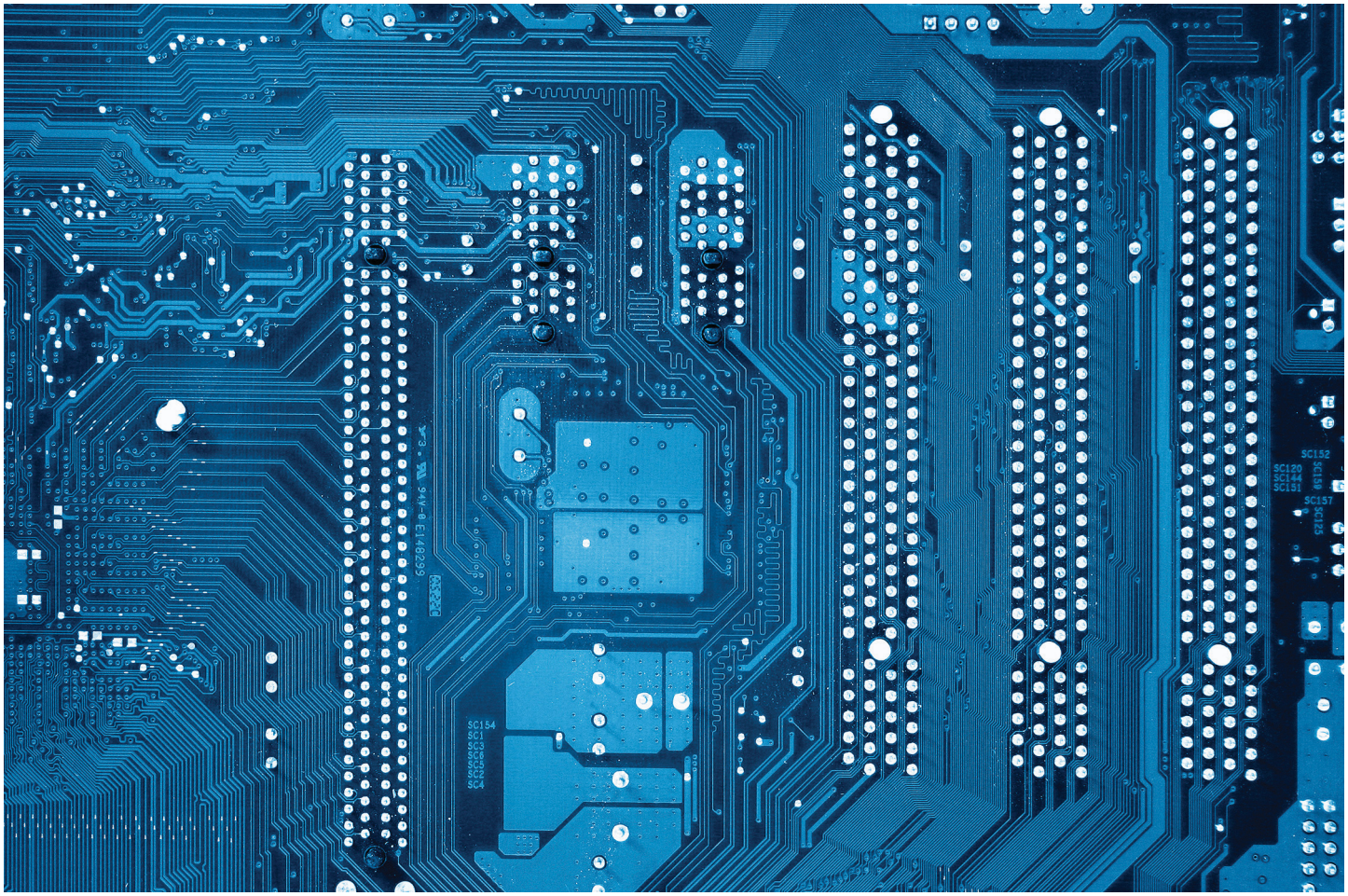
OpenLearn

The Open University

An introduction to software development



About this free course

This free course is an adapted extract from the Open University course M813 *Software development* http://www.open.ac.uk/postgraduate/modules/m813.

This version of the content may include video, images and interactive content that may not be optimised for your device.

You can experience this free course as it was originally designed on OpenLearn, the home of free learning from The Open University: www.open.edu/openlearn/science-maths-technology/introduction-software-development/content-section-0.

There you'll also be able to track your progress via your activity record, which you can use to demonstrate your learning.

Intellectual property

Contents

Introduction

This OpenLearn free course is about software development, the discipline concerned with the methods, techniques and processes of building software artefacts: today's software development professional will need an in-depth knowledge of this discipline to be able to create software which best serves the needs of our digital economy and society. Software development is also a fast-moving discipline, so that the software development professional also needs effective strategies for finding materials that track its leading edge.

This OpenLearn course is an adapted extract from the Open University course M813 *Software development*.

Learning Outcomes

After studying this course, you should be able to:

- appreciate the engineering nature of software development
- describe key activities in software development and the role of modelling
- explain key concepts in software development such as risk and quality
- explain the basics of an object-oriented approach to software development
- describe a simple workflow for interacting with the published literature on software development.

1 Software development as engineering

G. F. C. Rogers, writing in the early 1980s, defined engineering as ‘the practice of organising the design and construction of any artifice which transforms the physical world around us to meet some recognised need’ (Rogers, 1983).

Rogers was not a software developer, but an engineer specialising in thermodynamics and jet-engine development. Yet his definition captures the essence of engineering and has a wide applicability which extends to software. Much of our endeavour in software development is the design and construction of software to meet some recognised need – of people, organisations or society at large – with tangible effect on the real world.

Activity 1

Look at Rogers’ definition of engineering. How would you relate elements of that definition to your understanding of software development?

Discussion

Organising design and construction can be related to what is commonly known as a ‘software development process’: if you develop software in the context of a commercial organisation, it is highly likely that you and your team will follow one such process to coordinate the construction and deployment of software.

The *physical world around us* can be related to the context in which the software will be deployed – the context could be the way an information system is organised, the device in the case of a software controller, or the internet for a web application.

The *recognised need* in Rogers’ definition relates to the problems the software is meant to solve – what we commonly capture as requirements in software development. These problems are the reason why the software should exist, and express how software should affect its context once in operation.

In this course we take an engineering view of software development, and, guided by Rogers’ definition, will explore in some detail the constituent elements of software development, and the interrelations between these elements.

2 Software development processes

With reference to Rogers' definition, a **software development process** is the practice of organising the design and construction of software and its deployment in context.

Effective software development processes remain the holy grail of software development and over the years many contenders have emerged and then gone out of fashion. These vary in the detail of what activities they prescribe, their related artefacts, how the activities should be carried out in relation to one another, and how often each activity should be revisited. There are, however, some fundamental development activities that are well understood and common to most approaches. In very broad terms:

- **Analysis** involves understanding the problem which the software is intended to solve, i.e., the requirements in context, with **validation** as the means to check that understanding.
- **Design** involves describing, conceptually, a software solution that meets the requirements of the problem.
- **Implementation** involves realising such a solution in software.
- **Testing** involves making sure that the solution has certain inherent qualities, with **verification** as a means to check its adequacy with respect to the specified requirements and **validation** as a means to check that the solution does address the problem.
- **Deployment** involves making the developed solution available in its context of operation and use.

Note: **validation** and **verification** are sometimes confused. One way to understand the distinction is to think of validation as addressing the question 'are we solving the right problem?' and verification as addressing the question 'are we solving the problem right?'

Activity 2

If requirements are validated against the problem, and the implemented system is verified against those requirements, then the system should indeed be addressing the problem. Why then do we still need to validate the system against the problem during testing?

Discussion

The core of the problem is that the system needs to satisfy the requirements *in its context of operation*. During software design and construction we rely on assumptions about the properties of that context – which might turn out to be inaccurate once the system is in operation, or we may discover that there are other critical properties which we have overlooked. The context of operation of a software system is often a complex place whose characteristics and behaviour are difficult to grasp and predict completely and accurately, even with the best engineering effort.

While this course will focus primarily on software development activities, it is important to remember that the process does not end with a deployed product. The full **software life-cycle** also includes:

- **maintenance**: makes sure that the deployed solution continues to address the problem throughout the time it is in operation, and handles its **decommissioning** at the end of the life-cycle.

Note: although software maintenance is a well understood term, some authors prefer to talk about **software evolution**, meaning that software continually changes over its life-cycle, with changes often triggered by changes in context and in stakeholders' needs. As a consequence, software development activities typically carry on post-deployment as new releases of the software are developed and deployed.

Activity 3

Do a web search for 'software development processes'. Name three such processes that recur in your search results. What are their key characteristics, and differences with reference to the activities mentioned above?

In completing Activity 3 you may have encountered some of the following.

The **waterfall development process model** (or simply, **waterfall model**; Royce,1970; Benington, 1983) was inspired by manufacturing processes. Historically, this was the earliest software development process to be widely recognised and applied. It relies on the definition of sequential phases of analysis, design, implementation and testing, each starting only after the previous one has finished; in its purest form, all the analysis is done first, followed by all the design, then the implementation, and finally the testing. The waterfall process has been widely used and has yielded many successful software products, and variants of it (e.g. the V-model; see Forsberg and Mooz, 1992) are still in use in some industries, particularly in highly regulated sectors. However, the waterfall process is based on several assumptions that have been questioned. For example, it relies on the existence of a set of requirements that is defined beforehand and remains unchanged during development; it also assumes that all code is designed from scratch, making no allowance for the reuse of existing software. Because the development is carried out as a linear process, the waterfall process does not allow previous phases of the development to be repeated without repeating the whole sequence of steps.

