Software Engineering

"The amateur software engineer is always in search of magic, some sensational method or tool whose application promises to render software development trivial. It is the mark of the professional software engineer to know that no such panacea exists." - Grady Booch.

Introduction.

Software Engineering (SE), a phrase that originated in 1968, came about as a result of trying to describe the art of developing high quality software under given constraints. Many of the problems that existed back then, such as over budgeting, going beyond agreed upon time constraints and not meeting the customer's satisfaction, are still the same today, unfortunately. The research involved in Computer Science, which underpins SE, is mostly empirical and case study based and is still in its infancy. Since SE does not have a robust scientific underpinning, it is still a volatile process and is driven by heuristics. One of the biggest open-ended questions in SE is *how does the software process relate to the software product and vice versa?*

Regardless of these deficiencies, SE is still an important activity. SE mainly consists of modeling, problem solving, knowledge acquisition and rationale-driven activities. The SE process consists of three principle elements: *methodology*, *technology*, and of course, we the *people*. This document is a description of SE in a nutshell from an OO perspective, with the hope of conveying the main ideas and concepts. Moreover, it is primarily based on Jacobson's OOSE work, which originated in 1992, via Bruegge and Dutoit [C1]. Although, the best way to understand SE would be from iterations of formal education and experience. As you go through this document, keep in mind that SE encompasses a myriad of activities for trying to construct software in the general sense, and this document is by no means comprehensive. Moreover, not all software development processes are constructed the same way, thus it is prudent to know/have a bag of procedures/tools/methodologies to resort to in case of need.

"Software Engineering is the collective term applied to attempts to produce high quality, complex, and large software on a largely methodical and reasonably sustainable basis with an increasingly scientific and qualitative orientation." - Houman Younessi

SE Phases.

Although, the following phases may seem to be a 1-2-3 approach as in the Waterfall Model, they can be traversed in any direction (up or down) and even out of order any number of times depending on the complexity of the problem and solution. For instance, if there is an issue in the design phase that requires more domain knowledge then requirements can still be gathered to clarify the issue. This is the iterative nature of dealing with *complexity* and *change*. However, without loss of generality, the order prescribed below is only guideline.

1. **Project Agreement**
2. **Requirements & Analysis**
3. **Design**
4. **Implementation**
5. **Testing**
6. **Maintenance**

Project Agreement

Project Agreement (PA) is primarily a management activity, which defines and examines the feasibility of the scope, duration, deliverables, costs and risks. The PA can be in one of many forms, e.g., a business plan, a technical plan or even a contract. The client either formally accepts the PA or rejects it. Upon acceptance, management then moves to the project planning stage. During this stage project activities, tasks and high-level goals are set up, risk assessment is analyzed in more detail, and resources such as labor, time and hardware are assigned to work tasks.

Requirements & Analysis

"It has often been observed that we more frequently fail to face the right problem than fail to solve the problem we face." - Russell Ackoff

Overview

The Requirements & Analysis (RA) phase consists of *identifying* and *understanding* the application or *problem domain*. RA is a communication improvement activity between developers and clients/users. Requirements describe what the system should and should not do. It is important to capture requirements appropriately since requirements at too high of a level may not capture all needs and requirements that are too detailed may preclude valid design alternatives.

"You can observe a lot by just watching." - Yogi Berra

RA produces a *Requirements Document*. The following sub-documents can be a part of the Requirements Document:

- Requirements Vision Document
- Requirements Specification Document
- Requirements Analysis Document

The RA stage consists of two activities:

1. Elicitation
2. Analysis

A requirement can be one of two types:

1. Functional
2. Non-functional

There are six main activities of the RA stage:

- Identifying actors.
- Identifying scenarios.
- Identifying use cases.
- Refining use cases.
- Identifying relationships among use cases.
- Identifying nonfunctional requirements.

Techniques used:

- Setting up a prototype
- Build narratives
- Act out scenarios
- Build Use Cases
- Pictures, diagrams & cartoons
- Formal language (to capture more precision)
- Conduct interviews
- Set up questionnaires
- Group meetings
- Activity observations
- Inspecting technical manuals

Validation

Entails checking that the specification is:

- Correct (if it represents the client's view of the system)
- Complete (if all possible scenarios are described)
- Consistent (if there is not contradiction)
- Unambiguous (if exactly one system is defined & no other way to interpret the system)
- Realistic (if it can be implemented under give constraints)

Analysis

- Deals with the problem domain by analyzing requirements.
- Steps are taken to structure, formalize and verify the requirements specification.
- The goal is to completely understand the system needs.
- Further ambiguities can and will be encountered during analysis.
- More information can be elicited from the user or customer.
- Forces developers to deal with difficult issues as early as possible.
- Requirements are structured and formalized into a model(s) representing the problem.
- Help identify entities, boundaries, controls and sequences.
- *Prototypes* can be built to help understand the problem and gain more clarification.

