# EXPERIENCE WITH TEAM PROJECTS IN A SECOND-SEMESTER C++ PROGRAMMING COURSE

DAVID CASPERSON

ABSTRACT. Experience using project assignments in a second semester C++/programming course are discussed. These projects are similar to the elevator project found in "*C++ How to Program*" by Deitel and Deitel editions 1 and 2 ([2, 3]).

## 1. INTRODUCTION

How do students learn to program? In particular, how do students learn to think in the object-oriented paradigm? Part of learning to program consists of learning grammar and syntax, and how to translate ideas into a formal programming language. An equally important part of learning how to program is learning how to decompose a problem into smaller pieces, which in turns relies on learning how to abstract those smaller pieces into simple descriptions. Object-oriented design is a particular method of problem decomposition that emphasizes the use of problem language terminology and entities to aid the decomposition process. Supposing that students can learn the mechanical and formal aspects of programming through lectures and laboratory exercises, how can we best help them learn to engage in object-oriented design and programming?

One question that needs to be addressed in asking how students learn to apply object-oriented design is the question of scale. By second semester many students are capable of applying stepwise refinement to small problems without conscious thought. In order to appreciate the role of design and good programming practices students need to tackle larger problems. This paper reflects my experience using medium-sized projects such as the elevator project in *C++: How to program* by Deitel and Deitel [4, 3, 2, 1] to introduce second-semester programming students to object-oriented design and object-oriented programming.

CPSC 101, *Introduction to Programming II*, is a four-credit second-semester Computer Science programming course at the University of Northern British Columbia (UNBC). In section 2, I describe in more detail how CPSC 101 fits into the UNBC programming sequence, what

FIGURE 1. The UNBC programming course sequence

| | | *Semester* | |
|---|---|---|---|
| | | **Fall** | **Winter** |
| *Year* | **I** | CPSC 100–4 *Computer Programming I* | CPSC 101–4 *Computer Programming II* |
| | **II** | CPSC 200–3 *Data Structures and Algorithm Analysis* | CPSC 281–3 *Data Structures I* |

material it covers, and how the course is structured. Then, in section 3, I narrow my focus to the team projects and discuss the mechanics of the team projects.

Having outlined the mechanics of the CPSC 101 team projects, I then try to evaluate what benefits the students gain from them. In section 4, I present some of the possible benefits to students. In section 5, I present some of the things that can go wrong, and, in section 6, I summarize what I see as the successes of the team projects.

The CPSC 101 team projects have evolved slowly over the years. I have taught or co-taught the second-semester programming course, CPSC 101 *Introduction to Programming II*, at the University of Northern British Columbia (UNBC) every year since 1996 (excepting 2002). In 1996, Dr. Waqar Haque, who co-taught the course with me, introduced the elevator project as part of the course. In 2002, this course was taught by Dr. Liang Chen, who also used a team project. (Table 5 in Appendix A gives a complete history of the projects used.) In section 7, I share some of the teaching strategies that I have found to be effective after several years of using team term projects.

It is my hope that other instructors will incorporate similar team projects into their institutions programming sequence and share their teaching experiences. In section 8 I briefly suggest some guidelines for designing problem statements to use with team projects. Problems statements that have been used at UNBC in CPSC 101 are included in Appendix A.

## 2. The UNBC CPSC 101 course

The projects described in this paper are part of the University of Northern British Columbia second-semester first-year four-credit Computer Science course CPSC 101 *Introduction to Programming II.* In this

Table 1. *CPSC 101 Student Contact Hours Per Week*

| *Context* | *Hours* | *Comments* |
|---|---|---|
| Lab | 1×1.5h | (with undergraduate lab assistant). Maximum of 14 students. |
| Tutorial | 1×1.5h | Normally taught by faculty. |
| Lecture | 3×1.0h | One section |

Table 2. *The CPSC 101 Course Mark Breakdown*

| | |
|---|---|
| 15% | Laboratory assignments (approximately weekly, in 2003 there were 8 assignments). Although each student has a scheduled 1.5h lab each week, much of the work is done outside of the lab. |
| 15% | Midterm I |
| 15% | Midterm II |
| 5% | quizzes |
| 15% | the term-long team project discussed in this paper. |
| 35% | Final exam |

section I describe this course and its relation to the rest of the programming course sequence at UNBC in order to give readers an understanding of the context of the course projects.

