Software Engineering: Project Management Part 2

EBU6304 Software Engineering

Dr Matthew Huntbach

matthew.huntbach@qmul.ac.uk

Risk Management

Problems with Projects

- Projects do not always follow plans
- Good management involves being prepared for unexpected delays or failures in reaching milestones and producing deliverables
- We should make alternative plans for:
  - Failures which experience suggests often happens
  - Failures which would have particularly serious consequences
- We cannot plan for everything that can go wrong
- The extent to which we make alternative plans may vary depending on likelihood and consequence
- In most cases the alternative plans will have the aim of bringing the project back to its intended goals and timescale
- It may be necessary to increase the timescale and/or decrease the goals, or even abandon the project

Risk Management

- Risk management is the process of identifying possible failures, establishing where alternative plans should be developed, and making those alternative plans
- A risk is a probability of something undesirable happening during development
- Risks can be divided into:
  - Project risks : where the schedule or resources available for the project are affected
  - Product risks : where the quality or performance of the software produced is affected
  - Business risks : where the organisation which is developing or procuring the software is affected
- Some risks may fall into more than one of these categories

Examples of Project Risks

- Skilled staff leave, or are absent due to illness
- The organisation managing the project changes its priorities: may move staff, reduce available resources, ask for completion in a reduced timetable
- Resources expected from other suppliers (hardware, software) are not available on schedule, or do not meet requirements
- Physical environment: building works overrun, transport problems due to poor weather, etc

Risks which are both Project and Product Risks

- Larger number of changes to requirements than anticipated
- Analysis and specification is not available on schedule
- Implementation reveals the size of the system will need to be much larger than anticipated
- Implementation reveals the development team are lacking in necessary skills and knowledge

Examples of Product Risks

- CASE tools chosen to support the product do not perform as anticipated
- Serious design error discovered during implementation
- Serious error is not uncovered in testing - logical error, performance, security
- Third party software used in the system performs poorly
- User interface proves unattractive to customers

Examples of Business Risks

- The technology on which the system is built is superseded by new and better technology
- New technology means there is less need for the system
- A similar system developed elsewhere becomes available and is successful in the market before the system is completed
- The organisation responsible for the system experiences financial problems and closes down

The Risk Management Process

The tasks involved in risk management are

- Risk Identification
  - Identify project, product and business risks
- Risk Analysis
  - Assess the likelihood and consequences of these risks
- Risk Planning
  - Draw up plans to avoid or minimise the effects of the risk
- Risk Monitoring
  - Monitor the risks throughout the project

Avoidance Strategies

- Be careful about using new or unfamiliar products (tools, software, hardware), balance any benefit obtained from risk they will not perform as expected
- But keep awareness of new products, do not stick to old tools and systems out habit or unwillingness to change
- Ensure staff are well trained, avoid having just one person who has skills and/or knowledge of a vital topic
- Ensure working environment is pleasant, so staff are not tempted to move
- Maintain awareness of the market, new technologies, competing products
- Ensure senior management are well informed on progress of project and its importance to the organisation’s goals
- Actively monitor performance, be prompt if there is any slippage, do not “hide” problems from senior management or staff

Contingencies

- Contingencies are alternative plans which are established to be carried out if the original plan cannot be followed
- They might involve:
  - Alternative resources being identified (human, tools, location)
  - Possibility of outsourcing work and resource supply
  - Priority on which parts of the project to continue, which to postpone or abandon
  - Identifying aspects of the project which could be modified

Also

- Damage limitation - if a risk that may have already caused problems is discovered what can be done to identify and reduce the effect of those problems?
- Minimisation strategies - what can be done to minimise the future impact of a possible risk?

