

The Architecture and Design of 48433 Software
Architecture
Autumn 2004 Version

John Reekie
Faculty of Engineering
University of Technology, Sydney

15th December 2003

Contents

1 Preliminaries	4
1.1 Background	4
1.2 Context	4
1.3 Stakeholders	5
1.4 Vision	7
1.5 Constraints	8
2 Architecture	10
2.1 Qualities	10
2.2 Knowledge model	11
2.2.1 Concept map	11
2.2.2 Capabilities	12
2.2.3 Subject story	12
2.2.4 Cross-cutting concerns	15
2.3 Learning and delivery model	16
2.4 Physical model and mappings	16
3 Design	18
3.1 Subject content	18
3.1.1 Threads	18
3.1.2 Exercises	18
3.1.3 Assignments	18
3.2 Module structure	21
3.2.1 Lecture	21
3.2.2 Exploration pack	21
3.3 Assessment	22
3.3.1 Examinations	23
3.3.2 Assignments	23
3.3.3 Assignment conduct	24
3.4 Artifacts	25
3.4.1 Completion levels	25
3.4.2 Confidence levels	25
3.4.3 Delivery plan	27
3.4.4 Work estimates	27
4 Patterns and Anti-patterns	30
4.1 Evolutionary Curriculum Delivery	30
4.2 Managed Incremental Development	31
4.3 Metric Spaces	33
4.4 Pileup on the Marking Freeway	34
4.5 Chronically Late	35
4.6 Along for the Ride	36
5 Concluding remarks	39

List of Figures

1	Timetable view of the Software Engineering program	5
2	Dependency view of the Software Engineering program	6
3	Stakeholders of 48433 Software Architecture	7
4	The 48433 subject placed in context	8
5	Models of a curriculum	11
6	Concept map of 48433 Software Architecture	13
7	Threads drawn through the concept map	19
8	Assessment breakdown of 48433 Software Architecture	24
9	The Evolutionary curriculum delivery life-cycle	31

1 Preliminaries

This report describes the architecture and design of the undergraduate course 48433 Software Architecture, presently being developed and taught in the Faculty of Engineering of the University of Technology at Sydney. I use without apology techniques and terminology from the fields of software architecture and engineering throughout this report.

This report is the Autumn 2004 version of the architecture document for this subject. A new version will be produced at the conclusion of each semester.

1.1 Background

The subject 48433 Software Architecture, hereinafter referred to simply as “SA,” was first taught in the current semester (as of the writing of this report), Spring 2003. Preparation for the subject was woefully inadequate, and both students and teaching staff suffered as a result. This is attributable partly to confusion over whether, when, and how the subject would run. More importantly, however, it is attributable to the fact this is simply a difficult subject to teach at the undergraduate level—the field is still developing rapidly with a multitude of views about what software architecture is, there are no textbooks, and creating labs or other practice-based exercises is difficult.

The goal of this document and subsequent subject development work is to counter the above difficulties and produce a subject that will stand as a solid contribution to the undergraduate software engineering curriculum, both at UTS and potentially at other institutions.

1.2 Context

SA is a mid-level subject that, in the standard timetable, occurs in the fifth coursework semester of an academic program that includes eight coursework semesters. See Figure 1. (There are also two “experience” semesters, shown in grey, in which students intern in an engineering position.) In terms of student development, this subject occurs at a stage in which students will be solidifying core skills and establishing a foundation for the more advanced subjects that follow in coursework semesters 6 through 8.

Figure 2 illustrates the place of SA in the program in terms of subject prerequisite dependencies (an arrow from subject B to subject A indicates that A is a prerequisite of B). *Note:* This diagram is a modified version of the current dependency structure and is presently under consideration for adoption. For the purposes of this document, I’m going to assume that this new dependency structure is adopted.

SA thus occurs in the middle section of a series of subjects beginning with Object-Oriented Programming and Object-Oriented Design, followed by Software Engineering and Software Architecture, and culminating in the heavily practice-based subjects Software Systems Analysis and Software Systems Design. This core thread of subjects forms a strong backbone that develops in

Physical Modeling	EfS	Eng Comm'n	Uncert & Risks	Econ & Finance	Eng Mngmnt	Tech Assess.	Capstone
Math Mod 1	Math Mod 2	Software Eng	Formal SE	Software Arch	Database Fund	Emerging SW Tech	Capstone
Intro to Elec Eng	Electronic Ccts	Intro Dig. Sys.	RTOS	Embedded SW Sys	HCI	SSA	SSD
OOP	OOD	Embedded C	Comms Networks	Elective	Elective	Elective	Elective

Figure 1: Timetable view of the Software Engineering program

students the fundamentals of the practice of software engineering.

1.3 Stakeholders

Figure 3 illustrates the major stakeholders in 48433 Software Architecture and their major concerns. In more detail:

Student Students are primarily concerned with assessment. Many are focussed merely on passing, some aim higher, at a Distinction or High Distinction mark. To achieve this aim, students want to know what the course content is, how the subject is assessed, and what the workload required to achieve their goal will be.

As secondary considerations, students aim to be educated about the subject matter of the course, and even to enjoy learning and achieve a sense of achievement or satisfaction from doing so.

Teaching staff Teaching staff are primarily concerned with subject development and delivery. In this subject, they are also concerned with achieving their own goal of a sense of satisfaction from effective teaching, supported by a suitable workload. There is no significant distinction in this subject between subject coordinators, developers, and teachers. While these are necessary roles, seperating them into different people reduces involvement and committment and therefore the chances of successfully executing the subject.

Program overseer The program overseer is primarily concerned that the subject meshes with the rest of the Software Engineering program, and that the various procedures for ensuring this are followed. In our context, that would include both the Program Head and the Associate Dean of Teaching and Learning.

University The University is primarily concerned that the course does not break any rules, does not give students reason to sue the University, and does not consume any more than the appropriate amount of resources.

