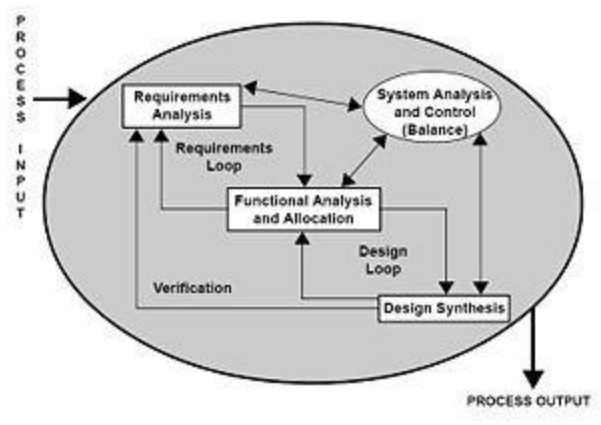
REQUIREMENTS ANALYSIS



Requirements analysis is the first stage in the systems engineering process and software development process.

Requirements analysis in systems engineering and software engineering, encompasses those tasks that go into determining the needs or conditions to meet for a new or altered product, taking account of the possibly conflicting requirements of the various stakeholders, such as beneficiaries or users.

Requirements analysis is critical to the success of a development project. Requirements must be actionable, measurable, testable, related to identified business needs or opportunities, and defined to a level of detail sufficient for system design. Requirements can be functional and non-functional.

Overview

Conceptually, requirements analysis includes three types of activity:

- Eliciting requirements: the task of communicating with customers and users to determine what their requirements are. This is sometimes also called requirements gathering.
- Analyzing requirements: determining whether the stated requirements are unclear, incomplete, ambiguous, or contradictory, and then resolving these issues.
- Recording requirements: Requirements might be documented in various forms, such as natural-language documents, use cases, user stories, or process specifications.

Requirements analysis can be a long and arduous process during which many delicate psychological skills are involved. New systems change the environment and relationships between people, so it is important to identify all the stakeholders, take into account all their needs and ensure they understand the implications of the new systems. Analysts can employ several techniques to elicit the requirements from the customer. Historically, this has included such things as holding interviews, or holding focus groups (more aptly named in this context as requirements workshops) and creating requirements lists. More modern techniques

include prototyping, and use cases. Where necessary, the analyst will employ a combination of these methods to establish the exact requirements of the stakeholders, so that a system that meets the business needs is produced.

Requirements engineering

Systematic requirements analysis is also known as *requirements engineering*. It is sometimes referred to loosely by names such as *requirements gathering*, *requirements capture*, or requirements specification. The term requirements analysis can also be applied specifically to the analysis proper, as opposed to elicitation or documentation of the requirements, for instance.

Requirement engineering according to Laplante (2007) is "a subdiscipline of systems engineering and software engineering that is concerned with determining the goals, functions, and constraints of hardware and software systems." In some life cycle models, the requirement engineering process begins with a feasibility study activity, which leads to a feasibility report. If the feasibility study suggest that the product should be developed, then requirement analysis can begin. If requirement analysis precedes feasibility studies, which may foster outside the box thinking, then feasibility should be determined before requirements are finalized.

Requirements analysis topics

Stakeholder identification

See Stakeholder analysis for a discussion of business uses. Stakeholders (SH) are persons or organizations (legal entities such as companies, standards bodies) which have a valid interest in the system. They may be affected by it either directly or indirectly. A major new emphasis in the 1990s was a focus on the identification of stakeholders. It is increasingly recognized that stakeholders are not limited to the organization employing the analyst. Other stakeholders will include:

- anyone who operates the system (normal and maintenance operators)
- anyone who benefits from the system (functional, political, financial and social beneficiaries)
- anyone involved in purchasing or procuring the system. In a mass-market product organization, product management, marketing and sometimes sales act as surrogate consumers (mass-market customers) to guide development of the product
- organizations which regulate aspects of the system (financial, safety, and other regulators)
- people or organizations opposed to the system (negative stakeholders; see also Misuse case)
- organizations responsible for systems which interface with the system under design
- those organizations who integrate horizontally with the organization for whom the analyst is designing the system

Stakeholder interviews

Stakeholder interviews are a common technique used in requirement analysis. These interviews may reveal requirements not previously envisaged as being within the scope of the project, and requirements may be contradictory. However, each stakeholder will have an idea of their expectation or will have visualized their requirements.

Contract-style requirement lists

One traditional way of documenting requirements has been contract style requirement lists. In a complex system such requirements lists can run to hundreds of pages.

An appropriate metaphor would be an extremely long shopping list. Such lists are very much out of favour in modern analysis; as they have proved spectacularly unsuccessful at achieving their aims; but they are still seen to this day.

Strengths

- Provides a checklist of requirements.
- Provide a contract between the project sponsor(s) and developers.
- For a large system can provide a high level description.

Weaknesses

- Such lists can run to hundreds of pages. It is virtually impossible to read such documents as a whole and have a coherent understanding of the system.
- Such requirements lists abstract all the requirements and so there is little context
    - This abstraction makes it impossible to see how the requirements fit together.
    - This abstraction makes it difficult to identify which are the most important requirements.
    - This abstraction means that the more people who read such requirements the more different visions of the system you get.
    - This abstraction means that it's extremely difficult to be sure that you have the majority of the requirements. Necessarily, these documents speak in generality; but the devil as they say is in the details.
- These lists create a false sense of mutual understanding between the stakeholders and developers.
- These contract style lists give the stakeholders a false sense of security that the developers must achieve certain things. However, due to the nature of these lists, they inevitably miss out crucial requirements which are identified later in the process. Developers can use these discovered requirements to renegotiate the terms and conditions in their favour.

- These requirements lists are no help in system design, since they do not lend themselves to application.

Measurable goals

Best practices take the composed list of requirements merely as clues and repeatedly ask "why?" until the actual business purposes are discovered. Stakeholders and developers can then devise tests to measure what level of each goal has been achieved thus far. Such goals change more slowly than the long list of specific but unmeasured requirements. Once a small set of critical, measured goals has been established, rapid prototyping and short iterative development phases may proceed to deliver actual stakeholder value long before the project is half over.

Prototypes

In the mid-1980s, prototyping was seen as the solution to the requirements analysis problem. Prototypes are Mockups of an application. Mockups allow users to visualize an application that hasn't yet been constructed. Prototypes help users get an idea of what the system will look like, and make it easier for users to make design decisions without waiting for the system to be built. Major improvements in communication between users and developers were often seen with the introduction of prototypes. Early views of applications led to fewer changes later and hence reduced overall costs considerably.

However, over the next decade, while proving a useful technique, prototyping did not solve the requirements problem:

- Managers, once they see a prototype, may have a hard time understanding that the finished design will not be produced for some time.
- Designers often feel compelled to use patched together prototype code in the real system, because they are afraid to 'waste time' starting again.
- Prototypes principally help with design decisions and user interface design. However, they can not tell you what the requirements originally were.
- Designers and end users can focus too much on user interface design and too little on producing a system that serves the business process.
- Prototypes work well for user interfaces, screen layout and screen flow but are not so useful for batch or asynchronous processes which may involve complex database updates and/or calculations.

Prototypes can be flat diagrams (often referred to as Wireframes) or working applications using synthesized functionality. Wireframes are made in a variety of graphic design documents, and often remove all color from the design (i.e. use a greyscale color palette) in instances where the final software is expected to have graphic design applied to it. This helps to prevent confusion over the final visual look and feel of the application.

Use cases

A use case is a technique for documenting the potential requirements of a new system or software change. Each use case provides one or more *scenarios* that convey how the system

should interact with the end user or another system to achieve a specific business goal. Use cases typically avoid technical jargon, preferring instead the language of the end user or domain expert. Use cases are often co-authored by requirements engineers and stakeholders.

Use cases are deceptively simple tools for describing the behavior of software or systems. A use case contains a textual description of all of the ways which the intended users could work with the software or system. Use cases do not describe any internal workings of the system, nor do they explain how that system will be implemented. They simply show the steps that a user follows to perform a task. All the ways that users interact with a system can be described in this manner.

Software requirements specification

A software requirements specification (SRS) is a complete description of the behavior of the system to be developed. It includes a set of use cases that describe all of the interactions that the users will have with the software. Use cases are also known as functional requirements. In addition to use cases, the SRS also contains nonfunctional (or supplementary) requirements. Non-functional requirements are requirements which impose constraints on the design or implementation (such as performance requirements, quality standards, or design constraints).

Recommended approaches for the specification of software requirements are described by IEEE 830-1998. This standard describes possible structures, desirable contents, and qualities of a software requirements specification.

