TDDD04: System level testing

Lena Buffoni lena.buffoni@liu.se

LINKÖPING
UNIVERSITY

Lecture plan

- System testing
  - Thread testing
  - Test automation
  - Model-based testing

Thread-based testing

Examples of threads at the system level

- A scenario of normal usage
- A stimulus/response pair
- Behavior that results from a sequence of system-level inputs
- An interleaved sequence of port input and output events
- **A sequence of MM-paths**
- **A sequence of atomic system functions (ASF)**

Atomic System Function (ASF)

- An *Atomic System Function(ASF)* is an action that is observable at the system level in terms of port input and output events.
- A system thread is a path from a **source** ASF to a **sink** ASF

Examples

Stimulus/response pairs: entry of a personal identification number

- A screen requesting PIN digits
- An interleaved sequence of digit keystrokes and screen responses
- The possibility of cancellation by the customer before the full PIN is entered
- Final system disposition (user can select transaction or card is retained)

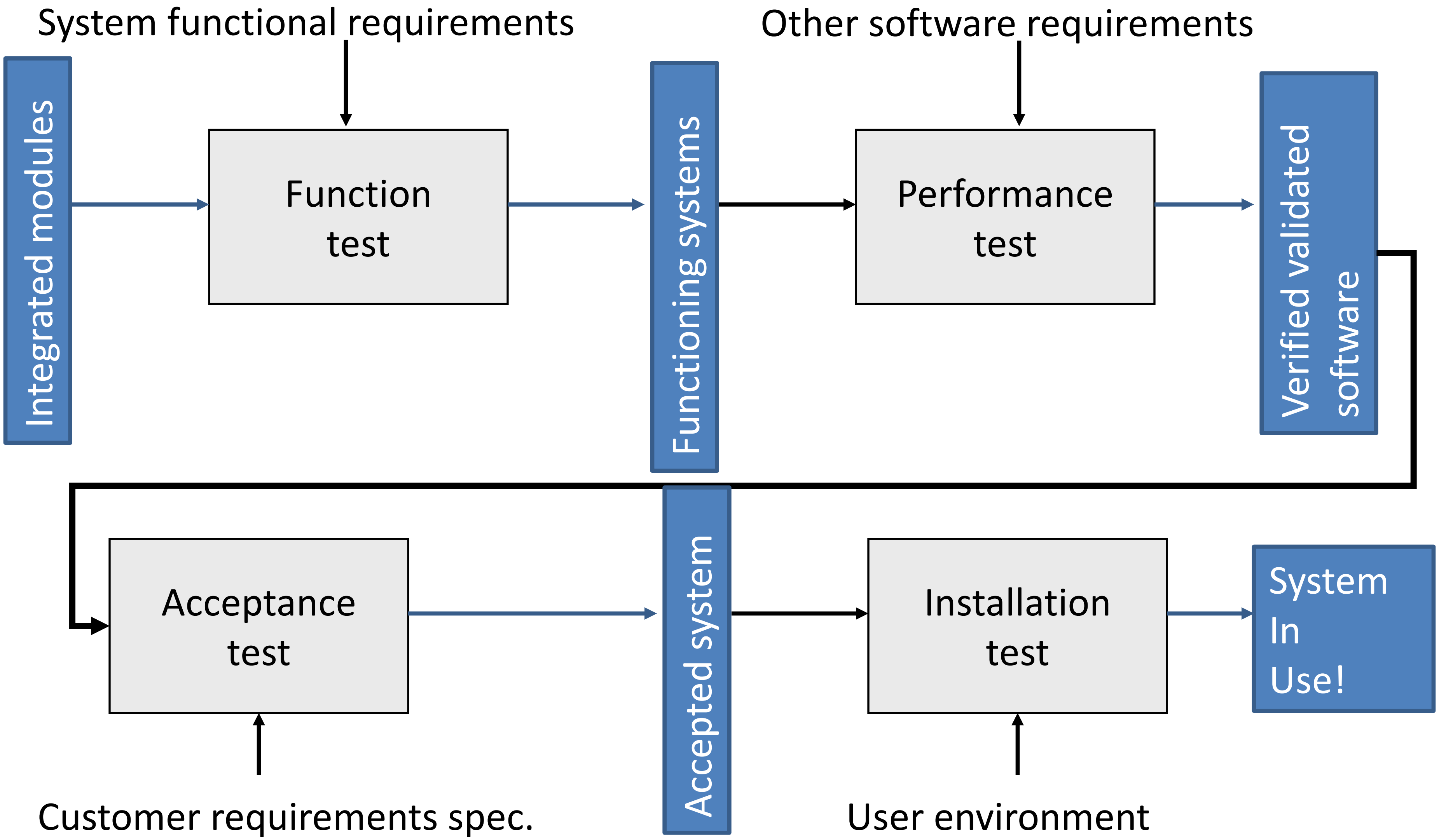
Sequence of atomic system functions

- A simple transaction: ATM Card Entry, PIN entry, select transaction type (deposits, withdraw), present account details (checking or savings, amount), conduct the operation, and report the results (involves the interaction of several ASFs)
- An ATM session (a sequence of threads) containing two or more simple transactions (interaction among threads)

Thread-based testing strategies

- Event-based

  Coverage metrics on input ports:
  - Each port input event occurs
  - Common sequences of port input events occur
  - Each port event occurs in every relevant data context
  - For a given context all inappropriate port events occur
  - For a given context all possible input events occur
- Port-based
- Data-based
  - Entity-Relationship (ER) based



System functional requirements

Integrated modules

Function test

Functioning systems

Other software requirements

Performance test

Verified validated software

Acceptance test

Customer requirements spec.