This course is the second course in the four course programming sequence at UNBC. The first and second year programming sequence at UNBC is extensive, and is shown in Figure 1 (see also [5]). This course normally runs for thirteen weeks. Each week students have three one-hour lectures, one 1.5 hour lab session, and one 1.5 hour tutorial (see Table 1). The course mark breakdown for `CPSC 101` is shown in Table 2.

In the previous programming course, `CPSC 100`, *Computer Programming I*, students learn basic imperative programming, functional decomposition, and the concepts of objects and classes (including inheritance, but not including virtual functions). Consequently, at the beginning of `CPSC 101` students are for the most part comfortable with writing imperative programs involving several functions, and know how to define a class and create objects.

On the other hand, they have not yet seen operator overloading, static member variables or static member functions, or non-trivial destructors. They have been briefly introduced to the use of `new` and `delete`, but are unlikely to understand the difference between dangling pointers and memory leaks. They know how to divide programs into multiple source files, but tend to associate this with programming with classes and objects (as the concepts are introduced at roughly the same time in the `CPSC 100` curriculum).

Some of the topics that `CPSC 101` students learn about are:

(1) a better understanding of the compiler, including the distinction between pre-processing, compilation proper, and linking;
(2) `make`-files (strictly as a lab exercise);
(3) a memory model appropriate for C++ programs (the role of the stack, the heap, code memory, and global static memory);
(4) how to draw memory diagrams showing the interaction of member functions, constructors, destructors;
(5) a detailed understanding of constructors and destructors and their role in an object's lifetime;
(6) static members;
(7) class invariants (briefly!);
(8) friend functions;
(9) a detailed discussion of pointers and pointer arithmetic and dynamic memory allocation;
(10) operator overloading;
(11) inheritance (with multiple inheritance mentioned, but not emphasized);
(12) run-time polymorphism (virtual functions, pure virtual functions, abstract classes);
(13) some Standard (Template) Library classes such as `string`s, `vector`s, and `deque`s.

The course covers material from Chapters 5–10 of [3]. Students are already familiar with formatted input and output, and are referred to Chapters 11 and 14 for additional material. The text [3] does not discuss memory diagrams for C++ programs (although it uses them to describe how virtual functions are implemented in Chapter 10). In discussing how C++ compilers work and the role of makefiles I have also gone beyond the material found in [3].

In short, `CPSC 101` is an intensive course on elementary C++ and object-oriented programming. Because it is part of a very long lower level programming sequence, it can afford to be much more in depth than many second-semester programming courses.

## 3. Mechanics of the CPSC 101 team project

In this section I discuss what the team projects entail. Team projects are completed over the course of a thirteen-week semester. At the same time that students are working on the project they are also

Table 3. *The CPSC 101 Project Mark Breakdown*

| Component | Percentage |
|---|---|
| Design | 30% |
| Implementation | 50% |
| Review | 10% |
| Response to review | 10% |

Table 4. *The CPSC 101 Project Time line*

| Week | Milestone |
|---|---|
| 2 | Student receive problem statement and list of team-members. |
| 5 | Design document due. |
| 6 | Designs reviewed. Students notified of serious design flaws |
| X | Reading week.[1] |
| 9 | Implementation completed. |
| 11 | Review of other team due. |
| 13 | Response to review due. |

completing weekly programming assignments. Students work on projects entirely on their own time, and it is the students' responsibility to schedule team meetings and find time to work on the projects.

As part of their projects students must: (a) complete a design document, (b) implement their design, (c) present their working program to the rest of the class, (d) review another team's project, and (e) respond to the review of their own project. The project mark is broken down as shown in Table 3. The time line for the project in a typical winter semester is shown in Table 4.

3.1. **Team Selection.** Students begin the team project in the first or second week of classes, at which time the composition of the project teams is announced and the students receive a handout describing the project. Teams normally consist of four students, although for logistical reasons there may be teams with either three or five students. See section 7.1 for a discussion of how teams are selected.

[1]UNBC students have a one-week break from classes at the end of February. Week numbers refer to weeks of classes, so the ninth week refers to the tenth week since the beginning of term.

3.2. **Design.** Students design documents are due at the end of the fifth week of classes. Design documents must include:

- a list of nouns,
- precisely worded paragraphs describing each noun,
- a list of facts,
- a list (by class) of attributes, behaviours, and collaborations with other objects,
- a cover page,
- table of contents,
- a percentage of work completed by each member of the group, and
- a proposed distribution of workload for the implementation part of the project.

