GRNSPG Internship

Introduction to Software Engineering and Digital Systems Reliability

Pisa, 15 June, 2009

Testing, Verification and Validation of I&C Systems

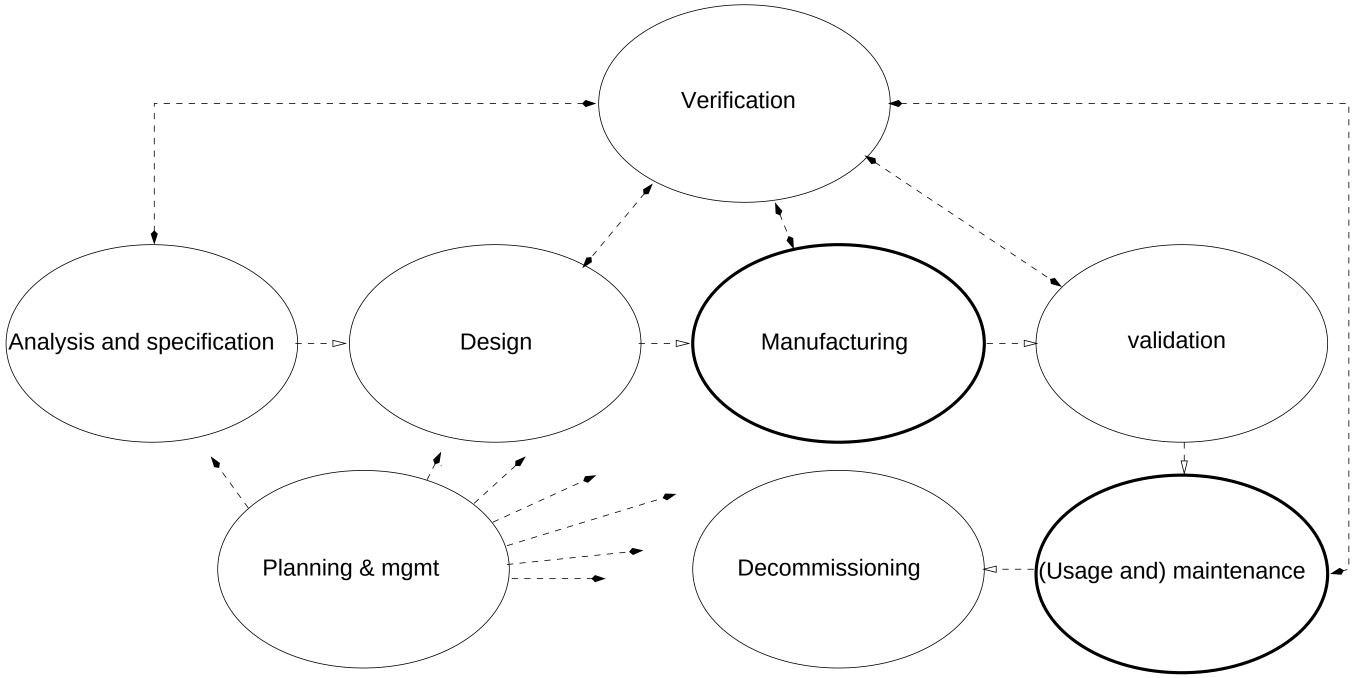
Andrea Domenici

DIIEIT, Università di Pisa

Andrea.Domenici@iet.unipi.it

SOFTWARE LIFECYCLE AND PROCESSES

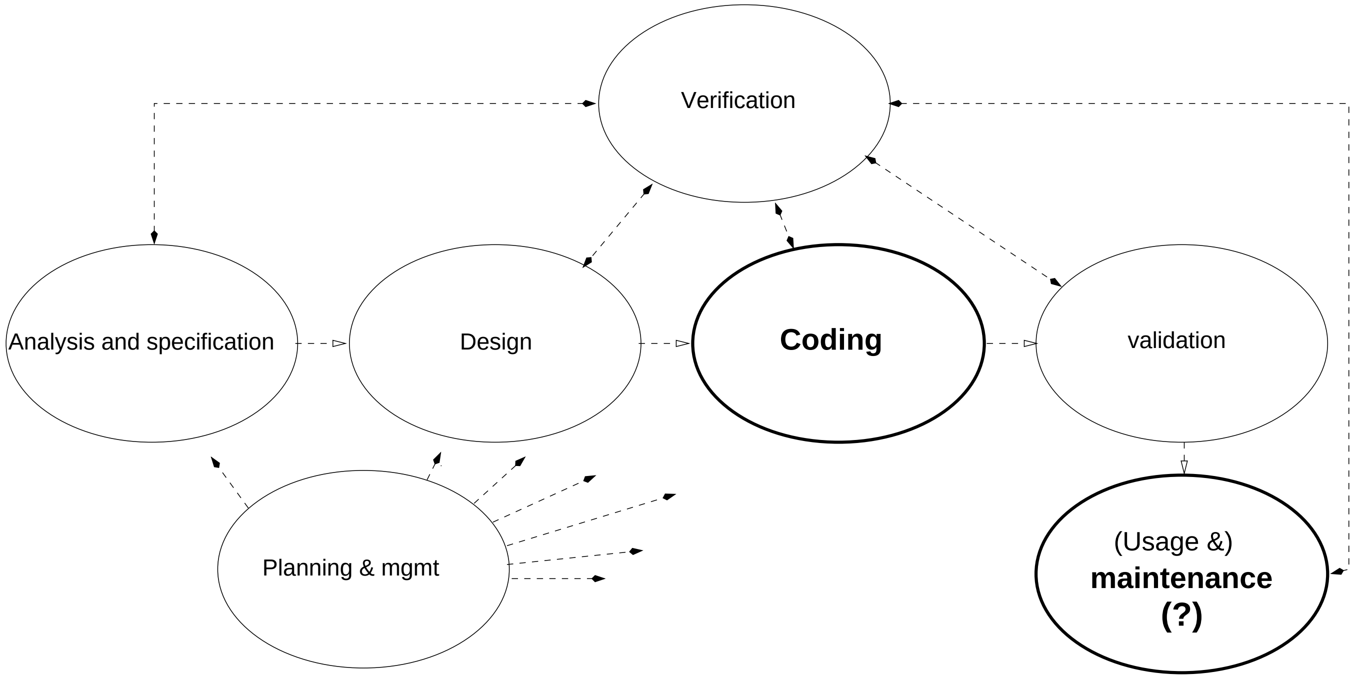
Generic product lifecycle



The *lifecycle* of a product is the set of *activities* affecting it from conception to retirement.

The lifecycle of software products is *similar* to the general lifecycle, **but** there are important differences.

Software lifecycle

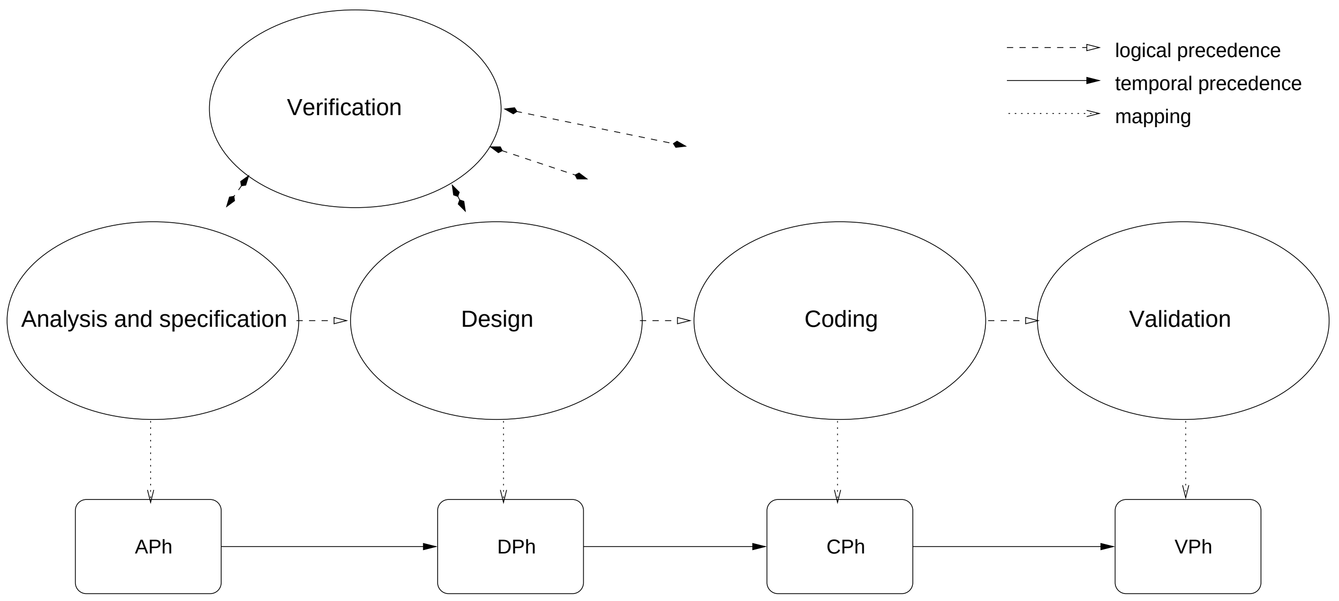


Coding is actually the final phase of design.

