UML and Design Layers Provide a Course Design Paradigm and Notation to Create Robust Technology Courses in Rapidly Changing Environments

C. Richard G. Helps, Stephen R. Renshaw,
Information Technology, Brigham Young University

Abstract

Rapid changes in technology require frequent course re-designs and new lab equipment for various portions of the course. Keeping up with these changes requires significant time from faculty and also requires substantial financial support. We need methods for designing and modifying portions of courses to allow teaching of the most current technology without a continuous complete re-design of the whole course. The Design Layers approach offers a thinking paradigm that allows us to think about instructional design in a way that parts of the course change at different rates and can more easily be replaced if they are identified as separable components. This is analogous to object-oriented software design, where independent objects are replaceable as the system evolves.

This paper shows how Design Layers can be used to analyze course design and then presents the Unified Modeling Language (UML) as a notation for instructional design. We show how a variety of instructional scenarios can be analyzed using UML notation. The notation reveals important relationships in the instructional design and helps identify weaknesses and dependencies between course components. It also allows designers to compare different structures and strategies in technical instruction.

The UML notation system can be used to illustrate outcomes-oriented teaching approaches including continuous-quality improvement mechanisms built into the course design and instruction. This notation can thus show not only the instruction but also the various evaluation mechanisms used to monitor and improve the course for future students

Introduction

Technology, as an academic discipline, is focused on application and integration of existing technologies. Students in technology disciplines need a substantive grounding in science and engineering design principles. They also need a deep understanding of their specialist discipline. Since technology has a strong emphasis on integrating various technology components it necessarily requires that technologists stay current with technological changes. Part of the challenge of teaching, learning and practicing in a rapidly evolving environment is the need for life-long learning. This presents a number of challenges for university faculty teaching in these disciplines.

One major challenge for technology faculty is the need to constantly review and re-design their course material to include significant technology developments within their field. This is

particularly true for computer-oriented technologies that are strongly influenced by Moore's Law, but the effects are also present in other technology disciplines. This constant updating can be an onerous burden and distract faculty from other responsibilities and opportunities in their scholarly work and teaching. The emphasis placed on applications of principles in technology disciplines means that not only does course material need to be revised frequently but laboratory assignments and project work needs to be revised. In senior and graduate classes in technology disciplines, where the technology being taught often reflects state-of-the-art practices in industry, this could require laboratory redesign or updating as frequently as yearly. The attendant financial costs and time invested by the faculty to provide well-designed laboratory experiences can become overwhelming. For example, in teaching a class related to modern computer operating systems involving, say the Windows and Linux operating systems, it is probable that at least one of them will go through significant changes as often as once per year, particularly if the various tools that are used to work with them are also considered. Updating lab configurations to reflect these changes requires changing lab computer software, and possibly hardware too.

In this changing environment it is highly desirable to explore paradigms of thought and methodologies which enable technology faculty to manage the necessary changes in their instructional design. This paper presents both a conceptual approach to the problem and a notational system which supports flexible and adaptable course design.

Design Languages and Layers

In the early days of software development computer programs were considered to be monolithic software constructions designed to solve various problems. As the discipline of software design evolved so paradigms were developed to view software design in terms of replaceable modules. That led to the current approach of object-oriented software design. Similarly other parts of computer design, both hardware and software, developed standards and interfaces so that a complete system could be built from cooperating modules coming from several different sources.

In a similar manner instructional design can be conceived not in terms of creating a monolithic pedagogical structure through various design steps but rather as creating a multi-layered structure where different aspects of the instructional design can be separated conceptually. This paradigm is known as the Design Languages or Design Layers approach.

The concept of Design Languages was strongly influenced by the work of Christopher Alexander[1] in the architectural field. Alexander is also credited with having inspired many of the concepts in software design patterns[2,3]. In architecture one can conceive of a building as being composed of many layers. For example we can think of the walls, the electrical systems, the furnishings, the décor and the climate control systems as different interacting layers. As the building evolves over time these layers can be separately removed and replaced. If the building is designed for evolution this removal and updating can be done with relatively little disturbance to the rest of the structure. In a similar manner we think of computer systems which allow us to remove software and install new software while leaving, say, the hardware and the operating system unchanged. In a like manner an instructional design can be considered to consist of a number of layers. As new technology or new teaching approaches are developed so the course design can be changed without tearing down and replacing the whole structure. A set of design

layers that reflect this approach have been developed by Gibbons[4]. He proposes seven design layers. A brief description of each layer and its purpose is as follows:

1. **Content**: the content layer is concerned with the *abstract* organization of the instructional content. How will it be captured, partitioned and represented?
2. **Strategy**: Every artificial thing the designer does. All of the dimensions of all the artificial events. Deciding how to structure the course presentation is included here.
3. **Control**: The actions that the learner can perform. How does the learner speak to the system?
4. **Message**: The structure and message types that the system will use to communicate with the learner. Messages can be verbal, written, body language etc.
5. **Representation**: Motions and signs, instantiation of the messages with specific multimedia content; colors, fonts, screen arrangement, etc.
6. **Medialogic layer**: The logic that controls the media that carry the message to the learner.
7. **Data management:** Record keeping, progress monitoring, accumulating the record of the instruction.

The layers can be applied to technology instruction. For example, if a course is teaching computer operating systems and the only change made is to update the operating system then it is likely that the instructor will till be teaching the same concepts but that the **representation** of those concepts will change.

Course Evolution

As has been discussed above courses evolve due to changes in technology. They also evolve due to changes in instructional approach and for other reasons. For example an on-campus course might change to distance education course. We are suggesting that this process of evolution is continuous and that different layers or aspects of the course will evolve at different rates. In other words we are designing course materials in a constantly changing dynamic environment. For example, equipment failures could cause changes in the instruction on a scale of days and changes in lab equipment could cause changes in a scale of years while changes in basic technological principles might change on a decade time-scale.

In order to manage this course evolution for the best learning of the students a system for documenting course design provides a way to identify the different layers of the course, to identify instructional dependencies and thus allow the instructional designer to remove the layers that need changing and replace them with minimal disturbance to the rest of the course design. In practical terms we hope to be able to update our course without having to replaces all of it by having a clear understanding of the aspects that are changing and clear documentation of their relationship to each other and their relationship to the outcomes and objectives of the course as a whole.

Documenting Course Design

One of the issues with any design notation system is its precision relative to its breadth. If a design notation system is sufficiently rich to convey detailed information, it is unlikely that it will be sufficiently general to convey the overall intent of the designer. Consider the example of

musical notation. Over the centuries this has developed so that it can express melody, harmonies, timing and dynamic variations, yet with all this it cannot express a whole ballet, just the music. The choreography, lighting and other aspects of the ballet are not conveyed by the musical notation.

The way this problem is usually solved in design disciplines is to use multiple independent but related design notations to convey a complete design. An example of this is provided by modern electronic designs. An electronic product is modeled by a combination of circuit schematic diagrams, PCB layout diagrams, specification lists of part parameters and three-dimensional drawings of the case showing the physical relationship between user controls and electronic sub-systems. All of these are required components of the design for a working unit.

The Unified Modeling Language (UML) has achieved both generality and precision by combining many related software design notations into a tightly coordinated whole to express a software design, incorporating aspects of data, control, timing, hardware constraints and more.

Computer software has developed over the last few decades within a climate that has recognized and rewarded rapid development of reliable systems. In the early days of computer software design the major focus was that the programs should work. As computers became more widely used outside of the technical community that developed them, the need for improved reliability, better user interfaces and lower development costs grew. Not only did customers require a more multi-faceted approach to design effectiveness but many more people demanded computer software. A key goal for software design became to create reusable and interchangeable components. In order to satisfy a software market facing explosive growth between the 1970s and the 1990s, design techniques were developed to meet different needs in different areas of application. Multiple techniques were developed to facilitate different ways of looking at design problems. Some design tools, such as flow-charting or pseudo-code, were designed to specify the control or sequencing of programs. Data-flow diagrams were used to map the flow of information through the system. In these the control structures were considered peripheral to the more important task of tracking the data. State "machines" were developed to describe how a system could change from one set of conditions to another, either without time constraints or including time constraints (Harel state charts)[5]. Neither of these types of state machines contained sequencing or data information. The importance of the end-user's interaction with computer systems became more apparent as computers became ubiquitous in modern life, and considerable effort was expended in developing various Human Computer Interaction (HCI) design techniques.

