Software Engineering notes

Todd Davies

January 15, 2015

Introduction

The development of software systems is a challenging process. Customers expect reliable and easy to use software to be developed within a set budget and to a tight deadline. As we come to depend upon software in so many aspects of our lives its increasing size and complexity, together with more demanding users, means the consequences of failure are increasingly severe. Experience from nearly 40 years of software engineering has shown that programming ('cutting code') is only one of a range of activities necessary for the creation of software systems that meet customer needs. The rest of the time is spent on: planning and acquiring resources for the project; investigating the business and technical contexts for the system; eliciting and documenting user requirements; creating a design for the system; and integrating, verifying and deploying the completed components.

This course unit builds on the programming skills you have gained in the first year, to provide you with an understand-

ing of the major challenges inherent in real-scale software development, and with some of the tools and techniques that can be used in their attainment.

Since software engineering is a subject best learned hands-on, this is a project-based module that involves less traditional lecturing than usual. Instead, relevant skills will be acquired during fortnightly two-hour workshops, supported by a weekly one hour lecture. The understanding gained will be practised through individual and team project assessments.

Aims

This unit aims to give students an introduction to the principles and practice of analysis, design and implementation in object orientated software engineering. Through experience of building a significant software system in a team, students will further their? experience and understanding of the problems that arise in building such a system. They will develop the analytical, critical and modelling skills that are required by a successful software engineer.

Contents

1 A history of software engineering

In the 60's, when programming projects started to get large enough to warrant their own software development strategies. The increased complexity of projects at this time was leading to many of them going over budget, over time, having low quality, not meeting requirements, and being difficult to maintain. This, even at the time, was referred to as the software crisis.

Dijkstra summed it up in an article in Communications of the ACM:

> The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem. (Dijkstra)

1.1 Why do projects fail?

We've seen that projects often failed early on in the history of software development, but why did each project fail? There are a large number of reasons why a specific project could end in failure, however, there also a number of common reasons for disaster:

- Unrealistic goals
- Inaccurate estimate of project complexity/resources needed
- Badly defined requirements
- Unmanaged risk
- Use of immature technology
- Sloppy development practices

Now, there are software development practices and methodologies that aim to mitigate the risk of project failure by defining how a project should be developed. Figure 1.1 shows the a generic view of such a process.

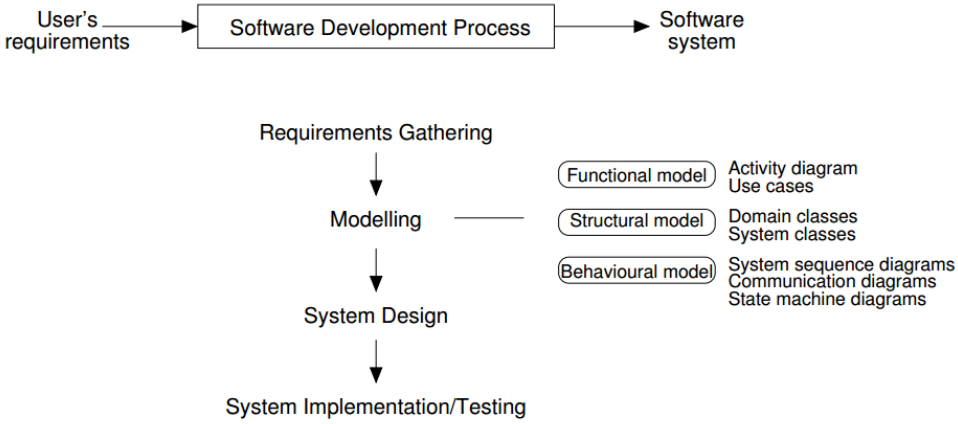

Figure 1: A generic software development process

2 Gathering requirements

Gathering the requirements of a software project is crucial, since if a developer doesn't understand what is being created, how can they create it properly? There are two types of requirements, functional and non functional.

Functional requirements are things that the system should **do**. For example, the system should produce both a a4 sized document, and a a5 sized e-readable copy.