Iterative and incremental processes emerged as a reaction to the recognised limitations of the waterfall process, particularly the latter's rigidity in the face of change and its lack of support for reuse. With iterative and incremental processes there is still a need for analysis, design, implementation and testing activities, but these activities are carried out in a more flexible way than in the waterfall process. Many iterative and incremental processes exist – for example, the spiral model (Boehm, 1988) or the Unified Process (Jacobson et al., 1999) – and all share some basic features: they are 'iterative' in the sense that one or more activities are repeated; and they are 'incremental' in the sense that software development proceeds from an initial subset of the requirements to more and more complete subsets, until the whole system is addressed. Hence, an iterative and incremental process consists of several cycles of analysis, design, implementation and testing, each providing feedback for the next cycle, in which evermore refined and enhanced levels of development are achieved.

A popular family of iterative and incremental processes emerged at the start of this century, covered by the umbrella term of **Agile development** (Cockburn and Highsmith, 2001; Fowler and Highsmith, 2001). These processes encourage continual realignment of development goals with the needs and expectations of the customer, and provide 'lighter-weight', faster and 'nimbler' software development processes that can adapt to the inevitable changes in customer requirements. Hence, such processes promote frequent, small, incremental cycles of analysis, design, implementation and testing. Also, working code is valued above other types of artefact, such as models or other documentation. Extreme Programming (XP) (Beck, 2000) and Scrum (Schwaber and Beedle, 2001) are two well-known Agile development approaches.

Within the same timescale of Agile development, **model-driven-development** (Selic, 2003) also arose. This approach sees models and not code as the key artefacts in software development. According to this approach, models allow the developer to work at a much higher level of abstraction than code, and hence be closer to the business and its processes; also, models are platform-independent and hence decoupled from specific technology. Well defined syntax (how models should be defined) and semantics (how they should be interpreted) allow code to be derived from models through transformations and a suite of automated tools are being developed to support the approach. With model-driven development, the boundary between analysis and design becomes blurred, with development progressing seamlessly through transformed models.

3 Why is software development difficult?

You may be wondering why so many development processes have been proposed, why they are the way they are, or even why we need software development processes at all. The simple answer is that developing software is hard and development processes are regarded as a way to manage the risk of getting it wrong: in 2004 the total cost of information systems failure across the European Union was an estimated €142 billion, while McManus and Wood-Harper's 2008 study of 214 large IT projects found that 23.8% were cancelled before delivery, and of those that ran to completion, 42% overran in terms of time and/or cost. Similar figures are cited in the CHAOS Summary (Standish Group, 2009), which reported that '32% of all projects succeeding (delivered on time, on budget, with required features and functions); 44% were challenged (late, over budget, and/or with less than the required features and functions); and 24% failed (cancelled prior to completion or delivered and never used)'.

So, why is software development so difficult? This is a topic that has exercised software thinkers and practitioners alike for many decades.

3.1 A philosophical perspective

In the late 1980s W. M. Turski (1986) observed that developing software is inherently difficult because it involves descriptions of properties of both formal and non-formal domains. **Formal domains** are those which pertain to the artificial, i.e., the software itself and its hardware – man-made artefacts which can be described using fully formal notations, such as code or statements of logic. **Non-formal domains** are those which pertain to the natural, such as people, organisations or natural phenomena; such domains don't always allow formal description, and even when they do, different linguistic descriptions are used, which raises the further challenge of how to relate them and reason about them. Consequently, Turski sees developing software as compounding the difficulties of mathematics, in its concern with relationships between formal domains, and of natural science, in its concern with descriptions of the properties of non-formal domains. In Turski's view an effective software development process would have to deal adequately with descriptions across the formal/non-formal divide.

Activity 4

Think of various descriptions used in software development. Which do you think pertain to the formal and non-formal domains identified by Turski?

Discussion

You may have considered some of the following.

Code is intrinsically a formal description, usually expressed in a language with formally specified syntax and semantics. **Software design** can also be captured formally, although semi-formal descriptions (e.g., sketches) can also be used to describe it. **Requirements statements** are usually non-formal descriptions, often expressed in natural language, that try to capture properties of formal and informal domains, e.g., how people might interact with the system and how the system might respond in turn. Some **requirements** can be specified using formal languages and notations; this is particularly true in certain domains, as with **embedded safety-critical software**,

although the limitations of formal language for expressing general requirements are widely acknowledged.

Building upon Turski's thinking, M. Jackson and others (Jackson, 1995; Gunter et al., 2000; Hall and Rapanotti, 2003) proposed a view of software development encompassing two distinct but related 'spaces': the **problem space**, in which the problem to be addressed is located; and the **solution space**, in which the software solution resides. In this view, descriptions, whether formal or non-formal, are further categorised into indicative and optative: an **indicative description** is a statement of fact – that is, an assumption which is believed to be true and on which further reasoning can be constructed; an **optative description** is a statement of wish – that is, something we would like to become true at some point. To give an example, a requirement is an optative description, as it states something that it is hoped will be true once the system is designed and operational; on the other hand, an existing business rule is an indicative description, as it states how the business works currently; that is, this is something taken for granted in developing the system.

Note: one way to understand the distinction between optative and indicative is to consider optative descriptions as statements of something we wish to be true in the future, but which may not be true now, and indicative descriptions as statements of something which is true now and which we expect still to be true in the future.

In this view, specifications provide the point of contact between the two spaces: a **specification** captures optative requirements as to how the software system should behave at its interface with the real world. As such, the specification can be validated against the problem: we can check whether building a software system to that specification will address the problem; and the software can be verified against the specification: i.e., we can check that the software has been built in such a way that it will satisfy the specification. In this view, therefore, an effective software development process would have to deal adequately with building and separating optative and indicative descriptions across the problem/solution divide, and transform optative requirements before development into indicative statements once the software is in operation.

Activity 5

Consider the problem and solution spaces proposed by Jackson and others. How would you describe the way the waterfall process spans the two? What about iterative and incremental development processes?

Discussion

The waterfall model assumes that we proceed in linear fashion from the problem to the solution space: first, through analysis, we understand the problem and express requirements and specification; then we proceed to design and implementation in the solution space, and test the outcome against the specification.

On the other hand, iterative and incremental processes assume that we cross the problem/solution divide many times: at each iteration we revisit the problem space to identify new requirements and define new specifications; we then proceed to the solution space for related design, implementation and testing.

The work of Turski and of Jackson and others provides useful insights into the nature of the descriptions required in software development, and processes can be, and have been, defined which establish when and where such descriptions are required. However, there are further dimensions to software development which are widely recognised as being highly influential, and which should be taken into account. One such dimension in those descriptions is **fitness for purpose** – and how this can be established. The other dimension is the **design rationale**, i.e., the explicit articulation and capture of the reasons behind and justifications for decisions made when designing an artefact, including alternatives considered and trade-offs.

Hall and Rapanotti (2009) have proposed a view of software development as spanning two orthogonal spaces: the **design space**, to which the descriptions of Turski and Jackson and others belong; and the **assurance space**, where consideration of design rationale and fitness for purpose take place and related descriptions are generated. In this view, contact between the two spaces occurs throughout the development process, both in the problem and solution spaces, at points at which assurance artefacts are shared and checked by different **stakeholders** – for example, as with a business analyst checking understanding of business processes with the customer, or a software architect reviewing and approving an architectural blueprint for a system. Hence, the fitness for purpose of any description, from problem statement to working code, can only be established by the stakeholders involved in the process, bringing stakeholders to the heart of software development and highlighting their identification and participation in the process as a key success factor. As a consequence, in this view, an effective software development process would have to deal adequately with building and separating descriptions not just across the problem/solution divide, but also across the design/assurance divide, and would also have to identify key representative stakeholders and account for their involvement in the development process.

Activity 6

Consider Hall and Rapanotti's design and assurance spaces. To what extent does the waterfall process address the relationship between these two spaces? What about iterative and incremental development processes?

Discussion

The waterfall model fits primarily within the design space, as it makes no explicit reference to assurance and its relevant stakeholders. The only activity that could be related to the assurance space is testing, which can be interpreted as developing assurance descriptions – the set of test cases – to demonstrate specific properties of the working software, often to the customer. Similar observations can be made of iterative and incremental processes, which have developed directly from the waterfall model.

Historically, software development processes have focused primarily on the design space. However, the importance of the assurance space has long been acknowledged and recognised, and a parallel discipline of project management has developed in a bid to address some of the assurance needs of software development processes highlighted by Hall and Rapanotti (2009). It should be noted that the importance of stakeholders is also explicitly recognised by Agile development, whose manifesto explicitly values individuals and interactions as well as customer collaboration.

