CS1Ah Lecture Note 7

Introduction to Software Engineering

In this note we provide an overview of Software Engineering. The presentation in this lecture is intended to map out much of what we will study in this course rather than to provide detailed descriptions of the topics — those will come in later lectures. This lecture is closely based on the introductory chapters of Sommerville[5].

What is Software Engineering?

In Software Engineering we study how to build and maintain software systems in a controlled, predictable way. In particular, good Software Engineering should give control over:

Functionality: The system should provide functions that meet the needs of all the stakeholders of the system. This will involve determining what these needs are, identifying and resolving conflicting needs, and designing a system that meets the identified needs.

Quality: The quality of a system determines if it is useable in its intended context[1]. The required quality of software varies enormously. Software on a desktop PC is usually of fairly low quality while the control software for railway signalling applications needs to be of the highest standard. The required quality strongly determines the cost of software production.

Resources: The purchasers of a system usually enter into a contractual relationship with the supplier companies. The suppliers need to predict resource requirements: quality of development team, human effort, timescales, and supporting tools and equipment; in order to determine the cost of developing the system. Failure to predict costs can have serious consequences for the suppliers and purchasers of systems.

Software Engineering is a relatively young discipline by comparison with most other engineering disciplines. As a result, many projects still suffer from poor Software Engineering. Most of you will be familiar with many software project disasters.

[1] Less than 1% of software purchased by the US Government is deployed without significant change.

Why is Software Engineering Important?

It is clear that software is important:

Critical applications: Software has found its way into many applications where failure has serious consequences. For example: car braking and steering, process control, safety systems in hazardous processes, civilian avionics, communication networks and telephony, ...

Competitiveness: Software is seen as the key to competitiveness in many areas. In retail, finance and entertainment, e-commerce is seen as a critical development; in many other areas of economic activity good software is seen as a key element in the competitiveness of firms.

Economically: The estimated value of systems containing embedded software will exceed 10^{12} dollars in the next few years. This is only one market for software, there are many others.

This does not mean Software Engineering is important at the moment. Critics point to well-publicised failures to supply well-engineered software systems by suppliers who do attempt to use best practice. These well-publicised failures mask many projects that are successful and are delivered on time and on price. We can argue that the aims of Software Engineering are important and in some contexts those aims can be realised.

Software Products

Stakeholders in a software product are usually concerned with two broad categories of characteristics of software:

Functionality: The important characteristic of a *function* supplied by software is that is is either present or absent. Stakeholders are usually concerned that the software has all the required functions present (or at least that they can be supplied eventually).

Attributes: An attribute of a software product is something that can be measured (directly or indirectly) and that measurement lies in some range. Stakeholders are also interested in seeing that the attributes of a software product meet some minimum level. Typical attributes are:

Maintainability: This is a measure of how easy a system is to maintain during its deployed life. This cannot be measured directly before the system goes into operation because it depends on many features of the software and on what changes the systems will be expected to undergo. At this stage the Maintainability of a system is assessed qualitatively on the basis of inspections and measures of the quality of the structure of the code. In operation, Maintainability can be measured (usually as mean time to repair). This measure is only significant if the product is used for a long time and/or it has a large number of installations.

Dependability: This is a measure of how "trustworthy" the software is. Usually this is a combined measure of the safety, reliability, availability and security of a system. The issues of measurement of Dependability are similar to those of Maintainability.

Efficiency: For some systems it is important to keep the use of system resources (time, memory, bandwidth) to a minimum. This is often at the cost of added complexity in the software. Improving the efficiency of a system often involves a detailed analysis of the interactions between different modules making up the system as a consequence the cost of improving efficiency often grows non-linearly in the size of the system and the required efficiency.

Usability: Usability is a measure of how easy the system is to use. Again this is hard to measure since it arises from many factors. Often this is approximated by very rough measures like learning time to carry out some operation.

Attributes can make conflicting demands on the software product. For example, improving efficiency may lead to more complex interactions between software modules and to more interactions between formerly independent modules. Changes like these can have a serious effect on the Maintainability of a system because more interaction between modules can make tracking down and fixing errors much more difficult.

Software Production Activities

The production of software involves a number of different activities during which we try to ensure we deliver the required functions and the required level of attributes. We can group production activities into four broad categories:

Specification: This is a description of what the software has to do and sets acceptable levels for software attributes. For most software systems going from the user needs to a statement of requirements and then to a precise specification is a difficult and error prone task. The study of Requirements Engineering is an increasingly important part of Software Engineering.

Design: This covers the high-level structural description of the *architecture* of the system, through the detailed design of the individual components in the system and finally to the implementation of the system on a particular computing platform (usually hardware plus operating software).

Validation and Verification: Validation is the activity of checking the correct system is being constructed (*building the right system*). Verification is the activity of checking the system is being constructed correctly (*building the system right*). These activities are essential to ensure a software project is going in the right direction.