Types of Requirements

Requirements are categorized in several ways. The following are common categorizations of requirements that relate to technical management:

Customer Requirements

> Statements of fact and assumptions that define the expectations of the system in terms of mission objectives, environment, constraints, and measures of effectiveness and suitability (MOE/MOS). The customers are those that perform the eight primary functions of systems engineering, with special emphasis on the operator as the key customer. Operational requirements will define the basic need and, at a minimum, answer the questions posed in the following listing:
>
> - *Operational distribution or deployment*: Where will the system be used?
> - *Mission profile or scenario*: How will the system accomplish its mission objective?
> - *Performance and related parameters*: What are the critical system parameters to accomplish the mission?
> - *Utilization environments*: How are the various system components to be used?
> - *Effectiveness requirements*: How effective or efficient must the system be in performing its mission?

- *Operational life cycle*: How long will the system be in use by the user?
- *Environment*: What environments will the system be expected to operate in an effective manner?

Functional Requirements

Functional requirements explain what has to be done by identifying the necessary task, action or activity that must be accomplished. Functional requirements analysis will be used as the toplevel functions for functional analysis.

Non-functional Requirements

Non-functional requirements are requirements that specify criteria that can be used to judge the operation of a system, rather than specific behaviors.

Performance Requirements

The extent to which a mission or function must be executed; generally measured in terms of quantity, quality, coverage, timeliness or readiness. During requirements analysis, performance (how well does it have to be done) requirements will be interactively developed across all identified functions based on system life cycle factors; and characterized in terms of the degree of certainty in their estimate, the degree of criticality to system success, and their relationship to other requirements.

Design Requirements

The “build to,” “code to,” and “buy to” requirements for products and “how to execute” requirements for processes expressed in technical data packages and technical manuals.

Derived Requirements

Requirements that are implied or transformed from higher-level requirement. For example, a requirement for long range or high speed may result in a design requirement for low weight.

Allocated Requirements

A requirement that is established by dividing or otherwise allocating a high-level requirement into multiple lower-level requirements. Example: A 100-pound item that consists of two subsystems might result in weight requirements of 70 pounds and 30 pounds for the two lower-level items.

Well-known requirements categorization models include FURPS and FURPS+, developed at Hewlett-Packard.

Requirements analysis issues

Stakeholder issues

Steve McConnell, in his book *Rapid Development*, details a number of ways users can inhibit requirements gathering:

- Users do not understand what they want or users don't have a clear idea of their requirements
- Users will not commit to a set of written requirements
- Users insist on new requirements after the cost and schedule have been fixed
- Communication with users is slow
- Users often do not participate in reviews or are incapable of doing so
- Users are technically unsophisticated
- Users do not understand the development process
- Users do not know about present technology

This may lead to the situation where user requirements keep changing even when system or product development has been started.

Engineer/developer issues

Possible problems caused by engineers and developers during requirements analysis are:

- Technical personnel and end users may have different vocabularies. Consequently, they may wrongly believe they are in perfect agreement until the finished product is supplied.
- Engineers and developers may try to make the requirements fit an existing system or model, rather than develop a system specific to the needs of the client.
- Analysis may often be carried out by engineers or programmers, rather than personnel with the people skills and the domain knowledge to understand a client's needs properly.

Attempted solutions

One attempted solution to communications problems has been to employ specialists in business or system analysis.

Techniques introduced in the 1990s like prototyping, Unified Modeling Language (UML), use cases, and Agile software development are also intended as solutions to problems encountered with previous methods.

Also, a new class of application simulation or application definition tools have entered the market. These tools are designed to bridge the communication gap between business users and the IT organization — and also to allow applications to be 'test marketed' before any code is produced. The best of these tools offer:

- electronic whiteboards to sketch application flows and test alternatives
- ability to capture business logic and data needs
- ability to generate high fidelity prototypes that closely imitate the final application

- interactivity
- capability to add contextual requirements and other comments
- ability for remote and distributed users to run and interact with the simulation

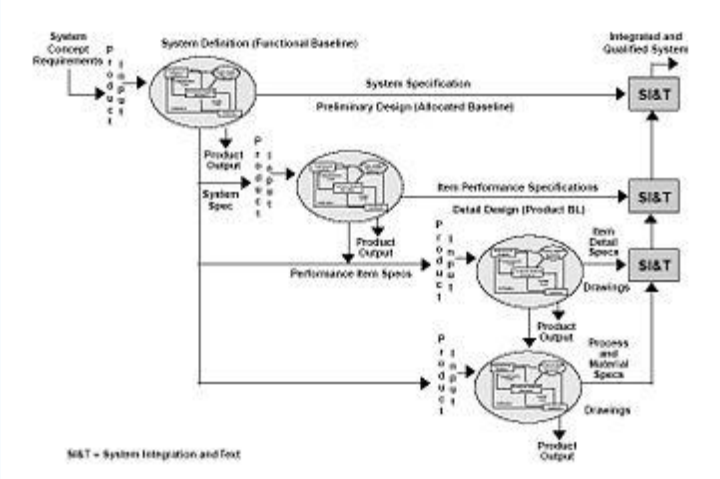
References

1. *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.swebok.org/ch2.html. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

Further reading

- Laplante, Phil (2009). *Requirements Engineering for Software and Systems* (1st ed.). Redmond, WA: CRC Press. ISBN 1-42006-467-3. http://beta.crcpress.com/product/isbn/9781420064674.
- McConnell, Steve (1996). *Rapid Development: Taming Wild Software Schedules* (1st ed.). Redmond, WA: Microsoft Press. ISBN 1-55615-900-5. http://www.stevemcconnell.com/.
- Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
- Andrew Stellman and Jennifer Greene (2005). *Applied Software Project Management*. Cambridge, MA: O'Reilly Media. ISBN 0-596-00948-8. http://www.stellman-greene.com.
- Brian Berenbach, Daniel Paulish, Juergen Katzmeier, Arnold Rudorfer (2009). *Software & Systems Requirements Engineering: In Practice*. New York: McGraw-Hill Professional. ISBN 0-07-1605479. http://www.mhprofessional.com.
- Walter Sobkiw (2008). *Sustainable Development Possible with Creative System Engineering*. New Jersey: CassBeth. ISBN 0615216307. http://books.google.com/books?id=7WJppXs-LzEC.

FUNCTIONAL SPECIFICATION



Systems engineering model of Specification and Levels of Development. During system development a series of specifications are generated to describe the system at different levels of detail. These program unique specifications form the core of the configuration baselines. As shown here, in addition to referring to different levels within the system hierarchy, these baselines are defined at different phases of the design process.

A **functional specification** (also, *functional spec*, *specs*, *functional specifications document (FSD)*, or *Program specification*) in systems engineering and software development is the documentation that describes the requested behavior of an engineering system. The documentation typically describes what is needed by the system user as well as requested properties of inputs and outputs (e.g. of the software system).

Overview

In systems engineering a specification is a document that clearly and accurately describes the essential technical requirements for items, materials, or services including the procedures by which it can be determined that the requirements have been met. Specifications help avoid duplication and inconsistencies, allow for accurate estimates of necessary work and resources, act as a negotiation and reference document for engineering changes, provide documentation of configuration, and allow for consistent communication among those responsible for the eight primary functions of Systems Engineering. They provide a precise idea of the problem to be solved so that they can efficiently design the system and estimate the cost of design alternatives. They provide guidance to testers for verification (qualification) of each technical requirement.

A functional specification does not define the inner workings of the proposed system; it does not include the specification how the system function will be implemented. Instead, it focuses on what various outside agents (people using the program, computer peripherals, or other computers, for example) might "observe" when interacting with the system. A typical functional specification might state the following:

> *When the user clicks the OK button, the dialog is closed and the focus is returned to the main window in the state it was in before this dialog was displayed.*

Such a requirement describes an interaction between an external agent (the user) and the software system. When the user provides input to the system by clicking the OK button, the program responds (or should respond) by closing the dialog window containing the OK button.

It can be *informal*, in which case it can be considered as a blueprint or user manual from a developer point of view, or formal, in which case it has a definite meaning defined in mathematical or programmatic terms. In practice, most successful specifications are written to understand and fine-tune applications that were already well-developed, although safety-critical software systems are often carefully specified prior to application development. Specifications are most important for external interfaces that must remain stable.

Functional specification topics

Purpose

There are many purposes for functional specifications. One of the primary purposes on team projects is to achieve some form of team consensus on what the program is to achieve before making the more time-consuming effort of writing source code and test cases, followed by a period of debugging. Typically, such consensus is reached after one or more reviews by the stakeholders on the project at hand after having negotiated a cost-effective way to achieve the requirements the software needs to fulfill.

