 5: | 9.5., Mon |
| | L 6: | 12.5., Thu |
| | - | 16.5., Mon |
| | - | 19.5., Thu |
| | T 2: | 23.5., Mon |
| | - | 26.5., Thu |
| Requirements Engineering | L 7: | 30.5., Mon |
| | L 8: | 2.6., Thu |
| | L 9: | 6.6., Mon |
| | T 3: | 9.6., Thu |
| Architecture & Design | L10: | 13.6., Mon |
| | L 11: | 16.6., Thu |
| | L12: | 20.6., Mon |
| | T 4: | 23.6., Thu |
| Software Mondelling | L13: | 27.6., Mon |
| | L14: | 30.6., Thu |
| | L15: | 4.7., Mon |
| | T 5: | 7.7., Thu |
| Quality Assurance (Testing, Formal Verification) | L16: | 11.7., Mon |
| | L 17: | 14.7., Thu |
| | L18: | 18.7., Mon |
| Wrap-Up | L19: | 21.7., Thu |



# Expectations Cont’d

- **code quality assurance**
  - ✔ methods for testing to guarantee high level of quality
  - (✔) how to conduct most exhaustive test as possible in reasonable time
  - ✔ formal methods like program verification
  - ✘ learn about practical implementation of these tools

- **extra information**
  - “will work as teacher”
  - “want to work on medical software”
  - “want to work in automotive industry”
  - “worked as software-engineer”

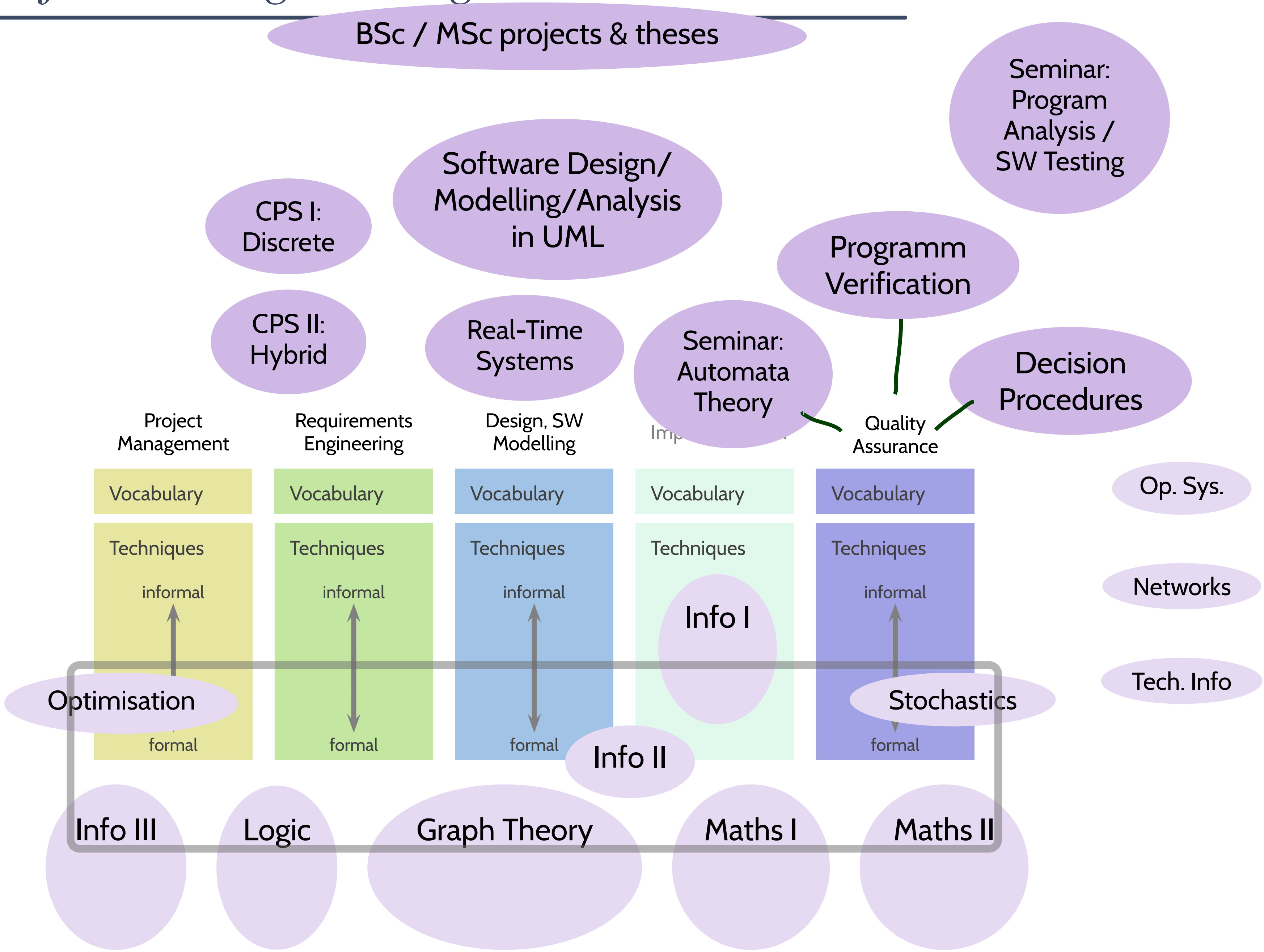
| | | |
|---|---|---|
| Introduction | L 1: | 18.4., Mon |
| Scales, Metrics, Costs | L 2: | 21.4., Thu |
| | L 3: | 25.4., Mon |
| | T 1: | 28.4., Thu |
| Development | L 4: | 2.5., Mon |
| | - | 5.5., Thu |
| Process | L 5: | 9.5., Mon |
| | L 6: | 12.5., Thu |
| | - | 16.5., Mon |
| | - | 19.5., Thu |
| | T 2: | 23.5., Mon |
| | - | 26.5., Thu |
| Requirements Engineering | L 7: | 30.5., Mon |
| | L 8: | 2.6., Thu |
| | L 9: | 6.6., Mon |
| | T 3: | 9.6., Thu |
| Architecture & Design | L10: | 13.6., Mon |
| | L 11: | 16.6., Thu |
| | L12: | 20.6., Mon |
| | T 4: | 23.6., Thu |
| Software Mondelling | L13: | 27.6., Mon |
| | L14: | 30.6., Thu |
| | L15: | 4.7., Mon |
| | T 5: | 7.7., Thu |
| Quality Assurance (Testing, Formal Verification) | L16: | 11.7., Mon |
| | L17: | 14.7., Thu |
| | L18: | 18.7., Mon |
| Wrap-Up | L19: | 21.7., Thu |

