Technische Universität
Kaiserslautern

o1o1seda o1o1oo

software engineering dependability

Software Quality Assurance
Test of Object-Oriented Software

Contents

- Object-Orientation and Quality Assurance
- Object-Oriented Programming and Quality Assurance
- Rules for development
- Properties of object-oriented systems
- Object-oriented module test: class test
- Object-oriented integration test
- Object-oriented system test

Object-Orientation and Quality Assurance

+ Starting point: conceptual faults
+ Use of commercial libraries (tested)
+ Re Use
+ Consistent concept for analysis, design, implementation
+ Clear concept (rules of the model)
+ Paradigm for problem analysis
- Single paradigm for problem analysis (paradigm-blindness)

Object-Oriented Programming and Quality Assurance

+ Consistent concept for analysis, design, implementation
+ High productivity (libraries)
+ Data abstraction
- Enormous analysis problems during the test
- Dynamic binding causes problems with real-time
- Polymorphism
- Complexities by inheritance

Rules for Development Analysis and Design

- Modularization is one of the major positive influencing factors wrt. testing
  - Modules are clearly identified (classes)
  - Independent testability of the classes due to encapsulation
- Consequence: Don't destroy the encapsulation concept (has to be considered during design at the latest)
- Keep up consistently the modularization concept according to object-orientation (e.g. generation of data abstractions by the combination of data and operations, belonging together logically, in classes)
  - No breaking of the encapsulation concept (e.g. no friend-classes in C++)
  - No public attributes
  - Try to produce consistency of the program architecture and the specification structure

Rules for Development Analysis and Design

- Inheritance in combination with polymorphism is a critical property of the object-orientation w.r.t. testing (understandability of the structure is reduced)
  - Careful use of inheritance
  - No too deep inheritance hierarchies
  - Use of multiple inheritance not too often
  - Consistent preservation of a particular inheritance hierarchy (typically from the general to the specific)

Rules for Development Implementation

- Observance of the rules mentioned above also for the implementation
- Enhancement of the observability of the class internals (e.g., values of the class attributes) by the use of assertions
- In complex or critical parts of the software preference of understandable, simple programming instead of *elegant* solutions
- No use of dynamic binding in time-critical software components

Properties of Object-Oriented Systems

- Objects and classes are more complicated than functions
- Minimum level of the complexity corresponds to that of data abstractions respectively abstract data types in classic software developments
- Objects respectively classes have
  - Relations
  - Properties
  - Parts
  - A state
- They are able to
  - Send messages
  - Execute operations

Properties of Object-Oriented Systems

- They have
  - Preconditions
  - Postconditions
  - Invariants
  - Exception handling
- The interactions between parts of objects are complicated ➔ in object-oriented software systems some parts of the component test are identical with integration test steps for classic software systems

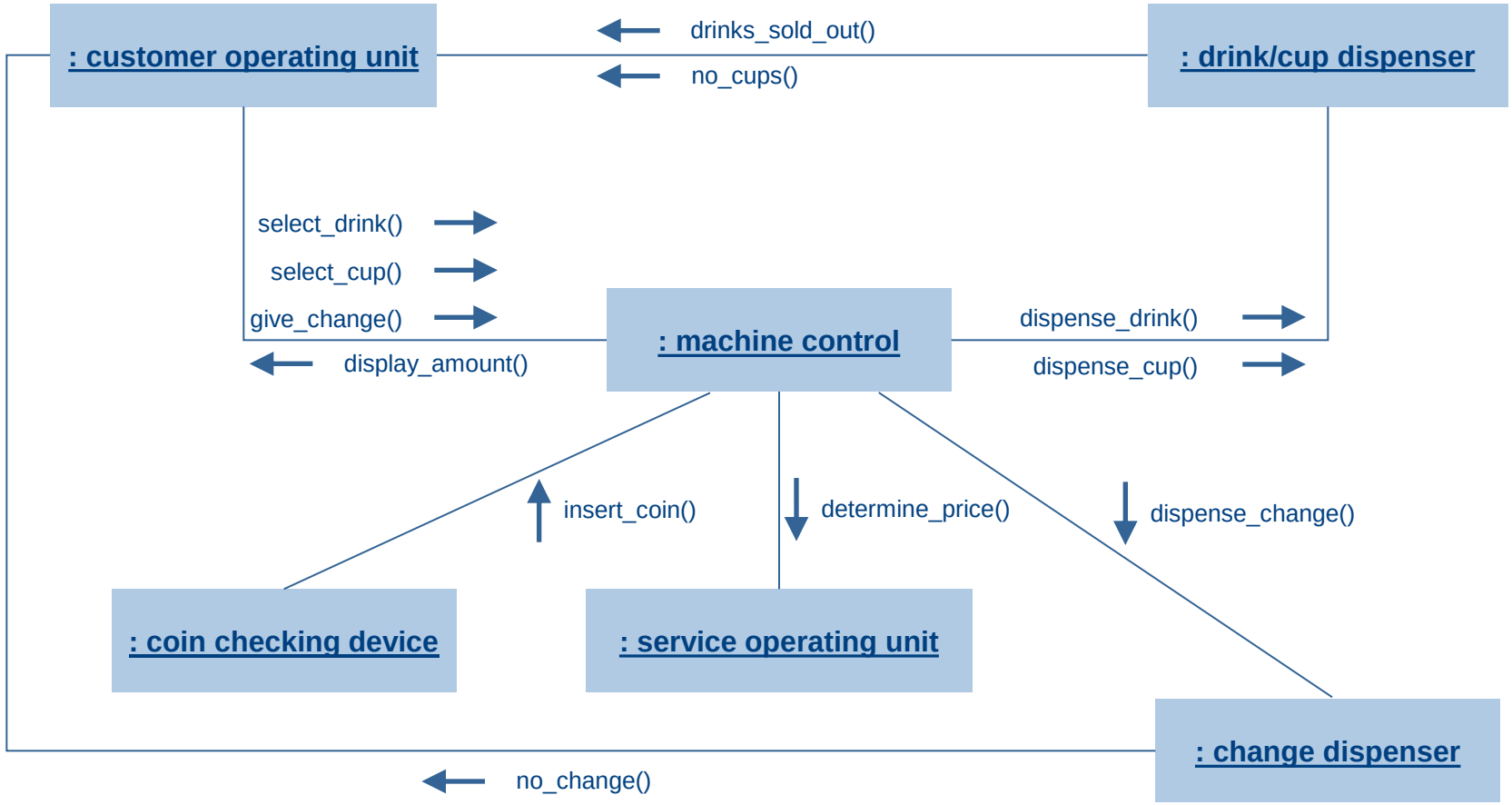
Properties of Object-Oriented Systems
Object Types

