# Advanced Topics of Software Engineering (ASE)

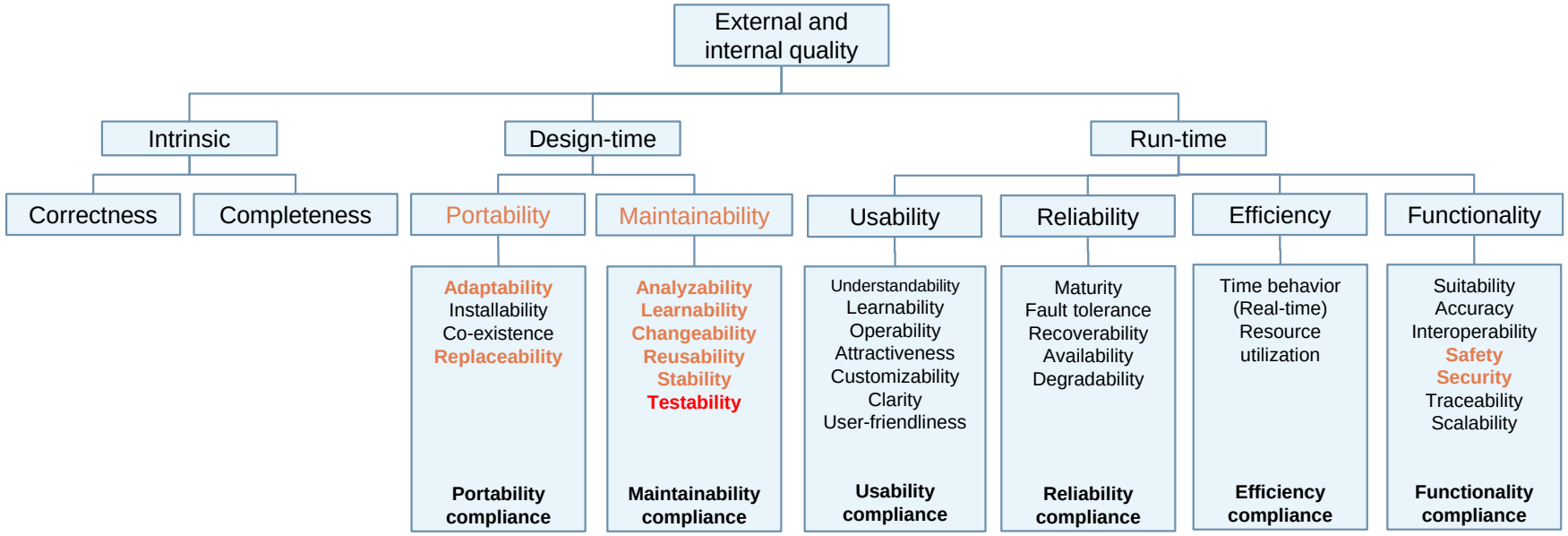
## Chapter 2. From requirements to system design (Part II)

Prof. Dr. Florian Matthes, Prof. Dr. Alexander Pretschner

Chair of Software Engineering for Business Information Systems (sebis)
Faculty of Informatics
Technische Universität München
wwwmatthes.in.tum.de

# From requirements to system design

# Testability has a positive impact on several other software quality attributes



# Problem

- What is a good test case?

One that finds a potential bug with good *cost* effectiveness

- Cost relates to designing the test, executing the test (often expensive for hardware like cars!), evaluating the result, but also the cost of localizing the problem in the code, and that of remaining bugs in the field.

- This obviously is not just a property of the test case but also the system to be tested.

- That said, are there properties of a program that make it easier to find problems?

- The following is to stimulate ideas. At present, more academic than practical.

# Testability?

- Put another way, if there is a specification and two programs written against that specification, does testing both programs involve the same effort?
- Measure difficulty by the number of tests that need to be applied in order to check if a system is correct.
- For instance, if one program is OO and one is written in C, then there may be problems related to dynamic binding that cannot happen in the C program. Vice versa, buffer overflows are not that common in Java programs.
- Presented ideas deliberately more academic than practical.

# Difficult problems

- **Testing GUIs**
  - How to observe? On different screens/machines? JavaScript code with dynamic IDs of the objects?
  - How to control? Possible if we know the (static) IDs of the GUI elements; use a tool like Selenium.

- **Testing systems with databases**
  - Testing a database in most cases means we change the state. That makes it difficult to reproduce the test – because the initial condition of the test case has changed.

- **Testing cloud-based systems**
  - Similar problem. Now the state is determined by the services in the cloud that we have contacted – how do we control/observe those?

- **Testing non-deterministic systems**
  - Again, similar problem with different origin: applying the same input may yield different outputs.

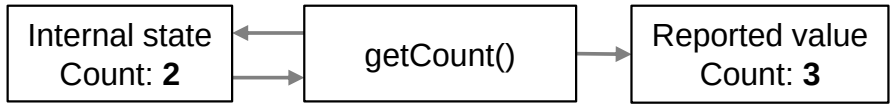
- **Testing embedded systems**
  - What is the PIN counter of a chip card?

