Software Engineering 2

January 17, 2013

Contents

1 Definitions

- **Systematic** approach to development, operation, maintenance, deployment, retirement of software
- **Methodological** and managerial discipline concerning the systematic production and maintenance of software products that are developed and maintained within anticipated and controlled time and cost limits

1.1 Product quality attributes

- **Correctness**: software is correct if it satisfies the specifications
- **Reliability** : can be defined mathematically as "probability of absence of failures for a certain time period"
- **Robustness**: software behaves "reasonably " even in unforeseen circumstances (e.g., incorrect input, hardware failure)
- **Performance**: efficient use of resources
- **Usability**: expected users find the system easy to use
- **Maintainability**
- **Reusability**: similar to maintainability, but applies to components
- **Portability**: similar to maintainability (adaptation to different target environment)
- **Interoperability**: coexist and cooperate with other applications

1.2 Productivity

Measured in person-month (person-day, person-year).

1.3 Waterfall organization

- **Feasibility**: cost/benefits analysis, it determines whether the project should be started, possible alternatives and the needed resources. Produces the **feasibility study document**.
- **Study Requirements**: produces the **requirement analysis and specification document** (RASD).
- **Analysis & Specification Design**: defines the software architecture. Produces the **design document.**
- **Coding & Unit Test**: determines the component implementations**.**
- **Integration & System Test**: modules are integrated into subsystem and tested (Alpha and Beta test).
- **Deployment**
- **Maintenance**
  - **Corrective**: deals with the repair of faults and defect founds.
  - **Adaptive**: adapts the software to changes in the environment (new hardware, new operating system).
  - **Perfective**: deals with new or changed user requirements.
  - **Preventive**: increases the system's maintainability.

2 CMM / CMMI: Capability Maturity Model

The CMM describes how well the process can deliver the expected product.

Level	Focus	Key process areas
5 Optimizing	Continuous process improvement	Defect prevention technology change management
4 Managed	Quantitative management	Quantitative process management software quality management
3 Defined	Process standardization	Organization process focus organization process definition training program integrated software management software product engineering intergroup coordination peer review
2 Repeatable	Basic project management	Requirements management software project planning software project tracking and oversight software subcontract management software quality assurance software configuration management
1 Initial	Competent people and heroics	

The aim of CMMI is to trigger a continuous improvement of the processes.

3 ISO-9000

ISO 9000 is a family of standards

- Related to quality management systems.
- Describe the requirements of a quality system.
- Define the minimal attributes that an organization should have.
- ISO 9000:2000 : fundamentals and vocabulary.
- ISO 9001:2000: requirements to meet ISO 9000.
- ISO 9004:2000 : guidelines for performance improvements.

Requirement to satisfy:

- **Systemic requirements**
  - e.g., control quality system documents
- **Management requirements**
  - e.g., satisfy your customer, define a quality policy
- **Resource requirements**
  - e.g., provide quality personnel
- **Realization requirements**

  - e.g., control realization planning

- **Remedial requirements**
  - e.g., monitor and measure quality

The ISO-9000 is just a certification of minimal acceptability requirement.

4 Project management

Project **scheduling** is the process of deciding how the work will be organized as separate tasks , and when and how the tasks will be executed.

The schedule are normally represented by graphical notation such as a **Gantt** chart.

5 Software effort and cost

The effort is estimated as $effort = A \cdot size^B \cdot M$ where A is an organization specific constant, B accounts for the possibly disproportionate effort in large project and M reflects product, project and processes attributes.

The attributes are estimated by project managers (most common product attribute is code size).

5.1 Size estimation

Many options:

- **Function points** (FP) of different types:
  - **Internal logical file (ILF)**: homogeneous set of data used and managed by the application
  - **External interface file (EIF)**: homogeneous set of data used b y the application but generated and maintained by other applications.
  - **External input**: elementary operation to elaborate data coming from external environment.
  - **External output**: elementary operation that generates data for the external environment.
  - **External inquiry**: elementary operation that invokes input and output.
- **Lines of code (LOC)** estimation (by converting the FP to LOCs)

5.2 Albrecht's method

Function types	Weight		
	Simple	Medium	Complex
Inputs	3	4	6
Outputs	4	5	7
Inquiry	3	4	6
ILF	7	10	15
EIF	5	7	10

Using the number of different function points and their relative weight given the previous table we obtain the **unadjusted function points** (UFP).

