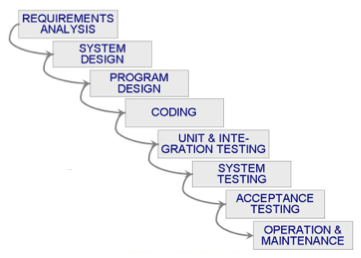
1. Define the stages of a software lifecycle. Describe and compare the waterfall model, the V model, the spiral model and the agile model of software development. Discuss the addition of prototyping.

**Stages of a software lifecycle**

* Requirements analysis and definition
* System design
* Program design
* Writing the program (program implementation)
* Unit testing
* Integration testing
* System testing
* System delivery
* Maintenance

**Waterfall model**



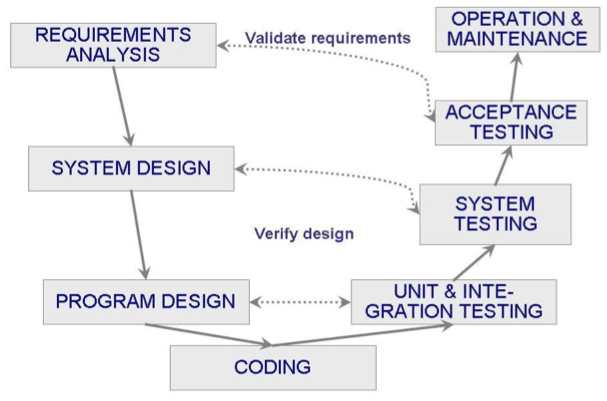
This is one of the first models to be proposed. The stages are depicted as cascading from one to another (i.e. one development stage should be completed before the next begins).

It works for well-understood problems with stable requirements.

Waterfall model is simple, easy to understand and explain (very high-level view).

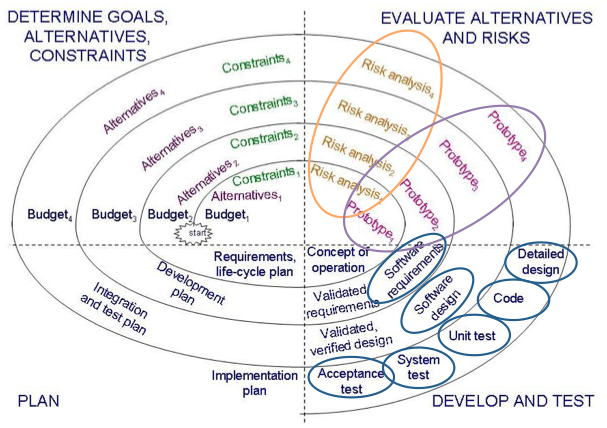
On the other hand, it cannot handle changes (frozen requirements, manufacturing process rather than creative, no iterative activities) and there is a long wait before a final product.

**V model**



This is a variation of the waterfall model that demonstrates how the testing activities are related to analysis and design (development activities). If problems are found, we loop back to the left side of the “V”. In other words, the V model makes more explicit some iterations and reworks that are hidden in the waterfall model.

**Spiral model**



This model combines development activities with risk management to minimize and control risk. It is a sort of iterative development (full system at the very beginning and then changes functionality of each subsystem with each new release)

Four basic activities that must occur in each cycle of the spiral model:

1. Consider the win conditions of all success-critical stakeholders.
2. Identify and evaluate alternative approaches for satisfying the win conditions.
3. Identify and resolve risks that stem from the selected approach(es) (e.g. you can try resolve it, by making a prototype which valid the approach).
4. Obtain approval from all success-critical stakeholders, plus commitment to pursue the next cycle.

*(Wikipedia)*

**Agile model**

The classical software development process models are rigorous process for software conception, documentation, development and testing.

The alternative is the agile method that is a flexible process that can adapt to changing requirements. The overall goal of agile development is to satisfy the customer by “early and continuous delivery of valuable software”.

Agile Manifesto:

* Value individuals and interactions over processes and tools
* Prefer to invest time in producing working software rather than in producing comprehensive documentation
* Focus on customer collaboration rather than contract negotiation
* Concentrate on responding to change rather than on creating a plan and then following it

Examples of agile processes:

* Extreme programming (XP)
* Crystal
* Scrum

**Addition of prototyping**

Prototyping means building a small version of a system (usually with limited functionality) that can be used to:

* Demonstrate the feasibility of a design or approach
* Help the user (or customer) to identify key requirements of a system

Often, the prototyping is iterative: We build a prototype, evaluate it (with user and customer feedback), consider how changes might improve the product or design, and then build another prototype. The iteration ends when our customers and we think we have a satisfactory solution to the problem at hand.

Imagine, in detail, a new product: hard

Critique, in detail, an existing product: easier → prototyping.

There are two approaches to prototyping (called rapid prototyping):

* Throwaway: software developed to learn more about the problem/proposed solution. “Quick and dirty” software that will be thrown away (not part of the delivered software).
* Evolutionary: software developed not only to help us answer questions but also to be incorporated into the final product.

1. Explain the goals and principles of postmortem analysis. Describe the Capability Maturity Model and its five levels. Compare to SPICE and ISO 9000.

**Postmortem analysis**

A postmortem analysis is a post-implementation assessment of all aspects of the project (products, processes and resources). It is intended to determine whether goals were met or not, and to identify areas of improvement for future projects.

The analysis usually takes place shortly after a project is completed (in practice, it can take place from just before delivery to 12 months afterward).

Postmortem analysis process:

* Project survey: Ask no more than needed, without compromising confidentiality.
* Objective information: Collect relevant data: cost (effort, LOC), schedule, quality.
* Debriefing meeting: Allow team members to report problems.
* Project history day: Identify the root causes of the key problems, review schedule predictability charts.
* Publishing the results: Focus on lessons learned.

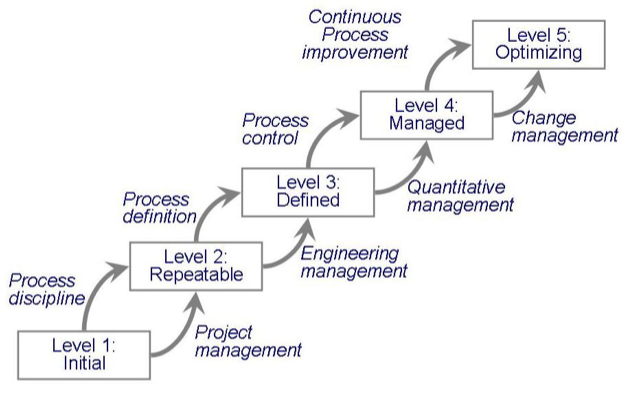
**Capability Maturity Model** (developed by the Software Engineering Institute)

It has 5 levels of maturity, each with a set of key process areas.

The assessment is based on:

* A questionnaire
* Requests for evidence to verify the answers

Note that these are not 5 discrete rankings, but a continuous scale (>< process maturity model).



*Level 1: Initial*

* Development process is ad hoc or even chaotic
* Cannot describe the process
* No key process areas

*Level 2: Repeatable*

* Identified inputs and outputs, constraints (budget, schedule), resources
* Project management in place
* Measurements on project
* Key process areas: management activities

*Level 3: Defined*

* Management and engineering activities are documented, standardized and integrated
* Measurements on products
* Key process areas: organization

*Level 4: Managed*

* Quality is measured, tracked and managed
* Early projects feedback to later projects
* Key process areas: quantitative and quality management

*Level 5: Optimizing*

* Quantitative feedback is incorporated to produce continuous process improvement
* Key process areas: change management

**SPICE** (ISO Standard 15504)

This is an international standard for process assessment. SPICE stands for Software Process Improvement and Capability dEtermination.

It harmonizes and extends the existing process assessment methods (CMM and its descendants).

Six levels of capability for each **process area**:

0: Not performed

1: Performed informally

2: Planned and tracked

3: Well-defined

4: Quantitatively controlled

5: Continuously improved

Whereas the CMM addresses organization, SPICE addresses processes.

**ISO 9000**

ISO 9000 is a series of standards that specify actions to be taken when any system (i.e. not necessarily a software system) has quality goals and constraints. In particular, ISO 9000 applies when a buyer requires a supplier to demonstrate a given level of expertise in designing and building a product.

Among the ISO 9000 standards, ISO 9001 is the most applicable to the way we develop and maintain software. It explains what a buyer must do to ensure that the supplier conforms to design, development, production, installation and maintenance requirements.

Since ISO 9001 is quite general, there is ISO 9000-3 that provides guidelines for interpreting ISO 9001 in a software context.

The ISO 9000 standards are used to regulate internal quality and to ensure the quality of suppliers.

*Summary:*

ISO 9000: quality management systems and principles

ISO 9001: requirements that organizations must fulfil

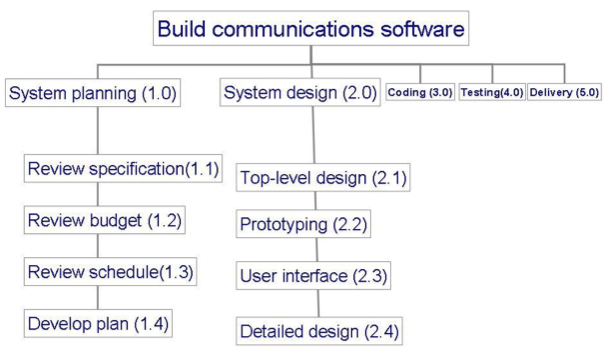
ISO 9000-3: guidelines for applying ISO 9001 to software

1. Describe elements of project schedules: work breakdown structure, tasks, deliverables and milestones. Explain the critical path method. Illustrate on PERT and Gantt charts. Discuss resource planning and tracking.