Accepted system

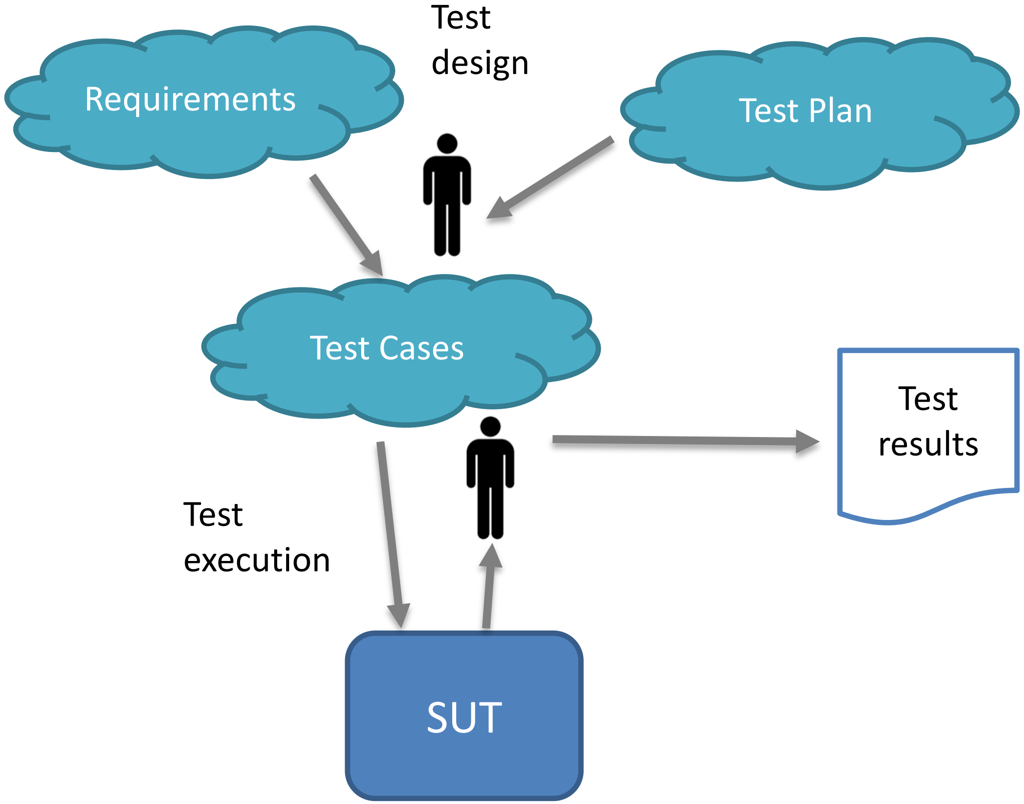
Installation test

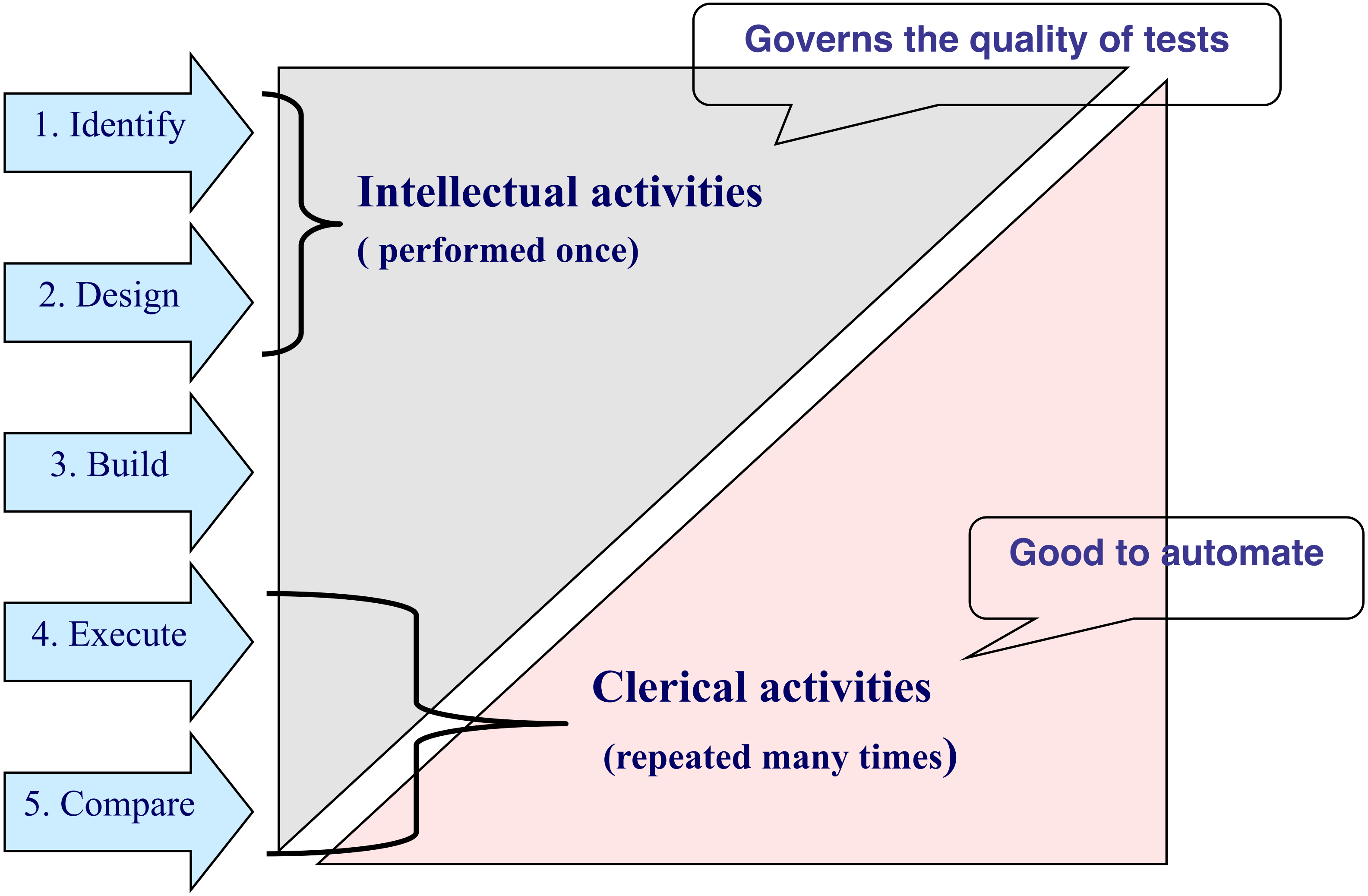
User environment

System In Use!

Test automation

Why automate tests?





Governs the quality of tests
1. Identify
2. Design
3. Build
4. Execute
5. Compare
Intellectual activities
(performed once)
Good to automate
Clerical activities
(repeated many times)

Test outcome verification

- Predicting outcomes – not always efficient/possible
- Reference testing – running tests against a manually verified initial run
- How much do you need to compare?
- Wrong expected outcome -> wrong conclusion from test results

Sensitive vs robust tests

- **Sensitive tests** compare as much information as possible – are affected easily by changes in software

- **Robust tests** – less affected by changes to software, can miss more defects

Limitations of automated SW testing

- Does not replace manual testing
- Not all tests should be automated
- Does not improve effectiveness
- May limit software development

Can we automate test case design?

Automated test case generation

- Generation of test input data from a domain model
- Generation of test cases based on an environmental model
- Generation of test cases with oracles from a behaviors model
- Generation of test scripts from abstract test

Impossible to predict output values

Model-based testing

Model-based testing

Generation of **complete test cases** from models of the SUT

- Usually considered a kind of black box testing
- Appropriate for **functional testing** (occasionally robustness testing)

Models must **precise** and should be **concise**

- **Precise** enough to describe the aspects to be tested
- **Concise** so they are easy to develop and validate
- Models may be developed specifically for testing

Generates **abstract test cases** which must be transformed into **executable test cases**

What is a model?