Software maintenance is actually redesign/recoding to correct or improve deployed software components.

In this seminar we ignore issues about software decommissioning.

Software development processes (1)

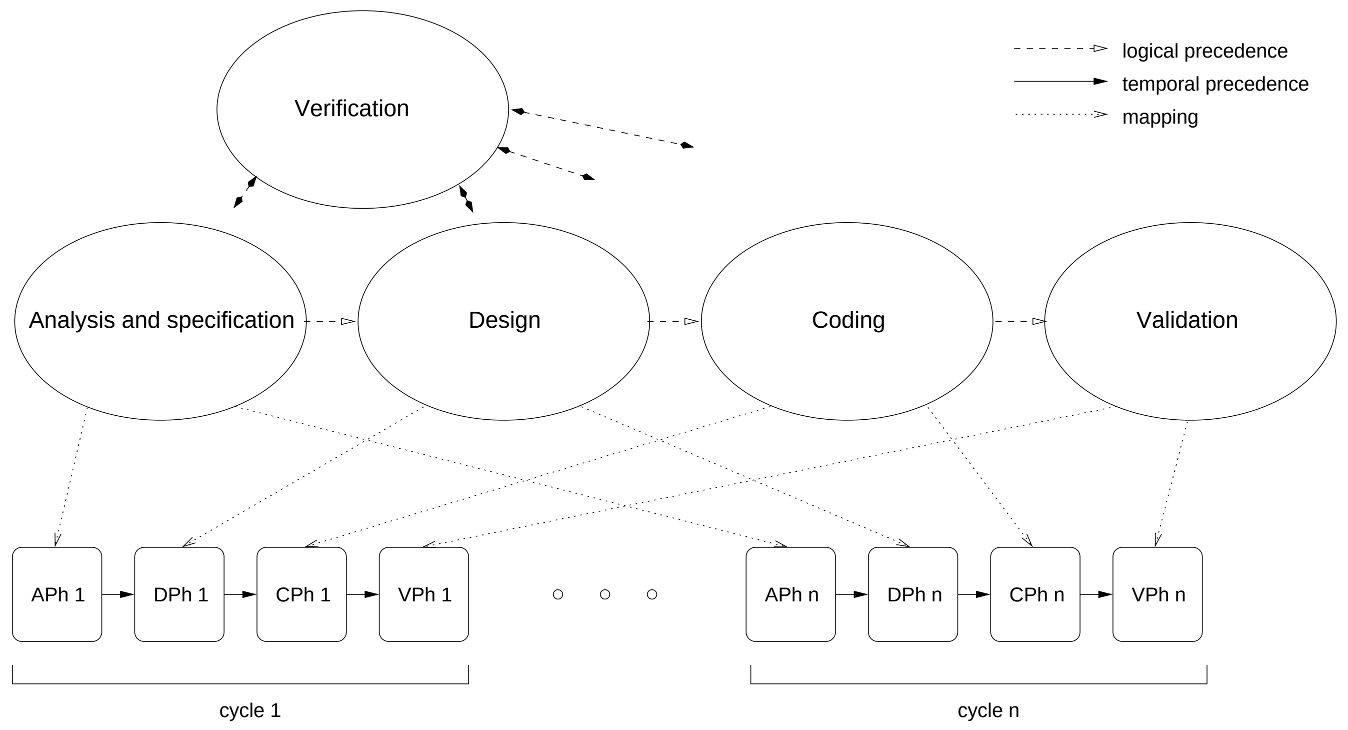


A *software development process* maps activities (i.e., things to be done) into *phases* (i.e., periods wherein activities are carried out).

Each phase has well-defined *milestones* (important events at planned dates) and produces some *deliverables* (documents or code).

The *waterfall* family of processes maps each activity to a distinct phase.

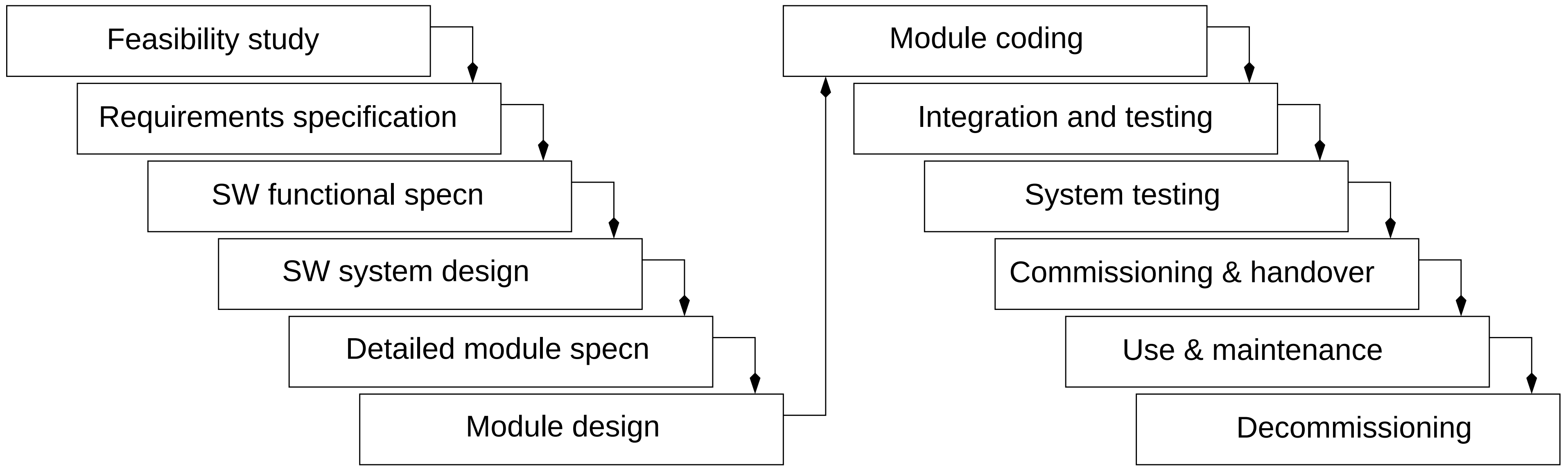
Software development processes (2)



Other families of processes, such as the *iterative* processes, have more complex mappings from activities to phases.

The waterfall processes, however, are usually preferred (or mandatory) for software with safety requirements.

The IAEA TRS 282 waterfall model



International Atomic Energy Agency, *Manual on Quality Assurance for Computer Software Related to the Safety of Nuclear Power Plants*, Tech. Reports Series No. 282, IAEA, Vienna, 1988.

The Gullfoss waterfall model

(brunnur.stjr.is/embassy/strasb.nsf/pages/index.html)

REQUIREMENTS ANALYSIS AND SPECIFICATION

Requirements

- A *requirement* is a statement about a relevant aspect of a system that must be developed: a service it must deliver, a property or constraint it must satisfy, and so on.
- Some requirements may concern the development process itself, rather than the system; for example, it may be a requirement that the system is developed following certain standard procedures.

Functional and non-functional requirements

- *Functional* requirements describe the services offered by the system in terms of relationships between inputs and outputs.
- A system that satisfies its functional requirements is said to be *correct*.
- *Non-functional* requirements describe properties of or constraints on the system or process.
- Some non-functional requirements are *reliability*, *robustness*, *safety*, and *performance*.

Constraints

- *Constraints* are non-functional requirements that impose limitations on design choices.
- *Time constraints* are particularly important in control systems.
- *Synchronization* constraints impose some *ordering* among events, without specifying bounds on time intervals between events.
  - E.g.: “*Valve B must not be opened before valve A*”.
- *Real-time* constraints impose *bounds* on time intervals between events.
  - E.g.: “*Valve B must be opened between 10 and 20 seconds after valve A*”.

Note that a real-time system is not necessarily a fast system. *Predictability* is more important than speed.

Analysis and specification

- Requirements *analysis* is the process of understanding what is required of a system.
- Requirements analysis involves studying the application domain, so that software developers may understand the relationship of the software with its environment.
- A requirements *specification* is the result of the requirements analysis process, i.e., a document that gives a precise and complete description of the requirements.
- The analysis process results in the construction of a *model*, i.e., an abstraction of the relevant aspects of the system and its environment.
- Using a specific analysis method helps making this model and its underlying assumptions more *explicit* and *verifiable*.