**Elements of project schedules**

*Work breakdown structure:*

A hierarchical decomposition of a project into tasks.



*Deliverable:*

An item to be delivered to the customer (software, documents, technical demonstrations).

*Task (activity):*

A part of the project that takes place over a period of time.

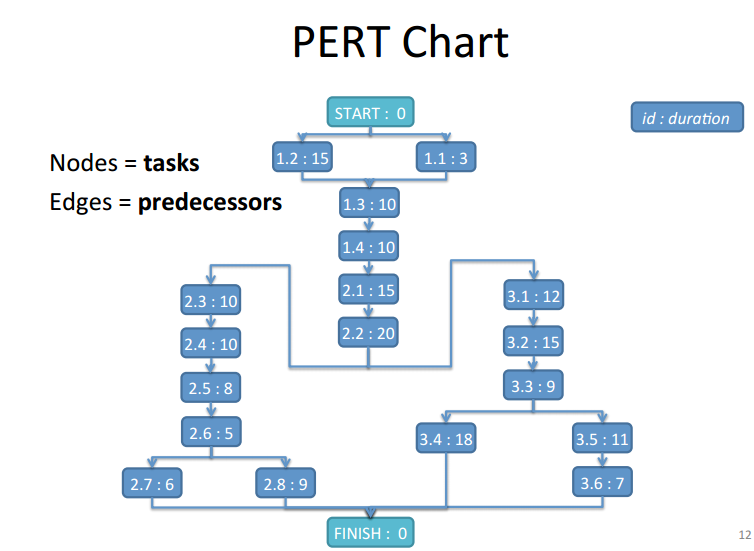
* Predecessors (precursors): events that must occur in order for a task to start
* Duration: length of time needed to complete a task
* Due date: date by which a task must be completed
* Endpoint: event marking the end of the task

*Milestone:*

A particular point in the progress of a project. Typically, the completion of a task.

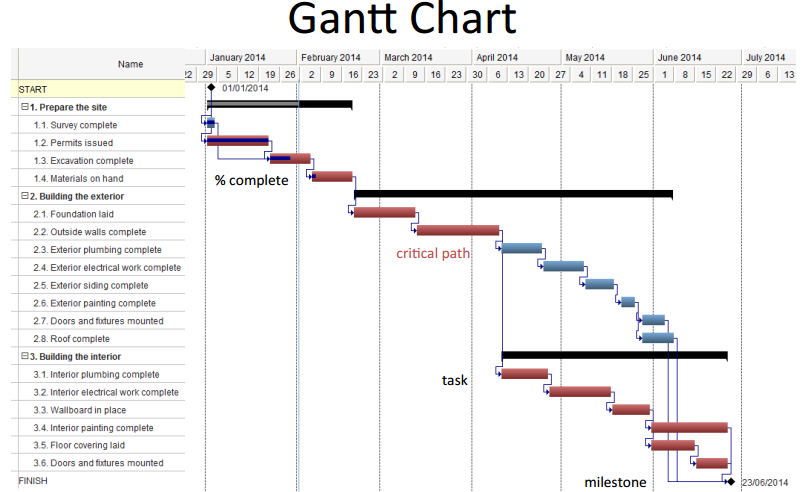
The delivery of a deliverable is a milestone.

**PERT chart**



Way to represent the arrangement of the different tasks of the project and the relations among them.

**Gantt chart**

****

Another way to represent the arrangement of the tasks. The X-axis represents the planning (time duration) of the tasks.

**Critical Path Method (CPM)**

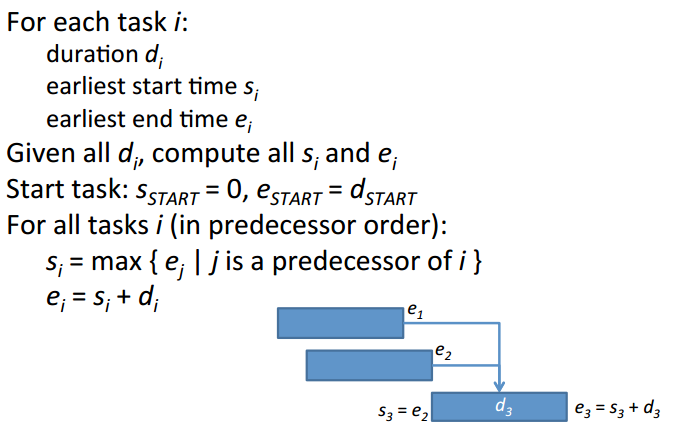
The CPM is an algorithm for scheduling a set of project activities.

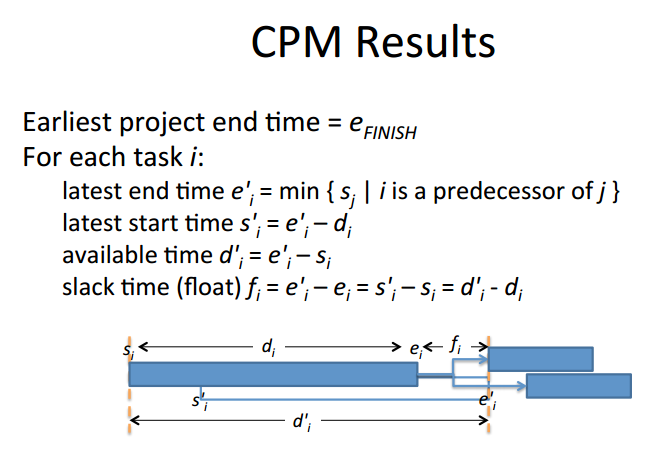
The critical path shows us the minimum amount of time it will take to complete the project, given our estimates of each activity’s duration and predecessors. It also reveals those activities that are most critical to completing the project on time.

The critical path is the path along which the activities have no slack time.

Gantt and PERT chart are appropriate to apply the algorithm because they contain all the required information. Although, in my opinion, PERT is a best representation to apply the algorithm and Gantt is better to show the plan resulting from the algorithm.

*Algorithm:*

****

****

**Resource planning and tracking**

Consist to estimate the resource usage for each task:

Staff [pers]

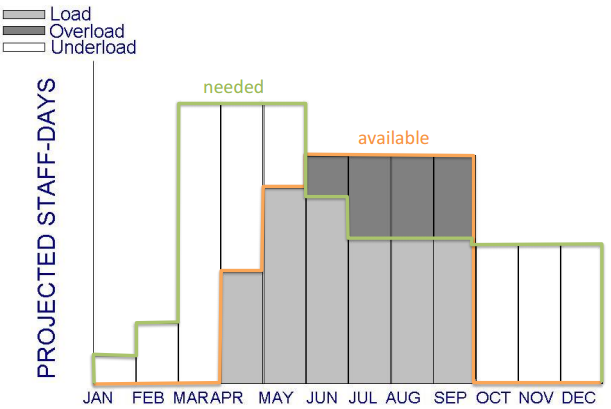
Supplies [units/day]

Expenditure [$/day]

Sum the resources required at each time.

Compare to the available resources.

Track actual vs. planned resource usage.



1. Discuss cost estimation, when it should be done and what it should estimate. Describe and compare expert judgment, cost matrix (Wolverton), algorithmic (COCOMO) and machine learning methods. Detail the three stages of the COCOMO II method. Define risk, risk impact and risk exposure. Discuss strategies for risk reduction.

**Cost estimation**

*When?*

Estimating project costs has to be done as early as possible (but imprecise at the start) and it should be done repeatedly.

*What?*

* Facilities: hardware, space, furniture, telephone, …
* Staff (effort): the biggest component and the greatest degree of uncertainty → depends on work style, organization, skills, training, …
* Methods and tools

**Methods comparison**

|  |  |  |  |
| --- | --- | --- | --- |
| *Expert judgment* | *Cost matrix (Wolverton)* | *Algorithmic (COCOMO)* | *Machine learning* |
| Based on analogies with past projects.  Beta-probability distribution:  → Use (x + 4y + z) / 6  x: optimist estimates  z: pessimist estimates  y: most likely estimates  Delphi technique:  → Use average of “secret” estimates  /!\ Subjective, simplistic! | $/LOC as a function of type of software, novelty, difficulty | E = b Sc  b, c: calibrated on past projects  S: estimated size of the system (LOC)  E: estimation in man-months | Case-based reasoning (use data from past projects)  Four steps:  1. User identifies problem as a case  2. System retrieves similar case from a repository  3. System reuse knowledge from previous case  4. System suggests a solution for the new case  But hard to define the problem as a case and determine similarity. |

No model works for all types of development.

**Three stages of the COCOMO II method**

*Application points:*

Object: report, program component etc…

Object points (numerical value) depend of complexity.

Application points: Sum of Object points

*Stages:*

1. Application Composition:

Prototyping stage

S in application points

**E = b . S = NOP / PROD**

1 / b = productivity (ex: developer experience and skills)

**S = NOP = application points**

In stage 2 and 3:

**E = b . Sc . m(x1, ..., xn)**

c between 0.91 and 1.23 depend of novelty team cohesion etc..

