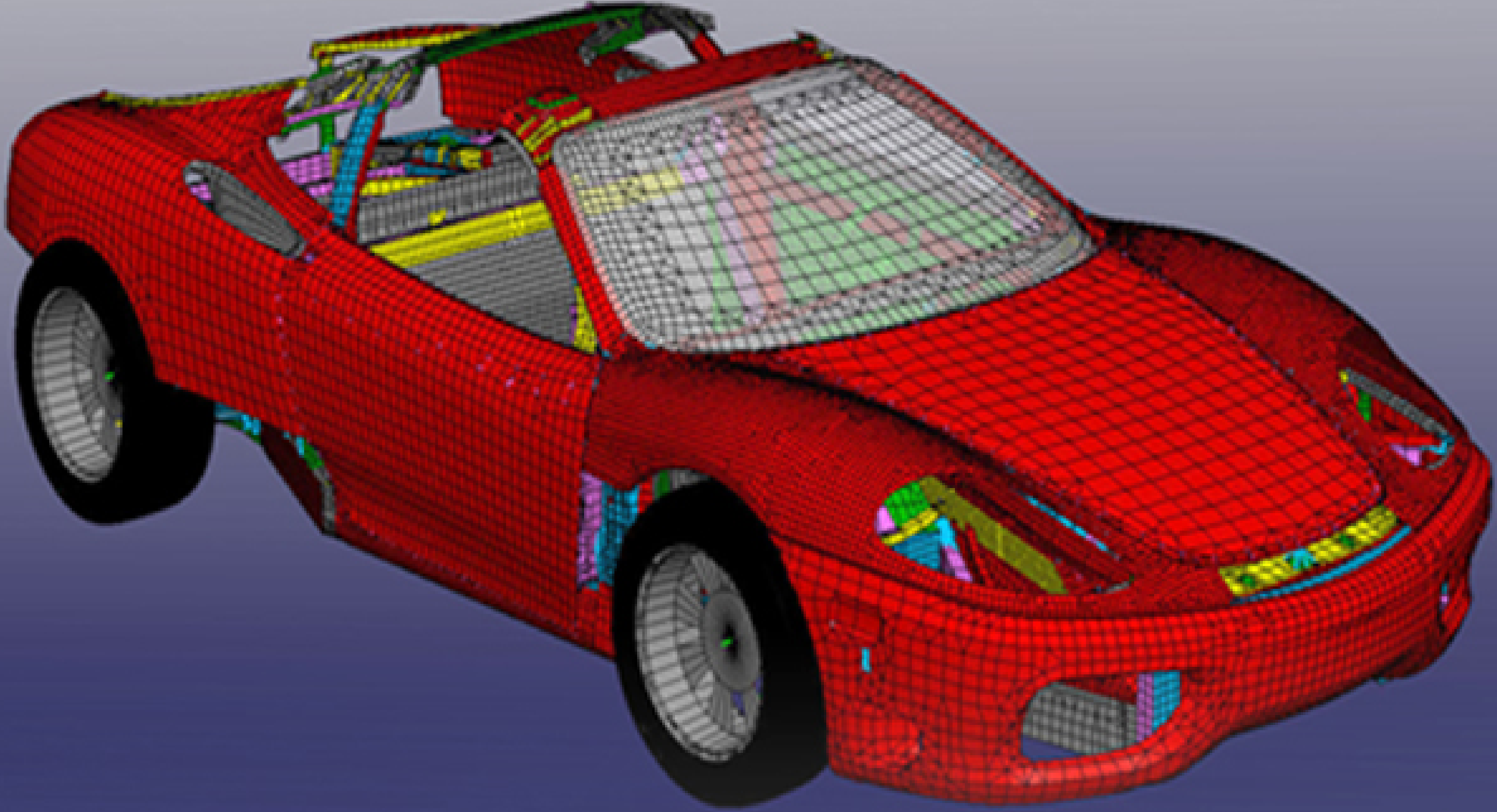
Handbook of

Software Engineering and Development



Milton Cash

Jamie Huskey

First Edition, 2012

ISBN 978-81-323-0929-1

Published by:
Academic Studio
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter 1

Introduction to Software Engineering

The Airbus A380 uses a substantial amount of software to create a "paperless" cockpit.

Software engineering (**SE**) is a profession dedicated to designing, implementing, and modifying software so that it is of higher quality, more affordable, maintainable, and faster to build. It is a "systematic approach to the analysis, design, assessment, implementation, test, maintenance and re-engineering of a software by applying engineering to the software". The term *software engineering* first appeared in the 1968 NATO Software Engineering Conference, and was meant to provoke thought regarding the perceived "software crisis" at the time. Since the field is still relatively young compared to its sister fields of engineering, there is still much debate around what *software engineering* actually is, and if it conforms to the classical definition of engineering. The IEEE Computer Society's *Software Engineering Body of Knowledge* defines "software engineering" as the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software. It is the application of Engineering to software because it integrates significant mathematics, computer science and practices whose origins are in Engineering.

Software development, a much used and more generic term, does not necessarily subsume the engineering paradigm. Although it is questionable what impact it has had on actual software development over the last more than 40 years, the field's future looks bright according to Money Magazine and Salary.com, who rated "software engineering" as the best job in the United States in 2006.

Employment

In 2004, the U. S. Bureau of Labor Statistics counted 760,840 software engineers holding jobs in the U.S.; in the same time period there were some 1.4 million practitioners employed in the U.S. in all other engineering disciplines combined. Due to its relative newness as a field of study, formal education in software engineering is often taught as part of a computer science curriculum, and many software engineers hold computer science degrees.

Many software engineers work as employees or contractors. Software engineers work with businesses, government agencies (civilian or military), and non-profit organizations. Some software engineers work for themselves as freelancers. Some organizations have specialists to perform each of the tasks in the software development process. Other organizations require software engineers to do many or all of them. In large projects, people may specialize in only one role. In small projects, people may fill several or all roles at the same time. Specializations include: in industry (analysts, architects, developers, testers, technical support, middleware analysts, managers) and in academia (educators, researchers).

Certification

The Software Engineering Institute offers certification on specific topics like Security, Process improvement and Software architecture. Apple, IBM, Microsoft and other companies also sponsor their own certification examinations. Many IT certification programs are oriented toward specific technologies, and managed by the vendors of these technologies. These certification programs are tailored to the institutions that would employ people who use these technologies.

Broader certification of general software engineering skills is available through various professional societies. As of 2006, the IEEE had certified over 575 software professionals as a Certified Software Development Professional (CSDP). In 2008 they added an entry-level certification known as the Certified Software Development Associate (CSDA). In the U.K. the British Computer Society has developed a legally recognized professional certification called *Chartered IT Professional (CITP)*, available to fully qualified Members (*MBCS*). In Canada the Canadian Information Processing Society has developed a legally recognized professional certification called *Information Systems Professional (ISP)*. The ACM had a professional certification program in the early 1980s, which was discontinued due to lack of interest. The ACM examined the possibility of professional certification of software engineers in the late 1990s, but eventually decided that such certification was inappropriate for the professional industrial practice of software engineering.

Impact of globalization

The initial impact of outsourcing, and the relatively lower cost of international human resources in developing third world countries led to the dot com bubble burst of the 1990s. This had a negative impact on many aspects of the software engineering profession. For example, some students in the developed world avoid education related to software engineering because of the fear of offshore outsourcing (importing software products or services from other countries) and of being displaced by foreign visa workers. Although statistics do not currently show a threat to software engineering itself; a related career, computer programming does appear to have been affected. Nevertheless, the ability to smartly leverage offshore and near-shore resources in an efficient fashion has improved the overall operational capability of many organizations. When Europeans are leaving work, Asians are just arriving to work. When Asians are leaving work, Europeans are arriving to work. This provides a continuous ability to have human oversight on business-critical processes 24 hours per day, without paying overtime compensation or disrupting key human resource sleep patterns.

Chapter 2

History of Software Engineering

In the **history of software engineering** the software engineering has evolved steadily from its founding days in the 1940s until today in the 2010s. Applications have evolved continuously. The ongoing goal to improve technologies and practices, seeks to improve the productivity of practitioners and the quality of applications to users.

Overview

There are a number of areas where the evolution of software engineering is notable:

- Emergence as a profession: By the early 1980s, software engineering had already emerged as a bona fide profession, to stand beside computer science and traditional engineering.
- Role of women: In the 1940s, 1950s, and 1960s, men often filled the more prestigious and better paying hardware engineering roles, but often delegated the writing of software to women. Grace Hopper, Jamie Fenton and many other unsung women filled many programming jobs during the first several decades of software engineering. Today, many fewer women work in software engineering than in other professions, a situation whose cause is not clearly identified. It is often attributed to sexual discrimination, cyberculture or bias in education. Many academic and professional organizations consider this situation unbalanced are trying hard to solve it.
- Processes: Processes have become a big part of software engineering and are hailed for their potential to improve software and sharply criticized for their potential to constrict programmers.
- Cost of hardware: The relative cost of software versus hardware has changed substantially over the last 50 years. When mainframes were expensive and required large support staffs, the few organizations buying them also had the resources to fund large, expensive custom software engineering projects. Computers are now much more numerous and much more powerful, which has several effects on software. The larger market can support large projects to create commercial off the shelf software, as done by companies such as Microsoft. The cheap machines allow each programmer to have a terminal capable of fairly rapid compilation. The programs in question can use techniques such as garbage collection, which make them easier and faster for the programmer to write. On the other hand, many fewer organizations are interested in employing programmers

for large custom software projects, instead using commercial off the shelf software as much as possible.

The Pioneering Era

The most important development was that new computers were coming out almost every year or two, rendering existing ones obsolete. Software people had to rewrite all their programs to run on these new machines. Programmers did not have computers on their desks and had to go to the "machine room". Jobs were run by signing up for machine time or by operational staff. Jobs were run by putting punched cards for input into the machine's card reader and waiting for results to come back on the printer.

The field was so new that the idea of management by schedule was non-existent. Making predictions of a project's completion date was almost impossible. Computer hardware was application-specific. Scientific and business tasks needed different machines. Due to the need to frequently translate old software to meet the needs of new machines, high-order languages like FORTRAN, COBOL, and ALGOL were developed. Hardware vendors gave away systems software for free as hardware could not be sold without software. A few companies sold the service of building custom software but no software companies were selling packaged software.

The notion of reuse flourished. As software was free, user organizations commonly gave it away. Groups like IBM's scientific user group SHARE offered catalogs of reusable components. Academia did not yet teach the principles of computer science. Modular programming and data abstraction were already being used in programming.

1945 to 1965: The origins

The term *software engineering* first appeared in the late 1950s and early 1960s. Programmers have always known about civil, electrical, and computer engineering and debated what engineering might mean for software.

The NATO Science Committee sponsored two conferences on software engineering in 1968 (Garmisch, Germany) and 1969, which gave the field its initial boost. Many believe these conferences marked the official start of the profession of *software engineering*.

1965 to 1985: The software crisis

Software engineering was spurred by the so-called *software crisis* of the 1960s, 1970s, and 1980s, which identified many of the problems of software development. Many software projects ran over budget and schedule. Some projects caused property damage. A few projects caused loss of life. The software crisis was originally defined in terms of productivity, but evolved to emphasize quality. Some used the term *software crisis* to refer to their inability to hire enough qualified programmers.

- Cost and Budget Overruns: The OS/360 operating system was a classic example. This decade-long project from the 1960s eventually produced one of the most complex software systems at the time. OS/360 was one of the first large (1000 programmers) software projects. Fred Brooks claims in *The Mythical Man Month* that he made a multi-million dollar mistake of not developing a coherent architecture before starting development.
- Property Damage: Software defects can cause property damage. Poor software security allows hackers to steal identities, costing time, money, and reputations.
- Life and Death: Software defects can kill. Some embedded systems used in radiotherapy machines failed so catastrophically that they administered lethal doses of radiation to patients. The most famous of these failures is the *Therac 25* incident.

Peter G. Neumann has kept a contemporary list of software problems and disasters. The software crisis has been slowly fizzling out, because it is unrealistic to remain in crisis mode for more than 20 years. SEs are accepting that the problems of SE are truly difficult and only hard work over many decades can solve them.

1985 to 1989: No silver bullet

For decades, solving the software crisis was paramount to researchers and companies producing software tools. Seemingly, they trumpeted every new technology and practice from the 1970s to the 1990s as a *silver bullet* to solve the software crisis. Tools, discipline, formal methods, process, and professionalism were touted as silver bullets:

- Tools: Especially emphasized were tools: Structured programming, object-oriented programming, CASE tools, Ada, documentation, and standards were touted as silver bullets.
- Discipline: Some pundits argued that the software crisis was due to the lack of discipline of programmers.
- Formal methods: Some believed that if formal engineering methodologies would be applied to software development, then production of software would become as predictable an industry as other branches of engineering. They advocated proving all programs correct.
- Process: Many advocated the use of defined processes and methodologies like the Capability Maturity Model.
- Professionalism: This led to work on a code of ethics, licenses, and professionalism.

In 1986, Fred Brooks published the *No Silver Bullet* article, arguing that no individual technology or practice would ever make a 10-fold improvement in productivity within 10 years.

Debate about silver bullets raged over the following decade. Advocates for Ada, components, and processes continued arguing for years that their favorite technology would be a silver bullet. Skeptics disagreed. Eventually, almost everyone accepted that no silver bullet would ever be found. Yet, claims about *silver bullets* pop up now and again, even today.

Some interpret *no silver bullet* to mean that software engineering failed. However, with further reading, Brooks goes on to say, “We will surely make substantial progress over the next 40 years; an order of magnitude over 40 years is hardly magical...”.

The search for a single key to success never worked. All known technologies and practices have only made incremental improvements to productivity and quality. Yet, there are no silver bullets for any other profession, either. Others interpret *no silver bullet* as proof that software engineering has finally matured and recognized that projects succeed due to hard work.

However, it could also be said that there are, in fact, a range of *silver bullets* today, including lightweight methodologies, spreadsheet calculators, customized browsers, in-site search engines, database report generators, integrated design-test coding-editors with memory/differences/undo, and specialty shops that generate niche software, such as information websites, at a fraction of the cost of totally customized website development. Nevertheless, the field of software engineering appears too complex and diverse for a single "silver bullet" to improve most issues, and each issue accounts for only a small portion of all software problems.

1990 to 1999: Prominence of the Internet

The rise of the Internet led to very rapid growth in the demand for international information display/e-mail systems on the World Wide Web. Programmers were required to handle illustrations, maps, photographs, and other images, plus simple animation, at a rate never before seen, with few well-known methods to optimize image display/storage (such as the use of thumbnail images).

The growth of browser usage, running on the HTML language, changed the way in which information-display and retrieval was organized. The widespread network connections led to the growth and prevention of international computer viruses on MS Windows computers, and the vast proliferation of spam e-mail became a major design issue in e-mail systems, flooding communication channels and requiring semi-automated pre-screening. Keyword-search systems evolved into web-based search engines, and many software systems had to be re-designed, for international searching, depending on Search Engine Optimization (SEO) techniques. Human natural-language translation systems were needed to attempt to translate the information flow in multiple foreign languages, with many software systems being designed for multi-language usage, based on design concepts from human translators. Typical computer-user bases went from hundreds, or thousands of users, to, often, many-millions of international users.

2000 to Present: Lightweight Methodologies

With the expanding demand for software in many smaller organizations, the need for inexpensive software solutions led to the growth of simpler, faster methodologies that developed running software, from requirements to deployment, quicker & easier. The use of rapid-prototyping evolved to entire *lightweight methodologies*, such as Extreme Programming (XP), which attempted to simplify many areas of software engineering, including requirements gathering and reliability testing for the growing, vast number of small software systems. Very large software systems still used heavily-documented methodologies, with many volumes in the documentation set; however, smaller systems had a simpler, faster alternative approach to managing the development and maintenance of software calculations and algorithms, information storage/retrieval and display.

Current trends in software engineering

Software engineering is a young discipline, and is still developing. The directions in which software engineering is developing include:

Aspect-oriented programming

In computing, **aspect-oriented programming** (**AOP**) is a programming paradigm which isolates secondary or supporting functions from the main program's business logic. It aims to increase modularity by allowing the separation of cross-cutting concerns, forming a basis for aspect-oriented software development.

AOP includes programming methods and tools that support the modularization of concerns at the level of the source code, while "aspect-oriented software development" refers to a whole engineering discipline.

Overview

Aspect-oriented programming entails breaking down program logic into distinct parts (so-called *concerns*, cohesive areas of functionality). All programming paradigms support some level of grouping and encapsulation of concerns into separate, independent entities by providing abstractions (e.g. procedures, modules, classes, methods) that can be used for implementing, abstracting and composing these concerns. But some concerns defy these forms of implementation and are called *crosscutting concerns* because they "cut across" multiple abstractions in a program.

Logging exemplifies a crosscutting concern because a logging strategy necessarily affects every single logged part of the system. Logging thereby *crosscuts* all logged classes and methods.

All AOP implementations have some crosscutting expressions that encapsulate each concern in one place. The difference between implementations lies in the power, safety, and usability of the constructs provided. For example, interceptors that specify the methods to intercept express a limited form of crosscutting, without much support for type-safety or debugging. AspectJ has a number of such expressions and encapsulates them in a special class, an aspect. For example, an aspect can alter the behavior of the base code (the non-aspect part of a program) by applying advice (additional behavior) at various join points (points in a program) specified in a quantification or query called a pointcut (that detects whether a given join point matches). An aspect can also make binary-compatible structural changes to other classes, like adding members or parents.

History

AOP as such has a number of antecedents: the Visitor Design Pattern, CLOS MOP, and others. Gregor Kiczales and colleagues at Xerox PARC developed AspectJ (perhaps the most popular general-purpose AOP package) and made it available in 2001. IBM's research team emphasized the continuity of the practice of modularizing concerns with past programming practice, and in 2001 offered the more powerful (but less usable) Hyper/J and Concern Manipulation Environment, which have not seen wide usage. EmacsLisp changelog added AOP related code in version 19.28. The examples here use AspectJ as it is the most widely known of AOP languages.

Motivation and basic concepts

Typically, an aspect is *scattered* or *tangled* as code, making it harder to understand and maintain. It is scattered by virtue of the function (such as logging) being spread over a number of unrelated functions that might use *its* function, possibly in entirely unrelated systems, different source languages, etc. That means to change logging can require modifying all affected modules. Aspects become tangled not only with the mainline function of the systems in which they are expressed but also with each other. That means changing one concern entails understanding all the tangled concerns or having some means by which the effect of changes can be inferred.

For example, consider a banking application with a conceptually very simple method for transferring an amount from one account to another:

```
void transfer(Account fromAcc, Account toAcc, int amount) throws
Exception {

  if (fromAcc.getBalance() < amount) {
    throw new InsufficientFundsException();
  }

  fromAcc.withdraw(amount);
  toAcc.deposit(amount);
}
```

However, this transfer method overlooks certain considerations that a deployed application would require. It lacks security checks to verify that the current user has the authorization to perform this operation. A database transaction should encapsulate the operation in order to prevent accidental data loss. For diagnostics, the operation should be logged to the system log. And so on. A simplified version with all those new concerns would look somewhat like this:

```
void transfer(Account fromAcc, Account toAcc, int amount, User user,
Logger logger)
  throws Exception {
  logger.info("transferring money...");
  if (! checkUserPermission(user)){
    logger.info("User has no permission.");
    throw new UnauthorizedUserException();
  }
  if (fromAcc.getBalance() < amount) {
    logger.info("Insufficient Funds, sorry");
    throw new InsufficientFundsException();
  }

  fromAcc.withdraw(amount);
  toAcc.deposit(amount);

  //get database connection

  //save transactions

  logger.info("Successful transaction.");
}
```

In the previous example other interests have become *tangled* with the basic functionality (sometimes called the *business logic concern*). Transactions, security, and logging all exemplify *cross-cutting concerns*.

Also consider what happens if we suddenly need to change (for example) the security considerations for the application. In the program's current version, security-related operations appear *scattered* across numerous methods, and such a change would require a major effort.

Therefore, cross-cutting concerns do not get encapsulated in their own modules. This increases system complexity and makes software evolution considerably more difficult.

AOP attempts to solve this problem by allowing the programmer to express cross-cutting concerns in stand-alone modules called *aspects*. Aspects can contain advice (code joined to specified points in the program) and inter-type declarations (structural members added to other classes). For example, a security module can include advice that performs a security check before accessing a bank account. The pointcut defines the times (join points) when one can access a bank account, and the code in the advice body defines how the security check is implemented. That way, both the check and the places can be maintained in one place. Further, a good pointcut can anticipate later program changes, so if another developer creates a new method to access the bank account, the advice will apply to the new method when it executes.

So for the above example implementing logging in an aspect:

```
aspect Logger {

        void Bank.transfer(Account fromAcc, Account toAcc, int amount,
User user, Logger logger)  {
                logger.info("transferring money...");
        }

        void Bank.getMoneyBack(User user, int transactionId, Logger
logger)  {
                logger.info("User requested money back");
        }

        // other crosscutting code...
}
```

One can think of AOP as a debugging tool or as a user-level tool. Advice should be reserved for the cases where you cannot get the function changed (user level) or do not want to change the function in production code (debugging).