Traceability

Requirements should be traceable to design; design should be traceable to code; requirements, design and code should be traceable to tests.

Traceability should be maintained when changes are made.

There should be traceability in the reverse direction, to ensure that no unintended functions have been created.

IAEA Safety Guide *Software for Computer Based Systems Important to Safety in Nuclear Power Plants*, NS-G-1.1

Testability

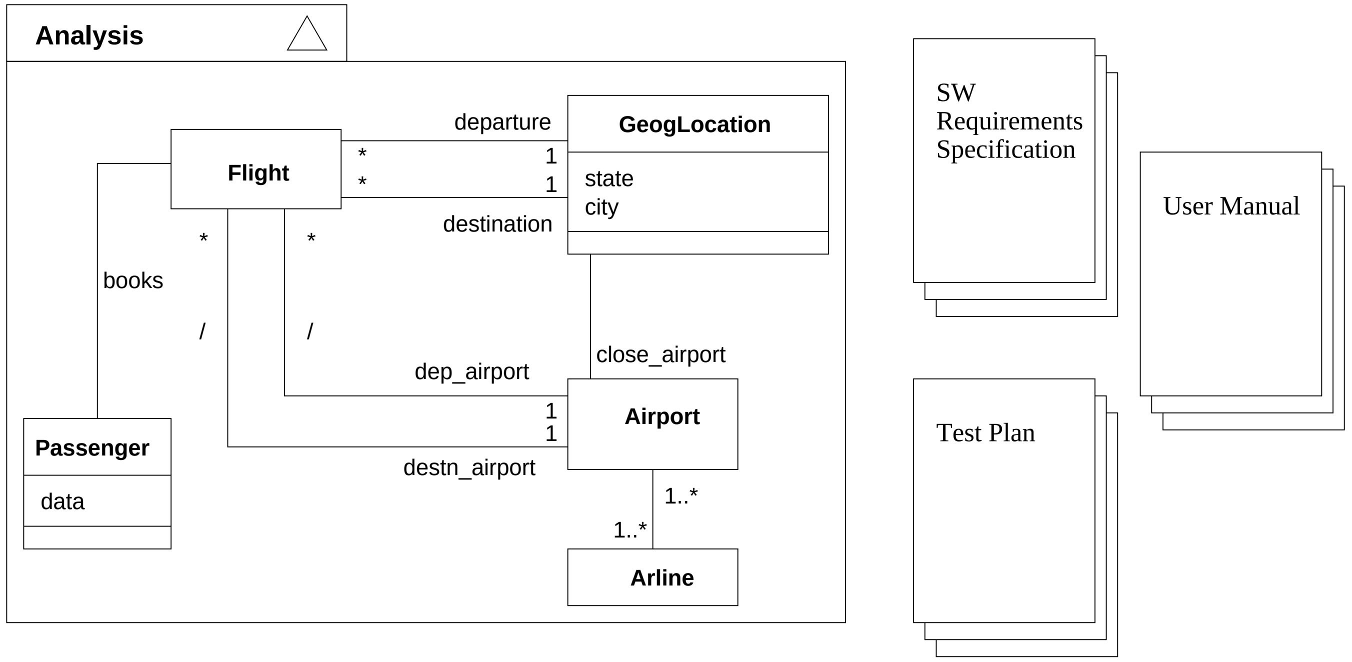
Each requirement and each design feature should be expressed in such a manner that a test can be done to determine whether that feature has been implemented correctly.

Both functional and non-functional requirements should be testable.

Test results should be traceable back to the associated requirements.

IAEA Safety Guide *Software for Computer Based Systems Important to Safety in Nuclear Power Plants*, NS-G-1.1

Models and deliverables



The analysis phase produces an *analysis* model.

The analysis model is defined by the *Software Requirements Specification* document.

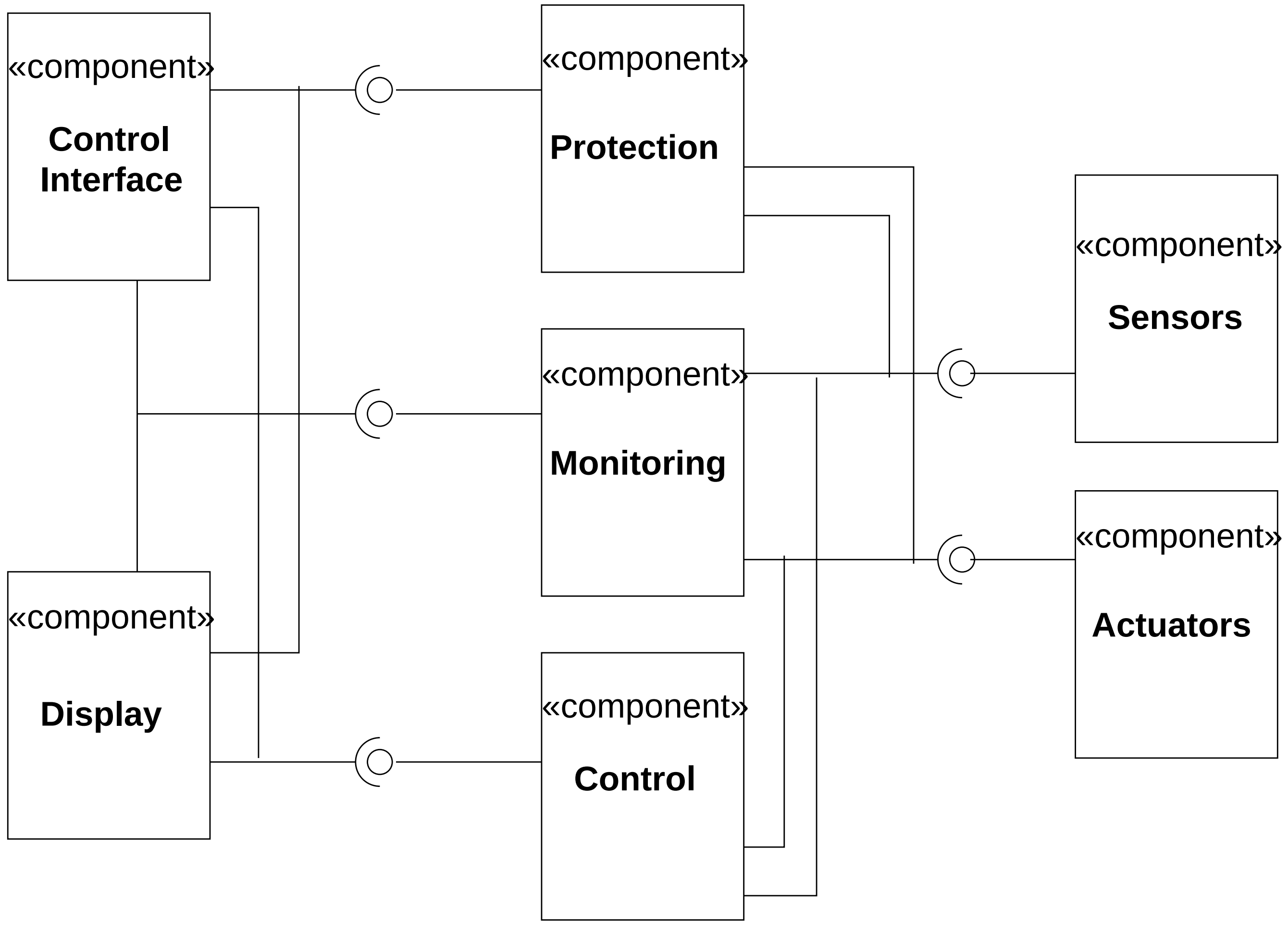
Other deliverables include the *Test Plan* and *User Manual(s)*.

DESIGN AND REALIZATION

Architectures and modules

- In the design phase developers find technical solutions to build a system that satisfies the requirements.
- The design phase produces a software *architecture*.
- An architecture is a model of the internal structure of the software.
- A system architecture consists in the set of the system’s components and their reciprocal connections and relationships.
- The architecture is hierarchical: the major components (*subsystems*) are composed of smaller components (*modules*) and so on, recursively.
- The smallest-scale modules are called *unit* modules.

An architectural model



A *hypothetical* architecture, using the UML notation for subsystems.

WARNING: this is *not* a real system!

Coding

- In the *unit coding and testing* phase the modules specified in the design phase are implemented in some programming language.
- Programming involves translating the structuring concepts used in the design, such as interfaces and relationships of various kinds, into the concepts made available by the chosen programming language.
- Programming also involves filling in several details left unspecified in the design phase.