# Example for a testability problem

Consider a chip card. If you incorrectly enter your PIN three times, the PIN is blocked. You then need to enter a PUK (personal unblocking key). The card has an internal counter for the number of incorrectly entered PINs.

(Chip cards are computers based on a von Neumann architecture - like laptops without a display. That is, you send them commands, they perform computations, and return a value. State is encoded by memory.)

- Smartcard manufacturers need to make sure that blocking PINs actually works.
- There is a command that retrieves the current PIN counter.
- However, how do you know that the state of the card corresponds with the number returned by this command?
- It could well be the case that even though the reported counter value is 3, it is still possible to enter a PIN again.




What's the problem here?

- The internal state allows another PIN to be entered.
- Then, there is some value given to the outside that may – but need not necessarily – reflect this state.

# Observability and Controllability

The previous example suggests that testability is positively correlated with

**Observability**:
Can everything that we care about be observed? For instance, parts of the internal state (e.g., diagnosis interfaces in cars)?

**Controllability:**
Can we directly modify (parts of) the state of the program in a "sufficiently easy" way without the need to cook up intricate interactions with the system?

# Testability (1)

First try: Testability measures the degree of difficulty to develop for a program *P* a test suite satisfying some test selection criterion *C*.

Testability hence depends on both *P* and *C*. (It also depends on the quality of the requirements because these are necessary to state the expected test results.)

Simple example: It is simpler to achieve branch coverage than to achieve modified condition/decision coverage. Testability of *P* for the former is usually higher than for the latter. (It is the same if *P* does not contain composite conditions).

Testability

- The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met.
- The degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met.

[IEEE Standard Glossary of Software Engineering Terminology. (1990)]

# Testability (2)

- Note that this definition somehow is a “white-box definition”. If tests are derived purely on the grounds of the specification, i.e., in a black-box way, their selection is independent of the code, and so is the “difficulty” of this selection.

- Also note that we are ultimately interested in is failure-free functioning of the code, or at least functioning without "bad" or "costly" failures.

- Not all test selection criteria necessarily correlate with bug detection likelihood (i.e.: higher degree of satisfying the test selection criterion does not necessarily imply higher likelihood of bug detection). In particular, this is the case for coverage-based testing, at least when coverage is used as test selection criterion.

For more attend "Advanced Topics of Software Testing (IN2256)" lecture.

# Testability (3)

Intuitively, code "may hide defects" which are then difficult to find using testing.

> Testability is the likelihood of a program to fail with the next test (given a particular assumed input distribution) if the software includes a bug.

["Software Testability: The New Verification." Voas and Miller. (1995)]

*(Put another way: Once we have derived our tests according to some test selection criterion (the input distribution), then the effectiveness of these tests to reveal defects defines the testability of a system.*

*Interesting. I rather think this defines the quality of tests, not that of systems.)*

# Consequences

If poor testability has been shown for some modules, subject them to other verification and validation (V&V) techniques (reviews, formal verification).

# How to assess? One approach.

One idea is to cast the problem is one of "information loss". Information loss occurs if the values computed in a program are not propagated to the output.

- Happens at the operator level if many input values yield the same output values (implicit loss).
- Happens also if the value of local variables is not checked after execution of a module (explicit loss).

Domain/range ratio (DRR) correlates with implicit losses:

- Cardinality of the input domain divided by the cardinality of the output domain.
- If |*input domain*| >> |*output domain*|, then testability is poor. (This information is not always accessible.)
- [Note that this is black box again.]

- See [“Factors that Affect Testability”, Voas (1991)]