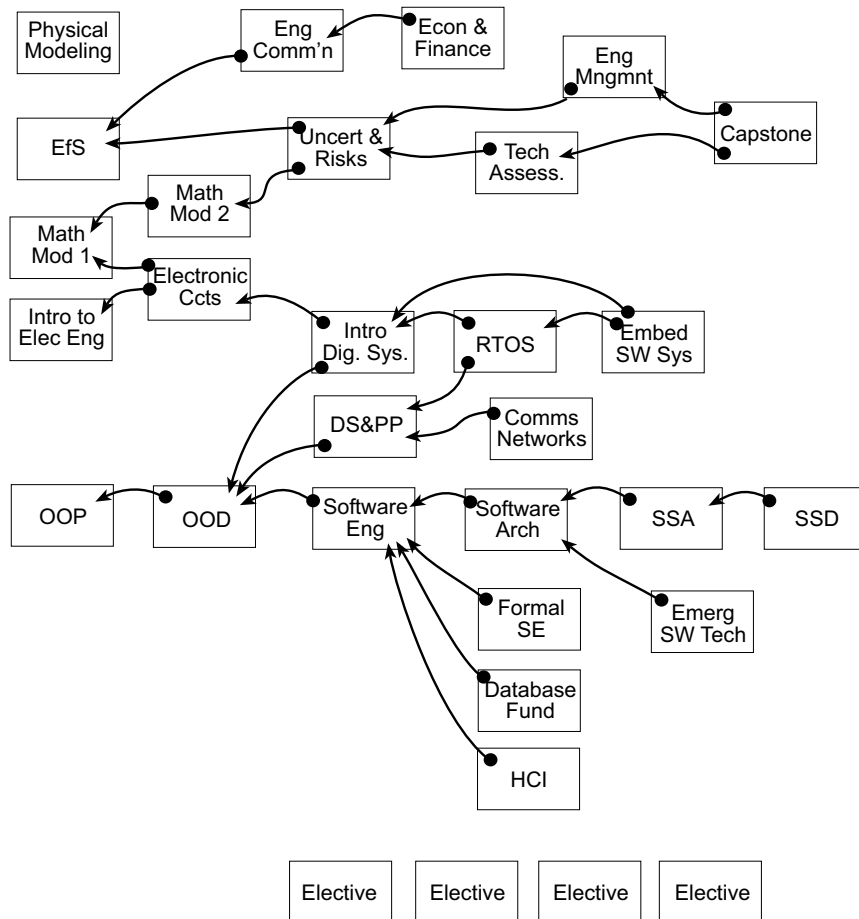


Figure 2: Dependency view of the Software Engineering program

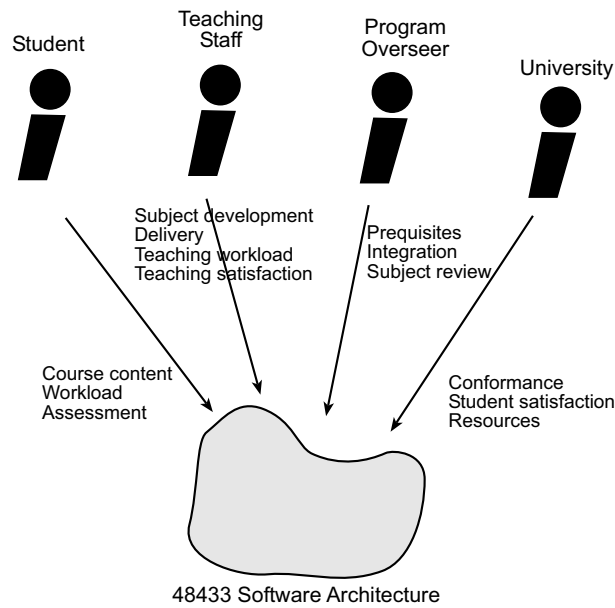


Figure 3: Stakeholders of 48433 Software Architecture

1.4 Vision

As a starting point for the vision of this course, I turn to the vision statements of two of the majors in the Faculty of Engineering, which were recently produced by academic staff:

Software Engineering

The Software Engineering Program will be regarded as the software engineering program of choice. Graduates of the program will have mastered the managerial and technical aspects of software engineering and be considered leaders of their profession, demonstrating ethical and professional conduct in the workplace and wider community.

Telecommunications

The Telecommunications Engineering program views an engineering education as the beginning of a life-long practice that demonstrates ethical and professional conduct in both the workplace and the community. As such, graduates of the program possess a firm telecommunications-specific technical foundation, a grounding in analytical thinking and the ability to abstract and model complexity, and a good understanding of their on-going role as professionals prac-

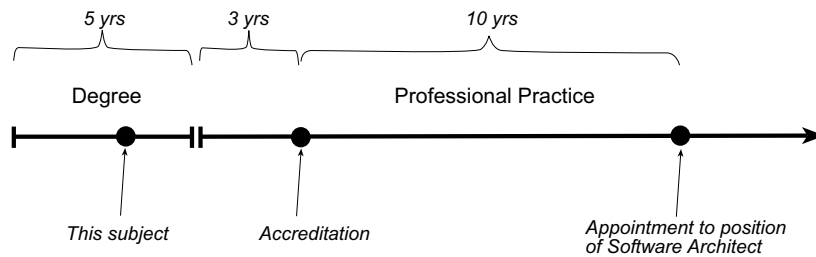


Figure 4: The 48433 subject placed in context

ting within the broader context of a complex and diversely-skilled society. To this end, the Telecommunications program provides a relevant and stimulating learning program that recognizes and accomodates the diverse backgrounds of its students.

The Telecommunications vision could just as well apply to the Software Engineering program, and since it's a little fuller I'll use it as well to guide the architecture in subsequent sections. This vision helps place the subject properly into the overall context of a software engineering education *and* career. If we depict the "lifecycle" of a software engineer or architect as in Figure 4, it becomes clear that this subject occurs quite early, and it should therefore be designed accordingly.

Based on the above vision and context, a useful vision of the subject itself is as follows:

The subject 48433 Software Architecture fulfils a critical role in the development of a student software engineer. The student has acquired a basic understanding of software construction and engineering, and now needs to broaden their scope to understand the system-level quality issues that are dealt with at the architectural level. The practice-based nature of our curriculum makes this subject a down-to-earth but solid introduction to the subject, what an emphasis on practicality over theory. Students who complete the course successfully will feel a sense of achievement with their new-found knowledge and abilities.

1.5 Constraints

Based on our experience with running the subject for one semester, a number of non-obvious constraints have now become apparent. Specifically:

No good textbook There is no suitable textbook for the subject of software architecture as an undergraduate course, especially at the middle stage in which our course occurs.

Confusion over the role of architecture There is not a consistent understanding of the role of software architecture.

Confusion over terminology and meaning There is not a consistent use and understanding of terminology and concepts.

Students do not understand concurrency At this level in their development, many students have at best a hazy grasp of the concept of concurrency, which impeded their ability to even produce a concurrency view of an architecture. Network security and databases are other areas in which a good understanding of the basic concepts cannot be assumed.

Poor quality or immaturity of architectural tools Many tools that could perhaps be used in a course of this nature are either of poor quality or immature. Examples of the former include tools for architecture description languages; examples of the latter include MDA (model-driven architecture) tools.

Other tools are out of reach financially.

2 Architecture

The process of curriculum design consists of constructing three models, and finding mappings between them. This is illustrated in figure 5. Each model should be represented by several views. (I used two views of the physical model of the curriculum in figures 1 and 2.) The knowledge model represents both the “functional” requirements, such as material learnt, and cross-cutting requirements such problem-solving skills. The physical model is the course structure, while the learning and delivery model informs and constrain the mapping of the knowledge model onto the physical model. In the diagram, the physical model is shown for a complete curriculum, but for a single subject we are concerned only with the bottom two layers—that is, the division of the semester and of each week.

The development of the subject is to follow the patterns *Evolutionary Curriculum Delivery* (section 4.1) and *Managed Incremental Development* (section 4.2). Although it would be desirable to follow *Metric Spaces* (section 4.3), I don’t think this is possible in this iteration.

2.1 Qualities

Qualities are the “non-functional” aspects of a system’s development or operation. For the purposes of architecting this subject, I adopt the following qualities as the most important (see [8] for a list containing other qualities):

Relevance The material taught in the subject is to be made relevant in the sense that students will be able to *use* what they have learnt within the context of the subject, of the Software Engineering program, and on any other software project that they undertake.

The high priority of this quality excludes exposing students to material “just so they have heard about it.”

Fairness The combination of content, delivery, and assessment will be *fair*. That is, expectations will be clear, required work with respect to student’s aims (marks, learning, or otherwise) will be described carefully, and assessment will be as stated in the Subject Guide. In addition, workload will be reasonable and the subject will not force students to steal time allocated for other subjects.

The high priority of this quality excludes students from being assessed on material that they have not had a reasonable opportunity to learn (the “deep end” approach to instruction).

Enjoyability The subject will be structured and delivered in such as way as to maximize the possibility of the learning and teaching experience being enjoyed (at both superficial and deep levels) by students and staff respectively.

