Document: 2004-2305

Software Engineering Emphasis for Engineering Computing Courses: An Open Letter to Engineering Educators

William Hankley
Department of Computing & Information Sciences
Kansas State University
Manhattan, KS 66506
hankley@cis.ksu.edu

Abstract
Software is an important component for engineering development for all engineering fields, not just for computing sciences. This paper addresses what might be included in a service course for engineering majors on the topic of software development. Typically, that consists of a single course on either programming or on using software packages, but those basic skills are inadequate foundation for real software product development. We recommend a second course on software design and development, to include concepts of interaction design, usability, aspects of common software structure, architecture patterns for common program kinds, standard libraries, and software tools. The notation for such design will center on UML, the unified modeling language, but the design experience is more than just knowing UML. These concepts form the basis for designing software rather than just programming in accordance with some design; design concepts are a foundation for communication between application engineers and software engineers. Such a software course can be an effective early design experience for engineering majors.

Introduction

This paper is addressed to engineering educators in departments in which students may take some computing courses as breadth topics. The point of the paper is to recommend that a software design course is an essential second course for engineers, to explain why this is so, and to identify what topics would be in such a course. Unfortunately, a design course is usually not the second course available for non-CIS majors. If that is the case at your school, then it will take negotiation with the CIS department to offer such a course.

For computer science and engineering instructors, the topics of this paper are not new. The figures comprise a tour of UML notation, which is commonly used in both programming and software design courses. The point is not that engineers should see UML notation; if that were the intent, then adequate introduction could be done in a first programming course. Rather, the point is that engineering students (beyond just Computer Science and Engineering majors) should have an experience of software design. Today, that design experience is often reserved as a senior capstone course for CSE majors.

A key point of a design course is that students should see many different kinds of software models, such as structures for data management, visual direct manipulation, real-time control, games, and many others. The second point is that students should have extensive experience in

the design process consisting of problem analysis, model development, and review and evaluation of the model components, and finally some implementation. The difficulty in teaching a design course is that correctness and quality of software models must be judged by the instructor, which is very time consuming.

It must also be emphasized that the design course we offer is balanced with programming assignments. Early assignments deal with using tools and composing standard components that will be used in the course project. The project consists of design phase, including several reviews of the design models, followed by implementation of core parts of the design. The major code evaluation is to check that program code matches the design model documents.

Background

Most engineering students take at least one computing course. If they take only one such course, that course is usually a programming course. Sometimes an engineering department will teach its own computing course, for example a course on numerical computing using FORTRAN. That occurs at our school because the CIS department does not teach FORTRAN. In other cases engineers take the core CIS programming course. At our school students from several engineering departments take the core programming course along with computer science, information systems, and computer engineering majors. That core programming course uses the Java language, which is founded on object concepts, which are widely accepted as key to building of large software systems. Some reasons why we use Java (rather than Microsoft languages such as C# or Visual Basic), is that Java does run on may machine platforms, it is relatively easy for students to set up Java for work on home computers, and at the time we selected Java it was certainly more secure than the alternative of C++.

Now, even though our first course has an object-oriented foundation, my observation is that most students who complete the course are not be able to create a reasonably designed new object-orient application. The reason is that design is a constructive task which requires some experience, whereas assignments in a first course are oriented to coding very specific assigned tasks. Of course, by necessity, the first step of learning focuses on programming rather than on design.

Should engineering students be advised to take a second computing course? I recommend this for the following reason: There is a "language" (in part a visual notation) and methodology for specifying the requirements and design of software systems. If engineers need to create their own programs they should know how to program; but, if engineers ever need to interact with software engineers in describing and designing software systems, particularly software to be incorporated into larger engineered systems, then those engineers should know the notations and processes of software design.

The curriculum problem is that software design is not the usual second course in computing. Many departments of computing were created during a period when the emphasis was on the underlying science concepts. In that era, the usual second course dealt with algorithms and data structures, often with emphasis on building basic data structures such as lists, trees, hash table, etc. It was felt that building such structures would establish understanding of use of pointers (references in Java) which are usually a major source of errors. However, in recent years, the

ACM recommendations for computing curricula have been expanded to accept several alternative patterns for the core courses in computing. One of the options for core courses, identified as "Objects-first" option, suggests either a two or three course sequence. The two course sequence consists of CS111o. Object-Oriented Programming followed by CS112o. Object-Oriented Design and Methodology [1] . The CS112o course is close to what we suggest in this paper. One key aspect of CS112o is that it deals with use of standard collections data structures but not implementation of them. For Java, this means students should understand the concepts and the use of the existing collections library rather than building lists and trees from scratch [2]. The ACM objects-first sequence reflects a software engineering emphasis, with focus on building software. Some schools, for example, the Bachelor of Software Engineering program at the Milwaukee School of Engineering, follow this structure [3].