We can make a final correction from -35% to $+35\%$ based on 14 parameters:
$FP = UFP \times (0.65 + 0.01 \times \sum_{i=1}^{14} F_i$.
Correcting a UFP is not necessarily a good idea.

5.3 COCOMO - Constructive Cost Model

We assume that the project follows the traditional waterfall scheme.
Given:

- M the **effort** expressed in person-month
- T the **development calendar time** expressed in month.
- **KLOC** kilo lines of code used to measure the code size.

The COCOMO focuses of four phases:

- Planning and requirement analysis
- System design
- Development 1.0
- Integration and test

The parameters T and M refers just for the last three phases.
The COCOMO identifies three classes of applications:

- **Simple** (Organic mode): for small teams with reasonable experience and non-rigid requirements
- **Intermediate** (Semi-detached mode): medium size teams with mixed skills and both rigid and non-rigid requirements.
- **Complex** (Embedded mode): large teams or tight constraints.

The estimation can by obtained at different levels: basic, intermediate or advanced.
Given:

- S the program size

The number of required person is $N = M/T$.

5.3.1 Corrective coefficients (lesson 3 slide 35)

RELY - Required Software Reliability

- **Very low**: a fault does not cause actual problems.
- **Low**: the effect of a fault can be easily recovered.
- **Nominal**: the effect of a fault is important for the user, but he is still able to manage it without major problems.
- **High**: the effect of a fault can cause significant financial losses.
- **Very high**: risks for human beings.

5.3.2 Basic

Static single-valued model that computes software development effort (and cost) as function of program size expressed in estimated lines of code.
$M[man - month] = a_b S^{b_b}$
$T[months] = c_b S^{d_b}$

Application type	a_b	b_b	c_b	d_d
Organic mode	2.4	1.05	2.5	0.38
Semi-detached mode	3.0	1.12	2.5	0.35
Embedded mode	3.6	1.20	2.5	0.32

5.3.3 Intermediate

Computes software development effort as function of program size and a set of "cost drivers" that include subjective assessment of product, hardware, personnel, and project attributes.

$M_{Nom} = a_i S^{b_i}$

$M = M_{Nom} \Pi_{j=1..15} c_j$

Application type	a_b	b_b
Organic mode	3.2	1.05
Semi-detached mode	3.0	1.12
Embedded mode	2.8	1.20

5.3.4 Advanced

Incorporates all characteristics of the intermediate version with an assessment of the cost driver's impact on each step (analysis, design, etc.) of the software engineering process.

6 Requirement engineering

The requirement engineering is the process of discovering the purpose for which the software was intended, by identifying stakeholder and their needs, as documenting these in a form that is amenable to analysis, communication, and subsequent implementation.

The cost of correcting an error depends on the number of subsequent decision that are based on it:

Phase	**Relative cost**
Requirements	1
Design	5
Code	10
Unit test	20
Acceptance test	50
Maintenance	200

The **machine** is the portion of system to be developed.

The **world** (the environment) the portion of the real world affected by the machine.

Some world phenomena are shared with the machine, they can be controlled by the world and observed by the machine or vice-versa.

Goals perspective assertions formulated in terms of world phenomena (non necessarily shared). (Ex: For every urgent call an ambulance should arrive at the incident scene within 14 minutes; Reverse thrust shall be enabled if and only if the aircraft is moving on the runway).

Domain properties or assumptions are descriptive assertions assumed to hold in the world. (Ex: for every urgent call, details about the incident are correctly encoded; If locked, push cannot occur; Wheels pulses on if and only if the wheels turning).