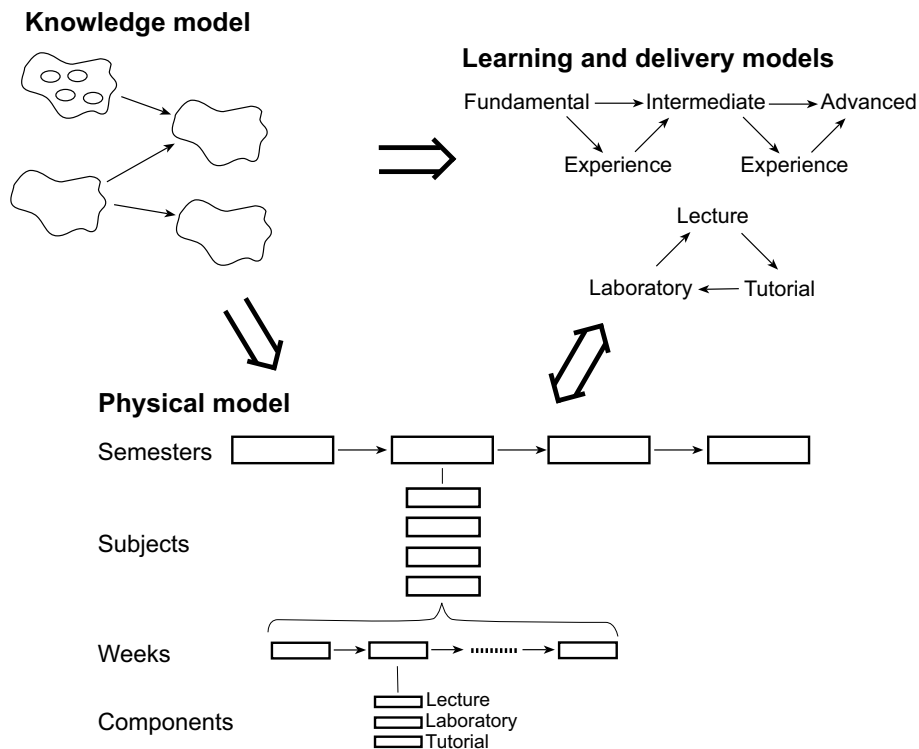


Figure 5: Models of a curriculum

At this stage in the subject's life, qualities such as Substitutability, Adaptability, and Modifiability are not given any priority. These qualities may become more important as the subject matures, and will thus be dealt with in future iterations of the subject.

2.2 Knowledge model

Particularly at the subject level, the knowledge model is quite critical. Several views may be constructed to illustrate the model.

2.2.1 Concept map

Figure 6 is the concept map for this subject. The strength of relationships between concepts is approximately indicated by (inverse) distance. To support *Relevance*, the concept map tends to take the "industrial" view of software architecture [1]. Thus, there are no ADLs (Architecture Description Languages); mockups, prototyping, metrics, and testing appear prominently; and the academic taxonomies of viewpoints, styles, and tactics are de-emphasized compared

to the course we ran in Spring 2003.

2.2.2 Capabilities

Capabilities, or learning outcomes, are formulated as problems or tasks that students become capable of solving or performing at some point in the course. One can view the concept map as the “input” (what is taught), whereas the capabilities are the “output” (what is learnt). For the purposes of this iteration of this document, I will simply use the detailed assignment statements (outline in section 3.1.3) as the capabilities view. Additional views of “must-know” and “like-to-have” capabilities will be added in future iterations.

2.2.3 Subject story

The story is a prose passage that captures the content of the subject, as expressed two-dimensionally in the Concept Map, in a linear narrative form. The story does not cover the same topics as the Concept Map, as its purpose is different. By writing the story, I hope to make the sequence of presentation more coherent and to provide better linking and reinforcement of key concepts throughout the subject.. The story also sets the underlying attitude and tone that will permeate the construction and delivery of the more formal materials.

Software architecture is the study and design of the most pervasive aspects of a software system. As such, it takes on the task of the high-level decomposition of the system into components, the interaction between them, and how, when combined, they meet the needs of the system’s many stakeholders.

Suppose you have been tasked with responsibility for design of a large and fairly complex software system. Confident in your ability to obtain satisfactory requirements when you need them, you embark on the task of designing the *architecture* of the system. You have identified the context in which the system will operate and its interfaces to external systems, and you have tracked down and interviewed the stakeholders in order to make sure that all their needs are met, as best as can be done. You have realized that there is a great deal more to system design than you thought: apart from meeting the so-called functional requirements, you have been given conflicting goals for performance, availability, and security! Not only that, but the marketing department demands that they be able to sell the system you deliver to this customer to a bunch of other customers, *and* it needs to meet their perceived market window *and* they also want to know what competitive edge you are going to give them against the competition!

Impossible? Well, yes, probably... but on the bright side, you’re off to as good a start as you are ever going to get. You’ve got your

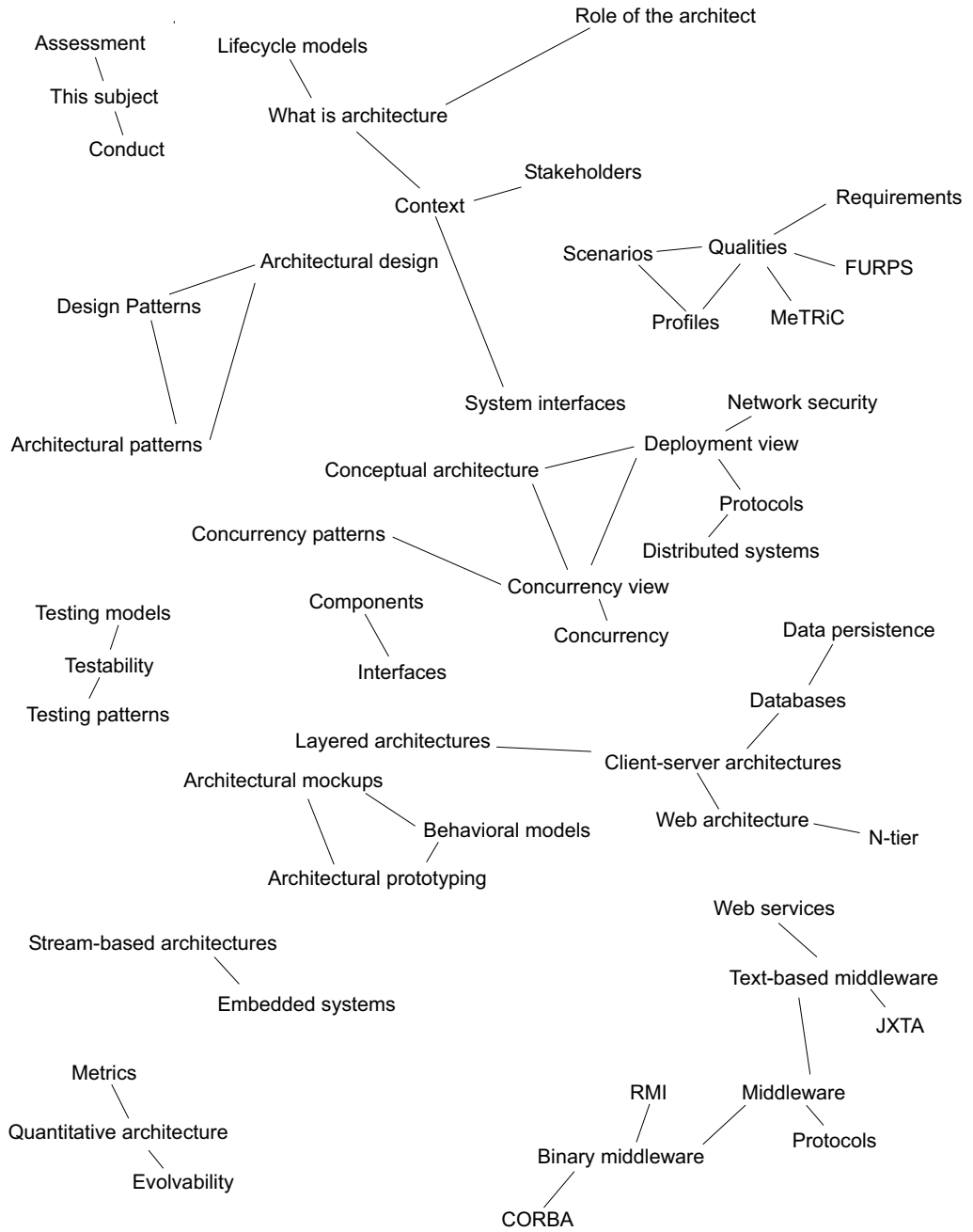


Figure 6: Concept map of 48433 Software Architecture

architectural work between your requirements analysis and your design, you've taking a *cooperative* rather than an *adversarial* approach with your customer and the other stakeholders, and... you've just now realized that there is a great deal more to architecture than the first paragraph of this story appears to be telling you!

