Model-Based Testing

(DIT848 / DAT260)

Spring 2015

Lecture 7
Introduction to MBT

Gerardo Schneider
Department of Computer Science and Engineering
Chalmers | University of Gothenburg

Many slides based on material provided by Mark Utting

What have we seen

- V&V: Validation & Verification
  - The V model
  - Black box testing
  - White box testing
  - Something on coverage
- (Extended) Finite State Machines

What remains

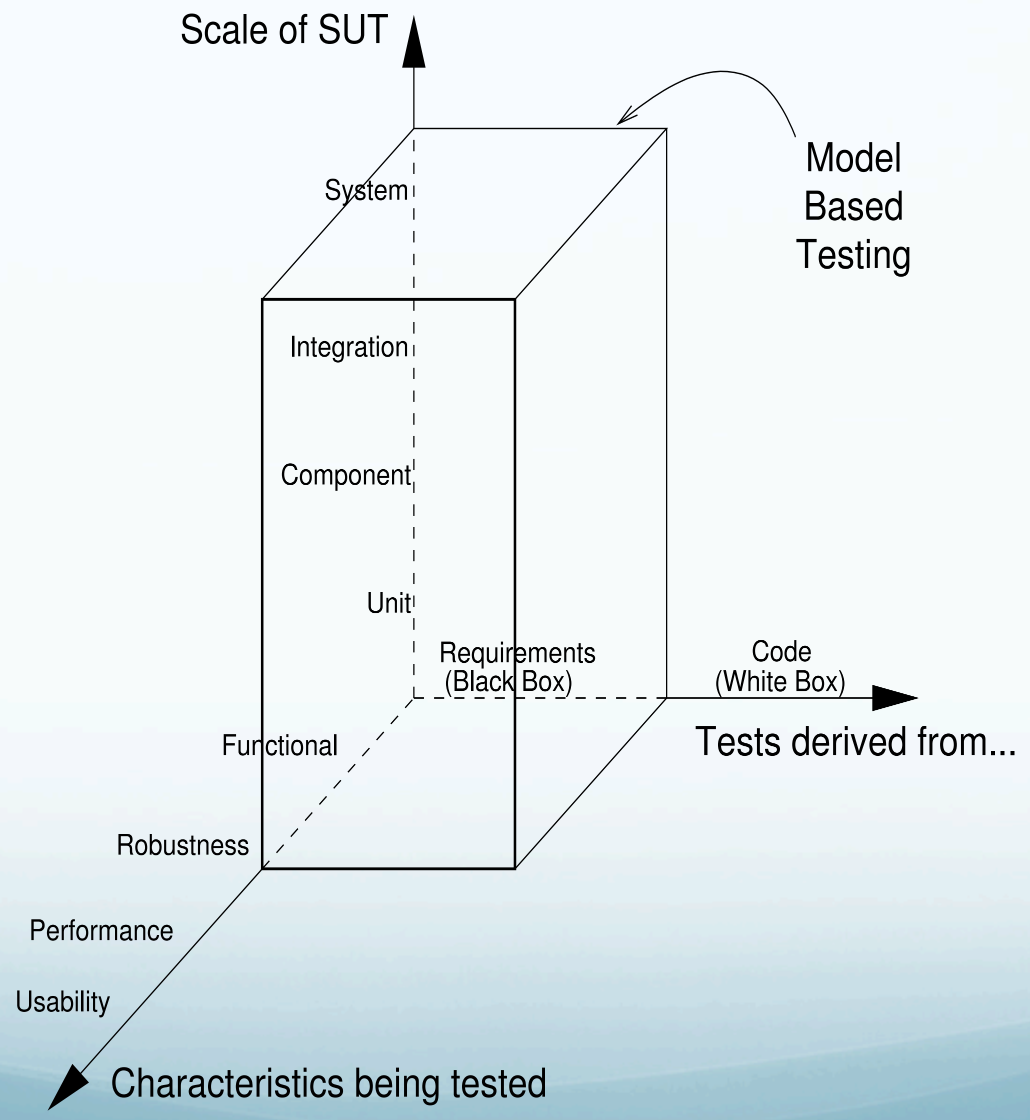
The rest of the lectures: MBT

1. Introduction (concepts, terminology,...) – Today
2. How to select your tests – Today
3. Graph theory for MBT – Wed this week
4. ModelJUnit – Wed this week
5. Making your tests executable – Wed this week
6. QuickCheck (by Koen Claessen)

NEW: *Guest lectures!!*

- Runtime Verification (Mon May 11)
- Behavoural-Driven Development (Wed May 13)

Kinds of Testing



Source: M. Utting and B. Legeard, *Practical Model-Based Testing*

What is Model-Based Testing

Four main approaches known as MBT

1. Generation of test input data from a domain model
   - Information on the domain of input values
   - Not known whether test passess or not
2. Generation of test cases from an environmental model
   - Environment: expected usage of SUT, operation frequences…
   - Do not specify expected output
3. Generation of test scripts from abstract tests
   - Abstract description of test cases (eg. UML seq. Diag.)
   - Transforms abstract test cases into low-level executable script
4. Generation of test cases with oracles from a behavior model
   - Executable tests with expected output
   - Model must describe expected behavior of SUT

Our focus!