Process

In the ordered industrial software engineering life-cycle (waterfall model), functional specification describes what has to be implemented. The next system specification document describes how the functions will be realized using a chosen software environment. In not industrial, prototypical systems development, functional specifications are typically written after or as part of requirements analysis.

When the team agrees that functional specification consensus is reached, the functional spec is typically declared "complete" or "signed off". After this, typically the software development and testing team write source code and test cases using the functional specification as the reference. While testing is performed the behavior of the program is compared against the expected behavior as defined in the functional specification

Types of software development specifications

- Advanced Microcontroller Bus Architecture
- Bit specification
- Design specification
- Diagnostic design specification
- Multiboot Specification
- Product design specification
- Real-time specification for Java

- Software Requirements Specification

See also

- Benchmarking
- Benchmark specification
- Extensible Firmware Interface
- Software development process
- Specification (technical standard)
- Verification and Validation (software)

References

1. *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001

External links

- Writing functional specifications Tutorial

SOFTWARE ARCHITECTURE

The **software architecture** of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships between them. The term also refers to documentation of a system's software architecture. Documenting software architecture facilitates communication between stakeholders, documents early decisions about high-level design, and allows reuse of design components and patterns between projects.

Overview

The field of computer science has come across problems associated with complexity since its formation. Earlier problems of complexity were solved by developers by choosing the right data structures, developing algorithms, and by applying the concept of separation of concerns. Although the term “software architecture” is relatively new to the industry, the fundamental principles of the field have been applied sporadically by software engineering pioneers since the mid 1980s. Early attempts to capture and explain software architecture of a system were imprecise and disorganized, often characterized by a set of box-and-line diagrams. During the 1990’s there was a concentrated effort to define and codify fundamental aspects of the discipline. Initial sets of design patterns, styles, best practices, description languages, and formal logic were developed during that time. The software architecture discipline is centered on the idea of reducing complexity through abstraction and separation of concerns. To date there is still no agreement on the precise definition of the term “software architecture”.

As a maturing discipline with no clear rules on the right way to build a system, designing software architecture is still a mix of art and science. The “art” aspect of software architecture is because a commercial software system supports some aspect of a business or a mission. How a system supports key business drivers is described via scenarios as non-functional requirements of a system, also known as quality attributes, determine how a system will behave. Every system is unique due to the nature of the business drivers it supports, as such the degree of quality attributes exhibited by a system such as fault-tolerance, backward compatibility, extensibility, reliability, maintainability, availability, security, usability, and such other –ilities will vary with each implementation.

to bring a software architecture user's perspective into the software architecture, it can be said that software architecture gives the direction to take steps and do the tasks involved in each such user's speciality area and interest e.g. the stakeholders of software systems, the software developer, the software system operational support group, the software maintenance specialists, the deployer, the tester and also the business end user. In this sense software architecture is really the amalgamation of the multiple perspectives a system always embodies. The fact that those several different perspectives can be put together into a software architecture stands as the vindication of the need and justification of creation of software architecture before the software development in a project attains maturity.

History

The origin of software architecture as a concept was first identified in the research work of Edsgerhttp://en.wikipedia.org/wiki/Edsger_Dijkstra Dijkstra in 1968 and David Parnas in the early 1970s.

These scientists emphasized that the structure of a software system matters and getting the structure right is critical. The study of the field increased in popularity since the early 1990s with research work concentrating on architectural styles (patterns), architecture description languages, architecture documentation, and formal methods.

Research institutions have played a prominent role in furthering software architecture as a discipline. Mary Shaw and David Garlan of Carnegie Mellon wrote a book titled Software Architecture: Perspectives on an Emerging Discipline in 1996, which brought forward the concepts in Software Architecture, such as components, connectors, styles and so on. The University of California, Irvine's Institute for Software Research's efforts in software architecture research is directed primarily in architectural styles, architecture description languages, and dynamic architectures.

The IEEE 1471: ANSI/IEEE 1471-2000: Recommended Practice for Architecture Description of Software-Intensive Systems is the first formal standard in the area of software architecture, and was adopted in 2007 by ISO as ISO/IEC 42010:2007.

Software architecture topics

Architecture description languages

Architecture description languages (ADLs) are used to describe a Software Architecture. Several different ADLs have been developed by different organizations, including AADL (SAE standard), Wright (developed by Carnegie Mellon), Acme (developed by Carnegie Mellon), xADL (developed by UCI), Darwin (developed by Imperial College London), DAOP-ADL (developed by University of Málaga), ByADL (Build Your ADL) developed by University of L'Aquila. Common elements of an ADL are component, connector and configuration.

Views

Software architecture is commonly organized in views, which are analogous to the different types of blueprints made in building architecture. Within the ontology established by ANSI/IEEE 1471-2000, views are instances of viewpoints, where a viewpoint exists to describe the architecture in question from the perspective of a given set of stakeholders and their concerns.

Some possible views (actually, *viewpoints* in the 1471 ontology) are:

- Functional/logic view
- Code/module view
- Development/structural view
- Concurrency/process/thread view
- Physical/deployment view
- User action/feedback view
- Data view

Several languages for describing software architectures have been devised, but no consensus has yet been reached on which symbol-set and view-system should be adopted. The UML was established as a standard "to model systems (and not just software)," and thus applies to views about software architecture.

Architecture frameworks

Frameworks related to the domain of software architecture are:

- 4+1
- RM-ODP (Reference Model of Open Distributed Processing)
- Service-Oriented Modeling Framework (SOMF)

Other architectures such as the Zachman Framework, DODAF, and TOGAF relate to the field of Enterprise architecture.

The distinction from detailed design

Software architecture, also described as strategic design, is an activity concerned with global design constraints, such as programming paradigms, architectural styles, component-based software engineering standards, design principles, and law-governed regularities. Detailed design, also described as tactical design, is an activity concerned with local design constraints, such as design patterns, architectural patterns, programming idioms, and refactorings. According to the Intension/Locality Hypothesis[8], the distinction between architectural and detailed design is defined by the Locality Criterion[8], according to which a statement about software design is non-local (architectural) if and only if a program that satisfies it can be expanded into a program which does not. For example, the Client-Server style is architectural (strategic) because a program that is built by this principle can be expanded into a program which is not client server; for example, by adding peer-to-peer nodes.

Architecture is design but not all design is architectural. In practice, the architect is the one who draws the line between software architecture (architectural design) and detailed design (non-architectural design). There aren't rules or guidelines that fit all cases.

Examples of Architectural Styles / Patterns

There are many common ways of designing computer software modules and their communications, among them:

- Blackboard
- Client-server (2-tier, n-tier, peer-to-peer, Cloud Computing all use this model)
- Database-centric architecture (broad division can be made for programs which have database at its center and applications which don't have to rely on databases, E.g. desktop application programs, utility programs etc.)
- Distributed computing
- Event Driven Architecture

- Front-end and back-end
- Implicit invocation
- Monolithic application
- Peer-to-peer
- Pipes and filters
- Plugin
- Representational State Transfer
- Rule evaluation
- Search-oriented architecture (A pure SOA implements a service for every data access point)
- Service-oriented architecture
- Shared nothing architecture
- Software componentry (strictly module-based, usually object-oriented programming within modules, slightly less monolithic)
- Space based architecture
- Structured (module-based but usually monolithic within modules)
- Three-tier model (An architecture with Presentation, Business Logic and Database tiers)

See also

- Anti-pattern
- Architecture Centric Design Method
- Architecture Tradeoff Analysis Method (ATAM)
- Common layers in an information system logical architecture
- Computer architecture
- Dependency Structure Matrix
- Enterprise architecture
- Process architecture
- Software architect
- Software Architectural Model
- Software design

- Software design pattern
- Software framework
- Software system
- Standard data model
- Systems architect
- Systems architecture
- Systems design
- Technical architecture
- Software Architecture Analysis Method

References

1. Bass, Len; Paul Clements, Rick Kazman (2003). *Software Architecture In Practice, Second Edition*. Boston: Addison-Wesley. pp. 21–24. ISBN 0-321-15495-9.
2. University of Waterloo (2006). "A Very Brief History of Computer Science". http://www.cs.uwaterloo.ca/~shallit/Courses/134/history.html. Retrieved 2006-09-23.
3. IEEE Transactions on Software Engineering (2006). "Introduction to the Special Issue on Software Architecture". http://csdl2.computer.org/persagen/DLAbsToc.jsp?resourcePath=/dl/trans/ts/&toc=comp/trans/ts/1995/04/e4toc.xml&DOI=10.1109/TSE.1995.10003. Retrieved 2006-09-23.
4. SEI (2006). "How do you define Software Architecture?". http://www.sei.cmu.edu/architecture/start/definitions.cfm. Retrieved 2006-09-23.
5. SoftwareArchitectures.com (2006). "Intro to Software Quality Attributes". http://www.softwarearchitectures.com/one/Designing+Architecture/78.aspx. Retrieved 2006-09-23.
6. Clements, Paul; Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, Judith Stafford (2003). *Documenting Software Architectures: Views and Beyond*. Boston: Addison-Wesley. pp. 13–15. ISBN 0-201-70372-6.
7. Amnon H. Eden, Rick Kazman (2003). "Architecture Design Implementation". http://www.eden-study.org/articles/2003/icse03.pdf.