Having located requirements and written concrete scenarios for the NFRs, you decompose the system into one or more module views. To scope the architectural work, you have already defined the system scope and its external interfaces. As you proceed through the iterative process of refining the architecture and creating additional views, together with refining and clarifying requirements, you start to identify candidate components and their interfaces, and employ layered and client-server architectural styles where appropriate. And, since even the most reliable software system is useless if it gets broken into, you pay particular attention to the physical deployment onto machines and networks and the interaction of the deployment view with the system software.

Even within a single machine, effective resource utilization demands significant degrees of concurrency, and so concurrency views are used to understand and manage this aspect of the architecture. Related concepts such as threading and scheduling algorithms and priorities also come into play, and many of these can be effectively demonstrated with a simple multi-threaded Java server.

A new model of client-server system has arisen just in the last ten years or so, and is changing many things about the way that we build systems and even think about software engineering. I refer of course to the Web. Apart from basic two-tier web architectures, more complex and critical web applications use a variant of three-tier architecture. Much of the infrastructure for building complex web applications as available "off-the-shelf" these days, including web servers, application servers, and databases to provide fast, reliable, persistent, and secure (hopefully) data storage. The ubiquity of the Web has given rise to XML and extensions to HTTP to support more sophisticated means of providing data in an open client-server environment, commonly known as web services.

The more traditional high-level way to access services on remote machines has been to use binary middleware protocols and services such as RMI (Java) and CORBA (language-independent). These forms of middleware will continue to find application in distributed and embedded systems where system structure is more static and performance requirements are more stringent. Embedded systems in particular are characterized these days by large numbers of processors attached to specific hardware sub-systems, communicating through various standard or proprietary network protocols. The dominant architectural form of many of these can be characterized

as stream-based—that is, data flows between components in continuous sequences. Quantitative characterization and measurement, although it is increasingly important as software systems become more complex, is doubly so in embedded systems because of their hard realtime throughput and performance requirements.

Finally, the architectural level is where you consider and plan for testing of the system. After all, the most elegant architecture in the world will likely fail if you are unable to test it. Patterns that specifically address the testability of an architecture can save enormous verification headaches in complex systems once development iterations have begun. Considerable expertise is captured in published design and architectural patterns that constitute a gold-mine (not all of it real gold, unfortunately!) of information for the student of software architecture.

2.2.4 Cross-cutting concerns

The following “cross-cut” the main thrust of the subject as expressed in the Concept Map and the Subject Story. Although the Story is slightly successful in conveying an attitude and approach to the field, I feel that being more explicit here will enable construction of materials that, over the course of a semester, *builds* key elements of the overall approach rather than merely stating them. We don’t for example, say somewhere in one lecture that “architecture is not an excuse for overly complex designs”; rather, the concepts of simplicity, fitness for purpose, and verification against real needs and expected results, are developed in a natural progression to eventually make the statement “architecture is not an excuse for overly complex designs” a self-evident conclusion.

- *Patterns.* A formal description of patterns is not given until Module 12. However, the approach of patterns is to be used throughout the subject. For example, when presenting an architectural style, the context and forces should be presented, and the style described as a commonly-practiced solution in that space. The benefits *and* liabilities of that solution should be presented. (This latter is in contrast to the “tactics” approach to realizing architectural qualities, which does not bother to present a balanced viewpoint of the trade-offs inherent in *any* solution.)
- *Documentation.* The subject does not foster in students the notion that documentation must conform to some standard to be effective. Rather, an approach to documentation is fostered that encourages clarity, brevity, and relevance. (If you like, call it an Agile approach to documentation.) This approach to documentation will be encouraged by providing suitable templates for the assignments, and by implementing example systems that demonstrate it. Students that wish to do, however, have the opportunity to submit their final assignment as an IEEE 1471-compliant document.

- *Fitness for purpose.* As indicated in the introduction to this section, architecture should be about fitness for purpose. The subject material should reinforce the notion that a “good” architecture is one that is fit for its purpose, not one that implements some particular style or has a high degree of complexity or cost.

2.3 Learning and delivery model

The following learning and delivery models are appropriate for this subject.¹

Foundation and Extension

Instill first familiarity with foundation concepts, and then branch into deeper and narrower topics that depend on these foundation concepts.

Learning as Exploration

When introducing a new area of study, provide first a “map” to the area, and then provide means for students to explore based on the map.

Questions Drive Answers

Undirected learning lacks context and motivation. So, direct learning by providing questions, to which students seek answers.

2.4 Physical model and mappings

The semester consists of thirteen teaching weeks, followed by formal examinations. Contact time each week is limited to a single three-hour session, divided roughly into a one-hour lecture and a two-hour tutorial and administrative session.

Use of a single contact session provides a certain amount of support for student’s tight schedules, in that we only require attendance at (and thus also travel to) one three-hour session per week (two sessions are typically scheduled), and aim to supplement this contact time with extensive support from online activities. Students are expected to participate in these online activities, as are teaching staff.

The course is divided into a set of *concept modules*, each of which is distributed over one or more teaching weeks. Concept modules can overlap each other (that is, students will be completing exploration of one module while being introduced to another; modules can be revisited in later modules; and so on).

¹If you’re wondering where these came from, I made them up just now, based on my own experience and intuition. I think they are neither particularly insightful nor controversial, but nonetheless, they will be refined and supported by references to published works by educational experts in future iterations of the subject...

Based on the *Foundation and Extension* learning model, the subject is roughly divided (chronologically) into two. It begins with a series of foundational concept modules which all students are expected to grasp (and be examined on). The second half consists of concept modules that support deeper and/or more specialized exploration, of which students can choose a subset on which to focus.

The *Learning as Exploration* model divides each session into a lecture session, in which the overview and “map” is provided, and a tutorial session and followup online discussion, in which students are guided through the initial part of their exploration of a particular concept module. This exploration is structured according to *Questions Drive Answers*.

Also in accordance with *Questions Drive Answers*, the second half of the course is motivated by an assessable project that students will use to guide their choice of subset of topics to explore, and to motivate this exploration.

3 Design

This section describes in more detail the lower-level organization of the subject. The subject is divided into twelve concept modules. Each is a week, except the first two, which are half a week and a week and a half respectively.

3.1 Subject content

3.1.1 Threads

The content of each module is expressed as a *thread* through the concept map, and shown in figure 7. The division into threads, the subject story, and the concept map were iteratively refined to get them all to “fit” with each other.

During the development of materials for each module, a story may be written for that module if necessary.

3.1.2 Exercises

Some modules will have one or a small number of practical exercises. These exercises have two aims: providing additional opportunity for understanding, and providing useful experience to feed into the project assignment. Each exercise will be small and simple, and there will be no significant additional research needed to complete the exercise.

Candidate exercises are:

- Open an HTTP connection using telnet.
- Compile and execute a simple socket-based server.
- Compile and run a simple multi-threaded server and client.
- Install a database server and “talk” to it with simple SQL queries.
- Install a web server and write a simple server-side program.
- Install and exercise an XML or O/R persistence layer.
- Install a scripting language and exercise parts of a prototype architecture.

3.1.3 Assignments

All assignments performed by each team are based on a single software system, chosen by the team from a selection of lightweight specifications [7]. Ambitious teams can propose their own. Possible systems include:

1. Digital radio station. Includes all aspects of real-time audio generation, storage, and replay, as well as console and equipment control.
2. Digital video and audio jukebox. Audio- and video-on-demand for hotels and resorts.

