# *Softwaretechnik / Software-Engineering*

# *Lecture 18: The Rest & Wrapup*

*2015-07-23*

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

Albert-Ludwigs-Universität Freiburg, Germany

---

## *Contents of the Block "Quality Assurance"*

(i) **Introduction and Vocabulary**
  - correctness illustrated
  - vocabulary: fault, error, failure
  - three basic approaches

(ii) **Formal Verification**
  - Hoare calculus
  - Verifying C Compiler (VCC)
  - over- / under-approximations

(iii) **(Systematic) Tests**
  - systematic test vs. experiment
  - classification of test procedures
  - model-based testing
  - glass-box tests: coverage measures

(iv) **Runtime Verification**

(v) **Review**

(vi) **Concluding Discussion**
  - Dependability Cases

| | | |
|---|---|---|
| Introduction | L 1: | 20.4., Mo |
| | T 1: | 23.4., Do |
| Development Process, Metrics | L 2: | 27.4., Mo |
| | L 3: | 30.4., Do |
| | L 4: | 4.5., Mo |
| | T 2: | 7.5., Do |
| | L 5: | 11.5., Mo |
| Requirements Engineering | - | 14.5., Do |
| | L 6: | 18.5., Mo |
| | L 7: | 21.5., Do |
| | - | 25.5., Mo |
| | - | 28.5., Do |
| | T 3: | 1.6., Mo |
| | - | 4.6., Do |
| | L 8: | 8.6., Mo |
| | L 9: | 11.6., Do |
| | L 10: | 15.6., Mo |
| | T 4: | 18.6., Do |
| Architecture & Design, Software Modelling | L 11: | 22.6., Mo |
| | L 12: | 25.6., Do |
| | L 13: | 29.6., Mo |
| | L 14: | 2.7., Do |
| | T 5: | 6.7., Mo |
| Quality Assurance | L 15: | 9.7., Do |
| | L 16: | 13.7., Mo |
| Invited Talks | L 17: | 16.7., Do |
| | T 6: | 20.7., Mo |
| Wrap-Up | L 18: | 23.7., Do |

---

## *Contents & Goals*

**Last Lecture:**

- Testing (test case, test suite, testing notions, coverage, etc.)

**This Lecture:**

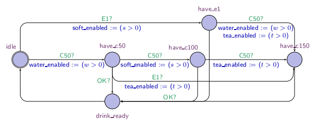
- **Educational Objectives:** Capabilities for following tasks/questions.
  - Give test cases for edge (location) coverage of a given CFA model.
  - What is runtime verification? What are examples?
  - How to conduct a review?
  - What are strengths and weaknesses of different quality assurance approaches (testing, formal verification, runtime verification, review, etc.)
- **Content:**
  - Model-based testing
  - Runtime-Verification
  - Review
  - Discussion of considered techniques
  - Dependability Cases

---

## *Model-Based Testing*

---

## *Model-based Testing*

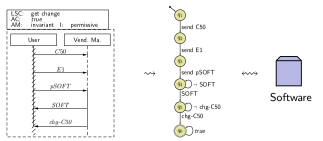


- Does some software **implement** the given CFA model of the CoinValidator?
- **One approach**: check whether **each state** of the model has some reachable **corresponding configuration** in the software.
  - $T_1 = (\text{C50}, \text{C50}, \text{C50};$
    $\{\pi \mid \exists\, i < j < k < \ell \bullet \pi^i \sim \text{idle}, \pi^j \sim \text{h\_c50}, \pi^k \sim \text{h\_c100}, \pi^\ell \sim \text{h\_c150}\})$

    checks: can we reach 'idle', 'have_c50', 'have_c100', 'have_c150'?
  - $T_2 = (\text{C50}, \text{C50}, \text{C50}; \dots)$ checks for 'have_e1'.
  - To check for 'drink_ready', more interaction is necessary.
- **Or**: Check whether **each edge** of the model has **corresponding** behaviour in the software.
- **Advantage**: input sequences can automatically be generated from the model.

---

## *Existential LSCs as Test Driver & Monitor* (*Lettrari and Klose, 2001*)



- If the LSC has designated **environment instance** lines, we can distinguish:
  - messages expected to originate **from** the environemnt (driver role),
  - messages expected adressed **to** the environemnt (monitor role).
- Adjust the TBA-construction algorithm to construct a **test driver & monitor** and have it (possibly with some **glue logic** in the middle) interact with the software (or a model of it).
- **Test passed** (i.e., test unsuccessful) if and only if TBA state $q_6$ is reached.
- We may need to **refine** the LSC by adding an activation condition, or communication which drives the system under test into the desired start state.