Further reading

- Len Bass, Paul Clements, Rick Kazman: *Software Architecture in Practice, Second Edition*. Addison Wesley, Reading 5/9/2003 ISBN 0-321-15495-9 (This book, now in second edition, eloquently covers the fundamental concepts of the discipline. The theme is centered around achieving quality attributes of a system.)
- Amnon H. Eden, Rick Kazman. *Architecture, Design, Implementation.* On the distinction between architectural design and detailed design.
- Garzás, Javier, and Piattini, Mario. An ontology for micro-architectural design knowledge, IEEE Software Magazine, Volume: 22, Issue: 2, March-April 2005. pp. 28 – 33.

- Tony Shan and Winnie Hua (2006). *Solution Architecting Mechanism*. Proceedings of the 10th IEEE International EDOC Enterprise Computing Conference (EDOC 2006), October 2006, p23-32
- SOMF: Bell, Michael (2008). "Service-Oriented Modeling: Service Analysis, Design, and Architecture". Wiley. http://www.amazon.com/Service-Oriented-Modeling-Service-Analysis-Architecture/dp/0470141115/ref=pd_bbs_2.

External links

- Software architecture vs. software design: The Intension/Locality Hypothesis
- Worldwide Institute of Software Architects (WWISA)
- International Association of Software Architects (IASA)
- SoftwareArchitecturePortal.org — website of IFIP Working Group 2.10 on Software Architecture
- Software Architecture — practical resources for Software Architects
- SoftwareArchitectures.com — independent resource of information on the discipline
- Architectural Patterns
- Software Architecture, chapter 1 of Roy Fielding's REST dissertation
- DiaSpec, an approach and tool to generate a distributed framework from a software architecture
- When Good Architecture Goes Bad
- Software Architecture and Related Concerns, What is Software Architecture? And What Software Architecture ***Is Not***
- Handbook of Software Architecture
- The Spiral Architecture Driven Development - the SDLC based on Spiral model is to reduce a risks of non effective architecture

SOFTWARE DESIGN

Software design is a process of problem-solving and planning for a software solution. After the purpose and specifications of software are determined, software developers will design or employ designers to develop a plan for a solution. It includes low-level component and algorithm implementation issues as well as the architectural view.

Overview

The software requirements analysis (SRA) step of a software development process yields specifications that are used in software engineering. If the software is "semiautomated" or user centered, software design may involve user experience design yielding a story board to help determine those specifications. If the software is completely automated (meaning no user or user interface), a software design may be as simple as a flow chart or text describing a planned sequence of events. There are also semi-standard methods like Unified Modeling Language and Fundamental modeling concepts. In either case some documentation of the plan is usually the product of the design.

A software design may be platform-independent or platform-specific, depending on the availability of the technology called for by the design

Software design topics

Design concepts

The design concepts provide the software designer with a foundation from which more sophisticated methods can be applied. A set of fundamental design concepts has evolved. They are:

- 1.Abstraction - Abstraction is the process or result of generalization by reducing the information content of a concept or an observable phenomenon, typically in order to retain only information which is relevant for a particular purpose.
- 2.Refinement - It is the process of elaboration. A hierarchy is developed by decomposing a macroscopic statement of function in a stepwise fashion until programming language statements are reached. In each step, one or several instructions of a given program are decomposed into more detailed instructions. Abstraction and Refinement are complementary concepts.
- 3.Modularity - Software architecture is divided into components called modules.
- 4.Software Architecture - It refers to the overall structure of the software and the ways in which that structure provides conceptual integrity for a system. A software architecture is the development work product that gives the highest return on investment with respect to quality, schedule and cost.
- 5.Control Hierarchy - A program structure that represent the organization of a program components and implies a hierarchy of control.
- 6.Structural Partitioning - The program structure can be divided both horizontally and vertically. Horizontal partitions define separate branches of modular hierarchy for each

major program function. Vertical partitioning suggests that control and work should be distributed top down in the program structure.

- 7.Data Structure - It is a representation of the logical relationship among individual elements of data.
- 8.Software Procedure - It focuses on the processing of each modules individually
- 9.Information Hiding - Modules should be specified and designed so that information contained within a module is inaccessible to other modules that have no need for such information.

Design considerations

There are many aspects to consider in the design of a piece of software. The importance of each should reflect the goals the software is trying to achieve. Some of these aspects are:

- **Compatibility** - The software is able to operate with other products that are designed for interoperability with another product. For example, a piece of software may be backward-compatible with an older version of itself.
- **Extensibility** - New capabilities can be added to the software without major changes to the underlying architecture.
- **Fault-tolerance** - The software is resistant to and able to recover from component failure.
- **Maintainability** - The software can be restored to a specified condition within a specified period of time. For example, antivirus software may include the ability to periodically receive virus definition updates in order to maintain the software's effectiveness.
- **Modularity** - the resulting software comprises well defined, independent components. That leads to better maintainability. The components could be then implemented and tested in isolation before being integrated to form a desired software system. This allows division of work in a software development project.
- **Packaging** - Printed material such as the box and manuals should match the style designated for the target market and should enhance usability. All compatibility information should be visible on the outside of the package. All components required for use should be included in the package or specified as a requirement on the outside of the package.
- **Reliability** - The software is able to perform a required function under stated conditions for a specified period of time.
- **Reusability** - the modular components designed should capture the essence of the functionality expected out of them and no more or less. This single-minded purpose renders the components reusable wherever there are similar needs in other designs.

- **Robustness** - The software is able to operate under stress or tolerate unpredictable or invalid input. For example, it can be designed with a resilience to low memory conditions.
- **Security** - The software is able to withstand hostile acts and influences.
- **Usability** - The software user interface must be usable for its target user/audience. Default values for the parameters must be chosen so that they are a good choice for the majority of the users. In many cases, online help should be included and also carefully designed.

Modeling language

A modeling language is any artificial language that can be used to express information or knowledge or systems in a structure that is defined by a consistent set of rules. The rules are used for interpretation of the meaning of components in the structure. A modeling language can be graphical or textual. Examples of graphical modelling languages for software design are:

- Business Process Modeling Notation (BPMN) is an example of a Process Modeling language.
- EXPRESS and EXPRESS-G (ISO 10303-11) is an international standard general-purpose data modeling language.
- Extended Enterprise Modeling Language (EEML) is commonly used for business process modeling across a number of layers.
- Flowchart is a schematic representation of an algorithm or a stepwise process,
- Fundamental Modeling Concepts (FMC) modeling language for software-intensive systems.
- IDEF is a family of modeling languages, the most notable of which include IDEF0 for functional modeling, IDEF1X for information modeling, and IDEF5 for modeling ontologies.
- Jackson Structured Programming (JSP) is a method for structured programming based on correspondences between data stream structure and program structure
- LePUS3 is an object-oriented visual Design Description Language and a formal specification language that is suitable primarily for modelling large object-oriented (Java, C++, C#) programs and design patterns.
- Unified Modeling Language (UML) is a general modeling language to describe software both structurally and behaviorally. It has a graphical notation and allow for extension with a Profile (UML).
- Alloy (specification language) is a general purpose specification language for expressing complex structural constraints and behavior in a software system. It provides a concise language based on first-order relational logic.

- Systems Modeling Language (SysML) is a new general-purpose modeling language for systems engineering.

Design patterns

A software designer or architect may identify a design problem which has been solved by others before. A template or pattern describing a solution to a common problem is known as a design pattern. The reuse of such patterns can speed up the software development process, having been tested and proved in the past.

Usage

Software design documentation may be reviewed or presented to allow constraints, specifications and even requirements to be adjusted prior to programming. Redesign may occur after review of a programmed simulation or prototype. It is possible to design software in the process of programming, without a plan or requirement analysis, but for more complex projects this would not be considered a professional approach. A separate design prior to programming allows for multidisciplinary designers and Subject Matter Experts (SMEs) to collaborate with highly-skilled programmers for software that is both useful and technically sound. preparing Functional Design documents, Technical design documents, Technical and System Architecture documents Technical requirements and Specification documents.