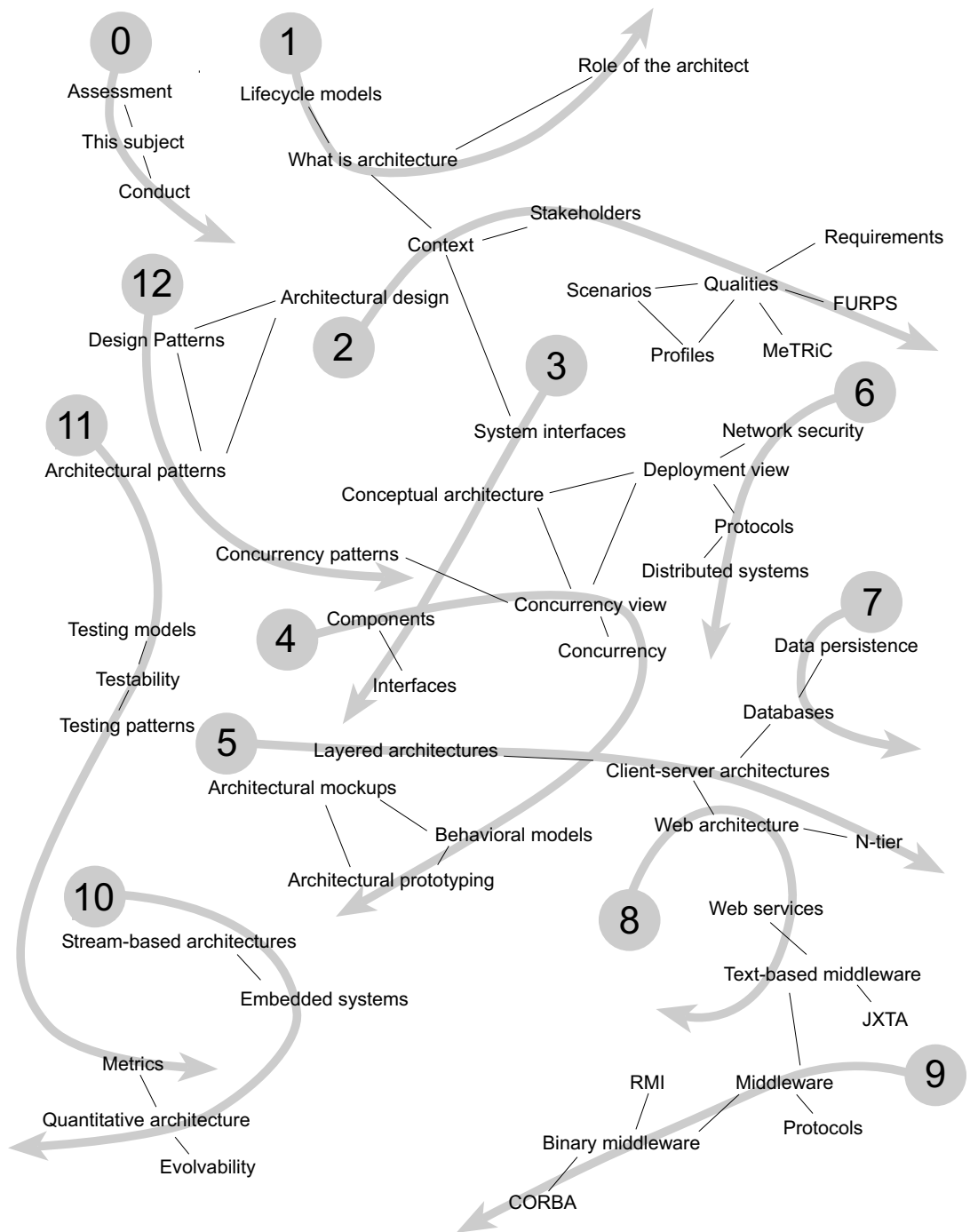


Figure 7: Threads drawn through the concept map

3. Software system architectural visualizer. A large software company wants to visualize all of their software products to identify commonalities and opportunities for performance and architectural improvements across their product lines.
4. University teaching and collaboration system. You want to develop and sell a *good* teaching and collaboration system to Australian Universities. The marketing department has written a spec...
5. Web market and content analyzer. Your client wants to sell a service to companies with large websites that analyzes the content on both their site and all their competitors, identifying competitive risks.

Based on the progression of knowledge in the Story and Concept Threads, the three foundational assignments will be as follows. To support *Fairness*, the detailed design of these assignments must be such that they are based only on modules undertaken up to a week prior to the due date of the Inspection (section 3.3.3).

1. Choose or propose a software system, and identify the context, stakeholders, and key functional, non-functional, and business requirements. Construct scenarios to serve as architectural drivers. This assignment will be assessed individually.
2. Produce an initial architecture of the system. As a team, document the results of the system architecture. As individuals, produce and document the architecture of one of the major components of the system architecture.
3. Produce an architectural mockup and elicit behavioral models of the architecture. As a team, produce and “run” the mockup of the system architecture and produce key behavioral models. As individuals, analyse the behaviour of a single component and its interaction with immediately-connected components.

Project assignments allow deeper exploration of some of the latter sections of the Story, still based on the chosen software system. The deliverable for the project assignment includes a working architectural prototype (if appropriate), and a report containing the description and reasoning behind the final version of the architecture—that is, it is expected to reuse and/or modify material submitted in earlier assignments.

Typical project assignments include:

- Construct an architectural prototype using CORBA
- Construct an architectural prototype using web services.
- Construct a prototype of an N-tier architecture using off-the-shelf web components.

- Analyse and measure an open-source product and compare to your architecture.

Bonus assignments can be anything that will challenge motivated students. Examples:

- Deliver your architectural model in an IEEE-1471 compliant document.
- Re-evaluate and re-document your architecture in the light of the new requirement for an entire product line.
- Analyse the architecture and design of 48433 Software Architecture and write the architecture for the next development iteration.
- Repackage your assignments to serve as a reference solution for future semesters.

3.2 Module structure

Each concept module consists of one lecture, and one “exploration pack.”

3.2.1 Lecture

The purpose of the lecture is to provide the “map” that introduces the concept to students and allows them to then explore further. It must be emphasized to students that the lectures slides are *not* the examinable materials.

To fulfil these goals, and to support the vision and other issues discussed earlier, lectures:

- Introduce the concept and indicate its breadth.
- Use specific examples to illustrate key points.
- Refer to real examples of software systems.
- Motivate further learning.
- Contain humour.

3.2.2 Exploration pack

The exploration pack provides a framework that students use to learn the concepts of that module. The exploration packs are to be designed to closely support the assignments, to reduce confusion and fragmentation [4] and to maximize *Fairness*. The exploration pack includes:

Questions

A series of questions, mostly multiple-choice or many-of-many, progressing from questions with straightforward answers apparent directly from lecture notes, through to questions that require more searching and finally to questions that are quite ambiguous. Students are not given answers to any questions. Rather, they are encouraged to find their own answers—this process is initiated in the tutorial session following the lecture and followed up in online discussion. Throughout, it is emphasized to students that the goal is not to get an answer, the goal is to understand *why* they would give that particular answer.

Exercises

A small set of exercises that students should accomplish. The exercises are designed to give students an opportunity to work through the mechanics of various notations and expression of concepts. The exercises are not assessable, but again it is emphasized to the students that if they do not or cannot do the exercises, then they will not be able to satisfactorily complete the assignments.

Readings

A listing of resources that can be used to support this concept module. These we divide into the following sections (with commentary written as instructions to students):

Supporting readings This material supports the lecture notes and tutorial discussion. Studying of this material is required for successful completion of the course, as is discussing it in the online forum.

Foundational support If you have not understood the material in the lecture, follow-up tutorial discussion, and on-line discussion, then studying the material in this section will help you to recover. You should also make sure that you seek help from a tutor during the allocated times (class and LDC). Note that failure to attend classes or to participate in on-line discussion will result in you getting a low priority for access to additional learning assistance.

Additional reading All of the information needed to achieve a Pass or Credit mark in the subject is provided in the lecture notes and Supporting Readings. Students who expect to receive a Distinction or High Distinction mark should study the material in this section in order to gain deeper and broader knowledge.

