Chen, W. et al. (Eds.) (2016). Proceedings of the 24th International Conference on Computers in Education. India: Asia-Pacific Society for Computers in Education

Using a Text Adventure Game for an Extended Series of Assignments in CS2

Clifton KUSSMAUL
Muhlenberg College, USA
kussmaul@muhlenberg.edu

Abstract: Students in computer science and related disciplines need experience with larger software projects. This paper describes a sequence of assignments for CS2 that guide students to develop a text-based adventure game. These assignments seek to increase motivation, provide context for core topics, and explore some of the challenges associated with larger projects. This paper also describes forms of support and scaffolding to make such projects more effective, and describes lessons learned for faculty seeking to adapt or create their own extended assignments.

Keywords: CS2, iterative development, object-oriented design, refactoring, text game

1. Introduction

Students in computer science and related disciplines need experience with software projects that are larger than a typical homework, even in introductory courses (e.g. Rebelsky and Flynt, 2000; Turner and Zachary, 1999). A variety of approaches have been proposed, e.g. for graphics (Bruce, Danyluk, and Murtagh, 2001; Kussmaul, 2008), object-oriented design (Alphonce and Ventura, 2002; Barry, Ellsworth, Kurtz, and Wilkes, 2005), data structures and algorithms (Huggins, 2000; Newhall and Meeden, 2002), and teamwork (Rebelsky and Flynt, 2000).

This paper describes a sequence of CS2 assignments to develop a text-based adventure game. Text-based games have a long history, from the early *Colossal Caves* (Crowther, Woods, and Black, 1976), through the *Choose Your Own Adventure* books and commercial games (e.g. from *InfoCom*), to scholarship on interactive fiction (Montfort, 2005), and popular press (Hudson, 2014).

Clearly, text games are less flashy and immersive, but require less programming experience, depend less on an API, and can be adapted to a wide range of interests, whereas graphics games may not appeal to all students. These assignments seek to: a) increase motivation through a more project that students can adapt to their own interests and that it more complex than they could write on their own; b) provide context and incentives for core topics in CS2; and c) explore some of the challenges associated with larger projects, including testing, refactoring, and iterative design. This paper also describes forms of support and scaffolding to increase the effectiveness of such projects, and describes lessons learned for faculty seeking to adapt or create their own extended assignments.

These approaches offer a variety of benefits. Students are exposed to issues and techniques which may not be apparent in smaller projects. For example, coding style and documentation matter much more when code was written days or weeks before, or when trying to understand someone else's code. Students spend time refactoring code to make it more flexible or to add new requirements; in addition to requiring deeper understanding of the code, it emphasizes the benefits of good design and testing. This approach is also flexible; it can be split and rearranged into different assignments based on the textbook and syllabus, teacher preference, and student aptitude and interest. Finally, using open-ended projects support student-initiated extensions, as course projects or larger independent projects.

2. Text Games & Assignments

Specific assignments (boxed) are summarized below. Note that most prompt students to plan before they start, and to reflect after they finish. The full assignments contain more details, sample code, etc.

Twine Game: Twine is a platform to make it easy for non-programmers to create text games, and should help you think about how such games work. Thus, you should:

1. Read Hudson (2014) Twine, video-game tech for all, *New York Times Magazine*.
2. Go to http://twinery.org, find 2 Twine games that look interesting, and play them.
3. Submit a text file with your name, the date, the game(s) you played, and areas of *strength* and for *improvement* for the Twine software.
4. Use Twine to create your own game with at least 16 places and 32 actions.
5. Submit your game as a Twine source file or output HTML file.

Procedural Game: Design, implement, and test a simple text game program. The game should minimize duplicate code and maximize flexibility, so that it is easy to expand the game, and so that the code could be used without changes for a variety of different games. Thus, you should:

1. Before you edit any code, plan your work and your tests. Email the instructor 100-200 words on how your code will be organized.
2. Design, implement, and test a game with *places*, and *actions* to move between places.
3. Create your own sample game with at least 8 places and 16 actions.
4. In README.TXT, describe areas of *strength* and for *improvement* in your code.
5. Submit a zip file with README.TXT and all of your Java source files (no other files).

Game, Place, & Action Classes: Object-oriented programming (OOP) is quite different from procedural programming. Thus, you should:

1. Study the starting code classes: `Game`, `Place`, `PlaceTest`, `ActionTest`
2. Create `Action` to pass all of `ActionTest`, and `GameTest` to thoroughly test `Game`.
3. Create your own sample game with at least 10 places and 20 actions.
4. Edit README.TXT to describe areas of *strength* and for *improvement* in your code.
5. Submit a zip file with README.TXT and all of your Java source files (no other files).

Items to List, Take, & Drop: The game will be more interesting if it supports **items**, such as prizes or tools (e.g. a key to open a lock). Thus, for this homework you should:

1. *Before you edit any code*, review the existing code and plan your work and tests. Email the instructor a list of methods you plan to change, and how each will change.
2. Create class `Item` and class `ItemTest` to test `Item` as well as possible.
3. Modify `Game` and `GameTest` so that a game contains a list of all items and a dummy place for the player's items, with methods to add, get, and remove items from each list.
4. Modify your code to list items in each place, and to list items carried by the player.
5. Modify your code so that the player can carry, take, & drop items.
6. Create or extend your own sample game with at least 10 places, 20 actions, and 5 items.
7. In README.TXT, describe the hardest parts of this homework, and how to fix them.
8. Submit a zip file with README.TXT and all of your Java source files (no other files).