Requirements are prescriptive assertions formulated in terms of shared phenomena. (Ex: when a call reporting a new incident is encoded, the automated dispatching software should mobilize the nearest available ambulance according to information available from the ambulances GPS and mobile data terminals; Reverse thrust enabled if and only if wheel pulses on).

6.1 Completeness

The requirements R are complete if

- R ensure satisfaction of the goals G in the context of domain properties D.
  - $R and D \models G$
    - A program P running on a particular computer C is correct if it satisfies the requirements R ($P and C \models G$).

6.2 Requirements

- **Functional**
  - Describe the interactions between the system and its environment independent of implementation.
    - A word processor user should be able to search for strings in the text.
- **Non-functional**
  - User visible aspect of the system not directly related to functional behavior. Also called Quality of Service (QoS) attributes.
    - The response time.
    - The availability.
- **Pseudo / Constraint**
  - Imposted by the client or the environment in which the system operates.
    - The implementation language.
    - The system should allow other system to use the same data.

6.3 Scenarios

The scenarios are a narrative description of what people do and experience as they try to make use of computer systems and applications.

- Identify actors
- Identify scenarios
  - As-is scenario.
  - Visionary scenario: describes a future system.
  - Evaluation scenario: user tasks against which the system is to be evaluated.
  - Training scenario: step by step instructions that guide a novice user through common tasks.
- Identify use cases
  - Participating actors.
  - Describe the entry condition.
  - Describe the flow of events.
  - Describe the exit condition.
  - Describe exceptions.
  - Describe special requirements
    - Constraint, non-functional requirements.
- Identify relationships among actor and use case
- Identify non functional requirements

6.4 Target qualities for RASD

- **Completeness**, the requirements are sufficient to satisfy the stakeholders goals given domain assumptions. The required software behavior is specified for all possible inputs.
- **Pertinence**: Each requirement or domain assumption is needed for the satisfaction of some goal. Each goal is truly needed by the stakeholders and the document does not contain info unrelated to the definition of requirements.
- **Consistency**: No contradiction in formulation of goals, requirements and assumptions.
- **Unambiguity**: Unambiguous vocabulary, assertions, responsibilities.
- **Feasibility**: the goals and requirements must be realizable within the assigned budged and schedules.
- **Comprehensibility**: must be comprehensible by all the target audience.
- **Traceability**: must indicate sources of goals, requirements and domain assumptions. Must link requirements and assumptions to underlying goals. Must facilitate referencing of requirements in future documentation (design, test case).
- **Good structuring**: every items must be defined before it is used.
- **Modifiability**: must be easy to adapt, extend or contract through local modifications. Impact of modifying an item should be easy to assess.

7 Software architectures and design

7.1 Design phases

- **Architectural design**: identify sub-systems.
- **Abstract specification**: Specify sub-systems.
- **Interface design**: Describe sub-system interfaces.
- **Component design**: Decompose sub-systems into components.
  - The boundary between the notation of subsystem and component is often blurred.
- **Data structure design**: design data structure to hold problem data.
- **Algorithm design**: design algorithms for problem functions.

7.2 Approaches for module identification

A module is a subsystem, its identification has different approaches:

- **Procedural abstraction**: involve functional decomposition. This leads to programs in which procedures represent distinct logical function in a program.
- **Functional programming**: a program is viewed as a function from a set of legal outputs and a set of desired outputs. This avoids persistent state (data flows through function while never really residing in any of them).
- **Object-oriented design:** the system is viewed as a collection of interacting objects. Objects are instances of object classes and interact via object methods.

7.3 Design methods

- **Top down \ hierarchical design**: starts at topmost components in the hierarchy and working down the hierarchy level by level. In practice large systems design is never truly top-down because some branches are designed before others.
- **Bottom up**: We have some pieces of code to reuse and we aggregate them increasing the level of abstractness reaching the topmost component.

7.4 Criteria for design methods

- Modular **decomposability**: this criterion is met if the methods supports the decomposition of a problem into smaller problems which can be resolved independently. Topdown design fulfill this criterion.
- Modular **composability**: this criterion is met if a methods leads to the production of modules that may be freely combined to produce new systems.
- Modular **understandability**: a methods fulfilling this encourages the development of modules which are easily understandable (has a meaningful names, is well documented, etc.)
- Modular **continuity**: a small change in the problem specification leads to a small change in modules.
- Modular **protection**: an unwanted condition at run-time only affects very few modules.