3.3 Assessment

Assessment is one of the most difficult parts of subject design for me. To enable me to think through the various issues and weigh possibilities, I formulated a number of assessment difficulties and strategies as *anti-patterns* [2]—*Pileup on*

the *Marking Freeway* (section 4.4), *Chronically Late* (section 4.5), and *Along for the Ride* (section 4.6). Based on those, here is my proposed assessment strategy for 48433 Software Architecture. The overall breakdown is shown in figure 8.

3.3.1 Examinations

I somewhat arbitrarily choose 50% of the total mark to be allocated to exams. Examinations do have their drawbacks, and I personally think that in an ideal world a software architecture subject should not have any exams at all. Nonetheless, the need for fair assessment in the light of the *Along for the Ride* anti-pattern leads me to decide that this approach is necessary for this subject.

Exam marks are further divided into 10% for a mid-semester exam, and 40% for a final exam. The mid-semester exam has two purposes: it serves as a “wake-up call” for students who are not keeping up with material; and it gives students a model for the type of questions that will be contained in the final exam. The final exam is a formal three-hour exam, and a pass in this exam is required to pass the subject. While this works against resolving the *Marking Pileup* anti-pattern, it does have the benefits that it does not eat up time from the teaching schedule, and students remain motivated to study up to the last teaching week.

To support *Foundation and Extension*, the final exam consists of three parts:

- A foundation part—50% of the examination mark.
- An exploration part, in which students choose three of five topics to answer—30% of the examination mark.
- An advanced part, which requires deep understanding to effectively answer—20% of the examination mark.

3.3.2 Assignments

There are five assignments with a total mark of 50%, consisting of:

- Three foundational assignments, each worth 10%. Each assignment requires in-depth exploration of foundational concepts covered to that point. They all focus around the architecture of the same system, which is chosen by the students early in the semester.
- One “project” assignment, worth 20%. The project allows deeper exploration of a subset of the concepts covered in the complete course. The project will be based on the architecture developed in earlier assignments, and has a strong hands-on flavor. A formal 30-minute presentation is required for this assignment.
- One “bonus” optional assignment, worth 5%. This assignment has marks that are disproportionally low relative to the amount of work required, but which gifted students may wish to undertake to extend themselves. This assignment can only be done individually, not in teams.

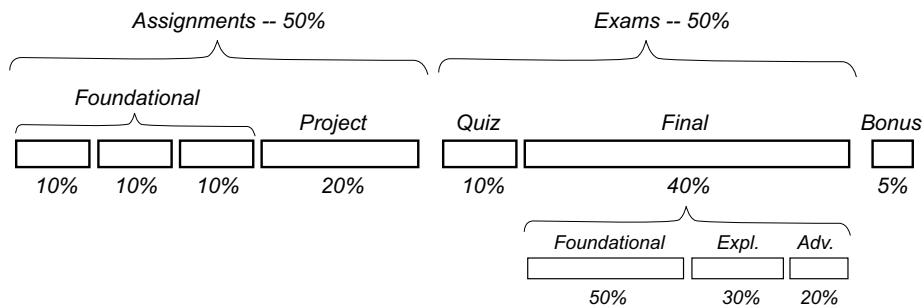


Figure 8: Assessment breakdown of 48433 Software Architecture

3.3.3 Assignment conduct

Assignment conduct is designed to alleviate the *Marking Pileup* and *Terminally Late* anti-patterns. It is time-boxed, and also incorporates elements of formal software inspections, so students are given better feedback and an opportunity to learn from each other. Specifically:

1. Each assignment is submitted for inspection and peer review, and resubmitted a week later with appropriate rework as a final version.
2. The inspection team works off a question sheet, while the submitting team notes issues raised during the inspection.
3. The inspection team assigns a mark to the submitted work based on a) its current quality and b) its expected quality if all issues are properly addressed. (The actual mark given is determined by teaching staff by examining the final submission).
4. The inspection team signs off on the inspection issue sheet.
5. The tutor signs off on the inspection issue sheet.
6. The final work is re-submitted a week later with all issues and their resolution added as an Appendix.

Tutors circulate during the inspection sessions to note participation by team members, ask questions, and note team members who appear to be non-participatory. This adds an element of oral assessment in accordance with *Along for the Ride*. Students suspected to be under-contributing will be asked to undertake a separate oral assessment in the next LDC session.

On-track and off-track

An assignment starts off *on-track*. If the assignment stays on-track, it will receive a mark between 50% and 100% of its allocated mark. If it fails to meet

any of the specified delivery criteria, it is moved *off-track*. Provided that it does not fail any further delivery criteria, it receives 50% of the mark allocated for the assignment; otherwise, it receives no marks.² The delivery criteria are:

- Meets the minimum specified requirements of the assignment.
- Passes inspection.
- Final is delivered on time.

No more than 10% of the total subject mark can be received from off-track assignments. In other words, if a foundational assignment is off-track and so is the final, then the maximum mark received will be 10%.

Additional penalties

- If the final submission is more than a week late, it receives no marks.
- Team members not present at an inspection will receive 75% of the mark received by the team for that assignment, subject to a successful supplementary oral assessment.

3.4 Artifacts

3.4.1 Completion levels

Table 1 lists the artifacts to be produced as part of this subject and the completion levels of each.

3.4.2 Confidence levels

Unless otherwise noted, all artifacts progress through four stages of confidence: Red, Yellow, Green, and Blue. The levels and their meaning are taken from [9]. Briefly:

- Red means that we have no confidence in the quality of the artifact, and it should either not be used in teaching, or used only with appropriate warnings and caveats attached.
- Yellow means that we have some confidence, sufficient to use the material in class with experienced teaching staff.
- Green means that we have high confidence in the artifact, sufficient to use it in teaching with less expert staff.
- Blue means that have very high confidence in the artifact, sufficient to publish it as open courseware and recommend to other institutions that they use it in their teaching.

²That is, if the assignment is worth 10%, then it receives 5%. It is not assessed out of 10 and then the resulting mark divided in half. This is a response to the *Markup Pileup*: if students do not meet the (perfectly reasonable) delivery criteria, then teaching staff do not need to spend time determining a percentage mark, a simple go/no-go test is sufficient.

Artifact	Level	Material
Slides	I	Main presentation slides, in Powerpoint
	II	Annotations to capture commentary made in the lecture for external (non-contact) learning
Expl pack	I	Questions and exercises
	II	Add supporting and extension readings
Course notes	I	Photocopied reader
	II	Original course notes
	III	Original textbook
Subject guide	I	UTS version, modified for dates each semester, in MS Word
	II	Open courseware version, in LaTeX
Assignment guide	–	Detailed instructions for all assignments
Inspection forms	–	Forms and instructions for peer inspections
Examinations	–	Quiz and final exam, with reference solutions
Past exams	I	A single set of past examinations with reference solutions
	II	Two sets
	III	Three sets
System specs	I	Lightweight software system specification for five systems
	II	For twelve systems
Reference solution	I	Reference solutions to assignments for one system
	II	For two systems

Table 1: Artifacts of 48433 Software Architecture

3.4.3 Delivery plan

Table 2 outlines the planned delivery schedule of the course artifacts. Confidence in the stages decreases the further out we go, and this plan will of course be revised at the completion of each delivery cycle.

There are four delivery iterations allowed for here. An “iteration” consists of leadup development prior to the semester, delivery, development and confidence-raising during the semester, and followup work after the end of semester. As much as possible, the artifacts are upgraded prior to the start of semester, while the semester itself is used to gain experience with the material and refine the artifacts as necessary. The followup work is primarily intended to review, to formally raise confidence levels, and to plan for the following iteration.

The first iteration is a bit special. Because of the large amount of development needed for the initial offering (or what is effectively the initial offering), this iteration terminates at the beginning of the first semester. The second iteration thus has no leadup period; from then on, everything is “normal.”

The four planned iterations are:

- Crawl—Prior to Autumn 04
- Walk— Autumn 04
- Run— Spring 04
- Fly— Autumn 05

3.4.4 Work estimates

Work estimates for development of Crawl and Walk are given in tables 3 and 4 respectively. Work estimates for teaching of the subject in Autumn 04 semester (Walk) are shown in table 5.