So… MBT is the automation of the design of black-box tests

MBT in context...

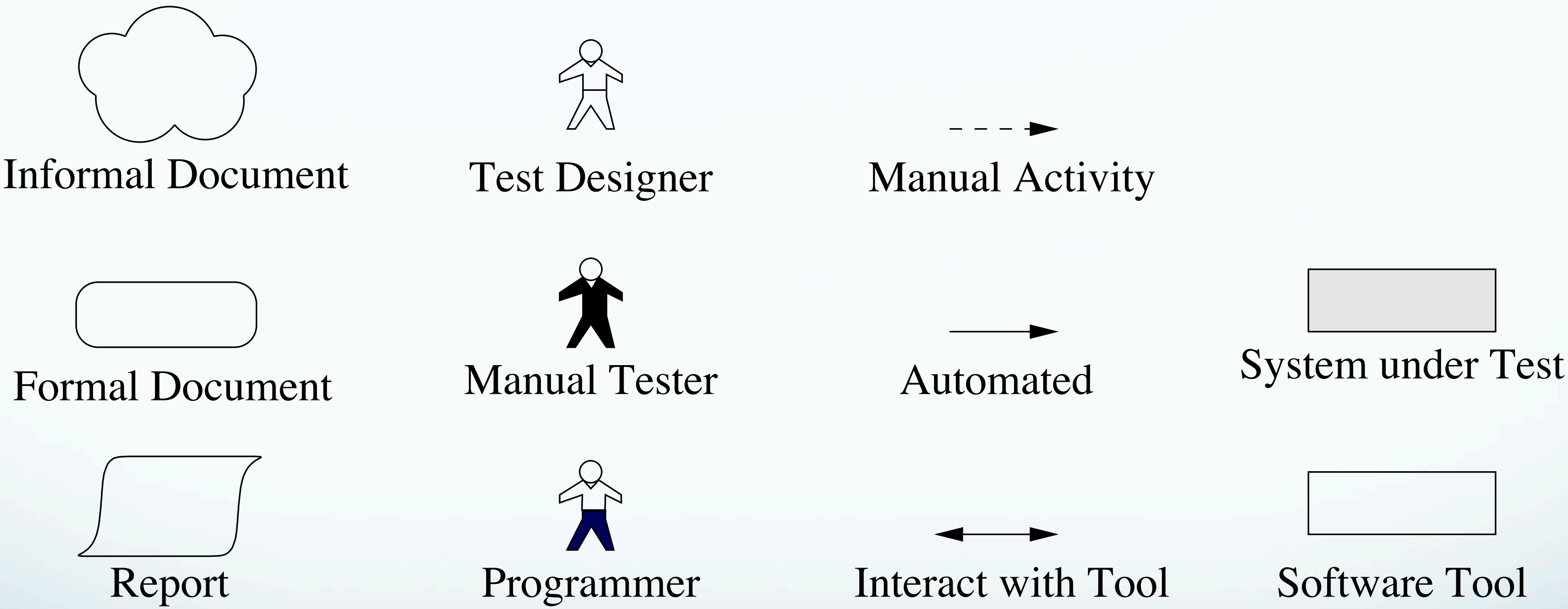
When designing **functional testing**, 3 key issues:

1. Designing the test case
2. Executing the tests and analyzing the result
3. Verifying how the tests cover the requirements

Different testing processes

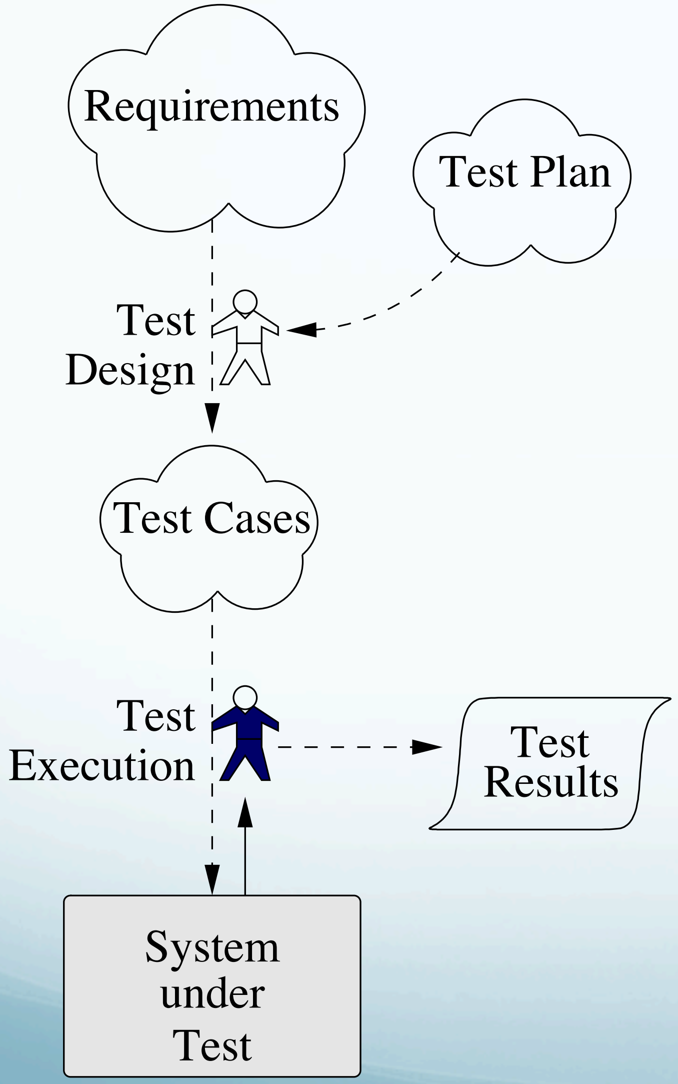
1. Manual testing process
2. A capture/reply testing process
3. A script-based testing process
4. A keyword-driven automated testing process
5. The MBT process