1. Early Design:

Architectural design stage

S in function points

UFP (unadjusted function points) Sum of application components.

**S = FP = UFP X TCF (technical complexity factor)**

1. Postarchitecture:

Development stage

**S in lines of code**

**Risk definitions**

*Risk:*

An unwanted event that has negative consequences.

*Risk impact:*

The loss associated with the event. [$, man-month]

*Risk probability:*

The likelihood that the risk will occur [0…1]

*Risk exposure:*

(risk probability) x (risk impact) [$, man-month]

**Strategies for risk reduction**

*Risk avoidance:*

Change requirements, add resources, training, …

*Risk transfer:*

Transfer to another system, partnering, buy insurance

*Risk acceptance:*

Accept and control the risk, contingency plans

To aid decision making about risk reduction, we must take into account the cost of reducing the risk. We call risk leverage the difference in risk exposure divided by the cost of reducing the risk. In other words, risk reduction leverage is:

(E – E') / C

with

E, E' = risk exposure before and after reduction

C = cost of risk reduction

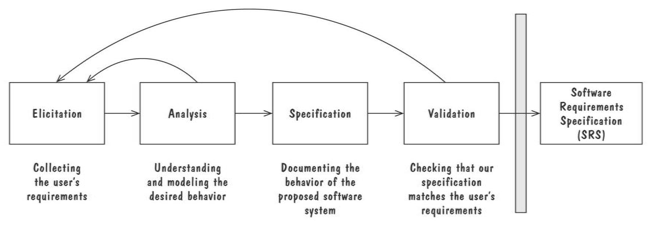
1. Define software requirements. Discuss means and stakeholders of requirements elicitation. Describe the different types of requirements and their desired characteristics. Explain and distinguish the nature of requirements definition and specification documents.

**Software requirements**

A requirement is an expression of desired behaviour. A requirement deals with objects or entities, the states they can be in, and the functions that are performed to change states or object characteristics. We are looking for requirements that identify key entities, limit entities or define relationships among entities.

Focus on the customer needs, NOT on the solution or implementa1on → what behaviour is needed, NOT how that behaviour will be realized.

*The requirements process:*



In agile method, requirements defined incrementally, essential requirements first, additional requirements in subsequent releases.

**Requirements elicitation**

Collecting what the customers (and users) want from the system to be developed.

Need to discuss the requirements with everyone who has a stake in the system.

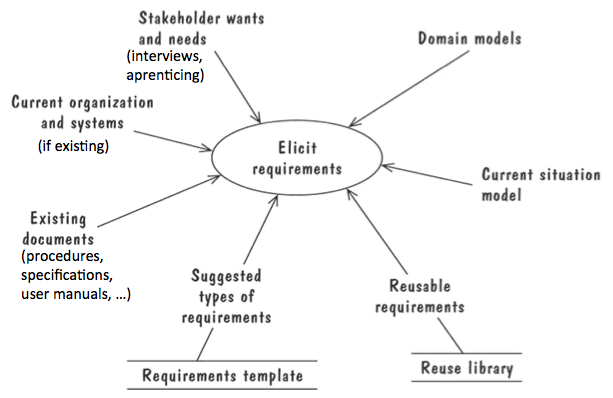
Document and review the requirements.

Reach an agreement!

*Stakeholders:*

* Clients: paying for the software to be developed.
* Customers: buying the software after it is developed.
* Users: familiar with the current system and will use the future system.
* Domain experts: familiar with the problem that the software must automate.
* Market researchers: have conducted surveys to determine future trends and potential customer’s needs.
* Lawyers or auditors: familiar with government, safety or legal requirements.
* Software engineers or other technology experts: ensure that the product is technically and economically feasible.

*Means:*



… je ne comprends vraiment pas bien ce que signifie cette image, le paragraphe qui suit n’est pas nécessaire si ca n’a aucun rapport…

You have to discuss with the stakeholder in group, to be inspired by other’s ideas. Different stakeholder have different views, they can lead to inconsistencies. First, prioritize the requirements and try to reach a compromise. Second, use the view points, in other words don’t resolve inconsistencies early, keep just separate viewpoints and solve it later when there is sufficient information.

**Types of requirements and their desired characteristics**

*Functional requirement:*

It describes a required behaviour in terms of required activities, such as reactions to inputs, and the state of each entity before and after an activity occurs.

*Quality (or nonfunctional) requirement:*

It describes some required quality characteristic such as fast response time, ease of use, high reliability, low maintenance costs, …

*Design constraint:*

It is an imposed design decision such as choice of platform or interface components.

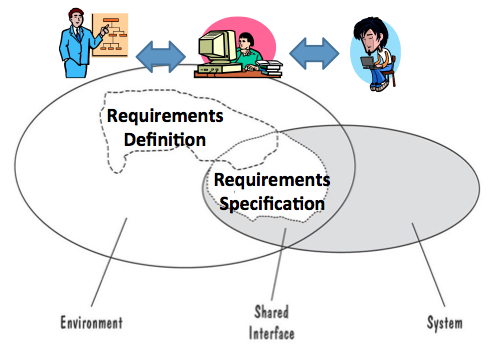
*Process constraint:*

It is an imposed restriction on the techniques or resources to be used such as agile methods, prototypes, …

*Desired qualities of the requirements:*

* Correct: the requirements conform to the desired understanding.
* Consistent: all requirements can be satisfied simultaneously.
* Unambiguous: requirements have a unique valid interpretation.
* Complete: requirements specify the behaviour under all possible inputs, states and contexts (externally) and define all terms (internally).
* Feasible: it is possible to realize a system that meets all the requirements.
* Relevant: all requirements correspond to desired or necessary functions or characteristics.
* Testable: it is possible to demonstrate whether every requirement is met.
* Traceable: all requirements are labelled and organized for easy reference.

**Nature of requirements definition and specification documents**



Two kinds of requirements documents

*Software Requirements Definition (SRD):*

Everything the customer wants to achieve, in terms of the environment of the system (i.e. expressed in the customer’s terms), aimed at a business audience, written by the client and the requirements analyst/

*Software Requirements Specification (SRS):*

A specification of how the proposed system shall behave, in terms of directly accessible environment (i.e. the boundary of the system), aimed at a technical audience, written by the requirements analyst for the developers.

Restate the SRD in terms of the system’s interface, with sufficient details, without forcing a particular design.

1. Cite and briefly describe, with small examples, the types of notation that can be used to describe the structure, the behaviour or the functions of a system. Define and distinguish the notions of verification and validation, and the techniques that can apply to them. Explain how they apply to requirements.

**Types of notation to describe the structure, the behaviour or the functions of a system**

*Structure:*

Entity-Relationship diagram (ER diagram)

Class diagram

*Behaviour:*

Event traces

Sequence diagram

State diagram

Petri nets

Data-flow diagram

Use-case diagram

*Functions:*

Decision table

Parnas table

Object constraint language (OCL)

Algebraic specifications

**Notions of verification and validation**

*Verification:*

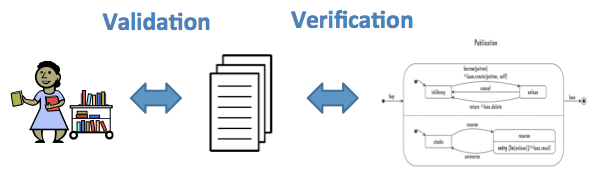
Specifications conform to requirements (i.e. "build the right system")

→ documents ÷ documents (easier)

*Validation:*

Requirements accurately reflects the customer's needs (i.e. "build the system right")

→ needs ÷ documents (harder)



*Techniques that apply to them:*

|  |  |
| --- | --- |
| Verification | Validation |
| Cross-referencing  Simulation  Consistency checks  Completeness checks  Reachability checks (states, transitions)  Model checking  Mathematical proofs | Walkthrough  Readings  Interviews  Reviews  Checklists  Formal inspections  Modeling  Scenarios  Prototypes  Simulation |

*How they apply to requirements:*

**Verification:**

Check that specifications are conforming to definitions and to other specifications (consistency).

Check traceability or (better) demonstrate.

Specification **and** assumptions **imply** requirements

Eventually computer helped (model checking or theorem proving).

**Validation:**

Review of the requirements by representatives of the customer and the developer.

Meeting to discuss the result.

Steps:

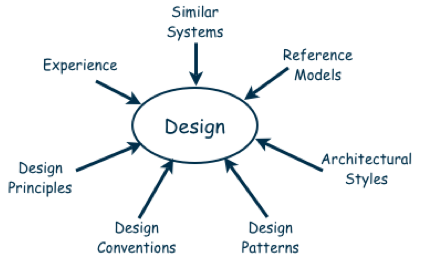
* Review the goals and objectives of the system
* Compare requirement with goal and objectives
* Review environment in which the system operates
* Etc…

1. Define software architecture, software units and architectural views. Describe the following architectural styles: pipes-and-filter, client-server, peer-to-peer, publish-subscribe, repositories and layering.

