Introducing Abstraction and Decomposition to Novice Programmers

Raja Sooriamurthi
Information Systems Program
Carnegie Mellon University
Pittsburgh, PA 15213
raja@cmu.edu

ABSTRACT

This paper discusses a learning exercise we use in our beginning programming classes to introduce students to the concepts of abstraction and decomposition. The assignment is to write a perpetual calendar generation program: given a month and a year the program will display the correct monthly calendar. The learning goals of the exercise include how to decompose a large problem into smaller pieces and how to specify what each piece needs to do. This exercise helps students learn the process of incremental and iterative development. More than the actual solution, the value of this exercise is in the several themes of software development that are discussed during its development. We have successfully used this assignment for several years in a variety of CS1/CS2 programming environments (Pascal, C, Java and .net) and also as a Java servlet based web application exercise. Over this period, the case-study has received very favorable feedback from students as to its interestingness and pedagogical value.

Categories and Subject Descriptors

D.1.0 [**Programming Techniques**]: General; K.3.2 [**Computer and Information Science Education**]: Computer science education

General Terms

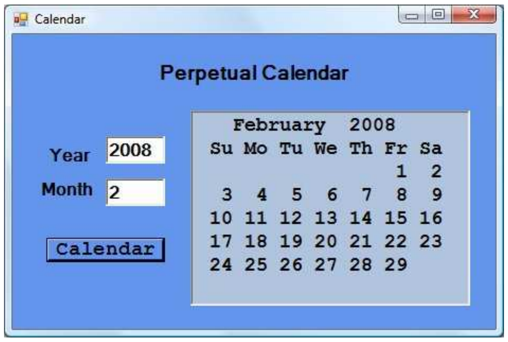
Design

Keywords

CS1/CS2, programming case study

1. INTRODUCTION

Mastering the art of design requires mastering the complexity of the artifacts being designed. Two key tools a student needs to learn to master complexity, what ever the design activity maybe, are abstraction and decomposition [9, 8, 6]. This paper discusses a case-study we use in our introductory programming class to introduce students to these foundational concepts in the context of software development. The assignment is to write a perpetual calendar generation program: given a month and a year the program will display the correct monthly calendar (e.g. Figure 1).



```
    February  2008
Su Mo Tu We Th Fr Sa
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29
```

Figure 1: Calendar for February 2008.

The final program takes around 310 lines of code (130 for the user interface and 180 for the logic). After a simple introductory programming exercise to familiarize themselves with the development environment, this is the second assignment we give to our students. The pedagogical motivations for this case-study are varied; several technical and non-technical lessons can be conveyed via this exercise. The goals of the exercise include, identifying how to decompose a large problem into smaller pieces and how to specify what each piece needs to do. The solution development strategy for this exercise helps students experience the process of incremental and iterative development[10]. Along the way students get a glimpse of a simple model-view[5] dichotomy and also get to learn of the data-code trade offs prevalent in many designs.

In additional to its technical lessons, this problem also forms an intriguing case study on the role of social and geopolitical issues involved in the adoption of technology. The program, as written for this assignment, will work for any month in the future and any month in the past up to September 1752 which is when the Gregorian calendar was adopted by England and other countries. This transition point in history and how the Gregorian calendar was finally adopted, amidst many voices of opposition, forms an interesting study on the importance of effective communication and how technology alone often does not win the day. Adapting the calendar program to accommodate this transition also serves as an example of what software maintenance is about – not necessarily about fixing a broken program but adapting an existing program to evolving needs.

Sun	Mon	Tue	Wed	Thu	Fri	Sat

```
class Calendar {   // model-1
  int[,] days = new int[5,7];
}


class Calendar {   // model-2
  int startDay;
  int daysOfMonth;
}
```

Figure 2: One view for presenting a calendar and two models for representing it. The model on the top right (model-1) requires more data but less computation to display whereas the model on the bottom right requires less data to represent the calendar but requires more computation to render it. (The code above uses 2D arrays of C#.)