Stages and Phases of Design

Deriving a solution which satisfies software requirements. software design stages to be followed problem understanding identify one or more problem,describe solutions abstraction,repeat process for each identifier abstraction.

when design stages complete,phases design will take over which include architectural design,abstract specification,interface design,component design,data structure design and algorithm design.The design process may be modeled as a directed graph made up of entities with attributes which participate in relationship.

Design takes place in overlapping stages,it is artificial to separate it into distinct phases but some separation is usually necessary

Software testing

Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Software testing also provides an objective, independent view of the software to allow the business to appreciate and understand the risks at implementation of the software. Test techniques include, but are not limited to, the process of executing a program or application with the intent of finding software bugs.

Software testing can also be stated as the process of validating and verifying that a software program/application/product:

1. meets the business and technical requirements that guided its design and development;
2. works as expected; and

3. can be implemented with the same characteristics.

Software testing, depending on the testing method employed, can be implemented at any time in the development process. However, most of the test effort occurs after the requirements have been defined and the coding process has been completed. As such, the methodology of the test is governed by the software development methodology adopted.

Different software development models will focus the test effort at different points in the development process. Newer development models, such as Agile, often employ test driven development and place an increased portion of the testing in the hands of the developer, before it reaches a formal team of testers. In a more traditional model, most of the test execution occurs after the requirements have been defined and the coding process has been completed.

Overview

Testing can never completely identify all the defects within software. Instead, it furnishes a *criticism* or *comparison* that compares the state and behavior of the product against oracles—principles or mechanisms by which someone might recognize a problem. These oracles may include (but are not limited to) specifications, contracts, comparable products, past versions of the same product, inferences about intended or expected purpose, user or customer expectations, relevant standards, applicable laws, or other criteria.

Every software product has a target audience. For example, the audience for video game software is completely different from banking software. Therefore, when an organization develops or otherwise invests in a software product, it can assess whether the software product will be acceptable to its end users, its target audience, its purchasers, and other stakeholders. **Software testing** is the process of attempting to make this assessment.

A study conducted by NIST in 2002 reports that software bugs cost the U.S. economy $59.5 billion annually. More than a third of this cost could be avoided if better software testing was performed.

History

The separation of debugging from testing was initially introduced by Glenford J. Myers in 1979. Although his attention was on breakage testing ("a successful test is one that finds a bug") it illustrated the desire of the software engineering community to separate fundamental development activities, such as debugging, from that of verification. Dave Gelperin and William C. Hetzel classified in 1988 the phases and goals in software testing in the following stages:

- Until 1956 - Debugging oriented
- 1957–1978 - Demonstration oriented
- 1979–1982 - Destruction oriented
- 1983–1987 - Evaluation oriented
- 1988–2000 - Prevention oriented

Software testing topics

Scope

A primary purpose for testing is to detect software failures so that defects may be uncovered and corrected. This is a non-trivial pursuit. Testing cannot establish that a product functions properly under all conditions but can only establish that it does not function properly under specific conditions. The scope of software testing often includes examination of code as well as execution of that code in various environments and conditions as well as examining the aspects of code: does it do what it is supposed to do and do what it needs to do. In the current culture of software development, a testing organization may be separate from the development team. There are various roles for testing team members. Information derived from software testing may be used to correct the process by which software is developed.

Functional vs non-functional testing

Functional testing refers to tests that verify a specific action or function of the code. These are usually found in the code requirements documentation, although some development methodologies work from use cases or user stories. Functional tests tend to answer the question of "can the user do this" or "does this particular feature work".

Non-functional testing refers to aspects of the software that may not be related to a specific function or user action, such as scalability or security. Non-functional testing tends to answer such questions as "how many people can log in at once", or "how easy is it to hack this software".

Defects and failures

Not all software defects are caused by coding errors. One common source of expensive defects is caused by requirement gaps, e.g., unrecognized requirements, that result in errors of omission by the program designer. A common source of requirements gaps is non-functional requirements such as testability, scalability, maintainability, usability, performance, and security.

Software faults occur through the following processes. A programmer makes an error (mistake), which results in a defect (fault, bug) in the software source code. If this defect is executed, in certain situations the system will produce wrong results, causing a failure. Not all defects will necessarily result in failures. For example, defects in dead code will never result in failures. A defect can turn into a failure when the environment is changed. Examples of these changes in environment include the software being run on a new hardware platform, alterations in source data or interacting with different software. A single defect may result in a wide range of failure symptoms.

Finding faults early

It is commonly believed that the earlier a defect is found the cheaper it is to fix it. The following table shows the cost of fixing the defect depending on the stage it was found. For example, if a problem in the requirements is found only post-release, then it would cost 10–100 times more to fix than if it had already been found by the requirements review.

		Time detected				
		Requirements	Architecture	Construction	System test	Post-release
Time introduced	Requirements	1×	3×	5-10×	10×	10-100×
	Architecture	-	1×	10×	15×	25-100×
	Construction	-	-	1×	10×	10-25×

Compatibility

A common cause of software failure (real or perceived) is a lack of compatibility with other application software, operating systems (or operating system versions, old or new), or target environments that differ greatly from the original (such as a terminal or GUI application intended to be run on the desktop now being required to become a web application, which must render in a web browser). For example, in the case of a lack of backward compatibility, this can occur because the programmers develop and test software only on the latest version of the target environment, which not all users may be running. This results in the unintended consequence that the latest work may not function on earlier versions of the target environment, or on older hardware that earlier versions of the target environment was capable of using. Sometimes such issues can be fixed by proactively abstracting operating system functionality into a separate program module or library.

Input combinations and preconditions

A very fundamental problem with software testing is that testing under *all* combinations of inputs and preconditions (initial state) is not feasible, even with a simple product. This means that the number of defects in a software product can be very large and defects that occur infrequently are difficult to find in testing. More significantly, non-functional dimensions of quality (how it is supposed to be versus what it is supposed to do)—usability, scalability, performance, compatibility, reliability—can be highly subjective; something that constitutes sufficient value to one person may be intolerable to another.

Static vs. dynamic testing

There are many approaches to software testing. Reviews, walkthroughs, or inspections are considered as static testing, whereas actually executing programmed code with a given set of test cases is referred to as dynamic testing. Static testing can be (and unfortunately in practice often is) omitted. Dynamic testing takes place when the program itself is used for the first time (which is generally considered the beginning of the testing stage). Dynamic testing may begin before the program is 100% complete in order to test particular sections of code (modules or discrete functions). Typical techniques for this are either using stubs/drivers or execution from a debugger environment. For example, spreadsheet programs are, by their very nature, tested to a large extent interactively ("on the fly"), with results displayed immediately after each calculation or text manipulation.

Software verification and validation

Software testing is used in association with verification and validation:

- Verification: Have we built the software right? (i.e., does it match the specification).
- Validation: Have we built the right software? (i.e., is this what the customer wants).

The terms verification and validation are commonly used interchangeably in the industry; it is also common to see these two terms incorrectly defined. According to the IEEE Standard Glossary of Software Engineering Terminology:

> Verification is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.
>
> Validation is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.

The software testing team

Software testing can be done by software testers. Until the 1980s the term "software tester" was used generally, but later it was also seen as a separate profession. Regarding the periods and the different goals in software testing, different roles have been established: manager, test lead, test designer, tester, automation developer, and test administrator.

Software quality assurance (SQA)

Though controversial, software testing may be viewed as an important part of the software quality assurance (SQA) process. In SQA, software process specialists and auditors take a broader view on software and its development. They examine and change the software engineering process itself to reduce the amount of faults that end up in the delivered software: the so-called defect rate.

What constitutes an "acceptable defect rate" depends on the nature of the software. For example, an arcade video game designed to simulate flying an airplane would presumably have a much higher tolerance for defects than mission critical software such as that used to control the functions of an airliner that really is flying!

Although there are close links with SQA, testing departments often exist independently, and there may be no SQA function in some companies.

Software testing is a task intended to detect defects in software by contrasting a computer program's expected results with its actual results for a given set of inputs. By contrast, QA (quality assurance) is the implementation of policies and procedures intended to prevent defects from occurring in the first place.

Testing methods

The box approach

Software testing methods are traditionally divided into and white- and black-box testing. These two approaches are used to describe the point of view that a test engineer takes when designing test cases.

White box testing

White box testing is when the tester has access to the internal data structures and algorithms including the code that implement these.

Types of white box testing

The following types of white box testing exist:

- API testing (application programming interface) - testing of the application using public and private APIs
- Code coverage - creating tests to satisfy some criteria of code coverage (e.g., the test designer can create tests to cause all statements in the program to be executed at least once)
- Fault injection methods - improving the coverage of a test by introducing faults to test code paths
- Mutation testing methods
- Static testing - White box testing includes all static testing

Test coverage

White box testing methods can also be used to evaluate the completeness of a test suite that was created with black box testing methods. This allows the software team to examine parts of a system that are rarely tested and ensures that the most important function points have been tested.

