Software development process

From Wikipedia, the free encyclopedia

A **software development process**
is a structure imposed on the development of a software product. Synonyms include **software lifecycle** 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.

Contents

Software development process

Activities and steps

Requirements | Architecture | Implementation | Testing | Deployment

Models

Agile | Cleanroom | Iterative | RAD | RUP | Spiral | Waterfall | XP

Supporting disciplines

Configuration management | Documentation | Project management | User experience design

Processes and meta-processes

A 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.

The Capability Maturity Model
(CMM) is one of the leading models. Independent assessments grade organizations on how well they follow their defined processes, not on the quality of those processes or the software produced. CMM is gradually replaced by CMMI. ISO 9000 describes standards for formally organizing processes with documentation.

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 CMM and 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.

Six Sigma is a methodology to manage process variations that uses data and statistical analysis to measure and improve a company's operational performance. It works by identifying and eliminating defects in manufacturing and service-related processes. The maximum permissible defects is 3.4 per one million opportunities. However, Six Sigma is manufacturing-oriented and needs further research on its relevance to software development.

Domain Analysis

Often the first step in attempting to design a new piece of software, whether it be an addition to an existing software, a new application, a new subsystem or a whole new system, is, what is generally referred to as

"Domain Analysis". Assuming that the developers (including the analysts) are not sufficiently knowledgeable in the subject area of the new software, the first task is to investigate the so-called "domain" of the software. The more knowledgeable they are about the domain already, the less the work required. Another objective of this work is to make the analysts who will later try to elicit and gather the requirements from the area experts or professionals, speak with them in the domain's own terminology and to better understand what is being said by these people. Otherwise they will not be taken seriously. So, this phase is an important prelude to extracting and gathering the requirements. The following quote captures the kind of situation an analyst who hasn't done his homework well may face in speaking with a professional from the domain: *"I know you believe you understood what you think I said, but I am not sure you realize what you heard is not what I meant."*[1]

Software Elements Analysis

The most important task in creating a software product is extracting the requirements. Customers typically know what they want, but not what software should do, while incomplete, ambiguous or contradictory requirements are recognized by skilled and experienced software engineers. Frequently demonstrating live code may help reduce the risk that the requirements are incorrect.

Specification

Specification is the task of precisely describing the software to be written, possibly in a rigorous way. In practice, most successful specifications are written to understand and fine-tune applications that were already well-developed, although safety-critical software systems are often carefully specified prior to application development. *Specifications are most important for external interfaces that must remain stable.*

Software architecture

The architecture of a software system refers to an abstract representation of that system. Architecture is concerned with making sure the software system will meet the requirements of the product, as well as ensuring that future requirements can be addressed. The architecture step also addresses interfaces between the software system and other software products, as well as the underlying hardware or the host operating system.

Implementation (or coding)

Reducing a design to code may be the most obvious part of the software engineering job, but it is not necessarily the largest portion.

Testing

Testing of parts of software, especially where code by two different engineers must work together, falls to the software engineer.

Documentation

An important (and often overlooked) task is documenting the internal design of software for the purpose of future maintenance and enhancement. *Documentation is most important for external interfaces.*

Software Training and Support

A large percentage of software projects fail because the developers fail to realize that it doesn't matter how much time and planning a development team puts into creating software if nobody in an organization ends up using it. People are occasionally 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 the most enthusiastic software users (build excitement and confidence), shifting the training towards the neutral users intermixed with the avid supporters, and finally incorporate the rest of the organization into adopting the new software. Users will have lots of questions and software problems which leads to the next phase of software.

Maintenance

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. Not only may it be necessary to add code that does not fit the original design but just determining how software works at some point after it is completed may require significant effort by a software engineer. About ⅔ of all software engineering work is maintenance, but this statistic can be misleading. A small part of that is fixing bugs. Most maintenance is extending systems to do new

things, which in many ways can be considered new work. In comparison, about ⅔ of all civil engineering, architecture, and construction work is maintenance in a similar way.

Process models

A decades-long goal has been to find repeatable, predictable processes or methodologies 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 is proving difficult.

Waterfall processes

The best-known and oldest process is the waterfall model, where developers (roughly) follow these steps in order:

- state requirements
- analyze requirements
- design a solution approach
- architect a software framework for that solution
- develop code
- test (perhaps unit tests then system tests)
- deploy, and
- Post Implementation.

After each step is finished, the process proceeds to the next step, just as builders don't revise the foundation of a house after the framing has been erected.

There is a misconception that the process has no provision for correcting errors in early steps (for example, in the requirements). In fact this is where the domain of requirements management comes in which includes change control.

This approach is used in high risk projects, particularly large defense contracts. The problems in waterfall do not arise from "immature engineering practices, particularly in requirements analysis and requirements management." Studies of the failure rate of the DOD-STD-2167
specification, which enforced waterfall, have shown that the more closely a project follows its process, specifically in up-front requirements gathering, the more likely the project is to release features that are not used in their current form.

More often too the supposed stages are part of joint review between customer and supplier, the supplier can, in fact, develop at risk and evolve the design but must sell off the design at a key milestone called Critical Design Review (CDR). This shifts engineering burdens from engineers to customers who may have other skills.

Iterative processes

Iterative development [1] (http://doi.ieeecomputersociety.org/10.1109/MC.2003.1204375) 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 software development
processes are built on the foundation of iterative development. To that foundation they add a lighter, 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.