- Essential objects
  - Model parts of the application, are identified as part of the system requirements
- Non-essential objects are generated during later phases of the software development (design, implementation)
  - Are elements of the technical realization of a software system
  - Are often standard components

Object-Oriented Example

- In the following as an example the control of an object-oriented refreshments vending machine is used
- The machine contains
  - The machine control
  - The customer operating unit
  - The service operating unit
  - The coin checking device
  - The drinks/cup dispenser
  - The change dispenser

Test of Object-Oriented Software
Object-Oriented Example



Object-Oriented Unit Test: Class Test
Test of Individual Operations

- The test of operations is comparable to classic testing of functions and procedures
- But operations
  - Often have a very simple control structure
  - Are highly dependent on the attributes of the object
  - Have interdependencies
- Usually, operations may not be tested separately

Object-Oriented Unit Test: Class Test
Test of Individual Operations

- Only in exceptional cases operations can be tested alone; here specification of the operation „dispense_change"
- *The class "change dispenser" contains the information about the coins available for paying change according to sort and number. Altogether max. possible are 50 coins at 2 Euros, 100 coins at 1 Euro, 100 coins at 50 Cent, 100 coins at 20 Cent and 200 coins at 10 Cent. Change is paid according to the following rules*
  - *Payment is done with the lowest number of coins, i.e. the change is paid at first if necessary with 2 Euro coins, then with 1 Euro coins, following 50 Cent coins, 20 Cent coins and 10 Cent coins. If a required sort of coins is not available anymore, the payment is done with the following smaller sort*
  - *If less than twenty 10 Cent coins are available, the message no_change (yes) is sent to activate the No-Change-display. This also happens if a total sum of the change except for the 2 Euro coins falls below 5 Euros. Otherwise the message no_change (no) is sent*

Object-Oriented Unit Test: Class Test
Test of Individual Operations

- Equivalence Classes

Condition	valid		invalid	
change	change ≥ 2 €	2 € > change ≥ 1 €	< 0	
	1 € > change ≥ 0,50 €	0,50 € > change ≥ 0,20 €		
	0,20 € > change ≥ 0,10 €	0,10 € > change ≥ 0 €		
coins				
2 €	50 ≥ coins > 0	coins =0	< 0	> 50
1 €	100 ≥ coins > 0	coins =0	< 0	> 100
0,50 €	100 ≥ coins > 0	coins =0	< 0	> 100
0,20 €	100 ≥ coins > 0	coins =0	< 0	> 100
0,10 €	200 ≥ coins ≥ 20	20 > coins ≥ 0	< 0	> 200
no_change	total sum of change (ex 2€) < 5 €	total sum of change (ex 2€) ≥ 5 € AND #10 Cent coins ≥ 20		
	#10 Cent coins < 20			