We have successfully used this assignment for the past several years in a variety of programming environments (Pascal, C, Java and .net). Extensions to this assignment have also been used in courses on systems programming in scripting languages (e.g., Ruby) and web applications (e.g., Java servlets). The assignment has received very favorable feedback from students as to its interestingness and pedagogical value.

In the next section we discuss our pedagogical motivations for case-study based assignments. The value of this exercise is not in the final solution but in what gets discussed along the way towards the solution. Several key ideas in software development are introduced, which are subsequently expanded upon later in the course.

The following sections of the paper discuss the problem, the learning goals and how these goals are realized, variations and extensions to this case-study. The paper concludes discussing student feedback to this exercise.

2. MOTIVATION: CASE STUDY VS INDEPENDENT EXERCISES

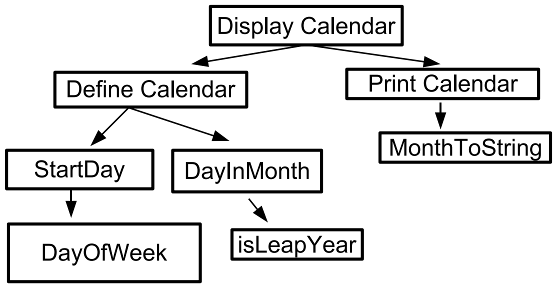
A question faced by an instructor of an introductory programming course when designing assignments is: Should I give several small independent programming problems or should I give a singe large case study[4]. There are advantages and disadvantages to both approaches. Our approach has been to use guided case studies. Our experience has been that an advantage of this approach is that students get more satisfaction having tackled a coherent larger problem. The most common concern expressed against a large case-study is that students can get overwhelmed by the complexity of the problem. We have found that extensively discussing in class how a larger problem could be decomposed into smaller problems and giving students a picture of this decomposition helps. Further, very early on in the course (2nd week) students are taught how to use the debugger of their development environment (e.g., Java/Eclipse or .net/VisualStudio). Learning how to effectively use breakpoints and trace the execution of a method helps them better understand the algorithmic aspects of their code and to better manage the overall complexity of their program. This naturally leads to an incremental, iterative and interactive way of software development[10]. Students are instructed on the general principles of how to develop each component (method) independent of the other and how to test its correctness. For example, the decomposition of the calendar assignment typically results in eleven methods each with a well defined purpose. Students work on each of these methods individually and then incrementally assemble the methods into the final solution.

Another concern expressed about the case-study approach where a deliverable is due in larger time intervals (say 2–3 weeks) instead of more rapid weekly deliverables is that it can lead to procrastination and thereby disadvantage already weak students. Our approach to this pedagogical problem has been to have intermediate milestones. For example, as mentioned above the calendar assignment requires about eleven methods, the development of which is spread across two weeks. To discourage procrastination and to encourage an early start, functional versions of five of these methods are due after the first week. It is important to keep in mind that this is an introductory course on software development and the case-study described in this paper is targeted at students learning programming for the first time. Hence, while we want to encourage early work we also don't want to penalize those who might take a while to get comfortable with the algorithmic way of thinking. Hence the scoring system is set up so as to give an extra 10% for meeting the one week milestone and still have 100% of the assignment weight on the final deliverable. This means that a person meeting the milestone and the specs of the final deliverable could get a score of 110%, and many do. We have found this slight incentive encouraging early work to be sufficient for students to aim for and successfully meet the intermediate milestone. We use this approach of rewarding the meeting of milestones, but at the same time keeping the bulk of the weight of the assignment for the final deliverable, in all our assignments. Student feedback has been encouraging for this approach.