# *Statistical Testing*

---

## *Another Approach: Statistical Tests*

One proposal to deal with the **uncertainty of tests**, and to **avoid bias** (people tend to choose expected inputs): classical **statistical testing**.

- Randomly choose and apply test cases $T_1, \dots, T_n$,
  - **if an error is found**: good, we certainly know there is an error,
  - **if no error is found**:
    refuse hypothesis "program is not correct" with a certain confidence interval.

  (Significance niveau may be unsatisfactory with small numbers tests.)

(Ludewig and Lichter, 2013) name the following objections against statistical testing:

- E.g., for interactive software: primary goal is often that does **no failures are experienced by the "typical user"**. Statistical testing (in general) also cover a lot of "**untypical user behaviour**", unless **user-models** are used.
- Statistical testing needs a method to **compute "soll"-values** for the randomly chosen inputs; that is easy for "does not crash" but can be difficult in general.
- There is a high risk for **not finding point** or **small-range** errors — if they live in their **"natural habitat"**, carefully crafted test cases would probably uncover them.

Findings in the literature can at best be called **inconclusive**.

---

## *General "Do's" and "Don'ts"*

- **Do not** use special **examination versions** for examination.
  (Test-harness, stubs, etc. can be used; yet may have errors which may undermine results.)
- **Do not stop examination** when first error is detected.
  **Clear:** Examination can (and should) be aborted if the examined program is not executable at all.
- **Do not modify** the artefact under examination during examinatin.
  - changes/corrections during examination:
    in the end unclear **what exactly** has been examined ("moving target"),
    (results need to be uniquely traceable to one artefact version.)
  - fundamental flaws 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** have at least one (systematic) test for each feature — otherwise (**grossly**?) **negligent**.
  (Without at least one test for each feature, can it be called software **engineering**. . . ?)

---

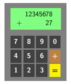
# *Run-Time Verification*

---

## *Run-Time Verification*

```
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();
  }
}
```



- If we have **an implementation** for checking
  whether an output is correct wrt. a given input (according to requirements),
- we can just **embed this implementation** into the actual software, and
- thereby **check satisfaction** of the requirement during **each execution**.

→ **run-time verification**.

---

## *Simplest Case: Assertions*

- Maybe the simplest instance of runtime verification: **Assertions**.
- Available in standard libraries of many programming languages, e.g. C:

```
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
       program.  The message "assertion failed in file foo.c, function
       do_bar(), line 1287" is of no help at all to a user.
```

# Simplest Case: Assertions

- Maybe the simplest instance of runtime verification: **Assertions**.
- Available in standard libraries of many programming languages, e.g. C:

```
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
       program.  The message "assertion failed in file foo.c, function
       do_bar(), line 1287" is of no help at all to a user.
```

- Assertions at work:

MAX_INT

```
int square( int x )
{
  assert( x < sqrt( x ) );

  return x * x;
}
```

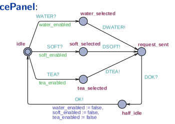
```
void f( ... ) {
  assert( p );
  ...
  assert( q );
}
```

```

int progress_bar_width( int progress, int window_left, int window_right )
{
  ...
  assert( 0 < progress && progress < 100); // extremal cases already treated
  ...
}
```