Mapping

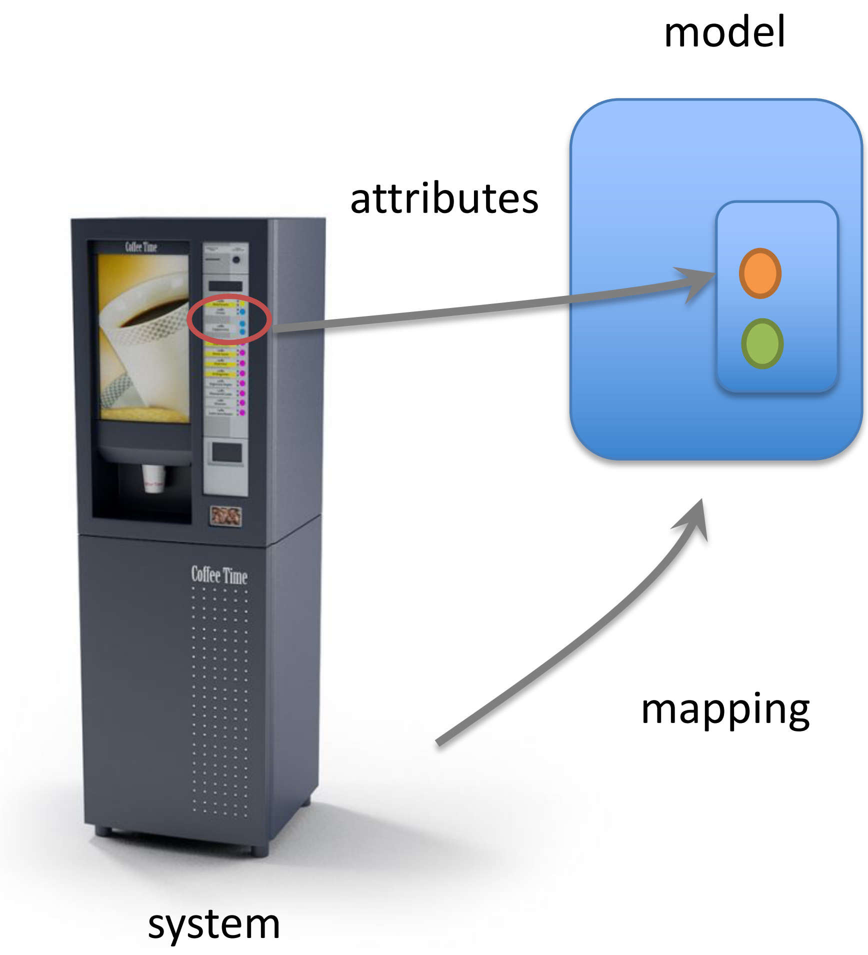
- There is an original object that is mapped to a model