In principle there are best practices for a good design:

- Few interfaces: the overall number of communication channels between modules should be as small as possible (ideally n-1).
- Small interfaces: the information exchanged should be as little as possible.
- Explicit interfaces: If two modules communicate each other it must be obvious from the text of the two.
- Information hiding: expose as little informations as possible.
- Cohesion: A component should implement a single logical entity or function.
  - **Coincidental** cohesion (weak): parts of a component are simply bundled together.
  - **Logical** association(weak): components which perform similar functions are grouped.
  - **Temporal** cohesion (weak): components which are activated at the same time are grouped.
  - **Communicational** cohesion (medium): all elements of a component operate on the same input or produce the same output.
  - **Sequential** cohesion (medium): the output for one part of a component is the input to another part.
  - **Functional** cohesion (strong): each part of a component is necessary for the execution of a single function.
  - **Object** cohesion (strong): each operation provides functionality which allows object attributes to be modified or inspected.

Two component are **loose coupled** if component's changes does not affect the other. They are **tight coupled** instead. Object-oriented design enforces loose coupling because there's no shared state, however this is not true for inheritance because the changes made to attributes or operation in a superclass propagate to all derived classes.

7.5 Views

A view is an architectural description of elements. Each of them highlights different aspects of the architecture.

- **Functional** view defines the allocation of functionality to different components.
  - Class diagrams and interaction diagrams.
- **Module** view provides a logical decomposition of code in different modules.
  - Class diagrams, interaction diagrams, state machine diagrams.
- **Runtime** view defines runtime units (the components available at runtime) and shows how they collaborate.
  - Class diagrams, component diagrams, interaction diagrams.
- **Deployment** view defines the main deployment units and the guidelines for their installation.
  - Deployment, package diagrams.

UML is a tool, use it when it fits.

7.6 Component interactions

Components come in different form and flavor because they are developed by different people using different technologies. A component must change its form to fit the other component.

A software **connector** is an architectural building block tasked with effecting and regulating interactions across components. Typically they provide application-independent interaction facilities.

7.7 Architectural degradation

May happen when developers introduce changes in the software system that are not reflected in the corresponding architectural description (developer sloppiness, short deadlines, lack of documentation)

- **Drift**: the introduction of a design or implementation decision in the SW system that is not included nor implied by the architecture but that does not necessarily imply its outright violation.
- **Erosion**: the introduction of a design or a implementation decision in the software system that violate the described architecture.

7.8 Architectural styles

Certain design choices regularly result in solution with superior properties (elegance, effectiveness, efficiency, dependability, evolvability, scalability...).

Using architectural styles leads to many benefits:

- Design reuse: well understood solutions applied to new problems.
- Code reuse: shared implementations of invariant aspects of a style.
- Understandability of a system organization (one word can carry a lot of information).
- Interoperability: supported by style standardization.
- Style-specific analyses: enabled by the constrained design space.
- Visualizations: style-specific depictions matching engineers' mental models.

Example of architectural styles:

- **Object oriented**
  - Components: objects.
  - Connectors: messages and method invocations.
  - Style invariants: objects are responsible for their internal representation.
  - Advantages: infinite malleability of object internals. System decomposition into sets of interacting agents.
  - Disadvantages: Objects must know identities of servers, side effects in object method invocations.
- **Client server**
  - Components: clients and servers.
  - Servers do not know the number or identities of clients.
  - Clients know the server's identity.
  - Connectors are RPC-based network interaction protocols.
    - Two, three tiers architecture.
- **Layered**
  - Each layer acts as a server (provides services to the layers above) and as a client (consumes data from below). An example of layered SW is an operating system.
    - Pros

        - Increases abstraction levels.
        - Evolvability.
        - Changes in a layer should affect at most the adjacent two layers.
        - Different implementations of layer are allowed as long as interface is preserved.
        - Standardized layer interfaces for libraries and frameworks
    * Cons
        - Not universally applicable.
        - Performances.