Join point models

The advice-related component of an aspect-oriented language defines a join point model (JPM). A JPM defines three things:

1. When the advice can run. These are called *join points* because they are points in a running program where additional behavior can be usefully joined. A join point needs to be addressable and understandable by an ordinary programmer to be useful. It should also be stable across inconsequential program changes in order for an aspect to be stable across such changes. Many AOP implementations support method executions and field references as join points.
2. A way to specify (or *quantify*) join points, called *pointcuts*. Pointcuts determine whether a given join point matches. Most useful pointcut languages use a syntax like the base language (for example, AspectJ uses Java signatures) and allow reuse through naming and combination.
3. A means of specifying code to run at a join point. AspectJ calls this *advice*, and can run it before, after, and around join points. Some implementations also support things like defining a method in an aspect on another class.

Join-point models can be compared based on the join points exposed, how join points are specified, the operations permitted at the join points, and the structural enhancements that can be expressed.

AspectJ's join-point model

- The join points in AspectJ include method or constructor call or execution, the initialization of a class or object, field read and write access, exception handlers, etc. They do not include loops, super calls, throws clauses, multiple statements, etc.

- Pointcuts are specified by combinations of *primitive pointcut designators* (PCDs).

  "Kinded" PCDs match a particular kind of join point (e.g., method execution) and tend to take as input a Java-like signature. One such pointcut looks like this:

```
execution(* set*(*))
```

  This pointcut matches a method-execution join point, if the method name starts with "`set`" and there is exactly one argument of any type.

"Dynamic" PCDs check runtime types and bind variables. For example

```
this(Point)
```

  This pointcut matches when the currently-executing object is an instance of class `Point`. Note that the unqualified name of a class can be used via Java's normal type lookup.

"Scope" PCDs limit the lexical scope of the join point. For example:

```
within(com.company.*)
```

  This pointcut matches any join point in any type in the `com.company` package. The `*` is one form of the wildcards that can be used to match many things with one signature.

Pointcuts can be composed and named for reuse. For example

```
pointcut set() : execution(* set*(*) ) && this(Point) &&
within(com.company.*);
```

  This pointcut matches a method-execution join point, if the method name starts with "`set`" and `this` is an instance of type `Point` in the `com.company` package. It can be referred to using the name "`set()`".

- Advice specifies to run at (before, after, or around) a join point (specified with a pointcut) certain code (specified like code in a method). The AOP runtime invokes Advice automatically when the pointcut matches the join point. For example:

```
after() : set() {
   Display.update();
}
```

  This effectively specifies: "if the *`set()`* pointcut matches the join point, run the code `Display.update()` after the join point completes."

Other potential join point models

There are other kinds of JPMs. All advice languages can be defined in terms of their JPM. For example, a hypothetical aspect language for UML may have the following JPM:

- Join points are all model elements.
- Pointcuts are some boolean expression combining the model elements.
- The means of affect at these points are a visualization of all the matched join points.

Inter-type declarations

Inter-type declarations provide a way to express crosscutting concerns affecting the structure of modules. Also known as *open classes*, this enables programmers to declare in one place members or parents of another class, typically in order to combine all the code related to a concern in one aspect. For example, if a programmer implemented the crosscutting display-update concern using visitors instead, an inter-type declaration using the visitor pattern might look like this in AspectJ:

```
aspect DisplayUpdate {
  void Point.acceptVisitor(Visitor v) {
    v.visit(this);
  }
  // other crosscutting code...
}
```

This code snippet adds the `acceptVisitor` method to the `Point` class.

It is a requirement that any structural additions be compatible with the original class, so that clients of the existing class continue to operate, unless the AOP implementation can expect to control all clients at all times.

Implementation

AOP programs can affect other programs in two different ways, depending on the underlying languages and environments:

1. a combined program is produced, valid in the original language and indistinguishable from an ordinary program to the ultimate interpreter
2. the ultimate interpreter or environment is updated to understand and implement AOP features.

The difficulty of changing environments means most implementations produce compatible combination programs through a process known as *weaving* - a special case of program transformation. An aspect weaver reads the aspect-oriented code and generates appropriate object-oriented code with the aspects integrated. The same AOP language can be implemented through a variety of weaving methods, so the semantics of a language should never be understood in terms of the weaving implementation. Only the speed of an implementation and its ease of deployment are affected by which method of combination is used.

Systems can implement source-level weaving using preprocessors (as C++ was implemented originally in CFront) that require access to program source files. However, Java's well-defined binary form enables bytecode weavers to work with any Java program in .class-file form. Bytecode weavers can be deployed during the build process or, if the weave model is per-class, during class loading. AspectJ started with source-level weaving in 2001, delivered a per-class bytecode weaver in 2002, and offered advanced load-time support after the integration of AspectWerkz in 2005.

Any solution that combines programs at runtime has to provide views that segregate them properly to maintain the programmer's segregated model. Java's bytecode support for multiple source files enables any debugger to step through a properly woven .class file in a source editor. However, some third-party decompilers cannot process woven code because they expect code produced by Javac rather than all supported bytecode forms.

Deploy-time weaving offers another approach. This basically implies post-processing, but rather than patching the generated code, this weaving approach *subclasses* existing classes so that the modifications are introduced by method-overriding. The existing classes remain untouched, even at runtime, and all existing tools (debuggers, profilers, etc.) can be used during development. A similar approach has already proven itself in the implementation of many Java EE application servers, such as IBM's WebSphere.

Terminology

Standard terminology used in Aspect-oriented programming may include:

- **Cross-cutting concerns**: Even though most classes in an OO model will perform a single, specific function, they often share common, secondary requirements with other classes. For example, we may want to add logging to classes within the data-access layer and also to classes in the UI layer whenever a thread enters or exits a method. Even though each class has a very different primary functionality, the code needed to perform the secondary functionality is often identical.
- **Advice**: This is the additional code that you want to apply to your existing model. In our example, this is the logging code that we want to apply whenever the thread enters or exits a method.
- **Pointcut**: This is the term given to the point of execution in the application at which cross-cutting concern needs to be applied. In our example, a pointcut is reached when the thread enters a method, and another pointcut is reached when the thread exits the method.
- **Aspect**: The combination of the pointcut and the advice is termed an aspect. In the example above, we add a logging aspect to our application by defining a pointcut and giving the correct advice.

Comparison to other programming paradigms

Aspects emerged out of object-oriented programming and computational reflection. AOP languages have functionality similar to, but more restricted than metaobject protocols. Aspects relate closely to programming concepts like subjects, mixins, and delegation. Other ways to use aspect-oriented programming paradigms include Composition Filters and the hyperslices approach. Since at least the 1970s, developers have been using forms of interception and dispatch-patching that resemble some of the implementation methods for AOP, but these never had the semantics that the crosscutting specifications were written in one place.

Designers have considered alternative ways to achieve separation of code, such as C#'s partial types, but such approaches lack a quantification mechanism that allows reaching several join points of the code with one declarative statement.

Adoption issues

Programmers need to be able to read code and understand what is happening in order to prevent errors. Even with proper education, understanding crosscutting concerns can be difficult without proper support for visualizing both static structure and the dynamic flow of a program. As of 2010, IDEs have begun to support the visualizing of crosscutting concerns, as well as aspect code assist and refactoring.

Given the power of AOP, if a programmer makes a logical mistake in expressing crosscutting, it can lead to widespread program failure. Conversely, another programmer may change the join points in a program – e.g., by renaming or moving methods – in ways that the aspect writer did not anticipate, with unintended consequences. One advantage of modularizing crosscutting concerns is enabling one programmer to affect the entire system easily; as a result, such problems present as a conflict over responsibility between two or more developers for a given failure. However, the solution for these problems can be much easier in the presence of AOP, since only the aspect need be changed, whereas the corresponding problems without AOP can be much more spread out.

Agile

Agile software development guides software development projects that evolve rapidly with changing expectations and competitive markets. Proponents of this method believe that heavy, document-driven processes (like TickIT, CMM and ISO 9000) are fading in importance. Some people believe that companies and agencies export many of the jobs that can be guided by heavy-weight processes. Related concepts include Extreme Programming, Scrum, and Lean software development.

Experimental

Experimental software engineering is a branch of software engineering interested in devising experiments on software, in collecting data from the experiments, and in devising laws and theories from this data. Proponents of this method advocate that the nature of software is such that we can advance the knowledge on software through experiments only.

Model-driven

Model Driven Design develops textual and graphical models as primary design artifacts. Development tools are available that use model transformation and code generation to generate well-organized code fragments that serve as a basis for producing complete applications.

Software Product Lines

Software Product Lines is a systematic way to produce *families* of software systems, instead of creating a succession of completely individual products. This method emphasizes extensive, systematic, formal code reuse, to try to industrialize the software development process.

The Future of Software Engineering conference (FOSE), held at ICSE 2000, documented the state of the art of SE in 2000 and listed many problems to be solved over the next decade. The FOSE tracks at the ICSE 2000 and the ICSE 2007 conferences also help identify the state of the art in software engineering.

Software engineering today

The profession is trying to define its boundary and content. The Software Engineering Body of Knowledge SWEBOK has been tabled as an ISO standard during 2006 (ISO/IEC TR 19759).

In 2006, Money Magazine and Salary.com rated software engineering as the best job in America in terms of growth, pay, stress levels, flexibility in hours and working environment, creativity, and how easy it is to enter and advance in the field.

Prominent figures in the history of software engineering

- Charles Bachman (born 1924) is particularly known for his work in the area of databases.
- Laszlo Belady (born 1928) the editor-in-chief of the IEEE Transactions on Software Engineering in the 1980s
- Fred Brooks (born 1931)) best-known for managing the development of OS/360.
- Peter Chen, known for the development of entity-relationship modeling.
- Edsger Dijkstra (1930-2002) developed the framework for proper programming.
- David Parnas (born 1941) developed the concept of information hiding in modular programming.

Chapter 3

Software Engineer

A **software engineer** is an engineer who applies the principles of software engineering to the design, development, testing, and evaluation of the software and systems that make computers or anything containing software, such as computer chips, work.

Overview

Prior to the mid-1990s, software practitioners called themselves *programmers* or *developers,* regardless of their actual jobs. Many people prefer to call themselves *software developer* and *programmer*, because most widely agree what these terms mean, while *software engineer* is still being debated. A prominent computing scientist, E. W. Dijkstra, wrote in a paper that the coining of the term *software engineer* was not a useful term since it was an inappropriate analogy, "The existence of the mere term has been the base of a number of extremely shallow --and false-- analogies, which just confuse the issue...Computers are such exceptional gadgets that there is good reason to assume that most analogies with other disciplines are too shallow to be of any positive value, are even so shallow that they are only confusing."

The term *programmer* has often been used as a pejorative term to refer to those without the tools, skills, education, or ethics to write good quality software. In response, many practitioners called themselves *software engineers* to escape the stigma attached to the word *programmer*. In many companies, the titles *programmer* and *software developer* were changed to *software engineer*, for many categories of programmers.

These terms cause confusion, because some denied any differences (arguing that everyone does essentially the same thing with software) while others use the terms to create a difference (because the terms mean completely different jobs).

A state of the art

In 2004, Keith Chapple U. S. Bureau of Labor Statistics counted 760,840 software engineers holding jobs in the U.S.; in the same period there were some 1.4 million practitioners employed in the U.S. in all other engineering disciplines combined. The label software engineer is used very liberally in the corporate world. Very few of the practicing software engineers actually hold Engineering degrees from accredited universities. In fact, according to the Association for Computing Machinery, "most people who now function in the U.S. as serious software engineers have degrees in computer science, not in software engineering".

Regulatory classification

The U.S. Bureau of Labor Statistics classifies *computer software engineers* as a subcategory of "computer specialists", along with occupations such as computer scientist, programmer, and network administrator. The BLS classifies all other engineering disciplines, including computer hardware engineers, as "engineers".

The U.K. has seen the alignment of the Information Technology Professional and the Engineering Professionals.

Software engineering in Canada has seen some contests in the courts over the use of the title "Software Engineer". The Canadian Council of Professional Engineers (C.C.P.E. or "Engineers Canada") will not grant a "Professional Engineer" status/license to anyone who has not completed a recognized academic engineering program. Engineers qualified outside Canada are similarly unable to obtain a "Professional Engineer" license. Since 2001, the Canadian Engineering Accreditation Board has accredited several university programs in software engineering, allowing graduates to apply for a professional engineering licence once the other prerequisites are obtained, although this does nothing to help IT professionals using the title with degrees in other fields (such as computer science).

Some of the United States of America regulate the use of terms such as "computer engineer" and even "software engineer". These states include at least Texas and Florida.

Education

About half of all practitioners today have computer science degrees. A small, but growing, number of practitioners have software engineering degrees. In 1987 Imperial College London introduced the first three-year software engineering Bachelor's degree in the UK and the world; in the following year the University of Sheffield established a similar programme. In 1996, Rochester Institute of Technology established the first software engineering Bachelor's degree program in the United States, however, it did not obtain ABET until 2003, the same time as Clarkson University, Milwaukee School of Engineering and Mississippi State University obtained theirs. In 1997 PSG College of Technology in Coimbatore, India was the first to start a five-year integrated Master of Science degree in Software Engineering.

Since then, software engineering undergraduate degrees have been established at many universities. A standard international curriculum for undergraduate software engineering degrees was recently defined by the CCSE. As of 2004, in the U.S., about 50 universities offer software engineering degrees, which teach both computer science and engineering principles and practices. The first software engineering Master's degree was established at Seattle University in 1979. Since then graduate software engineering degrees have been made available from many more universities. Likewise in Canada, the Canadian Engineering Accreditation Board (CEAB) of the Canadian Council of Professional Engineers has recognized several software engineering programs.

In 1998, the US Naval Postgraduate School (NPS) established the first doctorate program in Software Engineering in the world. Additionally, many online advanced degrees in Software Engineering have appeared such as the Master of Science in Software Engineering (MSE) degree offered through the Computer Science and Engineering Department at California State University, Fullerton. Steve McConnell opines that because most universities teach computer science rather than software engineering, there is a shortage of true software engineers. ETS University and UQAM were mandated by IEEE to develop the SoftWare Engineering BOdy of Knowledge SWEBOK, which has become an ISO standard describing the body of knowledge covered by a software engineer.

Other degrees

In business, some software engineering practitioners have MIS degrees. In embedded systems, some have electrical or computer engineering degrees, because embedded software often requires a detailed understanding of hardware. In medical software, practitioners may have medical informatics, general medical, or biology degrees.

Some practitioners have mathematics, science, engineering, or technology degrees. Some have philosophy (logic in particular) or other non-technical degrees. And, others have no degrees. For instance, Barry Boehm earned degrees in mathematics.

Profession

Employment

Most software engineers work as employees or contractors. Software engineers work with businesses, government agencies (civilian or military), and non-profit organizations. Some software engineers work for themselves as freelancers. Some organizations have specialists to perform each of the tasks in the software development process. Other organizations required software engineers to do many or all of them. In large projects, people may specialize in only one role. In small projects, people may fill several or all roles at the same time. Specializations include: in industry (analysts, architects, developers, testers, technical support, managers) and in academia (educators, researchers).

There is considerable debate over the future employment prospects for Software Engineers and other IT Professionals. For example, an online futures market called the Future of IT Jobs in America attempts to answer whether there will be more IT jobs, including software engineers, in 2012 than there were in 2002.

Certification

Professional certification of software engineers is a contentious issue. Some see it as a tool to improve professional practice.

Most successful certification programs in the software industry are oriented toward specific technologies, and are managed by the vendors of these technologies. These certification programs are tailored to the institutions that would employ people who use these technologies.

The ACM had a professional certification program in the early 1980s, which was discontinued due to lack of interest. As of 2006, the IEEE had certified over 575 software professionals. In Canada the Canadian Information Processing Society has developed a legally recognized professional certification called Information Systems Professional (ISP).

Impact of globalization

Many students in the developed world have avoided degrees related to software engineering because of the fear of offshore outsourcing (importing software products or services from other countries) and of being displaced by foreign visa workers. Although government statistics do not currently show a threat to software engineering itself; a related career, computer programming does appear to have been affected. Often one is expected to start out as a computer programmer before being promoted to software engineer. Thus, the career path to software engineering may be rough, especially during recessions.

Some career counselors suggest a student also focus on "people skills" and business skills rather than purely technical skills because such "soft skills" are allegedly more difficult to offshore. It is the quasi-management aspects of software engineering that appear to be what has kept it from being impacted by globalization.

Prizes

There are several prizes in the field of software engineering:

- The CODiE awards is a yearly award issued by the Software and Information Industry Association for excellence in software development the software industry.
- Jolt Awards are awards in the software industry.
- Stevens Award is a software engineering award given in memory of Wayne Stevens.

Debates within software engineering

Controversies over the term Engineer

Some people believe that *software engineering* implies a certain level of academic training, professional discipline, and adherence to formal processes that often are not applied in cases of software development. A common analogy is that working in construction does not make one a civil engineer, and so writing code does not make one a software engineer. It is disputed by some - in particular by the Canadian Professional Engineers Ontario (PEO) body, that the field is mature enough to warrant the title "engineering". The PEO's position was that "software engineering" was not an appropriate name for the field since those who practiced in the field and called themselves "software engineers" were not properly licensed professional engineers, and that they should therefore not be allowed to use the name.

The status of software engineering

The word *engineering* within the term software engineering causes a lot of confusion because it is a shallow analogy.

The wrangling over the status of software engineering (between traditional engineers and computer scientists) can be interpreted as a fight over control of the word "engineering".

Traditional engineers (especially civil engineers and the NSPE) claim that they have special rights over the term *engineering*, and for anyone else to use it requires their approval. In the mid-1990s, the NSPE sued to prevent anyone from using the job title *software engineering*. The NSPE won their lawsuit in 48 states. However, SE practitioners, educators, and researchers ignored the lawsuits and called themselves software engineers anyway. The U.S. Bureau of Labor Statistics uses the term software engineer, too. The term engineering is much older than any regulatory body, so many believe that traditional engineers have few rights to control the term. As things stand at 2007, however, even the NSPE appears to have softened its stance towards software engineering and following the heels of several overseas precedents, is investigating a possibility of licensing software engineers in consultation with IEEE, NCEES and other groups "for the protection of the public health safety and welfare".

In Canada, the use of the words 'engineer' and 'engineering' are controlled in each province by self-regulating professional engineering organizations, often aligned with geologists and geophysicists, and tasked with enforcement of the governing legislation. The intent is that any individual holding themselves out as an engineer (or geologist or geophysicist) has been verified to have been educated to a certain accredited level, and their professional practice is subject to a code of ethics and peer scrutiny.

In New Zealand, IPENZ, the professional engineering organization entrusted by the New Zealand government with legal power to license and regulate chartered engineers (CPEng), recognizes software engineering as a legitimate branch of professional engineering and accepts application of software engineers to obtain chartered status provided he or she has a tertiary degree of approved subjects. Software Engineering is included but Computer Science is normally not.

Chapter 4

Software Requirements and Software Design

Software requirements

Software requirements is a sub-field of Software engineering that deals with the elicitation, analysis, specification, and validation of requirements for software .

The software requirement specification (SRS) document generates all necessary requirements for project development. To derive the requirements we need to have clear and thorough understanding of the products to be developed. This is prepared after detailed communications with project team and the customer.

An SRS clearly defines the following:

1. External interfaces of the system: They identify the information which is to flow 'from and to' the system.
2. Functional and nonfunctional requirements of the systems. They stand for the finding of run-time requirements.
3. Design constraints

The SRS outline is given below:

1. **Introduction**
    1. purpose
    2. scope
    3. definitions,acronyms and abbreviations
    4. references
    5. overview
2. **Overall Descriptions**
    1. product perspective
    2. product functions
    3. user characteristics
    4. constraints
    5. assumptions and dependencies

3. **Specific requirements**
    1. External interfaces
    2. functional requirements
    3. performance requirements
    4. logical database requirements
    5. Design constraints
    6. Software system attributes
    7. organising the specific requirements
    8. additional comments
4. **supporting information**
    1. table of contents and index
    2. appendixes

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 good software architecture will yield a good return on investment with respect to the desired outcome of the project, e.g. in terms of performance, 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 software is able to add further features and modification with slight or no modification.
- **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.