Students are encouraged to create their design documents by following the following steps.

(1) identifying significant nouns in the problem statement,
(2) identifying facts in the problem statement,
(3) grouping facts by noun,
(4) determining attributes and behaviours from the list of facts, and then
(5) determining collaborations between objects.

These steps are taken from the earlier editions of *C++ How to Program* ([1, 2]), which use a CRC (Classes, Responsibilities and Collaborations) approach to object-oriented design. Later editions ([3, 4]) present object-oriented design through a case study rather than a project, and use UML to create a design model.

3.3. **Design Review.** I grade and review the design documents and return them within a week. Teams whose designs appear to be dangerously incomplete are either encouraged or required to re-submit their design before proceeding to implement their project.

3.4. **Implementation.** Students must complete implementation of the project in the ninth week of classes. Each team must hand in

- a revised design document,
- complete listings of their programs,
- a short users' guide,
- an assessment of the contributions of each member, and
- output for three representative simulations showing how to use their program.

3.5. **Presentations.** Immediately after project implementations are completed, students give brief demonstrations of their projects to the other students in the course. They then give a copy of their users' manual to another team (selected by me) who reviews their project.

It is my experience that about eight teams can give demonstrations in during a one-hour lecture slot. Because it is nearly impossible to find other common times in the students, the number of teams is limited by the number of lecture slots that I am willing to sacrefice.

Before Winter 2003 student team demonstrations were given in the Lower Level Laboratory. This year, demonstrations were given in the course lecture room using a laptop linked to the University network, and a large overhead projector. One consequence of this shift was that students paid far more attention to demonstrations. Another consequence was that demonstrators tended to take much longer, and I was only able to get through about 4 demonstrations per lecture slot.

3.6. **Reviews.** Each team reviews another teams project and produces a report. The reviewing team has access to the users' manual and executable programs of the team that it is reviewing. However, *there is no code review.* Reviewers test programs thoroughly for conformity to the problem specifications, bugs, and usability.

Reviews are due approximately one week after the program demonstrations. One copy of the review document is given to me and another copy to the team being reviewed.

3.7. **Responses.** Each team responds in writing to its review. In their response, teams are asked to comment on how easy it would be to modify their project in order to accommodate requests or criticisms of the reviewing team. (Part of the purpose of this exercise is to have the teams look at how their design decisions have resulted in either flexible or inflexible code.)

The projects are thus a semester-long activity that happens almost entirely outside the class-room. The most intense student activity happens during the ten days preceding the project implementation due date, and then tapers off towards the end of the semester.

## 4. Outcomes

In this section I discuss what I expect students to gain from the term projects. CPSC 101 students are already quite busy with weekly labs, tutorial sections, lectures—as well as other course work, which likely includes second semester calculus, second semester discrete mathematics, and second semester physics—so it is important to be able to justify team projects by understanding what benefits students gain through participating in such a project. Here are some of the expected benefits:

- an appreciation for the need for some kind of methodical design activity before tackling a larger project. Students notice that good designs lead to good teamwork in implementation and rapid debugging, and that, conversely, bad designs result in a lot of code rewriting and perverse bugs.

  By contrast, students rarely need to engage in careful design work in order to complete weekly programming assignments. Even when students do waste time through poor design, they are unable to separate time wasted due to poor design from time wasted debugging, or wasted because of poor understanding of the course material on which the assignment is based.
- a practical demonstration that object-oriented design is a useful technique in decomposing a problem.

  Again, weekly laboratory exercises tend to emphasize particular skills, such as managing dynamically allocated storage or programming with static member variables, rather than focussing on object-oriented design and problem decomposition through selection of classes. Often the decomposition is already given to the students (*e.g.,* "write a large `Integer`-class with overloaded arithmetic operators").
- experience working with medium-scale software projects. This is incredibly important in driving home the benefit of good programming practices. Poor programming practices are often "good enough" for laboratory assignments, but do not scale well.
- experience working collaboratively. Collaborative work is important in emphasizing the need for good programming practices, as the benefits of good programming practices are more pronounced when working with others.
- improved technical writing ability, and a better understanding of the need for good technical writing ability in Computer Science.
- a sense of responsibility and professionalism.
- awareness of the importance of good teamwork in computer programming.
- greater confidence in their ability to tackle complex problems.
- practice meeting implementation deadlines.