Agile Risk Management

Some aspects of the Agile approach to software development will help with risk management:

- Agile emphasises continuous testing throughout development, which should enable errors to be detected early
- Agile emphasises frequent interaction with the customer, so the customer can point out if the system is being developed in a way that seems to be unsatisfactory

Some aspects of the Agile approach, however, may increase risks:

- Agile does not emphasise long-term planning, which may mean it is harder to identify some risks
- Without long-term planning, it is possible that features may be added that cause problems later on

Quality Management

Quality Management

- Software quality management is concerned with ensuring a suitable level of quality is achieved in a software product
- Quality management involves establishing processes and standards within the organisation that will lead to it producing high quality software
- For individual projects, a <u>separate</u> quality management team should check the deliverables to ensure they are consistent with the organisation’s standards and goals
- Separating quality management from general development ensures an independent view is taken of the software
- In general, it is hard to be objective when considering the quality of material you have produced: problems are missed due to over-familiarity, poor solutions are kept in if they were time-consuming to produce, personal factors amongst team members may impede judgment

Agile Quality Management

- In the Waterfall approach to software development, quality assurance is seen as a separate process, that takes place after the system has been implemented
- In the Agile approach to software development, with its emphasis on developing software by continuous release of new versions with extra features, quality assurance is seen as a process that takes place at the same time as implementation
- Agile development involves a member of the development team having the responsibility of assessing the quality of the system from the point of view of the customer, or actual customer involvement with a customer representative as part of the team
- Early release of a system means it can be tested in real use, which often reveals issues that are missed when a system is tested in a way that is made up just for testing

What is high quality software?

- Quality means the software should meet its specifications
- It should be efficient and easy to use
- It should be easy to modify to meet changing requirements
- Attention to internal quality of the code - good structure which does not break the main design principles - will greatly contribute to high quality code in terms of meeting customer requirements
- However, code is “invisible”, customers are unlikely to be directly concerned with “good structure”
- Customers may find it hard to specify their exact requirements, leading to incomplete and sometimes inconsistent specifications
- The focus may be “fitness for purpose” rather than specification conformance

Fitness for purpose

- Have programming and documentation standards been followed in the development process?
- Has the software been properly tested?
- Is the software sufficiently dependable to be put into use?
- Is the performance of the software acceptable for normal use?
- Is the software usable?
- Is the software well-structured and understandable?
- This can also be expressed as “Is the software good enough?”

“Good enough” software

- This is an acknowledgement that it is difficult (and expensive) to guarantee software is completely free from errors
- A balance needs to be achieved between meeting consumer demand for new products or client demand for software to be produced by a deadline, and giving guarantees about performance
- Software for use in safety critical applications (controlling machinery where an error could result in injury or loss of life) needs higher standards than software for routine administration or entertainment
- Companies producing software need to balance the advantage of bringing new products to market quickly with the risk of damaging reputation if their products are often of poor quality
- “Good enough” does not mean lack of attention to good standards during software development: sloppy standards early on will lead to delay and more cost later on as well as poor quality products

“Bug or feature?”

- The question “Is it a bug or a feature?” is often asked by users of software systems
- A “bug” is some aspect of the system’s behaviour which it is acknowledged is incorrect
- A “feature” is an aspect of the system’s behaviour which is acknowledged as correct
- What about aspects of the system’s behaviour which are neither acknowledged as correct nor as incorrect?
- This comes about because it is usually impossible to specify every aspect of behaviour in advance
- Unspecified behaviour should be logically consistent with specified behaviour
- Users may become aware of an inconsistent or unintended behaviour and find it useful in some circumstances, in this way a “bug” becomes a “feature”
- It is hard to remove such features, even if they would lead to a better design, if users have come to rely on them - they may be deprecated in order to give backwards compatibility

Quality Attributes and Trade-offs

- The following are all attributes of quality we might want to see in software:

  Safety
  Security
  Reliability
  Resilience
  Robustness
  Understandability
  Testability
  Adaptability
  Modularity
  Complexity
  Portability
  Usability
  Reusability
  Efficiency
  Learnability

- It is often necessary to trade one off against another, for example extensive run-time checking to ensure resilience and robustness will lead to loss of performance when considering efficiency
- Some of these are “invisible” aspects (such as quality of code), some “visible” (such as quality of user experience)
- Project management should include an assessment of what are the most important quality attributes for the software being developed, and an agreed way of assessing the extent to which some quality attribute is present in the product

