7. Goals of Software Design

Design faces many challenges to produce a good product, e.g. shifting requirements.

But what do we mean by ***good*** ?

We need some clear goals here …

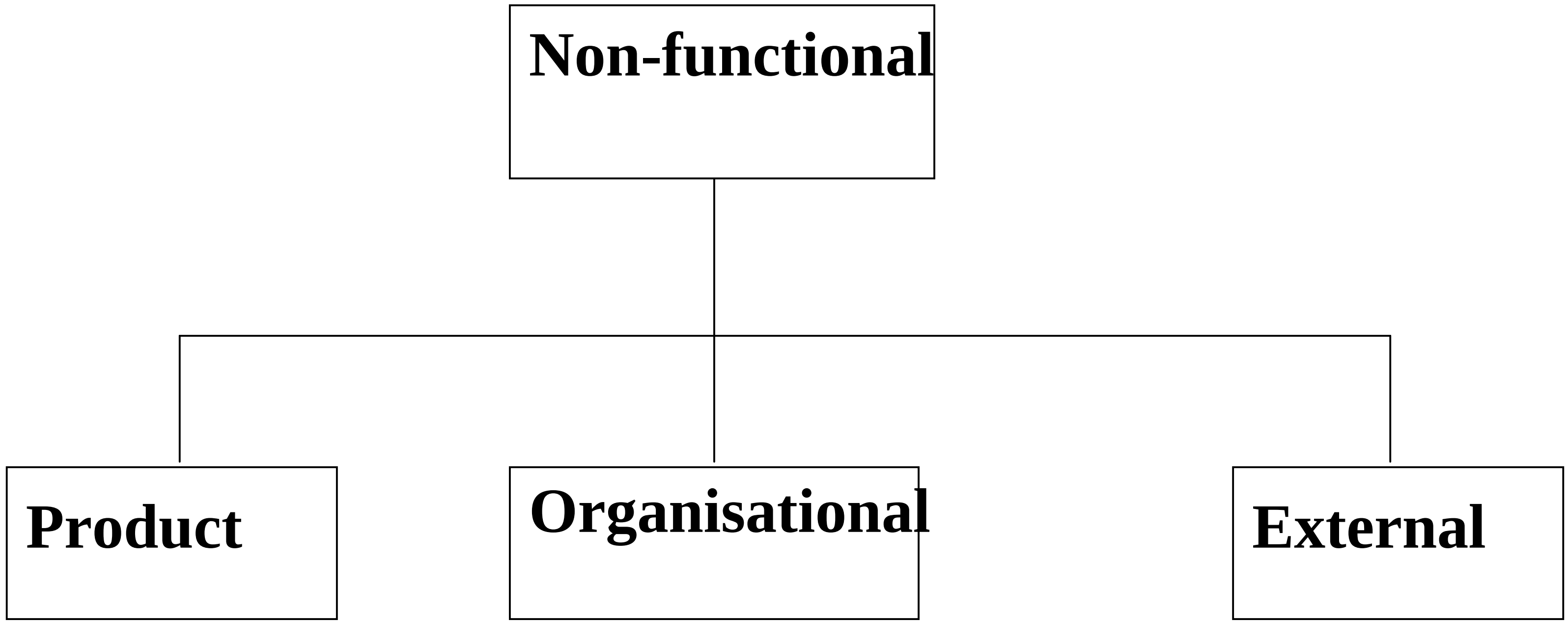
Good design leads to software that is:

1. **Correct** – does what it should
2. **Robust** – tolerant of misuse, e.g. faulty data
3. **Flexible** – adaptable to shifting requirements
4. **Reusable** – cut production costs for code
5. **Efficient** – good use of processor and memory