3.2 A historical perspective

Manchester University was the home of the world's first stored-program computer, the Small-Scale Experimental Machine – also known as 'Baby' – which ran its first program on 21 June 1948 (Tootill, 1998). Baby's first run was on a 17-line program that calculated the highest factor of an integer. This was not a mathematical problem of much significance, but was chosen as a proof-of-concept because it made use of all seven instructions in Baby's instruction set (Enticknap, 1998).

This trivial problem was, arguably, the beginning of the modern information age. At that time, a simple software process was sufficient: the program was simply crafted by inspection of the problem and debugged until it was correct. Things didn't stay that simple for long. By the early 1990s, **problem complexity** had increased significantly, with **code complexity** increasing in turn by several orders of magnitude; for instance, by 1991 the IBM CICS (Customer Information Control System) had grown to 800,000 lines of code.

A first response to the need to manage higher code complexity was to provide richer and richer structures within the programming languages in which to implement that code. High-level programming languages were developed to provide greater abstraction, often with concepts from the problem domain allowing more complex code to service more complex problems. There was a steady trend in sophistication, leading to the programming languages we still make use of today. Then the concept of software architecture was developed, placing at the heart of design repeated code and thought structures, which were independent of programming language and could be used to structure large systems (Bass et al., 2003).

Coincident with the rise of code complexity was a growing realisation of the importance of an engineering approach to software; here, the traditional engineering disciplines inspired early software development processes, as with the waterfall model. Also, with architecture at the heart of code design, problem complexity started to be addressed through software requirements engineering – see, for instance, Robertson and Robertson (2006). This complemented code design through phases of elicitation, modelling, analysis, validation (i.e., 'are we addressing the correct problem?') and verification (i.e., 'have we addressed the problem correctly?'). Now, software requirements engineering led from system requirements to software requirements specifications on which architectural analysis could be performed.

Things did not go smoothly with early development processes. Already in 1970 Royce saw the need – based on his extensive experience in developing embedded software – to iterate between software requirements and program design (and between program design and testing). For software, the early stages of development could not always be completed before commencing the later stages. Indeed, it was Royce's enhanced waterfall model, able to cope with requirements **incompleteness**, that inspired a generation of iterative and incremental software processes (Royce, 1970).

The waterfall model also assumed that a stable problem exists on the basis of which to begin development. This proved to be adequate in early applications. For instance, a very early business use of computers came about in 1951, when the Lyons Electronic Office (LEO) was first used for order control and payroll administration (Bird, 1994). Early applications typically involved scheduling, reporting and 'number-crunching', and although they provided many technical challenges that were complex for that time, what amounts to a first-mover advantage (Grant, 2007) meant the Lyons company enjoyed a stable environment in which to explore its use of computers. However, the drive for more sophisticated business computing meant that developers now had to deal with the

volatility of the business world and the way this affected its relationship with the formal world of the computer. Therefore, the waterfall assumption soon showed itself to be inadequate for many real-world applications.

Volatility also arises from the nature of technology: rapid technological change also drives changes in the context in which software is deployed, and any connection between the informal world outside the computer and the formal world of the computer must link *two* volatile targets. Due to volatility, the resulting relationship between requirements and architectures is an order of magnitude more complex than for stable systems, and the treatment of it offered by traditional techniques, such as those derived from Royce's processes, are increasingly seen as inadequate. Indeed, such volatility is now recognised as one of the greatest challenges software development faces.

4 Risk

As we have discussed, complexity and volatility are a cause of difficulty in software development and increase the risk of failure: failure to deliver in time and on budget, or to meet stakeholders' needs in context. In turn, failure can have profound consequences: it may lead to tangible harm or losses (e.g. harm to people or damage to goods, loss of contracts, loss of revenue, decrease in market share, penalties in contracts) or intangible harm (e.g. loss of trust, credibility or future business opportunities, damage to the reputations of people, organisations or trademarks, dissemination of confidential information, loss of intellectual-property rights).

Note: risk assessment (how to identify risk) and risk management (how to deal with risk) have been the subject of study in a wide variety of domains, from business to engineering, health or statistics, and a vast literature exists on the subject.

One important role of software development processes is to reduce risk, particularly in relation to two fundamental categories:

- Problem-related risks: these are risks associated with misunderstanding the problem and the stakeholders' requirements, or making inappropriate assumptions as to the context of the system.
- Solution-related risks: these are factors associated with developing software which does not address the identified problem or is not fit for its purpose.

Activity 7

Do a web search on software development risk. Which recurrent factors can you identify? How do they relate to the categories above?

Discussion

Your list will be different from ours. Among the recurrent risk factors we identified are:

- Problem-related risks: incomplete or inconsistent requirements; unachievable goals; wrong assumptions about the domain.
- Solution-related risks: technical infeasibility; breakdown of specification.

5 Software quality

In very abstract terms, the quality of an artefact, whether it is software or otherwise, relates to the presence of positive characteristics of the artefact which will satisfy stakeholders' needs and expectations in a particular real-world context. Quality also relates to the absence of negative characteristics which may harm or frustrate stakeholders and fail to meet their needs and expectations in that context. It follows that quality is not an absolute property: it can only be judged and measured with reference to stakeholders' needs and expectations within their context. This is particularly true of software, which is more often than not developed to address real-world problems as part of much wider systems, usually involving people, processes and other forms of technology.

So quality has to do with **fitness for purpose**, and the hallmark of a good quality product is that its diverse stakeholders are satisfied with it both now and in the foreseeable future. Managing software quality is then a continuous process aimed at ensuring that the software's fitness for purpose is maintained throughout the software's life-cycle.

Failure to attain fitness for purpose can have disastrous consequences, as witnessed by the many examples of software failure which have been reported over the years.

Activity 8

Do a web search on software failures. What did you find?

Discussion

You may have found plenty of high-profile software failures reported by the media and in the academic literature. We found a long list of postings and articles. For instance, Lake (2010) summarised 11 'epic' software failures attributable to deliberate decisions by programmers (rather than, say, faults or malicious attacks), in fields ranging from space exploration and telecommunications systems, in which the loss was mainly financial, to medical and aviation applications, in which the result was a catastrophic loss of lives.

Activity 9

What do you think the quality expectations for an air traffic control software system might be? What about those for a personal diary mobile app?

Discussion

As lives depend on it, an air traffic control system will need to be highly dependable and safe to operate, and provide accurate and reliable information in real time which can be readily interpreted and understood by highly trained people – particularly the air traffic controllers who need to direct pilots. Lack of integrity of data or unavailability of the system could have catastrophic consequences.

Some aspects of dependability – such as data integrity and reliability – would also be desirable but not nearly as critical in a personal diary app. Ease of use would also be desirable as the app is likely to be used by people with all levels of skills.

While quality expectations might be different in different domains, there is a common need to build quality into product design, to increase the likelihood that the product is fit for purpose. Here is yet another role for software development processes: while there is no absolute guarantee, process quality is likely to lead to product quality, and the concern for achieving and maintaining quality needs to permeate all software activities throughout the software life-cycle.

Note: this is in line with Hall and Rapanotti's view of assurance as a key factor in the development process.

Related to software quality is a distinction which is often made between the functional and non-functional characteristics that fit-for-purpose software should exhibit. Many such characteristics have been identified and variously classified in the literature.

Note: ISO/IEC 9126-1:2001, an international standard for software product quality, provides one such classification.

Some of these characteristics – those which are most relevant to this course – are listed below.

- **Functional fitness** determines the extent to which a system does what the customer wants and needs.
- **Usability** determines the effort required to learn about, operate, prepare input for and understand the output of a system – that is, how easy the system is to use.
- **Flexibility** determines the effort required to modify an operational system – that is, how easily the system can be changed while in service.
- **Testability** determines the effort required to test a system to ensure that it carries out its intended function – that is, how easily the system can be tested to show that the customer's requirements have been met.
- **Reusability** determines the extent to which a system (or system component) can be reused in other applications – that is, how easy it is to reuse some of the software to make future developments more cost-effective.

Activity 10

Based on your own experience and practice, in what way do you think the various activities forming part of the software development process may impact on the quality of the released product?

Discussion

You may have covered some of the following points.

In analysis, validation of problem understanding and of what is required of the system is key to quality, as this determines both the desired functionalities of the system and its many non-functional characteristics. Eliciting knowledge directly from stakeholders, e.g., users or businesspeople, and discussing our assumptions explicitly with them would increase confidence in the accuracy of our understanding.

During design and implementation, following tried-and-tested approaches, good practice and sound principles increases the chance of achieving high-quality software. Access to experienced developers able to review the software is also a way to maximise quality all round.

Comprehensive and systematic testing is essential to reduce the rate of occurrence of software defects and generally verify overall software correctness. Various levels of testing exist – some involving a wide range of stakeholders, including end users. In some domains – for instance, safety-critical applications – the involvement of independent certification authorities might also be required.

In maintenance, it is important to track and correct defects in the software which become apparent after deployment, possibly providing end users with means to report errors easily or make suggestions for product improvement.

Finally, while an appropriate investment in quality processes is important to software quality, a quality culture is also required: an organisational environment in which the concern for quality is highly valued and where there is support for developers who care for all aspects of quality – even the less tangible ones such as code readability or adequacy of documentation. A **quality culture** is characterised by highly professional behaviour and ethical decision-making on the part of those involved in the process, and by the allocation of an adequate level of resources for specific activities which contribute to software quality: it is widely acknowledged that some software failures can be traced back to excessive pressure from management to release a product which had not been properly verified and validated, or to situations where developers were forced to cut corners in the interest of quicker delivery. A quality culture implies an effective and mature use of estimation, planning and risk management, and an integrated view of both human-related and hard technological issues.