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 allows 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

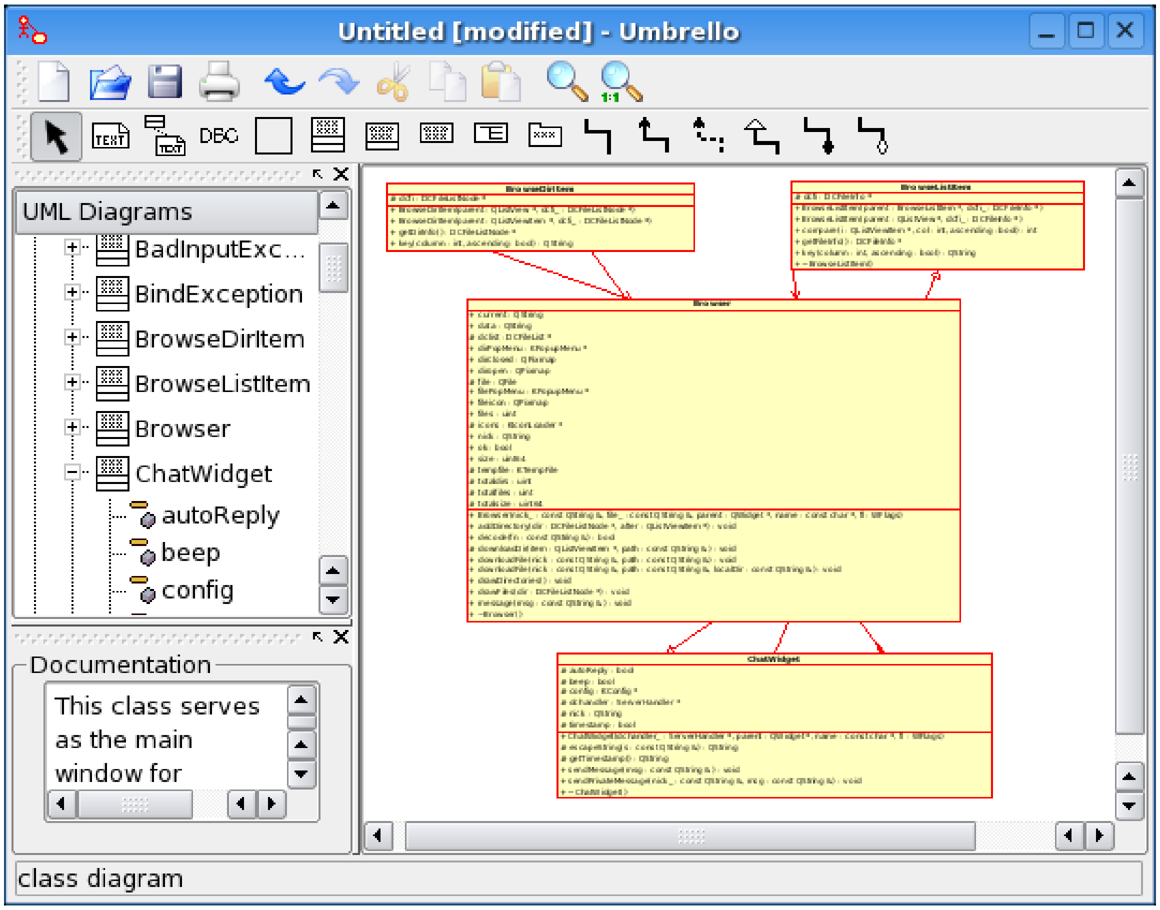
A software designer or architect may identify a design problem which has been solved by others before. A template or pattern describing a solution to a common problem is known as a design pattern. The reuse of such patterns can speed up the software development process, having been tested and proved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

Chapter 5

Computer-Aided Software Engineering



Example of a CASE tool.

Computer-aided software engineering (**CASE**) is the scientific application of a set of tools and methods to a software system which is meant to result in high-quality, defect-free, and maintainable software products. It also refers to methods for the development of information systems together with automated tools that can be used in the software development process.

Overview

The term "Computer-aided software engineering" (CASE) can refer to the software used for the automated development of systems software, i.e., computer code. The CASE functions include analysis, design, and programming. CASE tools automate methods for designing, documenting, and producing structured computer code in the desired programming language.

Computer Aided Software Engineering (CASE) is the name given to the software used to support the software process activities such as requirement engineering, design, program development and testing. Therefore, CASE tools include design editors, data dictionaries, compilers, debuggers, system building tools, etc.

Computer Aided Software Engineering (CASE) refers to the methods dedicated to an engineering discipline for the development of information system using automated tools.

The case is mainly used for the developement of quality softwares which will perform effectively .

History

The ISDOS project at the University of Michigan initiated a great deal of interest in the whole concept of using computer systems to help analysts in the very difficult process of analysing requirements and developing systems. Several papers by Daniel Teichroew fired a whole generation of enthusiasts with the potential of automated systems development. His PSL/PSA tool was a CASE tool although it predated the term. His insights into the power of meta-meta-models was inspiring, particularly to a former student, Dr. Hasan Sayani, currently Professor, Program Director at University of Maryland University College.

Another major thread emerged as a logical extension to the DBMS directory. By extending the range of meta-data held, the attributes of an application could be held within a dictionary and used at runtime. This "active dictionary" became the precursor to the more modern "model driven execution" (MDE) capability. However, the active dictionary did not provide a graphical representation of any of the meta-data. It was the linking of the concept of a dictionary holding analysts' meta-data, as derived from the use of an integrated set of techniques, together with the graphical representation of such data that gave rise to the earlier versions of I-CASE.

The term CASE was originally coined by software company Nastec Corporation of Southfield, Michigan in 1982 with their original integrated graphics and text editor GraphiText, which also was the first microcomputer-based system to use hyperlinks to cross-reference text strings in documents—an early forerunner of today's web page link. GraphiText's successor product, DesignAid, was the first microprocessor-based tool to logically and semantically evaluate software and system design diagrams and build a data dictionary.

Under the direction of Albert F. Case, Jr. vice president for product management and consulting, and Vaughn Frick, director of product management, the DesignAid product suite was expanded to support analysis of a wide range of structured analysis and design methodologies, notably Ed Yourdon and Tom DeMarco, Chris Gane & Trish Sarson, Ward-Mellor (real-time) SA/SD and Warnier-Orr (data driven).

The next entrant into the market was Excelerator from Index Technology in Cambridge, Mass. While DesignAid ran on Convergent Technologies and later Burroughs Ngen networked microcomputers, Index launched Excelerator on the IBM PC/AT platform. While, at the time of launch, and for several years, the IBM platform did not support networking or a centralized database as did the Convergent Technologies or Burroughs machines, the allure of IBM was strong, and Excelerator came to prominence. Hot on the heels of Excelerator were a rash of offerings from companies such as Knowledgeware (James Martin, Fran Tarkenton and Don Addington), Texas Instrument's IEF and Accenture's FOUNDATION toolset (METHOD/1, DESIGN/1, INSTALL/1, FCP).

CASE tools were at their peak in the early 1990s. At the time IBM had proposed AD/Cycle, which was an alliance of software vendors centered around IBM's Software repository using IBM DB2 in mainframe and OS/2:

The application development tools can be from several sources: from IBM, from vendors, and from the customers themselves. IBM has entered into relationships with Bachman Information Systems, Index Technology Corporation, and Knowledgeware, Inc. wherein selected products from these vendors will be marketed through an IBM complementary marketing program to provide offerings that will help to achieve complete life-cycle coverage.

With the decline of the mainframe, AD/Cycle and the Big CASE tools died off, opening the market for the mainstream CASE tools of today. Nearly all of the leaders of the CASE market of the early 1990s ended up being purchased by Computer Associates, including IEW, IEF, ADW, Cayenne, and Learmonth & Burchett Management Systems (LBMS).

Supporting software

Alfonso Fuggetta classified CASE into 3 categories:

1. *Tools* support only specific tasks in the software process.
2. *Workbenches* support only one or a few activities.
3. *Environments* support (a large part of) the software process.

Workbenches and environments are generally built as collections of tools. Tools can therefore be either stand alone products or components of workbenches and environments.

Tools

CASE tools are a class of software that automate many of the activities involved in various life cycle phases. For example, when establishing the functional requirements of a proposed application, prototyping tools can be used to develop graphic models of application screens to assist end users to visualize how an application will look after development. Subsequently, system designers can use automated design tools to transform the prototyped functional requirements into detailed design documents. Programmers can then use automated code generators to convert the design documents into code. Automated tools can be used collectively, as mentioned, or individually. For example, prototyping tools could be used to define application requirements that get passed to design technicians who convert the requirements into detailed designs in a traditional manner using flowcharts and narrative documents, without the assistance of automated design software.

Existing CASE tools can be classified along 4 different dimensions :

1. Life-Cycle Support
2. Integration Dimension
3. Construction Dimension
4. Knowledge Based CASE dimension

Let us take the meaning of these dimensions along with their examples one by one :

Life-Cycle Based CASE Tools

This dimension classifies CASE Tools on the basis of the activities they support in the information systems life cycle. They can be classified as Upper or Lower CASE tools.

- Upper CASE Tools: support strategic, planning and construction of conceptual level product and ignore the design aspect. They support traditional diagrammatic languages such as ER diagrams, Data flow diagram, Structure charts, Decision Trees, Decision tables, etc.
- Lower CASE Tools: concentrate on the back end activities of the software life cycle and hence support activities like physical design, debugging, construction, testing, integration of software components, maintenance, reengineering and reverse engineering activities.

Integration dimension

Three main CASE Integration dimensions have been proposed :

1. CASE Framework
2. ICASE Tools
3. Integrated Project Support Environment(IPSE)

Workbenches

Workbenches integrate several CASE tools into one application to support specific software-process activities. Hence they achieve:

- a homogeneous and consistent interface (presentation integration).
- easy invocation of tools and tool chains (control integration).
- access to a common data set managed in a centralized way (data integration).

CASE workbenches can be further classified into following 8 classes:

1. Business planning and modeling
2. Analysis and design
3. User-interface development
4. Programming
5. Verification and validation
6. Maintenance and reverse engineering
7. Configuration management
8. Project management

Environments

An environment is a collection of CASE tools and workbenches that supports the software process. CASE environments are classified based on the focus/basis of integration

1. Toolkits
2. Language-centered
3. Integrated
4. Fourth generation
5. Process-centered

Toolkits

Toolkits are loosely integrated collections of products easily extended by aggregating different tools and workbenches. Typically, the support provided by a toolkit is limited to programming, configuration management and project management. And the toolkit itself is environments extended from basic sets of operating system tools, for example, the

Unix Programmer's Work Bench and the VMS VAX Set. In addition, toolkits' loose integration requires user to activate tools by explicit invocation or simple control mechanisms. The resulting files are unstructured and could be in different format, therefore the access of file from different tools may require explicit file format conversion. However, since the only constraint for adding a new component is the formats of the files, toolkits can be easily and incrementally extended.

Language-centered

The environment itself is written in the programming language for which it was developed, thus enabling users to reuse, customize and extend the environment. Integration of code in different languages is a major issue for language-centered environments. Lack of process and data integration is also a problem. The strengths of these environments include good level of presentation and control integration. Interlisp, Smalltalk, Rational, and KEE are examples of language-centered environments.

Integrated

These environments achieve presentation integration by providing uniform, consistent, and coherent tool and workbench interfaces. Data integration is achieved through the *repository* concept: they have a specialized database managing all information produced and accessed in the environment. Examples of integrated environment are IBM AD/Cycle and DEC Cohesion.

Fourth-generation

Fourth-generation environments were the first integrated environments. They are sets of tools and workbenches supporting the development of a specific class of program: electronic data processing and business-oriented applications. In general, they include programming tools, simple configuration management tools, document handling facilities and, sometimes, a code generator to produce code in lower level languages. Informix 4GL, and Focus fall into this category.

Process-centered

Environments in this category focus on process integration with other integration dimensions as starting points. A process-centered environment operates by interpreting a process model created by specialized tools. They usually consist of tools handling two functions:

- Process-model execution
- Process-model production

Examples are East, Enterprise II, Process Wise, Process Weaver, and Arcadia.

Applications

All aspects of the software development life cycle can be supported by software tools, and so the use of tools from across the spectrum can, arguably, be described as CASE; from project management software through tools for business and functional analysis, system design, code storage, compilers, translation tools, test software, and so on.

However, tools that are concerned with analysis and design, and with using design information to create parts (or all) of the software product, are most frequently thought of as CASE tools. CASE applied, for instance, to a database software product, might normally involve:

- Modeling business / real-world processes and data flow
- Development of data models in the form of entity-relationship diagrams
- Development of process and function descriptions

Risks and associated controls

Common CASE risks and associated controls include:

- *Inadequate Standardization* : Linking CASE tools from different vendors (design tool from Company X, programming tool from Company Y) may be difficult if the products do not use standardized code structures and data classifications. File formats can be converted, but usually not economically. Controls include using tools from the same vendor, or using tools based on standard protocols and insisting on demonstrated compatibility. Additionally, if organizations obtain tools for only a portion of the development process, they should consider acquiring them from a vendor that has a full line of products to ensure future compatibility if they add more tools.

- *Unrealistic Expectations* : Organizations often implement CASE technologies to reduce development costs. Implementing CASE strategies usually involves high start-up costs. Generally, management must be willing to accept a long-term payback period. Controls include requiring senior managers to define their purpose and strategies for implementing CASE technologies.

- *Slow Implementation* : Implementing CASE technologies can involve a significant change from traditional development environments. Typically, organizations should not use CASE tools the first time on critical projects or projects with short deadlines because of the lengthy training process. Additionally, organizations should consider using the tools on smaller, less complex projects and gradually implementing the tools to allow more training time.

- *Weak Repository Controls* : Failure to adequately control access to CASE repositories may result in security breaches or damage to the work documents, system designs, or code modules stored in the repository. Controls include protecting the repositories with appropriate access, version, and backup controls.

Chapter 6

Software Development and Software Testing

Software development

Software development (also known as **Application Development; Software Design, Designing Software, Software Engineering, Software Application Development, Enterprise Application Development, Platform Development**) is the development of a software product in a planned and structured process. This software could be produced for a variety of purposes - the three most common purposes are to meet specific needs of a specific client/business, to meet a perceived need of some set of potential users (the case with commercial and open source software), or for personal use (e.g. a scientist may write software to automate a mundane task).

The term software development is often used to refer to the activity of computer programming, which is the process of writing and maintaining the source code, whereas the broader sense of the term includes all that is involved between the conception of the desired software through to the final manifestation of the software. Therefore, software development may include research, new development, modification, reuse, re-engineering, maintenance, or any other activities that result in software products. For larger software systems, usually developed by a team of people, some form of process is typically followed to guide the stages of production of the software.

Especially the first phase in the software development process may involve many departments, including marketing, engineering, research and development and general management.

Overview

There are several different approaches to software development, much like the various views of political parties toward governing a country. Some take a more structured, engineering-based approach to developing business solutions, whereas others may take a more incremental approach, where software evolves as it is developed piece-by-piece. Most methodologies share some combination of the following stages of software development:

- Market research
- Gathering requirements for the proposed business solution
- Analyzing the problem
- Devising a plan or design for the software-based solution
- Implementation (coding) of the software
- Testing the software
- Deployment
- Maintenance and bug fixing

These stages are often referred to collectively as the software development lifecycle, or SDLC. Different approaches to software development may carry out these stages in different orders, or devote more or less time to different stages. The level of detail of the documentation produced at each stage of software development may also vary. These stages may also be carried out in turn (a "waterfall" based approach), or they may be repeated over various cycles or iterations (a more "extreme" approach). The more extreme approach usually involves less time spent on planning and documentation, and more time spent on coding and development of automated tests. More "extreme" approaches also promote continuous testing throughout the development lifecycle, as well as having a working (or bug-free) product at all times. More structured or "waterfall" based approaches attempt to assess the majority of risks and develop a detailed plan for the software before implementation (coding) begins, and avoid significant design changes and re-coding in later stages of the software development lifecycle.

There are significant advantages and disadvantages to the various methodologies, and the best approach to solving a problem using software will often depend on the type of problem. If the problem is well understood and a solution can be effectively planned out ahead of time, the more "waterfall" based approach may work the best. If, on the other hand, the problem is unique (at least to the development team) and the structure of the software solution cannot be easily envisioned, then a more "extreme" incremental approach may work best. A software development process is a structure imposed on the development of a software product. Synonyms include software life cycle and *software process*. There are several models for such processes, each describing approaches to a variety of tasks or activities that take place during the process.

Software development topic

Marketing

The sources of ideas for software products are legion. These ideas can come from market research including the demographics of potential new customers, existing customers, sales prospects who rejected the product, other internal software development staff, or a creative third party. Ideas for software products are usually first evaluated by marketing personnel for economic feasibility, for fit with existing channels distribution, for possible effects on existing product lines, required features, and for fit with the company's marketing objectives. In a marketing evaluation phase, the cost and time assumptions become evaluated. A decision is reached early in the first phase as to whether, based on the more detailed information generated by the marketing and development staff, the project should be pursued further.

software development may involve compromising or going beyond what is required by the client, a software development project may stray into less technical concerns such as human resources, risk management, intellectual property, budgeting, crisis management, etc. These processes may also cause the role of business development to overlap with software development.

Software development methodology

A software development methodology is a framework that is used to structure, plan, and control the process of developing information systems. A wide variety of such frameworks have evolved over the years, each with its own recognized strengths and weaknesses. One system development methodology is not necessarily suitable for use by all projects. Each of the available methodologies is best suited to specific kinds of projects, based on various technical, organizational, project and team considerations.

Recent trends in the sector

The end of the 1990 provided W3C standards which enabled ontologies to unite four modelling functions in one knowledge model: the knowledge representation (in RDF(S) and OWL), the knowledge generation through inferences, the conceptual model through ontologies and the physical model through triple stores. The latest developments allow to generate applications straight from the knowledge systems (ontologies). This approach finds its justification in the use of semantic technologies with substitution of www data with verified production data. One business case for this development method is available at the finance ontology website.

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 of software implementation. 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 of testing is to detect software failures so that defects may be discovered 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 activities 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".

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.

Cost to fix a defect		**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; A flight simulator video game would have much higher defect tolerance than software for an actual airplane.

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 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. 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. van Veenendaal, Erik. "Standard glossary of terms used in Software 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 often 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 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
- TMPF TMap Next Foundation offered by the *Examination Institute for Information Science*
- TMPA TMap Next Advanced offered by the *Examination Institute for Information Science*

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.

Chapter 7

Software Maintenance and Software Configuration Management

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. .

this section 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 cohesion 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.).

Software configuration management

In software engineering, **software configuration management (SCM)** is the task of tracking and controlling changes in the software. Configuration management practices include revision control and the establishment of baselines.

SCM concerns itself with answering the question "Somebody did something, how can one reproduce it?" Often the problem involves not reproducing "it" identically, but with controlled, incremental changes. Answering the question thus becomes a matter of comparing different results and of analysing their differences. Traditional configuration management typically focused on controlled creation of relatively simple products. Now, implementers of SCM face the challenge of dealing with relatively minor increments under their own control, in the context of the complex system being developed.

Terminology

The history and terminology of SCM (which often varies) has given rise to controversy. Roger Pressman, in his book *Software Engineering: A Practitioner's Approach*, states that SCM is a "set of activities designed to control change by identifying the work products that are likely to change, establishing relationships among them, defining mechanisms for managing different versions of these work products, controlling the changes imposed, and auditing and reporting on the changes made."

Source configuration management is a related practice often used to indicate that a variety of artifacts may be managed and versioned, including software code, hardware, documents, design models, and even the directory structure itself.

Atria (later Rational Software, now a part of IBM), used "SCM" to mean "software configuration management". Gartner uses the term *software change and configuration management*.

Purposes

The goals of SCM are generally:

- Configuration identification - Identifying configurations, configuration items and baselines.
- Configuration control - Implementing a controlled change process. This is usually achieved by setting up a change control board whose primary function is to approve or reject all change requests that are sent against any baseline.
- Configuration status accounting - Recording and reporting all the necessary information on the status of the development process.
- Configuration auditing - Ensuring that configurations contain all their intended parts and are sound with respect to their specifying documents, including requirements, architectural specifications and user manuals.
- Build management - Managing the process and tools used for builds.
- Process management - Ensuring adherence to the organization's development process.
- Environment management - Managing the software and hardware that host the system.
- Teamwork - Facilitate team interactions related to the process.
- Defect tracking - Making sure every defect has traceability back to the source.

Chapter 8

Rapid Application Development

Rapid application development (**RAD**) refers to a type of software development methodology that uses minimal planning in favor of rapid prototyping. The "planning" of software developed using RAD is interleaved with writing the software itself. The lack of extensive pre-planning generally allows software to be written much faster, and makes it easier to change requirements.

Overview

Rapid application development is a software development methodology that involves methods like interactive development and software prototyping. According to Whitten (2004), it is a merger of various structured techniques, especially data-driven Information Engineering, with prototyping techniques to accelerate software systems development.

In rapid application development, structured techniques and prototyping are especially used to define users' requirements and to design the final system. The development process starts with the development of preliminary data models and business process models using structured techniques. In the next stage, requirements are verified using prototyping, eventually to refine the data and process models. These stages are repeated iteratively; further development results in "a combined business requirements and technical design statement to be used for constructing new systems".

RAD approaches may entail compromises in functionality and performance in exchange for enabling faster development and facilitating application maintenance.

History

Rapid application development is a term originally used to describe a software development process introduced by James Martin in 1991. Martin's methodology involves iterative development and the construction of prototypes. More recently, the term and its acronym have come to be used in a broader, general sense that encompasses a variety of methods aimed at speeding application development, such as the use of software frameworks of varied types, such as web application frameworks.

