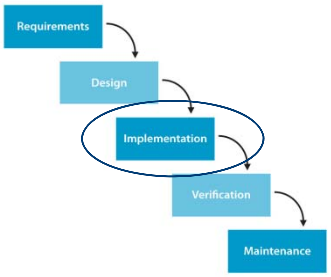
5

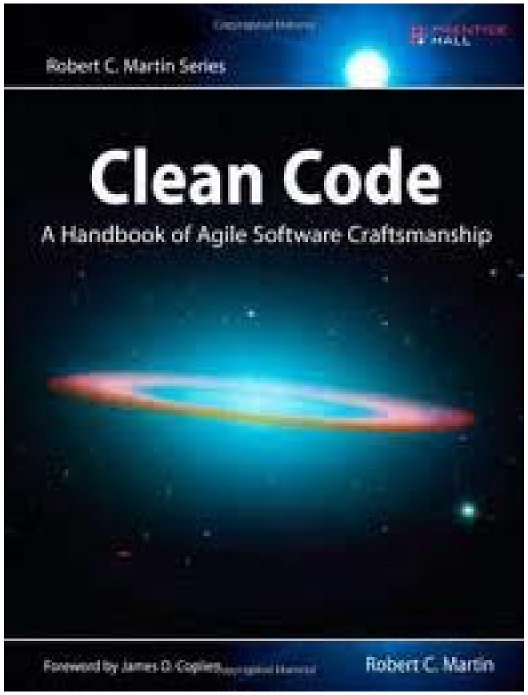
Implementation
Not in the book

Implementation
Learning Objectives



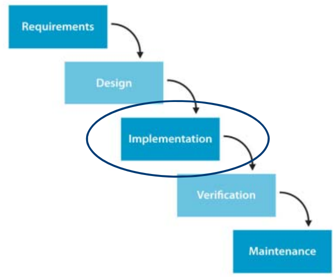
You can
- define cohesion and coupling
  - same as for modeling
- apply assertions
  - for online monitoring
- apply naming conventions
- write self-explaining code

Not in the book (or maybe Chapter 2?)



Implementation

High Cohesion

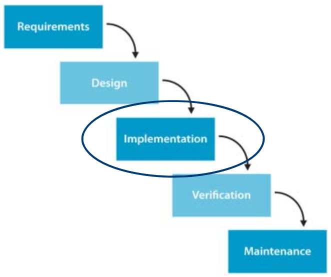


- Cohesion: group similar things together!
- Acceptable cohesion
  - Functional cohesion: one entity is responsible for ONE thing
  - Sequential cohesion: one entity is responsible for several things in a process, e.g. sharing data from step to step (e.g. class cohesion)
- Inacceptable cohesion
  - Procedural cohesion: operations are grouped because they are all part of one business process (goes against OO principles)
  - Coincidential cohesion: put the “leftovers” together

BUT: Crosscutting concerns → Aspect oriented programming (AOP)
- NF requirements (safety, security, logging, error handling, …)
- Can be weaved in automatically: AspectJ

Implementation

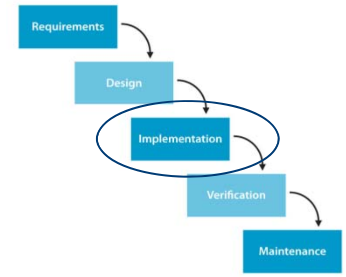
Low Coupling



- Coupling: entities that do not depend much on other entities
- Communication through small interfaces only
  - Implicit coupling → runtime platform (class library calls)
- Bad coupling
  - Global data (implicit coupling) → better call by variable/reference/object
- If you have an extensive interface (high coupling) between entities
  - maybe they should be combined?

Implementation

Assertions



- Stated and tested assumption about the execution of your code
  - Internal invariants
  - Control-flow invariants: “assert false” at any code location that cannot be reached
  - Preconditions: what must be true before a method can be called?
  - Post-conditions: what must be true after successful method completion?
  - Class invariants: what must be true about each class instance?