Preliminaries: notation…



Source: M. Utting and B. Legeard, *Practical Model-Based Testing*

1. Manual Testing



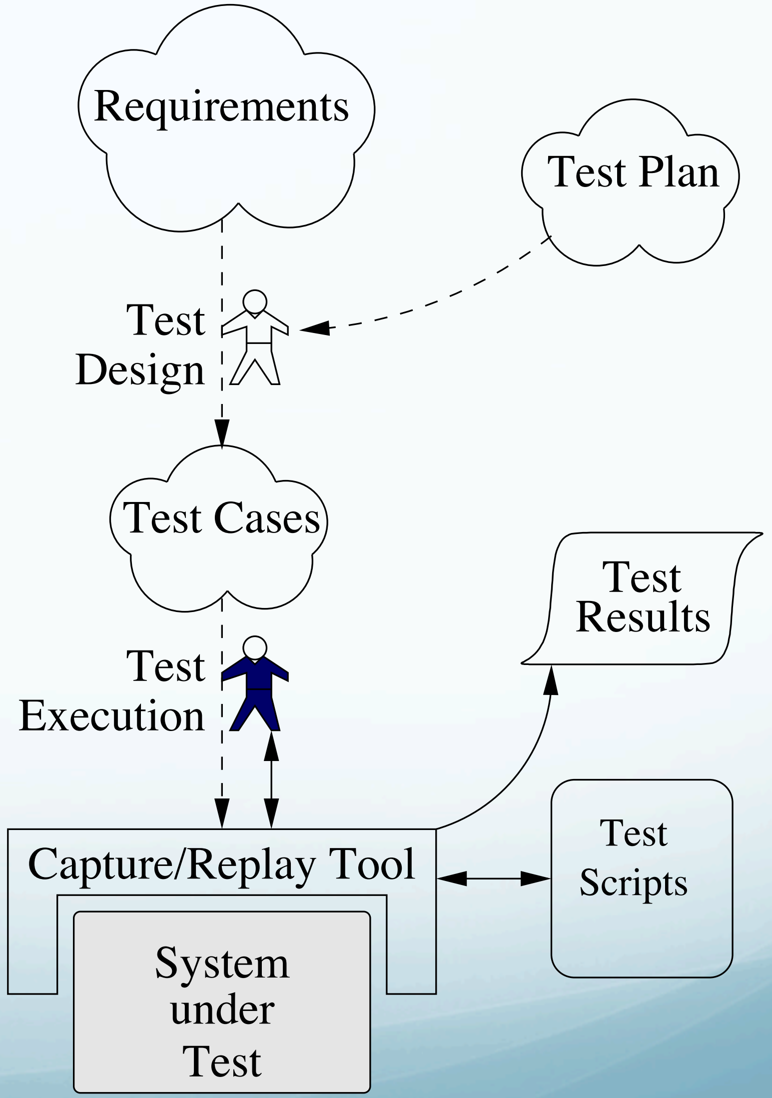
+ easy & cheap to start
+ flexible testing
- expensive every execution
- no auto regression testing
- ad-hoc coverage
- no coverage measurement