Each of these techniques was successful in its own specific application field; however, designers were required to use an eclectic collection of them for large, complex designs, and these techniques were not developed to work together. For large projects programmers, database designers, hardware engineers, HCI designers, system analysts and other specialists would each use their own set of tools to cooperatively develop a complete system. This led by stages to the development of what is probably the most widely accepted software design notation system today, the Unified Modeling Language. It should be understood at this point that UML is described as a language. Since the term "language" has multiple interpretations it will be noted here that UML is a language in the same sense as other computer languages such as C++, Java

and so on. Specifically it has a clearly defined syntax and notation system with limited contexts in which the symbols can be used. The complete specification is available on-line[6] but is difficult to read. Good summaries are available, such as Fowler's[7], illustrating the essential elements.

UML provides a mechanism for expressing many aspects of design, from initial analysis and user interactions through various aspects of the design and on to the final implementation. It provides a way to express component design detail as well as design interactions.

Benefits of UML (or any other design notation system)

The benefits of using UML are similar to those of other formal design notations. Firstly, a standardized design notation allows people to collaborate on and communicate designs from the conceptual phase through the implementation, testing and documentation phases. Particularly significant elements of design can be analyzed and debated by the broader community and, if successful, can be re-used in future designs, thus extending the vocabulary of the design language. Secondly, a formalized, standardized language, if it closely models the intended application domain, makes it possible to analyze the effects of a design before it is implemented. An unambiguous vocabulary and grammar make it possible for computers to manipulate and represent systems and, to a certain extent, verify their correctness. This verification is usually limited to the correctness of the syntax and symbol usage, but even that limited verification is extremely valuable. Another benefit is that of maintenance and adaptability of the design. By having the original design both broadly and precisely documented, it is much easier to understand the designer's intent later and to implement changes in the system. Since UML defines both user-system interactions as well as precise system logic it allows the modification of the implemented system while still meeting the user's needs.

The formal definition of UML is so strong that its use can be partially automated. That is, automated systems can check to see if the rules have been followed, can relate the same components appearing in different aspects of the design and can even generate computer code structures based on the design, rendering the implementation step much easier.

The formality of UML is both a benefit and a constraint. The precision of the definition leads to unambiguous communication of design ideas but the constraint of working within that language system does lead to certain kinds of design and encourage certain kinds of thinking. As has been noted by Kuhn[8] technological exploration tends to function within paradigms imposed by the culture of the technical community. UML is the product of such a technical community and is embedded in the current software development culture of object-oriented software design.

The generality, precision and formality of UML can be contrasted with instructional design (ID). ID in its recent history has developed many processes, design languages and methodologies for creating and recording instructional designs. Many of these processes, such as the ADDIE ("Analyze, Design, Develop, Implement, and Evaluate") design model, have become widely recognized and accepted. Similarly models, such as those developed by Gagne'[9] have also been widely adopted by instructional designers. Despite this wealth of available design methods, models and notations, there is no single system, widely used and recognized, for expressing complex instructional designs. Designers may use one of several design notation systems or, in

many cases, define their own system appropriate to their design philosophy. This necessitates explaining their notation system whenever they wish to share a design. The reader also needs to absorb the new, unfamiliar system and inevitably will fail to fully appreciate it at first, thus risking missing important aspects of the design.

It must be emphasized that UML is not a design methodology. No guidance is given within UML on how the designer should design. It provides a set of tools which encourage one to think of certain aspects of design or to view the design in certain ways. In this it bears an important similarity to Design Layers which also provide a way to think about instructional design without actually defining the ID process as, for example, the (ADDIE) instructional design model does.

UML Diagrams

A UML documented design consists of a series of "diagrams' with specific purposes. UML is a graphical language or notation system[10]. Designs are specified in terms of diagrams. The OMG specification[6] introduces them as follows

> "The choice of what models and diagrams one creates has a profound influence upon how a problem is attacked and how a corresponding solution is shaped. Abstraction, the focus on relevant details while ignoring others, is a key to learning and communicating. Because of this:
> ● Every complex system is best approached through a small set of nearly independent views of a model. No single view is sufficient.
> ● Every model may be expressed at different levels of fidelity.
> ● The best models are connected to reality."

The concept of abstraction, expressed above, is very important in ID as well as in software design. One of the ideas emphasized in the design layers approach is that almost all of the layers are abstract. We are trying to extract the essential concepts behind the content, strategy, controls and so on to be able to communicate with the learner effectively. There is also a notable similarity in the concept expressed here to the many nearly independent views of an ID design. This is directly reflected in the design layers approach to ID, where the diversity of viewpoints is recognized.