6 Modelling and the UML

> ‘Essentially, all models are wrong, but some are useful.’
>
> (George Box)

Models are forms of description often adopted in software development. They are abstractions used to represent and communicate what is important, devoid of unnecessary detail, and to help developers deal with the complexity of the problem being investigated or the solution being developed.

Modelling is used in other forms of design and engineering. For instance, architects develop different models of buildings – some addressing structures, others materials, others ergonomics, and so on. The same happens with modelling in software development: some models are used to capture properties of the problem domain, such as key elements of the business or how its processes work, while other models are used to consider different aspects of the software, such as how the code is divided up and organised, or how various elements of the software communicate and work together. Each model is an abstract representation of some view of the system, and such views may change as the development process unfolds.

Activity 11

Consider different types of model you have encountered in your own practice of software development. What was their purpose? How were they expressed?

In software development, we build models from different perspectives. In particular, we can distinguish between the following modelling types.

Domain modelling is concerned with understanding and modelling context information for a specific problem, independently of a decision to use a software system to deal with that problem. A domain model is a representation of the main concepts in the real-world problem context – for instance, a business under consideration. A domain model does not necessarily assume a software solution.

Specification modelling assumes that a software system will deal with the need in context. A specification model represents software elements used in the software solution to a problem, and is mainly concerned with the definition, at a high level of abstraction, of the services provided by the software.

Design modelling describes the software system itself, with the allocation of responsibilities to its various parts, and its behaviour and control flow.

Note: the distinction between ‘domain models’ and ‘design models’ was introduced by Cook and Daniels (1994). They called the domain model the ‘*essential* model’, and the design model the *‘implementation* model’. ‘*Business* model’ is also often used in the literature with a meaning similar to domain model.

Domain modelling plays an important role in understanding the need in context, before suggesting any software solution. The important elements of the context, whether they are people, products, departments, sensors, alarms or operators, need to be identified.

Only by understanding the need in context, and how, why and with what consequences that need changes, can a fit-for-purpose software solution be specified. However, modelling the domain may sometimes be unnecessary – for instance, when there is an accepted need for a well-defined software system to solve a well-understood problem. In such a case, modelling the domain would not bring much advantage.

Domain and specification modelling may produce very similar models, but the interpretations of the models are different; the former types of model are about real-world entities, the latter are about software representations of those entities.

Activity 12

How would you describe the relation between these three modelling perspectives and the traditional split between analysis and design in software development?

Discussion

You may consider domain modelling and specification modelling as analysis activities – that is, concerned with understanding what a problem is, and specifying what is required of the software system to be developed. As you might expect, design modelling corresponds to the design activity.

Models are usually constructed by following specific linguistic conventions, often referred to as **techniques**, and the models' level of formality will depend on the formality of those conventions; for instance, models can be based on narrative, diagrams or even mathematics.

Well known modelling techniques in software development are defined under the**Unified Modelling Language (UML)**. UML is one of the most popular and successful standards currently used by the software industry. UML is the result of the merging of several notations that appeared during the 1980s and early 1990s, augmented by new techniques and even a mechanism to extend the notation further. You will not learn UML in its entirety, but concentrate on a subset of the most commonly adopted techniques.

Note: to see the UML specification, click here.

It is important to note that UML is a modelling language, not a development process – it gives you a set of techniques, but does not prescribe whether or how these techniques should be used during development. In fact, because UML is the result of an exercise in unification, it can be used flexibly within many different processes and practices, and the same technique can serve different purposes. Also, UML models can be constructed at different levels of precision. UML can be used equally effectively both as a sketching notation, for example to jot down ideas or communicate them among stakeholders, and for precise description of aspects of software systems, for example as semi-formal specification of system functions.

Brief UML history

In the early 1990s, various object-oriented methods appeared along with a proliferation of techniques and notations. By the mid-1990s two authors of well-known methods – James Rumbaugh, one of the authors of the Object-Modelling Technique (OMT), and Grady Booch, author of the Booch method (Rumbaugh et al., 1991; Booch, 1994) – joined forces

in an attempt to unify their methods. They named their joint effort the Unified Method. Rumbaugh and Booch were soon joined by Ivar Jacobson and his colleagues, authors of Object-Oriented Software Engineering (OOSE), in the development of what became known as the Unified Modeling Language (UML); see Jacobson et al. (1992). As UML has evolved it has incorporated feedback from the object community, and has won the support of many people and organisations which considered the idea of a unified modelling language a valuable one. UML was then submitted for standardisation to the Object Management Group (OMG), a consortium of several large software companies that produces and maintains computer-industry specifications. UML was formally adopted in 1997. The UML specification is constantly updated. At the time of writing (2013), UML 2.5 is near completion; this is the basis of the notation we use in this course.

7 Object orientation

With the growth in code complexity came the idea of breaking up large sequential programs into more manageable, and smaller, named pieces of code. Initially these tended to represent *actions*, which in turn could be broken down into smaller and smaller actions, until a unit was reached that was small enough to understand, write and test independently. This type of code decomposition proved a useful device and became the basis of procedural and functional styles of programming. Some of these are still in use, although we will not consider them in this course.

The early1960s saw the idea of structuring software around the data structures or *objects* that were manipulated, rather than around the actions that manipulated them. This was the start of object-oriented programming, which subsequently developed into a set of concepts, principles and techniques used throughout software development to analyse software problems and design and construct software solutions. We use the term **object orientation** to refer to this holistic approach.

You should note that because of its roots in programming, object orientation has a distinct bias toward solution space descriptions. Nevertheless, over the years useful techniques applicable across the problem/solution divide have been developed.

Although object orientation is relevant to many domains of application, it is seen as particularly advantageous in business computing. It was observed that as businesses change, the things their computer systems manipulate remain fairly stable, even though the ways these systems are used may change rapidly. For instance, commercial systems will most likely always need to represent taxes, goods, payments, prices and deliveries, even though the business rules determining who is creditworthy, how orders are taken, and whether payment is required before delivery may all change frequently. Therefore, one advantage of structuring software around objects meaningful to the business is that it gives more stability and better traceability from the business to the code.

7.1 Modelling with objects

In object orientation, modelling is very much focused on constructing descriptions based on sets of interacting objects.

For instance, when exploring a real-world problem we identify the important objects which relate to that problem and build models in terms of these objects and their interactions. The purpose here is to get a better understanding of the domain, whether to elicit requirements or to understand the business needs that those requirements address. Ideally, the classes of object in our models should be chosen to correspond to natural categories of things in the real world – such as ‘customer’, ‘invoice’, ‘payment’, ‘bill’ – so that there will be a structural similarity between the real world and the software. This approach can lead to good traceability from problem descriptions and requirements through to code.

Later on, once we have assumed that a software system is needed to address the problem, we may start modelling elements of the solution as software representations of the real-world objects. Initially the emphasis is on the services provided by the software solution and not on how these services are provided.

As design progresses, modelling describes the software system in greater and greater detail, with the allocation of responsibilities to different software objects so that the functionality of the entire system is distributed among the classes. In object terms a design model will represent the software classes and objects, the messages exchanged between them and their order.

Note, however, that structuring the software around objects *per se* does not guarantee quality software as an outcome. This can only be achieved through the judicious application of sound design and construction principles, and indeed object orientation comes equipped with a rich set of principles and tried-and-tested design solutions to recurrent design problems.

7.2 Programming with objects

To represent real-world objects such as an account or a payment in software we construct units of code, called **objects**, which assemble both data and operations that can act on those data.

Note: this concept is in turn an evolution of the idea of programming with abstract data types (Liskov and Zilles, 1974), which developed in the 1970s.

For instance, an object representing a bank account in software may include both a variable holding the account's current balance and a number of operations for credit and debit transactions. Moreover, we expect that only those operations can access the data directly: other software wanting to affect the current balance would have to make use of those operations via an established **interface**. This particular feature of an object is called **encapsulation**.

Activity 13

What do you think the advantages of encapsulation might be?

Discussion

Encapsulation provides an explicit boundary separating information about which operations are available and information on how such operations are implemented. In our example, outside the account object, available operations are advertised via their interface only, which indicates how they can be invoked. The implementation of the operations, i.e. how they materially access and modify data, and how the data are represented through variables, are not visible outside the object. In this way, as long as the interface remains unchanged, the operation implementation or the data representation can be modified without affecting other software.

In most programming languages an object is defined as an **instance** of a homogeneous **class**, with the class providing an abstraction for the common characteristics of all its instances. For example, a class-defining current-account object might indicate that each account will have a distinguishing account number and a balance, and that money can be withdrawn or deposited; on the other hand, each of its instances will have its own distinct account number and balance reflecting how much money is in the corresponding account.

Activity 14

What do you think the advantages of distinguishing classes and objects might be?

Discussion

Classes allow for more succinct code and foster code reuse and flexibility. By defining a class we are defining the properties and behaviour of all the objects of that class that are created during software operation; when a change is needed, only the class code needs to change, with such changes automatically propagated to all its instances.