Rapid application development was a response to non-agile processes developed in the 1970s and 1980s, such as the Structured Systems Analysis and Design Method and other

Waterfall models. One problem with previous methodologies was that applications took so long to build that requirements had changed before the system was complete, resulting in inadequate or even unusable systems. Another problem was the assumption that a methodical requirements analysis phase alone would identify all the critical requirements. Ample evidence attests to the fact that this is seldom the case, even for projects with highly experienced professionals at all levels.

Starting with the ideas of Brian Gallagher, Alex Balchin, Barry Boehm and Scott Shultz, James Martin developed the rapid application development approach during the 1980s at IBM and finally formalized it by publishing a book in 1991, *Rapid Application Development*.

Relative effectiveness

The shift from traditional session-based client/server development to open sessionless and collaborative development like Web 2.0 has increased the need for faster iterations through the phases of the SDLC. This, coupled with the growing use of open source frameworks and products in core commercial development, has, for many developers, rekindled interest in finding a silver bullet RAD methodology.

Although most RAD methodologies foster software re-use, small team structure and distributed system development, most RAD practitioners recognize that, ultimately, no one “rapid” methodology can provide an order of magnitude improvement over any other development methodology.

All types of RAD have the potential for providing a good framework for faster product development with improved software quality, but successful implementation and benefits often hinge on project type, schedule, software release cycle and corporate culture. It may also be of interest that some of the largest software vendors such as Microsoft and IBM do not extensively use RAD in the development of their flagship products and for the most part, they still primarily rely on traditional waterfall methodologies with some degree of spiraling.

This table contains a high-level summary of some of the major types of RAD and their relative strengths and weaknesses.

Criticism

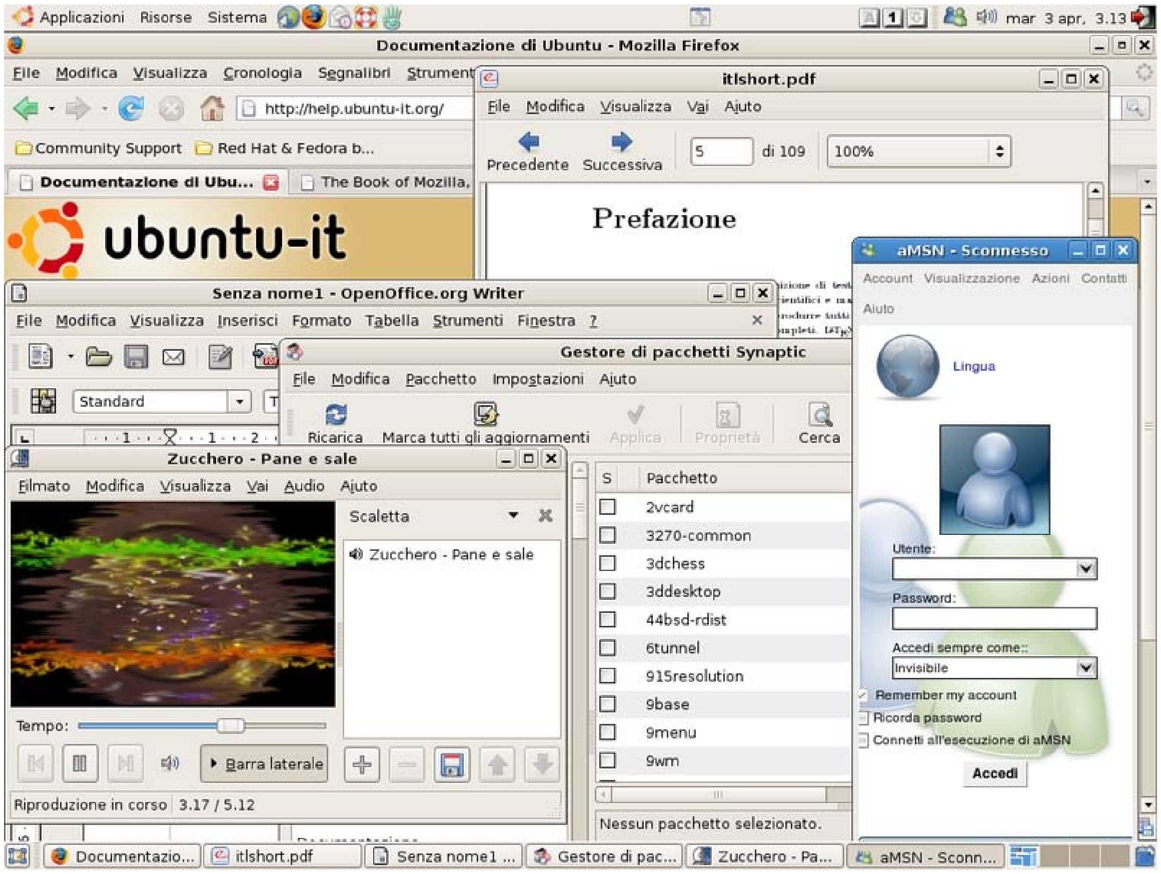
Since rapid application development is an iterative and incremental process, it can lead to a succession of prototypes that never culminate in a satisfactory production application. Such failures may be avoided if the application development tools are robust, flexible, and put to proper use. This is addressed in methods such as the 2080 Development method or other post-agile variants.

Practical implications

When organizations adopt rapid development methodologies, care must be taken to avoid role and responsibility confusion and communication breakdown within a development team, and between team and client. In addition, especially in cases where the client is absent or not able to participate with authority in the development process, the system analyst should be endowed with this authority on behalf of the client to ensure appropriate prioritisation of non-functional requirements. Furthermore, no increment of the system should be developed without a thorough and formally documented design phase.

Chapter 9

Internationalization and Localization


Applicazioni Risorse Sistema
mar 3 apr, 3.13
Documentazione di Ubuntu - Mozilla Firefox
File Modifica Visualizza Cronologia Segnalibri Strumenti
http://help.ubuntu-it.org/
Community Support
Red Hat & Fedora b...
Documentazione di Ubu...
The Book of Mozilla,
ubuntu-it
itlshort.pdf
File Modifica Visualizza Vai Aiuto
Precedente Successiva
5
di 109
100%
Prefazione
aMSN - Sconnesso
Account Visualizzazione Azioni Contatti
Aiuto
Lingua
Utente:
Password:
Accedi sempre come::
Invisibile
Remember my account
Ricorda password
Connetti all'esecuzione di aMSN
Accedi
Senza nome1 - OpenOffice.org Writer
File Modifica Visualizza Inserisci Formato Tabella Strumenti Finestra ?
Standard
Gestore di pacchetti Synaptic
File Modifica Pacchetto Impostazioni Aiuto
Ricarica
Marca tutti gli aggiornamenti
Applica
Proprietà
Cerca
S
Pacchetto
2vcard
3270-common
3dchess
3ddesktop
44bsd-rdist
6tunnel
915resolution
9base
9menu
9wm
Nessun pacchetto selezionato.
Zucchero - Pane e sale
Filmato Modifica Visualizza Vai Audio Aiuto
Scaletta
Zucchero - Pane e sale
Tempo:
Barra laterale
Riproduzione in corso 3.17 / 5.12
Documentazio...
itlshort.pdf
Senza nome1 ...
Gestore di pac...
Zucchero - Pa...
aMSN - Sconn...

Screenshot of software programs localized to Italian.

In computing, **internationalization and localization** are means of adapting computer software to different languages, regional differences and technical requirements of a target market. Internationalization is the process of designing a software application so that it can be adapted to various languages and regions without engineering changes. Localization is the process of adapting internationalized software for a specific region or language by adding locale-specific components and translating text.

The terms are frequently abbreviated to the numeronyms **i18n** (where 18 stands for the number of letters between the first *i* and last *n* in *internationalization*, a usage coined at

DEC in the 1970s or 80s) and **L10n** respectively, due to the length of the words. The capital L in L10n helps to distinguish it from the lowercase i in i18n.

Some companies, like IBM and Sun Microsystems, use the term "globalization" for the combination of internationalization and localization.

Microsoft defines Internationalization as a combination of World-Readiness and localization. World-Readiness is a developer task, which enables a product to be used with multiple scripts and cultures (globalization) and separating user interface resources in a localizable format (localizability).

This concept is also known as **NLS** (**National Language Support** or **Native Language Support**).

Nomenclature

The support of multiple languages by computer systems can be considered a continuum between localization ("L10n"), through multilingualization (or "m17n"), to internationalization ("i18n").

- A **localized** system has been adapted or converted for use in a particular **locale** (other than the one it was originally developed for), including the language of the user interface (UI), input, and display, and features such as time/date display and currency. Each instance of the system only supports a single locale, and there is no explicit support for languages that are not part of that locale (although the character set may coincidentally be usable for other languages).
- **Multilingualized** software supports multiple languages for concurrent display and input, but has a single UI language which cannot be changed. Multi-locale support for other features like date, time, number, and currency formats varies as the system tends towards full *internationalization*. In general, a multilingualized system is intended for use in one specific locale, but is capable of handling multilingual content as data.
- An **internationalized** system is equipped for use in a range of "locales" (or by users of multiple languages), by allowing the co-existence of several languages and character sets for input, display, *and* UI. In particular, a system may not be considered internationalized in the fullest sense unless the UI language is selectable by the user at runtime. Full internationalization may extend beyond support for multiple languages and orthography to compliance with jurisdiction-specific legislation (in respect of copyright, for instance) and other non-linguistic conventions.

The distinction arises because it is significantly more difficult to create a multi-lingual UI than simply to support the character sets and keyboards needed to express multiple languages. To internationalize a UI, every text string employed in interaction must be translated into all supported languages; then all output of literal strings, and literal parsing of input in UI code must be replaced by hooks to i18n libraries.

It should be noted that "internationalized" does not necessarily mean that a system can be used absolutely anywhere, since simultaneous support for *all* possible locales is both practically almost impossible and commercially very hard to justify. In many cases an internationalized system includes full support only for the most spoken languages, plus any others of particular relevance to the application.

Scope

Focal points of internationalization and localization efforts include:

- Language
    - Computer-encoded text
        - Alphabets/scripts; most recent systems use the Unicode standard to solve many of the character encoding problems.
        - Different systems of numerals
        - Writing direction which is e.g. left to right in German, right to left in Persian, Hebrew and Arabic
        - Spelling variants for different countries where the same language is spoken, e.g. *localization* (en-US, en-CA, en-GB-oed) vs. *localisation* (en-GB, en-AU)
        - Text processing differences, such as the concept of capitalization which exists in some scripts and not in others, different text sorting rules, etc.
        - Plural forms in text output, which differ depending upon language
    - Input
        - Enablement of keyboard shortcuts on any keyboard layout
    - Graphical representations of text (printed materials, online images containing text)
    - Spoken (Audio)
    - Subtitling of film and video
- Culture
    - Images and colors: issues of comprehensibility and cultural appropriateness
    - Names and titles
    - Government assigned numbers (such as the Social Security number in the US, National Insurance number in the UK, Isikukood in Estonia, and Resident registration number in South Korea.) and passports
    - Telephone numbers, addresses and international postal codes
    - Currency (symbols, positions of currency markers)
    - Weights and measures
    - Paper sizes
- Writing conventions
    - Date/time format, including use of different calendars
    - Time zones (UTC in internationalized environments)
    - Formatting of numbers (decimal separator, digit grouping)

    - Differences in symbols (e.g. quoting text using double-quotes (" "), as in English, or guillemets (« »), as in French).
- Any other aspect of the product or service that is subject to regulatory compliance

The distinction between internationalization and localization is subtle but important. Internationalization is the adaptation of products for *potential* use virtually everywhere, while localization is the addition of special features for use in a *specific* locale. Internationalization is done once per product, while localization is done once for each combination of product and locale. The processes are complementary, and must be combined to lead to the objective of a system that works globally. Subjects unique to localization include:

- Language translation
- National varieties of languages
- Special support for certain languages such as East Asian languages
- Local customs
- Local content
- Symbols
- Order of sorting (Collation)
- Aesthetics
- Cultural values and social context

Business process for internationalizing software

In order to *internationalize* a product, it is important to look at a variety of markets that your product will foreseeably enter. Details such as field length for addresses, ability to make the zip code field optional to address countries that do not have zip codes, plus the introduction of new registration flows that adhere to local laws are just some of the examples that make internationalization a complex project.

A broader approach takes into account cultural factors regarding for example the adaptation of the business process logic or the inclusion of individual cultural (behavioral) aspects.

Coding practice

The current prevailing practice is for applications to place text in resource strings which are loaded during program execution as needed. These strings, stored in resource files, are relatively easy to translate. Programs are often built to reference resource libraries depending on the selected locale data. One software library that aids this is gettext.

Thus to get an application to support multiple languages one would design the application to select the relevant language resource file at runtime. Resource files are translated to the required languages. This method tends to be application-specific and, at best, vendor-specific. The code required to manage date entry verification and many other locale-sensitive data types also must support differing locale requirements. Modern development systems and operating systems include sophisticated libraries for international support of these types.

Some tools help in detecting i18n issues and guiding software resolution of those issues, such as Lingoport's Globalyzer.

Difficulties

While translating existing text to other languages may seem easy, it is more difficult to maintain the parallel versions of texts throughout the life of the product. For instance, if a message displayed to the user is modified, all of the translated versions must be changed. This in turn results in a somewhat longer development cycle.

Many localization issues (e.g. writing direction, text sorting) require more profound changes in the software than text translation. For example, OpenOffice.Org achieves this with compilation switches.

To some degree (e.g. for Quality assurance), the development team needs someone who understands foreign languages and cultures and has a technical background. In large societies with one dominant language/culture, it may be difficult to find such a person.

Cost vs benefit tradeoff

In a commercial setting, the benefit from localization is access to more markets. Some argue that the commercial case to localize products into multiple languages is very obvious, and that all is needed is a budgetary commitment from the producer to finance the considerable costs. It costs more to produce products for international markets, but in an increasingly global economy, supporting only one language/market is scarcely an option. Still, proprietary software localization is impacted by economic viability and usually lacks the ability for end users and volunteers to self-localize, as is often the case in open-source environments.

Since open source software can generally be freely modified and redistributed, it is more amenable to localization. The KDE project, for example, has been translated into over 100 languages.

Chapter 10

Programming Tool

A **programming tool** or **software development tool** is a program or application that software developers use to create, debug, maintain, or otherwise support other programs and applications. The term usually refers to relatively simple programs that can be combined together to accomplish a task, much as one might use multiple hand tools to fix a physical object.

History

The history of software tools began with the first computers in the early 1950s that used linkers, loaders, and control programs. Tools became famous with Unix in the early 1970s with tools like grep, awk and make that were meant to be combined flexibly with pipes. The term "software tools" came from the book of the same name by Brian Kernighan and P. J. Plauger.

Tools were originally simple and light weight. As some tools have been maintained, they have been integrated into more powerful integrated development environments (IDEs). These environments consolidate functionality into one place, sometimes increasing simplicity and productivity, other times sacrificing flexibility and extensibility. The workflow of IDEs is routinely contrasted with alternative approaches, such as the use of Unix shell tools with text editors like Vim and Emacs.

The distinction between tools and applications is murky. For example, developers use simple databases (such as a file containing a list of important values) all the time as tools. However a full-blown database is usually thought of as an application in its own right.

For many years, computer-assisted software engineering (CASE) tools were sought after. Successful tools have proven elusive. In one sense, CASE tools emphasized design and architecture support, such as for UML. But the most successful of these tools are IDEs.

The ability to use a variety of tools productively is one hallmark of a skilled software engineer.

Categories

Software development tools can be roughly divided into the following categories:

- Performance analysis tools
- Debugging tools
- Static analysis and formal verification tools
- Correctness checking tools
- Memory usage tools
- Application build tools
- Integrated development environments

List of tools

Software tools come in many forms:

- Binary compatibility analysis: icheck, ABI Compliance Checker
- Bug Databases: Comparison of issue tracking systems - Including bug tracking systems
- Build Tools: Build automation, List of build automation software
- Code coverage: Code coverage#Software code coverage tools. Software Diagnostics
- Code Sharing Sites: Freshmeat, Krugle, Sourceforge, GitHub.
- Compilation and linking tools: GNU toolchain, gcc, Microsoft Visual Studio, CodeWarrior, Xcode, ICC
- Debuggers: Debugger#List of debuggers.
- Development Productivity Tools: JRebel eliminates the build and redeploy phases of Java EE Development by mapping the project workspace directly to any type application server in real-time
- Disassemblers: Generally reverse-engineering tools.
- Documentation generators: Comparison of documentation generators, help2man, Plain Old Documentation, asciidoc
- Formal methods: Mathematically-based techniques for specification, development and verification
- GUI interface generators
- Library interface generators: SWIG
- Integration Tools
- Memory Use/Leaks/Corruptions Detection: dmalloc, Electric Fence, duma, Insure++, Developer Edition. Memory leak detection: In the C programming language for instance, memory leaks are not as easily detected - software tools called memory debuggers are often used to find memory leaks enabling the programmer to find these problems much more efficiently than inspection alone.
- Parser generators: Parsing#Parser development software
- Performance analysis or profiling: List of performance analysis tool
- Refactoring Browser
- Revision control: List of revision control software, Comparison of revision control software
- Scripting languages: PHP, Awk, Perl, Python, REXX, Ruby, Shell, Tcl
- Search: grep, find
- Source code Clones/Duplications Finding: Duplicate code#Tools

- Source code formatting: indent
- Source code generation tools: Automatic programming#Implementations
- Static code analysis: List of tools for static code analysis
- Text editors: List of text editors, Comparison of text editors
- Unit testing: List of unit testing frameworks

IDEs

Integrated development environments (IDEs) combine the features of many tools into one package. They for example make it easier to do specific tasks, such as searching for content only in files in a particular project. IDEs may for example be used for development of enterprise-level applications.

Different aspects of IDEs for specific programming languages can be found in this comparison of integrated development environments.

Chapter 11

Software Documentation

Software documentation or **source code documentation** is written text that accompanies computer software. It either explains how it operates or how to use it, and may mean different things to people in different roles.

Involvement of people in software life

Documentation is an important part of software engineering. Types of documentation include:

1. Requirements - Statements that identify attributes, capabilities, characteristics, or qualities of a system. This is the foundation for what shall be or has been implemented.
2. Architecture/Design - Overview of softwares. Includes relations to an environment and construction principles to be used in design of software components.
3. Technical - Documentation of code, algorithms, interfaces, and APIs.
4. End User - Manuals for the end-user, system administrators and support staff.
5. Marketing - How to market the product and analysis of the market demand.

Requirements documentation

Requirements documentation is the description of what a particular software does or shall do. It is used throughout development to communicate what the software does or shall do. It is also used as an agreement or as the foundation for agreement on what the software shall do. Requirements are produced and consumed by everyone involved in the production of software: end users, customers, product managers, project managers, sales, marketing, software architects, usability engineers, interaction designers, developers, and testers, to name a few. Thus, requirements documentation has many different purposes.

Requirements come in a variety of styles, notations and formality. Requirements can be goal-like (e.g., *distributed work environment*), close to design (e.g., *builds can be started by right-clicking a configuration file and select the 'build' function*), and anything in between. They can be specified as statements in natural language, as drawn figures, as detailed mathematical formulas, and as a combination of them all.

The variation and complexity of requirements documentation makes it a proven challenge. Requirements may be implicit and hard to uncover. It is difficult to know exactly how much and what kind of documentation is needed and how much can be left to the architecture and design documentation, and it is difficult to know how to document requirements considering the variety of people that shall read and use the documentation. Thus, requirements documentation is often incomplete (or non-existent). Without proper requirements documentation, software changes become more difficult—and therefore more error prone (decreased software quality) and time-consuming (expensive).

The need for requirements documentation is typically related to the complexity of the product, the impact of the product, and the life expectancy of the software. If the software is very complex or developed by many people (e.g., mobile phone software), requirements can help to better communicate what to achieve. If the software is safety-critical and can have negative impact on human life (e.g., nuclear power systems, medical equipment), more formal requirements documentation is often required. If the software is expected to live for only a month or two (e.g., very small mobile phone applications developed specifically for a certain campaign) very little requirements documentation may be needed. If the software is a first release that is later built upon, requirements documentation is very helpful when managing the change of the software and verifying that nothing has been broken in the software when it is modified.

Traditionally, requirements are specified in requirements documents (e.g. using word processing applications and spreadsheet applications). To manage the increased complexity and changing nature of requirements documentation (and software documentation in general), database-centric systems and special-purpose requirements management tools are advocated.

Architecture/Design documentation

Architecture documentation is a special breed of design document. In a way, architecture documents are third derivative from the code (design document being second derivative, and code documents being first). Very little in the architecture documents is specific to the code itself. These documents do not describe how to program a particular routine, or even why that particular routine exists in the form that it does, but instead merely lays out the general requirements that would motivate the existence of such a routine. A good architecture document is short on details but thick on explanation. It may suggest approaches for lower level design, but leave the actual exploration trade studies to other documents.