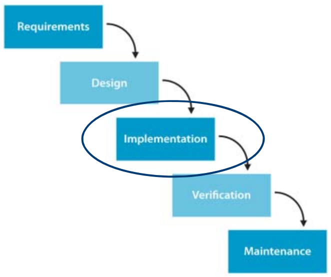
  can be seen as runtime monitoring of your code
- Should **NOT** be used for
  - argument checking in public methods
  - ensuring correct operation of your code

  Assertion error is thrown with a stack trace!

assert Expr1 : Expr2;

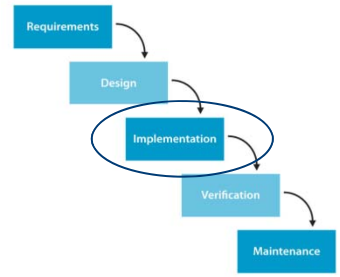
Implementation

Naming Conventions



- State in words what a variable is, and use domain lingo, e.g.
  ShippingRoute, LinesPerPage, MaxStackElements,
  NumOfDisabledCustomers , AccountAuthentication
- State objects in nouns, e.g.
  MinEntryLevel, AdmissionAge, PostalCode
- State operations in verbs, e.g.
  order, print, compile, ship, validate, login, edit, calculate
- Use short scratch values
  - temporary/local variables without domain semantics
  - e.g. *i, j, x, y, z, temp* (but Temperature!), *val* (but Valorisation)
  - don't use abbreviations/short names for domain variables
    - when you discuss code with the domain expert, they should be able to "find their domain in the code"

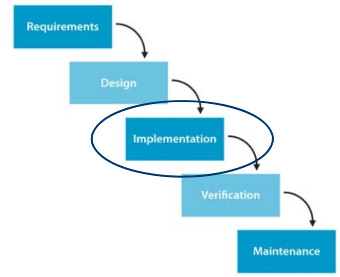
Implementation
Naming Conventions



- Computed value qualifiers in variable names, i.e. *Sum, Avg, Ref,* should go to the end of the variable name, e.g.
  OrderSum, RoutingRef, SalesAvg, ReservationIndex
- Loop indices should be meaningful
  i, j vs *element, pixel, range*, but
  element [i], pixel [j]
- Boolean variables; use yes/no names, e.g.
  found, done, error, success, OrderSaved, RoutePlanned
  Avoid negative names, e.g.
  notFound, notDone, unsuccessful, OrderNotSaved
- Don’t use variable names without semantics, e.g.
  Foo, Jeremy, Amstel, pooh, …

Implementation

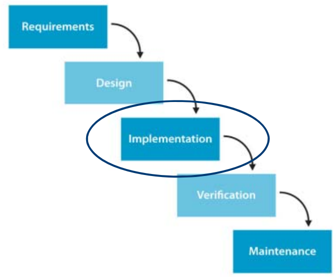
Naming Conventions



Avoid

- Ambiguous names, e.g. *SumOrder* & *OrderSum*, *RoutingRef* & *RoutingReference*, *RoutePlanned* & *RoutePlan*
- Similar names with different meaning, e.g. *RecNum* & *NumRecs*, *Input* & *Inval*, *ClientRecs* & *ClientReps*
- Names with similar sound, e.g. *wrap* & *rap*
  - Homonyms are difficult if you discuss your code
- Numerals in names, e.g. *File1* & *File2*
  - use concrete meaning instead
- Differentiation by capitalization, e.g. *file* & *File*, *customer* & *Customer*

Implementation
Self-Documenting Code

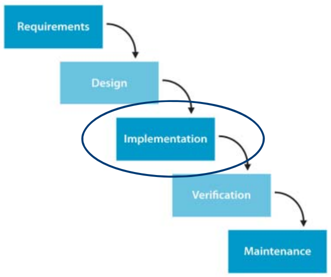


- Maintainability of the code

- External vs. internal documentation
  - Models (external)
    - High-level documentation
    - Difficult to align models and code (resulting in inconsistencies)
  - Code comments (internal)
    - Low-level documentation
  - *JavaDoc, Doxygen* (internal – external)
    - automatically generated documentation from code
    - in Latex or HTML

Implementation

Self-Documenting Code



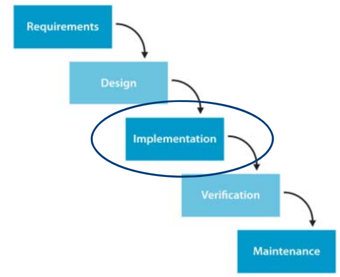
Checklist for internal documentation

- Does the source code contain most information about the class/program (establish commenting standard)?
- Can someone read and pickup the code and immediately understand it (do peer reviews)?
- Do comments explain the code's intent, in terms of *what* the code does rather than *how* it does it (repeating the code)?
- Has tricky code been rewritten rather than commented?
- Change the comments if you change the code!
- Does the code contain self-documentation?

```
...
// calculate sum
sum = a + b;
...
```

