Chapter 9
Best Practices for Teaching Information Systems Modelling

Steve Wade

Abstract The subject of Information Systems Modelling (ISM) grew out of computer science to fill a gap created by the difficulties programmers had in understanding and solving user problems. The intention behind ISM is to facilitate communication between technologists (many of whom have no idea of the complexity of organisations) and end-users and their managers (many of whom are unable to translate their problems into feasible demands upon technology). "Best practices" in information system development might therefore be considered to be those practices which contribute in some way to improving communication between these two parties. The work described here is primarily focussed on documenting practices that address the issues associated with the seamless transition from a requirements model seamlessly to a technology based system that satisfies those requirements. This has involved reflection on lessons learned during thirty years' experience of teaching Information Systems Modelling in the context of higher education.

Keywords Information systems modelling · Requirements model · Communication · Pattern language

9.1 Background

The subject of Information Systems Modelling (ISM) grew out of computer science to fill a gap created by the difficulties programmers had in understanding and solving user problems. The intention behind ISM is to facilitate communication between technologists (many of whom have no idea of the complexity of organisations) and end-users and their managers (many of whom are unable to translate their problems into feasible demands upon technology). "Best practices" in information system development might therefore be considered to be those practices which contribute in some way to improving communication between these two parties.

S. Wade (✉)
Department of Computer Science, University of Huddersfield, Huddersfield, UK
e-mail: s.j.wade@hud.ac.uk

J. Carter et al. (eds.), *Higher Education Computer Science*,
https://doi.org/10.1007/978-3-031-29386-3_9

Before considering examples of best practice and how these may be taught we will consider the possible consequences of poor communication between technologists and end-users. Focussing on these will enable us to be clearer about the specific benefits that the practices we consider might offer. We will focus on two possible consequences of poor communication.

1. The end-user holds limited views about what they need and fails to realise that alternative superior solutions (which they have been unable to imagine) are possible.
2. The developers hold misguided opinions about what the users need because this has not been clearly explained to them.

There is good evidence to suggest that many information system failures are a result of one or both of these leading to a discrepancy between the system "as required" and the system "as delivered". If we are to avoid this discrepancy we need to deploy development methods that:

1. Provide mechanisms to make sense of and understand the details of human activities that an information system is developed to support. This involves developing some kind of information requirements model.
2. Provide a seamless transition between developing the information requirements model and the design and implementation of a technology-based system to satisfy the requirements captured in the model.

The work described here is primarily focussed on documenting practices that address the above requirements. This has involved reflection on lessons learned during thirty years' experience of teaching Information Systems Modelling in the context of higher education. Some lessons were learned in the eighties when we still made use of plastic flowchart stencils, pencils and plenty of printer paper. More came in the nineties with the rise of powerful CASE tools supporting development methods that required the designer to develop and maintain increasingly elaborate, internally-consistent collections of diagrams. The most significant lessons have been learned more recently as the evolution of powerful programming environments has encouraged a more informal approach to modelling.

In addition to learning lessons from the past it is important to prepare students for the future. Most software systems are embedded in social systems and the resulting sociotechnical systems' boundaries and interactions can be hard to identify. For example, social networks, travel booking and online shopping applications have had far-reaching effects on the way people form relationships, how they travel and what they buy. Becker et al. (2016) have argued that software's critical role in society demands a paradigm shift in the software engineering mind-set. We argue that our students need to be prepared for this shift which will focus on architectural issues that can be addressed through Information Systems Modelling.

9.2 Introduction

This chapter primarily draws on a number of years' experience teaching an Information Systems Modelling module jointly to postgraduate students on an MSc Information Systems Management and an MSc Advanced Computer Science. The module concerns the application of the Unified Modelling Language (UML) throughout the development lifecycle from requirements analysis to implementation. All the students arrive on the module with some background in modelling but those on the MSc Information Systems Management tend to think in terms of business models whereas those on MSc Advanced Computer Science tend to view modelling as high-level programming. This presents the challenge of moving students into a deeper understanding from different starting points and with different preconceptions about the nature of the subject.

In the process of delivering this module we have engaged in the following activities each of which will be described in more detail in the remaining sections of this chapter:

- The design of a pattern language to organise best practice in the application of systems development techniques. In developing the pattern language we were mindful of the need to encourage maximum student ownership of the development process. The patterns could not therefore comprise simple lists of instructions to be followed slavishly.
- The development of teaching materials to document the pattern language.
- Running the module. Observing the progress of the module week-by-week in a number of ways including a range of on-going student feedback mechanisms.