Non Functional requirements are things the system should **be**. This could be that the system should be developed with ethical software development practices, or the system should be easily extensible for future development.

When gathering the requirements of the process, it is important to accurately capture the business process that is being encapsulated in the system. It is important to understand the state of the process now, and the desired state of the process at the end of the development time. UML is a way of representing business logic, and the relationship between different parts of the system. Activity diagrams can be used to create a logical model of the system.

Doing this avoids making assumptions that may turn out to be false later, and makes sure that the system supports the relevant activities.

Note:

Activities are composed of *actions*, which are non- decomposable pieces of behaviour. Activity diagrams are composed of activities.

We can use unit testing, acceptance testing and usability testing in order to check that we have met the requirements. It is important to keep the project 'agile' (keep in contact with users (for this module)), so that the requirements can be updated as the project develops.

Requirements can be gathered from customers, stakeholders and users (of which there may be several different types, such as clients, or staff members). They can be gathered from interviews, analysis of working practices, conversations, or questionnaires. It is important to remember that each different user of the eventual system will have different requirements, and talking to the right users is important.

The COMP23420 lecture slides give what I believe to be the worst acronym in the world, in order to remember how to prioritise requirements. Hopefully it's memorable because it's bad:

- **M**ust
- **o**
- **S**hould
- **C**ould
- **o**
- **W**on't this time (would)

2.1 Maintaining a glossary

A glossary is a collection of important terms and their definitions that relate to your project. It is important, since

the team must understand the domain with which they are working if they are to produce high quality software. It's important for documentation, communicating with users and consistency. As with any part of the project, it must be continually updated as the project matures.

The Pragmatic Programmer puts it very well:

> As soon as you start discussing requirements, users and domain experts will use certain terms that have specific meaning to them. They may differentiate between a 'client' and a 'customer' for example. It would then be inappropriate to use either word casually in the system.
>
> Create and maintain a project glossary - one place that defines all the specific terms and vocabulary used in a project. All participants in the project, from end users to support staff, should use the glossary to ensure consistency. This implies that the glossary needs to be widely accessible.
>
> **It is very hard to succeed on a project where the users and developers refer to the same thing by different names or, even worse, refer to different things by the same name.**
> (The Pragmatic Programmer, Andy Hunt, Dave Thomas, P

2.2 Formatting a requirements document

A requirements document must have all of the requirements set out in an unambiguous format, with separate requirements numbered (often using nested numbering) so that

they can be cross referenced, and glossary terms highlighted. UML can be used in order to better convey the ideas in the requirements document.

3 UML

The Unified Modelling Language is a standardised general-purpose modelling language used in software engineering. It lets developers use a common format for visualising the architecture of a system, where diagrams are used to aid in the creation and documentation of various aspects of the software engineering process.

For a requirements document, activity diagrams and use cases are used. Use case diagrams provide an overview, while use case descriptions go into the detail of what goes on. Both primarily capture functional requirements.

3.1 Actors and use cases

Actors perform use cases on a system, and have use cases performed on them. They are an entity (e.g. a human, database, AI algorithm etc) that directly uses or is used by the system, and activity diagrams should reflect what happens in the business process that will be realised by the new system.

Activities (which, as said above, are composed of actions) map to use cases in the system, and therefore describe functional requirements. For each use case, we want to know

the entry and exit conditions for the use case to happen as intended.

It is important to properly specify the actors in a system. Since they are often not controlled by the system (or at least included in the development process), you need to know exactly what an actor can and cannot do, and if they are a user, what they might like to do. This helps improve the user experience (UX).

Use cases contain the following information:

Name
: Use Cases are named in the form *verb noun* (such as 'load website', or 'create request'). Using CRUD terminology is good, and it is important to use a consistent level of abstraction (don't have one use case for initialising memory, and another for hashing a file).

Entry and exit conditions The conditions for the use case to be properly executed. It should handle all eventualities for the entry conditions, and have all of the eventualities listed as exit conditions.

Flow of events The use case should have **both normal and alternative flows** that describe what happens for the duration of the use case.