Reduction

- Not all properties of the original are mapped, but some are

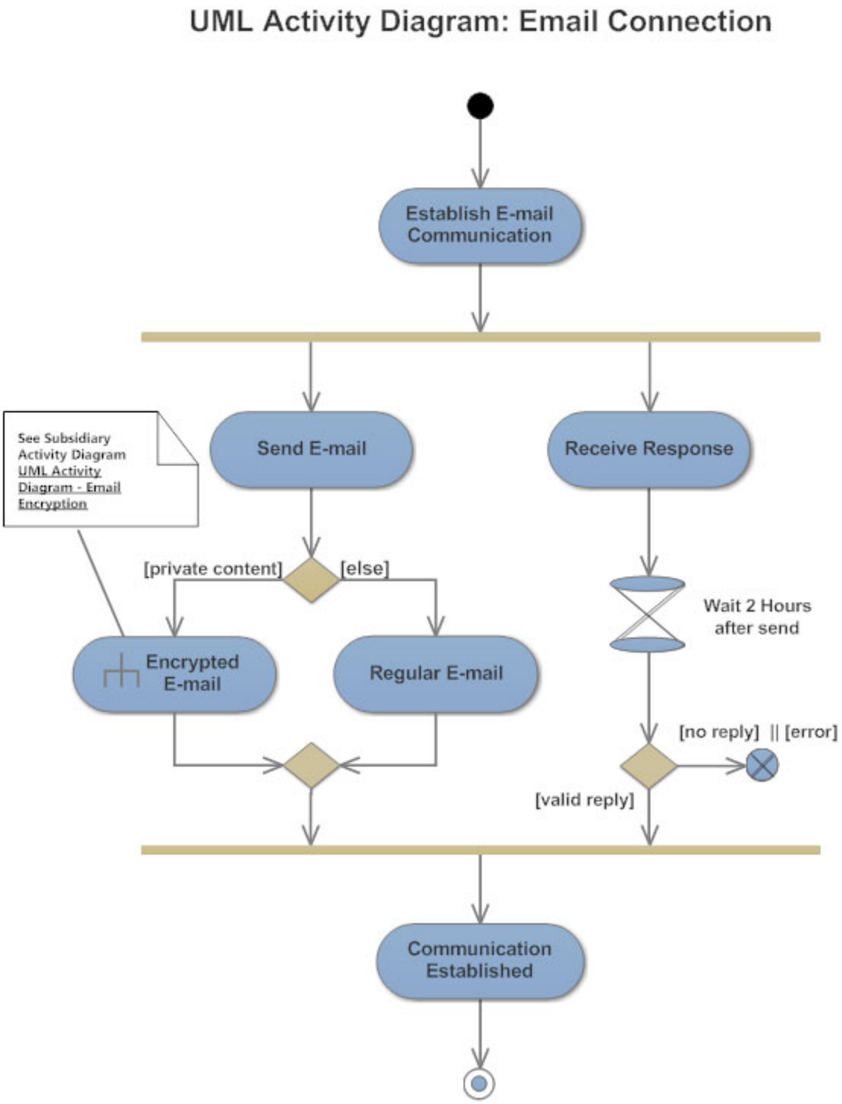
Pragmatism

- The model can replace the original for some purpose



Example model: UML activity diagram

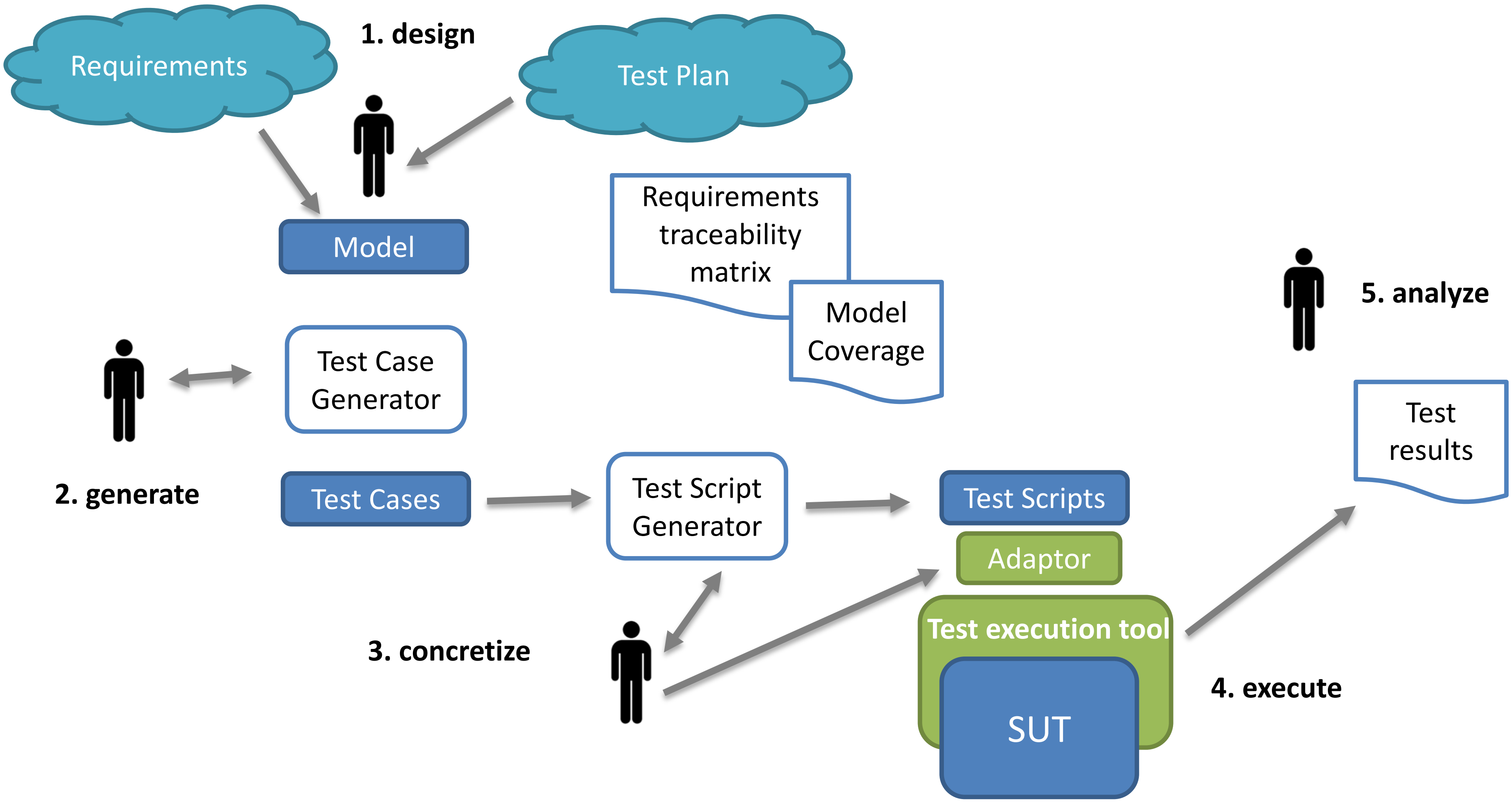
- Original object is a software system (mapping)
- Model does not show implementation (reduction)
- Model is useful for testing, requirements (pragmatism)


UML Activity Diagram: Email Connection
Establish E-mail Communication
See Subsidiary Activity Diagram UML Activity Diagram - Email Encryption
Send E-mail
Receive Response
[private content]
[else]
Wait 2 Hours after send
Encrypted E-mail
Regular E-mail
[no reply] || [error]
[valid reply]
Communication Established

How to model your system?

- Focus on the SUT
- Model only subsystems associated with the SUT and needed in the test data
- Include only the operations to be tested
- Include only data fields useful for the operations to be tested
- Replace complex data fields by simple enumeration

Model based testing



Model-based testing steps

1. Model the SUT and/or its environment
2. Use an existing model or create one for testing
3. Generate abstract tests from the model
   - Choose some test selection criteria
   - The main output is a set of abstract tests
   - Output may include traceability matrix (test to model links)
4. Concretize the abstract tests to make them executable
5. Execute the tests on the SUT and assign verdicts
6. Analyze the test results.