The remaining sections of this chapter reflect on what we have learned from engagement in these activities.

9.3 The Pattern Language

Patterns have been widely used in information systems design over the last ten years. A pattern in this context is a generic solution to a recurring problem expressed in a literary form. The approach has its roots in architecture specifically the work of Alexander (1979). In ISM patterns have been used to ease communication problems and the thinking behind complex design (Gamma et al. 1995). Patterns are usually described by templates which specify the style and structure of a pattern description. Typically the template will include sections for a description of the problem to be addressed, the forces acting to create the problem, a generic solution, a specific example of how this solution might be applied and a discussion of the benefits the solution should provide.

The following example relates to a common (and hopefully familiar) problem in domain modelling where students represent a many-to-many relationship between two objects when the relationship would be better represented by a third object.

Problem

How to model the relationship between two classes that have a many-to-many association with each other.

Forces

- Many-to-many relationships occur often in the real world.
- It can be difficult to implement many-to-many associations in some object oriented programming languages.
- Many-to-many relationships have no direct implementation in relational database systems.
- A many-to-many relationship is usually complicated enough to warrant the addition of an extra class.

Solution

Transform the many-to-many association between two classes into a trio of classes by creating an intermediary class with two one-to-many relationships. The name of the intermediary class should describe the type of relationship being captured.

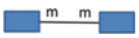
Example

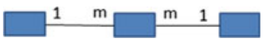
A many-to-many relationship between Student and Module is reconstructed as two one-to-many relationships. One between Student and Work Record and the other between Module and Work Record.

Discussion

We can now store details of module grades for each student as attributes of the new "work record" class.

Summary

If you find this: 

consider replacing it with this: 

The idea is that patterns such as this can be used to guide students away from common problems and into good practice. We have specified many more patterns related to commonly occurring problems. Initially we used patterns drawn from the publications of Ambler (1998), and Evitts (2000). We spent some time re-working and shaping the documentation for these patterns to give coherence to the collection. A key feature of the collection is that relationships are drawn between patterns. When a number of patterns are related to each other in this way we describe the result as a "pattern language". We are therefore trying to develop a pattern language to support the teaching of information systems modelling. Further examples of specific patterns and their relationships will be provided later in this chapter.

9.4 A Framework for the Pattern Language

Most modern courses in Information Systems Modelling are based on the Unified Modelling Language (UML). The UML provides a suite of diagrams that help us to visualise the design of a system. It is published by the International Organization for Standardization (ISO) as an approved ISO standard. Having decided to use the UML we needed to decide which development method to follow. The most popular model-centric approach currently in use is the Unified Software Development Process (USDP) but this is both large and complex. Instead of following the USDP we devised our own simplified method based on our earlier research into the design of a multi-method framework (Salahat and Wade 2009). In that research we proposed a framework for bringing together principles from object oriented approaches to designing software systems and the "soft systems" approach to analysing social systems as part of Business Analysis (Checkland 1999). This approach requires a few words of explanation.

The teaching of Information Systems Modelling tends to focus on issues related to 'hard' systems design. Hard systems are the technical systems that are produced during a development project. Each hard system will be embedded in its social context. This context can be seen as a "soft" densely interconnected system of human activities. It can be argued that hard systems should not be analysed in isolation from the soft systems within which they reside. In analysing the present system or designing a new system, there is the need to consider both the hard system that will be the product of the development, and the soft system within which it will be used. This is challenging because the workings of the soft system are often difficult to understand and the needs of the organisation can be difficult to predict. The UML covers all aspects of hard systems design but has much less to say about the soft system. In contrast Soft Systems Methodology (SSM) focusses on the soft approach.

In addition to augmenting UML modelling with techniques from SSM we also wanted to introduce students to Persona Analysis as a means of developing empathy for users.