The diagrams in UML are as follows (from [6]):

> "In terms of the views of a model, the UML defines the following graphical diagrams:
> ● use case diagram
> ● class diagram
> ● behavior diagrams:
> • statechart diagram
> • activity diagram
> • interaction diagrams:
> • sequence diagram
> • collaboration diagram
> ● implementation diagrams:

• component diagram
• deployment diagram"

Shalloway and Trott[11], suggest that the different UML diagrams apply to different phases of design. Without going into detail it is worth mentioning that this diversity is part of the power of a generalized language. As we develop instructional designs we deal with customer requirements and then develop strategies, controls and other aspects of the design. All the different aspects, or layers, are expressed in the final design but certain of them are emphasized more at different times.

No further background into UML is provided here. The reader is invited to look into some of the references for details as required.

Applications of UML and Layering to Technology Course Design

Adapting UML to document instructional design requires some simplifications. Martin Fowler points out that UML can be used "sketch" a design idea or "blueprint" a design or as a programming language, with increasing levels of formalism. Most appropriate to ID is to use UML to sketch or record course designs. Used in this mode the basic symbols relationships and ideas of UML are used but where necessary liberties are taken to simply express ideas.

With this in mind we could use Class Diagrams to show relationships between course components. See fig 1 below.

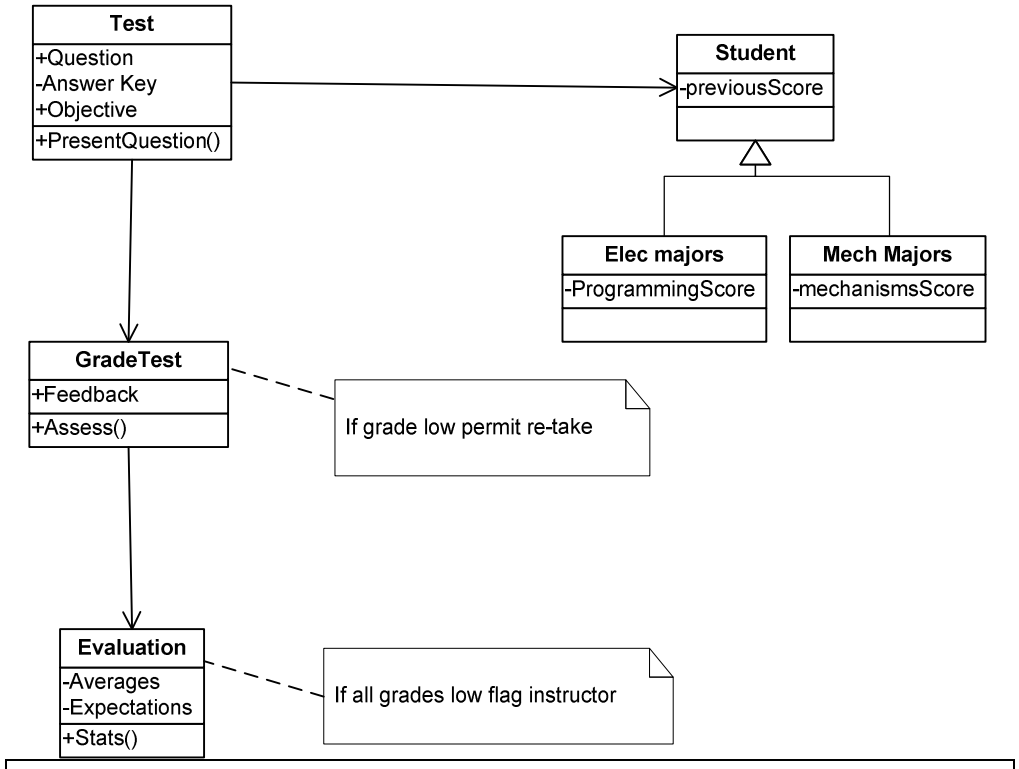

Figure 1 Course Objects

The dependency relationships in figure 1 show relationships between the entities. Note that we can view the instructional design in different ways. Consider figure 2 below, which shows essentially the same instructional objects but focused differently.