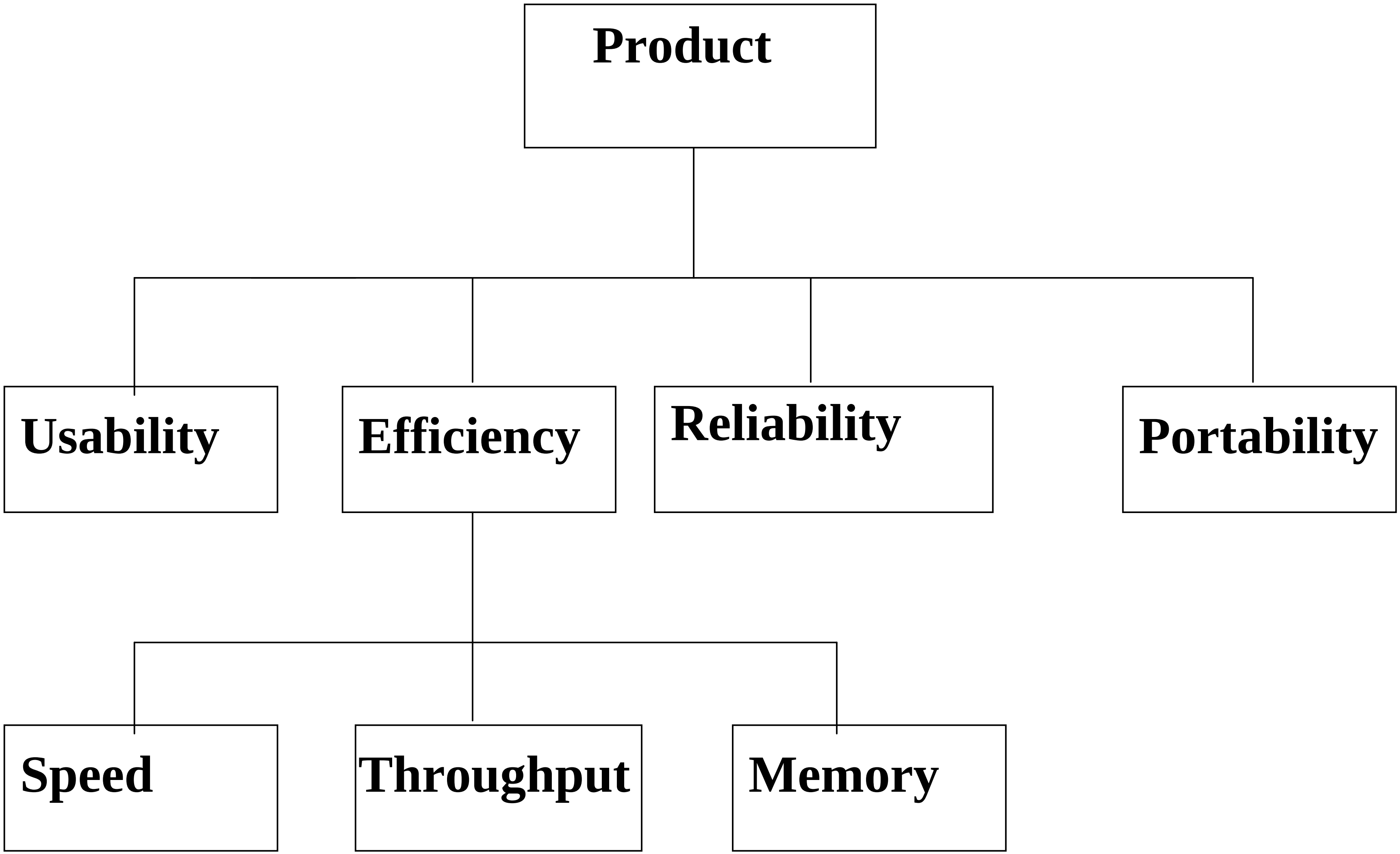
Also should be **Reliable** and **Usable**