Defining personas is an established practice in user-interface design. Blomkvist (2002) describes personas as follows:

*A persona is a model of a user that focuses on the individual's goals when using an artefact. The model has a specific purpose as a tool for software and product design. The persona model resembles classical user profiles, but with some important distinctions. It is an archetypical representation of real or potential users. It's not a description of a real, single user or an average user. The persona represents patterns of users' behaviour, goals and motives, compiled in a fictional description of a single individual. It also contains made-up personal details, in order to make the persona more 'tangible and alive' for the development team. (*Blomkvist 2002*).*

We ask students to develop personas that include a fictional name and life story, a picture, and a 'tag line'—a phrase, supposedly written by the persona, that represents the character of the persona as related to the development project. The case studies that we use in teaching relate to our own department. Accordingly we encourage students to develop a persona for each of the following: A Student of Computer Science, a Student of Information Systems, a Course Administrator, a Lecturer and a representative of an organisation providing industrial placement opportunities.

Although the primary application of personas has been in the context of user-interface design, we have found spending time developing them focusses attention on requirements in a concrete way. Rather than referring to users in an abstract form, students refer to personas by name. So our student of Computer Science becomes Jo Smith who has a first degree in Software Engineering a great deal of confidence in his programming ability but lacks confidence in writing essays and reports—his tag line is "I would rather write code than prose". In contrast our student of Information Systems becomes Sue Rachel who has a first degree in Law, is fascinated by the impact of technology on society but lacks confidence in her ability to write code. Her tag line is: "technology will never replace great people but it can help ordinary people to achieve great things".

Following the basic structure of this framework we developed patterns and teaching materials based around the following topics:

- How to use Persona Analysis to help the developer focus on the needs of the user.
- How to use Soft Systems Methodology to learn about a problem situation.
- How to extract Use Cases from the soft systems models.
- How to develop sequence diagrams related to each Use Case.
- How to develop a domain model from the collection of sequence diagrams.
- How to convert the domain model into a class diagram and database design.
- How to implement the class diagram as an object oriented software system using the, "naked objects", implementation pattern.

We have used the step-by-step approach implied by these questions as the basis for a course structure and an assignment specification. We have adopted a "scaffolded" approach to teaching (Larkin 2001) which involves working through a number of exercises following the structure implied above then asking students to apply the techniques to related case studies for their coursework. The case studies used for assessment were based around the needs of an academic department like our own.

It is beyond the scope of this chapter to discuss each of these topics in detail but for those unfamiliar with the techniques, the following examples are intended to give an idea of what deliverables are produced during each step of the method.

The example used here relates to the decision to introduce a Peer Tutoring System into an academic department to provide extra help to students on a programming module. The idea being that students who are confident in their programming skills would run support sessions for their less confident colleagues. We initially asked students to develop a simple persona for the type of people who would use this system. As explained above a persona is a fictional character that typically has a name, a picture, behavioural traits, common tasks, and a goal that describes the problem the persona wants to see solved or the benefit the character wants to achieve. Personas are not considered in the UML so we devised our own simple template based on the facets listed above. In filling out this template the developer is encouraged to visualise the user in a concrete, tangible way. So the personas mentioned above, named Jo Smith and Sue Rachel, may be involved in the peer tutor system as a peer tutor and peer tutee respectively.

Moving on from Persona Analysis we next ask students to conduct a simple business analysis—with no attention to the design of software. Again this is somewhat outside the scope of the UML so, as explained above, we have introduced techniques from Soft Systems Methodology. The first of these is a rich picture. Figure 9.1 shows a rich picture that we might use to get things started.

We have found rich pictures to be a good way to encourage discussion about the problem situation without focus on any proposed solution. The discussion leads to the development of a root definition: This is defined in SSM as a succinct description of the Human Activity System (HAS) that is required. It is possible to develop multiple root definitions each offering a different perspective on what is required. The following is a possible root definition for the peer tutoring system:

> A system owned by the course that provides programming skills support to students using volunteers with programming experience from the student cohort. The quality of this support will be monitored by academic staff.

Once we have developed a root definition, or set of definitions, we move to developing more detailed activity models. These are called Conceptual Models in SSM and we would develop one for each root definition. The following example, originally presented in Wade et al. (2012), includes activities that might be supported by software and others that will be enacted by humans without the assistance of software.