Implementation

Self-Documenting Code – Doxygen



Using Doxygen:

- in your code, use special commenting conventions
- indicate a comment-block (doxygen knows, this is a comment)

- JavaDOC-style with two *****'s

```
/**
 * ... comment text ...
 */
```

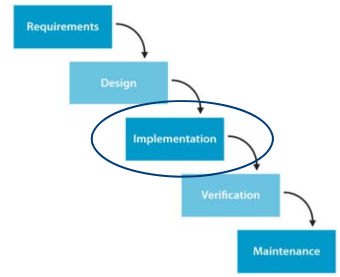
- Qt-style with !

```
/*!
 * ... comment text ...
*/
```

optional

Implementation

Self-Documenting Code – Doxygen



In every comment-block,
special keywords indicate more elaborate documentation.

/*! **@class** Name of the class
@brief Brief description.
Brief description continued.

Detailed description starts here.
*/

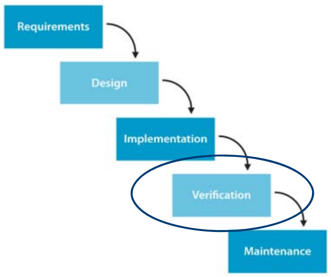
There is extensive *Doxygen* documentation available at the project home page: **www.doxygen.org**

6

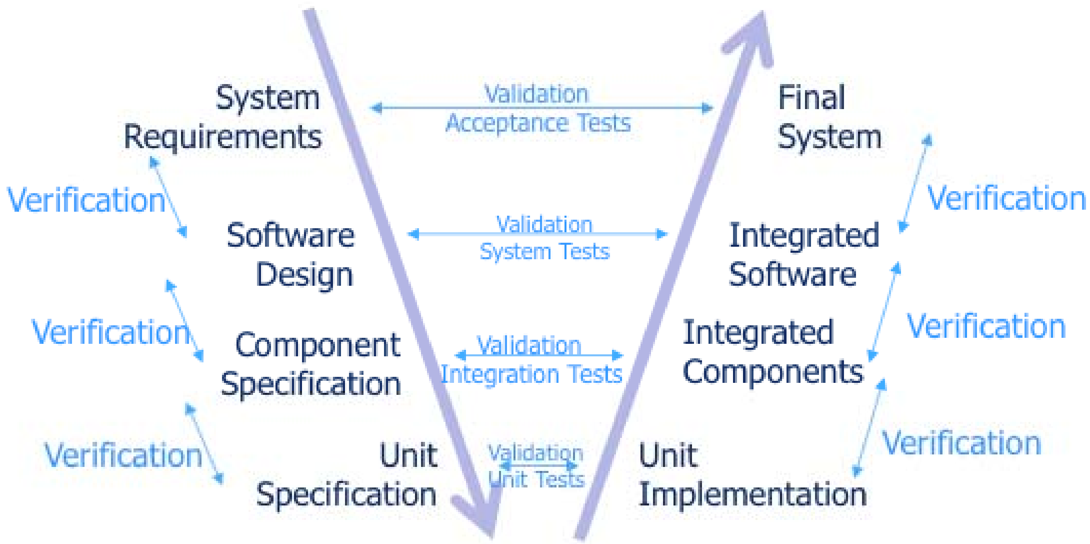
Verification & Validation (Test)

Verification vs. Validation

Recap



System / Software Quality



Verification (HOW?)
"Do we built the system right?"

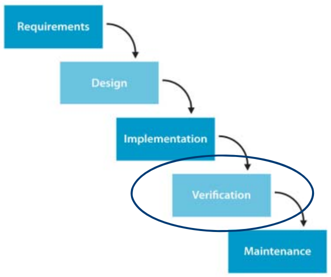
Validation (WHAT?)
"Do we built the right system?"

6.1

Verification
Book Chapter 10

Verification

"Do we built the system right?"



"How do we get from one document/model/artifact in the development process to the next document?"

3 Primary Issues: Are the development artifacts

- Correct
- Complete
- Consistent

Solutions: performed throughout the software life-cycle

Informal verification

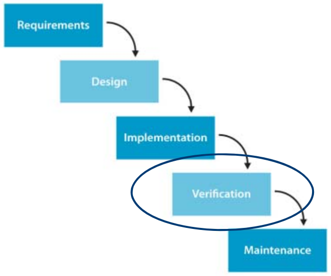
- inspection with reading techniques
- may be tedious (lacking tool support)

Formal verfication

- requires formal specification + tools
- mathematically daunting
- scalability problems for large systems

Informal Verification

Software Inspections



“Basically looking at (examining) the artifcat in a structured way.”