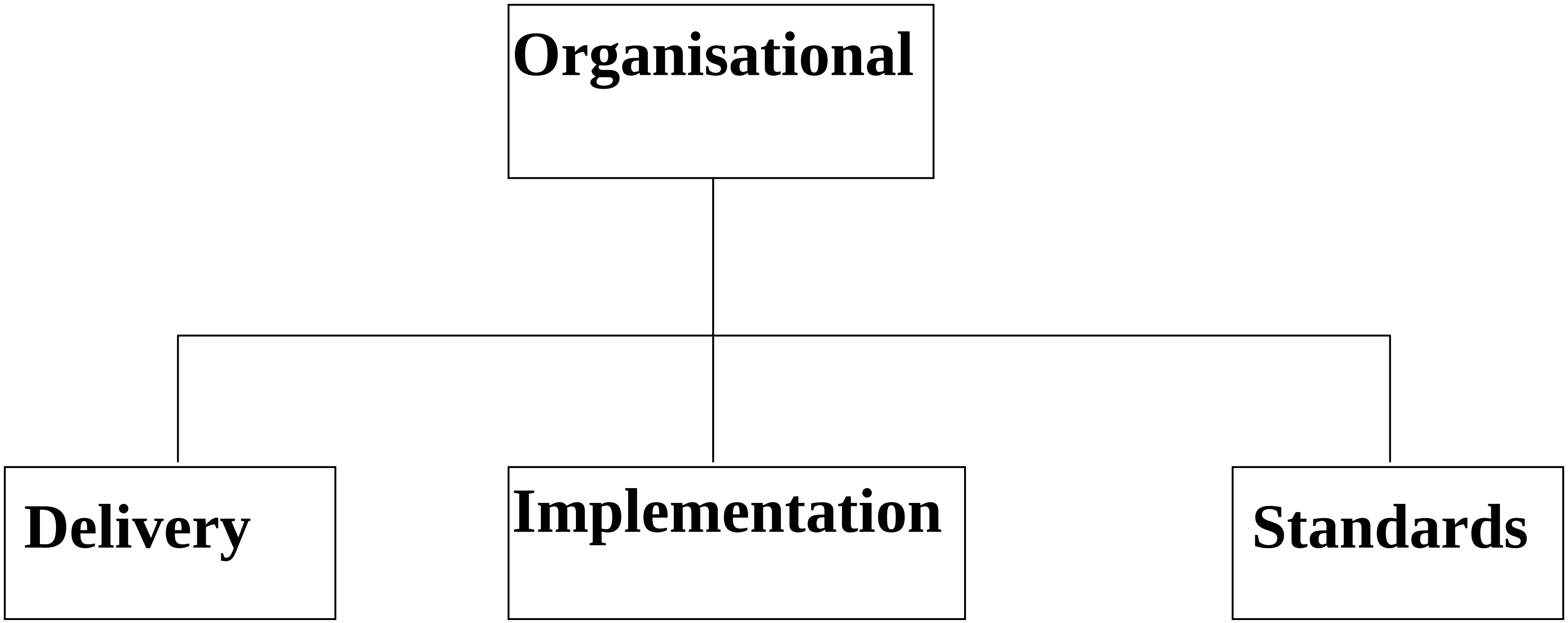
7.1. Correctness

- Software is correct, if it **satisfies its requirements**.
- A primary goal, incorrect software may look good, but will be poor or worse.
- Requirements are divided into **functional** (what it does) and **non-functional** (how it does it, etc.)