Two common forms of code coverage are:

- *Function coverage*, which reports on functions executed
- *Statement coverage*, which reports on the number of lines executed to complete the test

They both return a code coverage metric, measured as a percentage.

Black box testing

Black box testing treats the software as a "black box"—without any knowledge of internal implementation. Black box testing methods include: equivalence partitioning, boundary value analysis, all-pairs testing, fuzz testing, model-based testing, traceability matrix, exploratory testing and specification-based testing.

Specification-based testing: Specification-based testing aims to test the functionality of software according to the applicable requirements. Thus, the tester inputs data into, and only sees the output from, the test object. This level of testing usually requires thorough test cases to be provided to the tester, who then can simply verify that for a given input, the output value (or behavior), either "is" or "is not" the same as the expected value specified in the test case.

Specification-based testing is necessary, but it is insufficient to guard against certain risks.

Advantages and disadvantages: The black box tester has no "bonds" with the code, and a tester's perception is very simple: a code *must* have bugs. Using the principle, "Ask and you shall receive," black box testers find bugs where programmers do not. *But,* on the other hand, black box testing has been said to be "like a walk in a dark labyrinth without a flashlight," because the tester doesn't know how the software being tested was actually constructed. As a result, there are situations when (1) a tester writes many test cases to check something that could have been tested by only one test case, and/or (2) some parts of the back-end are not tested at all.

Therefore, black box testing has the advantage of "an unaffiliated opinion," on the one hand, and the disadvantage of "blind exploring," on the other.

Grey box testing

Grey box testing (American spelling: **gray box testing**) involves having knowledge of internal data structures and algorithms for purposes of designing the test cases, but testing at the user, or black-box level. Manipulating input data and formatting output do not qualify as grey box, because the input and output are clearly outside of the "black-box" that we are calling the system under test. This distinction is particularly important when conducting integration testing between two modules of code written by two different developers, where only the interfaces are exposed for test. However, modifying a data repository does qualify as grey box, as the user would not normally be able to change the data outside of the system under test. Grey box testing may also include reverse engineering to determine, for instance, boundary values or error messages.

Testing levels

Tests are frequently grouped by where they are added in the software development process, or by the level of specificity of the test.

Unit testing

Unit testing refers to tests that verify the functionality of a specific section of code, usually at the function level. In an object-oriented environment, this is usually at the class level, and the minimal unit tests include the constructors and destructors.

These type of tests are usually written by developers as they work on code (white-box style), to ensure that the specific function is working as expected. One function might have multiple tests, to catch corner cases or other branches in the code. Unit testing alone cannot verify the functionality of a piece of software, but rather is used to assure that the building blocks the software uses work independently of each other.

Unit testing is also called *component testing*.

Integration testing

Integration testing is any type of software testing that seeks to verify the interfaces between components against a software design. Software components may be integrated in an

iterative way or all together ("big bang"). Normally the former is considered a better practice since it allows interface issues to be localised more quickly and fixed.

Integration testing works to expose defects in the interfaces and interaction between integrated components (modules). Progressively larger groups of tested software components corresponding to elements of the architectural design are integrated and tested until the software works as a system.

System testing

System testing tests a completely integrated system to verify that it meets its requirements.

System integration testing

System integration testing verifies that a system is integrated to any external or third party systems defined in the system requirements.

Regression testing

Regression testing focuses on finding defects after a major code change has occurred. Specifically, it seeks to uncover software regressions, or old bugs that have come back. Such regressions occur whenever software functionality that was previously working correctly stops working as intended. Typically, regressions occur as an unintended consequence of program changes, when the newly developed part of the software collides with the previously existing code. Common methods of regression testing include re-running previously run tests and checking whether previously fixed faults have re-emerged. The depth of testing depends on the phase in the release process and the risk of the added features. They can either be complete, for changes added late in the release or deemed to be risky, to very shallow, consisting of positive tests on each feature, if the changes are early in the release or deemed to be of low risk.

Acceptance testing

Acceptance testing can mean one of two things:

1. A smoke test is used as an acceptance test prior to introducing a new build to the main testing process, i.e. before integration or regression.
2. Acceptance testing performed by the customer, often in their lab environment on their own hardware, is known as user acceptance testing (UAT). Acceptance testing may be performed as part of the hand-off process between any two phases of development.

Alpha testing

Alpha testing is simulated or actual operational testing by potential users/customers or an independent test team at the developers' site. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing, before the software goes to beta testing.

Beta testing

Beta testing comes after alpha testing. Versions of the software, known as beta versions, are released to a limited audience outside of the programming team. The software is released to groups of people so that further testing can ensure the product has few faults or bugs. Sometimes, beta versions are made available to the open public to increase the feedback field to a maximal number of future users.

Non-functional testing

Special methods exist to test non-functional aspects of software. In contrast to functional testing, which establishes the correct operation of the software (correct in that it matches the expected behavior defined in the design requirements), non-functional testing verifies that the software functions properly even when it receives invalid or unexpected inputs. Software fault injection, in the form of fuzzing, is an example of non-functional testing. Non-functional testing, especially for software, is designed to establish whether the device under test can tolerate invalid or unexpected inputs, thereby establishing the robustness of input validation routines as well as error-handling routines. Various commercial non-functional testing tools are linked from the software fault injection page; there are also numerous open-source and free software tools available that perform non-functional testing.

Software performance testing and load testing

Performance testing is executed to determine how fast a system or sub-system performs under a particular workload. It can also serve to validate and verify other quality attributes of the system, such as scalability, reliability and resource usage. Load testing is primarily concerned with testing that can continue to operate under a specific load, whether that be large quantities of data or a large number of users. This is generally referred to as software scalability. The related load testing activity of when performed as a non-functional activity is often referred to as *endurance testing*.

Volume testing is a way to test functionality. *Stress testing* is a way to test reliability. *Load testing* is a way to test performance. There is little agreement on what the specific goals of load testing are. The terms load testing, performance testing, reliability testing, and volume testing, are often used interchangeably.

Stability testing

Stability testing checks to see if the software can continuously function well in or above an acceptable period. This activity of non-functional software testing is oftentimes referred to as load (or endurance) testing.

Usability testing

Usability testing is needed to check if the user interface is easy to use and understand.

Security testing

Security testing is essential for software that processes confidential data to prevent system intrusion by hackers.

Internationalization and localization

Internationalization and localization is needed to test these aspects of software, for which a pseudolocalization method can be used. It will verify that the application still works, even after it has been translated into a new language or adapted for a new culture (such as different currencies or time zones).

Destructive testing

Destructive testing attempts to cause the software or a sub-system to fail, in order to test its robustness.

The testing process

Traditional CMMI or waterfall development model

A common practice of software testing is that testing is performed by an independent group of testers after the functionality is developed, before it is shipped to the customer. This practice often results in the testing phase being used as a project buffer to compensate for project delays, thereby compromising the time devoted to testing.

Another practice is to start software testing at the same moment the project starts and it is a continuous process until the project finishes.

Agile or Extreme development model

In counterpoint, some emerging software disciplines such as extreme programming and the agile software development movement, adhere to a "test-driven software development" model. In this process, unit tests are written first, by the software engineers (often with pair programming in the extreme programming methodology). Of course these tests fail initially; as they are expected to. Then as code is written it passes incrementally larger portions of the test suites. The test suites are continuously updated as new failure conditions and corner cases are discovered, and they are integrated with any regression tests that are developed. Unit tests are maintained along with the rest of the software source code and generally integrated into the build process (with inherently interactive tests being relegated to a partially manual build acceptance process). The ultimate goal of this test process is to achieve continuous deployment where software updates can be published to the public frequently.

A sample testing cycle

Although variations exist between organizations, there is a typical cycle for testing. The sample below is common among organizations employing the Waterfall development model.

- **Requirements analysis**: Testing should begin in the requirements phase of the software development life cycle. During the design phase, testers work with developers in determining what aspects of a design are testable and with what parameters those tests work.
- **Test planning**: Test strategy, test plan, testbed creation. Since many activities will be carried out during testing, a plan is needed.
- **Test development**: Test procedures, test scenarios, test cases, test datasets, test scripts to use in testing software.