## 5. Difficulties

In this section I discuss obstacles to overcome in using term projects such as the ones discussed in this paper. The single greatest reason that students fail to realize the benefits discussed above are bad intra-team interpersonal relations. Other obstacles that are less important to student success include: poor coding resulting from weak designs or incomplete understanding of object-oriented programming, and improper or inappropriate written documentation. I shall discuss these briefly in section 7 and dedicate the rest of this section to discussing problems relating to team dynamics.

Occasionally teams implode and team members end up fighting among themselves. Sometimes one team will produce two or more implementations of the project, or of pieces of the project. I haven't experienced this in recent years, but this may be more the result of statistical fluctuations than of any particular teaching strategy. Causes of such implosions seem to be the combination of multiple strong personalities on a team, and an incomplete or misunderstood design document that leads to code integration difficulties late in the project.

Another possible source of intra-team relationship problems are the occasional very very strong students that enroll in CPSC 101 (for instance, senior students who have suddenly become concerned that first-year $C^-$'s may be harming their chances of getting into graduate school, or professional programmers who are looking to upgrade their C++ skills or academic credentials). Without instructor intervention such students may completely dominate their group, making the project a miserable experience for other students in the group.

Here, one successful intervention strategy is to allow (or sometimes insist) that students who retake the course become a one-person project team. Another strategy is to carefully scrutinize the proposed division of labour submitted with the design documentation and use one's moral authority as instructor to insist that the interesting work is split more or less evenly among team members. A third strategy is to speak to very strong students early and emphasize that their grade depends on their ability to mentor the weaker students in their group. All of these strategies rely on having an early accurate assessment of the strengths and personalities of the students in the class, so may not be appropriate to large institutions or class sizes.

Exceptionally weak teams are also a problem. Sometimes all of the strong members of a team drop the course or get late co-op placements, creating a team with only one or two weak students. Surprisingly, such teams very frequently complete a working implementation of the project, thus gaining at least some of the expected benefits outlined in section 4.

However, although they produce code that compiles and runs, it is very brittle and only shows a vague relation to good programming concepts and almost no relation to object-oriented design and programming. Because their code is poor, these students only manage to produce a working project through the allocation of excessive amounts of time to debugging, to the detriment of their performance in the rest of the course. I have yet to find successful interventions that work with such a team.

## 6. Successes of the projects

Despite all of the difficulties outlined in the previous section, and despite the large effort required by both students and instructors to make such projects worthwhile, I consider them to be a success. Here is why.

Firstly, most students create functioning projects more-or-less within the time limits specified,[2] and that do more-or-less what was asked for. I have the sense that students are often quite surprised by their success and proud of their projects.

Secondly, in the vast majority of the cases, the project teams become cohesive units and take responsibility for and ownership of their projects. Frequently teams report that their members participated equally in the project, even when it is clear to both the team and me that this is a fiction.

Furthermore, most of these projects make fundamental use of object-oriented programming in their organization and coding. Because polymorphism isn't discussed in class until late in the implementation of the projects, it might be truer to characterize the project design and coding as object-based, rather than object-oriented. Nevertheless, most projects benefit significantly from the use of objects and classes.

It is also the case that both strong and weak teams discover that most implementation problems stem directly from weaknesses in the design. Even though the students have not had any exposure to formal software engineering, they see formal design as a practical need rather than a vague moral commandment.

[2]Project deadlines several weeks before the end of the semester help create a sense of success, because then even late projects are "successful."

Finally, some project teams produce projects that are significantly better than the sum of the efforts of the individual members would have been. Although these project teams are a minority, such successes must be included in the overall evaluation of the success of team projects.

## 7. Teaching Strategies

In this section I list in no particular order strategies that I have found to help in producing successful team projects.

7.1. **Team Selection.** Team selection is quite important. In order to equalize the strengths of of the students, teams are selected based on course marks in first-semester computer science. Teams normally consist of four members, although for logistical reasons there may be some teams with either three or five students. It seems that three is the ideal size for teams working on projects of this scope. However, starting with teams of three is dangerous as there are always students who withdraw or stop participating after the teams are selected.

7.2. **Design Review.** My experience suggests that a detailed and rapid design review by the instructor before implementation begins is an important component in ensuring a good project success rate. Bad designs cause intra-team friction when it becomes apparent that there were differing perceptions of what the design meant. Bad designs also often results in bad programming practices as students use global variables, public member variables, inappropriate friendship and the like to try to patch together their implementations.