Unified Modeling Language (UML)

This is probably a good place to discuss UML. UML is a well-defined notation for capturing and conveying abstractions of a software system. Moreover, it has been accepted as a standard notation in the industry. There are three main types of models that are used that correspond to five fundamental UML diagrams:

1. **Functional Model** (Use-Case Diagrams)
2. **Object Model** (Class Diagrams)
3. **Dynamic Model** (Sequence Diagrams, State-chart Diagrams and Activity Diagrams)

- **Use-Case Diagrams** describe the external functionality of the system.
- **Class Diagrams** describe the structure of the system.
- **Sequence Diagrams** describe the interaction of entities/objects.
- **State-chart Diagrams** describe the internal behavior of an entity/object.
- **Activity Diagrams** describe control (operations) and data (objects) flow.

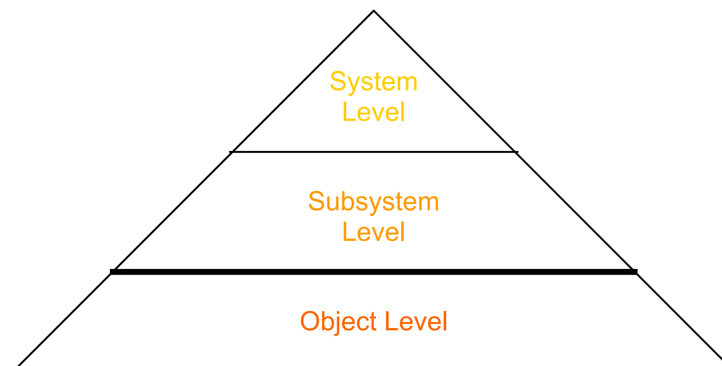
Design

"It is better to understand a little than to misunderstand a lot." - Anatole France

Design

The design phase consists of building the (internal) *solution* to the problem. It consists of transforming analysis artifacts into design artifacts. The primary product of (system) design is the *software architecture(s)*. The design phase primarily deals with recursively decomposing the system into *subsystems*. A subsystem provides a *service* to one or more other subsystems, where a service consists of a set of related operations that share a common purpose. The properties of a subsystem must consist of *low coupling* and *high cohesion*. Low coupling ensures independence between subsystems so that when a subsystem needs to change the impact on other subsystems is minimal. High cohesion ensures that the units or classes within a subsystem work together for a common purpose. Additionally, the relationship between subsystems can be *layered* (hierarchical) or *partitioned* (peer). A good example of a layered architecture is the OSI or TCP/IP stack for networks where each layer logically provides a particular set of services. Lastly, the design phase begins to incorporate the non-functional requirements of the system as well.

There exists three general levels of granularity of design, each having one to N sub-levels of granularity based on need, size, scope or complexity. A good heuristic is having 7 ± 2 partitions at any layer of abstraction, and 7 ± 2 layers total.



Some Software Architectures

- <u>Repository</u>

  This architecture provides a centralized data repository for multiple independent subsystems. It is ideal for a database management system. The main disadvantage is performance when many subsystems are accessing the central repository at the same time.

- <u>Model/View/Controller</u>

  The idea of MVC is to allow for a clean separation from the user interface (UI) and the domain knowledge or *model*. Since UI changes more often, the model does not have to be impacted. The *view* is the user display and the *controller* manages the interactions with the user.

- Client/Server

  This architecture consists of a server subsystem that provides services to one or more client subsystems, which interact with the user. Middleware such as COM/CORBA/Java RMI fall into this kind of architecture.

- Peer-to-Peer

  This architecture is actually a generalization of the Client/Server architecture in that entities or subsystems are both a client and server to each other. Of course, being peers the partitions are usually only horizontally different.

- Pipes and Filters

  Subsystems, called filters, process input data and send the output to other subsystems via pipes, which are the associations between subsystems. The main advantage of this architecture is that filters can be easily substituted or reconfigured.

Non-functional Requirements

- Mapping subsystems to hardware (logical and physical topology)
- Persistent data storage (ER/OO DBMS, flat files)
- Access control policies (security, encryption, authentication)
- Global control flow (procedural, event, threaded)
- Boundary conditions

"The more books you have read (or written), the more classes you have taken (or taught), the more programming languages you know (or designed), the more O-O software you have examined (or produced), the more requirements documents you have tried to decipher (or make decipherable), the more design patterns you have learned (or devised), the more design meetings you have attended (or led), the more talented co-workers you have met (or hired), the more projects you have helped (or managed), the better you will be equipped to deal with a new development." - Bertrand Meyer

Object Level Design

The object level of design is the detailed level of design dealing with the units within those subsystems realized above. After this phase, that is, upon enough iterations of object design, implementation takes place – finally, huh. Although, *design patterns* can be used in the system/subsystem level design, they most certainly play a role in object level design. Application objects are converted into solutions objects by closing the gap between the application objects and the off-the-shelf components. Object level design consists of the following:

- Interface specification

  Interface specification consists of specifying type signatures and visibility, determining object correctness via assertions, specifying exceptions and identifying missing class operations and properties.

