CS360: Chapter 1

Software Engineering

**Software Engineering (SE)** is the discipline of creating fault-free software, delivered within the acceptable parameters.

The scope of **SE** is broad, including aspects of math, and Computer Science (CS), even economics, management, and psychology.

Lots of software is delivered outside of acceptable parameters, the process of **SE** is unique and has unique problems.

In cases where you could code 9/10 faster and hence forth be 9/10 cheaper, it may be non-optimal. If you had a new technique then even if you could do so, it may cost more to introduce that technique. Additionally, the time required to make meaningful progress would be far greater.

The maintenance cost could be greater with the new technique in spite of the speed and comparable code quality. It is important to consider the long-term in software engineering.

**Life-cycle model**: A description of the steps that should be performed when creating software.

Most of these cycles will be chunked into separate smaller steps called **phases**. The number of phases varies.

The life of the software product from the first concepts to the final retirement of the system is called the **life-cycle**.

These steps may not always be carried out exactly as specified in the life-cycle model, especially when over the time and money budgets.

The very first software method was the **waterfall model**, which involves moving through 6 phases. Each phase must be completed and cannot be returned to, once you go from phase 1 to phase 2, you cannot return to phase 1.

Phases of Waterfall:

1. **Requirements**: The requirements must be gathered and the concept must be explored and refined.
2. **Analysis**: The client’s requirements are analyzed and presented in the form of **specifications document.** At the end of this phase the **software project management plan** is created and it will document the proposed software development process in full detail.
3. **Design**: The specifications will undergo a couple of design procedures during this phase, the initial procedure is the **architectural design**, it will get broken down into components that will be designed. The process of this creation of the module is called **detailed design**. The documents that are born from this work are the **design documents**, which describe how the software will work.
4. **Implementation**: This is where coding occurs and testing, typically each module will be tested separately. The components will be fused together and then tested as a complete package, this fusion is **integration**. When the programmers and software engineers are happy with how the components work, they will move to testing. It will be tested by the client (**acceptance testing**). This is when this phase ends.
5. **Postdelivery maintenance**:The software that has been developed will be used for its purpose, during this time it will be maintained. This includes all changes after the client has the software. These changes can come in three flavors:
   1. **Corrective maintenance:** fixes bugs and glitches in the program
   2. **Enhancement:** a form of corrective where specifications are changed
   3. **Perfective**: alterations that the client thinks will make the product better, this could be new features or streamlining a process
   4. **Adaptive**: changes made to a software that help keep it running when the system it is running on changes.
6. **Retirement**: This is when the product is no longer going to be maintained, typically when the product.

The International Electrotechnical Commission (IEC) has defined maintenance as “software undergoing modifications to code and associated documentation due to a problem or the need for improvement or adaptation.”

Maintenance occurs whenever a bug is fixed or requirements changes, irrespective of whether this takes place before or after installation. (This is a more workable definition than the original **development-then-maintenance model** since so many changes can occur during and after development)

In some instances, maintenance can be very expensive, especially as a result of developmental errors, so much so that in some cases it might be easier to start from scratch than perform maintenance.

Figures show that around 75% of costs come from maintenance post-delivery.

Additionally, much work may need to be done, in the beginning the project is mostly paper after delivery the project is code that needs to be trouble-shooted and more work must be done to make sure the fix works. A large number of faults will come from the first stages, which is why the work needs to be refined.

It is also important to keep code design unified, if functions are coded with different arguments or in different orders from function calls in other parts then it can cause errors that might not be caught for a while. Because communication is so important, a project that takes a single programmer an entire year could also take a six-man team an entire year.

In our previous evaluation of the steps of a classical life-cycle model, we note there is no strict planning or testing phase. The reason for a lack of planning is because it is hard to create a plan without knowing what exactly you are creating. In classical paradigms, planning is used for the requirements and analysis phase. Once a solid idea has been creating for the project a software project management plan (SPMP) is drawn up. This has figures regarding costs, staffing needs, and scheduling.

That SPMP is referenced throughout the project by management in case of deviations.