Notations

Pre/post notations: system is modeled by its internal state

- UML Object Constraint Language (OCL), B, Spec#, JML, VDM, Z

Transition-based: system is modeled as transitions between states

- UML State Machine, STATEMATE, Simulink Stateflow

History-based: system described as allowable traces over time

- Message sequence charts, UML sequence diagrams

Functional – system is described as mathematical functions

Operational – system described as executable processes

- Petri nets, process algebras

Statistical – probabilistic model of inputs and outputs

Pre/post example (JML)

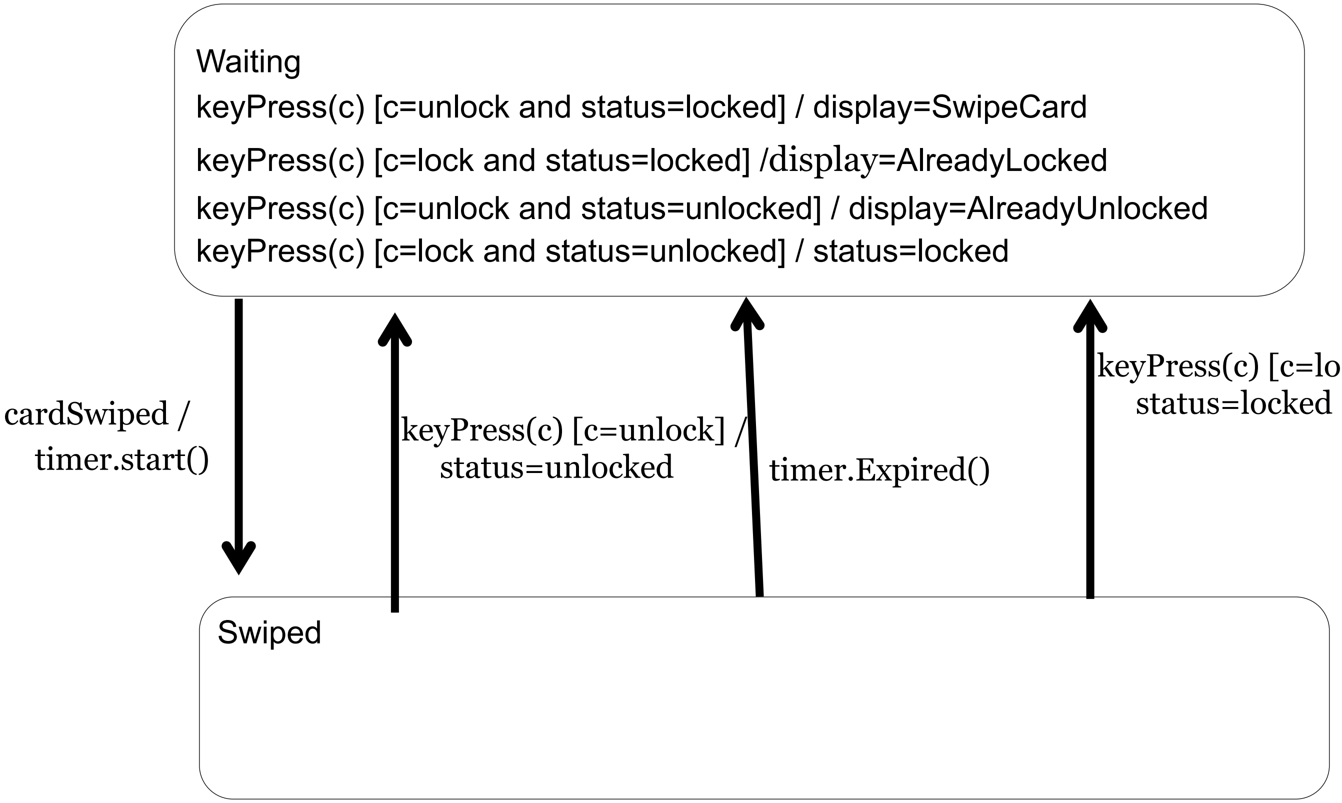
```
/*@ requires amount >= 0;
        ensures balance == \old(balance-amount)
        && \result == balance;

@*/
public int debit(int amount) {

…

}
```

Robustness testing

- Selecting unauthorized input sequences for testing
  - Format testing
  - Context testing

- Using defensive style models

Transition-based example (UML+OCL)



Waiting

keyPress(c) [c=unlock and status=locked] / display=SwipeCard

keyPress(c) [c=lock and status=locked] /display=AlreadyLocked

keyPress(c) [c=unlock and status=unlocked] / display=AlreadyUnlocked

keyPress(c) [c=lock and status=unlocked] / status=locked

cardSwiped /
timer.start()

keyPress(c) [c=unlock] /
status=unlocked

timer.Expired()

keyPress(c) [c=lock] /
status=locked

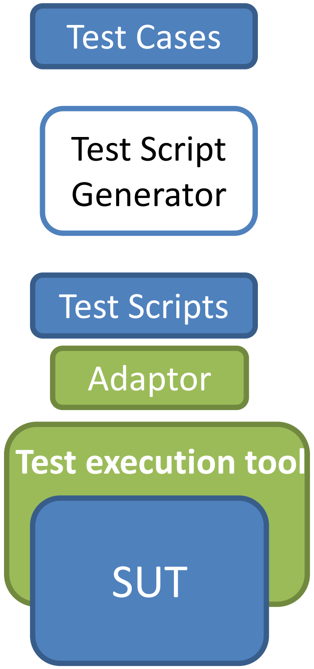
Swiped

Generate abstract test cases

- Transition-based models
  Search for sequences that result in e.g. transition coverage **Example (strategy – all transition pairs)**
  Precondition: status=locked, *state* = Waiting