*That’s Today’s Software Engineering — More or Less…*

!
!
Software Testing
Jorgensen
Fourth Edition
Basiswissen Softwaretest
Spillner · Linz
PATTERN-ORIENTED SOFTWARE ARCHITECTURE
A System of Patterns
Volume 1
Buschmann Meunier Rohnert Sommerlad Stal
PATTERN-ORIENTED SOFTWARE ARCHITECTURE
Volume 2
Schmidt Stal Rohnert Buschmann
Object-Oriented Software Engineering
A Use Case Driven Approach
Jacobson
MODELING SOFTWARE BEHAVIOR
Jorgensen
Modellgetriebene Softwareentwicklung
Stahl · Völter
REQUIREMENTS Engineering
Kotonya Sommerville
PROCESSES AND TECHNIQUES
Basiswissen Requirements Engineering
Pohl · Rupp
REQUIREMENTS-ENGINEERING UND -MANAGEMENT
chris RUPP & die SOPHISTen
Requirements Engineering
Pohl
Softwaretechnik
Grechenig Bernhart Breiteneder Kappel
Software Engineering
Pfleeger Atlee
Fourth Edition
Software Engineering
Sommerville
PEARSON Studium
Lehrbuch der Software-Technik I
Software-Entwicklung
Helmut Balzert
2. Auflage
Lehrbuch der Software-Technik
Software-Management
Software-Qualitätssicherung
Unternehmensmodellierung
Helmut Balzert
Lehrbuch der Softwaretechnik
Entwurf, Implementierung, Installation, Betrieb
Balzert
3. Aufl.
Spektrum
Software Engineering 2
Bjørner
Software Engineering 1
Bjørner
Software Engineering 3
Bjørner
Software Engineering
Ludewig · Lichter
HB C.2 44 3. Ex.
CD 2.0 18 1.
CD 2.0 18 2.
CD 2.0 12
HB C.2 5
HB C.2 76
HB C.2 77
HB C.2 62
HB C.2 53
HB C.2 47 2.
HB C.2 47 1.
HB C.2 47 3.
SB F.1 30 1.Ex.

*Coming Soon to Your Local Lecture Hall…*

# Course Software-Engineering vs. Other Courses



BSc / MSc projects & theses

Seminar: Program Analysis / SW Testing

CPS I: Discrete

Software Design/ Modelling/Analysis in UML

Programm Verification

CPS II: Hybrid

Real-Time Systems

Seminar: Automata Theory

Decision Procedures

Project Management

Requirements Engineering

Design, SW Modelling

Imp

Quality Assurance

Vocabulary

Techniques

informal

formal

Op. Sys.

Networks

Tech. Info

Info I

Optimisation

Stochastics

Info II

Info III

Logic

Graph Theory

Maths I

Maths II

*Thursday, 2016-07-21, 1200 to 1400:*

*Plenary Tutorial 6 & Questions Session*

*in 101-0-026 (right here)*