Non-functional
Product
Organisational
External

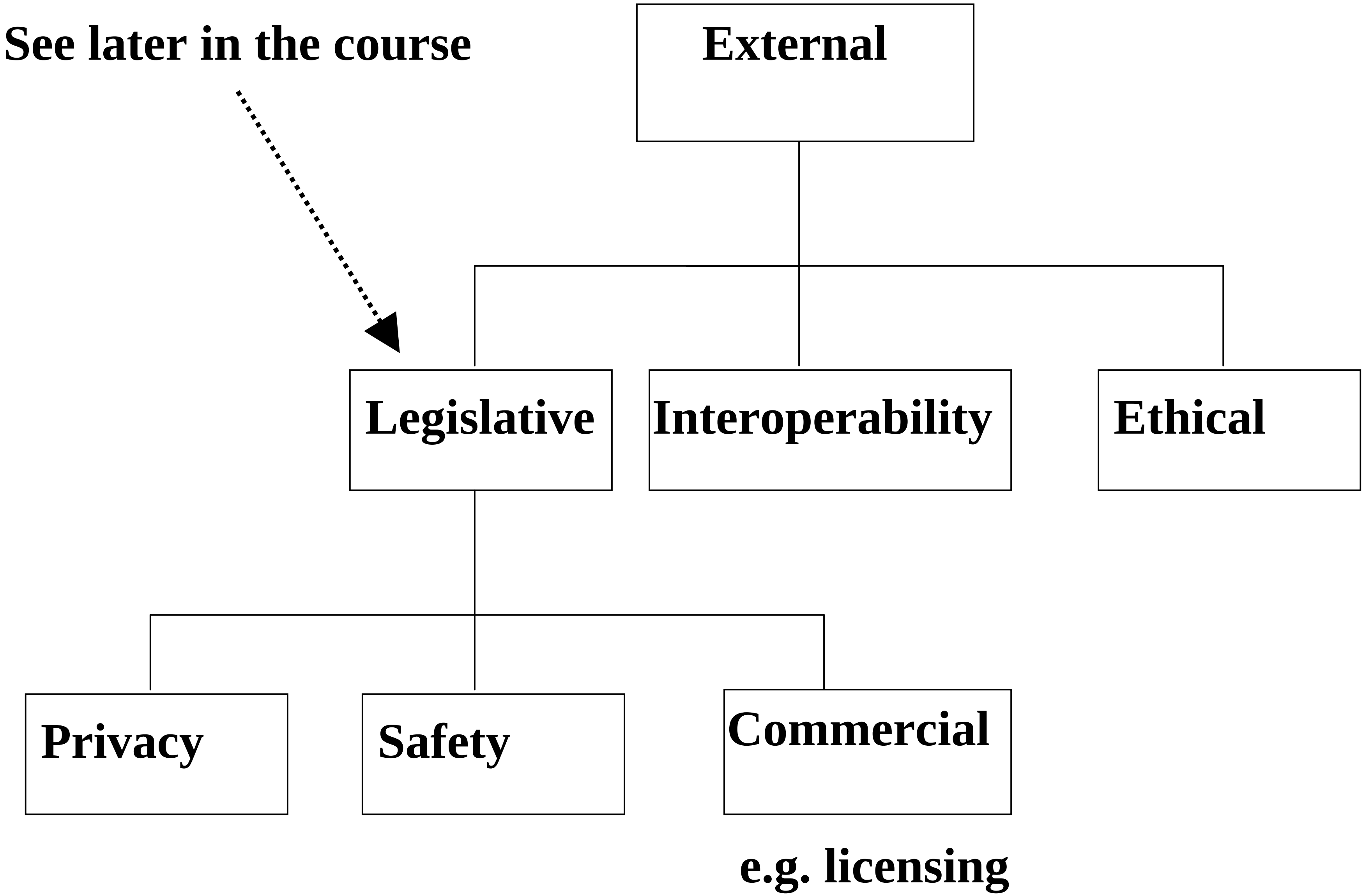

Product
Usability
Efficiency
Reliability
Portability
Speed
Throughput
Memory



e.g. quality of documentation, manuals, training, support

e.g. commenting, flexibility, openess, maintainability

e.g. ISO 9000


See later in the course
External
Legislative
Interoperability
Ethical
Privacy
Safety
Commercial
e.g. licensing

Requirements may be expressed in …

- **Text**

  "f() computes a square root function"

  also, text use cases.

- **Logic**

  " Tol $\geq$ | $f(x)^2 - x$ | "

  (see also UML Object Constraint Language)

- **Diagrams** "output should look like this"
- **UML Sequence Diagrams** "system should behave like this"

Many functional requirements can be expressed in terms of a **precondition**

e.g. ***"if the input is a positive integer"***

and a **postcondition**

e.g. ***"the output will be the square root of the input".***

$$\{ \text{ in} \geq 0 \} \text{ system } \{ \text{ Tol} \geq | \text{ out}^2 - \text{in} | \}$$

7.1.1. Verification

- Checking software against its requirements is called verification. Three main approaches:

1. **Testing**,
2. **Formal Verification**
3. **Code Inspections**

Testing takes a **contrarian** approach, falsify correctness claim by finding a counterexample to correctness, i.e. a test case that fails,

Thus **precondition is true, but postcondition is false**.

Testing can only uncover errors, it can never prove a system is correct.

Formal verification takes a mathematical approach:

Step 1: model the code mathematically,
Step 2: model the requirement mathematically
Step 3: prove "code satisfies requirement"

Important approach to **safety critical systems**: *medical*, *avionics*, *automobile*, *nuclear power*, *financial systems*, *smart cards*, *etc*.

<u>Code Inspection</u> is a manual process of code walk-through (discussion).

- Usually done in front of a group or team.
- Not mathematical, but more systematic than testing.
- Has been empirically shown to be more cost effective than manual testing – bugs found per dollar.
- But testing and formal verification getting cheaper through automation.

7.2. Robustness

A design or system is **robust** if it tolerates misuse without catastrophic failure.
aka. **fault-tolerant**.
Includes bad data, bad use, bad environment, bad programming.