Visible and Invisible Quality

- From the point of view of the customer, the most important quality aspects are what is immediately visible: good user interface and good range of features
- Customer experience of using the system will demonstrate other important quality aspects, such as ease of use of the system, efficiency, quality of its support documentation
- Quality aspects such as security and reliability are not immediately visible, but are of direct relevance to the customer
- Code quality (that is well structured, informed by Clean Code and Design Principles idea) is not of direct relevance to the customer, but is important
- Code quality is important because good quality code can be easily modified to add new features, or make required changes
- Poor quality code is more likely to contain bugs, leading to poor reliability and possible security problems

Analysis of Performance

- Users generally want software to function “instantaneously”
- Poor performance is a “bug”, even if the behaviour is logically correct
- Like other bugs, poor performance may only occur in unusual circumstances, or in large-scale use, so is missed during testing before release
- The first defence against poor performance is to employ skilled programmers who are familiar with algorithms
- Also, always use library software for routine algorithms such as sorting
- After this, poor performance may come from complex interactions which it is hard to predict in advance
- There are tools which will analyse code execution and show where most of the time is spent
- It may not be where you expect
- There is little point in improving the efficiency of code which does not contribute much to the execution time, especially if efficiency improvement leads to more complex code

Software Reliability

- Software reliability is the probability of a software system operating without failure for a specified period of time in a specified environment
- A figure for reliability may be part of the requirements specification, but it is difficult to estimate the reliability of a system accurately
- Software is not like physical machinery or living things, it does not age or wear out in the sense of it changing over time. However changes in the environment it is used may lead to failure
- Software may be unreliable because when it was designed and implemented, care was not taken to ensure it would deal correctly with all situations encountered when used in practice
- The problem could be due to the specification not taking account of a situation, or the implementation not meeting the specification
- As with risks in software development, when considering software reliability, two aspects of a software failure need to be considered:
  - Possibility of the failure happening
  - Damage caused if the failure happens

Security and Resilience

- A major aspect of programming is ensuring that software or the data it uses cannot be damaged by people using it unintentionally in an incorrect way, or people with an intent to damage the interests of legitimate users
- Software should execute predictably and correctly even when run by someone who wishes to cause damage or access data or services to which they are not entitled
- Software should contains few or no weaknesses that can be exploited to cause it to stop executing predictably or correctly
- Software should be able to continue operating even if attempts are being made to use it incorrectly, and it should be able to recover as quickly as possible from attacks made on it
- Different customers will have different requirements on these issues, security and resilience are less important with software intended for leisure use, but critical for software dealing with financial transactions

Investing in Software Resiliency

- Paper by Warren Axelrod on module website
- Abstract:

Software is inherently error-prone and such errors can lead to failure of those systems of which the software is part. On the other hand, with software being only one of many components of a system, there are many choices in regard to attaining a particular level of system resiliency, not all of which are software-related. It is important to consider software resiliency in relation to the resiliency of the entire system, including the human and operational components. The goal of this article is to help those who develop, implement, and operate computer networks and systems in determining the factors to include when investing in software resiliency.

Software Standards

- Standards are a documented set of rules which it is agreed all will keep to
- Standards can be international (e.g. ISO 9001), national, organisational or established just for one project
- Standards can apply to all aspects of software development, from human management to code layout to documentation
- Standards aid co-operation - we can work more easily with others if we know the standards they are keeping to
- Standards save time - we can agree to use a well-established set of rules for quality rather than develop our own for each project
- Standards are developed from experience - they incorporate practices which have been seen to work well in the past, and practices which will avoid mistakes that were made in the past
- Standards help new staff understand the organisation and provide a framework of good practice for them to work in

Problems with Standards

- Standards can become out of date due to technology changes or changes in experience
- For example, standards may call for action against problems which no longer occur as they were caused by technology which is no longer used
- Experience may show that a safeguard against one problem causes more problems elsewhere
- Standards may be seen as bureaucratic form-filling which wastes time
- Good aspects of standards may not be taken seriously if there are also poor aspects
- So standards should be reviewed regularly, with current practitioners involved in the review and the rationale clearly explained
- Standards should be supported by tools to minimise any extra clerical work, but poor tools can lead to even more overhead