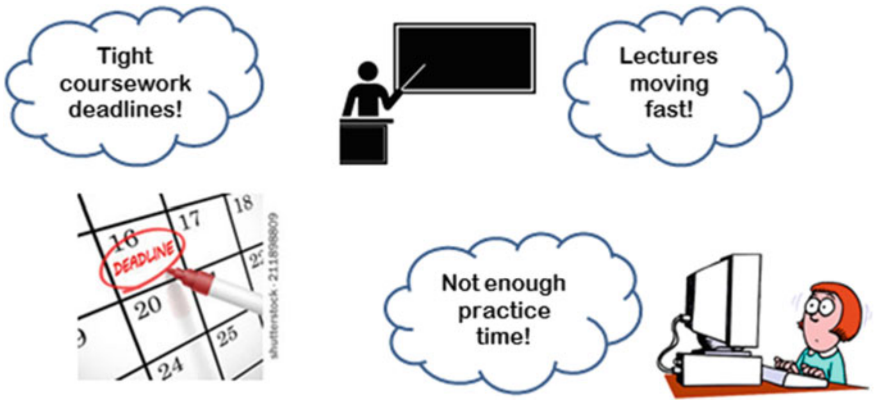


Fig. 9.1 Initial rich picture raising issues for the peer tutoring system

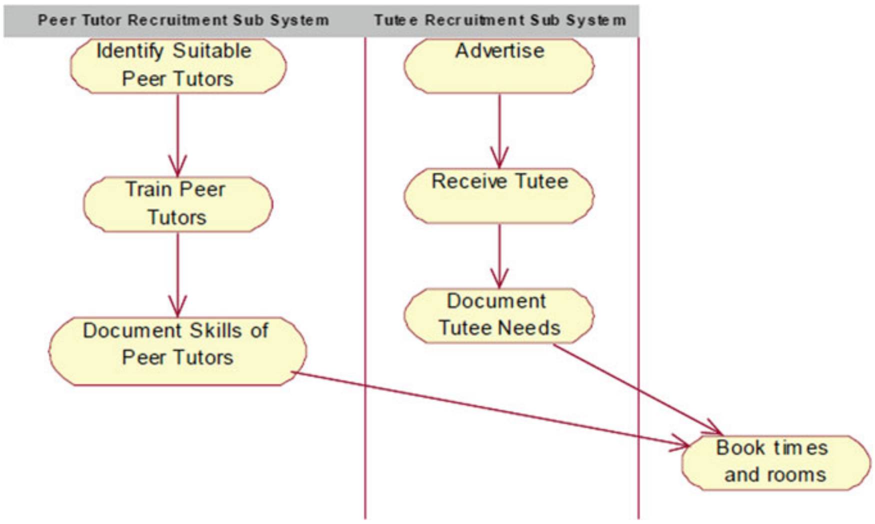

Peer Tutor Recruitment Sub System
Tutee Recruitment Sub System
Identify Suitable Peer Tutors
Train Peer Tutors
Document Skills of Peer Tutors
Advertise
Receive Tutee
Document Tutee Needs
Book times and rooms

Fig. 9.2 Activity diagram

In developing this type of diagram (Fig. 9.2) we encourage discussion about the social system we are supporting. In this simple case discussion might focus on the following questions:

- Will weaker students attend these sessions or will they be primarily attractive to students who are already competent programmers but want further opportunities to develop their skills?
- Should some (weaker) students be required to attend the sessions? If so how do we identify people in this position?
- Should we pay peer tutors? Are their alternatives to rewarding them with money?

This discussion might lead us to develop additional activity models. For example we might consider the need to monitor attendance at the sessions and develop the following (see Fig. 9.3) for an "attendance monitoring" system:

In developing these models we have not concerned ourselves with the question of how software might be able to assist the people involved. This would be the role of a Use Case Model. Use Cases are part of the UML and represent activities that require software support. If we were to develop a Use Case Diagram from our activity models we would be making the transition from business analysis to software design. The Use Case diagram in Fig. 9.4 could be derived from the conceptual model above:

If we focus on the "Print Class List" Use Case we might prototype a simple user interface like Fig. 9.5:

Behind this interface our software system might be composed of collaborating objects. The high-level sequence diagram in Fig. 9.6 depicts the role that a number of objects might have, behind the scenes".