Common problems with student designs include:

- an inability to perceive potential objects and classes, usually due to premature coding. For instance, students frequently complain that they fail to see the point of coding elevator buttons as a class "when a boolean variable would suffice."
- failing to make collaborations and behaviours correspond.
- vagueness in naming. Students often give a check-out lane in a grocery store an attribute called "`status`" or a behaviour called "`setStatus`" rather than something more specific like "`isStaffed`" or "`isOpen`".
- assuming that attributes and private member variables are identical concepts.

However, the biggest problem for later implementation is a lack of attributes for various objects (for instance, people in an elevator simulator that have no notion of their own physical location), leading to the *ad hoc* use of the bad programming practices mentioned above in order to get the various objects to be able to find one another in order to communicate.

Being aware of the above problems, I find that it is easy to detect weak designs. Meeting and talking with teams with weak designs seems to be the best strategy for getting them on the right track. To encourage good designs in the first place it is important to stress the need to use problem domain language, and to emphasize that it is far easier to remove parts of a design should it be discovered that they are not needed than it is to add them after the fact.

7.3. **Team Dynamics.** In order to develop the students sense of responsibility and professionalism and to give the students the maximum benefit of working collaboratively, I try to leave project decisions to the students and intervene as little as possible. In particular, I emphasize to the students that design and implementation decisions — including how to interpret the project specifications — are the responsibility of each team. I also leave team organization and management up to the teams, and tell the students up front that I will not intervene to solve intra-team problems, unless they become grossly dysfunctional.

To further this sense of responsibility, I require that the teams report to me:

- the proposed distribution of workload for the implementation part of the project in the design phase, and
- an assessment of the contributions of each member with the completed implementation.

Except where I have evidence that to do so would be grossly unfair, I accept the student assessments at face value.

7.4. **Laboratory resources and tools.** The greatest single technical need that teams face is laboratory space where team members can work together in the integration and debugging phases of implementation. Many students now have their own computers and off-campus internet access which helps relieve the day-to-day need for computer resources on campus. Face-to-face interaction is still the most successful way to engage in project integration, so it is important not to underestimate the laboratory needs of students for this phase.

7.4.1. *CVS.* In 2003, I experimented with providing students with a limited introduction to CVS (Concurrent Versioning System), a source-code control system to help with intra-team communication. The experiment was a mixed success. Some teams chose to use CVS. Others did not see the need, and used shared directories, e-mail and the like in order to communicate. There is a strong desire by both upper-level students and some of UNBC's co-op employers to give students exposure to CVS early in the programming sequence. While there are definite benefits to doing so, teaching students enough to be comfortable with the technology takes time from other activities.

7.4.2. *Makefiles.* Makefiles are introduced in the first lab assignment in CPSC 101, and most students incorporated them into their team projects although there was no explicit requirement to do so.

7.4.3. *UML.* This year, some students were fascinated by the UML diagrams in the third edition of Deitel and Deitel and attempted to use UML diagrams in their own design documents. In general, they did not understand the various purposes of the different kinds of diagrams. Even when they did, they often used the UML inappropriately. More generally I find that use of UML encourages students to engage in prematurely detailed activity. I would suggest that students ought to be actively discouraged from using UML until they have had a formal introduction to system design.

7.5. **Formal Documents, Writing Skills, and Professionalism.** I find that students do a poor job of writing formal documents unless they are given explicit checklists to guide them. Things that I find useful to tell students include:

- include a cover page
- number your pages
- put a date on every document
- put the team number as well as the team member names on each document
- always provide an introduction and a conclusion

Repeatedly stressing that the reports are evaluated for these elements vastly improves the quality of the reports that the students produce.

Despite the fact that students tend to write fair reviews of each others program, many students take the negative elements of reviews personally. It takes a major effort to encourage students to write responses that are neither sarcastic nor derisive. Informing students that such remarks result in an automatic loss of 20% seems to be effective. Merely characterising such responses as grossly unprofessional does not.

Ensuring good team dynamics is the most important part of having successful team projects. Detailed design review helps create good team dynamics.

## 8. Selection of Projects Topics

Students find object-oriented design activities, such as determing what nouns are involved in the problem statement, or what attributes are associated with an object, quite difficult. Consequently the choice of subject matter for the project is quite important. In this section, I give some principles that seem to help in the selection of project material.