3. REPRESENTING CALENDARS: MODELS AND TRADE OFFS

The history and mathematics behind the creation of calendars across various cultures is rich and diverse[12]. But given the task of just displaying the calendar of a given month, what information does one need? Consider the grid on the left of Figure 2 (the view of the calendar). If one were to fill this grid by hand what information does one need? Students are able to immediately recognize that we only need two pieces of information: (i) what day of the week the month starts on and (ii) the number of days there are in that month. A monthly calendar is thus uniquely determined by these two pieces of information leading to the data representation on the lower right of Figure 2 (model-2). We use the difference between these two models to provide a brief glimpse of two important issues a software developer needs to be cognizant of:

The Model-View dichotomy: The way something is *represented* and the way something is *presented* can be quite different and it helps to think of these two facets of informa-



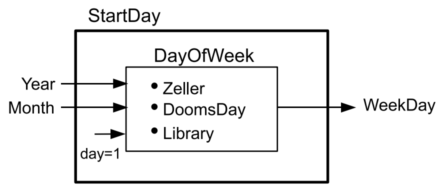


Figure 3: (a) Decomposing the problem of generating a calendar into sub-problems. (b) Solving a more general problem (DayOfWeek) and then tailoring it for a specific need (StartDay).

tion separately. In the follow-up course on OOAD we re-visit this example and at that time elaborate in more detail on the full fledged MVC pattern[5].

Data vs Code trade off: The model-1 representation requires more data but less computation to present it. Whereas the model-2 requires more computation to present the data. At this stage in the course we only briefly discuss these issues and elaborate on them in later assignments or subsequent courses. For instance, in a follow up assignment we sometimes have students take multiple calendars and combine them say three in a row etc. In that instance explicitly having the calendars represented as a grid (model-1) helps.

Let us now consider the problem decomposition process we guide our students through for them to get a better grip on the complexity involved.

4. PROBLEM DECOMPOSITION: REFINEMENTS, ALTERNATIVES AND GENERALIZATIONS

After our initial classroom discussion, students would have identified that any monthly calendar is uniquely determined by two numbers—the day of the week it starts on and the number of days in the month. The problem now reduces to figuring out algorithms to determine these two numbers. Via further discussion we elicit a problem decomposition structure as depicted in Figure 3. An interesting point worth discussing at this juncture is how the task of determining the day of the week a month starts on (StartDay) can be extended into something more general i.e., given an arbitrary date (year, month, day) determine the day of the week the date falls on. This generalization serves two pedagogical purposes:

• Sometimes it may be more useful to solve a general problem (DayOfWeek) and then tailor it to the particular circumstance (StartDay). (This is an instance of the *inventor's paradox*[11].)

• Once we know what we need to do there are several ways by means of which we can do it. This separation of what from the how is a pivotal issues in appreciating the power of abstraction in helping to manage complexity.

This subtle difference between StartDay and DayOfWeek and that the functionality of DayOfWeek can be realized in several ways is an important learning milestone for a novice programmer. As the course proceeds, several times, we revisit this pivotal notion that a single "what" can be realized with several "hows".

5. DETERMINING THE DAY OF THE WEEK

Given an arbitrary date there are several ways in which the day of the week may be calculated. One of the earliest is a remarkable congruence given by Reverend Zeller[14]:

$$w = (d + \lfloor (m+1)26/10 \rfloor + y + \lfloor y/4 \rfloor + \lfloor c/4 \rfloor - 2c) \ mod\ 7$$