Another breed of design docs is the comparison document, or trade study. This would often take the form of a *whitepaper*. It focuses on one specific aspect of the system and suggests alternate approaches. It could be at the user interface, code, design, or even architectural level. It will outline what the situation is, describe one or more alternatives, and enumerate the pros and cons of each. A good trade study document is heavy on research, expresses its idea clearly (without relying heavily on obtuse jargon to dazzle the reader), and most importantly is impartial. It should honestly and clearly explain the costs of whatever solution it offers as best. The objective of a trade study is to devise the best solution, rather than to push a particular point of view. It is perfectly acceptable to state no conclusion, or to conclude that none of the alternatives are sufficiently better than the baseline to warrant a change. It should be approached as a scientific endeavor, not as a marketing technique.

A very important part of the design document in enterprise software development is the Database Design Document (DDD). It contains Conceptual, Logical, and Physical Design Elements. The DDD includes the formal information that the people who interact with the database need. The purpose of preparing it is to create a common source to be used by all players within the scene. The potential users are:

- Database Designer
- Database Developer
- Database Administrator
- Application Designer
- Application Developer

When talking about Relational Database Systems, the document should include following parts:

- Entity - Relationship Schema, including following information and their clear definitions:
    - Entity Sets and their attributes
    - Relationships and their attributes
    - Candidate keys for each entity set
    - Attribute and Tuple based constraints
- Relational Schema, including following information:
    - Tables, Attributes, and their properties
    - Views
    - Constraints such as primary keys, foreign keys,
    - Cardinality of referential constraints
    - Cascading Policy for referential constraints
    - Primary keys

It is very important to include all information that is to be used by all actors in the scene. It is also very important to update the documents as any change occurs in the database as well.

Technical documentation

This is what most programmers mean when using the term *software documentation*. When creating software, code alone is insufficient. There must be some text along with it to describe various aspects of its intended operation. It is important for the code documents to be thorough, but not so verbose that it becomes difficult to maintain them. Several How-to and overview documentation are found specific to the software application or software product being documented by API Writers. This documentation may be used by developers, testers and also the end customers or clients using this software application. Today, we see lot of high end applications in the field of power, energy, transportation, networks, aerospace, safety, security, industry automation and a variety of other domains. Technical documentation has become important within such organizations as the basic and advanced level of information may change over a period of time with architecture changes. Hence, technical documentation has gained lot of importance in recent times, especially in the software field.

Often, tools such as Doxygen, NDoc, javadoc, EiffelStudio, Sandcastle, ROBODoc, POD, TwinText, or Universal Report can be used to auto-generate the code documents—that is, they extract the comments and software contracts, where available, from the source code and create reference manuals in such forms as text or HTML files. Code documents are often organized into a *reference guide* style, allowing a programmer to quickly look up an arbitrary function or class.

The idea of auto-generating documentation is attractive to programmers for various reasons. For example, because it is extracted from the source code itself (for example, through comments), the programmer can write it while referring to the code, and use the same tools used to create the source code to make the documentation. This makes it much easier to keep the documentation up-to-date.

Of course, a downside is that only programmers can edit this kind of documentation, and it depends on them to refresh the output (for example, by running a cron job to update the documents nightly). Some would characterize this as a pro rather than a con.

Donald Knuth has insisted on the fact that documentation can be a very difficult afterthought process and has advocated literate programming, writing at the same time and location as the source code and extracted by automatic means.

Elucidative Programming is the result of practical applications of Literate Programming in real programming contexts. The Elucidative paradigm proposes that source code and documentation be stored separately. This paradigm was inspired by the same experimental findings that produced Kelp. Often, software developers need to be able to create and access information that is not going to be part of the source file itself. Such annotations are usually part of several software development activities, such as code walks and porting, where third party source code is analysed in a functional way. Annotations can therefore help the developer during any stage of software development where a formal documentation system would hinder progress. Kelp stores annotations in separate files, linking the information to the source code dynamically.

User documentation

Unlike code documents, user documents are usually far more diverse with respect to the source code of the program, and instead simply describe how it is used.

In the case of a software library, the code documents and user documents could be effectively equivalent and are worth conjoining, but for a general application this is not often true.

Typically, the user documentation describes each feature of the program, and assists the user in realizing these features. A good user document can also go so far as to provide thorough troubleshooting assistance. It is very important for user documents to not be confusing, and for them to be up to date. User documents need not be organized in any particular way, but it is very important for them to have a thorough index. Consistency and simplicity are also very valuable. User documentation is considered to constitute a contract specifying what the software will do. API Writers are very well accomplished towards writing good user documents as they would be well aware of the software architecture and programming techniques used.

There are three broad ways in which user documentation can be organized.

1. **Tutorial:** A tutorial approach is considered the most useful for a new user, in which they are guided through each step of accomplishing particular tasks.
2. **Thematic:** A thematic approach, where chapters or sections concentrate on one particular area of interest, is of more general use to an intermediate user. Some authors prefer to convey their ideas through a knowledge based article to facilitating the user needs. This approach is usually practiced by a dynamic industry, such as Information technology, where the user population is largely correlated with the troubleshooting demands ʾ.
3. **List or Reference:** The final type of organizing principle is one in which commands or tasks are simply listed alphabetically or logically grouped, often via cross-referenced indexes. This latter approach is of greater use to advanced users who know exactly what sort of information they are looking for.

A common complaint among users regarding software documentation is that only one of these three approaches was taken to the near-exclusion of the other two. It is common to limit provided software documentation for personal computers to online help that give only reference information on commands or menu items. The job of tutoring new users or helping more experienced users get the most out of a program is left to private publishers, who are often given significant assistance by the software developer.

Marketing documentation

For many applications it is necessary to have some promotional materials to encourage casual observers to spend more time learning about the product. This form of documentation has three purposes:-

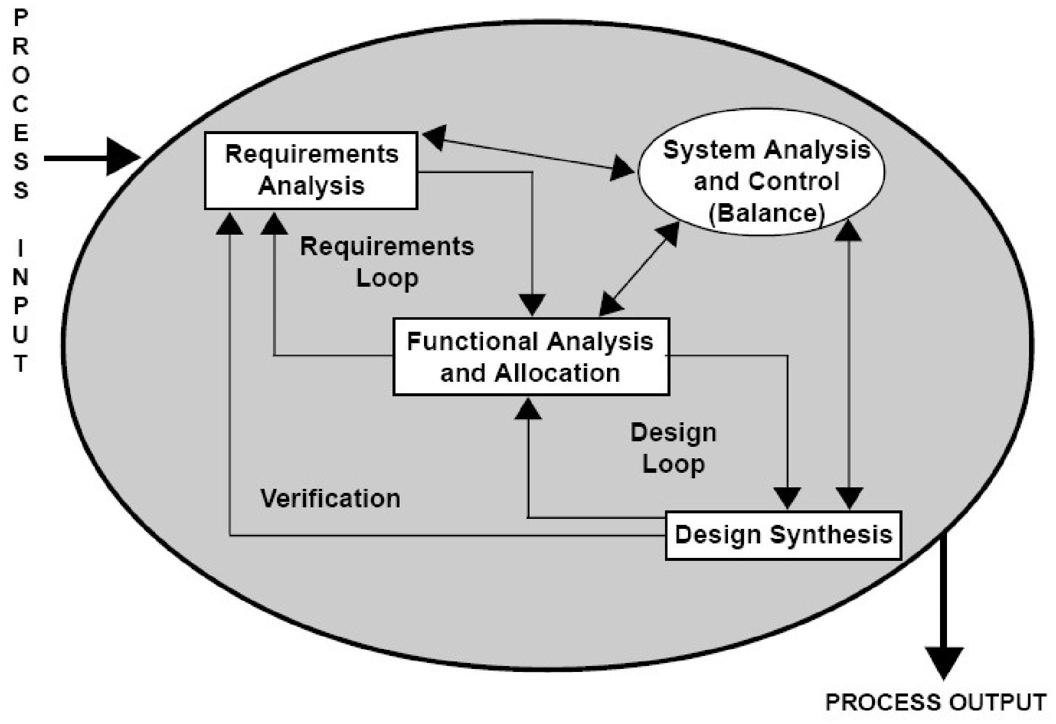
1. To excite the potential user about the product and instill in them a desire for becoming more involved with it.
2. To inform them about what exactly the product does, so that their expectations are in line with what they will be receiving.

3. To explain the position of this product with respect to other alternatives.

One good marketing technique is to provide clear and memorable *catch phrases* that exemplify the point we wish to convey, and also emphasize the interoperability of the program with anything else provided by the manufacturer.

Chapter 12

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 documented, actionable, measurable, testable, related to identified business needs or opportunities, and defined to a level of detail sufficient for system design. Requirements can be architectural, structural, behavioral, 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. Requirements Engineering can be divided into discrete chronological steps:

- Requirements elicitation,
- Requirements analysis and negotiation,
- Requirements specification,
- System modeling,
- Requirements validation,
- Requirements management.

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 suggests 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

Stakeholders (SH) are people 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)
- 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. Though they are generally idiosyncratic in nature and focused upon the perspectives and perceived needs of the stakeholder, very often without larger enterprise or system context, this perspective deficiency has the general advantage of obtaining a much richer understanding of the stakeholder's unique business processes, decision-relevant business rules, and perceived needs. Consequently this technique can serve as a means of obtaining the highly focused knowledge that is often not elicited in Joint Requirements Development sessions, where the stakeholder's attention is compelled to assume a more cross-functional context. Moreover, the in-person nature of the interviews provides a more relaxed environment where lines of thought may be explored at length.

Joint Requirements Development (JRD) Sessions

Requirements often have cross-functional implications that are unknown to individual stakeholders and often missed or incompletely defined during stakeholder interviews. These cross-functional implications can be elicited by conducting JRD sessions in a controlled environment, facilitated by a trained facilitator, wherein stakeholders participate in discussions to elicit requirements, analyze their details and uncover cross-functional implications. A dedicated scribe and Business Analyst should be present to document the discussion. Utilizing the skills of a trained facilitator to guide the discussion frees the Business Analyst to focus on the requirements definition process.

JRD Sessions are analogous to Joint Application Design Sessions. In the former, the sessions elicit requirements that guide design, whereas the latter elicit the specific design features to be implemented in satisfaction of elicited 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 or work together.
    - This abstraction makes it difficult to prioritize requirements properly; while a list does make it easy to prioritize each individual item, removing one item out of context can render an entire use case or business requirement useless.
    - This abstraction increases the likelihood of misinterpreting the requirements; as more people read them, the number of (different) interpretations of the envisioned system increase.
    - 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.

Alternative to requirement lists

As an alternative to the large, pre-defined requirement lists Agile Software Development uses User stories to define a requirement in every day language.

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 best 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?

Architectural Requirements

Architectural requirements explain what has to be done by identifying the necessary system architecture of a system.

Structural Requirements

Structural requirements explain what has to be done by identifying the necessary structure of a system.

Behavioral Requirements

Behavioral requirements explain what has to be done by identifying the necessary behavior of a system.

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

Chapter 13

Software Project Management

Software project management is the art and science of planning and leading software projects. It is a sub-discipline of project management in which software projects are planned, monitored and controlled.

History

The history of software project management is closely related to the history of software. Software was developed for dedicated purposes for dedicated machines until the concept of object-oriented programming began to become popular in the 1960's, making *repeatable solutions* possible for the software industry. Dedicated systems could be adapted to other uses thanks to component-based software engineering. Companies quickly understood the relative ease of use that software programming had over hardware circuitry, and the software industry grew very quickly in the 1970's and 1980's. To manage new development efforts, companies applied proven project management methods, but project schedules slipped during test runs, especially when confusion occurred in the gray zone between the user specifications and the delivered software. To be able to avoid these problems, software project management methods focused on matching user requirements to delivered products, in a method known now as the waterfall model. Since then, analysis of software project management failures has shown that the following are the most common causes:

1. Unrealistic or unarticulated project goals
2. Inaccurate estimates of needed resources
3. Badly defined system requirements
4. Poor reporting of the project's status
5. Unmanaged risks
6. Poor communication among customers, developers, and users
7. Use of immature technology
8. Inability to handle the project's complexity
9. Sloppy development practices
10. Poor project management
11. Stakeholder politics
12. Commercial pressures

The first three items in the list above show the difficulties articulating the needs of the client in such a way that proper resources can deliver the proper project goals. Specific software project management tools are useful and often necessary, but the true art in software project management is applying the correct method and then using tools to support the method. Without a method, tools are worthless. Since the 1960's, several proprietary software project management methods have been developed by software manufacturers for their own use, while computer consulting firms have also developed similar methods for their clients. Today software project management methods are still evolving, but the current trend leads away from the waterfall model to a more cyclic project delivery model that imitates a Software release life cycle.

Software development process

A software development process is concerned primarily with the production aspect of software development, as opposed to the technical aspect, such as software tools. These processes exist primarily for supporting the management of software development, and are generally skewed toward addressing business concerns. Many software development processes can be run in a similar way to general project management processes. Examples are:

- Risk management is the process of measuring or assessing risk and then developing strategies to manage the risk. In general, the strategies employed include transferring the risk to another party, avoiding the risk, reducing the negative effect of the risk, and accepting some or all of the consequences of a particular risk. Risk management in software project management begins with the business case for starting the project, which includes a cost-benefit analysis as well as a list of fallback options for project failure, called a contingency plan.
    - A subset of risk management that is gaining more and more attention is "Opportunity Management", which means the same thing, except that the potential risk outcome will have a positive, rather than a negative impact. Though theoretically handled in the same way, using the term "opportunity" rather than the somewhat negative term "risk" helps to keep a team focussed on possible positive outcomes of any given risk register in their projects, such as spin-off projects, windfalls, and free extra resources.
- Requirements management is the process of identifying, eliciting, documenting, analyzing, tracing, prioritizing and agreeing on requirements and then controlling change and communicating to relevant stakeholders. New or altered computer system Requirements management, which includes Requirements analysis, is an important part of the software engineering process; whereby business analysts or software developers identify the needs or requirements of a client; having identified these requirements they are then in a position to design a solution.
- Change management is the process of identifying, documenting, analyzing, prioritizing and agreeing on changes to scope (project management) and then controlling changes and communicating to relevant stakeholders. Change impact analysis of new or altered scope, which includes Requirements analysis at the change level, is an important part of the software engineering process; whereby

business analysts or software developers identify the altered needs or requirements of a client; having identified these requirements they are then in a position to re-design or modify a solution. Theoretically, each change can impact the timeline and budget of a software project, and therefore by definition must include risk-benefit analysis before approval.

- Software configuration management is the process of identifying, and documenting the scope itself, which is the software product underway, including all sub-products and changes and enabling communication of these to relevant stakeholders. In general, the processes employed include version control, naming convention (programming), and software archival agreements.
- Release management is the process of identifying, documenting, prioritizing and agreeing on releases of software and then controlling the release schedule and communicating to relevant stakeholders. Most software projects have access to three software environments to which software can be released; Development, Test, and Production. In very large projects, where distributed teams need to integrate their work before release to users, there will often be more environments for testing, called unit testing, system testing, or integration testing, before release to User acceptance testing (UAT).
    - A subset of release management that is gaining more and more attention is Data Management, as obviously the users can only test based on data that they know, and "real" data is only in the software environment called "production". In order to test their work, programmers must therefore also often create "dummy data" or "data stubs". Traditionally, older versions of a production system were once used for this purpose, but as companies rely more and more on outside contributors for software development, company data may not be released to development teams. In complex environments, datasets may be created that are then migrated across test environments according to a test release schedule, much like the overall software release schedule.

Project planning, monitoring and control

The purpose of project planning is to identify the scope of the project, estimate the work involved, and create a project schedule. Project planning begins with requirements that define the software to be developed. The project plan is then developed to describe the tasks that will lead to completion.

The purpose of project monitoring and control is to keep the team and management up to date on the project's progress. If the project deviates from the plan, then the project manager can take action to correct the problem. Project monitoring and control involves status meetings to gather status from the team. When changes need to be made, change control is used to keep the products up to date.

Issue

In computing, the term **issue** is a unit of work to accomplish an improvement in a system. An issue could be a bug, a requested feature, task, missing documentation, and so forth. The word "issue" is popularly misused in lieu of "problem." This usage is probably related.

For example, OpenOffice.org used to call their modified version of BugZilla IssueZilla. As of September 2010, they call their system Issue Tracker.

Problems occur from time to time and fixing them in a timely fashion is essential to achieve correctness of a system and avoid delayed deliveries of products.

Severity levels

Issues are often categorized in terms of **severity levels**. Different companies have different definitions of severities, but some of the most common ones are:

- Critical
- High - The bug or issue affects a crucial part of a system, and must be fixed in order for it to resume normal operation.
- Medium - The bug or issue affects a minor part of a system, but has some impact on its operation. This severity level is assigned when a non-central requirement of a system is affected.
- Low - The bug or issue affects a minor part of a system, and has very little impact on its operation. This severity level is assigned when a non-central requirement of a system (and with lower importance) is affected.
- Cosmetic - The system works correctly, but the appearance does not match the expected one. For example: wrong colors, too much or too little spacing between contents, incorrect font sizes, typos, etc. This is the lowest priority issue.

In many software companies, issues are often investigated by Quality Assurance Analysts when they verify a system for correctness, and then assigned to the developer(s) that are responsible for resolving them. They can also be assigned by system users during the User Acceptance Testing (UAT) phase.

Issues are commonly communicated using Issue or Defect Tracking Systems. In some other cases, emails or instant messengers are used.

Philosophy

As a subdiscipline of project management, some regard the management of software development akin to the management of manufacturing, which can be performed by someone with management skills, but no programming skills. John C. Reynolds rebuts this view, and argues that software development is entirely design work, and compares a manager who cannot program to the managing editor of a newspaper who cannot write.

Chapter 14

Software Development Process

A **software development process**, also known as a **software development lifecycle**, is a structure imposed on the development of a software product. Similar terms include software life cycle and *software process*. There are several models for such processes, each describing approaches to a variety of tasks or activities that take place during the process. Some people consider a lifecycle model a more general term and a software development process a more specific term. For example, there are many specific software development processes that 'fit' the spiral lifecycle model.

Overview

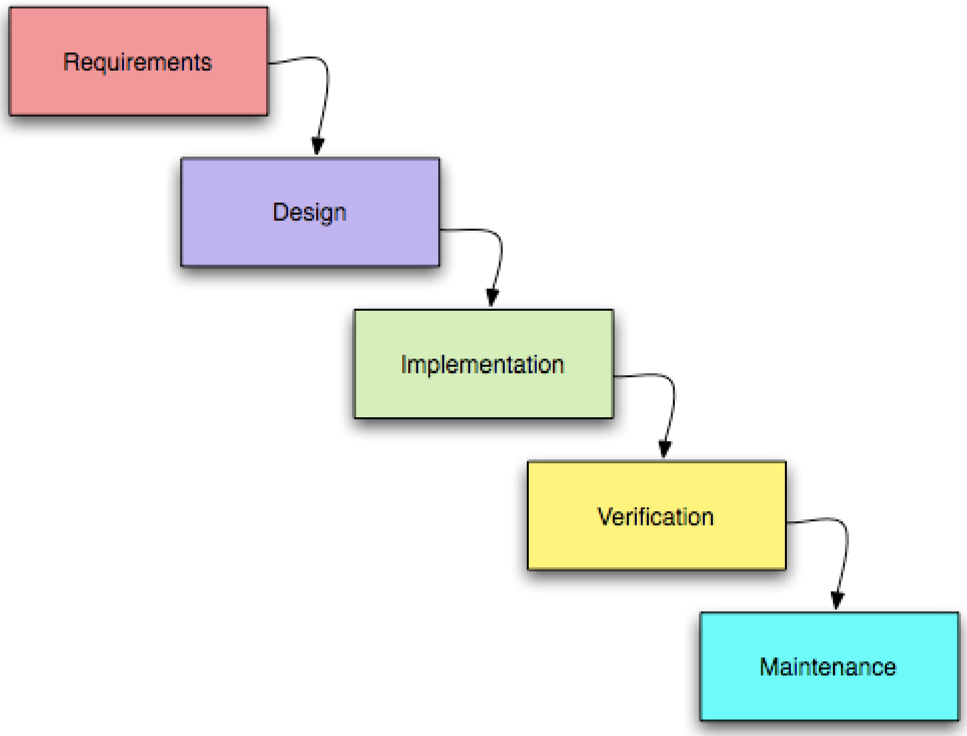
The large and growing body of software development organizations implement process methodologies. Many of them are in the defense industry, which in the U.S. requires a rating based on 'process models' to obtain contracts.

The international standard for describing the method of selecting, implementing and monitoring the life cycle for software is ISO 12207.

A decades-long goal has been to find repeatable, predictable processes that improve productivity and quality. Some try to systematize or formalize the seemingly unruly task of writing software. Others apply project management techniques to writing software. Without project management, software projects can easily be delivered late or over budget. With large numbers of software projects not meeting their expectations in terms of functionality, cost, or delivery schedule, effective project management appears to be lacking.

Organizations may create a Software Engineering Process Group (SEPG), which is the focal point for process improvement. Composed of line practitioners who have varied skills, the group is at the center of the collaborative effort of everyone in the organization who is involved with software engineering process improvement.

Software development activities



The activities of the software development process represented in the waterfall model. There are several other models to represent this process.