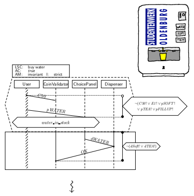
---

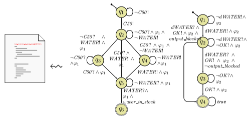
# More Complex Case: LSC Observer

**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();
          ;;
} }
```





---

# Run-Time Verification: Discussion

- **Experience**:

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

  - 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 this condition to hold, because. . . ".

- **Usually**:

  Development version **with** (cf. `assert(3)`) / release version **without** run-time verification.

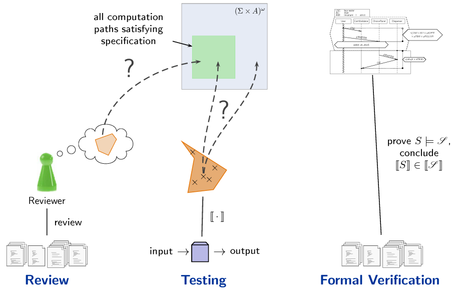
  If run-time verification enabled in release version,

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

- Run-time verification can be arbitrarily complicated and complex, e.g., construction of observers for LSCs or temporal logic, e.g., expensive checking of data, etc.
- **Drawback**: development and release software have different computation paths — with bad luck, the software only behaves well **because of** the run-time verification code. . .

---

# Recall: Three Basic Directions



$(\Sigma \times A)^\omega$

all computation paths satisfying specification

?

?

Reviewer

review

$[\![\cdot]\!]$

input → → output

prove $S \models \mathscr{S}$,

conclude $[\![S]\!] \in [\![\mathscr{S}]\!]$

**Review**

**Testing**

**Formal Verification**

---

# Review

---

# Reviews

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

  **Social aspect**: it is an **artefact** which is examined, not the **human** (who created it).

- **Input to Review Session**:
  - the **review item**, and **reference documents** which 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.

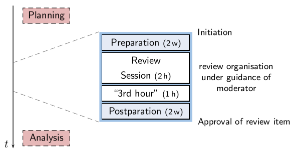
  **Author:** (representative of the) creator(s) of the artefact under review; is present to listen to the discussions, can answer questions; does not speak up if not asked.

  **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*



- review triggered, e.g., by submission to revision control system:
  moderator invites (include review item in invitation), state review missions,
- **preparation**: reviewers investigate review item,
- **review session**: reviewers report, evaluate and document issues; solve open questions,
- **"3rd hour"**: time for informal chat, reviewers may state proposals for solutions or improvements,
- **postparation**, rework: responsibility of author(s),
- reviewers re-assess reworked review item (until approval).
- **planning**: reviews need time in project plan; **analysis**: improve development and review process.

---

## *Review Rules (Ludewig and Lichter, 2013)*

(i) **moderator** organises, invites to, conducts review,

(ii) the review session is **limited to 2 hours** — if needed: more sessions

(iii) **moderator** may terminate review if conduction not possible (inputs, preparation, or people missing),

(iv) the **review item** is under review, not the author(s),
**reviewers** choose their wording accordingly,
**authors** neither defend themselves nor the review item,

(v) roles are **not mixed up**, the moderator does not act as reviewer,

(vi) **style** issues (outside fixed conventions) are not discussed,

(vii) the review team is **not** supposed to **develop solutions**,
issues are **not** noted 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) **review team** declares: accept **without changes**, accept **with changes**, do not accept.

(xii) **protocol** is signed by all participants.

---

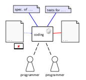
## *Weaker and Stronger Variants*

- **Careful Reading** ('Durchsicht')
  - done by developer,
  - recommendation: "away from screen" (use print-out or different device and situation)
- **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.
- **Structured Walkthrough**
  - simple variant of review: **developer** moderates walkthrough-session, presents artefact, reviewer pose (prepared or spontaneous) questions, issues are noted down,
  - variants: with or without preparation (do reviewers see the artefact before the session?)
  - less effort, less effective.

  **disadvantages**: unclear reponsibilities; "salesman"-author may trick reviewers.
- **Review**
- **Design and Code Inspection** (Fagan, 1976, 1986)
  - deluxe variant of review,
  - approx. 50% more time, approx. 50% more faults found.