Software Design Artifacts

This section give a small tour of the items by which one forms a design for software, which includes requirements, the static structure, and dynamic behavior. In one sense, such design is not just about software, but it defines requirements for an abstract system which has software components, typically the user interface and model for system control.

The core of software design is to represent essential structure (both static and dynamic) with a notation that is more rapid to write and more abstract that actual code. In the past, pseudo-code was though to be such a notation, but it is no longer viewed in that light. Both the scale and scope of pseudo-code are too small. Now, the commonly accepted notation for most aspects of software design is the Unified Modeling Language [4]. Just as design documents for a building consist of several layers (heating, wiring, plumbing, etc.), a UML model consists of several aspect model diagrams. Some key diagrams for basic design are described and illustrated below. The point of software design is that once these documents are developed, programmers should be able to develop program code to implement the target software. Increasingly, CASE (Computer Aided Software Engineering) tools will help in generating program code from UML models and in checking consistency of code with respect to the UML models.

1. Use-case model: The model consists of a use-case diagram, as Figure 1, and supporting documents. The use-case diagram shows the primary "users" and actions or operations they can invoke. It provides the focus for the "what is it" question for the software. Often, students tend to miss this aspect of development. Part of the elaboration and documentation of the use-case model should be an analysis of who are the users, what are their goals, and what are natural scenarios of interaction; however, such analysis not obvious in just viewing the use-case diagram. Cooper [7] presents several compelling examples of such analysis.

2. Class domain model: This consists of an abstract class diagram, as Figure 2, and supporting notes. The diagram shows the static conceptual class structure for the domain of discourse for the target software. Classes represent domain entities, transactions, descriptions, collections, and other such conceptual components. The diagram also shows key methods to be associated with each class. The concept of a "domain" model is that we do not want to see all of the classes in the actual implementation. Beginning students and some UML tools "reverse engineer" actual code to produce a class diagram, but that is not a domain model

and it is usually too complex to be useful. Many UML tools support automatic generation of program code framework from a class model.

3. State model: This includes an overall state model, as in item 5 below, and models for the state behavior for individual classes, as Figure 3. The state chart notation shows states, transitions, conditions, and actions. It may show nested sub-states. Some UML tools can produce executable state models that can be used for simulation and testing of program behaviors.

4. Sequence model: This consists of a collection of sequence diagrams, as Figure 4. Each sequence shows one trace of behavior (trace of execution) . Generally it is necessary to show several traces of execution to adequately describe the desired software. There is developing work on tools to check the consistence of sequence models with respect to each other and with respect to the class and state models. Collaboration diagrams (not shown here) present an alternate view of the interactions between difference sequence behaviors.

5. GUI (Graphic User Interface) prototype: In a strict view, a GUI model is not a UML component, but it is an essential component of design. Such a prototype may start as sketches of user interface screens (or interface states for interactions with non-human agents). The sketches are then composed to form a state model of user interaction, as Figure 5. Finally, the state model might be implemented as an interactive animation of screen "snapshots" (actually mock-up screens) by which users can assess the function and usability of software before it is built. Such prototypes generally do not actually compute anything, but rather they just present changing screens to represent common scenarios. Such prototypes are quite easy to construct using Integrated Develop Environment tools that have a GUI-builder.

There are several more kinds of UML model components, but they are more specialized than the ones listed here. There are UML tools to create machine drawing for the UML model diagrams. Yet, quite adequate design can be done with just paper and pencil sketches.