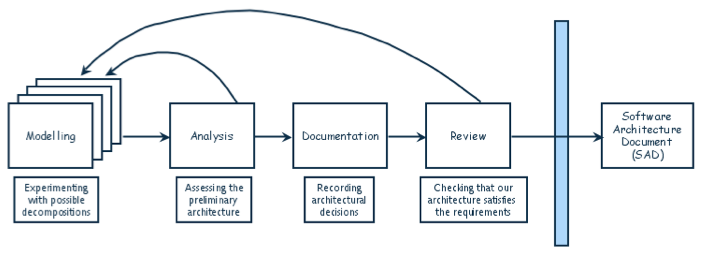
**Software architecture/design**

Software architecture refers to the high level structures of a software system. We now focus on the HOW (not on the WHAT) → HOW to implement the customer's requirements.

*Sources of design:*



*The design process:*



**Software units**

Architecture design decomposes the system into software units (depending on aspects considered):

* Components
* Subsystems
* Runtime processes
* Modules
* Classes
* Packages
* Libraries
* Procedures

We use the term software unit when we want to talk about a system’s composite parts without being precise about what type of part.

A design is modular when each activity of the system is performed by exactly one software unit (the inputs and outputs of each software unit are well-defined).

A software unit is well-defined when its externally visible behaviour is precisely specified.

**Architectural views**

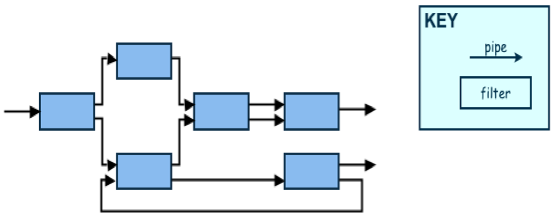
In addition to decompose the system in software units, the system could be analysed with different architectural views.

Common types of architectural views:

* Decomposition view (traditional view): portrays the system as programmable units (typically a hierarchy of models)
* Dependencies view: shows dependencies among software units (useful in project planning)
* Generalization view: shows software units that are generalizations or specializations of one another (useful when designing abstract or extendible software units)
* Execution view: shows the runtime structure of a system in terms of its components and connectors (the traditional box-and-arrow diagram)
* Implementation view: maps software units to source files (helps programmers find implementations)
* Deployment view: maps runtime entities to computer resources (helps the architect analyse the quality attributes)
* Work-assignment view: decomposes the design into work tasks (helps project managers to plan and allocate project resources and to track each team's progress)

**Architectural styles**

*Pipes-and-filter:*



In a pipe-and-filter style, system functionality is achieved by passing input data through a sequence of data-transforming components, called filters, to produce output data. Pipes are connectors that simply transmit data from one filter to the next without modifying the data.

+ Reason by functional composition

+ Filters can be reused easily

+ Easy evolution: add, replace, remove filters

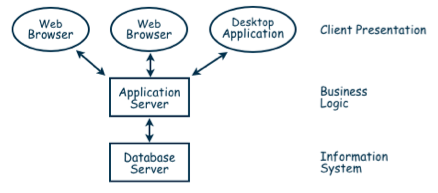
+ Allow concurrent execution of filters

+ Compositional throughput analysis

– Overhead of parsing inputs, producing outputs

– Batch processing, not good for interactive

*Client-server:*



Servers offer services and clients access them using a request/reply protocol.

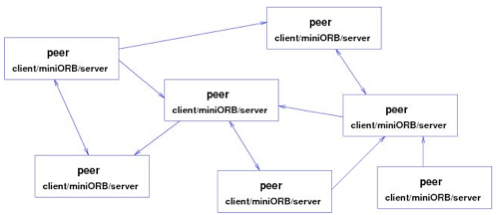
Note that clients know servers but servers don't know clients.

+ Can distribute servers among processors → improve performance

+ Servers can be layered (multitier)

+ Supports reuse of servers across applications

*Peer-to-peer (P2P):*



All components are peers (equals), they execute concurrently and act as both clients and servers (i.e. specify both provided and requested services).

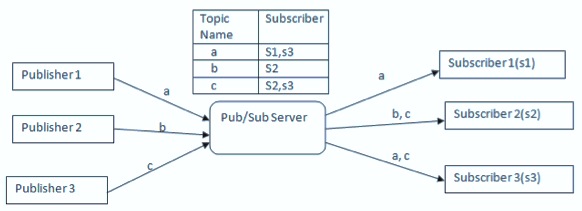
In other words, any component can call any component.

+ Scale up well

+ Increased system capabilities (replication)

+ Highly tolerant of failures (replication)

*Publish-subscribe:*



Subscribers subscribe to events from a server (bus) by registering a callback (implicit invocation). Publishers publish (broadcast) events to the server (bus). Registered subscribers receive the events (the server calls the callback).

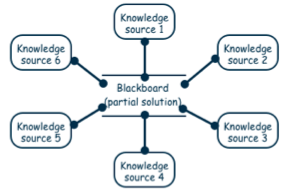
Note that publishers and subscribers don't know each other (all know the server and the events).

+ Easy evolution, customization, reuse

– Need shared repository for shared persistent data

– Difficult to test

*Repositories:*



A repository style of architecture consists of two types of components: a central data store and associated data-accessing components. Shared data are stockpiled in the data store, and the data accessors are computational units that store, retrieve, and update the information.

In a traditional database, components are active (clients) and data store is reactive.

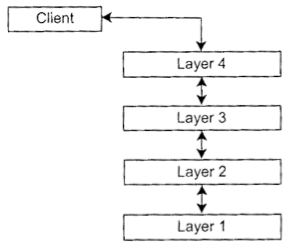
In the blackboard type of repository, blackboard (data store) is active and knowledge sources (components) are reactive.

+ Centralized management of data

+ Improve performance (distribute)

– More complex, consistency, security (distribute)

*Layering:*



Layered systems organize the system’s software units into layers, each of which provides services to the layer above it and acts as a client to the layer below. We can thus say that each layer raises the level of abstraction.

+ Relatively easy to add or modify a layer

+ Porting on different platforms

– Not applicable to all systems

– System performance may suffer extra coordination among layers

1. Discuss tactics to achieve modifiability, performance, security, reliability, robustness and usability.

**Modifiability**

→ Design must be easy to change.

The goal is to minimize the number of software units requiring changes. This affects units in two ways:

* Directly affected: responsibilities change
* Indirectly affected: only implementation changes

*Tactics for directly affected units:*

Cluster the anticipated changes.

* Anticipate expected changes: encapsulate each anticipated change in one unit
* Cohesion: all pieces, data, and functionality of each unit contribute to its purpose and responsibilities
* Generality: accommodate change by modifying a unit's inputs rather than modifying the unit itself

*Tactics for indirectly affected units:*

Reduce dependencies.

* Coupling: reduce the degree to which a unit depends on another
* Interfaces: interact with other units only through their specified interface
* Multiple interfaces: provide a new interface for new data or services (preserving existing interfaces)

**Performance**

→ Constraints on system speed and capacity (response time, throughput, load).

*Tactics:*

* Increase resources
* Improve utilization of resources (e.g. concurrency)
* Improve allocation of resources (scheduling)
* Reduce demand for resources (increase efficiency)

**Security**

→ Controlling access to resources.

*Tactics for immunity (preventing an attack):*

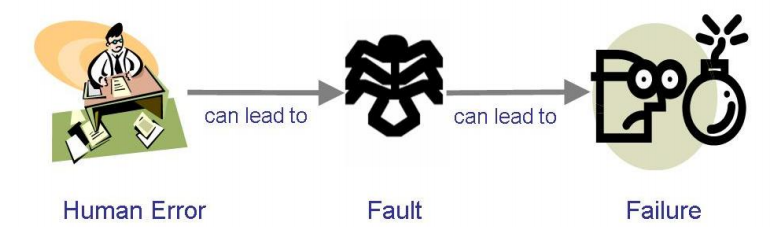
* Ensure that all security features are included
* Minimize exploitable security weaknesses

*Tactics for resilience (recovering from an attack):*

* Segment the functionality to contain attack
* Enable the system to quickly restore functionality

**Reliability**

→ Performing correctly under assumed conditions. System correct = 100% reliable.



Fault = a defect in a product

*Two kinds of tactics:*

* Preventing faults, by writing well the code and avoiding human error.
* Tolerating faults, FDIR -> Fault detection, identification and recovery.

Two kind of detection, active (look for symptoms) vs. passive wait fault.

**Robustness**

→ Performing correctly under adverse conditions.

Safe = free from unacceptable behaviours.

Defensive design: anticipate external problems

Mutual suspicion: check inputs for correctness, consistency, pre-conditions)

Redundant calculations (ex: space shuttle, radiation glitch)

State recovery tactics (same as for reliability)

**Recovery tactics**

Both robustness and reliability can use the following technics to recover from undesirable state:

* Undoing transaction (ex: database)
* Checkpoint / rollback (saved state)
* Backup
* Degraded service (provide an older stable version)
* Correct and continue (fix the symptoms)
* Report (returns to previous state and record the problem)

**Usability**

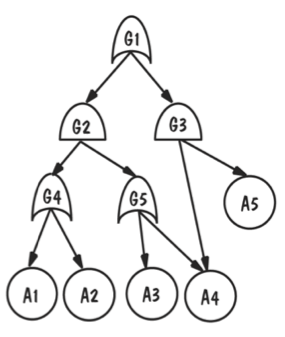
→ Ease of use of the system.

*Tactics at architectural level:*

* UI should be in its own software unit/layer
* Need a UI event handler process
* Support for UI functionality cancel/undo, views, …
* Maintain a model of its environment Mme, industrial process

1. Explain the principles of fault tree analysis and cut-set computation. Explain the principles of trade-off analysis with weighted comparisons. Describe cost-benefit analysis based on return on investment and payback period. Define the notion of quality model and describe an example.