Development tools

- *Basic* tools support programming and SW building: *text editors*, *compilers*, *debuggers*, *linkers*, tools for automatic compilation and configuration. . .
- *Integrated development environments* co-ordinate basic tools through a user-friendly interface, and may offer more advanced capabilities.
- *Computer assisted software engineering* (CASE) tools enable designers to create analysis and design models in some formal or semiformal language (we'll see them shortly).
- Tools may also support various forms of verification, and documentation.

Automatic code generation

- CASE tools can generate code for a usually partial implementation of the system.
- *If* the model is correct *and* the code generation mechanism is correct, then we are reasonably sure that the code is correct.

Integration

- *Software integration* is the process of assemblying the whole software system from its components.
- This process is carried out incrementally:
  - a first module is linked to a *test harness* that simulates the rest of the system,
  - and this simulated system (harness plus the integrated module) is tested,
  - then another module is integrated and tested, and so on.
- The process of testing each module as it is added to the system under integration is called *integration testing*.
- *Hardware and software integration* is the installation of the software on the computer(s) where it must run (that may be different from the development computer).
- The *Integrated computer system tests* are then performed.

V&V ACTIVITIES

Verification and validation (1)

A common definition:

- *Verification* checks an implementation against its specification.
  - *Are we making the product right?*
- *Validation* checks a product (final or intermediate) against its intended requirements.
  - *Are we making the right product?*

With these definitions, deliverables at any stage of the process may be validated.

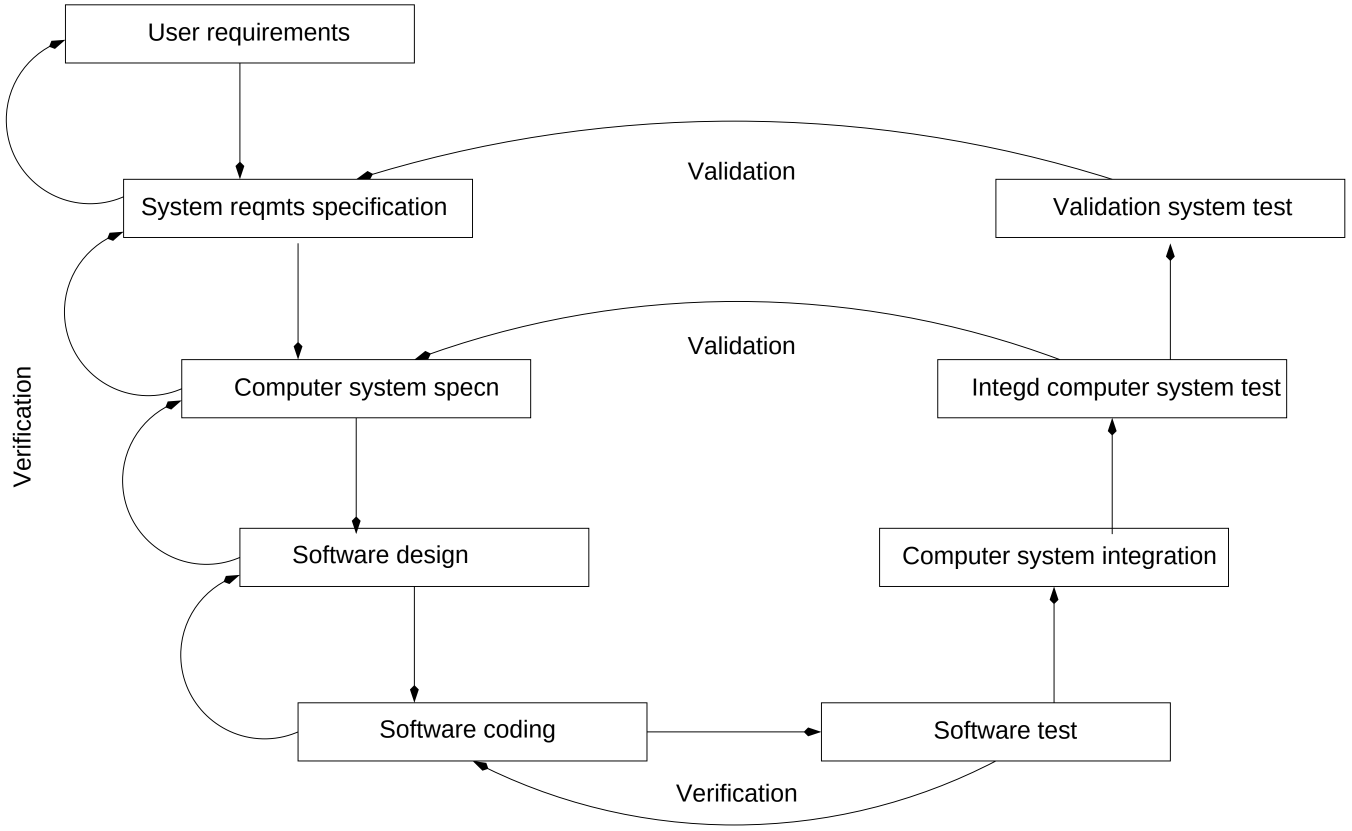
Verification and validation (2)

Another definition, as in IAEA TRS 384:

- *Verification*: The process of determining whether or not the product of each phase of the digital computer system process fulfils all the requirements imposed by the previous phase.
- *Validation*: The testing and evaluation of the integrated computer system (hardware and software) to ensure compliance with the functional, performance and interface requirements.

With these definitions, only the final product is validated. Verification applies to the intermediate deliverables.

Verification and validation (3)



The V-model (as in IAEA TRS 384).

(Picture from Wikipedia)

Wovon man nicht sprechen kann, darüber muß man schweigen.

L. Wittgenstein, *Tractatus*,

Satz No. 7

Apie ką negalima kalbėti,
apie tai reikia patylėti

Modeling and programming languages

- *Modeling* languages enable developers to define analysis and design models.
- *Programming* languages are the material out of which software is built.
- Programming languages can be seen as modeling languages with a very fine-grained level of detail.

Modeling languages

- *Informal*: natural language, or loosely defined graphical notations.
- *Formal*: textual or graphical languages with well-defined syntax and semantics.
- *Semiformal*: graphical languages with well-defined syntax but loosely defined semantics.

Formal languages (1)

- A formal language identifies some basic attributes that are simple and general enough to describe a large class of systems in an abstract way.
    - E.g., the behavior of many systems can be described in terms of sets of *states* and sequences of *actions*.
- The possible values of these attributes form the domain of the language (just like numbers form the domain of algebra).
- The language defines operations that act on the elements of the domain, such as forming sets and sequences, and combining them in various ways.
    - E.g., we may define operations for *parallel* and *sequential* composition to describe the interaction of two processes.
- We can then describe systems with formulas whose meaning can be understood in terms of mathematical concepts, such as sets and functions.

Formal languages (2)

- *Finite State Automata* (FSA). A large class of languages and a fundamental modeling paradigm, based on *states*, *transitions*, *inputs*, and *outputs* (and many extensions). Also called *(Finite) State Machines*.
- *Petri Nets*. More abstract, based on *places*, *transitions*, and *markings*. They can model the interaction of components of a complex system (whereas FSA's model a system as a whole).
- *Predicate logics*. Based on predicate logic and set theory, very general applicability.
- *Temporal logics*. Used to specify properties related to synchronization.
- *Process algebras*. A large class of languages that describe concurrent processes by means of operators on elementary actions. Often used in conjunction with temporal logics.
- ...

Example: LOTOS (1)

Subsystem S_1 is composed of processes P and Q. Subsystem S_2 is composed of processes R and S.

P can execute actions of type a and b, Q can execute actions of type c and d.