Source: M. Utting and B. Legeard, *Practical Model-Based Testing*

2. Capture-Replay Testing

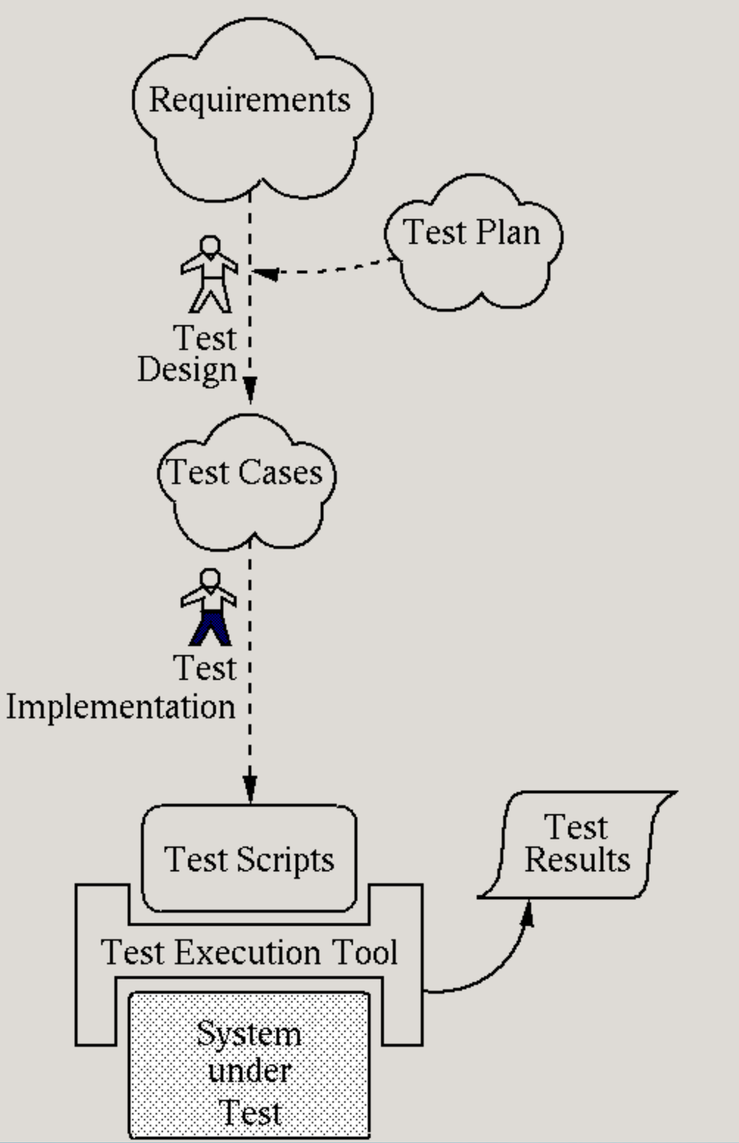
+ flexible testing
- expensive first execution
+ auto regression testing
- fragile tests break easily
- ad-hoc coverage
- no coverage measurement
- low-level recorded tests

NOTE: Mostly used to automate testing of graphical user interface (GUI)



Source: M. Utting and B. Legeard, *Practical Model-Based Testing*

3. Script-Based Testing



+/- test impl. = programming

+ automatic execution

+ auto regression testing

- fragile tests break easily?

  (depends on abstraction)

- ad-hoc coverage

- no coverage measurement

Source: M. Utting and B. Legeard, *Practical Model-Based Testing*

4. Keyword-Driven Testing

+ abstract tests

+ automatic execution

+ auto regression testing

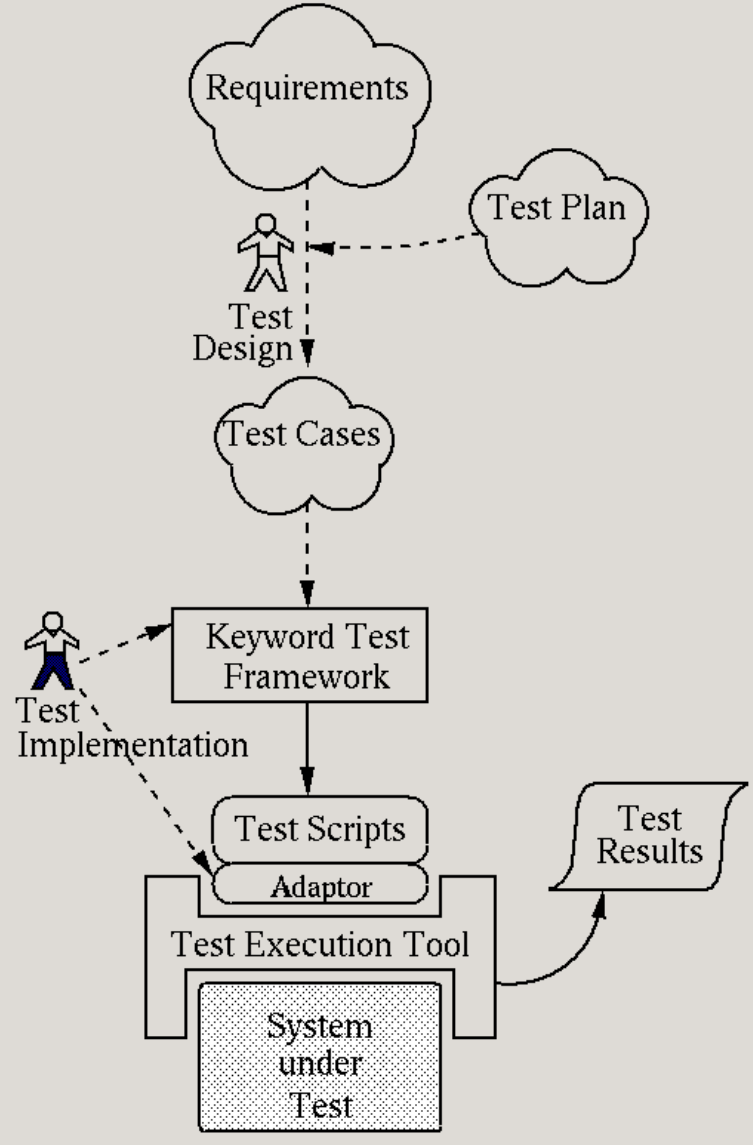
- robust tests

- ad-hoc coverage

- no coverage measurement

- manual design of test data and oracle

Note: action keywords (the "adaptor") allowing translate sequence of keywords and data into executable tests