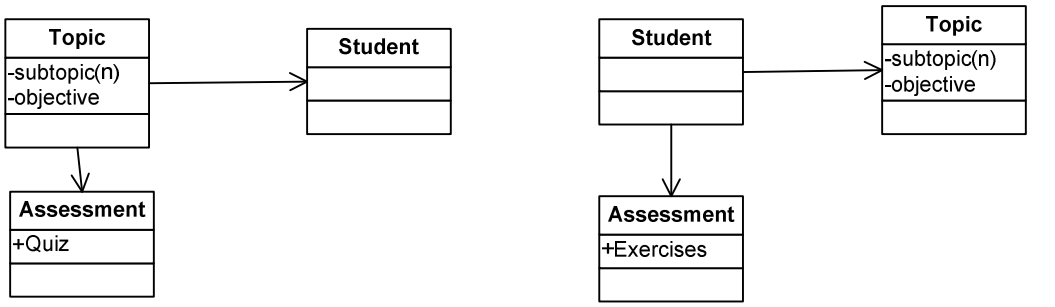

Figure 2 Alternative dependency relationships

Here the notation system helps us decide if the test, the topic or the student is the focus of our instructional design.

Here is another example of how the dependency arrows of UML show what is happening in an instructional situation. The situation documented here is that a very important concept is part of a course. In the left hand diagram of figure 3 the instructor wishes to ensure that students have a thorough understanding of the concept and practice. In order to do so the concept and also the associated lab assignment and equipment are discussed in class. Thus the dependency arrows flow from the presentation to the assignment and lab. Furthermore the instructor reinforces the lab assignment by linking it to a homework assignment. Furthermore when the lab assignment is complete the instructor reviews it in class (dependency arrow from lab to class presentation). With all this care the instructor has a high probability of being successful. Next time the course is taught the equipment technology has changed. The instructor now has a host of problems. Not only must new equipment be acquired, installed and tested, all the instructional material must also be changed. The class presentations, homework and lab assignment have all been affected.

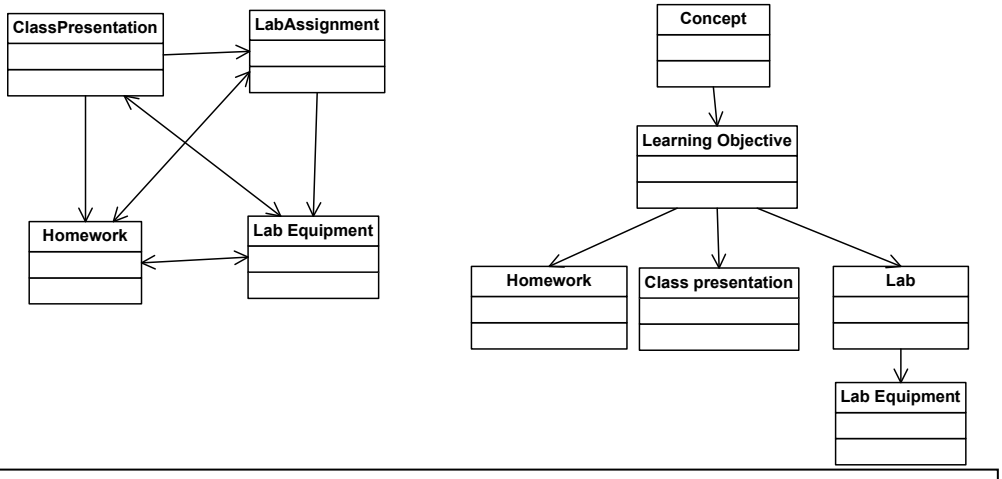

Figure 3 Two Alternative Dependency Relationships between Instructional Objects

Now consider the right-hand design. Here the instructor has made the choice to focus on teaching the concept leading to a defined learning objective. The homework, class and lab experiences are all designed to support that objective. The lab equipment is designed to support the lab experience. When the lab equipment needs to be updated most probably the only modules to be affected will be the lab equipment module and, to a lesser extent, the lab assignment module.

Class diagrams document dependency and inheritance relationships. Other diagrams reveal different approaches to teaching.