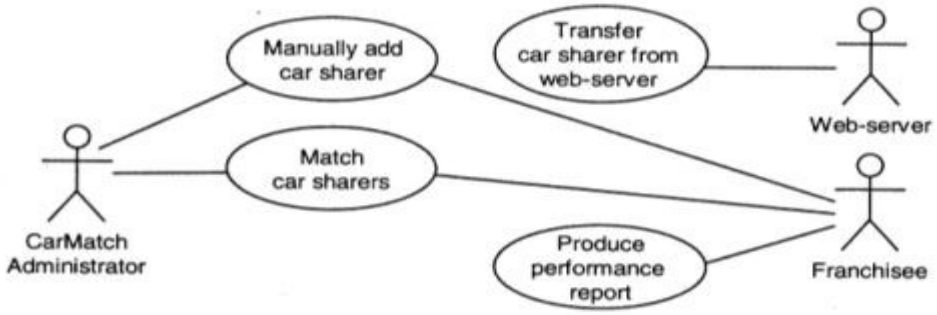

Figure 1. Use – Case Model

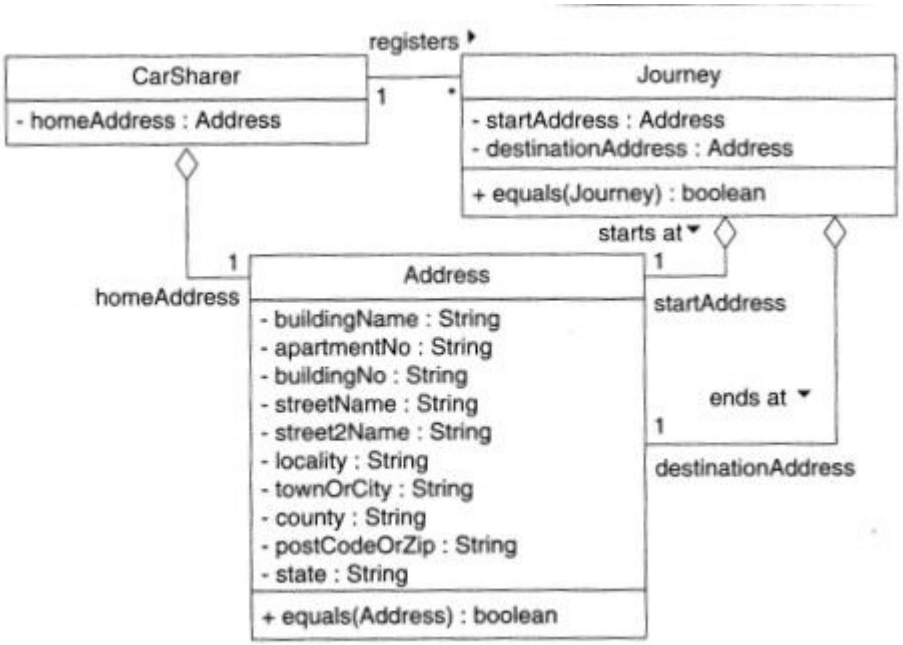

Figure 2. Class Domain Model

While UML and prototype models are the vehicle of design, that vehicle does not become meaningful for students until they can work with and understand examples to which they can relate. Many of the design books cover all of the modeling concepts (and design patterns), but very few engineering examples. One text that was rich in examples (but somewhat old in its notations) was the Object Models text by Coad [5] . That text included examples of a Point-of-Sales application, a Warehouse-Management system, an Orders-Management system for the warehouse system, a Conveyer-belt-control system for a factory, and an Auto-Pilot control for a simple aircraft. In my course [6] , in various semesters I have presented a simulation environment for robots, a network message center, a UML sequence builder, a speed control for an auto, a multi-person card game, a 20-questions quiz game, and others. For a service course for engineers, more engineering applications should be included. Please contact me to suggest engineering applications to be modeled.

Design Experience

Once a model, even a partial model is presented, students can learn about the model in various ways. They can answer factual questions about the model (and those are easy to grade). They can extend the model, such as by providing new sequence models or by adding new features to the model. However, such exercises do not capture the essence of the design task. That key process is to be able to analyze a weakly-defined description of a target application and to develop the use-case and the class models. Likely, the key to finding a class model is in part careful analysis of the description of the domain and in part being able to draw upon patterns, that is generalized models, of similar kinds of applications or components. Hence, a good portion of the design class is spent first, in viewing architectures for common kinds of applications and second, doing exercises building and reviewing class models. Those exercises lead up to the course project, which consists of the full process of building a model, presenting it

to the instructor for review and revisions, and then implementing a core part of the features and user interface. Of course, the program must match the model.