Software Engineering History: Fred Brooks and Ed Yourdon leading to Agile

Fred Brooks: The Mythical Man Month: Essays on Software Engineering

- Published in 1975 (revised second edition in 1995)
- The first book to give strong consideration to the human aspects of software development
- Considered a classic, and still widely quoted
- Based on Brooks’ experience at IBM

Mythical Man Month

- A “man month” is the amount of work one person can do in a month
- If we estimate a project involving P people can be done in M months, could it be done in M/2 months with 2×P people?
- No - some tasks can only be done sequentially (one after the other)
- Classic example “A woman can produce a baby in nine months, can nine women produce a baby in one month?”

More workers, more time!

- Adding more workers to a late software project makes it later
- New workers need to be trained
- Even if new workers have the necessary technical skills, it takes time to understand enough aspects of an existing project in order to contribute effectively
- The more workers there are, the more possible communication there are:
  - ∗ 2 workers - 1 pairwise communication
  - ∗ 3 workers - 3 pairwise communications
  - ∗ 4 workers - 6 pairwise communications
- Better to extend the deadline than attempt to meet it by adding more workers

Ed Yourdon: Decline and Fall

- Yourdon’s *Decline and Fall of the American Programmer* (1992) suggested software should be developed with a heavy emphasis on design
- He suggested programming would be a fairly mechanical process occurring afterwards which would not require high levels of skill
- Following from this, he suggested programming jobs would be lost in the USA, as companies outsourced the programming aspects of development to low-waged and only semi-skilled workers in poorer countries
- He felt unskilled programmers in poorer countries would be more likely to accept strong management

Ed Yourdon: Rise and Resurrection

- Yourdon's *Rise and Resurrection of the American Programmer* (1996) reversed much of what was written in his earlier book
- This was part of a movement returning towards seeing programming as the core aspect of software development
- It developed into the Agile movement
- This comes about from a realisation that high quality code is vital for successful software projects
- High quality code requires skills and experience
- Producing high quality code is a form of creative craftsmanship, programmers who can produce well-structured reliable and efficient code quickly are a valuable and scarce resource

Why the change?

- Too many software projects failing badly
- Poor quality code is very expensive in the long run as most software development is concerned with modifying existing code, not writing new code
- Poor quality code is hard to understand in order to modify
- Bugs are often introduced when poor quality code is modified due to unexpected and hard to follow interactions
- Design and specifications are often ambiguous, programmers need to interact with customers to get it right
- So programmers need good communication skills and good general insight, as well as excellent logical and technical skills

Problems with Outsourcing

- The idea that much programming could be “outsourced” to be done remotely in cheaper locations was undermined by the realisation that software systems are best developed incrementally through interaction with the customer
- Programmers need to be fluent in the human language of the customer to understand requirements and specification documentation
- Programmers need to be fluent in the human language of the customer to document their code
- Long distance communication does not work as well as face-to-face informal communication
- Different cultural assumptions can lead to misunderstandings
- People tend to identify closely with the “team” of people they work with day-to-day, “them and us” attitudes tend to occur with separate teams
- Formal contracts involve an overhead of lawyers, accountants etc

Programmers as craftsmen

- Good programming is creative work, and creative work is almost always best done with “light management”
- Yourdon noted the most productive software groups were those free to choose their own tools, methodologies and working styles
- Good programmers should not be subjected to “command and control” management philosophy
- The most skilled and productive individuals are likely to be those who most resent “bureaucracy”
- New styles of management decentralise as far as possible
- New software tools and programming languages have removed the more mechanical aspects of programming, but require higher skills to use well

Death March Projects

- Yourdon used the term “death march project” to mean a project which is highly likely to fail
- He claimed at least half of all software projects are “death march projects”
- One reason for this is pressure to complete projects within a short timescale in order to beat competition
- Another is inexperienced programmers who are too optimistic about what they can achieve in a particular time
- In a competitive market, teams who put in an unrealistic bid may win the contract over those whose estimates of time and cost are higher but more realistic
- Senior management can be unaware of issues surrounding software development, such as the Brook’s “mythical man-month”
- The invisible nature of software makes it hard for those not directly involved in its development to understand why a project is falling behind schedule or to pick up on warning signals about poor quality