**Fault tree analysis**



Root = failure

Leaves = faults, events

Nodes = logic gates (AND/OR)

Edges = caused-by

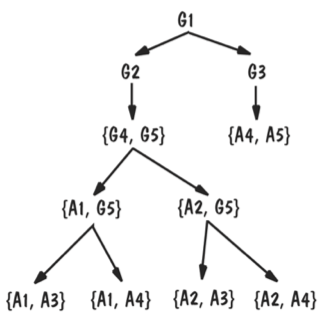
Fault tree analysis is a method that helps us to examine a design and look for fault that might lead to failure.

*Building fault tree:*

1. Identify possible failures

2. For each failure, build the fault tree trace backwards through the design

**Cut-set computation**



Nodes = sets of nodes of the fault tree

Root = {root of fault tree}

Leaves = sets of leaves of the fault tree

= cut-sets

The cut-set is the minimal set of events (faults) that can cause a failure.

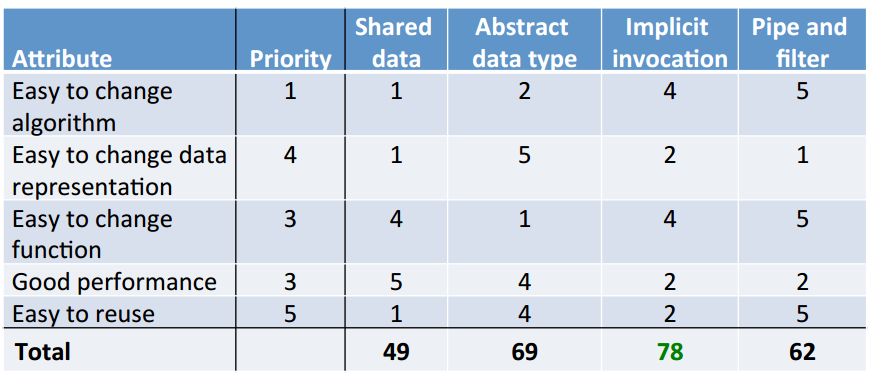
*Building cut-set tree:*

1. Assign the top node of the cut-set tree to match the logic gate at the top of the fault tree
2. Working from the top down, expand the cut-set tree as follows:
   1. Expand an or-gate node to have two children, one for each or-gate child
   2. Expand an and-gate node to hate a child composition node listing both of the and-gate children
   3. Expand a composition node by propagating the node to its children, but expanding one of the gates listed the node
3. Continue until all leaf nodes are basic events or composition nodes of basic events

**Trade-off analysis with weighted comparisons**

We have often several alternative designs to consider. To make a good choice we need a measurement-based method for comparing design alternatives.

Example of measurement method: weighted comparison.

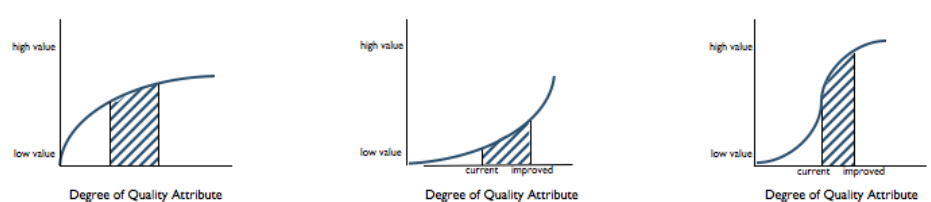


All the designs are on X-axis, and judgment criterions are on Y-axis.

Give a score for each, and select the solution with the best global score.

**Cost-benefit analysis**

Evaluation of the design based on business aspects and economical value of the product rather than design quality: benefits vs. cost.



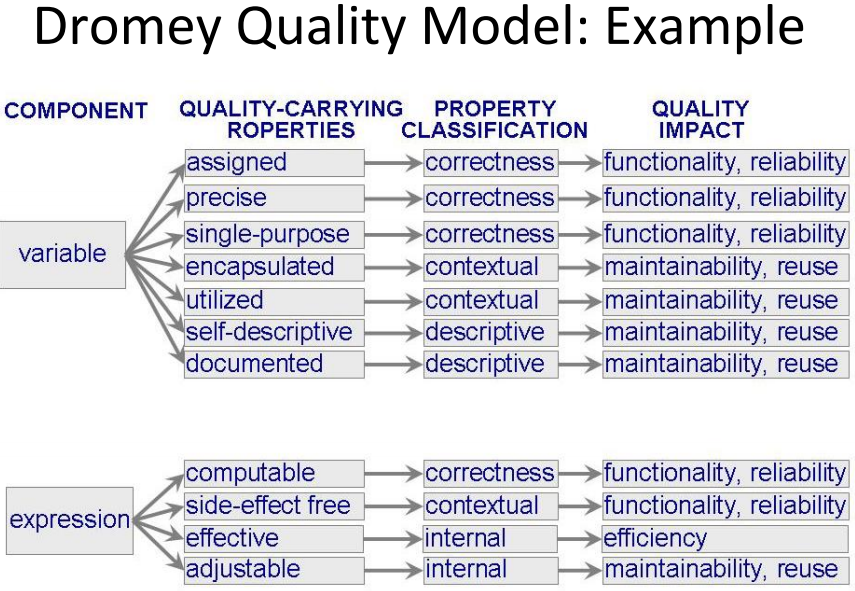
Value is not linearly dependant of quality…

You have to find the best compromise to increase ROI (Benefits / Cost) or the minimal payback period (cost / yearly benefits).

**Quality model**

Looking for the desirable attributes of a product (e.g. document, file, system…)

The product quality model makes hierarchical nomenclature of product quality characteristics.



In the example above, you see that different components (variable, expression) are associated to some quality (documented, encapsulated…) that are classified and associated to a quality impact.

Baseline and targets:

* Measure quality attributes relative to baseline (typical result)
* Require quality attribute relative to target (minimal acceptable result)

1. Define program design. Describe modularity in terms of coupling and cohesion. Discuss incremental development based on the uses graph. Discuss information hiding, abstraction and generality.

**Program design**

We focus here on how each software unit will be built. It is a basis for writing the code.

We use the term software unit when we want to talk about a system’s composite parts without being precise about what type of part (Library, module, class, package, etc.).

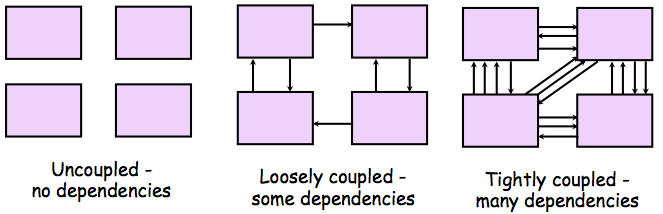
**Modularity**

→ Keeping unrelated concerns of a system in separate modules (i.e. one concern = one module).

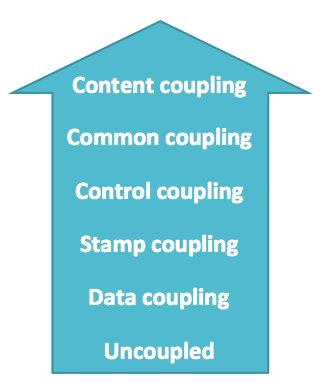
We can measure the separation of concerns in terms of coupling and cohesion.

*Coupling:*

The degree of interdependence between modules (the less, the better).



Types of coupling:



Tigh coupling

Loose coupling

Low coupling

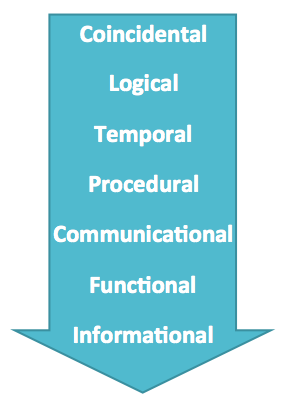
There are different types of coupling (worse to best order):

* Content: A branch into the code of B, method invocation
* Common: A and B share common data, public variable
* Control: A controls the behaviour of B, ???
* Stamp: A pass data structure to B, they are to agree on format
* Data: A pass simple data to B, **only necessary data**, an integer via parameter

*Cohesion:*

The dependence among a module's internal elements (the more, the better).

Types of cohesion:



Low cohesion

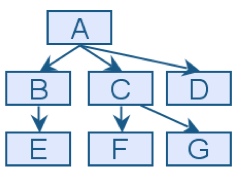
High cohesion

Different types of cohesion (worst to best order):

* Coincidental: parts are unrelated
* Logical: parts related by the logic structure of the code (eg. Common code but unrelated functions)
* Temporal: parts are used in the same time in an execution
* Procedural: parts belong to a common procedure (temporal + common purpose)
* Communication: parts operate on the same data set
* Functional (best): all and only parts essential to a single function
* Informational: all and only parts essential to a single abstraction

**Incremental development based on the use graph**

The goal is to implement and test incrementally (i.e. identify progressively larger sets of units). To do that, we need to know the dependencies between units → use graph.



Nodes = software units

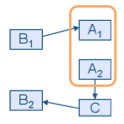
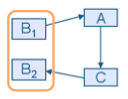
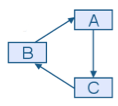
Edges = « uses »

Fan-in: the number of units that use a particular unit.

