AC 2012-3766: A COURSE FOR DEVELOPING PERSONAL SOFTWARE ENGINEERING COMPETENCIES

Tom Reichlmayr, Rochester Institute of Technology

Tom Reichlmayr is an Associate Professor in the Department of Software Engineering at the Rochester Institute of Technology. Prior to transitioning to his academic career, he worked as a software engineer in the process automation industry in a variety of roles over a span of 25 years. His teaching and research interests include the development of undergraduate software engineering curriculum, especially at the introductory level. Of primary interest is the study of software development process and its application to course curriculum and student team projects

Prof. Michael J. Lutz, Rochester Institute of Technology

Michael Lutz is a professor of software engineering at the Rochester Institute of Technology, where he founded the first undergraduate software engineering program in the United States in 1996. His professional interests include software engineering education, formal methods, software design, and engineering concurrent software systems.

A Course for Developing Personal Software Engineering Competencies

Abstract
The strength of a software development team is the sum of the capabilities of each individual team member. There exist at the personal level core software engineering competencies that need to be cultivated to allow an individual to fulfill their potential as an effective team contributor. Students in a course introducing team based software engineering typically possess adequate introductory programming skills, but often lack other competencies required to execute a software project successfully. Students have rarely been introduced to concepts beyond programming, such as estimation and planning, continuous integration, detailed design, debugging and unit testing. Part of being a software engineer is the knowledge of multiple programming languages and tools; without such knowledge it is impossible to make intelligent engineering decisions.

Contemporary education philosophy stresses active student initiative and personal responsibility learning. In our case, with a rapidly evolving technology landscape, students must come to realize that, as in the workplace, many skills are not so much taught as learned. This paper captures our experiences with a second year software engineering course designed to address these challenges. In addition to discussing the topics covered in the course we also present active and cooperative learning practices utilized in class activities.

Introduction

Our undergraduate software engineering curriculum incorporates team-based activities in all upper-division courses, as the ability to work effectively on teams is a critical aspect of software engineering practice. Effective team participation, however, assumes basic engineering competence on the part of each team member. The goal of this course is to both enhance and assess each student's individual technical skills in preparation for the team-based courses that follow. The Personal Software Engineering course (SE350) covers individual software construction skills: planning, detailed design, programming, debugging and unit testing. Software construction is the central activity guaranteed to happen on every project.[xx] As such, the quality of the construction substantially affects the quality of the software, and knowledge of construction principles is essential whether you are engaged in construction or are responsible for another phase of the project (architecture, requirements elicitation, quality assurance, maintenance, etc.).

Part of being a software engineer is knowledge of multiple programming languages; without such knowledge it is impossible to make intelligent technology tradeoffs. What is more, the language landscape evolves rapidly over time, so the ability to learn and apply new languages is an essential skill. The two languages we use are C and Ruby; we do not expect students to master either language by the end of the term, but to demonstrate competence with the core concepts in each.

Why C? Well, C was one of the earliest high level languages to support efficient use of computer resources. In addition, C is the progenitor of a whole host of C like languages, of which Java is

the most popular current example. This means we can focus on the unique aspects of C, particularly memory management and pointers, without having to spend much time on concrete syntax for loops, conditionals, expressions, etc. C is also the immediate ancestor of C++, the current language of choice when efficiency and direct access to hardware is critical. Those who eventually move on to engineer real-time and embedded systems will find C and C++ are pervasive in those domains.

We selected Ruby since dynamic languages like Ruby and Python have established themselves as excellent tools for scripting, rapid prototyping, and flexible development where raw efficiency is not a priority. Ruby also has a rich set of defined classes which support the development of concise, clear object-oriented applications. As our students were arriving at our course with a year of Java object-oriented experience already under their belt, we only needed to spend minimal time in the review of object-oriented principles before turning them loose in the Ruby environment.

Ruby is also the base for the web application framework Ruby on Rails. The section of the course devoted to Rails provides the opportunity to introduce relational database systems and web application components such as object to relational mapping. We also included this subject material as we had no other required course which directly addressed web or database applications. At minimum, the use of Rails armed our students about to begin co-op with at least basic knowledge of how web or database applications work. It also turned out that this brief introduction to Rails sparked the beginning of a number of individual student projects both inside and outside of the department.