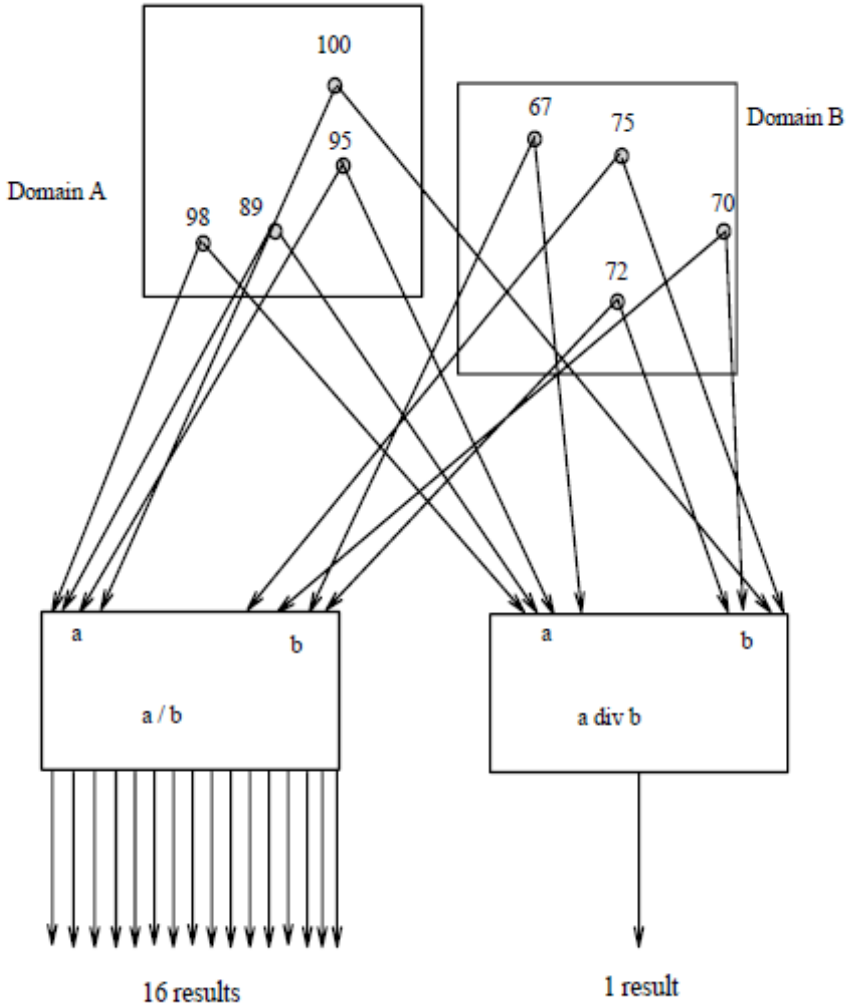
# Example for implicit losses

Real division vs. Integer division:

- Four possible values for A and four possible values for B.
- 16 possible real results and one possible integer result.

Real division intuitively easier to test because we have “more information,” or “finer information”, on the output: In a sense, “more can go wrong”.

Implicit losses depend, to a large extent, on the functional specification.



# What to do?

- **Implicit** losses at the system level depend on the functional specification. There is not too much we can do.
  - But we can design single modules with small(ish) or large(ish) DRRs.
  - When testing, we then concentrate on modules with large DRRs.

- Avoid re-use of variables. A:=f(A) necessarily means implicit loss (unless f=id), so use MORE rather than FEWER variables.

- **Explicit** losses depend more on how the software is designed.
  - One solution: increase the size of the output domain by explicit observations, i.e., the number of return values, and hence increase the degree of observability.

- Controllability adds more possibilities not just to read, but also write the program's internal state and just more directly test single modules in their context.

# Tradeoffs

- This means that we need to provide information about the inner workings of modules – which contradicts many ideas relating to modularity, encapsulation, and information hiding (!!!)

- If time-to-market is all that matters, people won't have time to think about testability

# By the way: measuring testability?

One possibility:

- Inject defects into a program.
- When executed, these defects should change the inner state.
- Then perform random testing (or testing with a specific usage profile) and see if the change of the program yields different output.
- Measure the likelihood of an injected defect changing the output.
- **High likelihood = high testability**.

- Remember one earlier definition:

> Testability is the likelihood of a program to fail with the next test (given a particular assumed input distribution) if the software includes a bug.

The idea is related to mutation testing [DeMillo et al. 1978] where the quality of a test suite is determined by checking if an existing test suite would catch an injected defect.

# Testability and effort

Mouchawrab et al. hypothesize that testability (IEEE definition) correlates with properties of code, contracts (specifications), and requirements, see table, and suggest metrics for these attributes.

IMO very sensible ideas, AFAIK no empirical backing

| | | Specifying test cases | Developing a driver | Developing a stub | Developing an oracle |
|---|---|---|---|---|---|
| U | Unit Size | | | | |
| | Local features (1) | + | + | | + |
| | Inherited features (2) | + | + | | + |
| | Unit Cohesion (6) | - | - | | - |
| | Operations sequential constraints complexity (9) | + | | | |
| | State behavior complexity | | | | |
| | Paths complexity (10) | + | + | | + |
| | Guard condition complexity (11) | + | + | | |
| | State invariant complexity (12) | | | | + |
| | Action complexity (13) | | | | + |
| | Inheritance design properties | | | | |
| | Compliance with LSP (3) | - | - | | - |
| | Inherited and overridden features interaction (4) | + | + | | + |
| U, I | Unit Coupling (7) | | | + | |
| U, I, S | Contracts complexity (8) | + | + | + | + |
| I | Inheritance design properties | | | | |
| | Size of inheritance hierarchies (5) | + | + | | + |
| | Structure complexity | | | | |
| | Dependency paths (14) | | + | + | + |
| | Dependency cycles (15) | | | + | |
| | Redundant paths (16) | + | + | | + |
| S | Use case complexity | | | | |
| | Scenario path complexity (17) | + | + | | + |
| | Scenario condition complexity (18) | | + | | |
| | Use cases sequential constraints complexity (19) | + | + | | |
| | System interface complexity (20) | + | + | + | + |

["A Measurement Framework for OO Software Testability", Mouchawrab, Briand, Labiche (2005)]

# Discussion

- Essentially, yet unsolved problem. Presented ideas are academic.
- At the same time, huge practical concern.

- Observability and controllability seem to be crucial but break information hiding.

- Discussion so far did not focus on different stages of testing, specifically unit and integration tests. This is where the architecture enters the picture, once again.

- Intuitively:
  - Little information hiding means lots of required testing at the integration level.
  - Weak cohesion means that functional testing requires testing (too) large subsystems
  - Strong coupling makes unit testing more difficult (stubs!)
  - …