Object-Oriented Unit Test: Class Test
Test of Individual Operations

- Test Cases for valid equivalence classes

Test case	1	2	3	4	5	6
change	2 €	1 €	0,50 €	0,20 €	0,10 €	0,00 €
coins						
2 €	50	1	0	10	0	10
1 €	100	1	0	10	0	10
0,50 €	100	1	0	10	0	10
0,20 €	100	1	10	0	9	10
0,10 €	200	20	1	19	4	40
no_change	change ≥ 5 € #10 Cent ≥ 20	change < 5 € #10 Cent ≥ 20	change < 5 € #10 Cent < 20	change ≥ 5 € #10 Cent < 20	change < 5 € #10 Cent < 20	change ≥ 5 € #10 Cent ≥ 20
Result	1*2 € don‘t activate no_change	1*1 € activate no_change	2*0,20 € 1*0,10 € activate no_change	2*0,10 € activate no_change	1 * 0,10 € activate no_change	don‘t activate no_change

- Test cases for invalid equivalence classes

Test case	7	8	9	10	11	12	13	14	15	16	17
change	1 €	1 €	1 €	1 €	1 €	1 €	1 €	1 €	1 €	1 €	**-0,10 €**
coins											
2 €	**51**	10	20	10	10	**-1**	20	30	20	14	10
1 €	10	**101**	20	10	10	10	**-1**	10	10	10	10
0,50 €	10	1	**101**	10	10	10	30	**-1**	22	24	10
0,20 €	10	1	10	**101**	9	10	40	2	**-1**	8	10
0,10 €	30	42	30	40	**201**	10	50	49	30	**-1**	50

Object-Oriented Unit Test: Class Test
Test of Operations in their Class Context

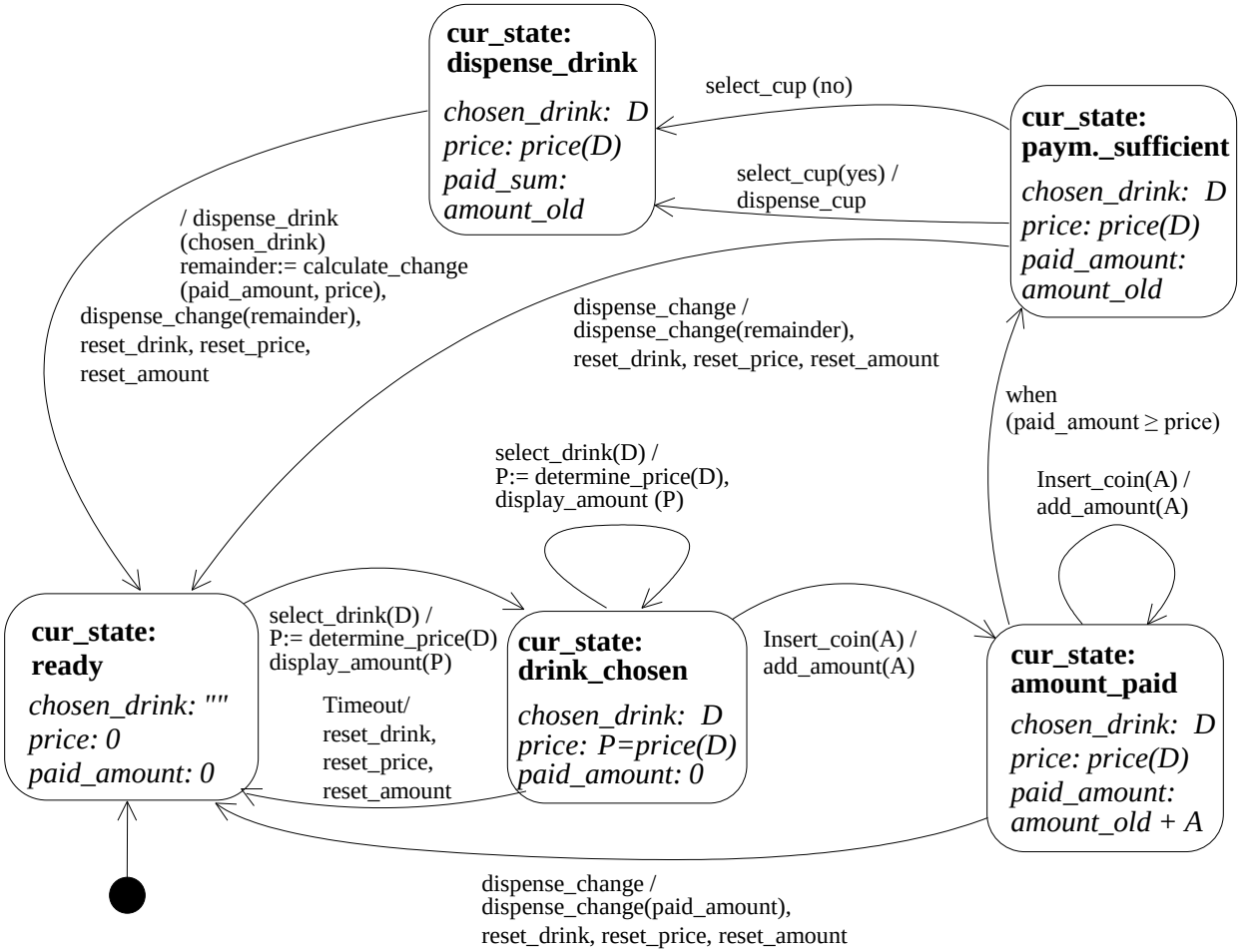
- Usually, operations have to be tested in the context of their class
- The operations of an object of a class interact via shared attributes
- The values of the attributes define the present state of the object