- **Mobile code**
  - A data element is dynamically transformed into a data processing component
  - Connectors: network protocols and elements for packaging code and data for transmission.
  - Data elements: representations of code as data; program state; data.
  - Variants: code on demand, remote evaluation and mobile agents.

- **Publish-subscribe**
  - Subscribers register/deregister to receive specific messages or specific content.
  - Publisher broadcast messages to subscribers either synchronously or asynchronously.
  - Components: publishers and subscribers.
  - Connectors: typically a network protocol is required. Content based subscription requires sophisticated connectors.
  - Data elements: subscriptions, notifications, published information.
  - Topology: subscriber connect to publisher either directly or may receive notifications via a network protocol from intermediaries (**broker**).
  - Qualities: with very low-coupling of components.

- **Peer to peer**
  - State and behavior are distributed among peers that can act as either clients or servers.
  - Peers: independent components, having their own state and control thread.
  - Connectors: network protocols, often custom.
  - Data elements: network messages.
  - Topology: network (may have redundant connections between peers), can vary arbitrarily and dynamically.
  - Supports decentralized computing with flow of control and resources distributed among peers.
  - Aims at being highly robust in the face of failure of any given node.
  - Scalable in terms of access to resources and computing power.

- **Rule based**
  - Uses a knowledge base as a container of facts and rules. An inference engine parses the user input and determines whether it is a fact\rule or a query. If it is a fact\rule it adds this entry to the KB, otherwise it queries the last one searching for a solutions.
  - Components: user interface, inference engine, knowledge base.
  - Connectors: components are tightly interconnected, with direct procedure calls and/or shared memory.
  - Data elements: facts and queries.
  - The behavior of the application can be very easily modified through addition or deletion of rules from the KB.

- **Pipe and filters**

  - Components are filters: they transforms input data streams into output data streams possibly increasing the production of outputs.
  - Connectors are pipes: they are conduits for data streams.
  - Filters are independent (they have no shared state) and has no knowledge of up or down-stream filters.
  - Variations
    * Pipelines: linear sequences of filters.
    * Bounded pipes: limited amount of data on a pipe.
    * Typed pipes: data strongly typed.
  - Advantages
    * System behavior is a succession of component behaviors
    * Filter addition, replacement and reuse: it is possible to hook any two filters together.
    * Certain analyses: throughput, latency, deadlock.
    * Concurrent execution.
  - Disadvantages:
    * Batch organization of processing.
    * Interactive applications.
    * Lowest common denominator on data transmission.

- **Blackboard**
  - Components:
    * Blackboard: central data structure.
    * Components operating on the blackboard via well-defined APIs.
  - System control is entirely driven by the blackboard state.

- **Event based**
  - Basically publish-subscribe without content filtering.
  - Independent components asynchronously emit and receive events communicated over event buses.
  - Components: independent, concurrent event generators and\or consumers (no fixed roles as in pub-sub).
  - Connectors: event buses (≥ 1).
  - Data elements: events, data sent as a first-class entity over the event bus.
  - Topology: components communicate with the event buses, not directly to each other.
  - Variants: component communication with the event bus may either be push or pull based.
  - Highly scalable, easy to evolve, effective for highly distributed applications.

7.9 Quality of service - QOS

- **Availability**: a service shall be continuously available to the user. It depends on complexity of the IT architecture, reliability of the individual components, ability to respond quickly and effectively to faults.
  - It is the probability that a system is working properly at a time t.
  - Specified in **nines notation**: 1-nine to 90%, 3-nines availability corresponds to 99.9%, 5-nines to 99.999% and so on.
  - Calculated by modeling the system as an interconnection of parts in series and parallel.
  - If a failure of a part leads to the combination becoming inoperable, the two parts are considered to be operating in **series**.
    * The combined availability of a series is $A = \prod A_i$, where A is the compound availability and A_i the availability of the i-th part.