No serious software system consists of a single object. Instead there will be a collection of objects which collaborate to achieve the functionality of the whole system. **Collaboration** occurs when one object requests a service from another object in order to perform some task. To fulfil a particular collaboration, each object takes on a different role: the object that makes the request can be seen as the **client**, and the object that receives the request (and provides the service in response) as the **supplier**. The request is communicated to the supplier from the client by **message passing** based on the objects' interfaces: the client sends a **message** to the supplier which on receipt executes a corresponding operation; when the supplier has completed the execution of its operation it returns a **message answer** to the client. Both the message and message answer may convey data between the objects, with the interface specifying the type of the arguments and the result of the operation.

Object collaboration through message passing is at the core of computation in an object-oriented software system, and many forms of collaboration can be forged between objects. In software development, a range of principles and techniques have been defined to design and implement object collaborations resulting in quality software.

8 Finding and reading academic articles

The area of software development is fast-moving and as a software development professional you cannot rely only on prepared materials to perform your role, but must have strategies for finding materials that track the leading edge of your subject. Finding relevant material is not a trivial task. There are myriad sources, including academic and professional journals, conferences, technical reports, blogs, podcasts, newspaper and magazine articles – the list is almost endless. Once your information resources have been found, there's still the investment of time needed to sift through the various resources, which will be of varying relevance, so as to extract the best information in the least amount of time. Then the task of understanding begins. Even once you've understood the information contained in a resource (or think you've understood it), much of this information's value comes from relationships it will allow you to build between other resources. Recording and developing those relationships – building your own literature – is therefore also worthwhile.

In this last part of the course you will learn about a simple workflow for interacting with the published academic literature on software development so as to underpin your professional development and practice.

8.1 A workflow for reading the academic literature

The academic literature consists of papers and other resources, some of which you may wish to read. To be read they need to be found. This section presents a simple workflow by means of which the academic literature can be accessed.

There are five stages to this workflow:

Stage 1: Preparation

Stage 2: Discovery

Stage 3: Assimilation

Stage 4: Recording

Stage 5: Relating

These can be arranged as in Figure 1.

We will consider each stage in turn. As you work through this process you will find that the assimilation stage itself consists of another workflow.

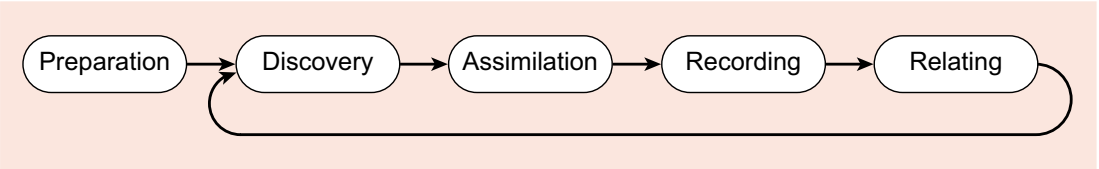


Figure 1 The M813 workflow for reading academic literature

The simple workflow outlined in this section can underpin research in many areas of computing and in other science-based disciplines. The workflow is lightweight but structures any future research you may do – for instance, as part of other taught or

research modules – and will underpin your studies as you progress through the programme.

8.1.1 Preparation

There are many tools available for creating and managing bibliographies. Indeed, you may already use a bibliographic database management system (BDMS) such as Zotero, EndNote, BibDesk or RefWorks. If you already use a BDMS then you are free to use it in this part of the course.

BDMSs are complex tools that can hold thousands of academic papers, white papers and many other information-resource items, keeping them for easy reference. As well as storing resources, some BDMSs allow an electronic version of an information resource to be kept alongside notes you have made about that resource. The resource and any associated notes are then ready for use when you need to cite the material. In some document preparation systems (e.g. Microsoft Word) this can be done automatically.

Even the simplest BDMS requires great flexibility and is a very complex piece of software. The learning curve can be quite steep. For this reason, if you don't already use a BDMS we suggest that you create a simple word-processing document to store the details of the few papers you might find useful to go alongside this course - perhaps something from the references that interested you.

If you already use a BDMS, you can create a new database for this course. Alternatively, you could simply copy and paste the template below into any Word/text-processing package of your choice for each new information resource that you keep in your bibliography.

CiteKey:

Title:

Author(s):

Format:

Publisher:

Year:

URL and last accessed:

Referenced by:

Abstract:

Notes:

Activity 15

What do you think the fields in the template capture?

Discussion

The template consists of a single table with the following fields (this structure is a subset of any commercial tool's structure, although in such a tool there may be many other fields too):

- CiteKey: a unique identifier (or primary key) for the entry
- Title: the title of the work

- Author(s): the author(s) of the work, presented if necessary as a list
- Format: the type of resource, i.e. whether a journal, magazine, technical report, commercial white paper, web page, personal communication, etc.
- Publisher: the publisher of the resource
- Year: the year the resource was published or the year it was last checked
- Pages: the page numbers, if applicable
- URL and last accessed: the web location where the resource was found, and the date on which you last accessed it
- Referenced by: which other item(s) in your BDMS (if any) suggested this information resource
- Abstract: a copy of the abstract for the resource, should one exist
- Notes: the notes that you have made on the resource.

8.1.2 Discovery

Discovering relevant academic literature is becoming easier and easier as most bibliographical collections are now accessible online. Most libraries have subscriptions which allows users to access copyright published materials: if you are currently studying an OU module, you should explore the rich catalogue of sources and resources which are available to you through the OU Library. Search engines also exist which specialise in searching academic literature online. Once such engine is Google Scholar, which you will use in the next activity.

Activity 16

Use Google Scholar from your web browser to search for Srinivasan Keshav's article titled 'How to read a paper': this is an excellent, easy-to-read introduction to an effective workflow for reading many research articles in the field of science and management (as opposed to, for instance, arts). Once you have found it, record as much information as you can in your newly created BDMS: you'll enhance this record later on, in the fourth step of your workflow, after engaging with the material.

Discussion

Hopefully you managed to find the paper without too much difficulty. The paper is included in a print journal and its full bibliographical reference is:

> Keshav, S. (2007) 'How to read a paper', *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 3, pp. 83–4.

This can be recorded using our template as:

CiteKey: Keshav2007How

Title: How to read a paper

Author(s): S. Keshav (keshav@uwaterloo.ca)

Format: Academic paper

Publisher: ACM SIGCOMM Computer Communication Review

Year: 2007

Pages: 83-84

This published version is copyright materials and its text can only be accessed if you are a licensed user, for instance an OU student using the OU Library. For this course, however, we have obtained permission from the author to reproduce the content of the article, which you can find in the Appendix.

Note: that, in general, Google Scholar will not necessarily find everything that could be available, as some publishers do not let Google's crawlers see their content. However, more significantly, Google Scholar may actually return a preprint or draft version of an article rather than the final published piece. Books are sometimes retrieved as a digital part-version rather than the full text.

8.1.3 Assimilating an academic article

Some of the most challenging information resources that exist come from the academic literature: even the most experienced researcher will not expect to read an academic paper once and immediately understand it. Accordingly, the workflow for assimilating the information is complex. In this subsection you will explore this complexity by using a workflow suggested by the author of the paper you have just 'discovered'.

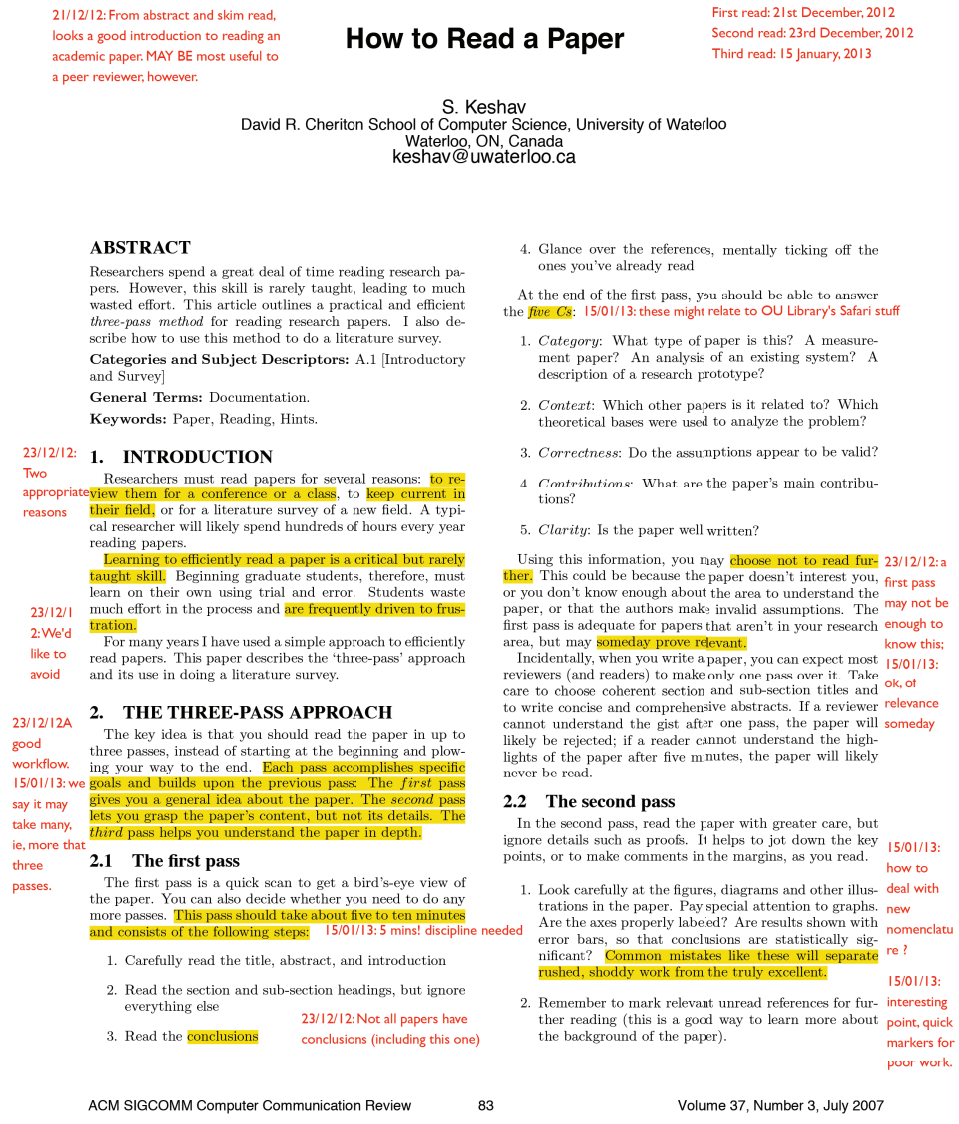
The barriers to understanding a paper are many and high. An academic paper is considered good if it makes a contribution to knowledge; therefore, to understand an academic paper is to understand the contribution to knowledge that it makes. This may require a substantial investment of time and be very challenging.