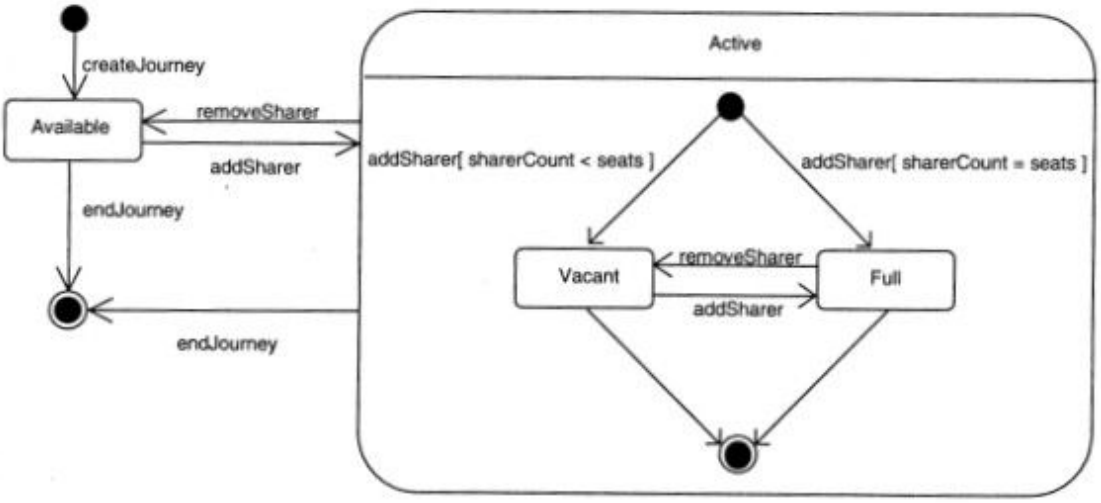

Figure 3. State Model

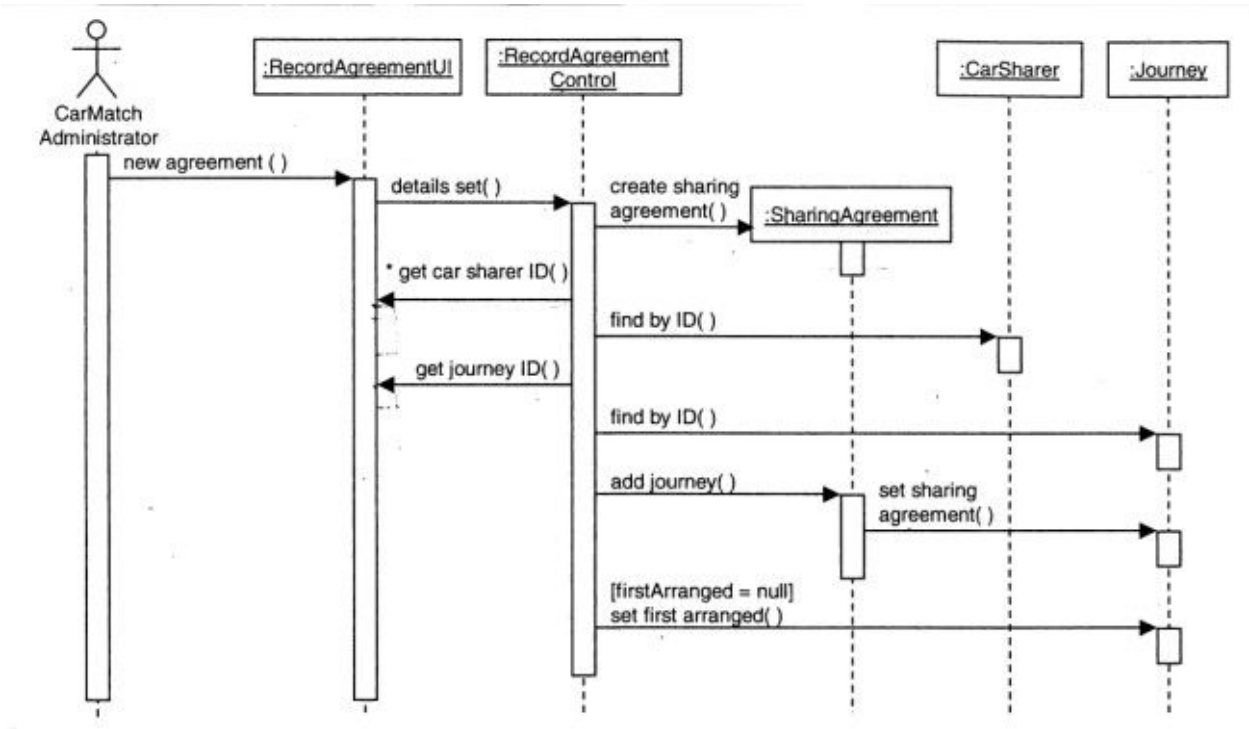

Figure 4. Sequence Model

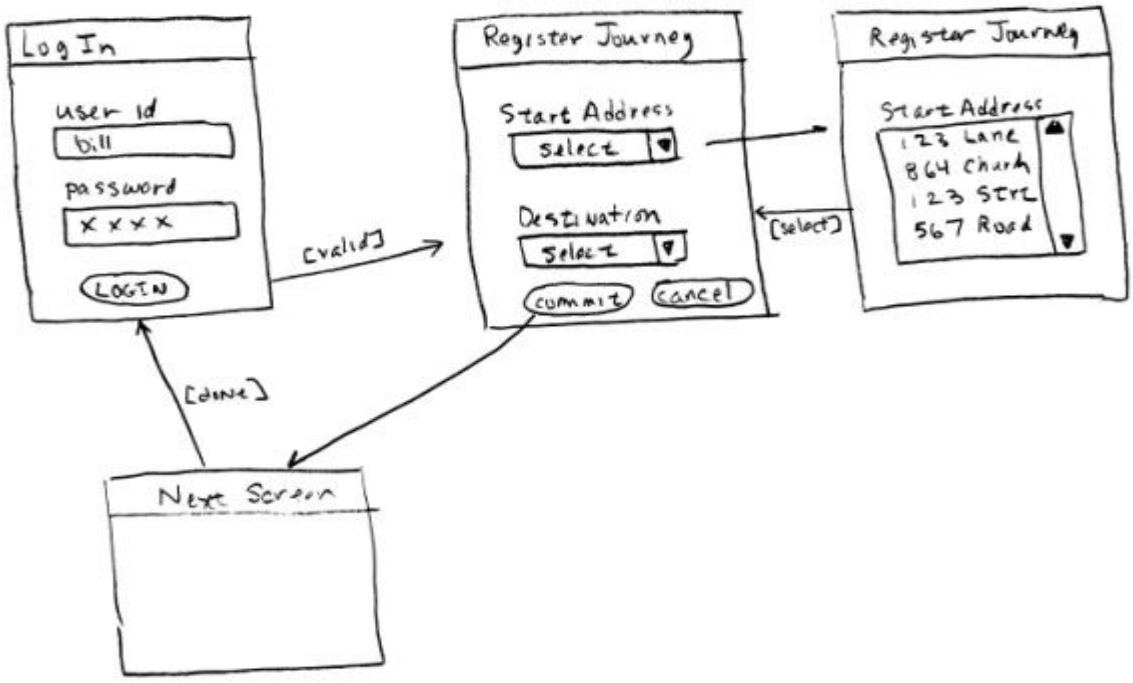

Figure 5. States for a GUI Model

Conclusion

The suggested second course on software design can provide a design experience for engineers within the context of software development. We hope that more engineers will recognize the difference between such design experience vs. just programming experience.

References

[1] Computing Curricula 2001, Chapter 7 Introductory Courses, http://www.computer.org/education/cc2001/final/chapter07.htm

[2] Java Collections Framework, 2002, http://java.sun.com/j2se/1.4.2/docs/guide/collections/

[3] Milwaukee School of Engineering, Bachelor of Software Engineering, http://www.msoe.edu/eecs/se/

[4] S. Bennett, J. Skelton, K. Lunn, UML, Schaum's Outlines, McGraw Hill, 2001.

[5] P. Coad, Object Models: Strategies, Patterns, & Applications, Prentice Hall, 1997.

[6] W. Hankley, "On Teaching Software Architecture and Design, 2003 ASEE Conference, paper 2494 on Conference CD.

[7] A. Cooper, The Inmates are Running the Asylum", Sams Publishing, 2004, pp255, Part 4 on Interaction design.

Biography

WILLIAM HANKLEY is professor of Computing and Information Sciences at Kansas State University. He received his Ph.D. from the Ohio State University. He developed and teaches a course on software architecture and design within the KSU computing curricula.