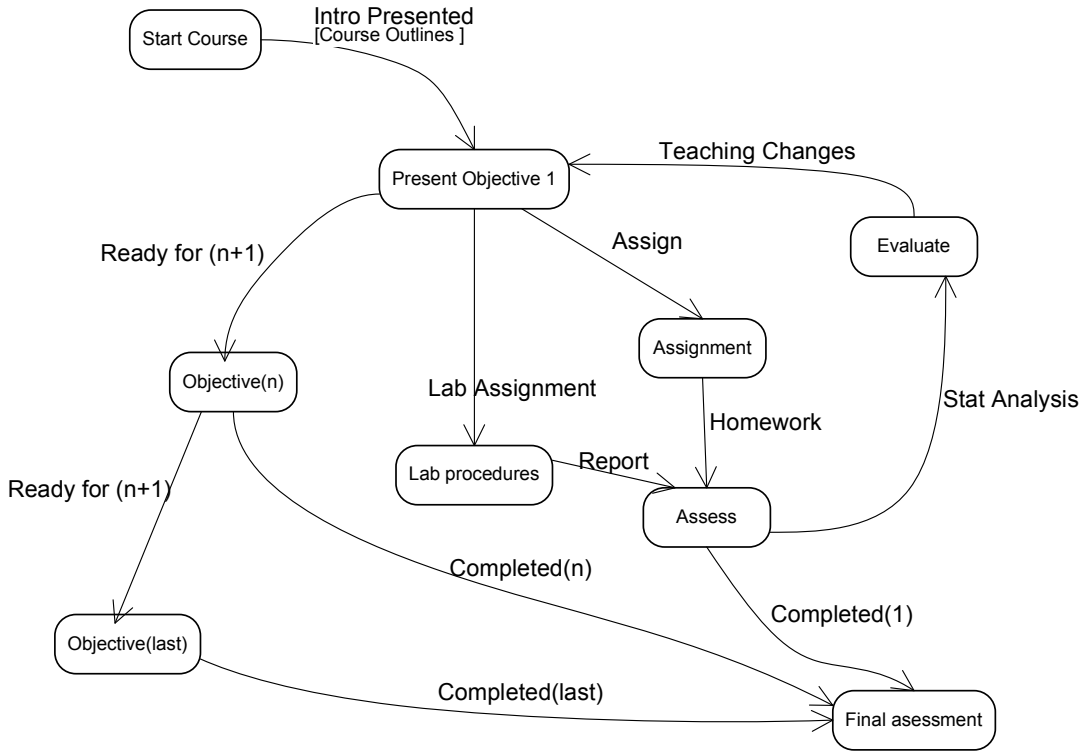

Figure 4 State Diagram for Event-Driven Instruction

The state diagram proposes that a class objective is presented and that triggers lab and assignment events. These are assessed and the results feed back into the objective. If the objective is satisfied the next class objective is triggered. This continues until all objectives are covered, which triggers a final assessment. Here instruction is triggered by objectives being completed rather than by class hours.

Further diagrams can be generated to illustrate the different aspects of the course design. We will only show one more example here. An important aspect of instruction is the relationship between the instructor and the student. These are represented in the Control and Message layers of the Design Layers paradigm discussed earlier. In order to document this relationship we can use the Use-case diagram from UML. A sample diagram is shown below.