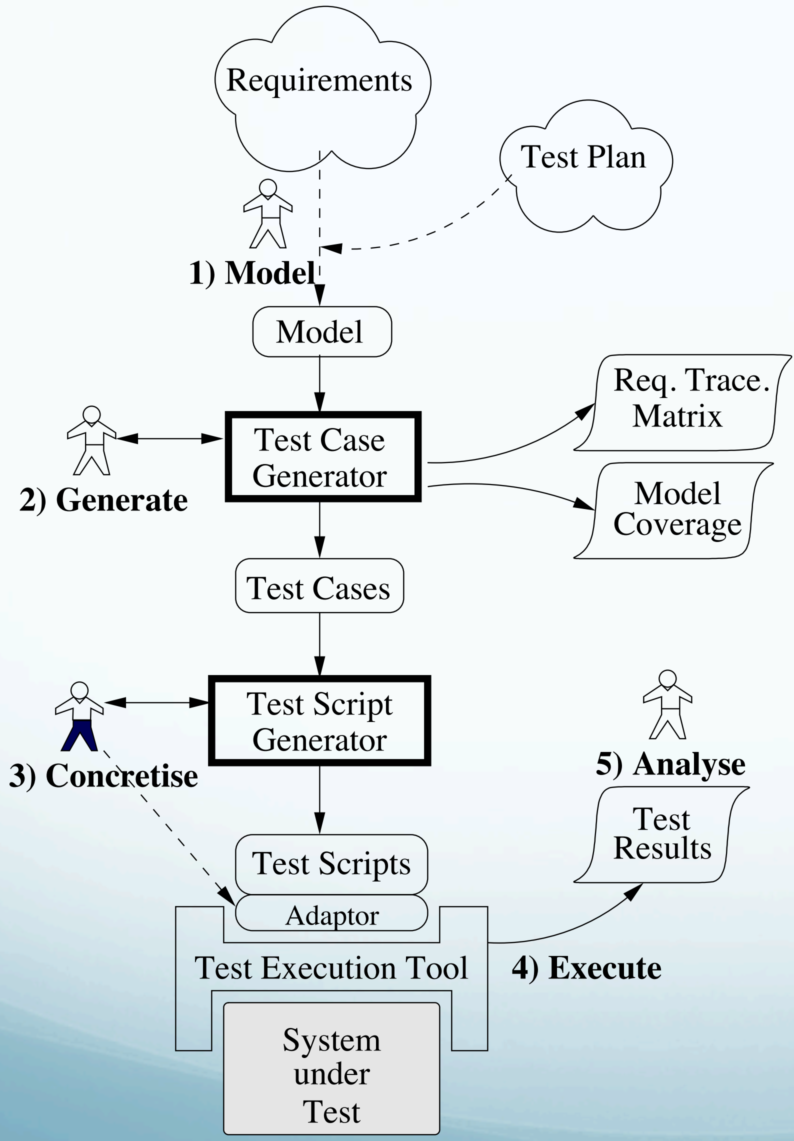


Source: M. Utting and B. Legeard, *Practical Model-Based Testing*

5. Model-Based Testing

1. **Model** the SUT and/or its environment
   - Write some abstract model / annotate with relationship between tests and requirements
2. **Generate** abstract tests from the model
   - Chose some test selection criteria to generate tests from the model. Coverage and results refer to the model!
3. **Concretize** the abstract tests to make them executable
   - Use a transformation tool to get concrete tests (on the SUT) from the abstract tests from the model
4. **Execute** the tests on the SUT and assign verdicts
5. **Analyze** the test results (and take corrective action)
   - A fault in the test case might be due to a fault in the adaptor code or in the model

5. Model-Based Testing



- \+ abstract tests
- \+ automatic execution
- \+ auto regression testing
- \+ auto design of tests
- \+ systematic coverage
- \+ measure coverage of model and requirements
- \- modeling overhead

Important: usually first **abstract tests** -> needs to get **concrete** tests: **adaptor**!

Source: M. Utting and B. Legeard, *Practical Model-Based Testing*

Building Models…

Reusing or building from scratch?

Reusing existing development model

- 100% reuse; not always possible:
  1. Develop. models usually contains too much detail
  2. Usually don't describe the SUT dynamic behavior
- Not abstract enough yet precise enough for test generation

Reuse something

- Some x% of reuse (0<x<100)
- Eg. reuse high-level class diagram and some use cases; add behavioral details

Developing model from scratch

- 0% reuse
- Maximize independence
- A lot of effort

Whatever approach: relate your model to the informal requirements as close as possible!