- Typically done as part of the project management (e.g. as part of Scrum)
- Recurring activity for all documents (models and code)

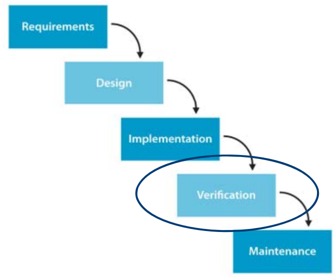
performed in two steps

- Individual reviews
  - Team members look at (inspects) other people’s documents
- Inspection meeting
  - Team members discuss problems identified; *“is it a bug, or is it a feature?”*
  - Refer to guidelines on performing effective inspection meetings (Book)

6.2

Validation (Test)
Book Chapter 10

Validation
Test Phases – Planning



“Do we build the right system?”, or “Is the right-hand-side of the V-model corresponding to the left-hand-side?”

in other words

“Does the implementation correspond to the specification?”

Phases follow the v-model (bottom up)

- Unit testing (every unit/class according to its **own** specification)
- Unit/Component integration testing (integration of packages/components)
- System (integration) testing (whole system incl. distributed components)
- Application acceptance testing (whole system according to user scenarios)

Aim at **high** code/model/requirements **coverage**

Validation

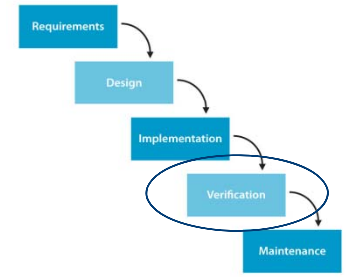
Unit Test

Test-Driven Development

"Before you implement a feature write a test for it!"

"If you cannot write a test for a feature, don't implement it!"

- UnitTest framework (e.g. JUnit, www.junit.org)
- Every business class gets an associated JUnit test class
- All functions should be covered




```
import org.junit.*;
import static org.junit.Assert.*;
import java.util.*;
import ... //your tested class

public class SimpleTest {

    private Collection collection;
```

helper objects

```
    @BeforeClass
    public static void oneTimeSetUp() {
        // one-time initialization code
    }
```

setup for all test cases

```
    @AfterClass
    public static void oneTimeTearDown() {
        // one-time cleanup code
    }
```

teardown for all test cases

```
    @Before
    public void setUp() {
        collection = new ArrayList();
    }
```

setup for each test case

```
    @After
    public void tearDown() {
        collection.clear();
    }
```

teardown for each test case

```
    @Test
    public void testEmptyCollection() {
        assertTrue(collection.isEmpty());
    }
```

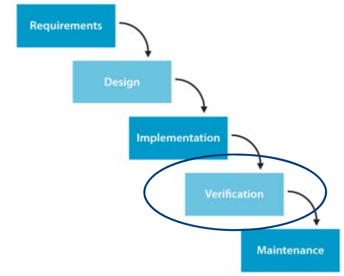
test case

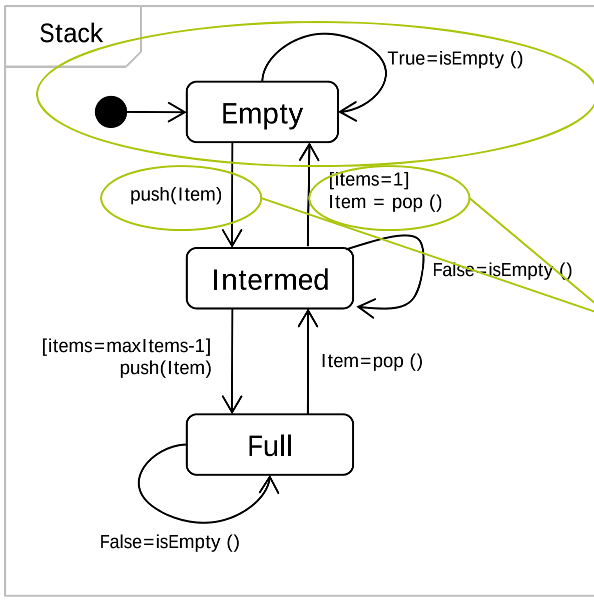
```
    @Test
    public void testOneItemCollection() {
        collection.add("itemA");
        assertEquals(1, collection.size());
    }
}
```