Artifact	Level	Crawl	Walk	Run	Fly
Slides	I	+	++	+++	
	II	-		++	+++
Expl pack	I	+			
	II		+	++	
Course notes	I	+	++	+++	
	II				+
	III				
Subject guide	I	++	+++		
	II			++	+++
Assignment guide	-	+	++	+++	
Inspection forms	-	+	++	+++	
Examinations	-	-	++	++	++
Past exams	I	-	++		
	II			++	
	III				++
System specs	I	+	++		
	II			++	
Reference solution	I		+	++	
	II				++

Table 2: Anticipated delivery of artifacts of 48433 Software Architecture. In order of increasing confidence: - (Red), + (Yellow), ++ (Green), +++ (Blue)

<i>Artifact</i>	<i>Qty</i>	blank	<i>Hours</i>	
		<i>Level</i>	<i>JohnR</i>	<i>LianL</i>
Lecture slides	12	I+	12	2
Expl pack	12	I+	2	4
Course notes	1	I+	4	-
Subject guide	1	++	4	4
Assignment guide	1	+	2	2
Inspection forms	1	-	2	-
Examinations	2	-	-	-
Past exams	1	-	-	-
System specs	5	I+	2	1
Reference solution	0	-	-	-
<i>Total Hours</i>			185	83

Table 3: Work allocation for development of “Crawl”

<i>Artifact</i>	<i>Qty</i>	blank	<i>Hours</i>	
		<i>Level</i>	<i>JohnR</i>	<i>LianL</i>
Lecture slides	12	I++	2	–
Expl pack	12	I+	1	–
Course notes	1	I++	4	–
Subject guide	1	+++	2	2
Assignment guide	1	++	2	4
Inspection forms	1	++	2	–
Examinations	1	++	16	8
Past exams	1	I++	1	1
System specs	5	I++	0.5	0.5
Reference solution	1	+	16	16
<i>Total Hours</i>			81.5	33.5

Table 4: Work allocation for development of “Walk”

<i>Artifact</i>	<i>Qty</i>	blank	<i>Hours</i>	
		<i>Task</i>	<i>JohnR</i>	<i>LianL</i>
Contact	13	Class	3	3
		LDC	1	1
Assignment marking	1	1	4	–
		2	4	–
		3	4	–
		4	4	–
		5	2	–
Exam marking	1	1	4	4
		2	8	8
<i>Total Hours</i>			82	64

Table 5: Work allocation for teaching activities, Spring 04

4 Patterns and Anti-patterns

This section contains patterns [3] and anti-patterns [2] that I have used to guide the design of this subject. These are probably more strictly referred to as proto-patterns, as work is yet to be done to ensure that they each capture expertise in the particular area that each addresses. Nonetheless, I found it useful to write them this way, and hope that in time they will grow into proper patterns.

These patterns originally arose in the text itself, but have been moved into a separate section to make the main sections more readable.

4.1 Evolutionary Curriculum Delivery

The development of 48433 Software Architecture uses the Evolutionary Curriculum Delivery lifecycle, expressed here as a Design Pattern [3]. This pattern is typically applied in conjunction with others, such as Managed Incremental Delivery.

Name

Evolutionary Curriculum Delivery

Category

Subject Development Pattern

Context

A tertiary curriculum with distinct subjects and semesters.

Forces

[Incomplete]

Solution

Use an evolutionary approach to curriculum development. This is an adaptation of Evolutionary Delivery as practiced in software development and described by McConnell [7, Chapter 20] to the tertiary curriculum. Figure 9 illustrates this lifecycle.

An evolutionary delivery model seems ideally suited to tertiary curricula, where one “iteration” of development corresponds to a single semester of teaching and associated review and re-development work. The review and revision work can be at each of three levels: a single artifact; the design level, which impacts some number of artifacts; and the architectural level, which may impact all artifacts.

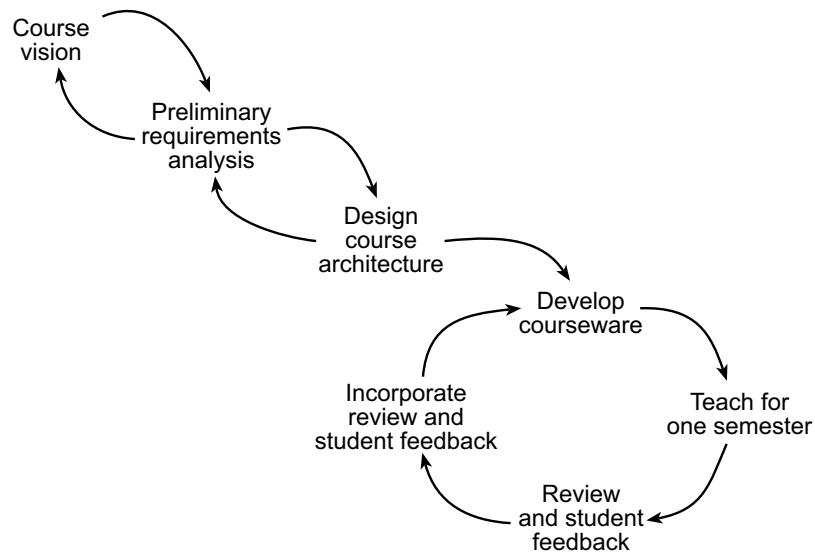


Figure 9: The Evolutionary curriculum delivery life-cycle

Known uses

As applied to a single subject, parts of evolutionary delivery are well established. Obviously, subjects are (commonly) taught once each semester (the “delivery”), although this is not usually thought of in terms of a lifecycle model. Review and revision of a subject on a semester basis is also common; at UTS, [some division of the University] conducts and collates a standard student survey for each subject at the end of each semester. In the Faculty of Engineering, Program Heads review each subject at the conclusion of each semester, although quality control and actual impact on subject quality appears to be highly variable.

Benefits and liabilities

The key benefit of an evolutionary delivery model is that all aspects of the subject can be refined on each iteration. Therefore, we don’t have to “get everything right” the first time around! This is nice if you are looking for a *manageable* and *incremental* approach to subject development and improvement—not something that fails to deliver either by trying too hard too soon, or by not trying at all (because it’s perceived to be too difficult to get it right).

I’m not aware of any liabilities of Evolutionary Curriculum Delivery.

4.2 Managed Incremental Development

48433 Software Architecture applies managed incremental development (MID) to its courseware. MID is expressed here as a pattern.

Name

Managed Incremental Development (MID)

Category

Subject Development Pattern

Context

A tertiary curriculum in which concrete materials are produced as part of the process of course development.

Forces

[Incomplete]

Solution

Define and manage the artifacts of the course (known as courseware) in a manner that allows them to be created and improved incrementally. More specifically:

- Define the complete set of artifacts.
- Define levels of completion of each artifact.
- Define levels of confidence that apply to each artifact, and a review process or metric requirement required to progress through levels of confidence.
- Use a document or content management system that allows the artifacts to be stored and tracked, and which makes visible each artifact's current level of completion or confidence.

Known uses

Typically this pattern, if applied at all, is applied only informally by individual academics. Storing courseware on an accessible web server is not uncommon, and motivated academics do improve their course material over the course of several semesters—the implementation of the course EECS20 [6, 5] at UCB is an example.

Benefits and liabilities

The key benefit of this pattern is increased visibility into the present state of the courseware. Related benefits that then accrue are the ability to implement evolutionary delivery (since the courseware doesn't have to be done all at once), higher quality (since the artifacts required are defined and there is a clear progression from lower to high quality), and more sensible workload (again, since

it doesn't have to be done all at once, and also because you now have a business case for resources needed to progress the artifacts along the path to higher quality).

The first key liability is the difficult of correctly specifying the artifacts and levels of completion, and in particular, the temptation to over-specify artifacts. Fortunately, if an evolutionary delivery approach is also applied (correctly), then an over-specified set of artifacts will be revised downwards an subsequent development iterations.