Event	Exp. state	Exp. variables
cardSwiped	Swiped	status=locked
keyPress(lock)	Waiting	status=locked
cardSwiped	Swiped	status=locked
keyPress(unlock)	Waiting	status=unlocked

Concretize test cases

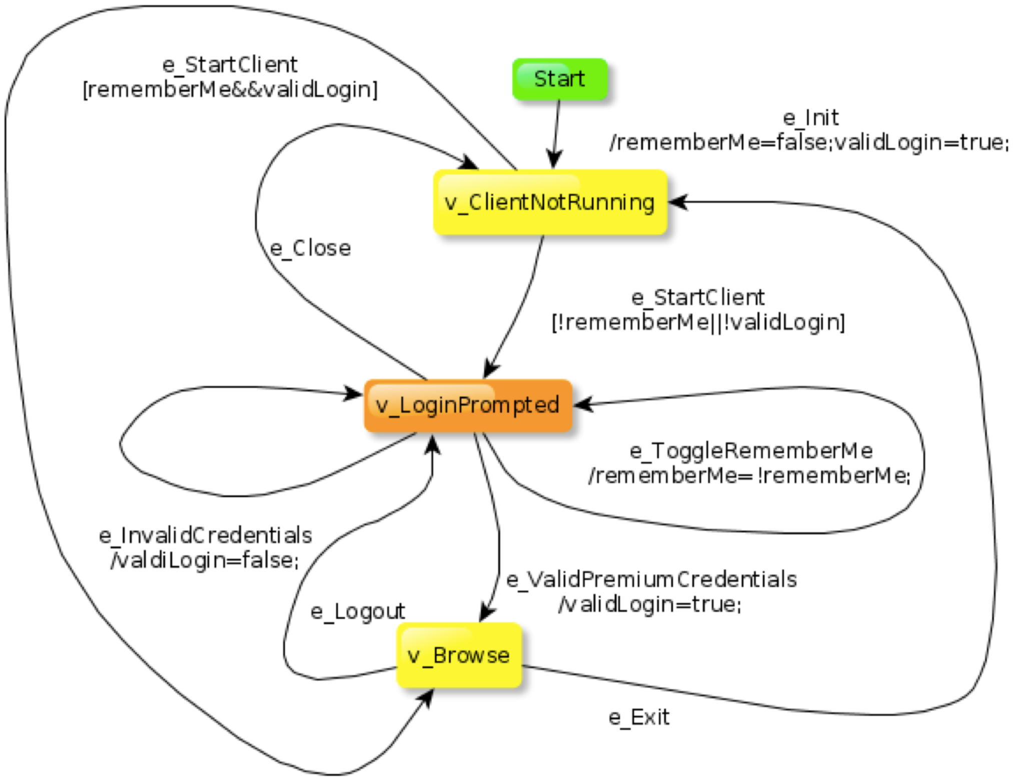


Analyze the results

- Same as in any other testing method
- Must determine if the fault is in the SUT or the model (or adaptation)
- May need to develop an oracle manually

GraphWalker is an Model-Based testing tool. It reads models in the shape of directed graphs, and generate [test] paths from these graphs.

A model can look like the one to the next. The model is collection of arrows and nodes and together they create a graph.



- An arrow represents an action.
- A node represents a verification.

Benefits of model-based testing

- Effective fault detection
  - Equal to or better than manually designed test cases
  - Exposes defects in requirements as well as faults in code
- Reduced Testing cost and time
  - Less time to develop model and generate tests than manual methods
  - Since both data and oracles are developed tests are very cheap
- Improved test quality
  - Can measure model/requirements coverage
  - Can generate very large test suites
- Traceability
  - Identify untested requirements/transitions
  - Find all test cases related to a specific requirement/transition
- Straightforward to link requirements to test cases
- Detection of requirement defects

Limitations

- Fundamental limitation of testing: won’t find all faults
- Requires different skills than manual test case design
- Mostly limited to functional testing
- Requires a certain level of test maturity to adopt
- Possible “pain points”
  - Outdated requirements – model will be incorrect!
  - Modeling things that are hard to model
  - Analyzing failed tests can be more difficult than with manual tests
  - Testing metrics (e.g. number of test cases) may become useless

Non functional testing

Performance Testing nonfunctional requirements

- Stress tests
- Timing tests
- Volume tests
- Configuration tests
- Compatibility tests
- Regression tests
- Security tests