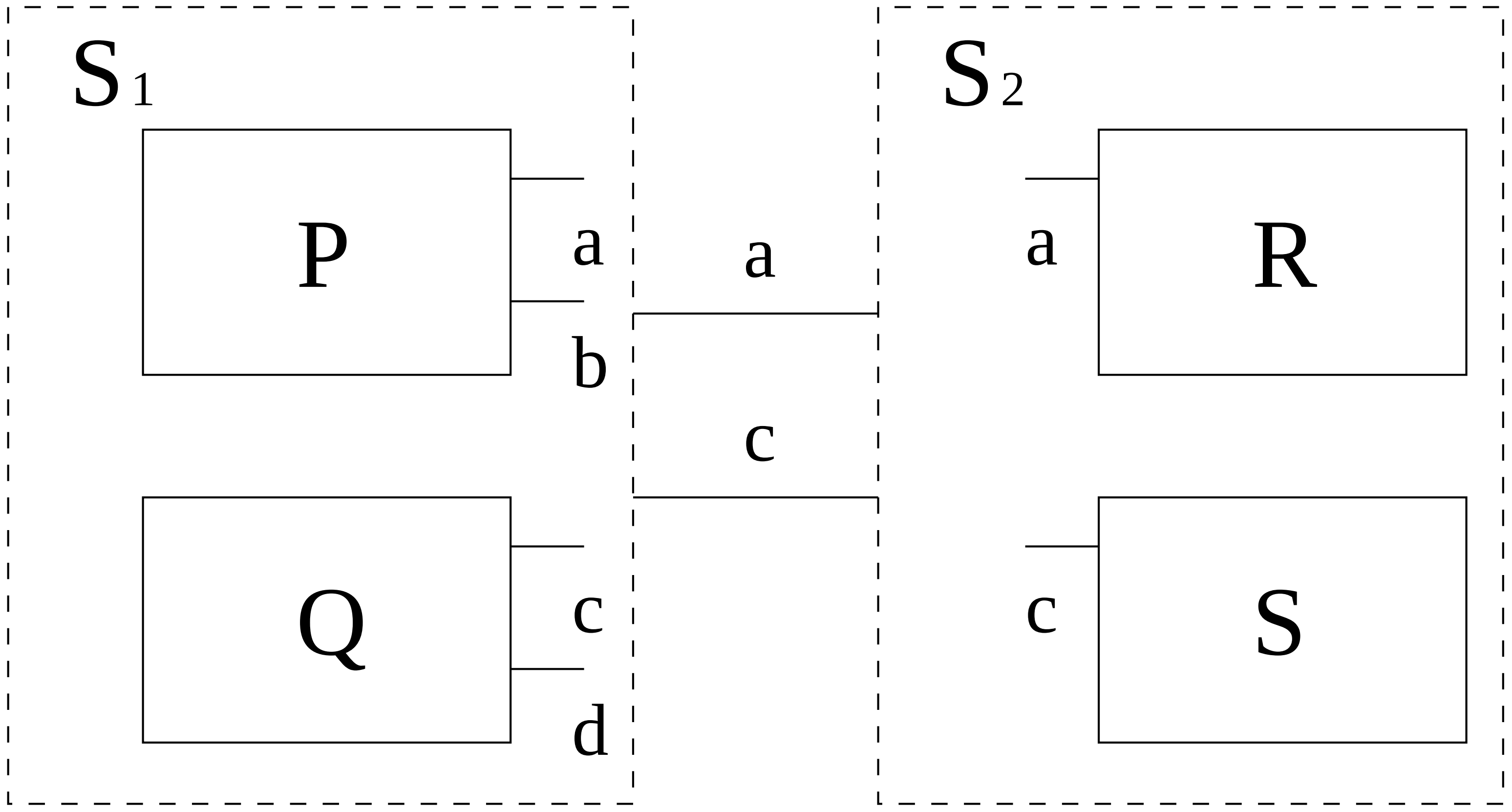
R can execute actions of type a, S can execute actions of type c.

P and Q execute their actions independently of each other: An *unconstrained* parallel composition. Similarly for R and S.

Subsystems S_1 and S_2 must execute “simultaneously” actions of type a or c: A *synchronized* parallel composition.

The next slide shows this specification as a diagram and as a LOTOS expression.

Example: LOTOS (2)



$$(P[a, b] \mid\mid\mid Q[c, d]) \mid[a, c]\mid (R[a] \mid\mid\mid S[c])$$

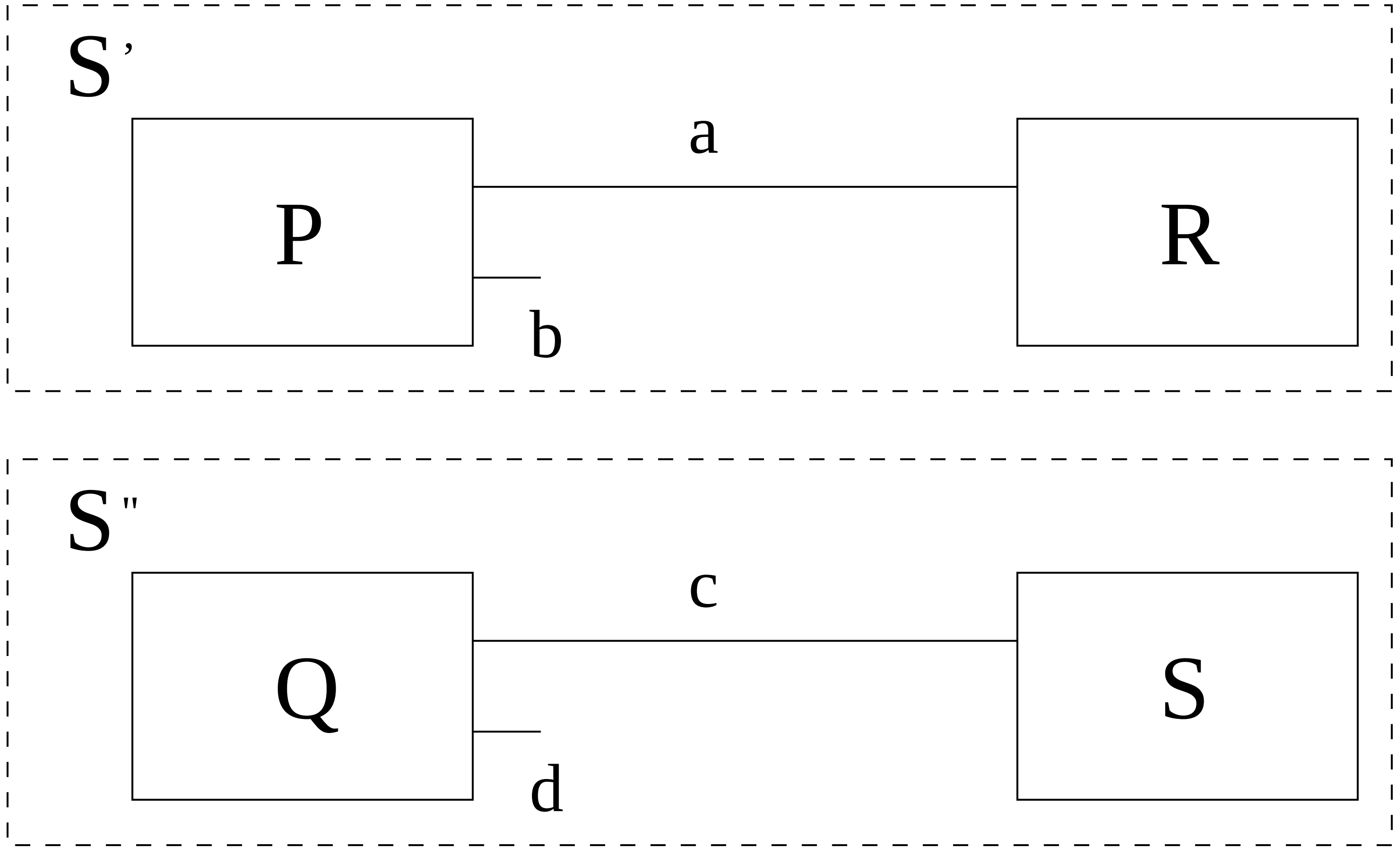
Example: LOTOS (3)

Within the theory underlying the LOTOS language, the previous expression can be transformed in an equivalent one, where the processes are rearranged.

The resulting system has the same behavior of the initial one, but a simpler structure.

The next slide shows the transformed system.

Example: LOTOS (4)



$$(P[a,b]\ |[a]|\ R[a])\ |||\ (Q[c,d]\ |[c]|\ S[c])$$

A few modeling languages

- *Z* /zεd/. Based on predicate logic and Zermelo-Fränkel set theory.
- *Vienna Development Method* (VDM). Well-known predicate logic formalism.
- *Calculus of Communicating Systems* (CCS). A process algebra.
- *Communicating Sequential Processes* (CSP). Another process algebra.
- *Language of Temporal Ordering Specification* (LOTOS). Yet another process algebra.
- *SDL* (Specification and Description Language). Based on an extension of FSA's, used for telecommunication systems, process control and real-time systems.
- *UML* (Unified Modeling Language). Probably the most popular language to date. Very large, general-purpose, composed of several sub-languages. Semiformal, with formal parts.