- **Test execution**: Testers execute the software based on the plans and test documents then report any errors found to the development team.
- **Test reporting**: Once testing is completed, testers generate metrics and make final reports on their test effort and whether or not the software tested is ready for release.
- **Test result analysis**: Or Defect Analysis, is done by the development team usually along with the client, in order to decide what defects should be treated, fixed, rejected (i.e. found software working properly) or deferred to be dealt with later.
- **Defect Retesting**: Once a defect has been dealt with by the development team, it is retested by the testing team. AKA Resolution testing.
- **Regression testing**: It is common to have a small test program built of a subset of tests, for each integration of new, modified, or fixed software, in order to ensure that the latest delivery has not ruined anything, and that the software product as a whole is still working correctly.
- **Test Closure**: Once the test meets the exit criteria, the activities such as capturing the key outputs, lessons learned, results, logs, documents related to the project are archived and used as a reference for future projects.

Automated testing

Many programming groups are relying more and more on automated testing, especially groups that use test-driven development. There are many frameworks to write tests in, and continuous integration software will run tests automatically every time code is checked into a version control system.

While automation cannot reproduce everything that a human can do (and all the strange ways they think of doing it), it can be very useful for regression testing. However, it does require a well-developed test suite of testing scripts in order to be truly useful.

Testing tools

Program testing and fault detection can be aided significantly by testing tools and debuggers. Testing/debug tools include features such as:

- Program monitors, permitting full or partial monitoring of program code including:
    - Instruction set simulator, permitting complete instruction level monitoring and trace facilities
    - Program animation, permitting step-by-step execution and conditional breakpoint at source level or in machine code
    - Code coverage reports
- Formatted dump or symbolic debugging, tools allowing inspection of program variables on error or at chosen points
- Automated functional GUI testing tools are used to repeat system-level tests through the GUI

- Benchmarks, allowing run-time performance comparisons to be made
- Performance analysis (or profiling tools) that can help to highlight hot spots and resource usage

Some of these features may be incorporated into an Integrated Development Environment (IDE).

Measurement in software testing

Usually, quality is constrained to such topics as correctness, completeness, security, but can also include more technical requirements as described under the ISO standard ISO/IEC 9126, such as capability, reliability, efficiency, portability, maintainability, compatibility, and usability.

There are a number of frequently-used software measures, often called *metrics*, which are used to assist in determining the state of the software or the adequacy of the testing.

Testing artifacts

Software testing process can produce several artifacts.

Test plan

> A test specification is called a test plan. The developers are well aware what test plans will be executed and this information is made available to management and the developers. The idea is to make them more cautious when developing their code or making additional changes. Some companies have a higher-level document called a test strategy.

Traceability matrix

> A traceability matrix is a table that correlates requirements or design documents to test documents. It is used to change tests when the source documents are changed, or to verify that the test results are correct.

Test case

> A test case normally consists of a unique identifier, requirement references from a design specification, preconditions, events, a series of steps (also known as actions) to follow, input, output, expected result, and actual result. Clinically defined a test case is an input and an expected result. This can be as pragmatic as 'for condition x your derived result is y', whereas other test cases described in more detail the input scenario and what results might be expected. It can occasionally be a series of steps (but often steps are contained in a separate test procedure that can be exercised against multiple test cases, as a matter of economy) but with one expected result or expected outcome. The optional fields are a test case ID, test step, or order of execution number, related requirement(s), depth, test category, author, and check boxes for whether the test is automatable and has been automated. Larger test cases may also contain prerequisite states or steps, and descriptions. A test case should also contain a place for the actual result. These steps can be stored in a word processor document, spreadsheet, database, or other common repository. In a database system,

you may also be able to see past test results, who generated the results, and what system configuration was used to generate those results. These past results would usually be stored in a separate table.

Test script

The test script is the combination of a test case, test procedure, and test data. Initially the term was derived from the product of work created by automated regression test tools. Today, test scripts can be manual, automated, or a combination of both.

Test suite

The most common term for a collection of test cases is a test suite. The test suite often also contains more detailed instructions or goals for each collection of test cases. It definitely contains a section where the tester identifies the system configuration used during testing. A group of test cases may also contain prerequisite states or steps, and descriptions of the following tests.

Test data

In most cases, multiple sets of values or data are used to test the same functionality of a particular feature. All the test values and changeable environmental components are collected in separate files and stored as test data. It is also useful to provide this data to the client and with the product or a project.

Test harness

The software, tools, samples of data input and output, and configurations are all referred to collectively as a test harness.

Certifications

Several certification programs exist to support the professional aspirations of software testers and quality assurance specialists. No certification currently offered actually requires the applicant to demonstrate the ability to test software. No certification is based on a widely accepted body of knowledge. This has led some to declare that the testing field is not ready for certification. Certification itself cannot measure an individual's productivity, their skill, or practical knowledge, and cannot guarantee their competence, or professionalism as a tester.

Software testing certification types

- *Exam-based*: Formalized exams, which need to be passed; can also be learned by self-study [e.g., for ISTQB or QAI]
- *Education-based*: Instructor-led sessions, where each course has to be passed [e.g., International Institute for Software Testing (IIST)].

Testing certifications

- Certified Associate in Software Testing (CAST) offered by the Quality Assurance Institute (QAI)
- CATe offered by the International Institute for Software Testing

- Certified Manager in Software Testing (CMST) offered by the Quality Assurance Institute (QAI)
- Certified Software Tester (CSTE) offered by the Quality Assurance Institute (QAI)
- Certified Software Test Professional (CSTP) offered by the International Institute for Software Testing
- CSTP (TM) (Australian Version) offered by K. J. Ross & Associates
- ISEB offered by the Information Systems Examinations Board
- ISTQB Certified Tester, Foundation Level (CTFL) offered by the International Software Testing Qualification Board
- ISTQB Certified Tester, Advanced Level (CTAL) offered by the International Software Testing Qualification Board

Quality assurance certifications

- CMSQ offered by the *Quality Assurance Institute* (QAI).
- CSQA offered by the *Quality Assurance Institute* (QAI)
- CSQE offered by the American Society for Quality (ASQ)
- CQIA offered by the American Society for Quality (ASQ)

Controversy

Some of the major software testing controversies include:

What constitutes responsible software testing?

Members of the "context-driven" school of testing believe that there are no "best practices" of testing, but rather that testing is a set of skills that allow the tester to select or invent testing practices to suit each unique situation.

Agile vs. traditional

Should testers learn to work under conditions of uncertainty and constant change or should they aim at process "maturity"? The agile testing movement has received growing popularity since 2006 mainly in commercial circles, whereas government and military software providers are slow to embrace this methodology in favour of traditional test-last models (e.g. in the Waterfall model).

Exploratory test vs. scripted

Should tests be designed at the same time as they are executed or should they be designed beforehand?

Manual testing vs. automated

Some writers believe that test automation is so expensive relative to its value that it should be used sparingly. More in particular, test-driven development states that developers should write unit-tests of the XUnit type before coding the functionality. The tests then can be considered as a way to capture and implement the requirements.

Software design vs. software implementation

Should testing be carried out only at the end or throughout the whole process?

Who watches the watchmen?

The idea is that any form of observation is also an interaction—the act of testing can also affect that which is being tested.

References

1. Exploratory Testing], Cem Kaner, Florida Institute of Technology, *Quality Assurance Institute Worldwide Annual Software Testing Conference*, Orlando, FL, November 2006
2. ^ Leitner, A., Ciupa, I., Oriol, M., Meyer, B., Fiva, A., "Contract Driven Development = Test Driven Development - Writing Test Cases", Proceedings of ESEC/FSE'07: European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering 2007, (Dubrovnik, Croatia), September 2007
3. Software errors cost U.S. economy $59.5 billion annually, NIST report
4. Kolawa, Adam; Huizinga, Dorota (2007). *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Press. p. 41–43. ISBN 0470042125. http://www.wiley.com/WileyCDA/WileyTitle/productCd-0470042125.html.
5. Kolawa, Adam; Huizinga, Dorota (2007). *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Press. p. 86. ISBN 0470042125. http://www.wiley.com/WileyCDA/WileyTitle/productCd-0470042125.html.
6. Section 1.1.2, Certified Tester Foundation Level Syllabus, International Software Testing Qualifications Board
7. Kaner, Cem; James Bach, Bret Pettichord (2001). *Lessons Learned in Software Testing: A Context-Driven Approach*. Wiley. p. 4. ISBN 0-471-08112-4.
8. McConnell, Steve (2004). *Code Complete* (2nd ed.). Microsoft Press. pp. 960. ISBN 0-7356-1967-0.
9. Principle 2, Section 1.3, Certified Tester Foundation Level Syllabus, International Software Testing Qualifications Board
10. Savenkov, Roman (2008). *How to Become a Software Tester*. Roman Savenkov Consulting. p. 159. ISBN 978-0-615-23372-7.
11. Black, Rex (December 2008). *Advanced Software Testing- Vol. 2: Guide to the ISTQB Advanced Certification as an Advanced Test Manager*. Santa Barbara: Rocky Nook Publisher. ISBN 1933952369.
12. Quality Assurance Institute
13. International Institute for Software Testing
14. "ISTQB". http://www.istqb.org/.
15. "ISTQB in the U.S.". http://www.astqb.org/.