Firstly, choose to simulate something that has lots of easily identifiable physical parts where the parts lie in the domain of the students' everyday experience. The project topics that I have used successfully are the Deitel and Deitel elevator simulation, a vending machine simulation, and a grocery store simulation.

Secondly, choose to simulate something without complex concurrency in the problem domain. Students find the elevator project harder than the other projects precisely because of the need to model different people simultaneously interacting with one elevator. Because grocery store customers enter line-ups and then interact directly with one teller who is interacting with no other customers the students find this much easier to simulate. However, the students have concerns about how to model time and multiple active customers even in this project.

Thirdly, keep the problem as simple as possible. Good students always find things to add to make their project "more realistic". It is more important that the project not overwhelm weaker students. Even good students have doubts at the beginning of the project. One of the benefits of the grocery store project as written is that teams can easily divide it into pieces and give some of the easier pieces to the weaker students.

## 9. Conclusions

A term-long team-project provides second-semester students with valuable experience in designing and implementing medium-sized projects. Most teams are successful in creating working programs that utilize object-oriented programming concepts appropriately. These projects simultaneously bolster the students confidence in their ability to tackle complex projects, and provide them with motivation for taking design seriously, for using object-oriented programming, and for learning how to co-operate with fellow students. For these reasons, it is worth making the effort to incorporate such projects in early programming courses, even if it means sacreficing other material and assignments.

Table 5. *Project Topic History*

| Year | Instructor(s) | Topic |
|---|---|---|
| 1996 | Haque and Casperson | Elevator Simulation |
| 1997 | Casperson | Elevator Simulation |
| 1998 | Casperson | Vending Machine Simulation |
| 1999 | Casperson | Elevator Simulation |
| 2000 | Casperson | Grocery Store Simulation |
| 2001 | Casperson | Grocery Store Simulation |
| 2002 | Chen | Vending Machine Simulation |
| 2003 | Casperson | Grocery Store Simulation |

I strongly encourage other instructors to undertake such projects, and to inform us of their challenges and successes.

## References

[1] H. M. Deitel and P. J. Deitel. *C++ How to Program.* Prentice Hall, first edition, 1994.

[2] H. M. Deitel and P. J. Deitel. *C++ How to Program: Starring the Standard Template Library.* Prentice Hall, second edition, 1998. This edition first introduces the STL.

[3] H. M. Deitel and P. J. Deitel. *C++ How to Program: Introducing Object-Oriented Design with the UML$^{TM}$.* Prentice Hall, third edition, 2001. This edition turns the elevator project into a case study using the UML.

[4] H. M. Deitel and P. J. Deitel. *C++ How to Program: Introducing Web Programming with CGI and Object-Oriented Design with the UML$^{TM}$.* Prentice Hall, fourth edition, 2003.

[5] Office of the Registrar. UNBC undergraduate calendar. University of Northern British Columbia, 3333 University Way, Prince George, BC Canada V2N 4Z9, 2002. See also `http://www.unbc.ca/calendar/`.

## Appendix A. Project descriptions

Table 5 lists the history of projects used for CPSC 101 at UNBC.

A.1. **The Deitel & Deitel elevator project.** Every edition of Deitel and Deitel's *C++ How to Program* presents non-object oriented programming first in Chapters 1 through 5 before switching to elementary class concepts in Chapter 6. In order to prepare students for this shift in paradigm Chapters 2 through 7 end with a section entitled "Thinking about Objects". In these sections, they present the problem of simulating an elevator to determine if it is adequate to meet a building's needs.

The first two editions ([1, 2]. of *C++ How to Program* presented the elevator simulation problem as a student project. In these editions

Chapters 2 through 5 contained exercises to help the students create a design for the elevator simulator, and Chapter 6 then asked them to implement it.

Later editions ([3, 4]) have converted the project into a case study where Chapters 2 through 5 show how to design the elevator project and later chapters give a complete working implementation. In addition, the authors have incorporated UML into their discussion of object-oriented design.

Whilst this provides the students with a detailed example of a carefully designed and coded medium-sized project, it means that the elevator simulation problem is no longer an option as a student project. It also means that there is no longer an explicit list of design activities to which instructors can refer their students.

A.2. **Vending machines.** Here is the specification given to the students for the vending machine project.

Problem Statement

A company is designing a new vending machine. The company wants you to develop an object-oriented software simulator so that they can see whether the machine that they are developing will meet the customer's and service people's needs.