Programming languages

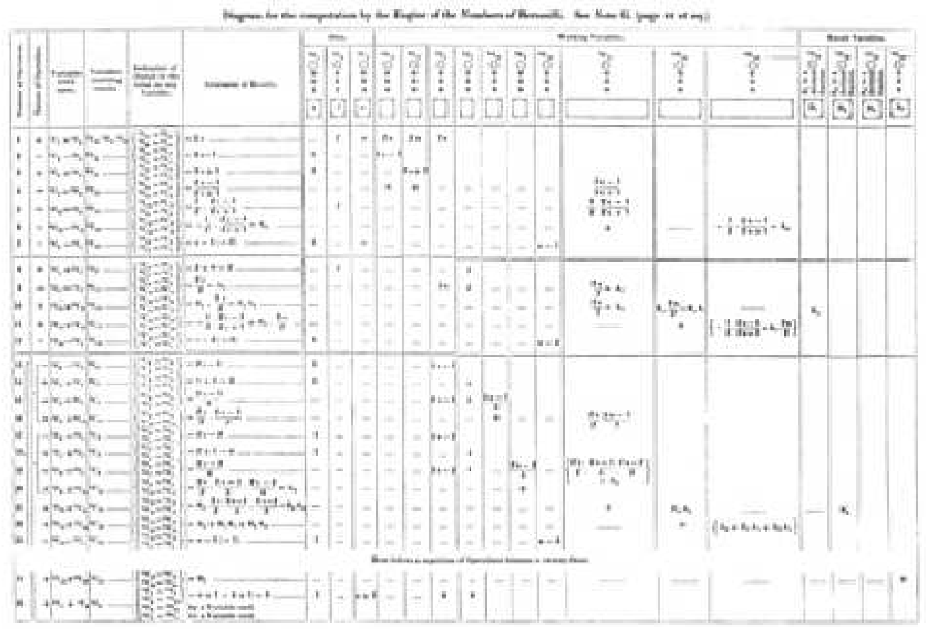
- *Imperative*: The programmer describes the *algorithm*, i.e., the sequence of steps, that the computer must execute to produce the solution.
  - *Procedure-oriented*: The elementary module is the *procedure* (or *function*, *subroutine*), a small subprogram that can be combined with other subprograms. Some data are private to each procedure, other data are shared among procedures.
  - *Object-oriented*: The elementary module is the *class*, a template that defines a set of related subprograms (*operations*, or *methods*) and the data they act upon. The instances of a class are called *objects*, and the data contained in an object are private to the object. A program is composed of objects that interact by calling each other's methods.
- *Declarative*: The programmer describes the conditions that the solution must satisfy. The language interpreter finds the solution with a built-in general-purpose problem-solving algorithm.

A few programming languages

- *Microcode*. Extremely low-level, defines the behavior of single HW components, such as CPU’s and network adapters.
- *Assembler*. Very low-level, specific to each processor type. Used to program the machine-dependent parts of the operating system.
- *C*. Wide-spectrum (from low- to high-level), procedure-oriented, used both for system and application programming.
- *Fortran*. High-level, procedure-oriented, a mainstay of scientific and engineering computation.
- *Ada*. High-level, object-oriented, conceived to support rigorous development processes, sometimes mandatory for SW with safety constraints.
- *C++*. Wide-spectrum, object-oriented, currently one of the most used languages in industrial applications.
- *Java*. High-level, object-oriented, used in a very wide range of applications, particularly web applications and graphical interfaces.
- *TTCN-3* (Testing and Test Control Notation). Used mainly in the communications area to program the test harness.

Ada

(www-history.mcs.st-andrews.ac.uk/Biographies/Lovelace.html)


[illegible]

(www.computerhistory.org)

Augusta Ada King, Countess of Lovelace, born Byron.

CONCEPTS OF SW DESIGN

Goals of SW design

Designers strive to maximize several properties of the design model:

- *Correctness*: The model satisfies the specification.
- *Comprehensibility*: Easy to understand (obvious, but hard to achieve).
- *Verifiability*: Easy to verify.
- *Modifiability*: Easy to change (“*Design for change!*”).
- *Reusability*: Some or all the components can be reused un other applications.
    - *Relevant to safety: old, proven components may be be more reliable*.
- . . .
- **Modularity**: The key all above properties.

A system is modular if each of its components has a clearly understood and defined boundary (*interface*), is responsible for a well defined task and depends on other components in a simple way.

Logical and physical architecture

- The *logical architecture* models the system as a set of abstractions (modules) that have structure and behavior, with their reciprocal relationships.
- The *physical architecture* describes the system as a set of software *artifacts* and hardware components. The software artifacts are the files (such as executables and libraries) containing the actually executed code, or the source code, or auxiliary data.

Layers of a SW system

- *Execution environment*: general-purpose operating system (e.g., Linux), real-time system, micro-kernel…
- *Libraries and frameworks*: E.g., for numerical computation, input/output, graphical user interface…
  - A library is a collection of independent modules that are assembled without modification to build a complex system.
  - A framework is a collection of inter-related and configurable modules that are specialized and assembled to build a complex system. Frameworks provide ready-made solution schemas for common problems.
- *Application-specific components*: They implement the application-specific logic.

Execution environments and libraries are often provided by third-party suppliers, i.e., they are *Commercial off-the-shelf software* (COTS), or, more generally, *pre-existing software* (PSW).

Modular design

- *Interface*: Specification of the services (including operations, data, and types) offered by a module.
- *Implementation*: The internal elements of a module, whose structure and behavior satisfy the interface specification.
- *Hiding*: Making the internal elements of a module invisible to elements outside that module.

Information hiding is the basis of modular design, as it avoids unnecessary inter-module dependencies.

Object-oriented languages allow each internal element of a module to be declared *public* (externally visible) or *private* (hidden).

TECHNIQUES FOR V&V

Static analysis in V&V (1)

The following is based on IAEA TRS 384:

- *Walk-through*: A document (such as code, design, etc.) is presented to a group including developers and people possibly not involved in development. The document is evaluated and criticized.
- *Inspection*: A document or set of documents is read by a group of people who check the document(s) for expected defects of some class (e.g., typical programming mistakes, common design flaws. . .). Checklists are commonly used.
- *Formalized descriptions*: Writing a specification (or design) document using a formal language. This is a way to validate an informal requirements statement, and a pre-requisite for formal verification.

Static analysis in V&V (2)

- *Program proving*: *Assertions* (statements about relationships among variables) are associated with the beginning (*pre-conditions*) and the end (*post-conditions*) of a program segment (such as a procedure), and if the program is correct, the post-conditions can be proved by logical arguments to be a consequence of the pre-conditions.
- *Symbolic execution*: The input variables are assigned symbolic values (say, x) instead of numeric ones. A symbolic interpreter applies the program statements to the input variables and computes the output values as symbolic (i.e., algebraic or logical) expressions that can be checked for compliancy with the program specification.
- *Automatic analysis*: Automatic tools can check a program for indicators of possible anomalies, such as unreachable statements or usage of non initialized variables [a]

[a] This technique is not mentioned in IAEA TRS 384.

Example: Program proving

```
for (i = 0; i < M; i++) {            // $0 \leq i < M$
    for (j = 0; j < M-i; j++) {      // $0 \leq j < M - i$
        if (v[j] > v[j+1]) {
            t = v[j];
            v[j] = v[j+1];
            v[j+1] = t;
        } // $v_j \leq v_{j+1}$                                   (i)
    } // $\forall k (M - i - 1 \leq k < M \Rightarrow v_k \leq v_{k+1})$     (ii)
} // $\forall k (0 \leq k < M \Rightarrow v_k \leq v_{k+1})$                 (iii)
```

This program sorts an M-element vector [a]. Formulas (i), (ii), and (iii) are conditions that must hold at the respective points in the program. Formula (iii) is the definition of an ordered vector.

[a] I.e., a sequence of values, in computer science jargon.

Formal verification

A formal model enables developers to find out about important properties of the system:

- *Safety* properties: undesired states will *not* be reached, undesired actions will *not* be executed.
  - E.g., *Will the temperature raise above a given threshold? Will the reactor trip on a false alarm?*
- *Liveness* properties: desired states *will* be reached, desired actions *will* be executed.
  - E.g., *Will the reactor reach full power? Will it trip on a real alarm?*

Dynamic analysis

- *Prototype execution*: A *prototype* is a partial or simplified version of a software system, whose internal structure is often unrelated to that of the final product. Prototypes can be used to assess the feasibility of design choices, to let users validate the specifiers' interpretation of the requirements, and to experiment with different design choices.
- *Simulation*: The environment (e.g., the plant) where the software will operate may be simulated by a simulator program.
- *Testing*: The software is exercised by a selected set of inputs (*test data*) to discover possible malfunctions.

TESTING

(www.adeptis.ru/vinci/m_part7.html)

Program testing can be used to show the *presence* of bugs, but never to show their absence.