⇒state machines are appropriate means for specification

⇒state machines can serve also as the basis for testing

- Field of application: Testing of operation sequences

Object-Oriented Unit Test: Class Test
Test of Operations in their Class Context



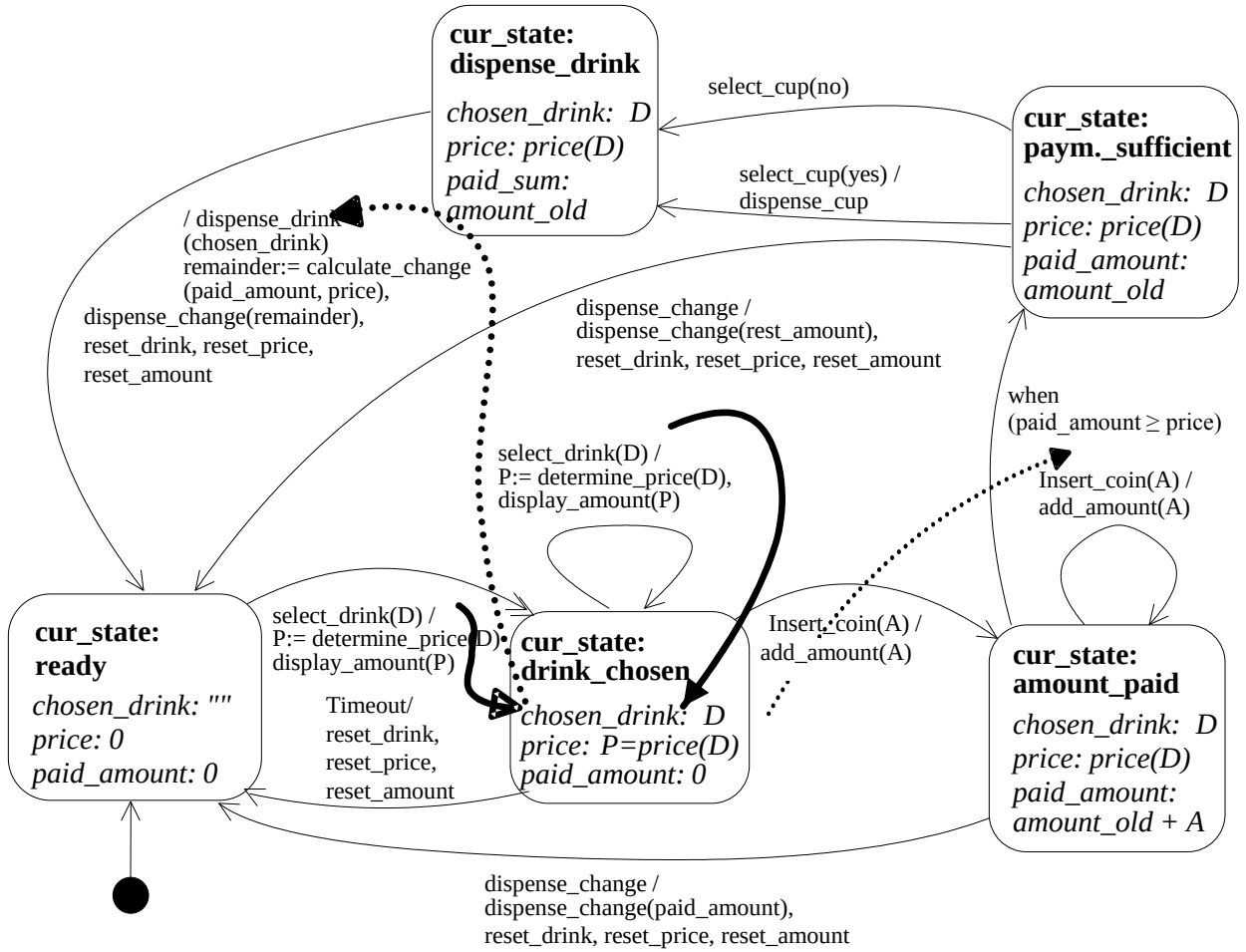
Object-Oriented Unit Test: Class Test
Test of Operations in their Class Context

- Hierarchy of completeness criteria
  - coverage of all states at least once
  - coverage of all transitions at least once
  - coverage of all events at all transitions at least once
- Hierarchy
  - all events $\supseteq$ all transitions $\supseteq$ all states
- Important: Do not forget to test the error handling