Robustness achieved in many ways:

- Use data abstraction and encapsulation
  - Create ADTs and simple interfaces
  - Shield from data corruption
- Initialize variables
- Qualify all inputs (e.g. range check)
  - Same as precondition checking
- Qualify all formal parameters to a method
- Qualify invariants
  - (e.g. non-null pointer, not end_of_file)
- Qualify postconditions

7.3. Flexibility

Requirements may change during or after the project.

- Obtaining more of what's present

  e.g. more kinds of different bank accounts

- Adding new kinds of functionality

  e.g. add internet banking to teller functionality

- Changing functionality

  e.g. allow withdrawals to create an overdraft

Flexibility achieved in many ways:

- Encapsulation (representation hiding)
- Different types of the same base category by means of abstract classes
- Extend functionality by new class methods or with an abstract class & several derived classes.

7.4. Reusability

Aim: cut cost of code production over 1 or more projects.

- Reuse object code (see later discussion of component technologies)
- Reuse source code – see next slides
- Reuse assemblies of related classes, e.g. software frameworks
- Reuse patterns of designs – see previous!

7.4.1. Promoting Source Code Reuse

- Use modularity
  - Use classes and interfaces which are independent and as general or specific as necessary
- Use Classes
  - Describe class with good <u>name</u> & <u>documentation</u>
  - Minimize <u>dependency</u> between classes
  - Maximally abstract and <u>general</u> ***or*** precisely <u>matched</u> to real objects and their function

7.4.1. Continued

- Write good methods
  - Explain the algorithm
  - Use good names
  - Specify pre + postconditions + invariants
  - Don’t couple closely to class

7.5. Efficiency

Aim: Make greatest use of the processing power, memory size, network speed, etc.

- But all these things are getting cheaper!
- But applications are getting bigger!
- Efficiency is often against the first 4 goals!

- Efficiency is often achieved by writing clever algorithms and data structures. Needs a good education in this area.
- Efficiency can also be obtained by “profiling” code – search for execution intensive code sections with code profiler, try to optimize these.
- Sometimes write low level routines, e.g. assembler.
- Use an optimizing compiler.
- Use a Java compiler – throw away portability.

7.6. Reliability & Usability

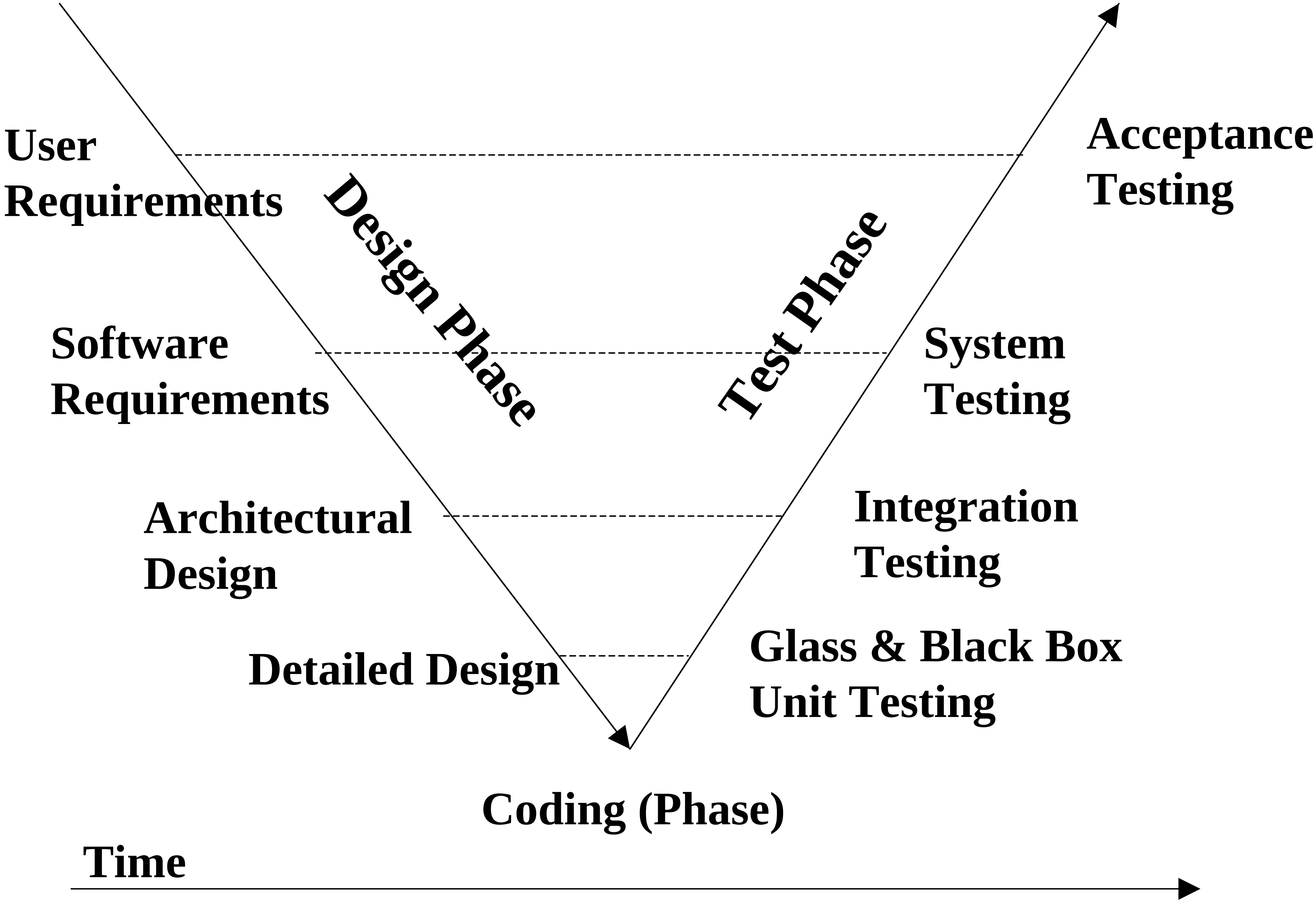
- **Reliability** – mean time to failure (system crash, error)
- On architectural level can use hardware support, backup servers, multiple processors, hot swap, etc
- On code level achieved by software quality assurance methods, testing, walkthroughs, formal methods etc.