Background

The students enrolled in our Personal Software Engineering course are in their second year and have already completed one year of a typical introductory computer science sequence (CS1-CS3) using Java. Prior to the creation of SE350, our Software Engineering students would take an additional Computer Science course (CS4) which dealt exclusively with C++. We found that our students emerged from this introductory programming sequence as knowledgeable programming tacticians, but lacking in good software construction habits. As students entered our sequence of Software Engineering courses, they were placed in project teams where the sometimes unbalanced distribution of programming tasks left them with limited opportunity to practice or refine their software construction skills. As a department we looked for an alternative course to CS4 that would allow us to engage our students sooner in their development as programmers so as to provide a means of personal assessment and help establish fundamental software engineering skills.

Classroom Environment

A section of SE350 enrolls a maximum of 40 students with one instructor and one upper division student as a course assistant. Classes are two hours in length and held twice a week in our studio labs, with one PC per student. Students are organized into groups of four; we used a “playing card” distribution to determine the groups so that for example, a class of 40 would have 10 groups (Aces, Ones, Twos, etc.) and each student in the group would be assigned a suit (spade, heart, club, diamond). [10] We enforce a strict pair programming [11] policy during class activities, but we rotate pairs within the group by announcing pair assignments for the day as

"diamonds and spades" or "hearts and clubs". The same organization also supports active learning exercises such as "jig-saws," where students from different groups but the same suits collaborate on part of an assignment and then report back to their groups with the collective knowledge needed to complete the activity. We change groups three or four times over the term, enabling each student to meet and work with most of the other students in the class.

Although we provide fully loaded Windows PC's with powerful IDE's in our lab, we intentionally limit class activities to a command line driven, Linux environment. Students learn (or are reacquainted with) text editors (emacs, vi, nano, etc.), bash scripts and source control - we require use of SVN for every artifact they create. For our C programming activities we use the GNU compiler (gcc) and debugger (gdb), with builds defined in make files. Our rationale was that small, precise "hand tools" are sometimes more appropriate than a large, multipurpose power tool such as Eclipse. In addition, students gain a better appreciation of what an IDE is doing under the hood, allowing them to adapt to situations where their favorite IDE is unavailable.

Course Outline

The course spans an eleven week term; Table 1 provides a week-by-week summary of topics covered. An important goal in designing the course was to devote the majority of each two hour session to an activity that encourages student collaboration while providing rapid response to problems by an instructor or student assistant.

Week	Topics
1	Linux Command Line Source Control (svn) C - Introduction, I/O
2	Make files C – Arrays & Strings
3	C – Pointers & Memory Allocation
4	Debugging (gdb) C – Linked Data Structures
5	Unit Testing C++ (General Overview)
6	Introduction to Ruby General Ruby Concepts
7	Ruby as a Scripting Language Regular Expressions
8	Ruby as an Object Oriented Language Ruby Unit Testing

9	Relational Databases Web Application Frameworks Introduction to Ruby on Rails
10	Ruby on Rails Testing in Rails
11	Final Exam Week (Practicum in C or Ruby)

Table 1. – SE350 Course Outline

Class Activities

Activities were developed for each class and are worked on by student pairs. Classes have s short up-front lecture on the topic of the day, and the activity is designed to reinforce the topic while being doable by the end of class. For programming activities, pairs are required to estimate the time expected to complete the activity, then track and report the actual time required and to reflect on the reasons for any difference (there are *always* differences). While the instructor and student course assistants were available for help, the "escalation ladder" for help is: your partner, other members of your group, and only then the instructor or course assistant. Some students initially revert to immediately raising their hand for help, but they are gently nudged to try working through issues with their group first.

Projects

Projects and practica account for the majority of the student's final grade; these are assessed on an individual basis, unlike the in-class activities and team- based projects in other courses. Four projects of 2-3 weeks duration are assigned during the term. Projects followed the topics being covered in class: a C project with no pointers, a C project with pointers and allocated memory, a Ruby project, and a Ruby on Rails project.

In addition to their code submissions, students are also responsible for maintaining a "tracking" document (Figure 1) that captures their estimated and actual time spent on each phase of the project. (Appendix A has the complete description of the Sudoku project.) At the completion of each phase, they assess the reasons for differences between estimated and actual time and provide a brief narrative as to what worked well and what they would change in the next phase. This reflection document is submitted along with work products associated with the phase.

At the conclusion of a project, groups meet in class to identify the top three process improvement items from their individual reflection documents. As a class we consolidate the group lists into a class-wide list, and target a subset of these for tracking during the next project. Many of the suggested improvements are what one might expect – *start the project sooner, don't procrastinate, read the project description, don't be afraid to ask for help,* and so on. The reflection session also provided the opportunity for students to share scripts they developed to automate some of their tasks, and to describe distinctive approaches to the assignment.