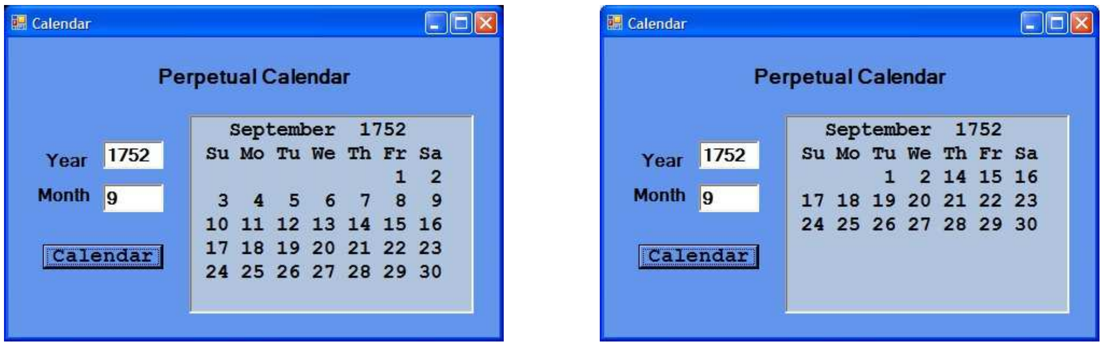
where w is the day of the week, d is the day of the month, m is the month, c is the previous century and y is the year of the century. The details of this congruence are not important for this paper but for the following observations that have pedagogical value for the students:

• Though the above congruence may appear intimidating, there is a straightforward multi-step algorithmic translation. The essential intuitive idea students need to appreciate is that the congruence calculates an offset from an anchor date and that offset modulo 7 gives us the needed day of the week.

• The algorithmic translation of the congruence works assuming that March is the 1st month and the January and February are the 11th and 12th month of the *previous* year. The user thinks in terms of March being the 3rd month and the algorithm needs to translate it accordingly. We use this as a simple illustration of how a user may think of a problem and how our algorithms may think of the same problem may be different and that we need to raise our programs to the level of the user and not the other way around.

• The modulo expressed in the above congruence is the true mathematical modulus[7]. Most programming languages (Java, C, C++, VB.net, C# inclusive) implement the *remainder* operation and not the true modulus. Scheme, Lisp, Ruby, Python are some notable exceptions that implement true modulus (i.e., $-2\%7$ is 5 and not -2). This forms a basis for a discussion of mapping mathematical concepts to programming languages.

An alternative to determining the day of the week is a nice method first given by John Conway known as the Doomsday rule[2]. The principle is the same but the approach is different and simple enough to be mentally done with some practice. Depending on the implementation language one could also use features of built in libraries. For example, Java offers a `GregorianCalendar` class. One can create an instance of this class for a given date and query the instance to determine the day of the week. In our Web applications course students develop a calendar based servlet application wherein they use this approach as opposed to Zeller's congruence. Yet another way to determine the day of the week is to use the current day as an anchor and to write methods to determine the number of days between the current day and the target date.



Year 1752
Month 9

```
  September  1752
Su Mo Tu We Th Fr Sa
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

Year 1752
Month 9

```
  September  1752
Su Mo Tu We Th Fr Sa
       1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

Figure 4: Two calendars for September 1752. On the left is the incorrect calendar many programs will produce. On the right is the historically correct calendar which reflects the adoption of the Gregorian calendar in September 1752.

The main point of discussing these alternatives is to demonstrate to novice programmers that the same sub-task can be implemented in numerous ways without changing the way the sub-task is used i.e., without changing its interface.

6. DISPLAYING THE CALENDAR

As discussed earlier, to encourage an early start we set an intermediate milestone for our assignments. In this case-study the milestone is to implement a method for the Zeller congruence. On completion of this milestone the main focus of the assignment shifts towards displaying the calendar with the right amount of indentation for the first line. Whether this assignment is given as a console application, or GUI based application or web application this part forms a useful reasoning exercise for our students. Often we see the class partition into two groups — those who are able to complete this part unaided and those who need a crucial hint on how to realize the first line indentation.

In the follow-up course to our introductory programming course we require students to extend their calendar application by associating it with a collection of dated web information (e.g., such as the New York Times, slashdot, our local newspaper etc). By specifying an information source, via their calendar application students can browse information pertinent to a particular date. This is the first assignment our students get to use cascading style sheets and we use this to illustrate the utility of separating the content of the calendar from how it appears.