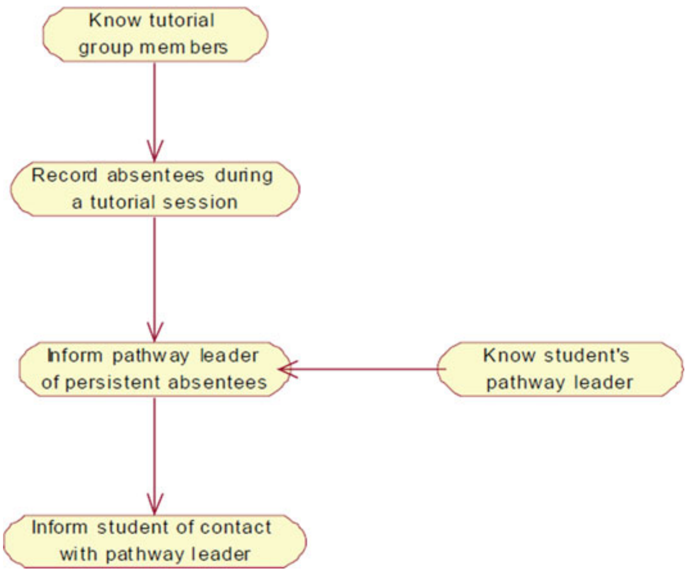

Know tutorial group members
Record absentees during a tutorial session
Inform pathway leader of persistent absentees
Know student's pathway leader
Inform student of contact with pathway leader

Fig. 9.3 Activity diagram for attendance monitoring. Taken from Wade et al. (2012)

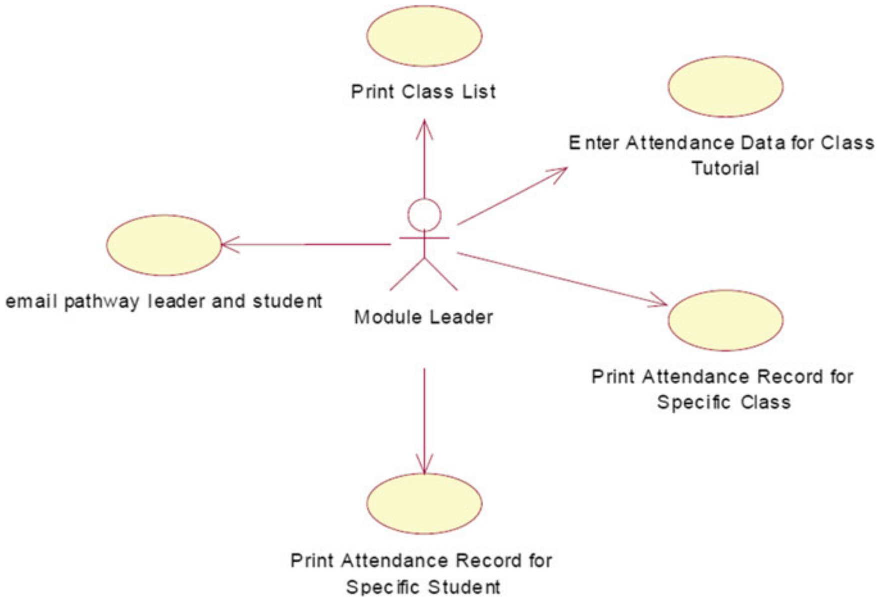
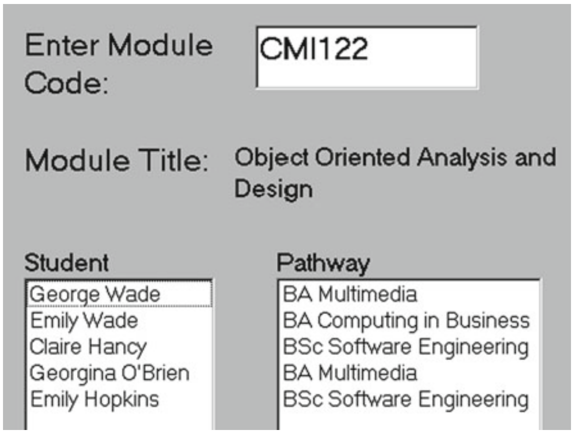

Print Class List
Enter Attendance Data for Class Tutorial
email pathway leader and student
Module Leader
Print Attendance Record for Specific Class
Print Attendance Record for Specific Student

Fig. 9.4 A Use case model. Taken from Wade et al. (2012)



Enter Module Code: CMI122

Module Title: Object Oriented Analysis and Design

Student	Pathway
George Wade	BA Multimedia
Emily Wade	BA Computing in Business
Claire Hancy	BSc Software Engineering
Georgina O'Brien	BA Multimedia
Emily Hopkins	BSc Software Engineering