- **Usability** – users must find software easy to easy.
- Intuitive GUI, standard layout & meanings
- Good documentation,
- Well known use metaphor – desktop, calculator, tape recorder, etc.
- Connections to ergonomics & cognitive psychology
- Hard to define and measure, user interviews, questionnaires, etc.

8. Testing

- Defines program functionality (partly)
- Can be used as documentation (c.f. XP!)
- Ensure program **is** testable
- Methods become callable
- Modules get a looser coupling
- If written first, then we avoid risk of testing code we know works instead of code which should work.

8.1. The “V” Model : Workflow



8.2. Types of Testing

8.2.1. Unit Testing

- Tests a basic component, module
- Glass box/structural test – path analysis
- Coverage measures

8.2.2. Regression Testing

- Does new functionality disturb existing functionality?
- Keep old test suite + old results … rerun.

8.2.3. System testing

- Black-box testing, structure of code is invisible
- Test the specification, not the code!
- Hard to find the tests … oracle problem
- Hard to define coverage
- Volume of testing, usage profiles?
- Use cases are an excellent source of tests.

UseCaseName **PurchaseTicket**

Precondition

The Passenger is standing in front of ticket Distributor and has sufficient money to purchase ticket.

Sequence

1. The Passenger selects the number of zones to be traveled. If the Passenger presses multiple zone buttons, only the last Button pressed is considered by the Distributor.
2. The Distributor displays the amount due.
3. The Passenger inserts money.

4. If the Passenger selects a new zone before inserting sufficient money, the Distributor returns all the coins and bills inserted by the Passenger.
5. If the Passenger inserted more money than the amount due, the Distributor returns excess change.
6. The Distributor issues ticket
7. The Passenger picks up the change and the ticket.

TestCaseName **PurchaseTicket_SunnyCase**

Precondition

The Passenger is standing in front of ticket Distributor and has two £5 notes and 3 * 10p coins

Sequence

1. The Passenger presses in succession the zone buttons 2, 4, 1, and 2.
2. The Distributor should display in succession £1.25, £2.25, £0.75 and £1.25
3. The Passenger inserts a £5 note.
4. The Distributor returns three £1 coins 75 and a 2-zone ticket.

Postcondition

The Passenger has one 2-zone ticket.

- We should also derive test cases from the use case that exercise rainy day scenarios (when something goes wrong).

8.2.4. Acceptance Testing

– On-site by the customer
– Tests come from requirements document
– Legal/contractual consequences
– Affected by the ***real*** environment.