Fan-out: the number of units used by a particular unit.

Goal: high fan-in, low fan-out.

Cycles in uses graph = mutual dependency → Sandwiching: decomposing a unit into two units



**Information hiding**

Each design decision is hidden inside a software unit = encapsulation.

Information hiding helps cohesion (hide data representation, hide an algorithm) and reduce coupling.

In OO design, an object is abstract type that hides data representation. Objects cannot be totally uncoupled.

**Abstraction**

Omitting some details (e.g. pseudo-code vs. full code (with some non-pertinent aspect about the algorithm)).

Different kinds of details:

* Decomposition hierarchy => abstract from internal structure
* Interface specification => abstract from internal design details
* Architectural views => abstract on one particular aspect
* Virtual machine (layer) => abstract from lower level machine (layer)

**Generality**

Make a software unit as universally applicable as possible.

Several ways of doing this:

* Parameterizing context-specific information => reuse by changing parameters instead of code
* Removing pre-conditions => less conditions for reuse
* Simplifying post-conditions => decompose into simpler units, more broadly reusable

1. Describe the principles of design-by-contract and the elements of an interface specification. Discuss how exceptions can be used to weaken pre-conditions.

**Answer of Charles to my question:**

La deuxième partie de la question est couverte un peu avant dans les slides.  Je pourrais en effet grouper cette question courte avec la question 10, par exemple.  L'important est que vous soyez à même de répondre à cette question.

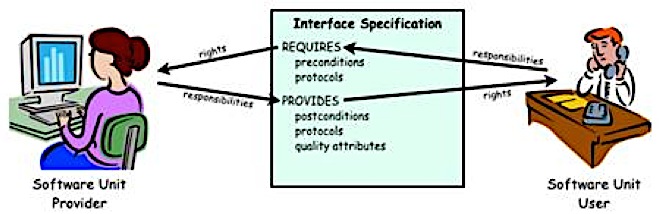
**Design-by-contract**

In design-by-contract, each software unit has an interface specification that precisely describes what the module is supposed to do. The documentation must provide enough details to support implementation.

Specification = contract between the provider and user of software unit.

Mutual assumptions and guarantees.

Mutual rights and responsibilities.



**Assertions**

**Function pre-conditions:** required when the function is invoked

**Function post-conditions:** ensured when the function is terminated

**Invariants:** ensured at initialization and after every (public) function invocation

**Protocols:** Required and ensured constraints on invocations (eg open/close db connection)

**Quality attribute:**  ensured by all functions

**Exceptions used to weaken pre-conditions**

Weaken preconditions allow to improve generality by making a software unit more universally applicable. Precondition limit the applicability of a method, by using exception, you can simplify the preconditions, by catching exception rather than require some conditions at method invocation.

Eg: function divide raise exception rather than check if divisor > 0.

But not all preconditions can be handled in this way…

1. Define interfaces and classes in object-oriented design. Explain typing and inheritance and the principles of polymorphism. Compare inheritance and composition. Explain Liskov’s substitutability principle. Explain and discuss the Law of Demeter. Describe the general principles and constituting elements of design patterns.

**Interfaces and classes in object-oriented design**

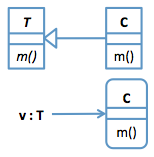
*Interface:*

Set of visible attributes and methods (signatures and specifications). It defines an abstract data type.

*Class:*

Software module implementing an interface (i.e. it implements an abstract data type).

**Typing, inheritance and polymorphism**



*Typing:*

If v : T refers to o : C

Static typing: only methods of T can be invoked on v, checked at compilation.

Dynamic binding: when a method is invoked on v, the implementation of C is executed, checked at runtime.

*Inheritance:*

Defining a new class (or interface) by reusing and extending an existing class (or interface).

A inherits from B iff A extends B iff A is a subtype of B.

*Polymorphism:*

Different implementations from different classes C are executed for the same invocation on v.

Ability of the code to act on values of multiple types.

Improve the code reusability, e.g. implement one time a data structure such as list, array for all the different data types.

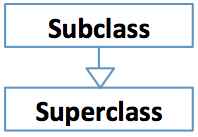
**Comparison between inheritance and composition**

Two main techniques for constructing larger objects from smaller objects:

* Inheritance: extend an existing class aka. White-box reuse. Use when subtyping is useful.
* Composition: combine simpler classes aka. Black-box reuse. Preferred in general.

|  |  |
| --- | --- |
| Composition | Inheritance |
| + Only Composite interface is visible  + Component can change dynamically  + Multiple components is possible  + Composite may use a Component interface  – Component methods must be forwarded in Composite  – Component unrelated to Composite | + Component methods invoked directly  + Composite subtype of Component  – Interface of Component is visible in Composite  – Component fixed at design time  – Only one Component is possible  – Composite must extend a Component class |

**Liskov’s substitutability principle**



*Substitutability:*

* Subclass is a subtype of Superclass
* Subclass instances can be used as Superclass instances
* Subclass must preserve the behaviour of Superclass
* Subclass must respect the contract of Superclass, so that Subclass instances are substitutable to Superclass instances

N.B. this is a guiding principle, not an absolute rule.

*Liskov’s substitutability principle:*

Subclass is substitutable for Superclass if:

1. Subclass supports all the methods of Superclass (with compatible signatures)

2. Subclass methods satisfy the specifications of Superclass methods

3. Subclass preserves all properties of Superclass

**Law of Demeter (aka. "Don't talk to strangers")**

Each unit should have knowledge only on directly related units.

In OO programming: each class should invoke methods only on directly referenced classes.

+ Reduces dependencies

+ Client of a composite need not know its components

– Needs additional wrapper functions, may degrade performance

**General principles and constituting elements of design patterns**

A design pattern codifies design decisions and practices for solving a particular design problem according to design principles.

They are templates for a solution that must be modified and adapted for each particular use.

*Elements of a design pattern:*

* Name: for easy reference, enriches our design vocabulary
* Problem: when to apply, objectives
* Solution: structure, elements and relationships
* Consequences: benefits and costs, trade-offs

1. Explain the goals and principles of software testing. Describe the different stages of a software testing process, from unit to installation. Discuss the different types of faults. Define and compare white-box and black-box testing.

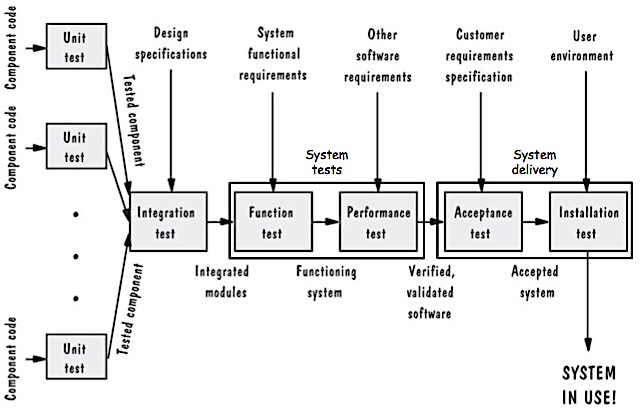
**Software testing**

The goal of software testing is to discover faults and not to demonstrate correctness,

In other words a test is successful when a fault is discovered.

When a fault is discovered, we identify the fault (what fault caused the failure) and correct it (making changes to remove the fault).

**Stages of a software testing process**



*Unit test (aka. module test, component test):*

Each program component.

*Integration test:*

Assemble components together and check correct interaction.

*Function test:*

The whole system.

Check that the behaviour conforms to functional requirement specifications.

*Performance test:*

The whole system.

Check that the behaviour conforms to nonfunctional requirement specifications.

*Acceptance test:*

The whole system.

Test using the system with the customer.

Check that the behaviour conforms to (customer) requirement documentation.

*Installation test:*

The system in its operational environment.

Check that the system performs properly in its environment.

**Types of faults**

* Algorithmic faults: the program logic produces a wrong computation
* Computation and precision faults: a mathematical computation is wrong or the result does not have the required precision
* Timing or coordination faults: incorrect event timing or coordination (real-time systems)
* Documentation faults: documentation doesn't match what program does
* Throughput or performance faults: system's performance not acceptable
* Capacity or boundary faults: System's performance not acceptable when activity increases
* Hardware and system software faults: supplied hardware or system software does not work as specified
* Standard and procedure faults: coding standards and procedures not respected

**White-box and black-box testing**

*Black box (or closed box):*

Content is unknown or ignored.

Test input/output behaviour (functional).

+ Not constrained by the structure of the unit under test

– Impossible to know whether a test is complete

*White box (or clear box):*

Content is visible and observed.

Test internal operation (structural).

+ Can check that program elements are tested

1. Discuss the principles of code reviews. Describe the principles of formal program proofs. Define test cases, test suites, test equivalence classes. Discuss functional and structural test objectives. Define the notion of code coverage and explain some coverage criteria.

**Principles of code reviews**

Similar to requirements and design reviews.

Review code and documentation.

Team: the programmer(s) + 3-4 experts (no customer because they are not concerned with the implementation).

*Code walkthrough: less formal*

The programmer presents the code and documentation and the team comments on correctness.

*Code inspection: more formal*

List of review concerns (data types, algorithms, comments, etc…)

Overview meeting -> individual review -> report meeting

Lead by a moderator (not the programmer)

Experience show that code review is very effective at detecting faults.