Maintenance: This activity ensures that the system keeps track with changes in its operating environment throughout its operational life. This involves correcting errors that are discovered in operation and modifying the system to take account of changes in the system requirements. Repairing Millennium Bug errors in software is an example of Maintenance activity (and a good example of the costs of

such activity). In one sense we can see the Millennium Bug problem as a change of requirement because when these systems were written their intended lifetime was much shorter than has turned out to be the case.

Software Production Processes

Organisations involved in creating software choose a particular way of organising the software development activities depending on the kind of system that is being constructed, and the needs of the organisation. The pattern of organisation is usually called a *software development process*. Such processes are used to give overall structure to the development process. Different software development processes have different characteristics. Process characteristics provide a basis for deciding which process to choose for a particular project. In particular, in many cases the choice of process should attempt to reduce project risk[2]. Typical software process characteristics are (this is not an exhaustive list):

Visibility: How easy is it for an external assessor to determine what progress has been made?

Reliability: How good is the process at detecting errors before they appear in a product?

Robustness: How well does the process cope with unexpected change?

Maintainability: Is the process easy to change to take account of changed circumstances?

Rapidity: How fast can a system be produced?

Software Production Process Models

Here we briefly introduce three popular software development processes. We will return in later notes to consider them in more detail. Their inclusion here is to illustrate possible organisations of the activities.

Waterfall Model

This is a linear model where each activity provides the input to the next stage in the process. This process usually has high visibility because at the close of each stage full documentation is generated for that stage. Because of the linear nature of the process it is not particularly robust because any changes tend to force us to loop back to some earlier change and then follow through each of the stages again. In the early days of software production this was the standard model used by most developers.

Evolutionary Development

This approach was introduced by Gilb in 1985[3, 4]. It is often associated with the object-oriented approach to system development. The aim of this process is to split the problem up into many sub-tasks each of which can deliver a part of the product that

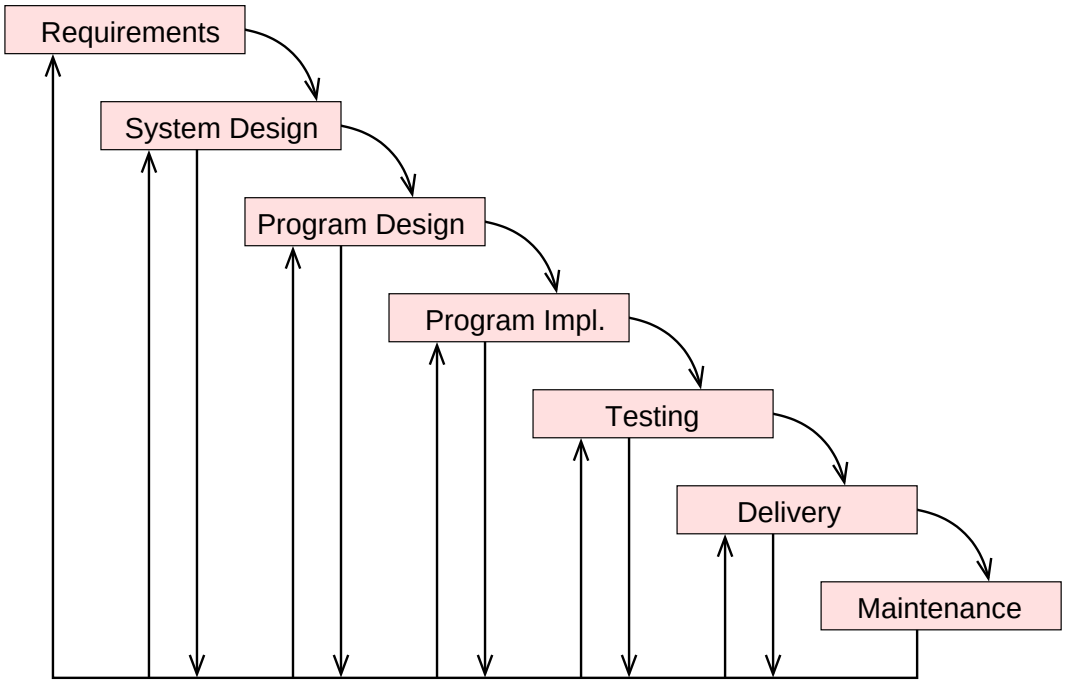

Figure 7.1: The Waterfall Model

offers a tangible improvement in the system to the users. Each sub-task is tackled and delivered as a separate deliverable in the life of the project. Delivery of a new component changes the perception of the project, then the priorities on completing the remaining tasks are re-evaluated in the light of the new component and the most important (from a user view) is chosen as the next to deliver. Evolutionary development is highly Robust because changes can easily be factored into the process but it is not particularly Visible because it may be difficult to keep track of many sub-tasks in an efficient way.

Spiral Model

This model was introduced by Barry Boehm in 1986[1]. It is an iterative approach to system design that is centred round the creation of prototype systems that more closely approximate the system on each iteration until an acceptable system is constructed. At the start of each cycle the risk of proceeding is assessed and a decision take on whether to proceed on the basis of the project risk. The spiral model is Robust because the iterative nature allows us to plan flexibly, it is also visible because each prototype is fully documented.