Agile processes seem to be more efficient than older methodologies, using less programmer time to produce more functional, higher quality software, but have the drawback from a business perspective that they do not provide long-term planning capability. In essence, the Agile approach claims it will provide the most bang for the buck, but won't say exactly when that bang will be or how big a buck will ultimately be required.

Extreme Programming, XP, is the best-known iterative process. In 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.

While Iterative development approaches have their advantages, software architects are still faced with the challenge of creating a reliable foundation upon which to develop. Such a foundation often requires a fair amount of upfront analysis and prototyping to build a development model. The development model often relies upon specific design patterns and entity relationship diagrams
(ERD). Without this upfront foundation, Iterative development can create long term challenges that are significant in terms of cost and quality.

Critics of iterative development approaches point out that these processes place what may be an unreasonable expectation upon the recipient of the software: that they must possess the skills and experience of a seasoned software developer. The approach can also be very expensive if iterations are not small enough to mitigate risk; akin to... "If you don't know what kind of house you want, let me build you one and see if you like it. If you don't, we'll tear it all down and start over." By analogy the critic argues that up-front design is as necessary for software development as it is for architecture. The problem with this criticism is that the whole point of iterative programming is that you don't have to build the whole house before you get feedback from the recipient. Indeed, in a sense conventional programming places more of this burden on the recipient, as the requirements and planning phases take place entirely before the development begins, and testing only occurs after development is officially over.

In fact, a relatively quiet turn around in the Agile community has occurred on the notion of "evolving" the software without the requirements locked down. In the old world this was called requirements creep and never made commercial sense. The Agile community has similarly been "burnt" because, in the end, when the customer asks for something that breaks the architecture, and won't pay for the re-work, the project terminates in an Agile manner.

These approaches have been developed along with web based technologies. As such, they are actually more akin to maintenance life cycles given that most of the architecture and capability of the solutions is embodied within the technology selected as the back bone of the application.

Refactoring is claimed, by the Agile community, as their alternative to cogitating and documenting a design. No equivalent claim is made of re-engineering - which is an artifact of the wrong technology being chosen, therefore the wrong architecture. Both are relatively costly. Claims that 10%-15% must be added to an iteration to account for refactoring of old code exist. However, there is no detail as to whether this value accounts for the re-testing or regression testing that must happen where old code is touched. Of course, throwing away the architecture is more costly again. In fact, a survey of the "designless" approach paints a picture of the cost incurred where this class of approach is used (Software Development at Microsoft Observed
(ftp://ftp.research.microsoft.com/pub/tr/TR-2005-140.pdf)). Note the heavy emphasis here on constant reverse engineering by programming staff rather than managing a central design.

Test Driven Development
(TDD) is a useful output of the Agile camp but raises a conundrum. TDD requires that a unit test be written for a class

before the class is written. Therefore, the class firstly has to be "discovered" and secondly defined in sufficient detail to allow the write-test-once-and-code-until-class-passes model that TDD actually uses. This is actually counter to Agile approaches, particularly (so-called) Agile Modeling, where developers are still encouraged to code early, with light design. Obviously to get the claimed benefits of TDD a full design down to class and responsibilities (captured using, for example, Design By Contract) is necessary. This counts towards iterative development, with a design locked down, but not iterative design - as heavy refactoring and re-engineering negate the usefulness of TDD.

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, 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 (see virtual finite state machine or event driven finite state machine).

Formal methods are most likely to be applied in avionics software, particularly where the software is safety critical. Software safety assurance standards, such as DO 178B 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 OCL (and specializations such as JML) and especially with MDA 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.

See also

Some software development methods:

- Waterfall model
- Spiral model
- Model driven development
- User experience
- Top-down and bottom-up design
- Chaos model
- Evolutionary prototyping
- Prototyping
- ICONIX Process (UML-based object modeling with use cases)
- Unified Process
- V-model
- Extreme Programming
- Software Development Rhythms
- Hysterical raisins, or hard-to-explain features maintained for backward compatibility

Related subjects:

- Rapid application development
- Software development
- Software Estimation
- Abstract Model
- Development stage
- IPO+S Model
- List of software engineering topics
- Performance engineering
- Process
- Programming paradigm
- Programming productivity
- Project
- Systems Development Life Cycle (SDLC)
- Software documentation
- Systems design
- List of software development philosophies
- Test effort
- Best Coding Practices

References

1. ^ Appears in Roger S. Pressman, *Software Engineering (A practitioner's approach)*, 5th edition, 2000, Mc Graw-Hill Education, ISBN 978-0071181822; however the quote is attributed to many sources, including Richard Nixon, Robert McCloskey, and Alan Greenspan. It may have originated several decades earlier in an anonymous academic joke.

External links

- Frederick P. Brooks, Jr., "No Silver Bullet: Essence and Accidents of Software Engineering", 1986
- Gerhard Fischer,
"The Software Technology of the 21st Century: From Software Reuse to Collaborative Software Design" (http://l3d.cs.colorado.edu/~gerhard/papers/isfst2001.pdf) , 2001
- Lydia Ash: *The Web Testing Companion: The Insider's Guide to Efficient and Effective Tests*, Wiley, May 2, 2003. ISBN 0471430218
- Software development life cycle (SDLC) [visual image], *software development life cycle* (http://www.notetech.com/images/software_lifecycle.jpg)
- Iterative Development and The Leaning Tower of Pisa (http://www.fromthetrench.com/2007/01/21/iterative-development-and-the-leaning-tower-of-pisa/) - From The Trench (http://www.fromthetrench.com/)