There is no testing because, each phase needs to be checked for errors and faults, especially towards the end of each phase. So by isolating the phase it could result in the exclusion of testing at each point in the phase. There are actually careers for people who ensure the quality of the software, they would be software quality assurance officers or something like that.

There is no documentation phase for reasons opposite to testing and planning. Specifications and documentation needs to be up-to-date at all times. The reasons for this is because:

* Large turnover means that projects are constantly having new members added on and removed, they all need to have a good idea of what is going on while they work on the project.
* Specification is impossible if documentation is incomplete, incorrect, or outdated.
* It is impossible to say software works if there is no metric for its performance.
* Maintenance is not feasible without a precise description of what the software does in the current version.

The **structured** or **classical paradigm** was developed between 1975 and 1985, although they were flawed. They would use structured system analysis, data flow analysis, structured programming, and testing. However they had size limitations, only handling smaller programs (less than 500K lines of code). Additionally, it had postdelivery maintenance costs on par with previous methods. Part of this was because of the split between doing either operation oriented tasks or attribute/data oriented tasks, but not both could both.

This is where object-oriented programming(OOP) comes in, it balances operations and attributes. OOP would use artifacts, which are component of a software product, a piece of code, a function or manual. These artifacts would unify both operations performed on attributes and the attributes themselves.

While seemingly not different from previous approaches, objects have the details of their operations and attributes hidden from code apart from the object. This is the basis of information hiding. This hiding of functionality is a strength to OOP.

The main program could send a message to an object telling it to perform some **method** (that being an implementation of an operation).

The separation of code reduces the chance of **regression faults** (the introduction of errors in the program as a result of changing the code in one section that is thought to be unrelated to other parts).

OOP takes advantage of the physical representations of real-world objects making development simpler.

Since OOP has objects that are independent of the rest of the program, all of the traits that the object models are found within the object, this is an example of **encapsulation**. Because of this encapsulation, the objects are responsible for returning information when called upon. This is why you could call OOP **responsibility-driven design** or **design by contract**.

The classical paradigm will create one huge unit for a program, however OOP will produce objects separate from a main program, this is the strength of OOP, because when done properly it will produce smaller less complicated units. These units can also be reused for other projects.

Phase is Classical Paradigm, Workflow is OOP. The two terms are completely distinct.

This leads to a difference in the steps used for OOP

1. Requirements Workflow
2. OO analysis workflow
3. OO design workflow
4. OO implementation workflow
5. Postdelivery maint.
6. Retirement

The major differences in the process is that Classes are extracted in Analysis workflow rather than modules being extracted in the design phase.

Terminology:

**Client:** individual desiring a software solution

**Developers:** The people who are responsible for coding and engineering the software solution

**Internal Software Development:** When the client and developers are part of the same organization

**Contract Software**: The client and developers are part fo different organizations.

**User**: The person who will be using the software, or the person on whose behalf the client has commissioned the software.

**Commercial off-the-shelf software:** Software that offers a basic service that can be sold in a volume.

**Sometimes called clickware**.

**Open-source software:** software which has its code available to the public and can be edited by any developer with input.

**Linus’ Law:** “Given enough eyeballs, all bugs are shallow”/All faults can be found by third parties.

**Software:** All the documentation and code that is inherent to a software project.

**Program:** A single piece of autonomous code

**System:** A group of related programs.

**System analysis:** the first two phases of development (requirements and analysis)

**Systems design:** the third phase (design)

**Product:** a nontrivial piece of software

**Methodology:** “A way of developing software products”/ Really means the science of methods

**Paradigm:** A way of viewing the world, or a model or pattern.

**Technique:** a part of the software process.

**A programmer makes a** mistake **resulting in a** fault **in the program.** **If this program runs then it will result in a** failure**. An** error  **is the amount by which a result is incorrect**.

Object-oriented terms for attributes:

**State variable –** used in programming lit.

**Instance variable** – used in Java programming lit.

**Field** – used in C++ programming lit.

**Property –** used in Visual Basic .NET lit.

When the main program sends a function call to a object, this is called **sending a message** to that object.

Attributes refer to the traits of an object

Methods refer to the actions performed by an object on it’s traits.

Read the IEEE Software engineering code of ethics and professional practice.