The second key liability is implementation of the document management system. A poor document management system is arguably worse than none at all.

4.3 Metric Spaces

Name

Metric Spaces

Category

Subject Development Pattern

Context

A tertiary curriculum in which concrete materials are produced as part of the process of course development.

Forces

[Incomplete]

Solution

Define metrics against which course and courseware quality can be assessed, and measure progress towards higher quality against them. Areas in which quality can be measured include:

- By student feedback
 - Quality of delivery
 - Quality of learning
 - Relevance of course
 - Enjoyment of taking the course
 - Value for money
- On courseware

- Levels of completion and confidence
- Adoption by external institutions
- Integration
 - With preceding subjects
 - With following subjects

Known uses

[Incomplete]

Benefits and liabilities

[Incomplete]

4.4 Pileup on the Marking Freeway

This anti-pattern is one to which I am particularly susceptible, so I need to put careful thought into this aspect of the course design.

Name

Pileup on the Marking Freeway *aka* Marking Pileup

Category

Assessment Anti-Pattern

Context

A tertiary subject that requires assessment of individual students on a percentage basis. One or more of the following apply to teaching staff:

- Like teaching, but dislike marking
- Busy with activities that have a “shorter fuse”
- Disorganized

AntiPattern Solution

Unmarked assignments pile up in teaching staff offices. Students are dissatisfied with the lack of—or lack of immediacy of—feedback. The result is more pressure on said staff to complete marking, which in turn makes the task more onerous.

Refactored Solution

Refactored solutions to the Marking Pile-up can be roughly divided into two categories: the “ounce of prevention,” or the “pound of cure.” For the purposes of providing effective refactoring, the former are best. (The latter encompasses things like “fixing” the teaching staff, which is unlikely to be very effective.) Thus, use one or more of the following tactics:

- Reduce the number or size of assessment tasks. This tactic can be applied only so far, in order to ensure that the provided mark is a reasonable sampling of whatever the measure of competence or completion is for the subject.
- Have students self-assess or use peer assessment. This tactic presents some issues with fairness.
- Use oral assessment at the time of delivery of an assessable task. This tactic gets assessment “over and done with” early. Care needs to be taken that adequate feedback is provided.
- Use go/no-go assessment. In its most basic form, this is a multiple-choice quiz. More sophisticated forms can be used effectively—certain tasks are either done satisfactorily, or they are not. This form of assessment requires that the “go” requirement is made very clear and properly supported by teaching materials.
- Use automated assessment. Again, multiple-choice is the simplest form. More sophisticated forms could include automated build and run tests (for software assignments).

4.5 Chronically Late

This last semester (Spring 2003), students were emailing staff lab assignments after all teaching activities had ceased, some of them several weeks past the due date. Clearly, this has to be improved.

Name

Chronically Late

Category

Assessment Anti-Pattern

Context

A tertiary subject that requires assessment of individual students on a percentage or go/no-go basis. Deliverable requirements with respect to timing and/or penalties are not made clear, or are ineffective. One or more of the following applies to teaching staff:

- More effective if they can mark all instances of the same assessment task at the same time.
- Hate waiting around for tasks to come out of the printer, or think they have better things to do.

AntiPattern Solution

Assessable deliverables are handed in or emailed late, sometimes by several weeks. Teaching staff are unable to mark all assignments at the same time, and/or are not able to provide feedback early enough (See also *Pile-up on the Marking Freeway*). Teaching staff get frustrated with students, making them less effective at their main task, *teaching*. The risk of losing assessment tasks is also substantially higher.

Refactored Solution

Provide positive encouragement and incentives to meet the specified delivery time (“carrot”), and penalize failure to meet the published deadline (“stick”). To do so, employ one or more of the following tactics:

- Explain clearly in course guides and handouts why on-time and bounded delivery is necessary and beneficial.
- Characterize the subject as *schedule-driven*, by analogy with the Timebox Development software practice [7, Chapter 39].
- Require presentations of draft deliverables *prior* to delivery of the final version. This is analogous to incremental software development practices in which a system is made to work, and kept working. Motivate this by making the draft a go/no-go acceptance test—that is, if the draft is not accepted, neither will the final.
- Penalize missed deadlines severely. For example, penalize by 50% for *any* lateness.
- Strictly bound delivery time. For example, give a mark of zero for any assessable task that is delivered more than a week after the deadline.

4.6 Along for the Ride

This past semester, at least one student performed little or no work in two of the major assignments. It is difficult to know how to be fair in a situation such as this. As always, prevention is better than “cure.”

Name

Along for the Ride

Category

Assessment Anti-pattern

Context

A tertiary subject that requires assessment of individual students, but which has material that is best learnt and practiced in teams.

AntiPattern Solution

Some students on a team do the lion's share of the work, while others do little. Despite that, and despite efforts to require students to declare amount of participation for each team member, all students receive the same mark.

Negative consequences:

- The non-working students learn nothing, as the real purpose of assessable tasks is to stimulate learning.
- Non-working students can pass a subject with no knowledge of it, leading to potential embarrassment for teaching staff and the Faculty.
- The harder-working students are effectively penalized for teaming up with the non-workers.

Refactored Solution

Simply not having team assessment is not a feasible solution in all subjects, because of the nature of the material being learnt. In addition, some team assessment is desirable because it promotes the UTS Engineering Graduate Attributes. Therefore, apply a number of the following tactics to refactor:

- Promote in students the idea that they should team with others expecting the same overall mark in the subject, not (necessarily) with their mates. This tactic implies that team formation should take place several weeks into the semester, and perhaps that teaching staff should guide team formation based on early individual assessment.
- Have deliverables consist of a clearly-identified team part and individual parts. For example, in application of architectural analysis, the top-level architecture is performed by the team, and modules are analysed by individual team members. Divide the assessment between the team and individual parts.
- Perform oral assessment of team assignments, in order to detect students that do not have a minimum level of understanding of the content of the assignment. If any are detected, give a pre-stated penalized mark, such as 50% or zero.

- Require a minimum mark for certain individual assessments, such as exams, to pass the subject. This ensures that students must at least demonstrate a minimum level of competence even if they do not really earn the marks awarded for team assignments.
- Allow teams of one.

5 Concluding remarks

This document describes the architecture and design of this subject, and serves for the first two iterations of the development cycle, known as Crawl and Walk. At the conclusion of Walk and the following iteration, Run, a new version of the document will be produced to reflect changes in the architecture and design based on the development and delivery (that is, teaching) of the subject up to that time.

While not everything can be accounted for in a document such as this, I believe that it serves as a sound basis for implementation of Crawl and Walk, and for further iterations of the subject.

References

- [1] Jan Bosch. *Design and Use of Software Architectures*. Addison-Wesley, 2000.
- [2] William J. Brown, Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1998.
- [3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [4] John Leaney. Fragmentation and learning efficacy within the School of Electrical Engineering. Personal communication, October 2003.
- [5] Edward A. Lee and Pravin Varaiya. EECS 20n: Structure and interpretation of signals and systems. Course website—online at <http://ptolemy.eecs.berkeley.edu/eecs20/>.
- [6] Edward A. Lee and Pravin Varaiya. Introducing signals and systems—the Berkeley approach. In *First Signal Processing Education Workshop*, October 2000. Online at <http://ptolemy.eecs.berkeley.edu/publications/papers/00/spe1/>.
- [7] Steve McConnell. *Rapid development : taming wild software schedules*. Microsoft Press, 1996.
- [8] John Reekie. Re-architecting the telecommunications program – second proposal. Online at <http://www.eng.uts.edu.au/johnr/pdf/telco-revision-2.pdf>, October 2003.
- [9] John Reekie, Stephen Neuendorffer, Christopher Hylands, and Edward A. Lee. Software practice in the Ptolemy project. Technical Report GSRC-TR-1999-01, Gigascale Silicon Research Center, April 1999.