Inheritance: As programs get larger and more complex, **duplicate (or similar) code** will cause extra work and problems, and should be refactored. In the game, `Place`, `Item`, `Action`, and `Game` contain duplicate code, which can be improved. Thus, you should:

1. *Before you edit any code*, review the existing code and plan your work and tests. Email the instructor a list of duplicate and near-duplicate code you plan to refactor.
2. Create class `Parent` with this duplicate code, and `ParentTest` to test it thoroughly.
3. Remove all duplicate code from `Place`, `Item`, `Action`, `Game`, and their test classes.
4. Create or extend your own sample game with at least 10 places, 20 actions, and 5 items.
5. In README.TXT, describe the hardest parts of this homework, and how to fix them.
6. Submit a zip file with README.TXT and all of your Java source files (no other files).

The text game project can be continued and extended with a variety of other assignments.

- Convert the entire game system to generate and handle exceptions.
- Use inheritance to add support for *multiple descriptions* for a place, action, or item.
- Add a system of *points* (e.g. health or money) that increase or decrease based on player actions.
- Add the ability to *read and write files* with all of the data for a game.
- Use other data structures (e.g. the Java ArrayList, hash maps, linked lists, stacks queues, trees).

3. Conclusions

We have learned a variety of lessons from such projects.

Have a mature design and staged implementation plan for the project, particularly in earlier courses. This avoids wrong turns or dead ends that can lead to significant rework and student frustration, although this is an integral part of software development.

Allow students to affect the direction and final form of the project. We often try to develop the design through classroom activities and discussion. Some students feel more invested in a project if they realize that the teacher has not completely determined what will happen.

Emphasize testing (e.g. Edwards, 2003; Edwards 2004; Wellington, Briggs, and Girard, 2007). Unit test suites are particularly helpful for special cases that students might not consider or might have trouble detecting. In the future, we would like to create more comprehensive test suites and use a system like WebCAT (Edwards and Perez-Quinones, 2008).

Provide detailed requirements, particularly for the first few assignments. *Gradually reduce the level of detail* as students gain experience (e.g. Buck and Stucki, 2000). This helps students learn to analyze problems, elicit incomplete or hidden requirements, and design flexible solutions.

Interleave project assignments with unrelated assignments. This provides some variety for students (and teachers). It increases the sense of returning to previous work, but also gives students more time to catch up and respond to teacher feedback.

4. References

Alphonce, C. and Ventura, P. (2002). Object orientation in CS1-CS2 by design. *Proceedings of the Annual Conference on Innovation and Technology in Computer Science Education*, 70-74.

Barry, E.F., Ellsworth, C.C., Kurtz, B.L. and Wilkes, J. T. (2005) Teaching OO methodology in a project-driven CS2 course. *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 338-343.

Bruce, K.B., Danyluk, A., and Murtagh, T. (2001). A library to support a graphics-based object-first approach to CS 1. *ACM SIGCSE Bulletin*, 33, 6-10.

Buck, D. and Stucki, D.J. (2000). Design early considered harmful: Graduated exposure to complexity and structure based on levels of cognitive development. *SIGCSE Bulletin*, 32, 75-79.

Crowther, W., Woods, D., & Black, K. (1976). Colossal cave adventure. *Computer Game*.

Edwards, S.H., & Perez-Quinones, M.A. (2008). Web-CAT: Automatically Grading Programming Assignments. *ACM Conference on Innovation and Technology in Computer Science Education* (ITiCSE) (pp. 328–328).

Edwards, S.H. (2003). Rethinking computer science education from a test-first perspective. *ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, 148-155.

Edwards, S.H. (2004). Using software testing to move students from trial-and-error to reflection-in-action. *SIGCSE Bulletin*, 36, 26-30.

Hudson, L. (2014, Nov 19) Twine, video-game technology for all. *New York Times Magazine*.

Huggins, J. (2000). Modular term-long CS2 projects. *Frontiers in Education Conference*.

Kussmaul, C. (2008). Scaffolding for multiple assignment projects in CS1 and CS2. *ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, 873-876.

Montfort, N. (2005). *Twisty Little Passages: An Approach to Interactive Fiction*. The MIT Press.

Newhall, T. and Meeden, L. (2002). A comprehensive project for CS2: combining key data structures and algorithms into an integrated web browser and search engine. *SIGCSE Bulletin* 34, 386-390.

Rebelsky, S.A. and Flynt, C. (2000). Real-world program design in CS2: the roles of a large-scale, multi-group class project. *SIGCSE Bulletin*, 32, 192-196.

Wellington, C.A., Briggs, T.H., and Girard, C.D. (2007). Experiences using automated tests and test driven development in Computer Science I. *AGILE 2007*, 106-112.