Understanding an academic (or other) article may require you to understand a new concept or new terminology, to look at an existing model, concept or 'thing' from a new perspective, or to see a group of things in a new relationship. Accomplishing tasks such as these will very probably take more than one reading.

If you were on holiday – for instance – you might be able to dedicate a full weekend or a week to an academic article and read it through as many times as you need. However, you will probably not always be able to dedicate a block of time to the many readings an individual academic paper might need, and so you must find a way of recording your understanding of a paper from one reading to the next.

Annotation

Here are the annotations we made on the published version of Keshav's paper. They appear in red and as electronic highlights in yellow.



21/12/12: From abstract and skim read, looks a good introduction to reading an academic paper. MAY BE most useful to a peer reviewer, however.

First read: 21st December, 2012
Second read: 23rd December, 2012
Third read: 15 January, 2013

How to Read a Paper

S. Keshav
David R. Cheriton School of Computer Science, University of Waterloo
Waterloo, ON, Canada
keshav@uwaterloo.ca

ABSTRACT

Researchers spend a great deal of time reading research papers. However, this skill is rarely taught, leading to much wasted effort. This article outlines a practical and efficient *three-pass method* for reading research papers. I also describe how to use this method to do a literature survey.

Categories and Subject Descriptors: A.1 [Introductory and Survey]

General Terms: Documentation.

Keywords: Paper, Reading, Hints.

23/12/12: Two appropriate reasons

1. INTRODUCTION

Researchers must read papers for several reasons: to review them for a conference or a class, to keep current in their field, or for a literature survey of a new field. A typical researcher will likely spend hundreds of hours every year reading papers.

Learning to efficiently read a paper is a critical but rarely taught skill. Beginning graduate students, therefore, must learn on their own using trial and error. Students waste much effort in the process and are frequently driven to frustration.

23/12/12: We'd like to avoid

For many years I have used a simple approach to efficiently read papers. This paper describes the 'three-pass' approach and its use in doing a literature survey.

2. THE THREE-PASS APPROACH

23/12/12A good workflow. 15/01/13: we say it may take many, ie, more that three passes.

The key idea is that you should read the paper in up to three passes, instead of starting at the beginning and plowing your way to the end. Each pass accomplishes specific goals and builds upon the previous pass: The *first* pass gives you a general idea about the paper. The *second* pass lets you grasp the paper's content, but not its details. The *third* pass helps you understand the paper in depth.

2.1 The first pass

The first pass is a quick scan to get a bird's-eye view of the paper. You can also decide whether you need to do any more passes. This pass should take about five to ten minutes and consists of the following steps: 15/01/13: 5 mins! discipline needed

1. Carefully read the title, abstract, and introduction
2. Read the section and sub-section headings, but ignore everything else
3. Read the conclusions

23/12/12: Not all papers have conclusions (including this one)

4. Glance over the references, mentally ticking off the ones you've already read

At the end of the first pass, you should be able to answer the *five Cs*: 15/01/13: these might relate to OU Library's Safari stuff

1. *Category*: What type of paper is this? A measurement paper? An analysis of an existing system? A description of a research prototype?
2. *Context*: Which other papers is it related to? Which theoretical bases were used to analyze the problem?
3. *Correctness*: Do the assumptions appear to be valid?
4. *Contributions*: What are the paper's main contributions?
5. *Clarity*: Is the paper well written?

Using this information, you may choose not to read further. This could be because the paper doesn't interest you, or you don't know enough about the area to understand the paper, or that the authors make invalid assumptions. The first pass is adequate for papers that aren't in your research area, but may someday prove relevant.

23/12/12: a first pass may not be enough to know this; 15/01/13: ok, ot relevance someday

Incidentally, when you write a paper, you can expect most reviewers (and readers) to make only one pass over it. Take care to choose coherent section and sub-section titles and to write concise and comprehensive abstracts. If a reviewer cannot understand the gist after one pass, the paper will likely be rejected; if a reader cannot understand the highlights of the paper after five minutes, the paper will likely never be read.

2.2 The second pass

In the second pass, read the paper with greater care, but ignore details such as proofs. It helps to jot down the key points, or to make comments in the margins, as you read.

15/01/13: how to deal with new nomenclature ?

1. Look carefully at the figures, diagrams and other illustrations in the paper. Pay special attention to graphs. Are the axes properly labeled? Are results shown with error bars, so that conclusions are statistically significant? Common mistakes like these will separate rushed, shoddy work from the truly excellent.

15/01/13: interesting point, quick markers for poor work.

2. Remember to mark relevant unread references for further reading (this is a good way to learn more about the background of the paper).

Figure 2 Our annotation of Keshav's article (Keshav, 2007)

Activity 17

By looking at the text of the notes, identify and list the three dates on which the paper was read. What do you think we were trying to convey with the annotation?

Provide your answer...

Discussion

The dates are:

- First read: 21 December 2012
- Second read: 23 December 2012
- Third read: 15 January 2013

The notes appear to record observations on the workflow, and any differences from what the reader had expected. For instance, in two cases an initial note is followed up with a further note. The first case has:

- '23/12/12: A good workflow.'

followed up by:

- '15/01/13: we say it may take many, i.e., more that [sic] three, passes.'

The second note has:

- '23/12/12: a first pass may not be enough to know this.'

followed up by:

- '15/01/13: ok, of relevance someday.'

The first pairing of notes suggests that in the reader's experience, you sometimes have to read an academic paper several times before you really understand it.

The second pairing of notes shows the reader's growing understanding of what the purpose of the first pass is.

Keshav (2007) advocates following a workflow for reading an academic paper. There are three passes to this workflow, and the subsections that follow will examine these as a way of assimilating the information presented.

8.1.4 Keshav's first pass

Keshav writes (emphasis added):

> The first pass is a quick scan to get a bird's-eye view of the paper. You can also decide whether you need to do any more passes. This pass should take about *five* to *ten* minutes and consists of the following steps:
>
> 1. Carefully read the title, abstract, and introduction
> 2. Read the section and sub-section headings, but ignore everything else
> 3. Read the conclusions
> 4. Glance over the references, mentally ticking off the ones you've already read.

Keshav adds that at the end of the first pass, the reader should be able to answer 'the five Cs', which are:

- *Category*: What type of paper is this? A measurement paper? An analysis of an existing system? A description of a research prototype?
- *Context*: Which other papers is it related to? Which theoretical bases were used to analyse the problem?
- *Correctness*: Do the assumptions appear to be valid?
- *Contributions*: What are the paper's main contributions?
- *Clarity*: Is the paper well written?

Activity 18

Apply the first stage of the workflow to Keshav's own paper: carry out a first pass and see if you can answer the five Cs, i.e., *Category*, *Context*, *Correctness*, *Contributions* and *Clarity*.

Provide your answer...

Discussion

We came up with the following. The five Cs are:

Category: Keshav's paper analyses a process that researchers can use to read an academic article. The paper is an experience report of the reader, in consultation with some academic colleagues. It provides a contribution to practice.

Context: the paper does not include a bibliography of academic papers; rather, there is a list of website resources that relate the paper to other contributions to practice in this area.

Correctness: assumptions stated in the abstract are:

- that researchers spend a great deal of time reading research papers
- that this skill is rarely taught, which leads to wasted effort.

Contributions: the paper's main contribution is that it outlines a practical and efficient three-pass method for reading research papers. There is a subsidiary contribution: it describes how to use the three-pass method to undertake a literature survey.

Clarity: as is evident from the headings used, the paper's structure is generally logical. However, there is no conclusion, so we struggled to know what to do to complete the first pass.