Benefits of MBT

1. SUT Fault detection
   - Increase the possibility of finding errors
2. Reduced testing cost and time
   - Less time and effort spent on writing tests and analyzing results
   - Could generate shortest test sequences
3. Improved test quality
   - Possible to measure the "quality" by considering coverage (of model)
4. Requirements defect detection
   - Modeling phase exposes requirements issues
5. Traceability
   - Between requirements and the model
   - Between informal requirements and generated test cases
6. Requirements evolution
   - Update test suite to reflect new requirements: update model and do it automatically

Limitations of MBT

1. Cannot guarantee to find all differences between the model and the implementation
2. Need of skilled model designers: abstract and design models
3. Mostly (only) for functional testing
4. Some tests not easily automated: eg. installation process

After you adopt MBT:

1. Outdated requirements
   - Might build the wrong model
2. Inappropriate use of MBT
   - Parts difficult to model; may get the wrong model
3. Time to analyze failed tests
   - It may give complex test sequences
4. Useless metrics
   - Number-of-tests metrics not useful (huge number!) – other metrics needed

How to model your system?

1. Decide on a good level of abstraction
   - What to include and what not to
2. Think about the data it manages, operations it performs, subsystems, communication…
   - Maybe start from a UML class diagram?
   - Be sure you simplify your class diagram! (simpler for testing than for design!)
3. Decide notation
4. Write the model
5. Ensure your model is accurate
   - Validate the model (it specifies the behavior you want)
   - Verify it (correctly typed and consistent)
6. Use your model to generate your tests

Notations for modeling

Seven possible "paradigms"

1. Pre/post (state-based)

Snapshot of internal state of the system + operations

- B, Z, UML OCL,m VDM,

2. Transition-based

- FSMs, statecharts, LTS, I/O automata

3. History-based

Allowable traces if behavior over time

- MSC, sequence diagrams, …

4. Functional

Collection of mathematical functions

- FOL, HOL

5. Operational

Collection of executable parallel processes

- CSP, CCS, Petri nets, PI-calculus

6. Statistical

Probabilistic model of the event and input values

- Markov chains

7. Data-flow

- Lustre, Block diagrams in Simulink

Choosing a notation

For **MBT**, transition-based and pre/post notations are the most used

- Guidelines: Is the system data-oriented or control-oriented?

Data-oriented systems have state variables, rich types (sets, relations, sequences,…).

Operations to access and manipulate data

Data-oriented systems are most easily specified using pre/post notations

- Eg. **B**, having powerful libaries of data structures

In control-oriented systems the set of available operations depends on the state

Control-oriented systems are most easily specified using transition-based notations

- Eg. **FSMs**

- **Note 1**: Possible to use transition-based notations for data-oriented systems: handle data structures too (eg. **EFSMs**)

Note 2: In MBT the model should be **formal**!

Our focus in this course: transition-based notations!

Drinking Vending Machine (DVM)
Case Study

Utting & Legeard book: sec 3.2, pp.66!

Requirements:

Source: M. Utting and B. Legeard, *Practical Model-Based Testing*

DVM case study

Use case

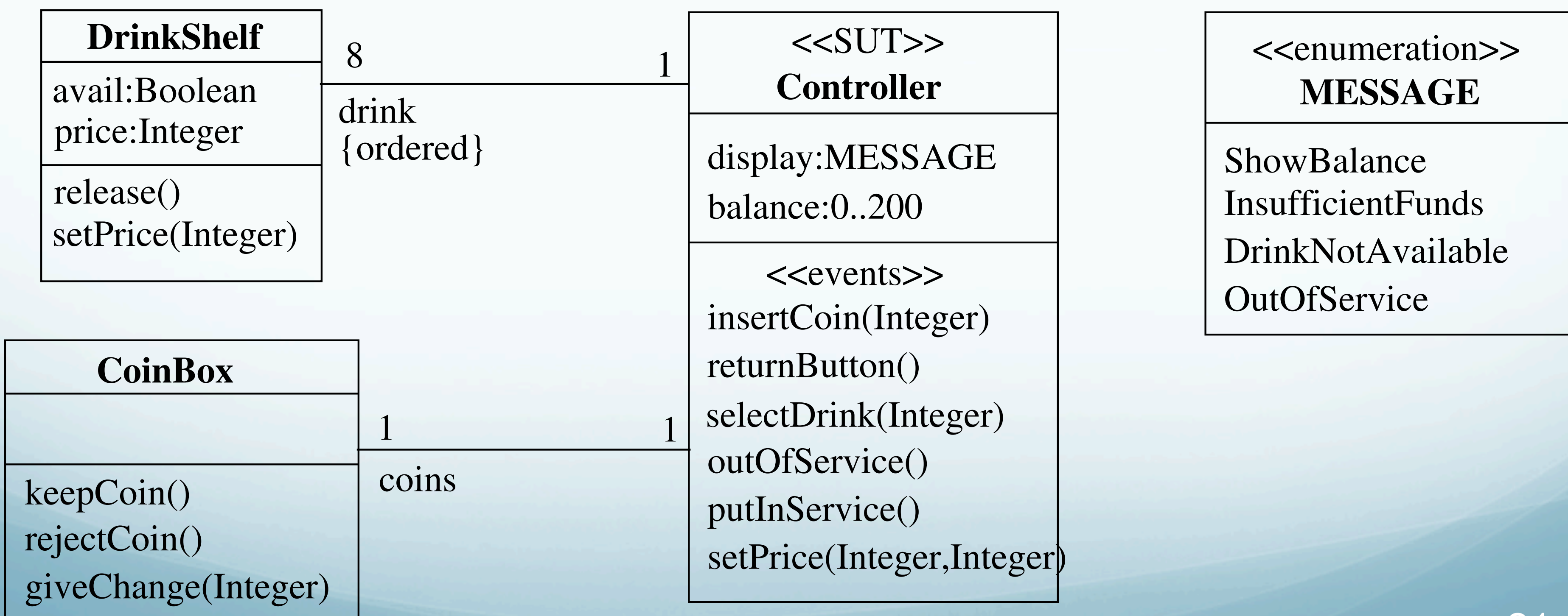
Utting & Legeard book:
Use Case 3.1, pp.67!