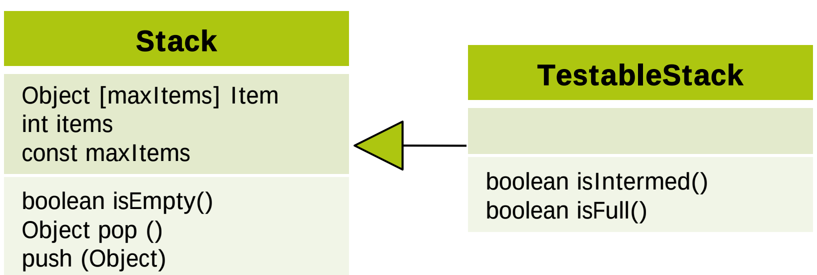
test case

Validation

Unit Test Example







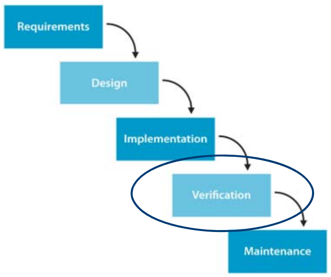

```
import org.junit.*;
import static org.junit.Assert.*;
import java.util.*;

public class SimpleTest {

    private Stack s;
    private Object Item ...

    // before ... after ....

    @Test
    public void testEmptyStack() {
        assertTrue(s.isEmpty());
    }

    @Test
    public void testOneItemPush() {
        s.push(Item);
        assertFalse(s.isEmpty());
        assertEquals(1, s.items); // if items is public
    }

    @Test
    public void testOneItemPosp() {
        assertTrue(Item == s.pop());
        assertTrue(s.isEmpty());
    }

    .....

}
```

Aim for high model coverage

Model is the test oracle

Validation

Integration & System Test



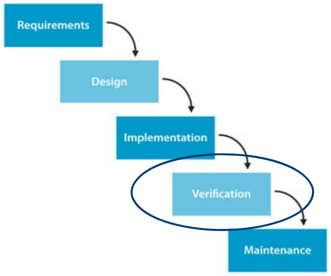
Integration Test

- same as Unit Test with several units integrated as components
- use higher-level models as test oracles
- aim at high model-coverage of the design models
- can be done with JUnit

System Test

- same as Integration Test with all units integrated
- use system models as test oracles
- aim at high coverage of the system models
- can be done with JUnit

Validation
Acceptance Test

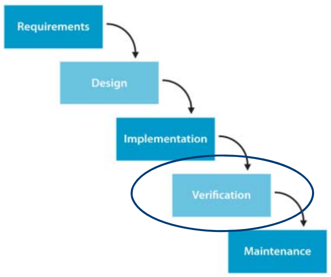


Test from the viewpoint of the user

- perform the use case scenarios, business processes
- aim at high coverage of the system usage
- cannot be done with JUnit (User Interface)
- difficult to automate

Validation

Terminology



Fault – Error – Failure

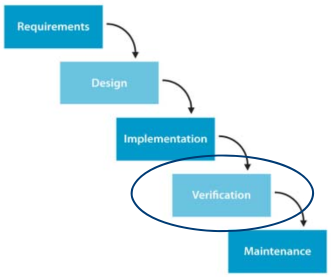
- Fault: Root cause of the error/failure (e.g., in the code)
- Error: Erroneous state in the system caused by the fault
- Failure: External observation of misbehavior (difference between observation and expectation)

Test Case

- Controlled experiment
- Stimulus (invocation)
- Expectation (model=oracle, pre- and post-conditions)
- Observation (execution, observability!)
- Verdict (pass/fail, difference between model and observation)

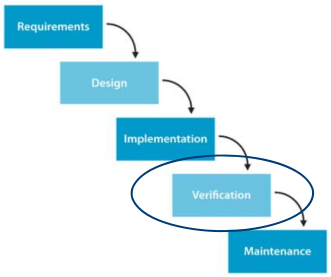
Validation

Terminology



- Failure
  - Testing
- Error
  - Monitoring
  - Debugging
  - Tracing
- Fault (Bug)
  - Debugging
  - Tracing
  - Diagnosis