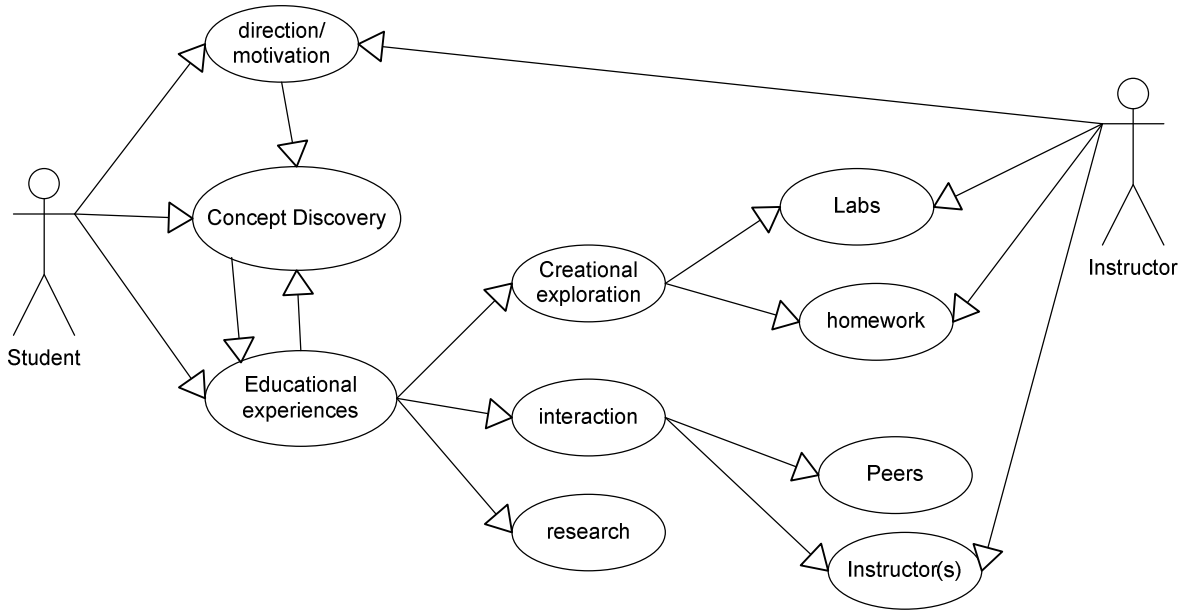

Figure 5 Use-Case Diagram for Instructor Student Interactions

The different UML diagrams can thus be used to illustrate an endless variety of instructional design aspects. Activity diagrams (flow charts) can show instructional strategy, Sequence diagrams can show relative (or precise) timing of courses and obviously UML is an appropriate tool for illustrating media-logic and data management layers of instructional design.

Conclusion

The rapid pace of technology change is both a challenge and a delight to students, professionals and instructors in technology professions. The constant re-design of courses to meet this challenge can be daunting for technology faculty. Techniques exist to make the system not only manageable but flexible and adaptable. The paradigm of Design Languages provides a way to view technology instruction so that the different rates of evolution of the different aspects are revealed. Once revealed it is much easier to design technology instruction so that it can be updated as necessary without disrupting design of the entire course. UML is a flexible and powerful design notation system. It provides a useful notation system for documenting instructional design. The relationships between instructional course aspects are revealed by the different UML diagrams.

Neither Design Languages nor UML constrain the instructional design, neither of them represents a design methodology although they permit analysis of instructional designs.

The work done so far has shown that these approaches are powerful enough to analyze and document a variety on instructional design situations and that the documentation can indeed isolate those aspects of instructional design which require updating.

Bibliography

1. Christopher W. Alexander. *Notes on the Synthesis of Form* Harvard University Press (June 1, 1970)
2. Alan Shalloway, James R. Trott, *Design Patterns Explained: A New Perspective on Object-Oriented Design* Addison-Wesley Pub Co; 1st edition (July 9, 2001)
3. Gamma, Erich, Helm, Richard, Johnson, Ralph, Vlissides. *Design Patterns; elements of reusable Object-Oriented Software* Addison Wesley, 1995
4. Andrew S. Gibbons, *What and how do designers design?* TechTrends, v47, Issue 5
5. Phillip Laplante, A, *Real-time systems design and analysis: an Engineer's Handbook* IEEE Computer Society Press, 2nd ed. c1997.
6. Object Management Group, Inc., *OMG Unified Modeling Language Specification An Adopted Formal Specification of the Object Management Group, Inc*. March 2003 Version 1.5 formal/03-03-01 Accessed from http://www.omg.org/docs/formal/03-03-01.pdf
7. Martin Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, Addison Wesley, 3rd Edition, 2004
8. Thomas S. Kuhn *The Structure of Scientific Revolutions* University of Chicago Press, 3rd Edition 1996.
9. Gagné, R. M. *The conditions of learning and Theory of Instruction***,** (4nd ed. 1985). New York: Holt, Rinehart & Winston, Inc.
10 Grady Booch, James Rumbaugh, Ivar Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley Professional 1999
11. Alan Shalloway, James R. Trott, *Design Patterns Explained: A New Perspective on Object-Oriented Design* Addison-Wesley Pub Co; 1st edition (July 9, 2001)

Biographical Information

C. RICHARD G. HELPS
Richard Helps is the Program Chair of the Information Technology program at BYU. He is also a TAC-ABET and CAC-ABET program evaluator. He spent ten years in industry as a control systems design engineer. He completed BS and MS degrees in Electrical Engineering at the U of Witwatersrand, South Africa and a further graduate degree at the University of Utah in Electrical Engineering. His primary scholarly interests are in embedded and real-time computing with its instrumentation and control aspects and in technology education

STEPHEN R. RENSHAW
Stephen R. Renshaw is an Instructor of Information Technology at BYU in Provo, UT. He received a B.S. and an M.S. in Computer Science from BYU in 1985 and 1987. He has worked in various Information Technology areas including: telephony, process control, embedded systems, system integration, networking, and health care computing. Current research interests include: system integration, system modeling, networking, and experiential and distance learning applied to technology.