Fig. 9.5 Screenshot for a use case. Taken from Wade et al. (2012)

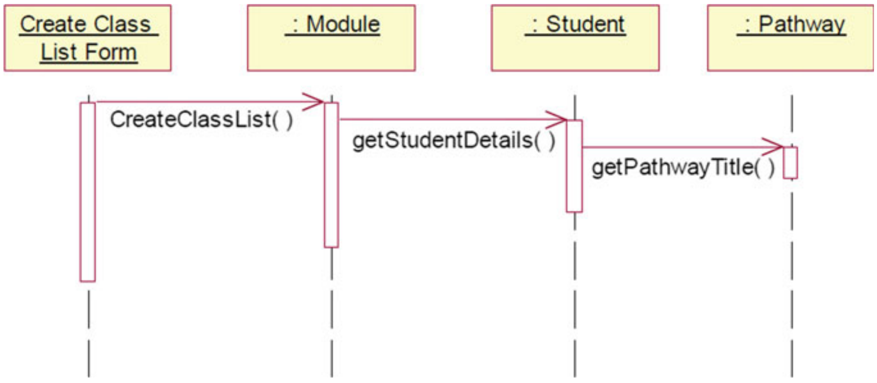


Fig. 9.6 A sequence diagram. Taken from Wade et al. (2012)

We found that students found the transition from Use Case Models to Sequence Diagrams difficult so we provided the following pattern and discussed it in class.

Problem

It is hard to develop sequence diagrams from the Use Case Module. What can I do to make this transition easier?

Forces

A high level Use Case Diagram (such as the one presented above) is fine for a, “mile high”, view of the computer systems behaviour. For many stakeholders, such as sponsors and managers, this will be enough. As designers however we need to open these up and define them in detail. We know what the system presents to the various users (or actors), we need to define in fine detail the, “how”, of that interaction; until we have done this we cannot begin to develop a sequence diagram. There is no prescription in UML regarding what detailed information should be recorded about a use case.

Solution

Document the detailed logic of a Use Case as a series of steps. Where appropriate each step should include reference to one or more domain classes and identify the role that this class should play in the implementation of the Use Case. We can use this description as the basis for developing an initial sequence diagram. The way in which this is done is specified in the "Develop Sequence Diagram from Primary Path" pattern.

Example

This Use Case is concerned with enrolling an existing student in a peer tutor session for which she is eligible.

Key Steps:

1. The use case begins when a student wants to enrol in a peer-tutor session.
2. The student inputs her name and student number into the system.
3. The system verifies the student is eligible to enrol in classes at the university.
4. The system displays the list of available peer tutor sessions.
5. The student indicates the session in which she wishes to enrol.
6. The system checks that the student is enrolled on the appropriate module to join the session.
7. The system asks the student to confirm that she wants to enrol in the session.
8. The student indicates she wants to enrol in the session.
9. The system creates an enrolment the student in the session.

These steps can then be mapped to messages passed between objects in a sequence diagram like the one presented above. It may be that each line in the Use Case description would map to a single message passed between objects. We would develop a sequence diagram for every use case then develop a domain model consistent with all of these sequence diagrams. A domain model derived from this single sequence diagram might look like that in Fig. 9.7.

This domain model can be used as the basis for an object oriented software system design and a relational database structure. We have developed patterns to translate the domain model to a physical database design principally by adding primary and foreign keys to create relationships between tables. A separate pattern discusses ways of mapping inheritance relationships to relational structures.

In developing User Interfaces we encourage students to use the Naked Objects architectural pattern (2012) to generate a graphic user interface directly from the domain model. The pattern uses reflection to automatically generate an initial user interface. Typically this interface will present a series of windows containing icons representing each of the domain classes. A class can then be accessed by double -clicking on its icon to reveal its operations. In the above example I can select, "Module," select a specific module then right-click on that module's "Create Class List" operation to see a list of students currently enrolled on the module. Other