7. TRANSFERENCE: A QUESTION AND A THOUGHT EXPERIMENT

The ability to apply knowledge acquired in one context to another context is a measure of *deep learning*[13]. To evaluate how well our students have understood the conceptual principles behind this assignment we pose the following puzzle and thought experiment to them after the assignment is due and we have discussed our sample solution. The puzzle is: *How many unique yearly calendars are there?* In other words, consider entire yearly calendars printed on cards by institutions such as banks etc. How many such unique cards could there be? The ability to answer this question depends on how well a student is able to transfer some of the concepts of this assignment into a related context. We pose this question as a quick quiz or a minute-paper type task in class. Typically about one fifth of the class is able to answer it immediately[1].

Consider the following thought experiment: Why do years have 365 days as opposed to some other number? Could an alternative choice be made? Understanding the history and origin of ideas plays an important role in coming up with creative alternatives[1]. Concluding the discussion of this case study with a brief discussion of some systems such as the New Earth Calendar (which includes a 364 year of 13 identical months of 28 days each and a leap week every fifth year) opens students minds to alternatives to commonly accepted solutions.

8. A LESSON FROM HISTORY

As the introductory course proceeds students are aware of the fact that 80% of the cost of a piece of software is in maintenance. But the word maintenance, as applied to software, is bit of a misnomer in that it is not like house or automobile maintenance — there is no wear and tear. Rather maintenance is about fixing bugs but more often it is about enhancing the functionality of a system to do things above and beyond it was expected to do when originally conceived. We use this case study to illustrate this point. The program that the students write will work correctly for any day in the future. It will also work for any day in the past up to September 1752. The program will not work for September 1752 and for any month before it. The output of the program and the historically correct version of the calendar for that month are given in Figure 4.

The reason for this discrepancy is socio-political rather than mathematical. It was in September 1752 that most of the English speaking countries switched from the Julian calendar to the Gregorian calendar which more accurately reflected the natural calendar. Similar to the chronological shift that occurs during transition into day-light savings time, in September 1752 a calendar shift took place requiring several days to be skipped. Though the Gregorian calendar was first introduced in the 14th century, it took several centuries for most countries to adopt it. The efforts of Lord Chesterfield in bringing about the adoption of the Gregorian calendar and the aftermaths of its adoption (widespread

[1]Similar to a monthly calendar being uniquely determined by the day of the week the month starts on and the number of days in the month, a yearly calendar is also uniquely determined by the day of the week the year starts on as well as the number of days in the year ((365 or 3655 days) thereby giving $7 * 2 = 14$ unique yearly calendars.

riots) offers many valuable lessons for any student of technology[3]. The main lesson being that technology alone does not win the day.

9. SOFTWARE MAINTENANCE

The software engineering question to be taken away from this discussion is: Is the program developed as part of this exercise "broken"? It is interesting to note how various calendar programs actually handle the adoption of the Gregorian calendar. The Unix `cal` utility correctly accommodate for this adjustment. But the Unix `pcal` utility, which generates postscript versions of calendars, does not handle pre September 1752 dates and explicitly flags an error. Contrarily the `calendar` function of the popular emacs editor behaves like the program described in this exercise: it doesn't correctly handle the adoption of the Gregorian calendar nor does it flag an error that it can not handle it. Contrasting the behavior of these three programs (`cal`, `pcal`, and `calendar`) is a useful discussion.

10. ASSESSMENT AND LESSONS LEARNED

On completion of this case-study we handout a feedback sheet requiring students to anonymously answer three questions: (i) What did you learn from this assignment (ii) What did you like about it (iii) What did you not like about it? i.e., is there anything you feel that could be improved. This section summarizes student feedback, describes the various challenges faced and the primary lessons about programming and software development learned.

Overcoming analysis paralysis. For a novice programmer, as the second programming exercise, this case-study is challenging (130 GUI lines of code and 200 program lines of code). The logic is spread across several methods and a common concern amongst students is not knowing where to start. But as we systematically work through the decomposition depicted in Figure 3 students realize that each chunk is manageable. Experiencing this realization is one of our pedagogical goals.