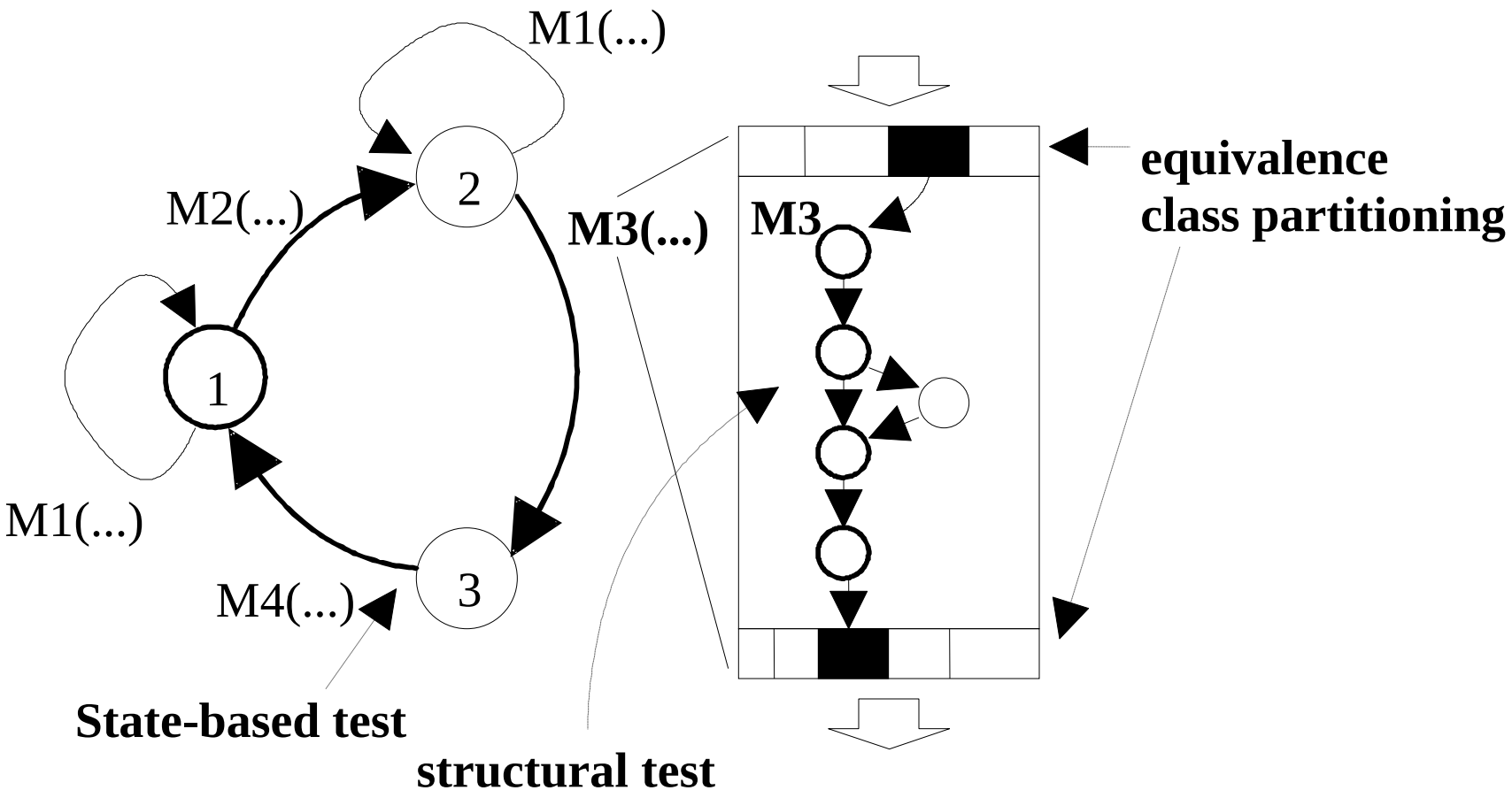
Object-Oriented Unit Test: Class Test Structural Test

- Control flow test techniques (e.g. branch coverage test) are relatively inappropriate, because they disregard the interactions between operations through shared attributes
- Data flow test techniques are more appropriate
- The attributes are written (defined) and read (used) by operations. A data flow test based on the attributes demands to test interactions concerning the shared data

Object-Oriented Unit Test: Class Test
Test of Operations in their Class Context



Object-Oriented Unit Test: Class Test
Test of Operations in their Class Context



Object-Oriented Unit Test: Class Test
Test of Operations in their Class Context

- Problem
  - abstract and parameterized classes don‘t permit direct instantiation of objects
  - diversity of the producible objects increases the complexity
  - only concrete objects can be tested. Question: Which?
  - abstract and parameterized classes are to concrete classes as normal classes are to their objects
- Abstract classes
  - Define the syntactic and semantic interface for operations, without offering an implementation
  - The implementation for abstract methods is given in a subclass of the abstract class
- Parameterized classes
  - contain formal class parameters which have to be replaced with actual values

Object-Oriented Unit Test: Class Test
Testing of Abstract and Parameterized Classes

- Instantiation of a concrete class
- Testing of this class as a normal class
- Questions
  - Which instantiation has to be chosen?
  - How should it be tested?
- Rule: Generation of a concrete class as simple as possible, i.e.
  - abstract classes
    - realization of implementations for abstract methods
      - if possible, empty