The company's vending machine is similar to many of the candy vending machines at UNBC, but smaller. There are ten rows and eight columns of spring coils that can hold chips, candy, and other merchandise. Each spring coil can hold up to 10 items.

The machine operates in two modes, depending on whether or not the front door is open. When the door is shut it operates in vending mode. Customers may enter 5¢, 10¢, 25¢, \$1.00, and \$2.00 coins[3]. They may also press buttons labelled "A" through "J" to select the row, and "1" through "8" to select the column of the item they wish to purchase. They may also push the coin return button at any time.

In vending mode, the machine normally responds to coin entries by displaying the current total entered on a small display. When the machine's customers push one of the buttons their choice is displayed instead. When a customer completes a choice one of three things happens. Either the machine dispenses the requested item and releases the customer's change through the coin return; or it displays a message saying that the requested selection isn't available; or it displays a message saying that the customer hasn't entered enough money to purchase the item. When a customer presses the

[3]the vending machine rejects 50¢ coins

coin return button any unspent money in the machine is returned through the coin return.

When the door is open, the machine operates in restock mode and none of the normal vending functions are available. In this mode it is possible:

- to set the price for each of the spring coils;
- to refill or partially refill each of the spring coils;
- to empty the cash box;
- to refill or partially refill the change box; and
- to close the door.

The coin-handling mechanism of the vending machine consists of a coin entry slot; a coin return slot; a cash box; and a change box that has a column for each kind of coin. When a coin is put in the entry slot the machine can control whether it goes back out the coin return slot or gets dropped into the cash box. The machine can also direct the change box to drop a coin from a particular column into the coin return slot. Note that once coins enter the cash box the machine cannot move them elsewhere. When the change box is close to empty the machine can display a message asking the user to enter exact change only.

Your simulation should allow the simulation user either to directly control the events that happen or enter an automatic mode where customers and machine restockers arrive randomly and use/restock the machine. It should be able to help answer questions such as how much change needs to be in the change box, so that most of the time the machine runs out of stock before it runs out of change. In order to do this your simulation needs to keeps statistics as it runs and print them out on request, and needs to use random number generators to simulate customer behaviour.

END OF PROBLEM STATEMENT

A.3. **Grocery store staffing.** Here is the specification given to the students for the grocery store simulation project.

PROBLEM STATEMENT

A grocery store company is revising its cashier and check-out staffing. The company wants you to develop an object-oriented software simulator so that they can see whether the new policy that they are developing will meet the customer's and store's needs.

The store has eight checkout lanes, one of which is always marked "Express: nine items or less" and one of which is always marked "Express: nineteen items or less". Not all of the checkout lanes are always staffed.

The time taken for a cashier to complete a transaction with a customer depends on

- the number of items that the customer has,
- whether the customer is paying by (a) cash, (b) cheque, or (c) debit card, and
- a small random factor.

To begin with, assume that it takes a cashier 5s per item, and that it takes a customer 1 minute to pay by cash, 2 minutes to pay by debit card, and 2.5 minutes to pay by cheque.

The time taken for a customer to pass through a checkout lane also depends on how busy the checkout registers are when the customer decides to enter a lane, and the time to process the customers ahead in the lane.

The company wants to be able to run the same pattern of customers through various different cashier configurations to see what configuration works best, so they want three separate programs.

The first program creates a file specifying the customers for a given simulation. The file should list: the customers in the order of their time of arrival at the checkouts, the number of items that the customer intends to buy, and the method of payment that the customer intends to use.

The second program creates a file specifying the configuration of cashiers for a given simulation. This file should list when a cashier comes on duty, and when the cashier goes off duty. For simplicity, assume that there are a fixed number of cashiers available at any given time, and that they are all tending tills.

The third program runs a simulation by reading a customer file and cashier file and simulating and timing the interactions. At a minimum, the third program should measure how often cashiers are idle and how often customers must wait a long time to be served.

Here are some questions you might want to think about:

(1) How do you decide when the express check-out lanes are staffed?
(2) Can you use a debit card on either of the express lanes?
(3) How does a customer choose which lane to stand in?
(4) What happens if there is a lineup when cashiers are at the end of their shift?
(5) How long does it take for cashiers to switch at a till?

END OF PROBLEM STATEMENT

DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF NORTHERN BRITISH COLUMBIA, 3333 UNIVERSITY WAY, PRINCE GEORGE, BC, V2N 4Z9

*E-mail address*: `casper@unbc.ca`