16. American Society for Quality
17. context-driven-testing.com
18. Article on taking agile traits without the agile method.
19. "We're all part of the story" by David Strom, July 1, 2009
20. IEEE article about differences in adoption of agile trends between experienced managers vs. young students of the Project Management Institute. See also Agile adoption study from 2007
21. Agile software development practices slowly entering the military
22. IEEE article on Exploratory vs. Non Exploratory testing

SOFTWARE DEPLOYMENT

Software deployment is all of the activities that make a software system available for use.

The general deployment process consists of several interrelated activities with possible transitions between them. These activities can occur at the producer site or at the consumer site or both. Because every software system is unique, the precise processes or procedures within each activity can hardly be defined. Therefore, "deployment" should be interpreted as a *general process* that has to be customized according to specific requirements or characteristics. A brief description of each activity will be presented later.

Deployment activities

Release

The release activity follows from the completed development process. It includes all the operations to prepare a system for assembly and transfer to the customer site. Therefore, it must determine the resources required to operate at the customer site and collect information for carrying out subsequent activities of deployment process.

Install and activate

Activation is the activity of starting up the executable component of software. For simple system, it involves establishing some form of command for execution. For complex systems, it should make all the supporting systems ready to use.

In larger software deployments, the working copy of the software might be installed on a production server in a production environment. Other versions of the deployed software may be installed in a test environment, development environment and disaster recovery environment.

Deactivate

Deactivation is the inverse of activation, and refers to shutting down any executing components of a system. Deactivation is often required to perform other deployment activities, e.g., a software system may need to be deactivated before an update can be performed. The practice of removing infrequently used or obsolete systems from service is often referred to as application retirement or application decommissioning.

Adapt

The adaptation activity is also a process to modify a software system that has been previously installed. It differs from updating in that adaptations are initiated by local events such as changing the environment of customer site, while updating is mostly started from remote software producer.

Update

The update process replaces an earlier version of all or part of a software system with a newer release.

Built-In

Mechanisms for installing updates are built into some software systems. Automation of these update processes ranges from fully automatic to user initiated and controlled. Norton Internet Security is an example of a system with a semi-automatic method for retrieving and installing updates to both the antivirus definitions and other components of the system. Other software products provide query mechanisms for determining when updates are available.

Version tracking

Version tracking systems help the user find and install updates to software systems installed on PCs and local networks.

- Web based version tracking systems notify the user when updates are available for software systems installed on a local system. For example: VersionTracker Pro checks software versions on a user's computer and then queries its database to see if any updates are available.
- Local version tracking system notifies the user when updates are available for software systems installed on a local system. For example: Software Catalog stores version and other information for each software package installed on a local system. One click of a button launches a browser window to the upgrade web page for the application, including auto-filling of the user name and password for sites that require a login.
- Browser based version tracking systems notify the user when updates are available for software packages installed on a local system. For example: wfx-Versions is a Firefox extension which helps the user find the current version number of any program listed on the web.

Uninstall

Uninstallation is the inverse of installation. It is a remove of a system that is no longer required. It also involves some reconfiguration of other software systems in order to remove the uninstalled system’s files and dependencies. This is not to be confused with the term "deinstall" which is not actually a word.

Retire

Ultimately, a software system is marked as obsolete and support by the producers is withdrawn. It is the end of the life cycle of a software product.

- Application lifecycle management
- Product lifecycle management
- Systems management
- System deployment
- Software release

External links

- Standardization efforts
  - Solution Installation Schema Submission request to W3C
  - OASIS Solution Deployment Descriptor TC
  - OMG Specification for Deployment and Configuration of Component-based Distributed Applications (OMG D&C)
  - JSR 88: Java EE Application Deployment

SOFTWARE MAINTENANCE

Software maintenance in software engineering is the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.

Overview

ISO/IEC 14764:2006 describes the six software maintenance processes as:

1. The implementation processes contains software preparation and transition activities, such as the conception and creation of the maintenance plan, the preparation for handling problems identified during development, and the follow-up on product configuration management.
2. The problem and modification analysis process, which is executed once the application has become the responsibility of the maintenance group. The maintenance programmer must analyze each request, confirm it (by reproducing the situation) and check its validity, investigate it and propose a solution, document the request and the solution proposal, and, finally, obtain all the required authorizations to apply the modifications.
3. The process considering the implementation of the modification itself.
4. The process acceptance of the modification, by confirming the modified work with the individual who submitted the request in order to make sure the modification provided a solution.
5. The migration process (platform migration, for example) is exceptional, and is not part of daily maintenance tasks. If the software must be ported to another platform without any change in functionality, this process will be used and a maintenance project team is likely to be assigned to this task.
6. Finally, the last maintenance process, also an event which does not occur on a daily basis, is the retirement of a piece of software.

There are a number of processes, activities and practices that are unique to maintainers, for example:

- Transition: a controlled and coordinated sequence of activities during which a system is transferred progressively from the developer to the maintainer;
- Service Level Agreements (SLAs) and specialized (domain-specific) maintenance contracts negotiated by maintainers;
- Modification Request and Problem Report Help Desk: a problem-handling process used by maintainers to prioritize, documents and route the requests they receive;
- Modification Request acceptance/rejection: modification request work over a certain size/effort/complexity may be rejected by maintainers and rerouted to a developer.

A common perception of maintenance is that it is merely fixing bugs. However, studies and surveys over the years have indicated that the majority, over 80%, of the maintenance effort is used for non-corrective actions (Pigosky 1997). This perception is perpetuated by users submitting problem reports that in reality are functionality enhancements to the system.

Software maintenance and evolution of systems was first addressed by Meir M. Lehman in 1969. Over a period of twenty years, his research led to the formulation of eight Laws of Evolution (Lehman 1997). Key findings of his research include that maintenance is really evolutionary developments and that maintenance decisions are aided by understanding what happens to systems (and software) over time.

Lehman demonstrated that systems continue to evolve over time. As they evolve, they grow more complex unless some action such as code refactoring is taken to reduce the complexity.

The key software maintenance issues are both managerial and technical. Key management issues are: alignment with customer priorities, staffing, which organization does maintenance, estimating costs. Key technical issues are: limited understanding, impact analysis, testing, maintainability measurement.

Categories of maintenance in ISO/IEC 14764

E.B. Swanson initially identified three categories of maintenance: corrective, adaptive, and perfective. These have since been updated and ISO/IEC 14764 presents:

- Corrective maintenance: Reactive modification of a software product performed after delivery to correct discovered problems.
- Adaptive maintenance: Modification of a software product performed after delivery to keep a software product usable in a changed or changing environment.
- Perfective maintenance: Modification of a software product after delivery to improve performance or maintainability.
- Preventive maintenance: Modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults.

There is also a notion of pre-delivery/pre-release maintenance which is all the good things you do to lower the total cost of ownership of the software. Things like compliance with coding standards that includes software maintainability goals. The management of coupling and coheasion of the software. The attainment of software supportability goals (SAE JA1004, JA1005 and JA1006 for example). Note also that some academic institutions are carrying out research to quantify the cost to ongoing software maintenance due to the lack of resources such as design documents and system/software comprehension training and resources (multiply costs by approx. 1.5-2.0 where there is no design data available.).

Note also that microsoft, the main culprits in the "we don't need no skinking documentation" camp, are spending millions on research to work out how NOT to put bugs into software - and it sounds like it's get requirements right, design and document the code etc - ground breaking stuff (if only people would read the IEEE/ISO standards).

See also

- Software development
- Computer software
- Software Engineering
- Software evolution

References

1. ISO/IEC 14764:2006 Software Engineering — Software Life Cycle Processes — Maintenance

Further reading

- Gopalaswamy Ramesh; Ramesh Bhattiprolu (2006). *Software maintenance : effective practices for geographically distributed environments*. New Delhi: Tata McGraw-Hill. ISBN 9780070483453.
- Grubb, Penny; Takang, Armstrong (2003). *Software Maintenance*. New Jersey: World Scientific Publishing. ISBN 9789812384256.
- Lehman, M.M.; Belady, L.A. (1985). *Program evolution : processes of software change*. London: Academic Press Inc. ISBN 0-12-442441-4.
- Page-Jones, Meilir (1980). *The Practical Guide to Structured Systems Design*. New York: Yourdon Press. ISBN 0-917072-17-0.