Surviving Death March Projects

- Be realistic - admit when timescales cannot be met, the problem will get worse if promises about full completion on time continue to be made
- Be adamant about trading off time against functionality - better to complete some aspects of the project on time than try to complete all and have nothing completed on time
- Avoid the “heroic hacker” syndrome - programmers working on their own for long hours are likely to produce poor quality code that will cause more problems in the future
- Avoid trying to solve the problems by bringing in new workers
- Avoid trying to solve the problem by using new and unfamiliar tools
- Avoid bureaucracy - heavy management pressure will not solve the problem and is likely to slow it down with more overheads

Agile

It can be seen that the advice given by Ed Yourdon fits in with the principles of using the Agile approach to develop large scale software. A group of experienced and well-known software developers wrote a set of principles for software development, called the Agile Manifesto in 2001.

- Agile uses the technique of producing a working prototype and then developing it by adding new features, rather than detailed planning the final system in advance
- Agile involves continuous interaction with the customer to decide what new features to add next, and to release new versions with the features to enable the customer to experience them
- Agile places emphasis on consideration of the actual code, with continuous testing, close teamwork, and use of good design principles to avoid poor quality and unreliable software

Technical Debt and Refactoring

Taken from Martin Fowler's bliki
http://martinfowler.com/bliki/TechnicalDebt.html

- *"You have a piece of functionality that you need to add to your system. You see two ways to do it, one is quick to do but is messy - you are sure that it will make further changes harder in the future"*
- Technical debt means accepting that you have produced code quickly in order to produce a working system or new features, so that the customer can experiment with them, but realising that the code may not be well structured
- Leaving code poorly structured will lead to higher costs and slowing down later on when the code is further modified
- Putting effort into improving the structure of code is termed refactoring
- Producing code quickly and then refactoring it could be considered like borrowing money to achieve something quickly, and then paying back the debt so that costs do not increase in the long run

Code Smells and Design Patterns

- The term code smell to mean an indication in code of a need for refactoring
- A “code smell” is not necessarily an indication of poor design or coding in the first place, it is a recognition that aspects of design which made sense at one stage can become poor as the code is modified to accommodate new features
- Agile development recommends keeping code simple, so do not introduce complexities into design if they are not immediately needed
- As the code becomes more complex, we may see ways in which we can break the code into parts and generalise it in order to keep to the design principles
- Design patterns are particular ways of structuring code to do this
- However, excessive use of design patterns before the growth in requirements make it useful can make code too complex, which is itself a “smell”

Code Smell Examples

- Long methods: if the code for a method has become so long that it cannot be completely viewed on a screen, that is a sign the method could be broken into several methods
- Large classes: if a class has many methods and fields that is a sign it could be broken into separate classes
- Data clumps: if many methods take the same values as several arguments, that is a sign that putting those values together as fields of an object and passing the object as a single argument would be a good idea
- Refuse Bequest: if a class extends another class, but does not make use of what it inherits, or overrides its methods with code that serves no purpose, that is a sign that inheritance was not a good technique to use for that class
- Duplicate Code: if a class has a lot of code that is similar to that of another class, that is a sign that techniques should be used to avoid duplicated code, in that case it might be appropriate to use inheritance so the two classes inherit code from a superclass

Summary

- Software engineering is about human issues as well as coding issues, we have covered some of the human issues here
- Human issues include both management of people writing the code and interaction with people who are customers who require the software system being developed
- An important aspect of management is that software development requires highly skilled workers
- An important aspect of software engineering is that many aspects of quality are not visible to customers, but could have serious consequences if they are not taken into account
- Agile development involves continuous interaction between programmers and customers
- Programmers need to balance meeting customer requirements with specialist knowledge of how easy it is to meet those requirements, and use this to give advice in agreeing on the priority of requirements