8.3. Unit Testing with JUnit

- Developed by XP community 2002
- Framework for automating the execution of unit tests for Java classes.
- Write new test cases by subclassing the TestCase class.
- Organise TestCases into TestSuites.
- Automates testing process
- Built around Command and Composite patterns

8.3.1. Why Use JUnit?

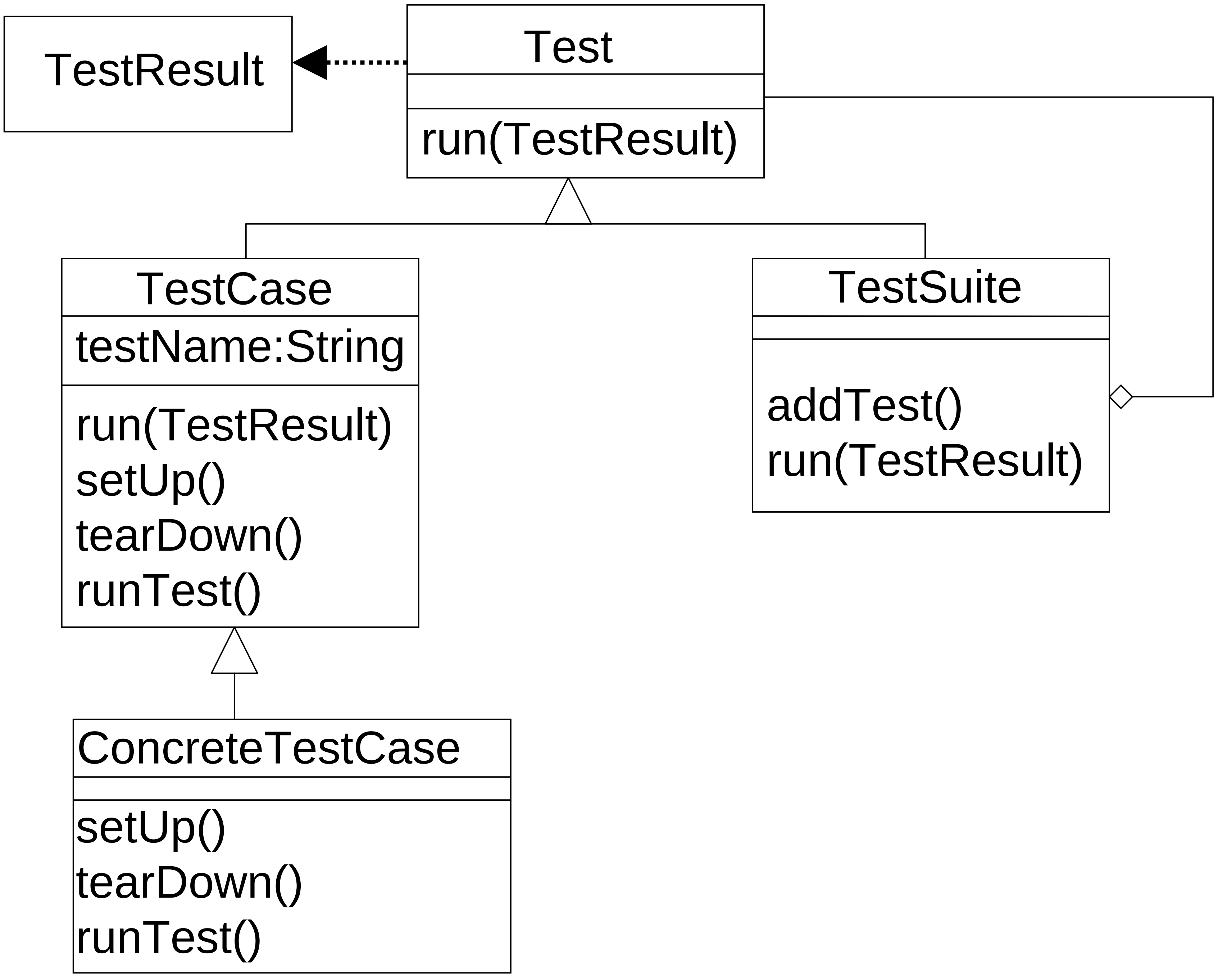
- JUnit tightly **integrates** development and testing, supports the XP approach
- Allows you to write code **faster** while increasing quality (!!!)
  - Can refactor code without worrying about correctness.
- JUnit is **simple**
  - Easy as running the compiler on your code.