    - ∗ **A chain is as strong as the weakest link**.
  - If a part leads to the other part taking over the operations of the failed part, the two part are considered to be operating in **parallel**.
    - ∗ The combined availability of a parallel is $A = 1 - \prod(1 - A_i)$ which is 1 minus the probability that all the components in parallel are unavailable.

- **Reliability**: adequate reliability means that the service is available for an agreed period without interruptions. This increases if downtime are prevented. It is determined by the reliability of the single components, the ability of a service or component to operate effectively despite failure of one or more subsystems. Preventive maintenance prevents downtime.
  - It is the probability that a system has always been working properly during a time interval (0,t).

7.10 System life cycle

- **Time of occurrence:** time at which the user becomes aware of the fault.
- **Detection time**: the service provider is informed of the fault.
- **Response time**: time required by the service provider to respond to the user (diagnosis).
- **Repair time**: time required to restore the service or the components that caused the fault.
- **Recovery time:** time required to restore the system.
- **Mean time to repair (MTTR)**: average time between the occurrence of a fault and service recovery, also known as downtime.
- **Mean time between failures** (**MTBF)**: mean time between the recovery from one incident and the occurrence of the next incident, also known as uptime.
- **Mean time between system incidents (MTBSI)**: mean time between the occurrences of two consecutive incidents.
- **Downtime** : calculated as $Downtime = (1 - A) \cdot (365 \cdot 24 \cdot 60)$.

7.11 Fault, failure and errors

- Fault : An incorrect step, process, or data definition in a computer program which causes the program to perform in an unintended or unanticipated manner. These faults lead to a failure when the exact scenario is met.
- Failure: The inability of a system or component to perform its required functions within specified performance requirements .
- Error : A discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.

8 Verification and validation

- **Verification**: did we built the program right?
- **Validation**: did we built the right program?
- **Dynamic V&V technique** (software testing): concerned with exercising and observing product behavior. The system is executed with test data and its operational behavior is observed.
- **Static V&V** techniques are based on the examination of the project documentation, the software, and other related information about requirements and design and not on software execution.

There are three level of testing:

- **Unit testing**: aimed at exercising algorithm correctness, functional specs and structural coverage. It is conducted by the same developers.
- **Integration test**: aimed at exercising interfaces and module interactions.
- **System test**: aimed at exercising global behavior, performance, Hw\Sw integration. It is conducted by independent team or organization.

8.1 Approaches

- **Testing**
  - Experimenting with behavior of the products.
  - Sampling behaviors. The goal is to find counterexamples.
  - Is a dynamic technique.
- **Analysis**
  - Analytic study of properties.
  - Is a static technique.
  - It does not involve the actual execution of software.
  - Two approaches
    - Manual inspection.
    - Automated static analysis.
  - Can be applied at any stage of the development process. Suited for early stages of specification and design.

8.2 Fagan's inspection

- **Moderator**
  - Typically borrowed from another project.
  - Chairs meeting, chooses participants, controls process.
- **Readers, testers**:
  - Read code to group, look for flaws.
- **Author**:
  - Passive participant, answers questions when asked.

The inspection process flows as follows:

- **Planning**: moderator checks entry criteria, choose participants and schedule the meeting.
- **Overview**: provide background education, assign roles.
- **Preparation**.
- **Inspection.**
- **Rework.**
- **Follow-up** (possible re-inspection).

The goal of the meeting is to find as many faults as possible (max 2x2 hour session per day, 150 source liners per hour).
The code is paraphrased line-by-line, the reader tries to reconstruct the intent of the code from its source.
Testers find defects and log them but don't fix them.

- Faults found in inspection are not used in personnel evaluation. Programmer has no incentive to hide faults.
- Faults in testing (after inspection) are used in personnel evaluation. Programmer has incentive to find faults in inspection but not by inserting more.

The pros of this approach are:

- Detailed, formal process, with recording keeping.
- Check-lists; self-improving process.
- Social aspects of process, especially for authors.
- Consideration of whole input space.
- Applies to incomplete programs.