E. Dijkstra, quoted in Dahl et al., *Structured Programming*.

Software faults and failures

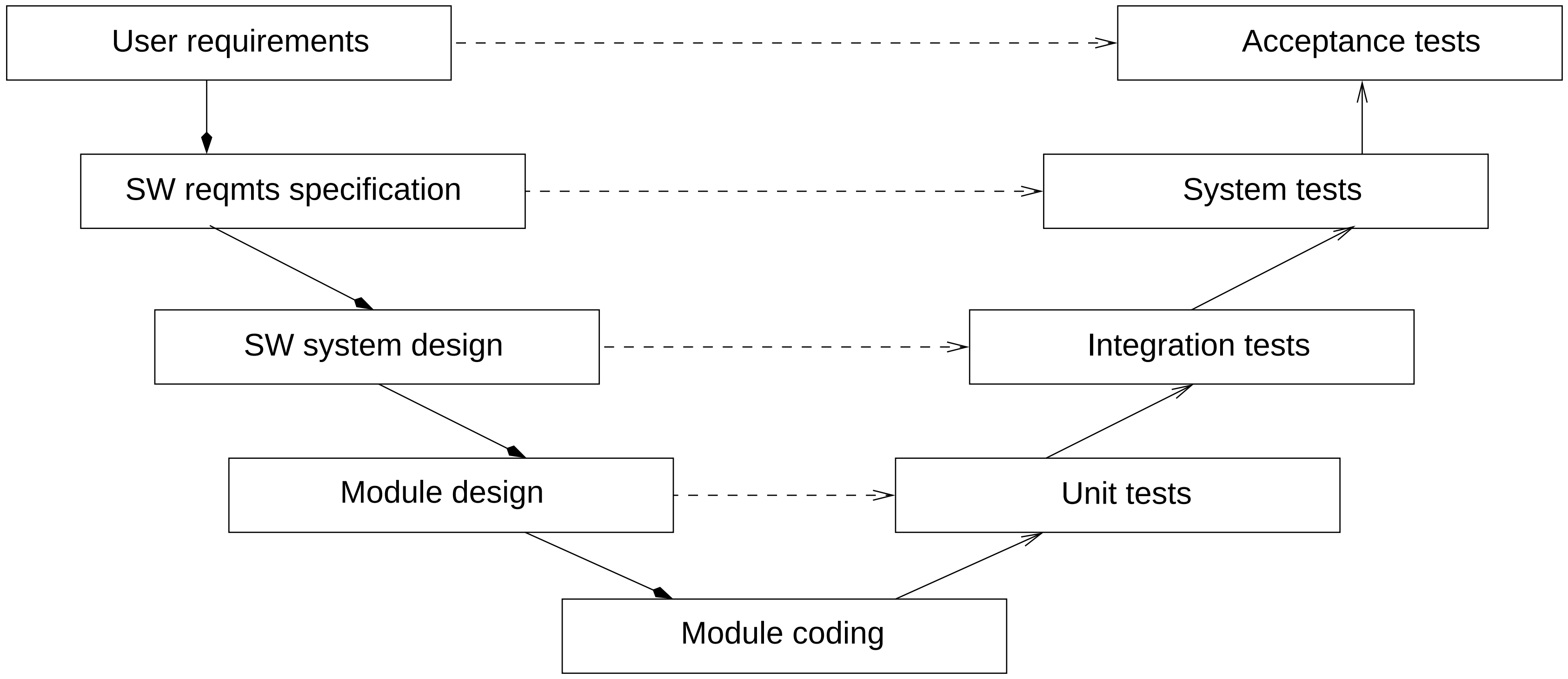
- A *failure* is an incorrect behaviour of the software.
- A *fault* (commonly known as *bug*) is a defect in the software that *may* cause an observable failure.
- Software failures are *systematic*, not random.
- Nevertheless software failure appear to occur randomly, since a given failure may occur only when a particular “unlucky” input *activates* the fault.
- Further, a given fault may cause different failures, a given failure may be produced by different faults, a fault may hide another one. . .

Oracles

An *oracle* is someone or something that tells us if a test result is correct or not. How does the oracle know?

- Well known data (e.g., standard numerical tables).
- Hand computation.
- Comparison with results of previous or alternate versions.
- *Executable specifications*: A prototype is written in a formal executable language (e.g., LOTOS, Prolog. . .), and used as a reference.

Testing in the development process (1)

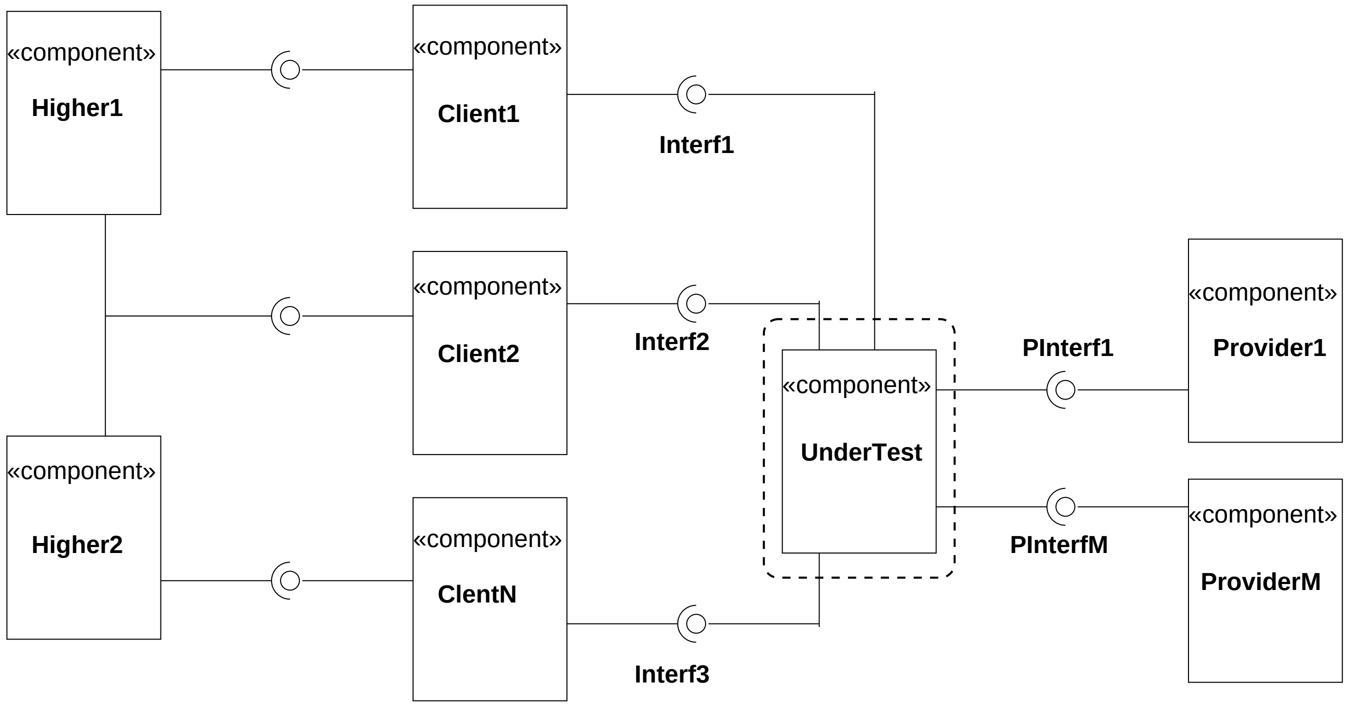


A more specific V-model, adapted from H. Waeselynck, *Introduction to Software Testing*, in F. von Henke et al., *Testing, Verification, and Validation*, ReSIST NoE courseware, http://resist.isti.cnr.it/files/corsi/courseware_slides/testing.pdf

Testing in the development process (2)