Source: M. Utting and B. Legeard, *Practical Model-Based Testing*

DVM case study
High-level design

We need a high-level architecture of the DVM: how the controller interacts with other components

UML class diagram:



Source: M. Utting and B. Legeard, *Practical Model-Based Testing*

DVM case study
What's next?

- Informal description, use cases, high-level design, etc. give us an idea of what a DVM controller does
- But… do not specify all the input conditions, alternatives, exception cases, we want to **test**
- **Not precise enough for test generation**

We need to write a model "for testing"!

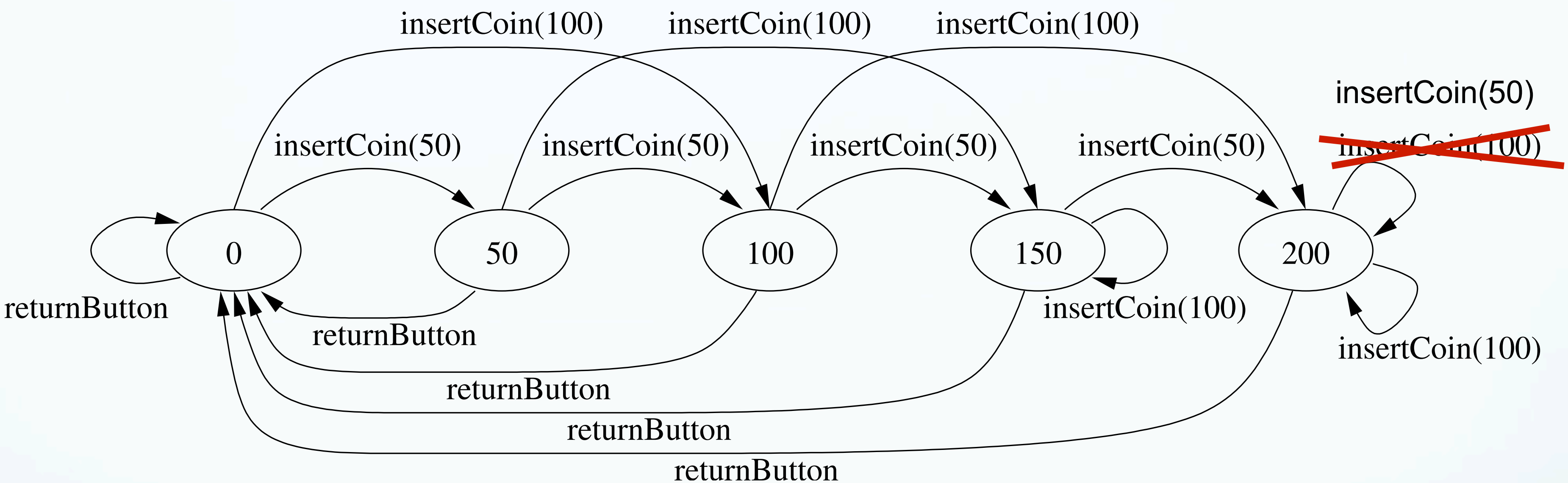
DVM – Transition-based model
Group exercise

- Come up with a **finite state machine (FSM)** that models the Controller component of the DVM
  - Start with a machine for the money operation *insertCoin* and *returnButton*

Groups 2-5 persons: 15 min

DVM - FSM model

Partial solution to FSM for the DVM money operation (*insertCoin, returnButton*)