Incremental, iterative and interactive development. Many of the pieces of the overall solution architecture can be worked on separately. As with prose, we also highlight the value of re-writing code. Case in point, one can start off with an incomplete version of a method to test for leap years and then extend it to take care of century years properly. By the end of this assignment student's skill in using a debugger has also improved considerably.

Alignment of responsibilities. An error that we see at this introductory stage is a misalignment of responsibilities. For example, everything that has to do with printing the calendar should be contained within the method for PrintCalendar. Occasionally we see students printing the banner (the month, year, weekday names) in a place different from where they print the calendar grid.

Importance of starting early. Many students have favorably commented on the usefulness of having a reward associated with an early start and successful meeting of an intermediate milestone.

Grading rubric. Reflecting the earlier discussed problem decomposition, student are provided with a detailed grading rubric to help guide their development efforts. Students are also required to self evaluate their performance on the assignment using the rubric. The course grader uses the same rubric to determine their official score. Comparing their evaluation against the graders emphasizes the importance of self-assessment. We use this grading strategy in all our assignments.

11. SUMMARY

For several years we have successfully used this case-study in different programming classes — introductory programming (in Pascal, C, Java, .net), scripting languages (Ruby) and web applications (Java servlet). This assignment forms a good exercise in algorithmic reasoning for introductory students. It introduces students to many important ideas of software development that we expand upon further in the curriculum: abstraction, decomposition, iterative development, using a debugger, data-code trade off, glimpse of the MVC pattern etc. The case-study provides an example of what software maintenance is about and provides an intriguing historical backdrop for what it often takes to have new technology accepted. In the .net environment students have personalized their application by placing it in the shortcuts area of the Windows task-bar. The web based version involves the creation of a calendar based information browser servlet. Students who have demoed their web application to friends have commented that their friends have requested copies of the program for their personal use. It has been rewarding to see that based on class feedback and evaluations students have found this case-study to be both interesting and pedagogically useful.

12. REFERENCES

[1] J. L. Adams. *Conceptual Blockbusting: a guide to better ideas.* Perseus Publishing, 4th edition, 2001.

[2] E. R. Berlekamp, J. H. Conway, and R. K. Guy. *Winning Ways for Your Mathematical Plays: Games in Particular*, volume 2. Academic Press, 1982.

[3] P. D. S. Chesterfield. *Dear Boy: Lord Chesterfield's letters to his son.* Bantam, 1989.

[4] M. J. Clancy and M. C. Linn. *Designing Pascal Solutions: case studies with data structures.* W.H.Freeman, 1992.

[5] E. Freeman and E. Freeman. *Head First Design Patterns.* O'Reilly Media, 2004.

[6] D. Ginat. On varying perspectives of problem decomposition. *Proceedings of SIGCSE*, pages 331–335, 2002.

[7] R. F. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science.* Addison-Wesley, 2nd edition, 1994.

[8] O. Hazzan. Reflections on teaching abstraction and other soft ideas. *SIGCSE Bulletin*, 40(2):40–43, 2008.

[9] J. Kramer. Is abstraction the key to computing? *Commun. ACM*, 50(4):36–42, 2007.

[10] R. C. Martin. *Agile Software Development, Principles, Patterns, and Practices.* Prentice Hall, 2002.

[11] G. Polya. *How to Solve It: A new aspect of mathematical method.* Princeton University press, 2nd edition, 1957.

[12] E. M. Reingold and N. Dershowitz. *Calendrical Calculations.* Cambridge University Press, 2nd edition, 2001.

[13] M. D. Svinicki. *Learning and Motivation in the Postsecondary Classroom.* Anker Publishing, 2004.

[14] C. Zeller. Kalender-formeln. *Acta Mathematica*, 9:131–136, 1886.