At this point in the process, given the five Cs, you should be able to decide whether whatever paper you are reading is interesting and relevant. As for Keshav's paper, since we will be using it for the remainder of this section, we can answer the question as to where it is interesting or relevant with a resounding 'yes' (whether you actually find the paper interesting is irrelevant at this point).

8.1.5 Keshav's second pass

Keshav says that the second pass should take about an hour. The goal of the second pass is to be able to summarise the 'main thrust of the paper, with supporting evidence, to someone else', i.e., to be able to reconstruct the reasoning that the paper uses in its contribution to knowledge or practice.

Keshav advises the reader to take greater care on the second pass, but to ignore highly technical details such as proofs, implementations or detailed justification of any argument – indeed, anything that will extend outside of the one-hour window you have to read the paper a second time.

Keshav proposes indicators of a paper's quality. For him, figures, diagrams and other illustrations should have been carefully set out so as to give the reader all the information he or she needs.

Finally, Keshav suggests marking interesting references, so that the portion of the literature upon which the arguments of the paper rest can be read (see Figure 3).

8. REFERENCES

[1] T. Roscoe, "Writing Reviews for Systems Conferences," http://people.inf.ethz.ch/troscoe/pubs/review-writing.pdf.

[2] H. Schulzrinne, "Writing Technical Articles," http://www.cs.columbia.edu/ hgs/etc/writing-style.html.

23/12/12: Interesting

[3] G.M. Whitesides, "Whitesides' Group: Writing a Paper," http://www.che.iitm.ac.in/misc/dd/writepaper.pdf.

[4] ACM SIGCOMM Computer Communication Review Online, http://www.sigcomm.org/ccr/drupal/.

Figure 3 The references list at the end of Keshav's paper (Keshav, 2007)

Activity 19

Do a second pass through Keshav's paper. Try to explain your understanding of the paper to someone else.

Discussion

In trying to explain Keshav's work, we came up with the workflow in Figure 4 (here's a scalable PDF download of this image).

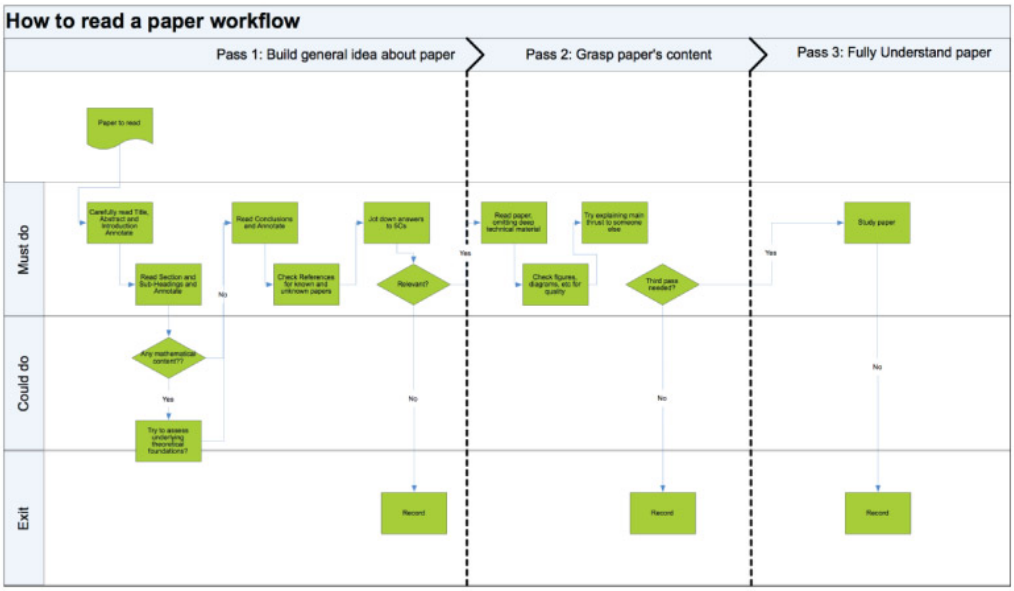


Figure 4 Our diagrammatic representation of Keshav's workflow

This paper's workflow depends upon various conventions being followed by an academic article and which are to all intents and purposes followed in the sciences (including software development as an academic discipline). In other areas of software development such conventions may not necessarily hold – for example, commercial (or white) papers may deviate.

The conventions assumed of a written academic item are that it will have:

- *An abstract*, which is a brief summary of the item – whether it be a research article, thesis, review, conference proceeding or any in-depth analysis of a particular subject or discipline. The abstract is used to help the reader quickly ascertain the item's purpose.

- *A number of sections*, beginning with an introduction and ending with conclusions and/or a discussion.
- *A bibliography*, essentially a list of literature and other material, such as web pages (and even personal communications between researchers) to which the reader of the article can refer to check the arguments and evidence that others have presented and which form the paper's basis.

8.1.6 Keshav's third pass

It may be that, for Keshav's paper at least, you've already gained enough understanding to make a third pass unnecessary. However, for most academic papers this won't be true – moreover, for some of the most complex research articles a third, or even fourth, fifth or sixth pass still will not suffice. Academics have been known to spend their whole working lives trying to understand (and build on) the research covered by a single paper.

Keshav offers some very useful observations:

> Sometimes you won't understand a paper even at the end of the second pass. This may be because the subject matter is new to you, with unfamiliar terminology and acronyms. Or the authors may use a proof or experimental technique that you don't understand, so that the bulk of the paper is incomprehensible. The paper may be poorly written with unsubstantiated assertions and numerous forward references. Or it could just be that it's late at night and you're tired.
>
> You can now choose to: (a) set the paper aside, hoping you don't need to understand the material to be successful in your career, (b) return to the paper later, perhaps after reading background material or (c) persevere and go on to the third pass.

Decision criteria for a third pass

As Keshav argues, not all papers require a third pass and you should take care in deciding whether a particular paper should have one at this particular point. We say this because the importance of understanding a paper may only become apparent later – it may turn out to be a highly cited paper that needs to be understood so that other papers become accessible; or, it may be that a career change on your part means you need to come back to it. Here we see the motivation for annotating a paper and recording your present understanding: if you find you do need to return to the paper at some point, you won't start from scratch if you've made good notes. This also makes clear why you should keep a master version of a paper, with your earlier notes stored with it.

If you decide that investing time in a third pass of a paper is indeed necessary, Keshav's key skill is to attempt to 'virtually re-implement the paper' – that is, making the same assumptions as the authors: recreate their work. This means being able to restate their assumptions, and to recreate the reasoning they use to reach their conclusions. Essentially, you are asking whether, starting at the same point, you could have made the same contribution to knowledge or practice as the authors. *As an aside, that's a good definition of truly understanding an academic paper.*

There are many reasons why you might not have been able to make the same contribution, by the way – and not all of them point to a lack of understanding (or creative genius) on your part:

- perhaps the paper has failings, makes inappropriate assumptions, and/or fails to cite other work that you know of
- perhaps the arguments the paper puts forward are circular and/or nonsensical
- perhaps the research method used does not make sense.

Keshav adds: 'During this pass, you should also jot down ideas for future work'. This is an excellent idea, should you be a researcher (as Keshav is), or a practitioner, and you may wish to identify projects or the like to which the understanding you have gained can be applied.

According to Keshav, the third pass will take about four or five hours for a beginner, and about an hour for an experienced reader. However, longer periods are quite possible: indeed, whether even more passes are required will very much depend on how critical to you the paper is.

Activity 20

Carry out a third pass on Keshav's paper, and then try to construct a virtual re-implementation of the paper.

8.1.7 Recording

Recording your understanding of an academic paper is an important aspect of reading it. Being able to relate it to the literature you already know – that is, *contextualising the paper in the literature* – is another. At the moment, of course, your knowledge of the academic (and other) literature may be very small – or indeed non-existent. Everyone has to start somewhere. Here you will begin recording your understanding of the academic literature in your BDMS.

Activity 21

Having read and annotated Keshav's paper, update the entry in your BDMS with your notes.

Discussion

Having added the abstract and our notes, our own entry now reads:

CiteKey: Keshav2007How

Title: How to read a paper

Author(s): S. Keshav (keshav@uwaterloo.ca)

Format: Academic paper

Publisher: ACM SIGCOMM Computer Communication Review

Year: 2007

Pages: 83–84

URL and last accessed: http://dl.acm.org.libezproxy.open.ac.uk/citation.cfm?doid=1273445.1273458 Last accessed: 19 November 2013.

Referenced by: OpenLearn free course materials

Abstract: 'Researchers spend a great deal of time reading research papers. However, this skill is rarely taught, leading to much wasted effort. This article outlines a practical and efficient *three-pass method* for reading research papers. I also describe how to use this method to do a literature survey.'

Notes: An excellent resource that provides an efficient workflow for reading an academic article.

As you go on collecting information resources, you'll find that it is worth being critical about what you retain in your BDMS – not all information resources are worth keeping. It is also good to be systematic. It is good practice to record a resource directly after having found it; then record your thoughts after the first pass and update your notes after subsequent passes.

Your BDMS needs to both discriminate and retain a level of understanding of the resource. With this in mind, we have positioned the Recording stage after the Assimilation stage (although, as noted, you may make an initial note of the discovery at an earlier point).