- Component selection

  This consists of identifying and adjusting class libraries and application frameworks.

- <u>Class diagram restructuring</u>

  Restructuring consists of increasing reuse, identifying associations, and removing implementation dependencies.

- <u>Object model optimization</u>

  Optimization consists of collapsing objects into attributes when necessary, caching the result of expensive operations, and using appropriate object structures.

Implementation

"Established technology tends to persist in the face of new technology." - Gerrit Blaauw

The design artifact is analogous to a set of blue prints for the construction of the actual software. Ideally, the implementation phase should be one of the shortest if not the shortest step, but in reality it is not. The implementation is actually the culminating point of design. That is, after multiple iterations of design, if when attempting to go another iteration of design the result is much like pseudo-code, then the software engineer knows that it is time to begin implementation. Finally, as a recommended guideline, it is important to comply with a *coding standard*.

Testing

"Testing can only prove the presence of errors and not their absence." – Edsger Dijkstra

Testing primarily deals with discovering differences between the expected behavior and observed behavior of the software system. *Verification* is the means of checking specification conformance and *validation* ensures that the program meets the expectations of the customer. Validation is actually a common thread throughout all of the steps in SE.

Some Definitions:

- <u>Failure</u> – expectations are not satisfied
- <u>Fault</u> – entering an invalid state during run time
- <u>Defect</u> – problem in the static code (Functional, Reliable, Usable, Maintainable)
- <u>Bug</u> – a defect that causes a fault
- <u>Error</u> – leaving out good code or putting invalid code into a program resulting in a defect

There are three types of testing:

1. **Static Testing** (White-box testing)
2. **Dynamic Testing** (Black-box testing)
3. **Fault Toleration** (Release)

Static Techniques Used:

- IEEE 1028 Management Review
- MIL-STD 1521B
- IEEE 1028 Technical Review
- IEEE 1028 Walkthrough
- Yourdon Structured Walkthrough
- Freeman & Weinburg Walkthrough
- Fagan Inspection
- Gilb & Graham Style Inspection
- In-Process Inspection
- Cleanroom Verification Based Inspection
- Van Emden Style Inspection
- Automated Inspection
- Limited Scope Inspections
- Younessi Inspection

Dynamic Techniques Used:

- Partitioning Sub-Domain Testing
- Specification Based Approach
- Equivalence Partitioning
- Cause Effect Graphing
- Boundary Value Analysis
- Category Partition Method
- Program Based Testing
- Defect Based Approach
- Statistical Testing Approach
- Error Seeding

Dynamic testing can be further divided into three general categories:

- Unit Testing
- Integration Testing
- System Testing

Other types of testing:

- Requirements Testing
- Performance Testing
- Field Testing
- Regression Testing
- Alpha Testing
- Beta Testing
- Acceptance Testing

Maintenance

Requirements inevitably change even after release. New bugs surface and customers report problems with the software that need to be dealt with quickly. Maintenance deals with *upgrading* or *changing* the software system after delivery. Maintenance issues should certainly be considered in the design phase by providing a means to easily add new functionality, and taking away existing functionality without degrading the software product. With an effective OO design, concerns are minimized and the system can thrive much longer.

Quality

It is worth mentioning a few things about quality. *Process Quality* should be controlled as much as possible at every SE stage. *Product Quality* is a customer perception. Quality is achieved when the product does what it is supposed to do and does not do what it is not supposed to do.

Primary elements of quality are:

- Functionality
- Reliability
- Usability
- Maintainability

Secondary elements of quality:

- Portability
- Learnability
- Efficiency
- Security
- Reusability
- Formality
- Cost-effectiveness

Concluding Remarks.

To handle complexity and change, SE is an iterative process dealing with granular abstractions appropriately. The quality of the software product depends on the quality of the software development process. With the help of formality, measurement and research, *complexity* can be handled. During all of the SE phases, defects should be prevented, identified and fixed, and ambiguities should turn into points of knowledge as quickly as possible. Lastly, since software requirements can continue to *change* during development and even well after release, designing a software solution should consider extensibility and flexibility so that when the inevitable change does occur the software system can adjust a lot easier than otherwise being the case.

Links.

http://www.sei.cmu.edu
http://www.cs.queensu.ca/Software-Engineering
http://www.rspa.com/spi
http://www.computer.org/tse
http://manta.cs.vt.edu/ase
http://mingo.info-science.uiowa.edu/soft-eng
http://www.acm.org/pubs/tosem
http://www.software-engineer.org
http://www.computer.org/tab/seprof/code.htm
http://www.sei.cmu.edu/cmm
http://www.seaq.net.au/index.jsp