4010-350 Tracking & Reflection Document

SuDoKu

Level 1 – Initialize & Print Empty Puzzle

Estimation & Plan

Estimated Time 02 **Hrs** 00 **Mins**

Plan for this level: What do you have to do? What order will you complete the work?

For this level, I have to complete the init_puzzle() and print_puzzle() functions. For init_puzzle(), I have to initialize the puzzle array to all zeros and initialize the fixed array to all FALSE values. For the print_puzzle() function, I have to write code that will print a Sudoku board in the correct format given by the project requirements.

I will start by completing the print_puzzle() function, since I believe it will be easier to implement than the init_puzzle() function. Then I will complete the init_puzzle() function.

Actual time and reflection.

Actual Time 01 **Hrs** 15 **Mins**

Actual / Estimate 0.63

Ratio explanation (required if < 0.90 or > 1.10)

I overestimated the amount of time it would take me to complete the print_puzzle() and init_puzzle() functions. I overestimated because I am only just starting the project and am not completely familiar with the entire project.

Lessons learned, problems encountered, obstacles overcome, etc.

I guessed that print_puzzle() would take less time to implement than init_puzzle(), but in reality it took much more time to implement than init_puzzle(). I confused the init_puzzle() function with the configure(pf) function, so the lesson to be learned is that I should fully understand what is to be implemented for each function before I make time estimates.

Figure 1. Sample Project Tracking Document

Practica

Practica (or "lab practicals" as they are sometimes referred to in other disciplines) are designed to assess and motivate and assess student learning [1]. This mirrors a common practice in industry hiring, where companies often include a short logic or programming problem as part of the interview process. The goal in all cases to gage how the individual works through a problem and to provide an indicator of their technical ability.

Practica are given in class at the conclusion of each major topic (C with no pointers, C with pointers, Ruby, etc.). Appendix B contains a sample practicum description. We focus on short programming problems that a competent engineer can complete within an hour. The problems reflect the in-class activities and project assignment, and are submitted in stages to reward incremental development and submission. Practica are open book, open notes, open internet – in essence, open everything except mouths. Practica turn out to be the most accurate individual assessment method we have, as students can't hide in a group, so we better identify those in need

of additional help. Students quickly realize that there is no real way to "study" for a practicum in the way one might study for an exam; instead, the most effective preparation for a practicum is the practice and experience obtained from the class activities and projects. This motivates students to attend class and seek out help for projects on more frequently.

Good Software Engineering Habits

> *"I'm not a great programmer, but I'm a good programmer with great habits"*
> - Kent Beck

This quote by Kent Beck is a mantra within our course. The goal of the programming activities and projects is not to reward students who can produce the cleverest solution, but to develop repeatable engineering habits that competent developers employ when building quality software products. These same habits are directly transferable to all activities in the software development life-cycle. [9] The following sections include a sampling of those habits we identified which contribute to success, and thus we want our students to acquire as they evolve into professional software engineers.

"Spend more time reading your code than you do writing it."
Software developers write a lot of code, but unless they have reason to go back and modify it, they rarely spend an equivalent amount of time reading it. When developing C programs, students are encouraged to play "beat the compiler." The goal was to complete a program with the minimal number of compiles that produce errors and warnings which careful inspection could have removed up front. This technique was advocated by Watts Humphrey in his Personal Software Process (PSP) methodology [4] and can be traced back to the days of submitting a deck of IBM punch cards to a central batch system to compile, link and execute your program. As the turnaround time for a single batch submission could be anywhere from minutes to hours based on the availability of the system, programmers were very careful to "desk check" their source code so as not waste their submission on a trivial compile error. A side benefit to this desk checking exercise is that in the process of scrutinizing their code in an attempt to identify compiler errors, programmers would also uncover logic and algorithmic errors as well.

With turnaround times now in the range of milliseconds, modern day programmers, including our students, have developed the habit of rapid, sometimes random, edit-compile cycles until a clean compile occurs. This habit can be dangerous in a liberal language such as C that permits direct access and manipulation of memory while performing limited type checking. The focus on checking before compiling also introduces students to the concept of detecting and removing defects at the earliest point in the development cycle. Humphrey noted that in his classes students were surprised to see how much more effective manual code review was at identifying defects as compared to finding them later during testing. [5]

"Do the simplest thing that could possibly work"
We continually emphasize the value of incremental development and building in very small steps. Much like a large software project strives to continuously deliver business value to its customer in small increments that can be evaluated and used as feedback for subsequent product deliveries; we adopted the same though process for even small programming assignments. The

first thought when starting on a new assignment should be, "what is the simplest thing I can do that will progress me to the next step", at which point the thought process would repeat. Our students would experience the drawback of not taking this approach when they created even a modest amount of source code in the editor and then attempted their first compile. The incremental approach also holds true for testing, Test Driven Development (TDD) [7] has extolled the virtues of test a little, code a little, as any debugging process is usually restricted to new code that has just been added to a base of tested code. Every programmer is guilty of running to far ahead of uncompiled or untested code, and our students are no different.