Planning

The important task in creating a software product is extracting the requirements or requirements analysis. Customers typically have an abstract idea of what they want as an end result, but not what software should do. Incomplete, ambiguous, or even contradictory requirements are recognized by skilled and experienced software engineers at this point. Frequently demonstrating live code may help reduce the risk that the requirements are incorrect.

Once the general requirements are gathered from the client, an analysis of the scope of the development should be determined and clearly stated. This is often called a scope document.

Certain functionality may be out of scope of the project as a function of cost or as a result of unclear requirements at the start of development. If the development is done externally, this document can be considered a legal document so that if there are ever disputes, any ambiguity of what was promised to the client can be clarified.

Implementation, testing and documenting

Implementation is the part of the process where software engineers actually program the code for the project.

Software testing is an integral and important part of the software development process. This part of the process ensures that defects are recognized as early as possible.

Documenting the internal design of software for the purpose of future maintenance and enhancement is done throughout development. This may also include the writing of an API, be it external or internal. It is very important to document everything in the project.

Deployment and maintenance

Deployment starts after the code is appropriately tested, is approved for release and sold or otherwise distributed into a production environment.

Software Training and Support is important and a lot of developers fail to realize that. It would not matter how much time and planning a development team puts into creating software if nobody in an organization ends up using it. People are often resistant to change and avoid venturing into an unfamiliar area, so as a part of the deployment phase, it is very important to have training classes for new clients of your software.

Maintaining and enhancing software to cope with newly discovered problems or new requirements can take far more time than the initial development of the software. It may be necessary to add code that does not fit the original design to correct an unforeseen problem or it may be that a customer is requesting more functionality and code can be added to accommodate their requests. If the labor cost of the maintenance phase exceeds 25% of the prior-phases' labor cost, then it is likely that the overall quality of at least one prior phase is poor. In that case, management should consider the option of rebuilding the system (or portions) before maintenance cost is out of control.

Software Development Models

Several models exist to streamline the development process. Each one has its pros and cons, and it's up to the development team to adopt the most appropriate one for the project. Sometimes a combination of the models may be more suitable.

Waterfall Model

The waterfall model shows a process, where developers are to follow these phases in order:

1. Requirements specification (Requirements analysis)
2. Software Design
3. Integration

4. Testing (or Validation)
5. Deployment (or Installation)
6. Maintenance

In a strict Waterfall model, after each phase is finished, it proceeds to the next one. Reviews may occur before moving to the next phase which allows for the possibility of changes (which may involve a formal change control process). Reviews may also be employed to ensure that the phase is indeed complete; the phase completion criteria are often referred to as a "gate" that the project must pass through to move to the next phase. Waterfall discourages revisiting and revising any prior phase once it's complete. This "inflexibility" in a pure Waterfall model has been a source of criticism by supporters of other more "flexible" models.

Spiral Model

The key characteristic of a Spiral model is risk management at regular stages in the development cycle. In 1988, Barry Boehm published a formal software system development "spiral model", which combines some key aspect of the waterfall model and rapid prototyping methodologies, but provided emphasis in a key area many felt had been neglected by other methodologies: deliberate iterative risk analysis, particularly suited to large-scale complex systems.

The Spiral is visualized as a process passing through some number of iterations, with the four quadrant diagram representative of the following activities:

1. formulate plans to: identify software targets, selected to implement the program, clarify the project development restrictions;
2. Risk analysis: an analytical assessment of selected programs, to consider how to identify and eliminate risk;
3. the implementation of the project: the implementation of software development and verification;

Risk-driven spiral model, emphasizing the conditions of options and constraints in order to support software reuse, software quality can help as a special goal of integration into the product development. However, the spiral model has some restrictive conditions, as follows:

1. The spiral model emphasizes risk analysis, and thus requires customers to accept this analysis and act on it. This requires both trust in the developer as well as the willingness to spend more to fix the issues, which is the reason why this model is often used for large-scale internal software development.
2. If the implementation of risk analysis will greatly affect the profits of the project, the spiral model should not be used.
3. Software developers have to actively look for possible risks, and analyze it accurately for the spiral model to work.

The first stage is to formulate a plan to achieve the objectives with these constraints, and then strive to find and remove all potential risks through careful analysis and, if necessary, by constructing a prototype. If some risks can not be ruled out, the customer has to decide whether to terminate the project or to ignore the risks and continue anyway. Finally, the results are evaluated and the design of the next phase begins.

Iterative and Incremental development

Iterative development prescribes the construction of initially small but ever larger portions of a software project to help all those involved to uncover important issues early before problems or faulty assumptions can lead to disaster. Iterative processes are preferred by commercial developers because it allows a potential of reaching the design goals of a customer who does not know how to define what they want.

Agile development

Agile software development uses iterative development as a basis but advocates a lighter and more people-centric viewpoint than traditional approaches. Agile processes use feedback, rather than planning, as their primary control mechanism. The feedback is driven by regular tests and releases of the evolving software.

There are many variations of agile processes:

- In Extreme Programming (XP), the phases are carried out in extremely small (or "continuous") steps compared to the older, "batch" processes. The (intentionally incomplete) first pass through the steps might take a day or a week, rather than the months or years of each complete step in the Waterfall model. First, one writes automated tests, to provide concrete goals for development. Next is coding (by a pair of programmers), which is complete when all the tests pass, and the programmers can't think of any more tests that are needed. Design and architecture emerge out of refactoring, and come after coding. Design is done by the same people who do the coding. (Only the last feature — merging design and code — is common to *all* the other agile processes.) The incomplete but functional system is deployed or demonstrated for (some subset of) the users (at least one of which is on the development team). At this point, the practitioners start again on writing tests for the next most important part of the system.

- Scrum

Process Improvement Models

Capability Maturity Model Integration

The Capability Maturity Model Integration (CMMI) is one of the leading models and based on best practice. Independent assessments grade organizations on how well they follow their defined processes, not on the quality of those processes or the software produced. CMMI has replaced CMM.

ISO 9000

ISO 9000 describes standards for a formally organized process to manufacture a product and the methods of managing and monitoring progress. Although the standard was originally created for the manufacturing sector, ISO 9000 standards have been applied to software development as well. Like CMMI, certification with ISO 9000 does not guarantee the quality of the end result, only that formalized business processes have been followed.

ISO 15504

ISO 15504, also known as Software Process Improvement Capability Determination (SPICE), is a "framework for the assessment of software processes". This standard is aimed at setting out a clear model for process comparison. SPICE is used much like CMMI. It models processes to manage, control, guide and monitor software development. This model is then used to measure what a development organization or project team actually does during software development. This information is analyzed to identify weaknesses and drive improvement. It also identifies strengths that can be continued or integrated into common practice for that organization or team.

Formal methods

Formal methods are mathematical approaches to solving software (and hardware) problems at the requirements, specification and design levels. Examples of formal methods include the B-Method, Petri nets, Automated theorem proving, RAISE and VDM. Various formal specification notations are available, such as the Z notation. More generally, automata theory can be used to build up and validate application behavior by designing a system of finite state machines.

Finite state machine (FSM) based methodologies allow executable software specification and by-passing of conventional coding.

Formal methods are most likely to be applied in avionics software, particularly where the software is safety critical. Software safety assurance standards, such as DO178B demand formal methods at the highest level of categorization (Level A).

Formalization of software development is creeping in, in other places, with the application of Object Constraint Language (and specializations such as Java Modeling Language) and especially with Model-driven architecture allowing execution of designs, if not specifications.

Another emerging trend in software development is to write a specification in some form of logic (usually a variation of FOL), and then to directly execute the logic as though it were a program. The OWL language, based on Description Logic, is an example. There is also work on mapping some version of English (or another natural language) automatically to and from logic, and executing the logic directly. Examples are Attempto Controlled English, and Internet Business Logic, which does not seek to control the vocabulary or syntax. A feature of systems that support bidirectional English-logic mapping and direct execution of the logic is that they can be made to explain their results, in English, at the business or scientific level.

The Government Accountability Office, in a 2003 report on one of the Federal Aviation Administration's air traffic control modernization programs, recommends following the agency's guidance for managing major acquisition systems by

- establishing, maintaining, and controlling an accurate, valid, and current performance measurement baseline, which would include negotiating all authorized, unpriced work within 3 months;
- conducting an integrated baseline review of any major contract modifications within 6 months; and
- preparing a rigorous life-cycle cost estimate, including a risk assessment, in accordance with the Acquisition System Toolset's guidance and identifying the level of uncertainty inherent in the estimate.

Chapter 15

Aspect-Oriented Software Development

In computing, **Aspect-oriented software development** (AOSD) is an emerging software development technology that seeks new modularizations of software systems in order to isolate secondary or supporting functions from the main program's business logic. AOSD allows multiple concerns to be expressed separately and automatically unified into working systems.

Traditional software development focuses on decomposing systems into units of primary functionality, while recognizing that there are other issues of concern that do not fit well into the primary decomposition. The traditional development process leaves it to the programmers to code modules corresponding to the primary functionality and to make sure that all other issues of concern are addressed in the code wherever appropriate. Programmers need to keep in mind all the things that need to be done, how to deal with each issue, the problems associated with the possible interactions, and the execution of the right behavior at the right time. These concerns span multiple primary functional units within the application, and often result in serious problems faced during application development and maintenance. The distribution of the code for realizing a concern becomes especially critical as the requirements for that concern evolve — a system maintainer must find and correctly update a variety of situations.

Aspect-Oriented Software Development focuses on the identification, specification and representation of cross-cutting concerns and their modularization into separate functional units as well as their automated composition into a working system.

History

Aspect-Oriented Software Development describes a number of approaches to software modularization and composition including, in order of publication, reflection and metaobject protocols, Composition Filters, developed at the University of Twente in the Netherlands, Subject-Oriented Programming (later extended as Multidimensional Separation of Concerns) at IBM, Feature Oriented Programming at University of Texas at Austin, Adaptive Programming at Northeastern University, USA, and Aspect-Oriented Programming (AOP) at Palo Alto Research Center. The term **aspect-oriented** was introduced by Gregor Kiczales and his team at Palo Alto Research Center who also first developed the explicit concept of AOP and the AOP language called AspectJ which has gained considerable acceptance and popularity within the Java developer community.

Currently, several aspect-oriented programming languages are available for a variety of languages and platforms.

Just as object-oriented programming led to the development of a large class of object-oriented development methodologies, AOP has encouraged a nascent set of software engineering technologies, including methodologies for dealing with aspects, modeling techniques (often based on the ideas of the Unified Modeling Language, UML), and testing technology for assessing the effectiveness of aspect approaches. AOSD now refers to a wide range of software development techniques that support the modularization of crosscutting concerns in a software system, from requirement engineering to analysis and design, architecture, programming and implementation techniques, testing and software maintenance techniques.

Aspect-oriented software development has constantly gained in popularity, and is the subject of an annual conference, the International Conference on Aspect-Oriented Software Development, held for the first time in 2002 in Enschede, The Netherlands. AOSD is a rapidly evolving area. It is a popular topic of Software Engineering research, especially in Europe, where research activities on AOSD are coordinated by the European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe), funded by the European Commission.

Motivation

Crosscutting concerns

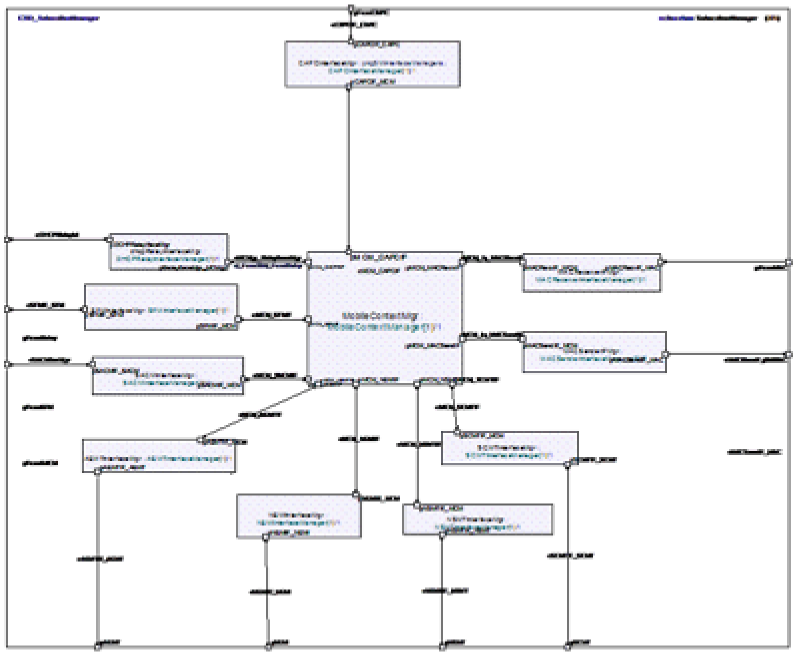


Figure 3 — An UML architecture diagram for a telecom component

The motivation for aspect-oriented programming approaches stem from the problems caused by code scattering and tangling. The purpose of Aspect-Oriented Software Development is to provide systematic means to modularize crosscutting concerns.

The implementation of a concern is **scattered** if its code is spread out over multiple modules. The concern affects the implementation of multiple modules. Its implementation is not modular.

The implementation of a concern is **tangled** if its code is intermixed with code that implements other concerns. The module in which tangling occurs is not cohesive.

Scattering and tangling often go together, even though they are different concepts.

Aspect-oriented software development considers that code scattering and tangling are the symptoms of crosscutting concerns. **Crosscutting concerns** can not be modularized using the decomposition mechanisms of the language (object or procedures) because they inherently follow different decomposition rules. The implementation and integration of these concerns with the primary functional decomposition of the system causes code tangling and scattering.

Example 1: Logging in Apache Tomcat

Figure 1 illustrates a decomposition into classes of the Apache Tomcat web container. The vertical bars in the diagram correspond to different packages in the Tomcat distribution. The length of these bars corresponds to the size of the code of these packages. In Figure 1, the code that corresponds to the classloading concern of the application have been highlighted in red. Classloading in Tomcat is a modular concern with respect to the system decomposition. Its implementation is contained in a small number of classes and is not intertwined with the implementation of other concerns.

Figure 2 represents the implementation of the logging concern in Tomcat. Logging in Tomcat is a crosscutting concern. Its implementation spreads over many classes and packages and is intermixed with the implementation of many other concerns.

Example 2: Coordination of components

Figure 3 represents the UML architecture diagram of a telecom component. Each box corresponds to a process that communicates with other processes through connectors.

Figure 4 illustrates the impact of a coordination concern on the architecture of the system, such as *When the system starts up, all parts must initialize successfully, otherwise the system must shut down*.

The highlighted box corresponds to a coordinator process. This concern has an impact on the implementation of each process in the diagram. Its implementation crosscuts the implementation of the other processes.

Examples of crosscutting concerns

Examples of concerns that tend to be crosscutting include:

- Synchronization
- Real-time constraints
- Error detection and correction
- Product features
- Memory management
- Data validation
- Persistence
- Transaction processing
- Information security
- Caching
- Logging
- Monitoring
- Business rules
- Code mobility
- Internationalization and localization
- Domain-specific optimizations

Problems caused by scattering and tangling

Scattering and tangling of behavior are the symptoms that the implementation of a concern is not well modularized. A concern that is not modularized does not exhibit a well defined interface. The interactions between the implementation of the concern and the modules of the system are not explicitly declared. They are encoded implicitly through the dependencies and interactions between fragments of code that implement the concern and the implementation of other modules.

The lack of interfaces between the implementation of crosscutting concerns and the implementation of the modules of the system impedes the development, the evolution and the maintenance of the system.

System development

A module is primarily a unit of independent development. It can be implemented to a large extent independently of other modules. Modularity is achieved through the definition of well defined interfaces between segments of the system.

The lack of explicit interfaces between crosscutting concerns and the modules obtained through the functional decomposition of the system imply that the implementation of these concerns, as well as the responsibility with respect to the correct implementation of these concerns, cannot be assigned to independent development teams. This responsibility has to be shared among different developers that work on the implementation of different modules of the system and have to integrate the crosscutting concern with the module behavior.

Furthermore, modules whose implementation is tangled with crosscutting concerns are hard to reuse in different contexts. Crosscutting impedes reuse of components. The lack of interfaces between crosscutting concerns and other modules makes it hard to represent and reason about the overall architecture of a system. As the concern is not modularized, the interactions between the concern and the top-level components of the system are hard to represent explicitly. Hence, these concerns become hard to reason about because the dependencies between crosscutting concerns and components are not specified.

Finally, concerns that are not modularized are hard to test in isolation. The dependencies of the concern with respect to behavior of other modules are not declared explicitly. Hence, the implementation of unit test for such concerns requires knowledge about the implementation of many modules in the system.

System maintenance and evolution

The lack of support for the modular implementation of crosscutting concerns is especially problematic when the implementation of this concern needs to be modified. The comprehension of the implementation of a crosscutting concern requires the inspection of the implementation of all the modules with which it interacts. Hence, modifications of the system that affect the implementation of crosscutting concern require a manual inspection of all the locations in the code that are relevant to the crosscutting concern. The system maintainer must find and correctly update a variety of poorly identified situations.

Overview

Nature of aspect-orientation

The focus of Aspect-Oriented Software Development (AOSD) is in the investigation and implementation of new structures for software modularity that provide support for explicit abstractions to modularize concerns. Aspect-Oriented Programming approaches provide explicit abstractions for the modular implementation of concerns in design, code, documentation, or other artifacts developed during the software life-cycle. These modularized concerns are called **aspects**, and aspect-oriented approaches provide methods to compose them. Some approaches denote a root concern as the **base**. Various approaches provide different flexibility with respect to composition of aspects

Quantification and obliviousness

The best known definition of the nature of AOSD is due to Filman and Friedman, which characterized AOSD using the equation *aspect orientation = quantification + obliviousness*.

AOP can be understood as the desire to make quantified statements about the behavior of programs, and to have these quantifications hold over programs written by oblivious programmers.

AOP is the desire to make statements of the form: In program P, whenever condition C arises, perform action A over a conventionally coded program P.

Obliviousness implies that a program has no knowledge of which aspects modify it where or when, whereas quantification refers to the ability of aspects to affect multiple points in the program.

The notion of **non-invasiveness** is often preferred to the term obliviousness. Non-invasiveness expresses that aspects can add behavior to a program without having to perform changes in that program, yet it does not assume that programs are not aware of the aspects.

Filman's definition of aspect-orientation is often considered too restrictive. Many aspect-oriented approaches use annotations to explicitly declare the locations in the system where aspects introduce behavior. These approaches require the manual inspection and modification of other modules in the system and are therefore invasive. Furthermore, aspect-orientation does not necessarily require quantification. Aspects can be used to isolate features whose implementation would otherwise be tangled with other features. Such aspects do not necessarily use quantification over multiple locations in the system.

The essential features of Aspect-Oriented Software Development are therefore better characterized in terms of the modularity of the implementation of crosscutting concerns, the abstractions provided by aspect-oriented languages to enable modularization and the expressiveness of the aspect-oriented composition operators.

Concepts and terminology

Aspect-oriented approaches provide explicit support for localizing concerns into separated modules, called **aspects**. An aspect is a module that encapsulates a concern. Most aspect-oriented languages support the non-invasive introduction of behavior into a code base and quantification over points in the program where this behavior should be introduced. These points are called join points.

Join point model

Join points are points in the execution of the system, such as method calls, where behavior supplied by aspects is combined. A join point is a point in the execution of the program, which is used to define the dynamic structure of a crosscutting concern.

The **join point model** of an aspect-oriented language defines the types of join points that are supported by the aspect-oriented language and the possible interaction points between aspects and base modules.

The dynamic interpretation of join points makes it possible to expose runtime information such as the caller or callee of a method from a join point to a matching pointcut. Nowadays, there are various join point models around and still more under development. They heavily depend on the underlying programming language and AO language.

Examples of join points are

- method execution
- method call
- field read and write access
- exception handler execution
- static and dynamic initialization

A method call join point covers the actions of an object receiving a method call. It includes all the actions that compose a method call, starting after all arguments are evaluated up to return.

Many AOP approaches implement aspect behavior by weaving hooks into **join point shadows**, which is the static projection of a join point onto the program code.

Figure 5 illustrates possible join points in the execution of a small object-oriented program. The highlighted join points include the execution of method *moveBy(int, int)* on a *Line* object, the calls to methods *moveBy(int, int)* on the *Point* objects in the context of the *Line* object, the execution of these methods in the context of the *Point* objects and the calls and execution of the *setX(int)* and *setY(int)* methods.

Pointcut designators

The quantification over join points is expressed at the language level. This quantification may be implicit in the language structure or may be expressed using a query-like construct called a pointcut. Pointcuts are defined as a predicate over the syntax-tree of the program, and define an interface that constrains which elements of the base program are exposed by the pointcut. A pointcut picks out certain join points and values at those points. The syntactic formulation of a pointcut varies from approach to approach, but a pointcut can often be composed out of other pointcuts using the boolean operators AND, OR and NOT. Pointcut expressions can concisely capture a wide range of events of interests, using wildcards. For example, in AspectJ syntax, the move pointcut

```
pointcut move: call(public * Figure.* (..))
```

picks out each call to Figure's public methods.

cflow poincuts identify join points based on whether they occur in the dynamic context of other join points. For example, in AspectJ syntax *cflow(move())* picks out each join point that occurs in the dynamic context of the join points picked out by the move pointcut.