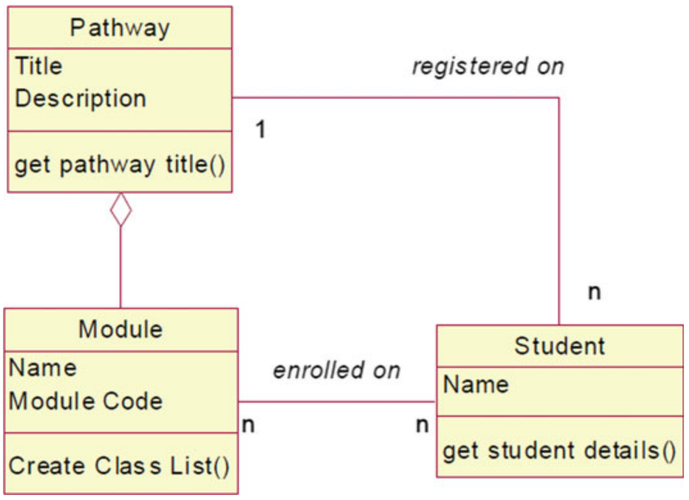


Fig. 9.7 A Domain model. Taken from Wade et al. (2012)

functionality can be achieved by dragging and dropping; so for example if I wish to enrol a student onto a module I can drag the icon representing that student on to the icon representing the module thereby creating the relationship between them. An advantage of applying this pattern is that the resulting relationship between the domain model and what appears in the interface is very direct. A change in the domain model (e.g., the addition of an operation on, "Student", named "Get Coursework Marks") feeds through into the code and then directly into the user interface. From a teaching perspective this helps to reinforce the idea that modelling is both about representing the real world and designing software.

An important part of our teaching has been to identify specific issues that cause difficulties for students then provide specific, detailed guidance of how to ameliorate these difficulties. More specifically we have considered the difficulty of conducting a thorough business analysis before transitioning to design, the transition from a Use Case view to a behind-the-scenes view of the software architecture, the transition from design to implementation with specific attention being paid to the design of an interface that reflects the structure of the software. We have discussed these transitions in terms of patterns that capture good practice and the relationships between these patterns.

We present the patterns in a manner based around the metaphor of different development "rooms". The first room is concerned with developing user personas it is adjacent to a room for developing an "analysis" model based on Soft Systems Methodology. This room contains patterns for developing a range of soft systems models including rich pictures, root definitions and conceptual models. An adjoining room contains patterns for making the transition from analysis to design by translating the analysis model into a Use Case Model with carefully structured documentation of each use case. The next room contains patterns for moving into physical design and then into code. These include: "Develop a sequence diagram showing how domain classes may co-operate in the implementation of a use case". This will involve ensuring that the detailed steps in our use case description relate to the messages being passed on the sequence diagram. A related pattern will explain how to assign operations to classes that map to messages on the sequence diagram.

As mentioned above when a number of patterns are related to each other in this way we describe the result as a "pattern language". We are therefore trying to develop a pattern language to support information systems modelling. We would argue that patterns are particularly suited to this purpose. They are descriptive, not prescriptive (unlike most detailed development methods). They capture expertise in an open-ended format that lends itself to a "hypertextual" structure of resources with links between related patterns that can be explored without forcing a specific sequence of activities. The patterns can also be used as the basis for developing assignment specifications. We will say more on this latter topic in the closing sections of this chapter.

9.5 Running the Module

In light of the above discussion we have been able to propose the following guidelines for developing a module in this area:

1. Design a portfolio-based assessment that can be completed in instalments each instalment being aligned to patterns used in teaching. For each pattern we specify deliverables that can be represented in an assessment grid. In the case of the patterns described above, one instalment could be a Use Case model that is consistent with earlier conceptual models and which is described in steps that map to messages in a sequence diagram. The patterns then become part of the explanation of what is required and are clearly linked to the feedback grid.
2. Provide formative in-class surveys that encourage students to reflect on their understanding of key patterns. In the first example above can they provide an example of a many to many relationship between two classes that could be better represented by a third class? Can they see how the proposed solution would help? Can they apply this learning to the coursework case studies?
3. Encourage students to discuss the individual patterns and how they may be applied to case studies before they complete the in-class surveys or work on the assignment. Students should be encouraged to identify new patterns and fit them into the pattern language or to improve the documentation of existing patterns.
4. Collect data on a regular basis by inspecting samples of student coursework on a week-by-week basis and in-class surveys. Use the feedback to inform improvements to pattern descriptions and the identification of new patterns.

These four guidelines work together to steer the students through the assessment process by frequently monitoring their progress. Hopefully this will lead to continuous improvement in the clarity of the coursework specification and the teaching materials.