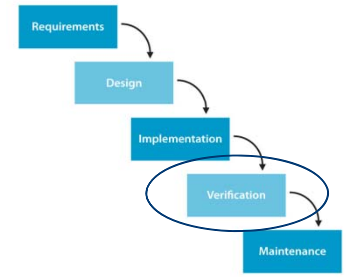
Validation
Terminology

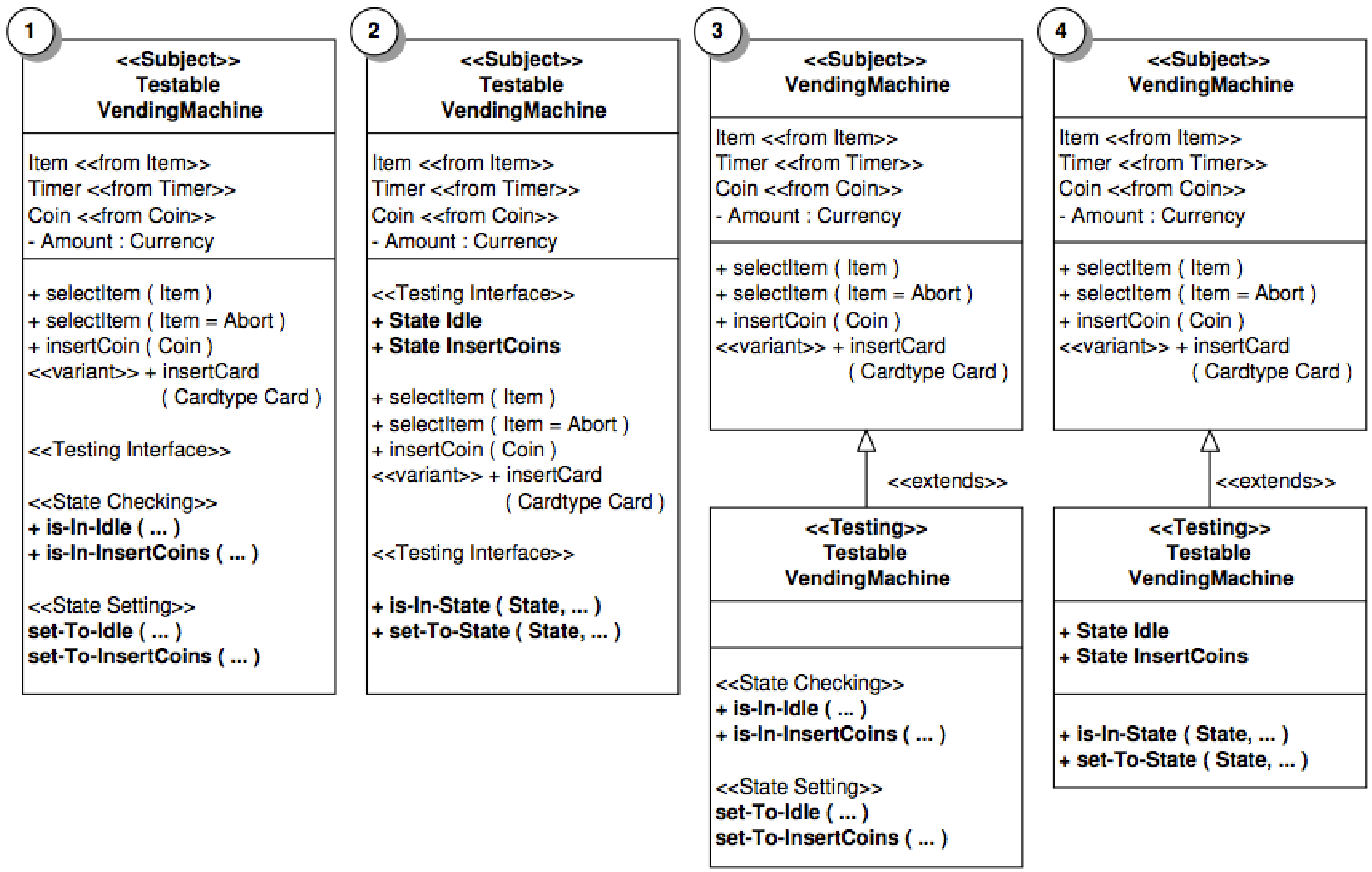


- Controllability vs. Observability
  - What you cannot control, you cannot test
  - What you cannot observe, you cannot control
  - Aim for high controllability and high observability
    - Output states
    - Test interfaces
    - Query interfaces (states)
- Aim for high testability
  - Ability of the testing technique to uncover faults in the unit
  - Ability of the unit to hide faults from the testing technique

Validation – Built-in Testing

Terminology





Validation – Built-in Testing

Terminology