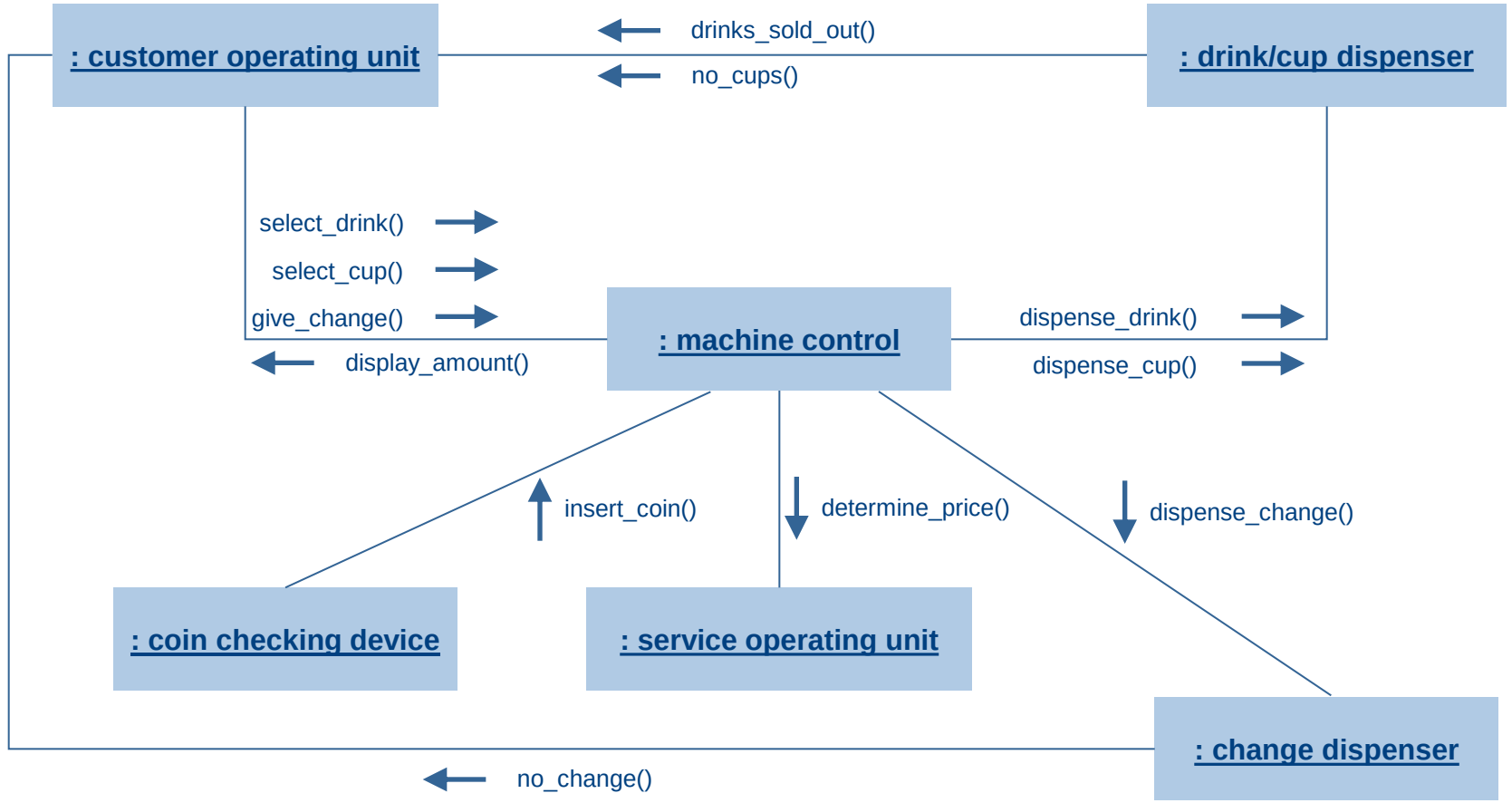
      Otherwise
      - As simple as possible, but the specification must be fulfilled; only as complicated as necessary
  - parameterized classes
    - Selection of parameters which make the test as simple as possible (e.g. “stack for integer"

Object-Oriented Integration Test
Base Classes

- Questions
  - Does the interface between two objects work in both directions (passing of parameters and results)?
- Idea
  - Coverage of the specification in both directions
- Generation of test cases
  - which cover the different parameters, that might be used by the calling object
  - which cover the different return values generated by the service provider

➔ Equivalence class partitioning of the interface between service user and service provider

- Example: Integration test of the objects „coin checking device" and „machine control"



Object-Oriented Integration Test
Integration Test of Base Classes

- Test of the interaction of the classes *coin checking device* and *machine control* via the message *insert_coin()*
  - Interface specification of the operation *insert_coin()*
    - The operation *insert_coin()* expects a non-negative value (maximum value = 1000). It specifies the value of the coin in cents
    - The operation has no return value
  - Interface parameters of the calling routine at the *coin checking device*
    - The following values may be used for the interface of the message *insert_coin()*: 10, 20, 50, 100, 200