**Principles of formal program proofs**

Prove code correctness, by proving that program respect the specifications formally.

+ Gives a formal understanding of the program

+ Strong guarantee of correctness

– Much work (more than writing the program) writing assertions, carrying the proofs

– More complex programs are harder to prove data structures, pointers, concurrency, …

– Only proves assertions

– Proofs can be faulty

**Test definitions**

*Test case (or test point):*

Test an output for an input

*Test suite:*

A series of test cases.

*Test equivalence classes:*

The space of possible inputs X is partitioned into equivalence classes:

x ≈ x' iff x and x' will detect the same faults

Goal: choose one test case in each equivalence class (more than one => rendundant ; less than one => incomplete).

**Functional and structural test objectives**

The test objectives can be functional or structural.

*Functional:*

Test cases based on functional specifications.

All functions are performed correctly => black box.

*Structural:*

Test cases based on code structure (test the branches).

All statements are executed correctly => white-box.

**Code coverage**

*Coverage criteria:*

In relation with structural test, to measure what parts of the code have been exercised/tested.

Statement coverage: all program statements

Branch coverage: all program branches

Path coverage: all program paths

->impossible in general (unbounded loop)

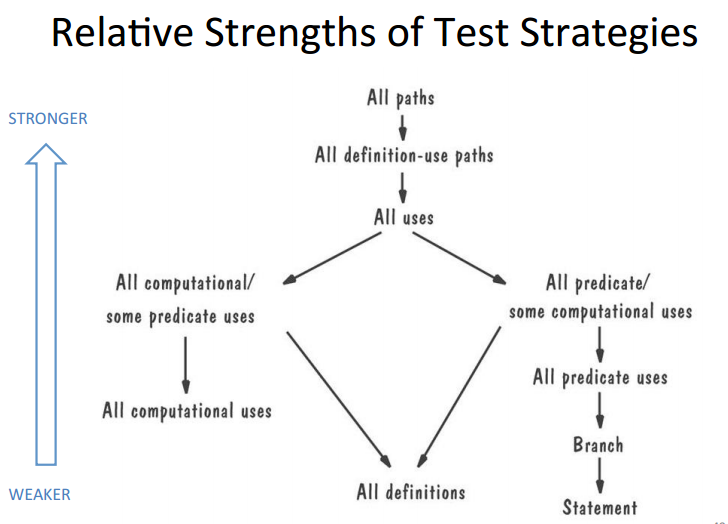
*Def-use coverage criteria:*

Def-use pair: from a variable definition to a use of that variable that is reachable from this definition.

All-uses coverage: all def-use pairs

Def-use path coverage: all paths of all def-use pairs.

Also distinguish predicate and computational uses…



1. Describe and compare different strategies for integration testing, and the need for stubs and drivers. Discuss the case of testing object-oriented designs. Cite software tools that can be used for program testing and analysis. Describe fault estimation using fault seeding and by independent test groups.

**Drivers and stubs**

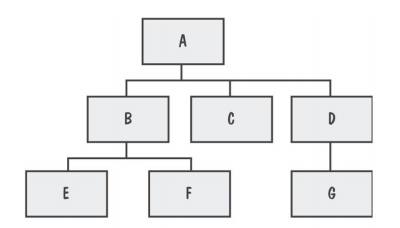
How can you test a function that depends on other functions?  
Use test drivers and stubs.

**Driver**: a program that calls a component (passes a test case to it).

**Stub**: a program that is called by a component (simulates the activity of a missing component), i.e. function which simulates the behaviour of the called function…

**Never modify a component to support testing**! Drivers and stubs are separate programs.

**Strategies for integration testing**

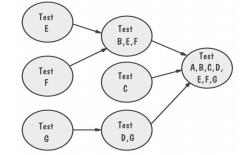


Consider the component hierarchy

*Bottom-up:*

Useful if many general-purpose utility routines, reused components at lowest level.

Needs drivers.



+ Suitable for object-oriented programs

– The top-level components are tested last (most important, may reveal design bugs)

*Top-down:*

Test the top-level, controlling component first.

Needs stubs.



+ Allows test by external function

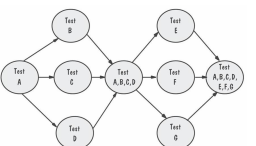
+ Major design faults or issues revealed early

+ Drivers not needed

– Stubs can be difficult to develop, affects validity of the test

*Modified top-down:*

Test components individually before integrating components



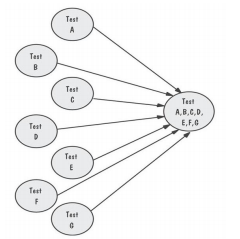
+ Avoids large number of stubs

– Needs stubs and drivers

*Bing-bang:*

Everything integrated in one shot.

For small systems only.



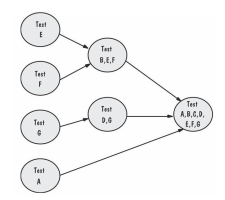
– Needs drivers and stubs

– Faults are hard to localize

– Interface faults are hard to distinguish

*Sandwich:*

Combine bottom-up and top-down, converge to target middle layer.

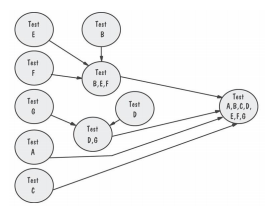


+ Early test of top layer

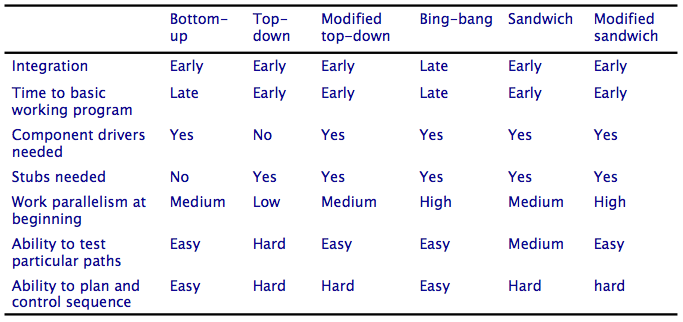
– No stubs for utility components in bottom layer

*Modified sandwich:*

Test upper-level components before merging.



*Comparison:*



All except bottom-up need stubs!

All except top-down need drivers!

**Testing object-oriented designs**

Unit testing is easier but integration testing is more extensive.

Ex: Dynamic binding, complex interface, polymorphism, inheritance are aspect related to OO which cause difficulties during test.

**Software tools for program testing and analysis**

**Automated testing tools:**

*Test execution: planning and running of the tests*

* Capture and replay (keystrokes)
* Generating stubs and drivers
* Automated testing environments (database, measurement, analysis, simulation, ...)

*Test case generators:*

* Structural (structure of source code)
* Functional (specifications)
* Random

**Code analysis tools:**

*Static analysis: analyse the source code*

* Code analyser
* Structure checker
* Data analyser
* Sequence checker
* Measurements: paths, fan-in/fan-out, decision points, ...
* Program provers

*Dynamic analysis: execute the program*

* Program monitoring (coverage)
* Instrumentation, breakpoints

**Fault estimation**

How to estimate how many faults remain? (Help us to know how to stop tests)

Intentionally insert faults in the program:

* A known number S
* Same kind and complexity as actual faults

Use the number of discovered seeded faults s and the number of discovered actual faults n to estimate the number of actual faults N.

Hypothesis: same effectiveness → n / N = s / S → N = S . n / s

Manque Independent test groups<w

1. Define function testing. Explain the principles of cause-and-effect graphs and their use in testing. Describe different types of performance tests. Define and differentiate acceptance and installation testing. Describe different types of acceptances tests. Describe and compare the different types of test documents.

**Function testing**

Does the integrated system perform as required by the requirements specification?

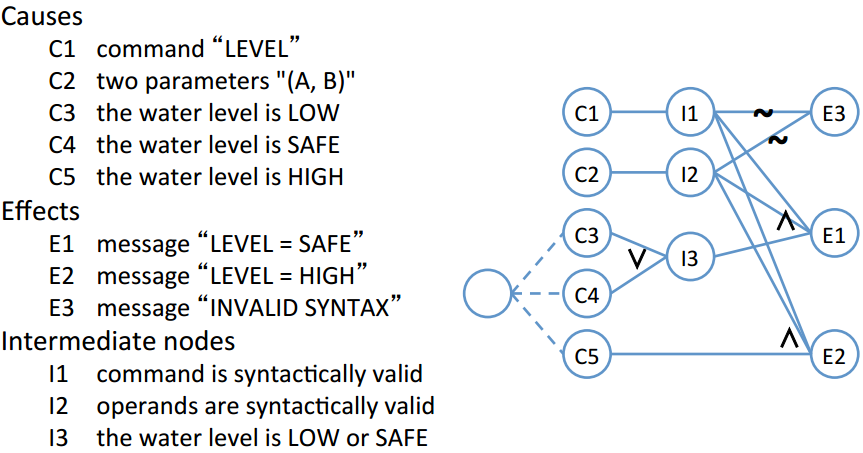
Ignores structure, focuses on functionality (black box).

Follow functional requirement specifications subdivided in smaller units (in successive sets, on partial systems (spins)).

**Cause-and-effect graph**

Idea, deriving test cases from the requirements. Formalise the requirements inputs/outputs.

Make the cause-and-effect graph: Logical relationships between causes (inputs) and effects (outputs, transformations). Same idea as fault trees.