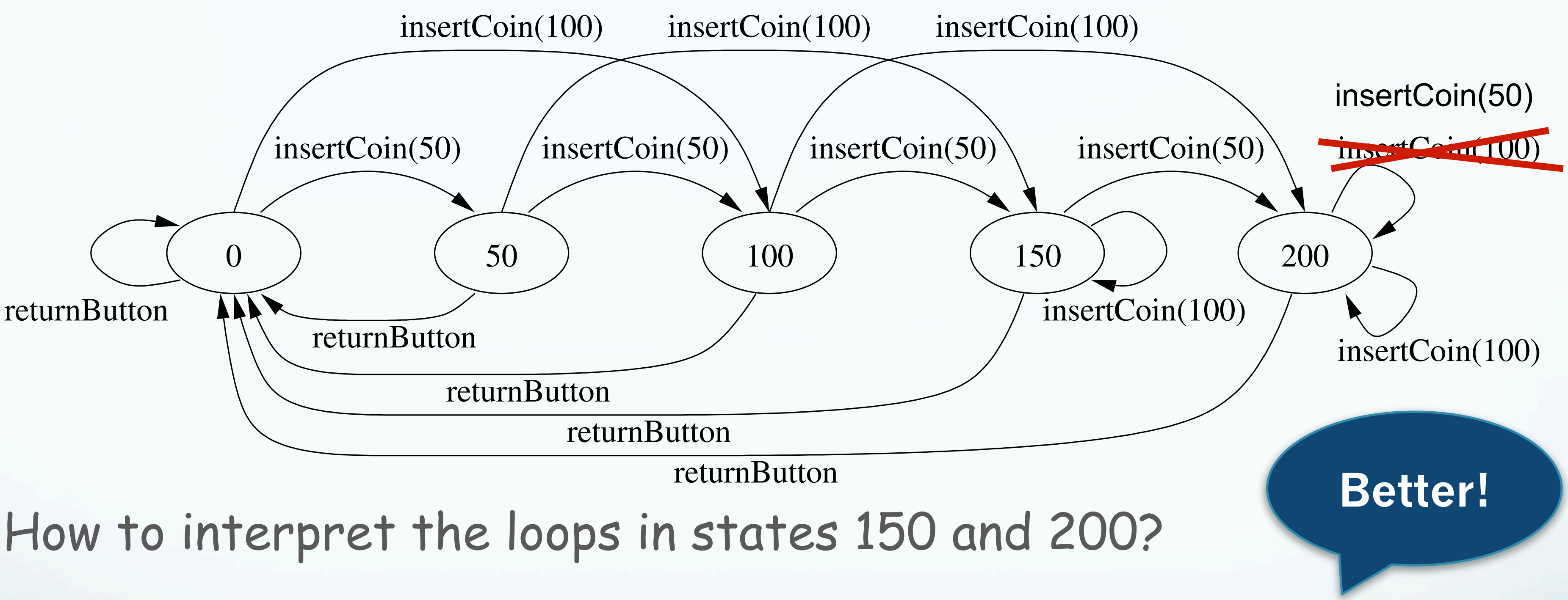
- You will need to come with **more complex transition-based notations** (UML state machine diagrams, EFSMs, etc.) for a full solution useful for **test generation**

Btw, anything wrong with the proposed solution?

- 2 transitions insertCoin(100) from state "200"
  - Correction: insertCoin(100) + insertCoin(50)

Source: M. Utting and B. Legeard, *Practical Model-Based Testing*

DVM - FSM model
Some comments…



How to interpret the loops in states 150 and 200?

1. Nothing happens -> the content of the cash box doesn't change
2. Wrong in state 150 -> add a transition with insertCoin(100) from 150 to 200 and interpret state 200 as "containing at least 200"

- In both cases: **Underspecified what happens with the coins (change needs to be given) -> fix when full model**

Source: M. Utting and B. Legeard, *Practical Model-Based Testing*

Pre/Post models in B… in 1 slide

- The B abstract machine notation: formal modeling notation for specifying software
  - High-level libraries of data structures
  - Code-like notation for post-conditions
- Development starts from an abstract model
  - High-level function view
- Write a series of increasingly detailed designs: refinement
- B supports tools for **automatic** generation of proof obligations to prove correct refinement
- **MBT using B**: checks the model against the implementation, but via testing (does not guarantee to find all errors)!

DVM – B model

Utting & Legeard book: listing 3.1, pp.80!

Partial: models money only

Invariant: doesn't change in the program

||: Multiple assignments

reject: output var
insertCoin: name operation
coin: input var

What follows only holds provided the **precondition** holds

Source: M. Utting and B. Legeard, *Practical Model-Based Testing*

MBT – How to do in practice?

- Next lecture on how to select your tests
  - More on coverage…
- In practice: future lectures
  - Testing from (E)FSM
  - ModelJUnit

MBT – Summary

- MBT is the automation of black-box test design
  - Test cases can be automatically generated from the model using MBT tools
- The model must be precise and concise
- Tests extracted are abstract; they must be transformed into executable tests
- Not practically to (completely) reuse a development model for MBT
- Transition-based notations: better for control-oriented systems
- Pre/post notations: preferable for data-oriented systems
- Possible to write partial models and refine
  - A very abstract model: few high-level tests covering few aspects of the system
  - A more detail model: tests covering more

The quality and number of tests that you get from MBT depend on the quality and precision of your model

References

- M. Utting and B. Legeard, *Practical Model-Based Testing*. Elsevier - Morgan Kaufmann Publishers, 2007
  - Chapters 1-3