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



---

## *Quality Assurance — Concluding Discussion*

---

## *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;
- provides reproducible counter-examples (good starting point for repair).

**Weaknesses:**
- (in most cases) **vastly non-exhaustive**, thus no proofs of correctness;
- creating test cases for complex functions (or complex conditions) can be difficult;
- maintaining many, complex test cases be challenging.
- executing many tests may need substantial time (but: can 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, not necessarily reproducible;
- (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:**
- 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

| | auto-matic | prove "can run" | toolchain considered | exhaus-tive | 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 (at the price of many false positives);
- 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);
- 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-matic | prove "can run" | toolchain considered | exhaus-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 difficult: failing to find a proof does not allow any useful conclusion;
- false negatives (broken program "proved" correct) hard to detect.

---

## Proposal: Dependability Cases (Jackson, 2009)

- A **dependable** system is one you can **depend** on — that is, you can place your trust in it.
- **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.
- IOW: "Developers [should] **express the critical properties** and **make an explicit argument** that the system satisfies them."

  (As opposed to, e.g. requiring **term coverage** (which is usually not exhaustive), or requiring **only** coding conventions and procedure models, which may support, but do not **prove** dependability.)

---

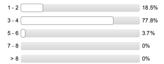
# Looking Back:
# 17.5 Lectures on Software Engineering

---

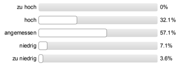
## Contents of the Lecture

| | | |
|---|---|---|
| Introduction | L 1: | 20.4., Mo |
| | T 1: | 23.4., Do |
| Development Process, Metrics | L 2: | 27.4., Mo |
| | L 3: | 30.4., Do |
| | L 4: | 4.5., Mo |
| | T 2: | 7.5., Do |
| | L 5: | 11.5., Mo |
| Requirements Engineering | - | 14.5., Do |
| | L 6: | 18.5., Mo |
| | L 7: | 21.5., Do |
| | - | 25.5., Mo |
| | - | 28.5., Do |
| | T 3: | 1.6., Mo |
| | - | 4.6., Do |
| | L 8: | 8.6., Mo |
| | L 9: | 11.6., Do |
| | L 10: | 15.6., Mo |
| | T 4: | 18.6., Do |
| Architecture & Design, Software Modelling | L 11: | 22.6., Mo |
| | L 12: | 25.6., Do |
| | L 13: | 29.6., Mo |
| | L 14: | 2.7., Do |
| | T 5: | 6.7., Mo |
| Quality Assurance | L 15: | 9.7., Do |
| | L 16: | 13.7., Mo |
| Invited Talks | L 17: | 16.7., Do |
| | T 6: | 20.7., Mo |
| Wrap-Up | L 18: | 23.7., Do |

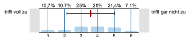
## Evalution Results



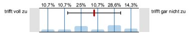
Anzahl Fachsemester (bezogen auf den aktuellen Studiengang)
(71.4 % B.Sc., 25 % M.Sc.)



Das inhaltliche Niveau der Veranstaltung ist. . .

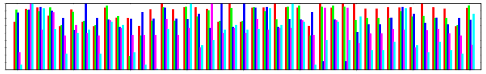


Ich habe in dieser Lehrveranstaltung viel gelernt.



Ich habe meine Fähigkeiten im wissenschaftlichen Problemlösen verbessert.

---



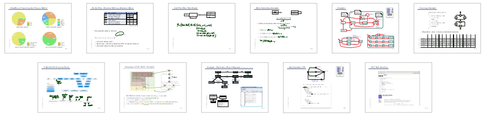
---

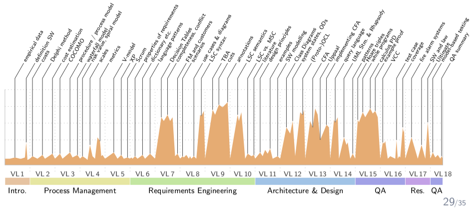
## Conclusion?

[...] in the end it's anyway only a lot of "blabla" without real **right** or **wrong**.

---

## What Did We Do?



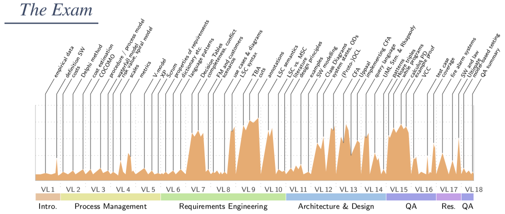


---

That's Today's Software Engineering — More or Less. . .

---

# The Exam



1,0 — 90
84

4,0 — 36
30

Proc. Models
- V-Model
- Metric
- Scales

RE
- DT
- UC
- ...

LSC
- TBA
- runs
...

CD/OD
OCL
...

CFA
- Trans. graph
- deadlock
...

Hoare
- par/tot
- DP calculus
- while
...

Test
- coverage
- exhaust.
- MBT
...

90

---

- Points for Ex. 6: Tue

- ADVERTISEMENT:
  - Project, BSc thesis, MSc thesis
  - UML lecture in Winter

---

# 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.

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

Lettrari, M. and Klose, J. (2001). Scenario-based monitoring and testing of real-time UML models. In Gogolla, M. and Kobryn, C., editors, *UML*, number 2185 in Lecture Notes in Computer Science, pages 317–328. Springer-Verlag.

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