Pointcuts can be classified in two categories:

- Kinded pointcuts, such as the call pointcut, match one kind of join point using a signature.
- Non-kinded pointcuts, such as the cflow pointcut match all kinds of join points using a variety of properties.

Advice bodies

An advice body is code that is executed when a join point is reached. Advice modularizes the functional details of a concern. The order in which the advice bodies contributed by aspects (and by the base) may be controlled in a variety of ways, including:

- as a join point is reached, before the execution proceeds with the base
- after the base semantics for the join point. When the join point corresponds to the execution of a method, an after advice can be executed after the method returned or after raising an exception
- as the join point is reached, with explicit control over whether the base semantics is executed. Around advice can modify the control flow of the program.

More general ways to describe the ordering of advice bodies in terms of partial-order graphs have also been provided.

When the execution of a join point satisfies a pointcut expression, the base and advice code associated with the join point are executed. The advice may interact with the rest system through a join point instance containing reflective information on the context of the event that triggered the advice, such as the arguments of a method call or the target instance of a call.

Inter-type declarations

Inter-type declarations allow the programmer to modify a program's static structure, such as class members and classes hierarchy. New members can be inserted and classes can be pushed down the class hierarchy.

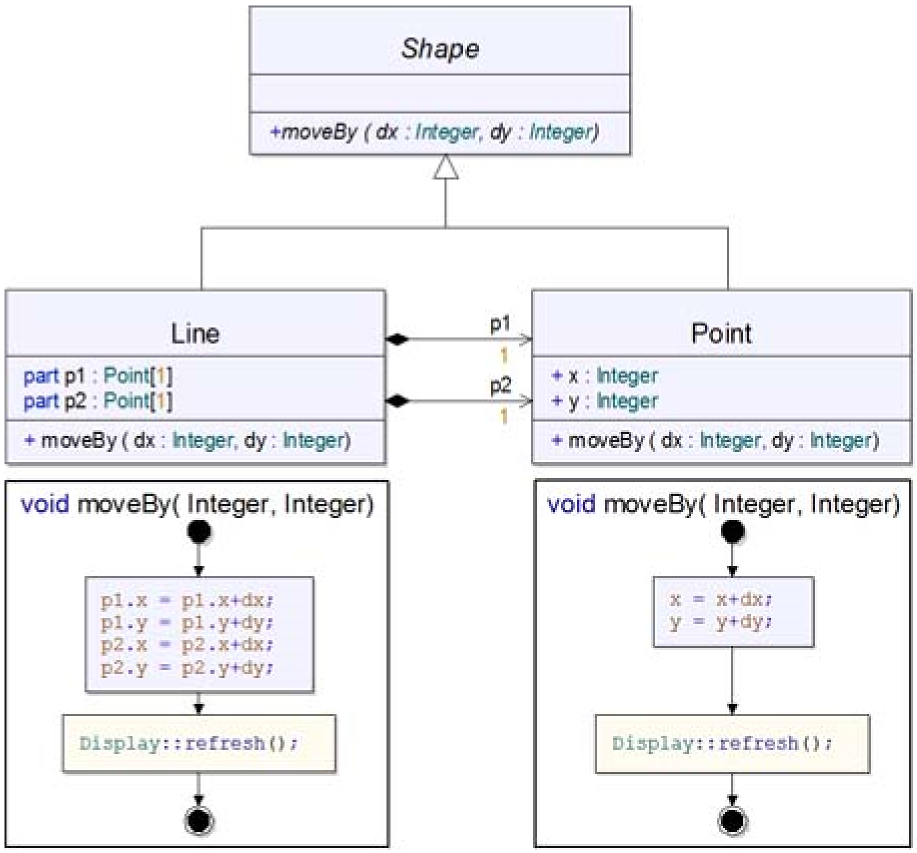
Aspects

An aspect is a module that encapsulates a concern. An aspect is composed of pointcuts, advice bodies and inter-type declarations. In some approaches, and aspect may also contain classes and methods.

Aspect weaving

Aspect weaving is a composition mechanism that coordinates aspects with the other modules of the system. It is performed by a specialized compiler, called an aspect weaver.

Example

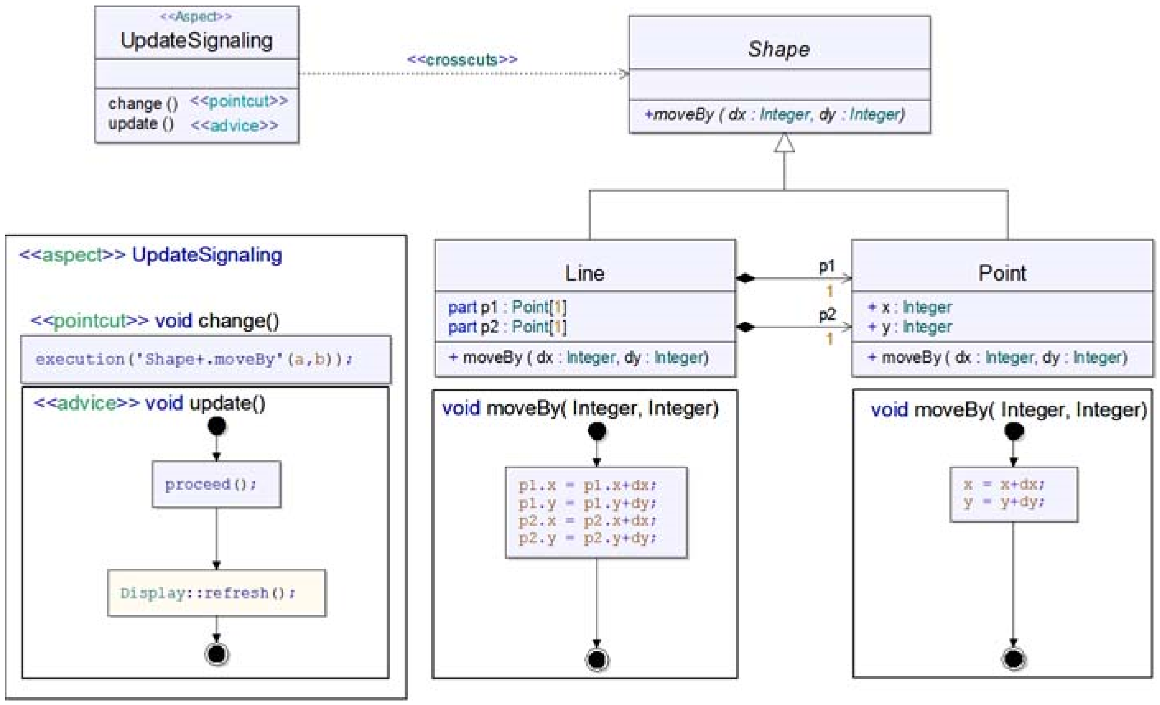


Shape
+moveBy (dx : Integer, dy : Integer)
Line
part p1 : Point[1]
part p2 : Point[1]
+ moveBy (dx : Integer, dy : Integer)
p1
1
p2
1
Point
+ x : Integer
+ y : Integer
+ moveBy (dx : Integer, dy : Integer)
void moveBy(Integer, Integer)
void moveBy(Integer, Integer)

```
p1.x = p1.x+dx;
p1.y = p1.y+dy;
p2.x = p2.x+dx;
p2.y = p2.y+dy;
Display::refresh();
```

```
x = x+dx;
y = y+dy;
Display::refresh();
```

Figure 6 — Figure Editor in UML



<<Aspect>>
UpdateSignaling
change () <<pointcut>>
update () <<advice>>
<<crosscuts>>
Shape
+moveBy (dx : Integer, dy : Integer)
<<aspect>> UpdateSignaling
<<pointcut>> void change()
<<advice>> void update()
Line
part p1 : Point[1]
part p2 : Point[1]
+ moveBy (dx : Integer, dy : Integer)
p1
1
p2
1
Point
+ x : Integer
+ y : Integer
+ moveBy (dx : Integer, dy : Integer)
void moveBy(Integer, Integer)
void moveBy(Integer, Integer)

```
execution('Shape+.moveBy'(a,b));
proceed();
Display::refresh();
```

```
p1.x = p1.x+dx;
p1.y = p1.y+dy;
p2.x = p2.x+dx;
p2.y = p2.y+dy;
```

```
x = x+dx;
y = y+dy;
```

Figure 7 — Aspect-Oriented Figure Editor in UML

Figure 6 illustrates a classic example of a crosscutting concern in a figure editor example taken from the AOSD literature. The example describes an abstract Shape class that can be moved in the editor. Whenever a shape is moved, the display needs to be refreshed. Figure 6 also depicts two Shape subclasses, Line and Point that implement the Shape functionality. The display refresh concern is scattered across the implementation of both subclasses. Figure 7 represents an aspect-oriented implementation of the same system, where an aspect encapsulates the display updating functionality.

The move pointcut descriptor of Figure 7 captures all the executions of the moveBy methods of a subclass of Shape and invokes the display refresh functionality after the execution proceeds. The concern is modularized, which makes it easier to evolve and maintain.

Aspect-oriented requirement engineering

Aspect-oriented requirement Engineering (also referred to as "Early Aspects") focuses on the identification, specification and representation of crosscutting properties at the requirement level. Examples of such properties include security, mobility, availability and real-time constraints. Crosscutting properties are requirements, use cases or features that have a broadly-scoped effect on other requirements or architecture components.

Aspect-oriented requirements engineering approaches are techniques that explicitly recognise the importance of clearly addressing both functional and non-functional crosscutting concerns in addition to non-crosscutting ones. Therefore, these approaches focus on systematically and modularly treating, reasoning about, composing and subsequently tracing crosscutting functional and non-functional concerns via suitable abstraction, representation and composition mechanisms tailored to the requirements engineering domain.

Specific areas of excellence under the denominator of AO Requirements Analysis are:

- the aspect-oriented requirements process itself,
- the aspect-oriented requirements notations,
- aspect-oriented requirements tool support,
- adoption and integration of aspect-oriented requirements engineering, and
- assessment/evaluation of aspect-oriented requirements.

Aspect-oriented system architecture

Aspect-oriented system architecture focuses on the localization and specification of crosscutting concerns in architectural designs. Crosscutting concerns that appear at the architectural level cannot be modularized by redefining the software architecture using conventional architectural abstractions. Aspect-oriented system architecture languages propose explicit mechanisms to identify, specify and evaluate aspects at the architecture design level.

Aspect-oriented architecture starts from the observation that we need to identify, specify and evaluate aspects explicitly at the architecture design level. Aspectual architecture approaches describe steps for identifying architectural aspects. This information is used to redesign a given architecture in which the architectural aspects are made explicit. In this regard, specific areas of excellence are:

- the aspect-oriented architecture process itself,
- the aspect-oriented architecture notations,
- aspect-oriented architecture tool support,
- adoption and integration of aspect-oriented architecture, and
- assessment/evaluation of aspect-oriented architecture.

Aspect-oriented modeling and design

Aspect-oriented design has the same objectives as any software design activity, i.e. characterising and specifying the behavior and structure of the software system. Its unique contribution to software design lies in the fact that concerns that are necessarily scattered and tangled in more traditional approaches can be modularized. Typically, such an approach includes both a process and a language. The process takes as input requirements and produces a design model. The produced design model represents separate concerns and their relationships. The language provides constructs that can describe the elements to be represented in the design and the relationships that can exist between those elements. In particular, constructs are provided to support concern modularization and the specification of concern composition, with consideration for conflicts. Beyond that, the design of each individual modularized concern compares to standard software design.

Here, specific areas of excellence areas are:

- the aspect-oriented design process itself,
- the aspect-oriented design notations,
- aspect-oriented design tool support,
- adoption and integration of aspect-oriented design, and
- assessment/evaluation of aspect-oriented design.

Aspect-Oriented Programming (AOP)

AOP includes programming techniques and tools that support the modularisation of concerns at the level of the source code.

Just like any other programming language, an aspect-oriented language typically consists of two parts: a language specification and an implementation. Hence, there are two corresponding areas of excellence: support for language developers and support for application developers.

Support for application developers

An aspect-oriented approach supports the implementation of concerns and how to compose those independently implemented concerns. While the specification of such a language is the primary manual for application developers, it provides obviously no guarantee that the application developer will produce high-quality aspect-oriented programs. Specific areas of excellence:

- the crucial concepts of aspect-oriented programming,
- programming in aspect-oriented languages,
- composing software components written in any language using aspect-oriented composition mechanisms, or
- aspect-oriented programming environments.

Support for language developers

Excellence on support for constructing aspect languages includes the following areas:

- constructing languages or teels for specific domains and/or platforms, and
- transferring implementation principles of aspect-oriented execution environments, including
    - interpreters,
    - compilers, and
    - virtual machines.

Formal method support for aspect-orientation

Formal methods can be used both to define aspects semantically and to analyze and verify aspect-oriented systems. Aspect-oriented programming extends programming notations with aspect modules that isolate the declaration of when the aspect should be applied (join points) and what actions should be taken when it is reached (advice). Expertise in formal semantic definitions of aspect constructs is useful for language designers to provide a deep understanding of the differences among constructs. Aspects potentially can harm the reliability of a system to which they are woven, and could invalidate essential properties that already were true of the system without the aspect. It is also necessary to show that they actually do add intended crosscutting properties to the system. Hence, numerous questions of correctness and verification are raised by aspect languages. Among the kinds of expertise are:

- specially designed testing techniques to provide coverage for aspects,
- program slicing and code analysis approaches to identify interactions among aspects and between aspects and underlying systems,
- model checking techniques specialized for aspects, and
- inductive techniques to verify aspect-oriented systems.

Each of the above approaches can be used to

- specify and analyze individual aspects relative to an existing system,
- define conditions for composing multiple aspects correctly, and
- detect and resolve potential interferences among aspects.

Although some approaches are already used in aspect languages, others are still subject of research and are not ready for routine industrial application. Nevertheless, awareness of these issues is essential for language designers, and for effective use of aspects, especially in safety-critical contexts.

Aspect-oriented middleware

Middleware and AOSD strongly complement each other. In general, areas of excellence consist of

- support for the application developer, which includes
  - the crucial concepts of aspect supporting middleware,
  - aspect-oriented software development using a specific middleware, involving the aspect programming model, aspect deployment model, platform infrastructure, and services of the middleware, and
- Product Family Engineering (methods, architectures, techniques) in distributed and ambient computing, and
- support for the middleware developer with respect to
  - host-infrastructure middleware,
  - distribution middleware,
  - common middleware services, and
  - domain-specific middleware services.

Chapter 16

Fork (Software Development)

In software engineering, a **project fork** happens when developers take a legal copy of source code from one software package and start independent development on it, creating a distinct piece of software.

Free and open source software is that which, by definition, may be forked from the original development team without prior permission without violating any copyright law. However, licensed forks of proprietary software (*e.g.* Unix) also happen.

Etymology

The term fork comes from the POSIX standard for portable operating systems. Here, `fork()` is a system call which a process uses to create a copy of itself. After the fork two copies of the process exist: a parent and a child. The two then run separately and may then do completely different things.

The independent development of two software projects, one cloned from the other's source code, is analogous to Unix forking.

Branching

A kind of internal fork that is standard practice for the development of many software projects is a stable or release branch, modified only for bug fixes, while a development branch develops new features. Such internal forks are usually referred to as "branches".

Forking free and open source software

Free and open source software may be legally forked without the approval of those currently managing a software project or distributing the software, per the definitions of "free software" copyright license ("Freedom 3: The freedom to improve the program, and release your improvements to the public, so that the whole community benefits") and "open source" ("3. Derived Works: redistribution of modifications must be allowed. (To allow legal sharing and to permit new features or repairs.)").

In free software, forks often result from a schism over different goals or personality clashes. In a fork, both parties assume nearly identical code bases but typically only the larger group, or whoever controls the web site, will retain the full original name and the associated user community. Thus there is a reputation penalty associated with forking. The relationship between the different teams can be cordial or very bitter.

Forks are considered an expression of the freedom made available by free and open source software, but a weakness since they duplicate development efforts and can confuse users over which forked package to use. Developers have the option to collaborate and pool resources with free and open source software software, but it is not ensured by free software licenses, only by a commitment to cooperation.

Eric Raymond, in his seminal essay *The Cathedral and the Bazaar*, stated in 1997 that "The most important characteristic of a fork is that it spawns competing projects that cannot later exchange code, splitting the potential developer community". However, this is not common present usage.

In some cases, a fork can merge back into the original project or replace it. EGCS (the Experimental/Enhanced GNU Compiler System) was a fork from GCC which proved more vital than the original project and was eventually "blessed" as the official GCC project. Some have attempted to invoke this effect deliberately, *e.g.*, Mozilla Firefox started as an unofficial project within Mozilla that soon replaced the Mozilla Suite as the focus of development.

On the matter of forking, the Jargon File says:

Forking is considered a Bad Thing—not merely because it implies a lot of wasted effort in the future, but because forks tend to be accompanied by a great deal of strife and acrimony between the successor groups over issues of legitimacy, succession, and design direction. There is serious social pressure against forking. As a result, major forks (such as the Gnu-Emacs/XEmacs split, the fissioning of the 386BSD group into three daughter projects, and the short-lived GCC/EGCS split) are rare enough that they are remembered individually in hacker folklore.

It is easy to declare a fork, but can require considerable effort to continue independent development and support. As such, forks without adequate resources can soon become inactive, *e.g.*, GoneME, a fork of GNOME by a former developer, which was soon discontinued despite attracting some publicity. Some well-known forks have enjoyed great success, however, such as the X.Org X11 server, a fork from XFree86 which gained widespread support from developers and users and notably sped up X development.

More recently, the use of DVCS tools has made the term 'fork' less emotive. With a DVCS such as Mercurial or Git, the normal way to contribute to a project is to first 'fork' the repository, and later seek to have your changes integrated with the main repository. These tools have been designed to make creating, maintaining and merging branches (internal forks) much easier than with a centralised VCS, and in so doing they eliminate the difference between a branch and a fork from the point of view of the VCS tool. In addition, sites such as Github, Bitbucket and Launchpad provide free DVCS hosting with very easy-to-use support for this kind of forking, so that the technical, social and financial barriers to forking a source code repository are massively reduced. While forking the community necessarily remains costly and painful, having many competing forks of the source code has become a more natural and accepted part of the development process.

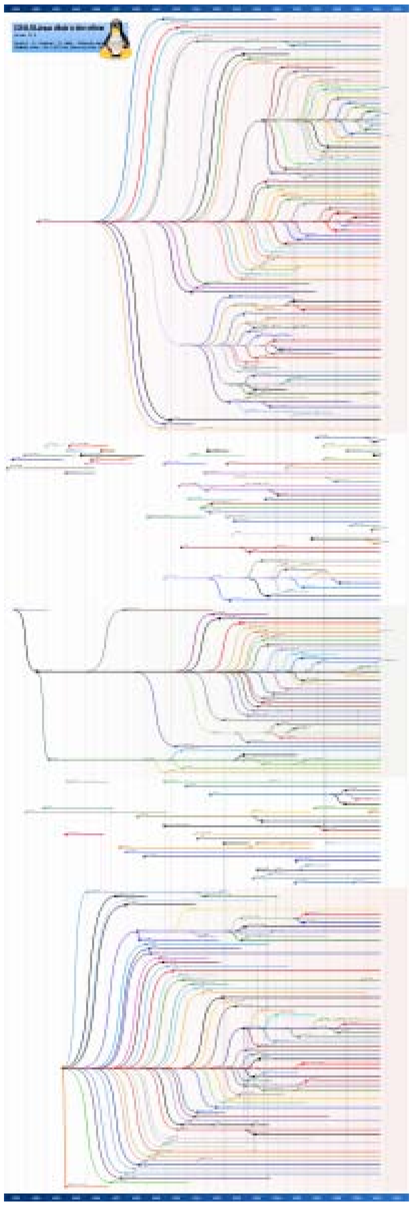
Forking proprietary software

In proprietary software, the copyright is usually held by the employing entity, not by the individual software developers. Proprietary code is thus more commonly forked when the owner needs to develop two or more versions, such as a windowed version and a command line version, or versions for differing operating systems, such as a wordprocessor for IBM PC compatible machines and Macintosh computers. Generally, such internal forks will concentrate on having the same look, feel, data format, and behavior between platforms so that a user familiar with one can also be productive or share documents generated on the other. This is almost always an economic decision to generate a greater market share and thus pay back the associated extra development costs created by the fork.

A notable proprietary fork not of this kind is the many varieties of proprietary Unix — all derived from AT&T Unix and all called "Unix", but increasingly mutually incompatible.