Object-Oriented Integration Test
Integration Test of Base Classes

- Consequences for the integration test
  - It has to be ensured that the value of the interface parameter fulfills the following condition (so-called assertion)
    - *(value >= 0) AND (value <= 1000)*
  - Equivalence classes and test cases
    - value = 10
    - value = 20
    - value = 50
    - value = 100
    - value = 200
  - Testing of the return value is not possible as no return values exist

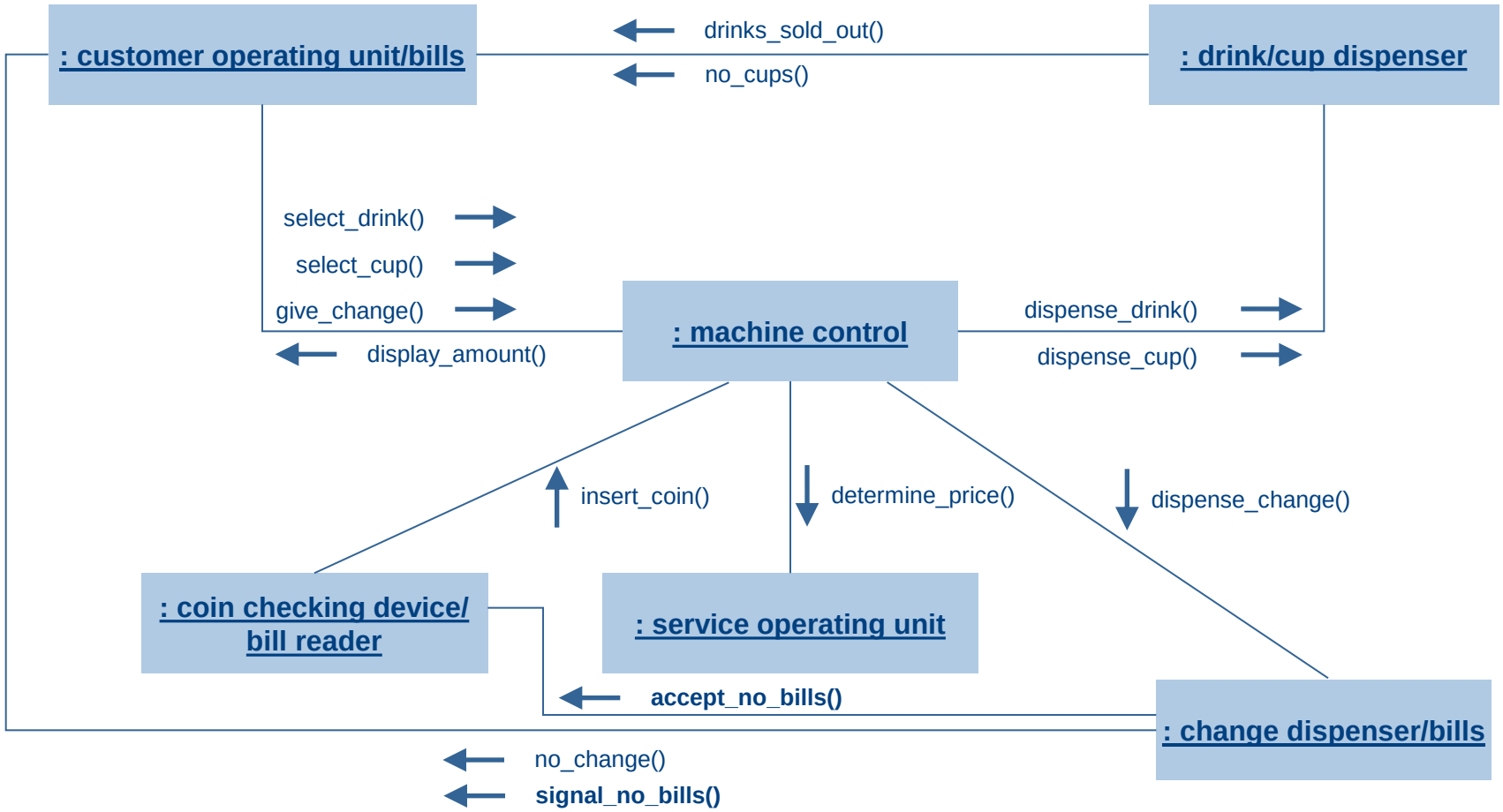
Object-Oriented Integration Test
Integration Test and Inheritance

- Situations
  - Inheritance at the service provider
  - Inheritance at the service user
  - Inheritance at the service provider and at the service user