Craig Larman describes three types of use case; casual, brief and full. Casual use cases are basically prose, and can be hard to parse and understand. Brief use cases are usually a bulleted, list of short sentences, which is what we're to use in this course. Full use cases are the works, with lots of detail, diagrams, references etc.

Refer to figures 3.1 and 3.1 for examples of activity and use case diagrams.

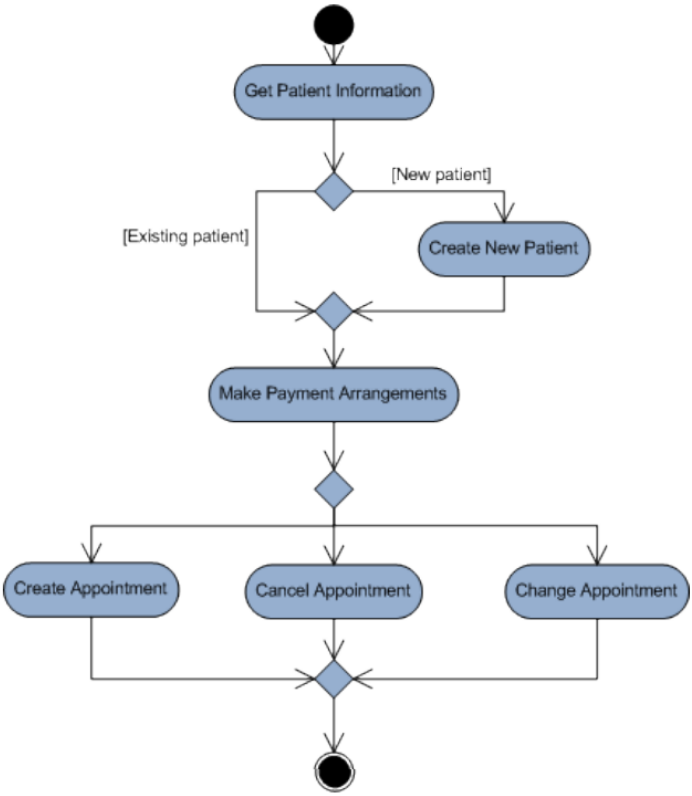

Figure 2: A sample activity diagram

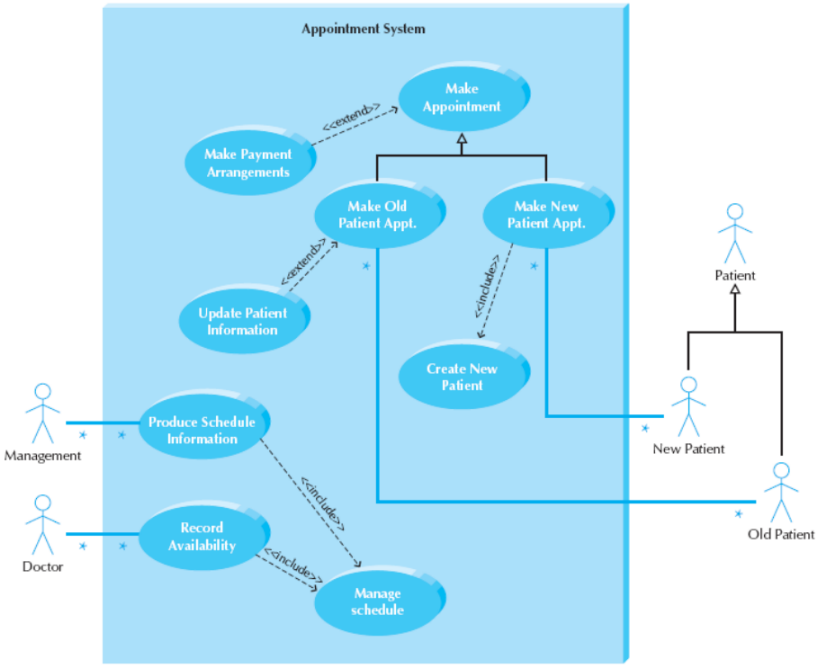

Figure 3: A sample use case diagram