With respect to Step 4 above we employ a variety of different ways to collect evaluative information. These include: a pre-course questionnaire that we distribute before teaching begins this is intended to establish the background knowledge and expectations of our students; a series of anonymous in-class surveys to test students understanding and self-confidence in applying the patterns under discussion; short reflective essays were made part of the coursework portfolio in which students were asked to give their personal opinions about the usefulness of the pattern-based approach and focus group discussions were held in class.

In addition to the above, during marking, we carried out an analysis of the most common mistakes made by students in their coursework. We discovered a number of recurring mistakes and made changes to the pattern language to discourage these. A number of examples are given below.

- Inconsistencies between diagrams. For example operations appearing in the sequence diagram that are not present in the class diagram.
- Failure to use domain-specific vocabulary as presented in the case study materials. In the above example we referred to "Pathway" where others may have used the term "Course". It is important that the language used in the models can be found in the user documentation.
- Operations that have ambiguous or misleading names. We have seen operations with names like "Update all" or "Reconsider" these names are almost meaningless to anyone but the original programmer.
- Database concepts (e.g. primary and foreign key dependencies) used in the domain model. The domain model is meant to be an abstract representation that might be used in the design of object oriented software or an ontology it is not a physical database design.
- Operations not supported by attributes or relationships. Some operations depend on the availability of connections to other classes or of data properties that must be included in the domain model.
- A lack of consistency between the SSM models and the Use Case Model. For example Use Cases that cannot be inferred from the activities in conceptual models.

We continue to work on developing patterns that will steer students away from these types of mistake. We plan to present these via a website based on our "rooms" metaphor with hyperlinks between related patterns. We would argue that working in this way has encouraged us and our students to consider important aspects of information systems design that are often overlooked in courses that teach Information Systems Modelling. A few of these are listed below:

- We encourage students to adopt multiple system viewpoints from different personas. The acknowledgment and exploration of these viewpoints emphasises the important point that typically most systems have more than one purpose and many unexpected consequences.

- We are encouraging our students to consider the total problem situation and to be aware of the need to ensure that all, not just the most obvious, significant issues are addressed;
- Our approach is strongly goal-oriented. The central focus on Use Cases ensures that derived requirements are justified with respect to stated goals;
- All the techniques documented in our pattern language are well established and well tried. We haven't presented patterns for anything that has not, in one form or another, proved useful to developers. We hope the manner of presentation is more effective than user manuals or detailed methodology documentation but the intention is not to present anything new but to organise and document the distilled wisdom of the many talented software engineers and systems analysts who have worked in this area over the years.

9.6 Conclusion

This paper has described our approach to teaching Information Systems Modelling over a number of years. We have described the approach as being built around the scaffold of a multi-method systems development framework which we have documented in the form of a pattern language. This basic structure has been tried and tested through a number of feedback mechanisms (including in-class surveys, focus group discussions and reflective essays) and a concordant assessment strategy. The results obtained through these feedback mechanisms have encouraged us to continuously refine our teaching materials and assessment strategies—we believe these changes have all been improvements.

References

Alexander C (1979) The timeless way of building. Oxford University Press, New York

Ambler SW (1998) Software process patterns. Cambridge University Press

Becker C, Betz S, Chitchyan R, Duboc L, Easterbrook S M, Penzenstadler B, Venters CC (2016) Requirements: the key to sustainability. IEEE Softw 33(1):56–65

Blomkvist S (2002) The user as a personality: using personas as a tool for design. In: Position paper for the workshop 'theoretical perspectives in human computer interaction' at the interaction and presentation laboratory of the royal institute of technology, Sweden, September 3, 2002. Available at http://www.nada.kth.se/~tessy/Blomkvist.pdf

Checkland P (1999) Soft systems methodology: a 30-year retrospective. Wiley, Chichester

Evitts P (2000) A UML pattern language. Macmillan Technical, Indianapolis, Ind

Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software. Addison Wesley, Reading

Larkin MJ (2001) Providing support for student independence through scaffolded instruction. Teach except Child 34(1):30–34

Salahat M, Wade S (2009) A systems thinking approach to domain-driven design. In: The proceeding of UKAIS2009 conference, Oxford University, Oxford, UK

Wade S, Salahat M, Wilson D (2012) A scaffolded approach to teaching information systems design. Innov Teach Learn Inf Comput Sci 11(1):56–70