Object-Oriented Integration Test
Integration Test and Inheritance - Example

- In the new version of the refreshments vending machine it is possible to pay with money bills (5 Euros and 10 Euros). The following changes are made
  - A class *coin checking device/bill reader* is implemented. The operation *checking()* is augmented w.r.t. checking bills. This operation overwrites the original operation. A new operation *accept_no_bills()* is added, that disables or enables the acceptance of bills
  - The class *change dispenser* gets a subclass, with a new operation *dispense_change(),* that overwrites the old operation. The new operation signals whether it is possible to pay with bills. The payment with banknotes is disabled if the money stock in coins falls below 15 Euros. Bills are not returned as change
  - The class *customer operating unit* gets a subclass which contains a new operation *signal_no_bills().* This operation signals whether it is possible to pay with bills

Object-Oriented Integration Test
Integration Test and Inheritance



Object-Oriented Integration Test
Inheritance at the Service Provider

- Situation
  - Integration test of the service user and the superclass of the service provider is executed according to the procedure for the integration test of base classes
- Problem
  - Operations of the service provider can be inherited from the superclass (the old service provider), but not necessarily. Methods of the new service provider as well as methods of the old service provider (the super class) can be executed

Object-Oriented Integration Test
Inheritance at the Service Provider

- Procedure
  - No additional test cases for inherited operations, as this case is covered already by the integration test of the base classes ➔ repeat test cases
  - No additional test cases concerning overwritten operations for which only the implementation has changed, as the interface specification remained identical and this case is also covered yet ➔ repeat test case

- If the interface specification of the overwriting method has changed, the following cases are to be distinguished
  - The interface of the overwriting method is more specific (i.e. accepts less data) than the interface of the overwritten method
    - Definition of a new assertion
    - Repetition of all test cases from the integration test of the base classes
  - If the interface becomes more general by the overwriting of the method no additional test cases are required, as this case is covered already by the test of the base classes ➔ repeat test cases

Inheritance at the Service Provider - Output Interface (parameters), Example

- If necessary additional test cases for the coverage of a wider interface which was not covered sufficiently during the test of the base classes
- Example: Enhanced version of the refreshments vending machine
  - The new change dispenser is a service provider (*dispense_change()*) concerning the *machine control*
  - The overwriting operation *dispense_change()* has only a modified implementation (transmission of additional messages). The interface specification is unchanged
  - It is sufficient to repeat the old test cases. The assertion is unchanged

Inheritance at the Service User - Output Interface (parameters)

- No additional test cases for inherited operations ➔ repeat test cases
- No additional test cases if the interface of the overwriting operation in call direction is more specific than the interface of the overwritten operation (i.e. calls which were possible before are not possible anymore) ➔ repeat test cases
- If the interface becomes wider (i.e. calls which were not possible before are possible now) the old test cases are to be completed accordingly ➔ repeat old test cases and execute new test cases additionally
- Comment: the assertion is unchanged

- Repeat test cases. If a failure occurs due to a more specific interface an appropriate correction is required

- Procedure
  - Apply technique to deal with inheritance at the service provider
  - Apply technique to deal with inheritance at the service user
  - Add test cases for the new interactions between service provider and service user

- Between the derived classes *change dispenser/bills* and *coin checking device/bill reader* there is a service provider-service user-relation. Additionally to the described tests the interaction by the message *accept_no_bills()* has to be tested
- Test cases
  - *accept_no_bills(yes)*
  - *accept_no_bills(no)*

Object-Oriented Integration Test
Inheritance and Integration Test: Summary

- Table 0 (start table) for handling inheritance

service user	service provider	action
unmodified	unmodified	repeat test cases
unmodified	generated by inheritance	evaluate tables 1.1 and 1.2
generated by inheritance	unmodified	evaluate table 2
generated by inheritance	generated by inheritance	evaluate tables 1.1, 1.2 and 2; add test cases for interaction of the subclasses

Object-Oriented Integration Test
Inheritance and Integration Test: Summary

- Table 1.1: Testing the call interface

service providing operation	call interface of the service providing operation	action
inherited	-	repeat test cases
overwriting	identical	repeat test cases
overwriting	more specific	new assertion; repeat test cases
overwriting	more general	repeat test cases

Object-Oriented Integration Test
Inheritance and Integration Test: Summary

- Table 1.2: Testing the return interface

service providing operation	return interface of the service providing operation	action
inherited	-	repeat test cases
overwriting	identical	repeat test cases
overwriting	more specific	repeat test cases
overwriting	more general	repeat test cases; generate additional test cases

Object-Oriented Integration Test
Inheritance and Integration Test: Summary

- Table 2: test of the call and the return interface

service providing operation	call interface of the service providing operation	action
inherited	-	repeat test cases
overwriting	identical	repeat test cases
overwriting	more specific	repeat test cases
overwriting	more general	repeat test cases; generate additional test cases

Object-Oriented System Test

- Except for the specification-based test there are no differences compared to the test of classic software
  - The system is a black box ➔ it is irrelevant, whether its internal structure is object-oriented or not
- Development of specification-based test cases from OOA-diagrams
  - DFDs
  - State machines
  - Use cases according to Jacobson
    - Scenarios from a system user‘s viewpoint (human or other system)
    - Not very systematic
    - No completeness in complicated systems
    - Can be annotated with time conditions (MSCs) ➔ performance test!