The BSD licenses permit forks to become proprietary software, and some say that commercial incentives thus make proprietisation almost inevitable. Examples include Mac OS X (based on Nextstep and thus BSD), Cedega and CrossOver (proprietary forks of Wine, though CrossOver tracks Wine and contributes considerably), EnterpriseDB (a fork of PostgreSQL, adding Oracle compatibility features), Fujitsu Supported PostgreSQL with their proprietary ESM storage system, and Netezza's proprietary highly scalable derivative of PostgreSQL. Some of these vendors contribute back changes to the community project, while some keep their changes as their own competitive advantages.

Other notable forks



A timeline chart of how Linux distributions forked.

- Most Linux distributions are descended from other distributions, most being traceable back to Debian, Red Hat or Slackware. Since most of the content of a distribution is free and open source software, ideas and software interchange freely as is useful to the individual distribution. Merges (*e.g.*, United Linux or Mandriva) are rare.
- Pretty Good Privacy was forked outside of the United States to free it from restrictive US laws on the exportation of cryptographic software.
- The game *NetHack* has spawned a number of variants using the original code, notably *Slash'EM*, and was itself a fork of *Hack*.
- OpenBSD was a fork of NetBSD 1.0 by Theo de Raadt

- OpenSSH was a fork from SSH, which happened because the license for SSH 2.x was non-free (even though the source was available), so an older version of SSH 1.x, the last to have been licensed as free software, was forked. Within months, virtually all Linux distributions, BSD versions and even some proprietary Unixes had replaced SSH with OpenSSH.
- XOrg forked from XFree86 in 2004, due to the latter's change to a license many distributors found unacceptable.
- DragonFly BSD was forked from FreeBSD 4.8 by long-time FreeBSD developer Matt Dillon, due to disagreement over FreeBSD 5's technical direction.
- Adempiere is a community maintained fork of Compiere 2.5.3b, due to disagreement with commercial and technical direction of Compiere Inc.
- NeoOffice is a fork of OpenOffice.org, with an incompatible license (GPL rather than LGPL), due to disagreements about licensing and about the best method to port OpenOffice.org to Mac OS X.
- Joomla! is a fork of Mambo.
- The Inkscape vector-graphics program started as a fork of Sodipodi.
- Xvid was a fork of OpenDivX.
- Boxee's GUI and media player software was forked from XBMC media center.
- LibreOffice is a fork of OpenOffice.org that resulted from Oracle's purchase of Sun Microsystems.

Chapter 17

Software Quality

In the context of software engineering, **software quality** measures how well software is designed (*quality of design*), and how well the software conforms to that design (*quality of conformance*), although there are several different definitions, including conformance to customer expectectations. It is often described as the 'fitness for purpose' of a piece of software.

Whereas *quality of conformance* is concerned with implementation, *quality of design* measures how valid the design and requirements are in creating a worthwhile product.

Definition

One of the challenges of software quality is that "everyone feels they understand it".

In addition to more software specific definitions given below, there are several applicable definitions of quality which are used in business.Quality_(business)#Definitions

Software quality may be defined as conformance to explicitly stated functional and performance requirements, explicitly documented development standards and implicit characteristics that are expected of all professionally developed software.

The three key points in this definition:

1. Software requirements are the foundations from which quality is measured.

   Lack of conformance to requirement is lack of quality.

2. Specified standards define a set of development criteria that guide the management in software engineering.

   If criteria are not followed lack of quality will usually result.

3. A set of implicit requirements often goes unmentioned, for example ease of use, maintainability etc.

If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspected.

A definition in Steve McConnell's *Code Complete* divides software into two pieces: **internal** and **external quality characteristics**. External quality characteristics are those parts of a product that face its users, where internal quality characteristics are those that do not.

Another definition by Dr. Tom DeMarco says "a product's quality is a function of how much it changes the world for the better." This can be interpreted as meaning that user satisfaction is more important than anything in determining software quality.

Another definition, coined by Gerald Weinberg in *Quality Software Management: Systems Thinking*, is "Quality is value to some person." This definition stresses that quality is inherently subjective - different people will experience the quality of the same software very differently. One strength of this definition is the questions it invites software teams to consider, such as "Who are the people we want to value our software?" and "What will be valuable to *them*?"

History

Software product quality

- Product quality
    - conformance to requirements or program specification; related to Reliability
- Scalability
- Correctness
- Completeness
- Absence of bugs
- Fault-tolerance
    - Extensibility
    - Maintainability
- Documentation

The Consortium for IT Software Quality (CISQ) was launched in 2009 to standardize the measurement of software product quality. The Consortium's goal is to bring together industry executives from Global 2000 IT organizations, system integrators, outsourcers, and package vendors to jointly address the challenge of standardizing the measurement of IT software quality and to promote a market-based ecosystem to support its deployment.

It is essential to supplement traditional testing – functional, non-functional, and run-time – with measures of application structural quality. Structural quality is the quality of the application’s architecture and the degree to which its implementation accords with software engineering best practices. Industry data demonstrate that poor application structural quality results in cost and schedule overruns and creates waste in the form of rework (up to 45% of development time in some organizations). Moreover, poor structural quality is strongly correlated with high-impact business disruptions due to corrupted data, application outages, security breaches, and performance problems. As in any other field of engineering, an application with good structural software quality costs less to maintain and is easier to understand and change in response to pressing business needs.

Source code quality

A computer has no concept of "well-written" source code. However, from a human point of view source code can be written in a way that has an effect on the effort needed to comprehend its behavior. Many source code programming style guides, which often stress readability and usually language-specific conventions are aimed at reducing the cost of source code maintenance. Some of the issues that affect code quality include:

- Readability
- Ease of maintenance, testing, debugging, fixing, modification and portability
- Low complexity
- Low resource consumption: memory, CPU
- Number of compilation or lint warnings
- Robust input validation and error handling, established by software fault injection

Methods to improve the quality:

- Refactoring
- Code Inspection or software review
- Documenting the code

Software reliability

Software **reliability** is an important facet of software quality. It is defined as "the probability of failure-free operation of a computer program in a specified environment for a specified time".

One of reliability's distinguishing characteristics is that it is objective, measurable, and can be estimated, whereas much of software quality is subjective criteria. This distinction is especially important in the discipline of Software Quality Assurance. These measured criteria are typically called software metrics.

History

With software embedded into many devices today, software failure has caused more than inconvenience. Software errors have even caused human fatalities. The causes have ranged from poorly designed user interfaces to direct programming errors. An example of a programming error that lead to multiple deaths is discussed in Dr. Leveson's paper (PDF). This has resulted in requirements for development of some types software. In the

United States, both the Food and Drug Administration (FDA) and Federal Aviation Administration (FAA) have requirements for software development.

Goal of reliability

The need for a means to objectively determine software reliability comes from the desire to apply the techniques of contemporary engineering fields to the development of software. That desire is a result of the common observation, by both lay-persons and specialists, that computer software does not work the way it ought to. In other words, software is seen to exhibit undesirable behaviour, up to and including outright failure, with consequences for the data which is processed, the machinery on which the software runs, and by extension the people and materials which those machines might negatively affect. The more critical the application of the software to economic and production processes, or to life-sustaining systems, the more important is the need to assess the software's reliability.

Regardless of the criticality of any single software application, it is also more and more frequently observed that software has penetrated deeply into most every aspect of modern life through the technology we use. It is only expected that this infiltration will continue, along with an accompanying dependency on the software by the systems which maintain our society. As software becomes more and more crucial to the operation of the systems on which we depend, the argument goes, it only follows that the software should offer a concomitant level of dependability. In other words, the software should behave in the way it is intended, or even better, in the way it should.

Challenge of reliability

The circular logic of the preceding sentence is not accidental—it is meant to illustrate a fundamental problem in the issue of measuring software reliability, which is the difficulty of determining, in advance, exactly how the software is intended to operate. The problem seems to stem from a common conceptual error in the consideration of software, which is that software in some sense takes on a role which would otherwise be filled by a human being. This is a problem on two levels. Firstly, most modern software performs work which a human could never perform, especially at the high level of reliability that is often expected from software in comparison to humans. Secondly, software is fundamentally incapable of most of the mental capabilities of humans which separate them from mere mechanisms: qualities such as adaptability, general-purpose knowledge, a sense of conceptual and functional context, and common sense.

Nevertheless, most software programs could safely be considered to have a particular, even singular purpose. If the possibility can be allowed that said purpose can be well or even completely defined, it should present a means for at least considering objectively whether the software is, in fact, reliable, by comparing the expected outcome to the actual outcome of running the software in a given environment, with given data. Unfortunately, it is still not known whether it is possible to exhaustively determine either the expected outcome or the actual outcome of the entire set of possible environment and input data to a given program, without which it is probably impossible to determine the program's reliability with any certainty.

However, various attempts are in the works to attempt to rein in the vastness of the space of software's environmental and input variables, both for actual programs and theoretical descriptions of programs. Such attempts to improve software reliability can be applied at different stages of a program's development, in the case of real software. These stages principally include: requirements, design, programming, testing, and runtime evaluation. The study of theoretical software reliability is predominantly concerned with the concept of correctness, a mathematical field of computer science which is an outgrowth of language and automata theory.

Reliability in program development

Requirements

A program cannot be expected to work as desired if the developers of the program do not, in fact, know the program's desired behaviour in advance, or if they cannot at least determine its desired behaviour in parallel with development, in sufficient detail. What level of detail is considered sufficient is hotly debated. The idea of perfect detail is attractive, but may be impractical, if not actually impossible. This is because the desired behaviour tends to change as the possible range of the behaviour is determined through actual attempts, or more accurately, failed attempts, to achieve it.

Whether a program's desired behaviour can be successfully specified in advance is a moot point if the behaviour cannot be specified at all, and this is the focus of attempts to formalize the process of creating requirements for new software projects. In situ with the formalization effort is an attempt to help inform non-specialists, particularly non-programmers, who commission software projects without sufficient knowledge of what computer software is in fact capable. Communicating this knowledge is made more difficult by the fact that, as hinted above, even programmers cannot always know in advance what is actually possible for software in advance of trying.

Design

While requirements are meant to specify what a program should do, design is meant, at least at a high level, to specify how the program should do it. The usefulness of design is also questioned by some, but those who look to formalize the process of ensuring reliability often offer good software design processes as the most significant means to accomplish it. Software design usually involves the use of more abstract and general means of specifying the parts of the software and what they do. As such, it can be seen as a way to break a large program down into many smaller programs, such that those smaller pieces together do the work of the whole program.

The purposes of high-level design are as follows. It separates what are considered to be problems of architecture, or overall program concept and structure, from problems of actual coding, which solve problems of actual data processing. It applies additional constraints to the development process by narrowing the scope of the smaller software components, and thereby—it is hoped—removing variables which could increase the likelihood of programming errors. It provides a program template, including the specification of interfaces, which can be shared by different teams of developers working on disparate parts, such that they can know in advance how each of their contributions will interface with those of the other teams. Finally, and perhaps most controversially, it specifies the program independently of the implementation language or languages, thereby removing language-specific biases and limitations which would otherwise creep into the design, perhaps unwittingly on the part of programmer-designers.

Programming

The history of computer programming language development can often be best understood in the light of attempts to master the complexity of computer programs, which otherwise becomes more difficult to understand in proportion (perhaps exponentially) to the size of the programs. (Another way of looking at the evolution of programming languages is simply as a way of getting the computer to do more and more of the work, but this may be a different way of saying the same thing). Lack of understanding of a program's overall structure and functionality is a sure way to fail to detect errors in the program, and thus the use of better languages should, conversely, reduce the number of errors by enabling a better understanding.

Improvements in languages tend to provide incrementally what software design has attempted to do in one fell swoop: consider the software at ever greater levels of abstraction. Such inventions as statement, sub-routine, file, class, template, library, component and more have allowed the arrangement of a program's parts to be specified using abstractions such as layers, hierarchies and modules, which provide structure at different granularities, so that from any point of view the program's code can be imagined to be orderly and comprehensible.

In addition, improvements in languages have enabled more exact control over the shape and use of data elements, culminating in the abstract data type. These data types can be specified to a very fine degree, including how and when they are accessed, and even the state of the data before and after it is accessed..

Software Build and Deployment

Many programming languages such as C and Java require the program "source code" to be translated in to a form that can be executed by a computer. This translation is done by a program called a compiler. Additional operations may be involved to associate, bind, link or package files together in order to create a usable runtime configuration of the software application. The totality of the compiling and assembly process is generically called "building" the software.

The software build is critical to software quality because if any of the generated files are incorrect the software build is likely to fail. And, if the incorrect version of a program is inadvertently used, then testing can lead to false results.

Software builds are typically done in work area unrelated to the runtime area, such as the application server. For this reason, a deployment step is needed to physically transfer the software build products to the runtime area. The deployment procedure may also involve technical parameters, which, if set incorrectly, can also prevent software testing from beginning. For example, a Java application server may have options for parent-first or parent-last class loading. Using the incorrect parameter can cause the application to fail to execute on the application server.

The technical activities supporting software quality including build, deployment, change control and reporting are collectively known as Software configuration management. A number of software tools have arisen to help meet the challenges of configuration management including file control tools and build control tools.

Testing

Software testing, when done correctly, can increase overall software *quality of conformance* by testing that the product conforms to its requirements. Testing includes, but is not limited to:

1. Unit Testing
2. Functional Testing
3. Regression Testing
4. Performance Testing
5. Failover Testing
6. Usability Testing

A number of agile methodologies use testing early in the development cycle to ensure quality in their products. For example, the test-driven development practice, where tests are written before the code they will test, is used in Extreme Programming to ensure quality.

Runtime

Runtime reliability determinations are similar to tests, but go beyond simple confirmation of behaviour to the evaluation of qualities such as performance and interoperability with other code or particular hardware configurations.

Software quality factors

A software quality factor is a non-functional requirement for a software program which is not called up by the customer's contract, but nevertheless is a desirable requirement which enhances the quality of the software program. Note that none of these factors are binary; that is, they are not "either you have it or you don't" traits. Rather, they are characteristics that one seeks to maximize in one's software to optimize its quality. So rather than asking whether a software product "has" factor *x*, ask instead the *degree* to which it does (or does not).

Some software quality factors are listed here:

Understandability
: Clarity of purpose. This goes further than just a statement of purpose; all of the design and user documentation must be clearly written so that it is easily understandable. This is obviously subjective in that the user context must be taken into account: for instance, if the software product is to be used by software engineers it is not required to be understandable to the layman.

Completeness
: Presence of all constituent parts, with each part fully developed. This means that if the code calls a subroutine from an external library, the software package must provide reference to that library and all required parameters must be passed. All required input data must also be available.

Conciseness
: Minimization of excessive or redundant information or processing. This is important where memory capacity is limited, and it is generally considered good practice to keep lines of code to a minimum. It can be improved by replacing repeated functionality by one subroutine or function which achieves that functionality. It also applies to documents.

Portability
: Ability to be run well and easily on multiple computer configurations. Portability can mean both between different hardware—such as running on a PC as well as a smartphone—and between different operating systems—such as running on both Mac OS X and GNU/Linux.

Consistency
: Uniformity in notation, symbology, appearance, and terminology within itself.

Maintainability
: Propensity to facilitate updates to satisfy new requirements. Thus the software product that is maintainable should be well-documented, should not be complex, and should have spare capacity for memory, storage and processor utilization and other resources.

Testability
: Disposition to support acceptance criteria and evaluation of performance. Such a characteristic must be built-in during the design phase if the product is to be easily testable; a complex design leads to poor testability.

Usability
: Convenience and practicality of use. This is affected by such things as the human-computer interface. The component of the software that has most impact on this is the user interface (UI), which for best usability is usually graphical (i.e. a GUI).

Reliability

Ability to be expected to perform its intended functions satisfactorily. This implies a time factor in that a reliable product is expected to perform correctly over a period of time. It also encompasses environmental considerations in that the product is required to perform correctly in whatever conditions it finds itself (sometimes termed robustness).

Efficiency

Fulfillment of purpose without waste of resources, such as memory, space and processor utilization, network bandwidth, time, etc.

Security

Ability to protect data against unauthorized access and to withstand malicious or inadvertent interference with its operations. Besides the presence of appropriate security mechanisms such as authentication, access control and encryption, security also implies resilience in the face of malicious, intelligent and adaptive attackers.

Measurement of software quality factors

There are varied perspectives within the field on measurement. There are a great many measures that are valued by some professionals—or in some contexts, that are decried as harmful by others. Some believe that quantitative measures of software quality are essential. Others believe that contexts where quantitative measures are useful are quite rare, and so prefer qualitative measures. Several leaders in the field of software testing have written about the difficulty of measuring what we truly want to measure well.

One example of a popular metric is the number of faults encountered in the software. Software that contains few faults is considered by some to have higher quality than software that contains many faults. Questions that can help determine the usefulness of this metric in a particular context include:

1. What constitutes “many faults?” Does this differ depending upon the purpose of the software (e.g., blogging software vs. navigational software)? Does this take into account the size and complexity of the software?
2. Does this account for the importance of the bugs (and the importance to the stakeholders of the people those bugs bug)? Does one try to weight this metric by the severity of the fault, or the incidence of users it affects? If so, how? And if not, how does one know that 100 faults discovered is better than 1000?
3. If the count of faults being discovered is shrinking, how do I know what that means? For example, does that mean that the product is now higher quality than it was before? Or that this is a smaller/less ambitious change than before? Or that fewer tester-hours have gone into the project than before? Or that this project was tested by less skilled testers than before? Or that the team has discovered that fewer faults reported is in their interest?

This last question points to an especially difficult one to manage. All software quality metrics are in some sense measures of human behavior, since humans create software. If a team discovers that they will benefit from a drop in the number of reported bugs, there is a strong tendency for the team to start reporting fewer defects. That may mean that email begins to circumvent the bug tracking system, or that four or five bugs get lumped into one bug report, or that testers learn not to report minor annoyances. The difficulty is measuring what we mean to measure, without creating incentives for software programmers and testers to consciously or unconsciously "game" the measurements.

Software quality factors cannot be measured because of their vague definitions. It is necessary to find measurements, or metrics, which can be used to quantify them as non-functional requirements. For example, reliability is a software quality factor, but cannot be evaluated in its own right. However, there are related attributes to reliability, which can indeed be measured. Some such attributes are mean time to failure, rate of failure occurrence, and availability of the system. Similarly, an attribute of portability is the number of target-dependent statements in a program.

A scheme that could be used for evaluating software quality factors is given below. For every characteristic, there are a set of questions which are relevant to that characteristic. Some type of scoring formula could be developed based on the answers to these questions, from which a measurement of the characteristic can be obtained.

Understandability

Are variable names descriptive of the physical or functional property represented? Do uniquely recognisable functions contain adequate comments so that their purpose is clear? Are deviations from forward logical flow adequately commented? Are all elements of an array functionally related?...

Completeness

Are all necessary components available? Does any process fail for lack of resources or programming? Are all potential pathways through the code accounted for, including proper error handling?

Conciseness

Is all code reachable? Is any code redundant? How many statements within loops could be placed outside the loop, thus reducing computation time? Are branch decisions too complex?

Portability

Does the program depend upon system or library routines unique to a particular installation? Have machine-dependent statements been flagged and commented? Has dependency on internal bit representation of alphanumeric or special characters been avoided? How much effort would be required to transfer the program from one hardware/software system or environment to another?

Consistency

Is one variable name used to represent different logical or physical entities in the program? Does the program contain only one representation for any given physical or mathematical constant? Are functionally similar arithmetic expressions similarly constructed? Is a consistent scheme used for indentation, nomenclature, the color palette, fonts and other visual elements?

Maintainability

Has some memory capacity been reserved for future expansion? Is the design cohesive—i.e., does each module have distinct, recognizable functionality? Does the software allow for a change in data structures (object-oriented designs are more likely to allow for this)? If the code is procedure-based (rather than object-oriented), is a change likely to require restructuring the main program, or just a module?

Testability

Are complex structures employed in the code? Does the detailed design contain clear pseudo-code? Is the pseudo-code at a higher level of abstraction than the code? If tasking is used in concurrent designs, are schemes available for providing adequate test cases?

Usability

Is a GUI used? Is there adequate on-line help? Is a user manual provided? Are meaningful error messages provided?

Reliability

Are loop indexes range-tested? Is input data checked for range errors? Is divide-by-zero avoided? Is exception handling provided? It is the probability that the software performs its intended functions correctly in a specified period of time under stated operation conditions, but there could also be a problem with the requirement document...

Efficiency

Have functions been optimized for speed? Have repeatedly used blocks of code been formed into subroutines? Has the program been checked for memory leaks or overflow errors?

Security

Does the software protect itself and its data against unauthorized access and use? Does it allow its operator to enforce security policies? Are security mechanisms appropriate, adequate and correctly implemented? Can the software withstand attacks that can be anticipated in its intended environment?

User's perspective

In addition to the technical qualities of software, the end user's experience also determines the quality of software. This aspect of software quality is called usability. It is hard to quantify the usability of a given software product. Some important questions to be asked are:

- Is the user interface intuitive (self-explanatory/self-documenting)?
- Is it easy to perform simple operations?
- Is it feasible to perform complex operations?
- Does the software give sensible error messages?
- Do widgets behave as expected?
- Is the software well documented?
- Is the user interface responsive or too slow?

Also, the availability of (free or paid) support may factor into the usability of the software.