- (physical) Environment tests
- Quality tests
- Recovery tests
- Maintenance tests
- Documentation tests
- Human factors tests / usability tests

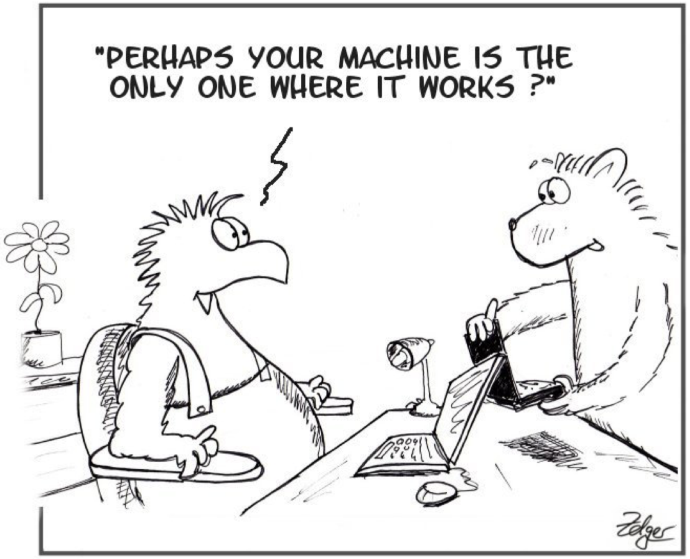
Non functional testing is mostly domain specific

Regression testing

- Re-executing old tests to ensure changes in software do not generate new failures

- Incidence matrix between features and implementation modules

Acceptance Testing



It works on my machine

Benchmark test: a set of special test cases

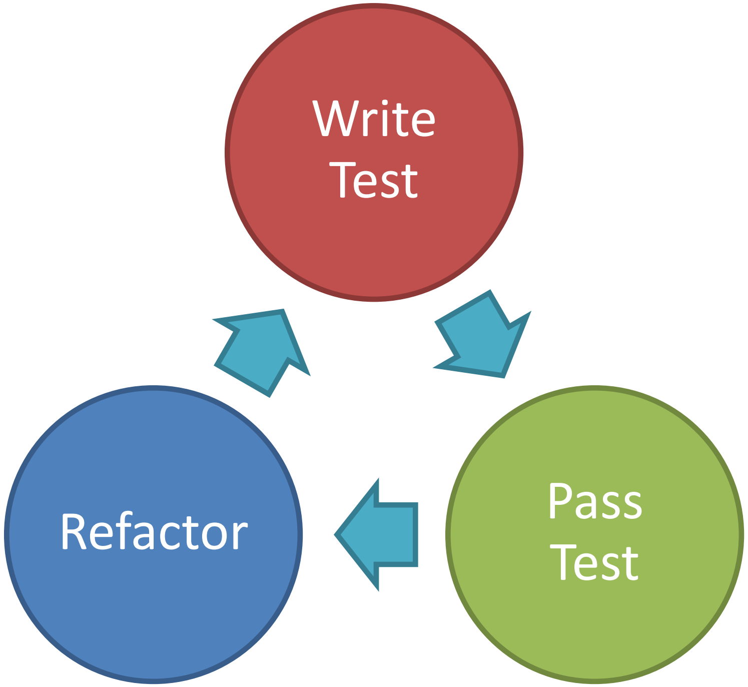
Pilot test: everyday working

Alpha test: at the developer's site, controlled environment

Beta test: at one or more customer site.

Parallel test: new system in parallel with previous one

Test-driven development



- Guided by a sequence of user stories from the customer/user
- Needs test framework support (eg: Junit)

NextDate:

User Stories

1: the program compiles

2: a day can be input and displayed

2: a month can be input and displayed

TEST

Input	Expected Output
Source Code	OK
15	Day = 15

15, 11	Day = 15
	Month = 11

Code

```
Program NextDate
End NextDate
```

```
Program NextDate
        input int thisDay;
        print (“day =“ + thisDay);
End NextDate
```

```
Program NextDate
        input int thisDay;
        input int thisMonth;
        print (“day =“ + thisDay);
        print (“month =“ + thisMonth) ;
End NextDate
```

Pros and cons

+ working code

+ regression testing

+ easy fault isolation

+ test documented code

- code needs to be refactored

- can fail to detect deeper faults

Evaluating a test suite

- Number of tests?
- Number of passed tests?
- Cost/effort spent?
- Number of defects found?

Defect Detection Percentage = defects found by testing / total known defects

When to stop testing : coverage criteria

- Structural coverage criteria
- Data coverage criteria
- Fault-mode criteria
- Requirements based criteria
- Explicit test case specification
- Statistical test generation methods

When to stop testing?

No single criterion for stopping, but…

- previously defined coverage goals are met
- defect discovery rate has dropped below a previously defined threshold
- cost of finding “next” defect is higher than estimated cost of defect
- project team decides to stop testing
- management decides to stop testing
- money/time runs out

Thank you!

Questions?

LINKÖPING UNIVERSITY