**Types of performance tests**

(Verify non functional requirements)

* **Stress tests**: max capacity for a short time
* **Volume tests**: large amounts of data
* **Configuration tests**: all possible configurations
* **Compatibility tests**: interactions with other systems
* **Regression tests**: performance is at least as good
* **Security tests**: availability, confidentiality, integrity
* **Timing tests**: response times
* **Environmental tests**: physical factors: humidity, temperature, chocks, etc…
* **Quality tests**: reliability, maintainability, availability
* **Recovery tests**: response to faults, loss of data, loss of power, etc...
* **Maintenance tests**: diagnostic tools and procedures
* **Documentation tests**: available, consistent, easy to read, compliant
* **Human factors (usability) tests**: ease of use, so/ware and documents

**Differences between acceptance and installation testing**

**Accepting testing**, goal: enable the customers and users to determine if the system meets their needs. Written, conducted and evaluated by the customers. If the customer approve then the product is accepted (wrt. The contract).

**Installation test**, at the user’s site, when configured, test if system works on site as in the lab.

**Types of acceptance tests**

**Benchmark** **test**:

Test case representing typical operation conditions to value competing products

**Pilot test**:

Test through real use but with guidance

**Alpha/beta test**:

Alpha in house

Beta with selected pilot customer

For widely distributed products

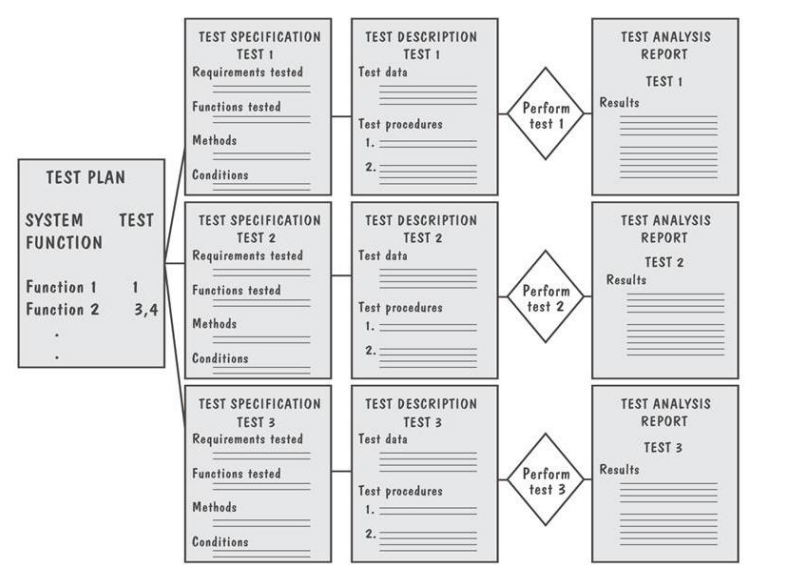
**Parallel test**:

New system operates in parallel with old system

E.g. Payroll system, you compare pay checks from old and new system

Combine compatibility and function test

**Types of test documents**

****

* **Test plan**: describe the system and planned tests.
  + Contains objectives, a summary, schedule etc…
* **Test specification and evaluation**: describe each test and evaluation of test results
  + Specify requirements, test approach, completion criteria, evaluation

methods, etc...

* **Test** **description**: test data and procedure (to realise it) for each test
  + Specify means of control, the data (input output), the procedures (kind of script), etc…
* **Test analysis report**: result of each test
  + Documents the results of a test, show how to determine completion, shows how to duplicate failure etc…

1. Define reliability, maintainability and availability, and their expression as MTBF and MTTR. Discuss the case of a constant failure rate. Explain the principles of failure prediction models. Explain and discuss the principles of statistical testing.

**RAM attribute (reliability, availability, maintainability)**

*Reliability* is the probability that a system will perform its intended function satisfactorily for a given time interval.

*Maintainability* is the probability that maintenance of the system will retain the system in, or restore it to, a specified condition within a given time period.

*Availability* is the probability that the system is operating satisfactorily at any time, and it depends on the reliability and the maintainability.

Hence the study of probability theory is essential for understanding the reliability, maintainability, and availability of the system.

**Mean time between failures (MTBF)**

Average time between successive failures. Measures reliability.

**Mean time to repair (MTTR)**

Average time to fix a fault. Measures maintainability.

Availability: A = MTBF / (MTBF + MTTR)

**Constant failure rate**

The system will fail at constant probability rate of once per hour.

Then, MTBF = 3600 s

Density function: f(t) = 1/3600 \* exp ( -t / 3600) (exponential law)

Distribution function : F(t) = 1 – exp(-t / 3600)

Reliability (R (t) = 1 – F(t)) : R(t) = exp(-t / 3600)

Reliability (for t = MTBF): R(MTBF) = exp (-1) = 37%

**Failure prediction models**

Goal: Given a series of failure times t1, t2, …, tn-1

Predict the distribution of future failure times Tn, Tn+1, …

A good prediction model must include:

* Prediction model: probability specification of the stochastic process.
* Inference procedure: infer unknown parameter from t1…tn-1.
* Prediction procedures: combine model and inference procedure to make predictions about future failure behaviour.

**Principles of statistical testing**

Reliability prediction based on failures occurring during testing

Will this be accurate for typical system usage?

*Operational profile*: probability distribution on inputs (reflecting usage)

*Statistical testing*: select tests according to operational profile

Tests focus on more used parts (=> better observable reliability)

Test reflect usage (=>reliability predictions more accurate)

But operational profiles are difficult to define.

Mislead, a small % of operational profile may account for a large % of failures.

1. Discuss issues with testing safety-critical systems. Describe the principles of safety cases and FMEA analysis. Define and discuss the merits of software diversity and formal verification.

**Issues with testing safety-critical systems**

A safety-critical system is a system in which a failure can harm or kill people (e.g. airplane, nuclear plant, medical device, etc.).

Safety-critical systems demand very high reliability:

10-9 failures per hour = once per 100 000 years

=> It cannot be tested directly!

**Principles of safety cases and FMEA analysis**

*Safety case:*

We can decompose the safety goals and assign failure rates or constraints to each component of the design, so that satisfying each lower-level goal will “roll up” to allow us to meet safety goals for the entire system. In this way, we make a safety case for the system, making explicit the ways in which our software meets performance goals for safety-critical systems.

***Dans les slides:*** A structured argument, supported by evidence, intended to justify that a system is acceptably safe.

*Failure Modes and Effects Analysis (FMEA):*

A failure mode, is a situation which contains risks.

FMEA, analyse the failures mode (e.g. by using a fault tree), to look after some unknown system effect. A system effect is a hazardous situation which cans lead to a crash. By default (because we don’t know) we considers this situation as a crash.

FMEA is highly labour-intensive and based on the experience of the analysts. It usually involves our performing an initial analysis of the software design, abstracting modes that might lead to failures. Then we look at how combinations of the basic failure modes might lead to actual failures.

Summary: Identify failure modes, analyse system effects.

**Merits of software diversity and formal verification**

*Design diversity:*

The same system is built according to the same requirements specification but in several independent ways, each according to a different design. Each system runs in parallel with the others, and a voting scheme coordinates actions when one system’s results differ from the others’.

The underlying assumption is that it is unlikely that at least three of the five groups of developers will write incorrect software for a given requirement.

However, there is empirical evidence suggesting that independently developed software versions will not fail independently; diverse designs do not always offer reliability higher than of a single version.

*Formal verification:*

Idea: usage of formal verification techniques with our requirements, designs, and code.

But formal evaluation of natural language is impossible, and important information may be lost if we translate natural language to mathematical symbols. Even formal proofs of specification and design are not infallible, because mistakes are sometimes made in the proofs.

1. Describe and compare the different types of training and documentation with respect to different types of user.

**Training**

*Types of users:*

* Users: Exercise the main system functions (create, edit, analyse data, draw graphs, communicate, …)
* Operators: Perform supplementary functions (create back up copies of data files, define who has access to the system, …)

*User training:*

* Focus on primary functions
  + Data creation, deletion, sorting etc…
  + Navigation through the system
* Focus on the what, not the how (no mention of internal structures and algorithms)
  + No mention of internal structure
* Relate to how the functions were performed before the new system

*Operator training:*

* Focus on support functions
  + System configuration, start up, shutdown etc…
* Focus on the how, not the what (how the system works, internal structures)

*Different types of training:*

* Initial training (complete training at system delivery)
* Later training (complete training for new users)
* Brush up training (review training on selected features)
* Infrequent user training (only basic functions)
* Specialized training (specific function)

**Documentation**

*Types of users:*

* Users: Primary functions
* Operators: Support function
* Customer staff: General description
* Development team: Implementation details

*Types of documentations:*

* User’s manual
  + Start with general purpose
  + Progress to detailed functional description (manual, focus on how to use)
  + User friendly, colours, glossary etc…
* Operator’s manual
  + Globally the same as previous
  + Additional advanced functions details (back up, configuration, access, etc...)
* General system guide (Customer)
  + Describe the solution provided
  + Hardware and software configuration
  + Philosophy behind the system’s construction
* Programmer guide (maintenance team, extension developers)
  + Describe components, functions performed
  + Support functions (debugging, diagnostics…)
* Failure reference guide
  + Additional information about exceptions (friendly compare to Blue screen)
* Quick reference guide