8.1.8 Relating

As you will realise from your professional practice, there are many contributors to any area of research or practice. The relationship between contributors is complex (sometimes circular), extends in time and can be affected by politics. At this point, however, we will look only at indicators of dependence as revealed through a bibliography.

The bibliography of a paper indicates the parts of the context of the paper that have been influential in its writing. Keshav's bibliography consists of four websites. Such a source citation is appropriate for his paper, but actually is not typical of an academic paper. Even so, we can plot the relationship between Keshav's paper and the material it cites as in Figure 5.

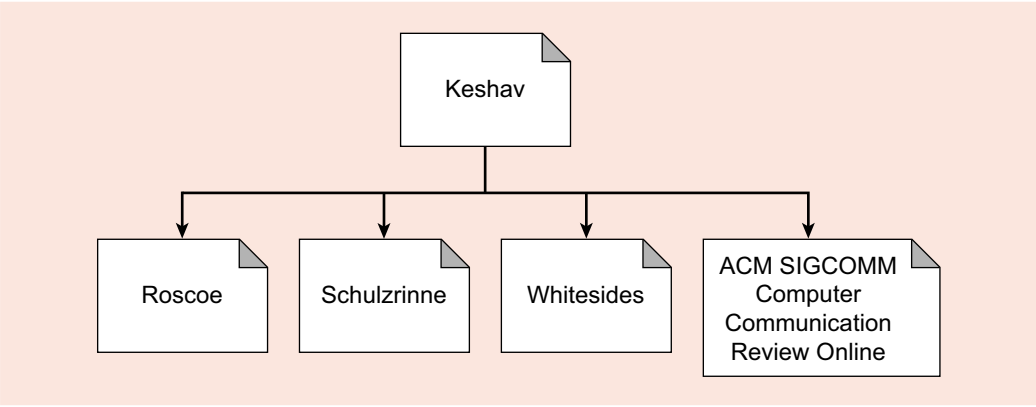


Figure 5 The relationship between Keshav's paper and the material it references

Activity 22

Add each element in Keshav's bibliography to your BDMS.

Discussion

You may have noticed that the Roscoe website itself lists as a resource the Keshav paper that you have just analysed. That means that if we were to extend Figure 5 there would be a loop in it. You would not often find this phenomenon in typical archival research articles because a paper is normally printed and once printed is unchangeable. Online resources such as Roscoe's can of course be updated, so can incorporate references to other resources that were created after the original item.

We added five entries to our BDMS, corresponding to the five references. For each we set the 'Referenced by' field to 'Keshav2007How' as suggested by the diagram we drew.

Now complete Activity 23, which returns you to the discovery stage.

Activity 23

Consider a few topics that interest you in the area of software development. Focus on one of them, and using a search method of your choice, find three research articles that are relevant to your interests. Run through Keshav's workflow (first pass only) for each item found and if you think the item is sufficiently interesting, create an entry in your BDMS. Record your notes on each.

Do you think that you'd conduct a second (or third) pass on any paper that you found?

Conclusion

In this course we have looked at some key concepts, themes and skills related to software development, as an adapted extract from the Open University course M813 *Software development*.

Should you decide to study further, this course will allow you to develop the fundamental knowledge, understanding, and analysis and synthesis skills that you need to develop fit-for-purpose software in an organisational context, by taking a practice-based approach based on an organisation you are familiar with. The course will also give you an opportunity to investigate emerging trends in software development and carry out some independent research into issues in software development that interest you.

References

Bass, L., Clements, P. and Kazman, R. (2003) *Software Architecture in Practice*, Addison-Wesley.

Beck, K. (2000) *Extreme Programming Explained*, Addison-Wesley.

Benington, H. D. (1983) 'Production of large computer programs', *Annals of the History of Computing*, vol. 5, no. 4, pp. 350–61.

Bird, P. (1994) *LEO: The First Business Computer*, Workingham, UK, Hasler Publishing.

Boehm, B. W. (1988) 'A spiral model of software development and enhancement', *Computer*, vol. 21, no. 5, pp. 61–72.

Booch, G. (1994) *Object-Oriented Analysis and Design*, Addison-Wesley.

Cockburn, A. and Highsmith, J. (2001) 'Agile software development: the people factor', *Computer*, vol. 34, no. 11, pp. 131–133.

Cook, S. and Daniels, J. (1994) *Designing Object Systems: Object-Oriented Modeling with Syntropy*, Upper Saddle River, NJ, Prentice-Hall.

Forsberg, K. and Mooz, H. (1992) 'The relationship of system engineering to the project cycle', *Engineering Management Journal*, vol. 4, no. 3, pp. 36–43.

Fowler, M. and Highsmith, J. (2001) 'The Agile manifesto', *Software Development*, vol. 9, no. 8, pp. 28–35.

Grant, R. M. (2007) *Cases to Accompany Contemporary Strategy Analysis*, 6th edn, Wiley.

Gunter, C. A., Gunter, E. L., Jackson, M. and Zave, P. (2000) 'A reference model for requirements and specifications', *Software, IEEE*, vol. 17, no. 3, pp. 37–43.

Hall, J. G. and Rapanotti, L. (2003) 'A reference model for requirements engineering', in *Proceedings of the 11th International Requirements Engineering Conference, 2003*. Montery Bay, CA, September 8–12 2003. New York, IEEE, pp. 181–7.

Hall, J. G. and Rapanotti, L. (2009) 'Assurance-driven design in problem oriented engineering', *International Journal on Advances in Systems and Measurements*, vol. 2, no. 1, pp. 119–30.

Jackson, M. (1995) 'The world and the machine', in *ICSE 1995: 17th International Conference on Software Engineering*. Seattle, WA, April 23–30 1995. New York, IEEE, pp. 283–92.

Jacobson, I., Booch, G. and Rumbaugh, J. (1999) *The Unified Software Development Process*, Addison-Wesley.

Jacobson, I., Christerson, M., Jonsson, P. and Overgaard, G. (1992) *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley.

Keshav, S. (2007) 'How to read a paper', *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 3 [Online]. Available at http://dl.acm.org.libezproxy.open.ac.uk/citation.cfm?doid=1273445.1273458 (Accessed 19 November 2013).

Liskov, B. and Zilles, S. (1974) 'Programming with abstract data types', *ACM Sigplan Notices* , vol. 9, no. 4, pp. 50–9. ACM.

McManus, J. and Wood-Harper, T. (2008) *A Study in Project Failure: Technical Report*, BCS.

Robertson, S. and Robertson, J.C., (2006) *Mastering the Requirements Process*, 2nd edn, Addison-Wesley.

Rogers, G. F. C. (1983) *The Nature of Engineering: A Philosophy of Technology*, Palgrave Macmillan.

Royce, W. (1970) 'Managing the development of large software systems', *Proceedings of IEEE WESCON*, vol. 26, pp. 1–9.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. (1991) *Object-Oriented Modeling and Design*, Prentice Hall.

Schwaber, K. and Beedle, M. (2001) *Agile Software Development with Scrum*, Prentice Hall.

Selic, B. (2003) 'The pragmatics of model-driven development', *Software*, vol. 20, no. 5, pp. 19–25.

Turski, W. (1986) 'And no philosopher's stone either', *Information Processing '86: Proceedings of the IFIP Congress, Dublin, Sept. 1–5, 1986*, London, Elsevier, pp. 1077-–1080.

Online only

Enticknap, N. (1998) 'Computing's Golden Jubilee', *Resurrection*, no. 20 [online]. Available at http://www.cs.man.ac.uk/CCS/res/res20.htm (Accessed 8 July 2014).

Lake, M. (2010) 'Epic failures: 11 infamous software bugs', *ComputerWorld* [online]. Available at www.computerworld.com/s/article/9183580/Epic_failures_11_infamous_software_bugs (Accessed 21 May 2013).

Standish Group (2009) *CHAOS Summary 2009* [online]. Available at http://blog.standishgroup.com (Accessed 25 March 2013).

Tootill, G. (1998) 'The original original program', *Resurrection*, no. 20 [online]. Available at http://www.cs.man.ac.uk/CCS/res/res20.htm (Accessed 8 July 2014).

Acknowledgements

This course was written by Lucia Rapanotti and Jon G. Hall.

Course image: Texture X in Flickr made available under Creative Commons Attribution 2.0 Licence.

The material acknowledged below is Proprietary and used under licence (not subject to Creative Commons Licence). Grateful acknowledgement is made to the following sources for permission to reproduce material in this course:

Figure 2: Keshav, S. (2007) 'How to read a paper', ACM SIGCOMM Computer Communication Review, vol. 37, no. 3 [Online], ACM New York, ACM DL Digital Library.

Figure 3: Keshav, S. (2007) 'How to read a paper', ACM SIGCOMM Computer Communication Review, vol. 37, no. 3 [Online], ACM New York, ACM DL Digital Library.

Every effort has been made to contact copyright owners. If any have been inadvertently overlooked, the publishers will be pleased to make the necessary arrangements at the first opportunity.

Don't miss out:

If reading this text has inspired you to learn more, you may be interested in joining the millions of people who discover our free learning resources and qualifications by visiting The Open University - www.open.edu/openlearn/free-courses