- JUnit tests **check** their own results and provide immediate **feedback**.
  - No manual comparison of expected with actual
  - Simple visual feedback
- JUnit tests can be composed into a **hierarchy** of test suites.
  - Can run tests for any layer in hierarchy
- Writing JUnit tests is **inexpensive**
  - No harder than writing a method to exercise the code.
- JUnit tests increase **stability** of software
  - More tests = more stability

- JUnit tests are **developer tests**
  - Test fundamental building blocks of system
  - Tests delivered with code as a certified package
- JUnit tests are **written in Java**
  - Seamless bond between test and code under test
  - Test code can be refactored into software code and vice-versa
  - Data type compatibility (floats etc)
- JUnit is **free**!

8.3.2. JUnit Design

- A TestCase is a Command object.
- A class of test methods subclasses TestCase
- A TestCase has public testXXX() methods
- To check expected with actual output invoke assert() method
- Use setUp() and tearDown() to prevent side effects between subsequent testXXX() calls.


TestResult
Test
run(TestResult)
TestCase
testName:String
run(TestResult)
setUp()
tearDown()
runTest()
TestSuite
addTest()
run(TestResult)
ConcreteTestCase
setUp()
tearDown()
runTest()

- TestCase objects can be composed into TestSuite hierarchies. Automatically invoke all the testXXX() methods in each object.
- A TestSuite is composed of TestCase instances or other TestSuite instances.
- Nest to arbitrary depth
- Run whole TestSuite with a single pass/fail result.
- See course web page for installation instructions.

8.3.3. Writing a Test Case

1. Define a subclass of TestCase
2. Override the setUp() method to initialize object(s) under test.
3. Optionally override the tearDown() method to release objects under test.
4. Define 1 or more public testXXX() methods that exercise the object(s) under test and assert expected results.

```
import junit.framework.TestCase;

public class ShoppingCartTest extends TestCase {
    privateShoppingCart cart;
    private Product book1;

    protected void setUp() {
        cart = new ShoppingCart();
        book1 = new Product (“myTitle”, “500SEK”);
        cart.addItem(book1);
    }
```

```
protected void tearDown() {
    // release objects under test here, if necessary
}
public void testEmpty() {
    cart.empty();
    assertEquals(0, cart.getItemCount() );
}
```

```
public void testAddItem() {
    product book2 = new Product(“title2””650SEK”);
    cart.addItem(book2);
    double expectedBalance =
          book1.getPrice() + book2.getPrice();
    assertEquals(expectedBalance, cart.getBalance(),
    0.0);
    assertEquals(2, cart.getItemCount() );
}

public void testRemoveItem() throws
    productNotFoundException {
          cart.removeItem(book1);
          assertEquals(0, cart.getItemCount() );
}
```

```
public void testRemoveItemNotInCart() {
    try {
    Product book3 = new Product(“title3”, “100SEK”);
    cart.removeItem(book3);
    fail(“Should raise a
    ProductNotFoundException”);
    }
    catch(ProductNotFoundException expected) {
    // passed the test!
    }
}
} // of class ShoppingCartTest
```

8.3.4. Writing a Test Suite

1. Write a Java class that defines a static suite() factory method that creates a TestSuite containing all the tests.
2. Optionally define a main() method that runs the TestSuite in batch mode.

```
import junit.framework.Test;
import junit.framework.TestSuite;

public class EcommerceTestSuite{
public static Test suite() {
    TestSuite suite = new TestSuite();
    // Add one test case
    suite.addTestSuite(ShoppingCartTest.class);
    // Add a suite of test cases
    suite.addTest(CreditCardTestSuite.suite());
    return suite;
}
public static void main(String[ ] args) {
    junit.textui.TestRunner.run(suite( ) ); } }
```

8.3.5. Running Tests

- Running a TestCase runs all its public testXXX() methods
- Running a TestSuite runs all its TestCases and subordinate TestSuites.
- **<u>Text user interface</u>**

  java junit.textui.TestRunner ShoppingCartTest
- **<u>Graphical user interface</u>**

  java junit.swingui.TestRunner EcommerceTestSuite

8.3.6. Graphical User Interface

8.3.7. Testing Idioms

- Software does well what tests check
- Test a little, code a little, test a little , …
- Run all tests at least once a day
- Write tests for areas of code with highest probability of error
- Write unit tests before writing the code and only write new code when a test is failing.