Measurement

Once a process is established it is possible for an organisation to measure concrete elements of its software development process. Such measurement is carried out to discover the effort used on projects and measures of the attributes of the systems produced. Many organisations also attempt to use such measurements to create a predictive model of development costs in terms of features of the customer requirement.

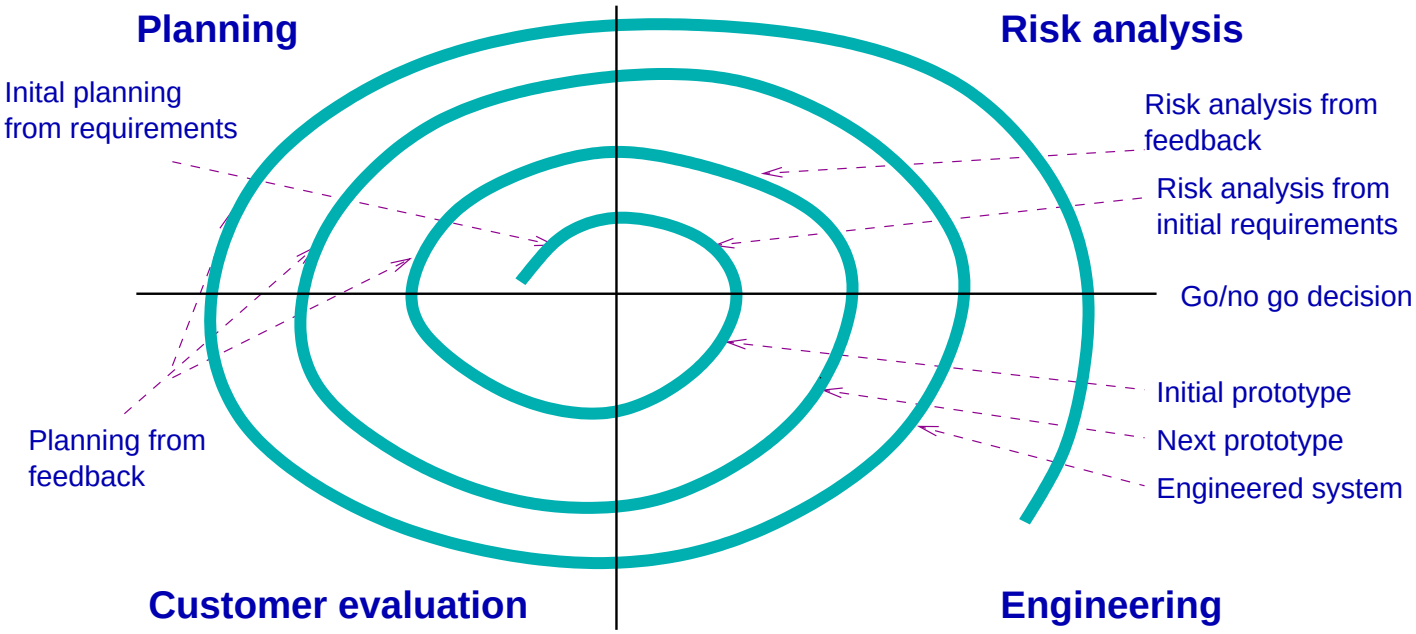

Figure 7.2: The Spiral Model

Provided the products are reasonably well focussed and stable and the development teams are also stable this approach is an essential tool in providing accurate estimates of development costs. Most companies producing high cost software keep extensive historic data on past projects and use this as a basis for costing new projects.

Summary

We have introduced the main elements of Software Engineering. In particular:

- The aim of Software Engineering.
- Software functions and attributes.
- The main activities involved in developing software.
- The notion of Software Development Processes and Process Characteristics.
- Some sample lifecycle models.
- The importance of measurement in Software Engineering.

Stuart Anderson.
David Aspinall, 21st October 2002.

Questions

1. *Over the past 30 years hardware engineering has been significantly more successful then software engineering.* Identify arguments that support and contradict this statement.

2. Think of a few of your favourite software project disasters. Write down what you think were the main causes of your chosen disasters. Many of the explanations for these disasters take the form of folklore that has little to do with the complex causes that surround software disasters. Use your favourite web search engine to try to get to the bottom of one or two disasters.

3. Consider each of the software development processes outlined above try to assess how well you think each will do on each of the process characteristics listed above.

Bibliography

[1] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[2] Barry W. Boehm and Tom DeMarco. Guest Editors' introduction: Software risk management. *IEEE Software*, 14(3):17–19, May/June 1997.

[3] Tom Gilb. Evolutionary delivery versus the "waterfall model". *ACM SIGSOFT Software Engineering Notes*, 10(3):49–61, July 1985.

[4] Tom Gilb. *Principles of Software Engineering Management*. Addison-Wesley, 1988.

[5] I. Sommerville. *Software Engineering, 5/e*. Addison-Wesley Publishing Company, 1996.