The limitations are:

- Scale: inherently a unit-level technique.
- Non-incremental: what about evolution?

8.3 Data flow analysis

Possible check:

- Is a variable always initialized when used?
- Is a variable assigned and then never used?
- Does a variable always get a new value before being used?

Whenever a variable is declared or a value is assigned to it we say that it is **defined**.

Whenever a we use the value of a variable for operations or checks we say that it is **used**.

Given a program we create the flowchart of it and mark the variables that have been used (**use** set) and defined (**def** set) for every instruction.

Data flow cannot distinguish between paths that can actually be executed and paths that cannot and it cannot always determine whether two names or expression refer to the same object.

8.4 Symbolic state

Values are expression over symbols. Executing statements computes new expressions.

8.5 Path condition

A path condition (pc) is needed to record condition on input data that specified the path followed during symbolic execution.

If the code fragment has preconditions this gives the initial value to path condition, otherwise true is assumed.

8.6 Testing

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

An **execution path** is basically a list of numbers telling which instructions has been executed in order.

Given P (program), I(input domain) and O (output domain), where $P : I \to O$ (may be partial), $OR \subseteq I \times O$:

- $P(i), i \in I$ is **correct** if $< i, P(i) > \in OR$
- P is **correct** if all P(i) are correct for all possible $i \in I$.
- Test case t where $t \in I$
- Test set T where $T \subseteq I$

- Test case is **successful** if P(t) is correct.
- Test set is **successful** if P is correct $\forall t \in T$.

A well done covering of a finite state machine representing the system to test requires:

- State coverage: every state in the model should be visited by at least one test case.
- Transition coverage: every transition between states should be traversed by at least one test case. This is the most commonly used criterion.

8.7 Integration test

An integration fault raises because:

- Inconsistent interpretation of parameters or values (ex: mixed units, meters\yards).
- Violations of value domains, capacity or size limits (buffer overflows).
- Side effects on parameters or resources (conflict on unspecified temporary file).
- Omitted or misunderstood functionality (inconsistent interpretation of web hits).
- Nonfunctional properties (unanticipated performance issues).
- Dynamic mismatches (incompatible polymorphic methods calls).

A **big-bang** integration test is an extreme and desperate approaches in which the system is tested only after integrating all modules at once.

8.7.1 Structural and functional strategies

- **Structural**: modules constructed, integrated and tested based on a hierarchical project structure. The approach is simple and can be mixed together.
  - Top-down uses **stubs** to emulate the lower components of the hierarchy.
  - Bottom-up uses **drivers** to emulate the upper components.
  - Sandwich uses a mixed approach.
- **Functional**: Modules integrated according to application characteristics or features. Require more planning but these techniques are well-suited for larger subsystems.
  - A **thread** is a portion of several modules that together provide a user-visible program feature. The test integrates a functionality and then integrates the remaining ones.
  - **Critical modules** strategy integrates the riskiest modules first and may resemble thread or sandwich process in tactics for flexible build order.

8.8 Dependability verification

Dependability is the extent to which a (critical) system is trusted by its users.

The failure are classified as:

- **Transient**: caused only by some input.
- **Permanent**: caused by all types of input.
- **Recoverable**: the system can restart without external intervention.
- **Unrecoverable**: the system can restart only with the intervention of human operator.
- **Non corruptive**: the failure does not lead to data or system corruption.
- **Corruptive**: the failure leads to the corruption of data and/or system state.

8.8.1 Statistical testing

The goal is to test the software for reliability rather than fault detection.

- Establish the **operational profile** for the system.
  - An operational profile is a set of test data whose frequency matches the actual frequency of these inputs from normal usage of system.
  - Can be generated from real data collected from an existing system or depends on assumptions made about the pattern of usage of a system.
- Construct test data reflecting the operational profile.
- Test the system and observe the number of failures and the times of these failures.
- Compute the reliability after a statistically significant number of failures have been observed.

The operational profile can be used as a guided document in designing user interfaces, to focus on major features for early version of the system and to determine where to put more resources.