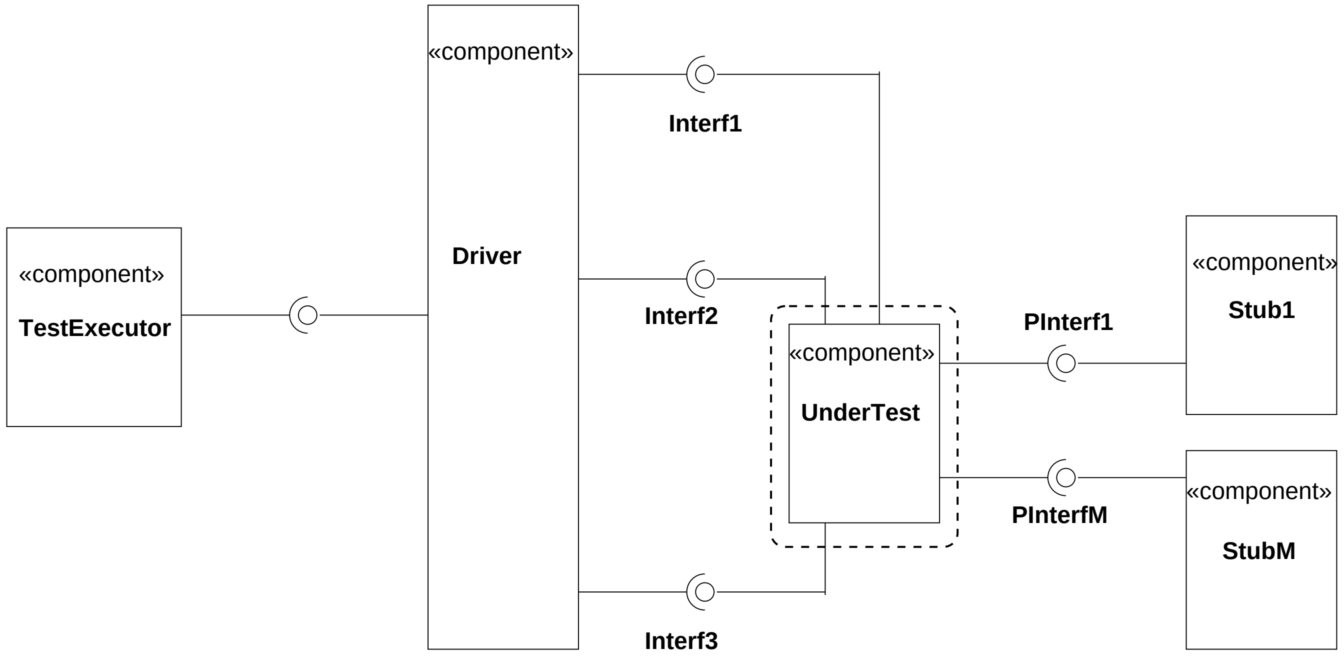
- *Unit testing*: Each unit module is tested in the coding phase.
- *Integration testing*: In the SW integration phase, the correct interfacing of each unit module with the rest of the system is tested.
- *Regression testing*: After each change in the software, tests are made to ensure that at least the previous functionalities are preserved. Regression testing is then applied during coding, integration, and maintenance.
- *System testing*: The final system is tested.

SW integration testing (1)



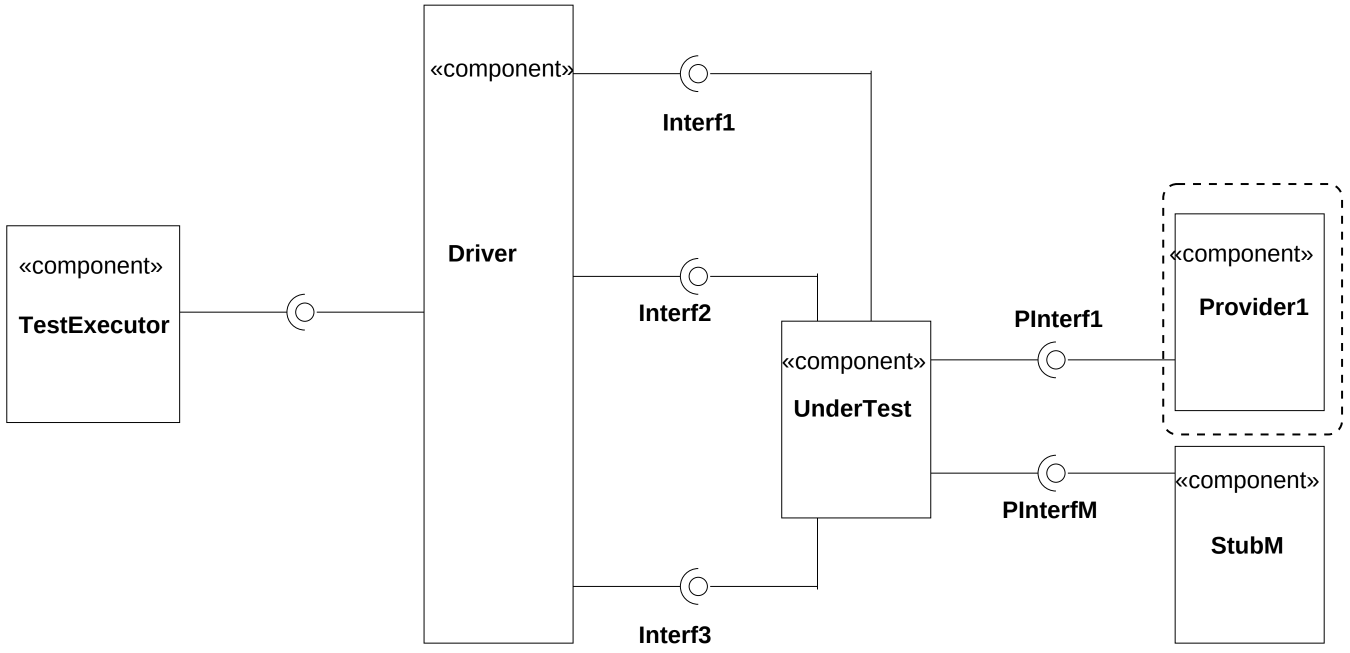
A module within the architecture.

SW integration testing (2)



A module under test within the test harness.

SW integration testing (3)



Another module is integrated.

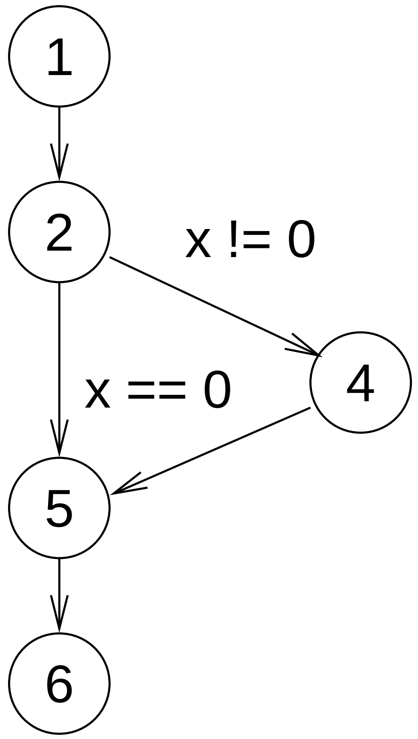
Data selection criteria

Testing techniques may be grouped by their criteria for data selection:

- *Structural* (white-box): the choice of test data is driven by our knowledge of the internal structure of the software, i.e., we try to *exercise* (probe) the various parts of the software.
- *Functional* (black-box): the choice of test data is driven by our knowledge of the requirements.
  - *Statistical testing*: test data are selected according to a statistical distribution of the input data.

Structural testing

```
1  read(x);
2  read(y);
3  if (x != 0)
4      x = x + 10;
5  y = y / x;
6  write(x, y);
```



Any single pair (x_1, y_1) with $x_1 \neq 0$ *covers* all statements, but two pairs $\{(x_1, y_1), (0, y_1)\}$, with $x_1 \neq 0$, cover all branches, and find the fault in stmt 5.

Different *coverage criteria* spot different types of faults.

Data flow criteria

A form of structural criteria, where data are selected in order to cover all paths containing significant operations on variables:

- *Variable definition*.
- *Variable usage in a computation*.
- *Variable usage in the evaluation of a logical condition*.

Functional testing

- *Equivalence classes and boundary values*: The set of possible inputs (including invalid ones) is divided into subsets, such that the values within each class produce equivalent outputs (under some criterion). Test data are then chosen “inside” each class and at the class boundaries.
- *Decision tables*: First, we identify logical conditions that different outputs must satisfy (e.g., a variable may be positive or negative, a signal may be ON or OFF. . .), then we find combinations of conditions on inputs that make output conditions true or false. These relationships are summarized in a decision table (or graph). Data are selected to cover all columns of the table.
- *Formal models*: If there is a formal model, test data may be selected by criteria based on the model. For example, if the system is modeled as a state machine, possible criteria are coverage of all states, all transitions, all paths. . .

Statistical testing

The statistical distribution from which test data are obtained can be derived from experimental data and/or assumptions on:

- The application's input space.
- The fault distribution in the object to be tested.
- The frequency with which the different parts of the input space will be exercised.

(Adapted from IAEA TRS 384).

SAFETY-RELATED STANDARDS

- IAEA Safety Guide *Software for Computer Based Systems Important to Safety in Nuclear Power Plants*, Safety Guide NS-G-1.1, Vienna, 2000.
- *Licensing of safety critical software for nuclear reactors – Common positions of seven European nuclear regulators and authorised technical support organizations*, Revision 2007, © AVN (Belgium), BfS (Germany), CNS (Spain), ISTec (Germany), NII (United Kingdom), SKI (Sweden), STUK (Finland).
- IEC, *Software for Computers in the Safety Systems of Nuclear Power Stations*, Standard 880, IEC, Geneva, 1986.
- IEEE, *Standard Criteria for Digital Computers in Safety Systems of Nuclear Power Generating Stations*, IEEE Std 7-4.3.2-2003.
- IEC, *Functional safety of electrical/electronic/programmable electronic safety-related systems*, IEC/TR 61508-0, 2005.