"Strive for Continuous Improvement"

One of the key principles of Humphrey's Personal Software Process is "to help software engineers to know their own performance: to measure their work, to recognize what works best, and to learn how to repeat it and improve upon it" [5]. We have embraced that philosophy in our course by emphasizing the need to collect quantitative data and qualitative observations during each activity undertaken in the course. Opportunities are offered to the students to reflect on an individual or group level in order to identify areas for improvements for future assignments. The mentality of continuously looking for process improvement opportunities is a common characteristic of successful professional software development teams. Engraining this habit early at the individual level contributes to making the student a more effective member on future team projects and ultimately in their career.

"Care About Your Craft"

This is the first of many valuable "tips" we used from *The Pragmatic Programmer* [6] which is required reading for our students. It states in simple terms, *"Why spend your life developing software unless you care about doing it well?"* One of the course goals is to instill a sense of professionalism and pride into our students. In addition to our programming activities, we included assigned readings from contemporary professional journals, magazines and blogs that students discussed in online forums. We also include material examining the state of software engineering practice.

Student Feedback

Students completed a course survey we have used as feedback to continually fine tune the course. Among some of the more interesting results from a survey of 121 students over four offerings of the course were:

Attitude towards pair-programming

71% - I enjoyed working in pairs most or all of the time
29% - Usually not at all

There is a stereotype with computing students that due to their introverted nature, they would prefer to work alone as opposed to working in pairs. We were pleasantly surprised to see that this was not the case and bolstered our enthusiasm for continuing the pair policy.

Pace of in-class activities

15% - Too slow

64% - Just Right
21% - Too Fast

Difficulty of projects
5% - Too Hard
75% - Just Right
20% - Too Hard

One of the challenges of the course is that the students have a wide spectrum of programming experience. Students with substantial programming backgrounds can complete the activities and projects rather quickly, while less experienced students struggle to keep up. One experiment we have tried is that after the first project and practicum we create new groups that attempt to more evenly distribute students based on their performance and experience. The results in forming "novice-expert pairs" during class activities can be more or less successful depending on how willing the more "expert" student is to collaborate with his or her partner. [12] In any event, we feel comfortable that pace and difficulty of assignments is appropriate for the vast majority of the students.

Student Comments (Favorite Parts of the Course)

"I enjoyed being exposed to and learning new languages. I feel it will be very beneficial to me; not only the languages I learned, but the process I learned in learning new languages"

"I really like that we got to learn C. Having an understanding of linked lists helped me a ton on a Microsoft interview, and I think C is a very useful language after this course."

"I actually liked the practicums. I felt that they were a good way of assessing knowledge in the curriculum."

"Learning something besides Java!"

Student Comments (Least Favorite Parts of the Course)

"It just seems like too much is jammed into the course. I learned everything, but I would have liked more time to explore things."

"Having to teach myself different languages in a limited amount of time"

"Segmentation faults in C!"

We selected these specific comments since they reflect the views of two contrasting sets of students in terms of their classroom expectations. One of the characteristics of the course that we are continually evaluating is the idea of having students be more responsible for their own learning versus a more heavily teaching centric approach. We feel strongly that the learning centric approach will be more beneficial to students in the long term and will instill more

confidence in their own ability to address new programming languages and development environments.

Conclusions

We have developed a personal software engineering course that embraces active and cooperative learning in achieving the overarching goals of having students:

- Construct and test a software component in accordance with contemporary practice.
- Plan, estimate, track and analyze the effort required to construct and test a software component.
- Apply fundamental software construction techniques in using new languages and tools.
- Achieve growth in professional software development including technical writing / documentation and review of contemporary literature.

The course has been positioned early in the curriculum of our undergraduate software engineering program to encourage early adoption of programming habits that contribute to the development of quality software products The acquisition and practice of these individual skills will translate into more effective team performance when they move on to larger, team-based projects.

References

[1] Bennedsen, J. & Caspersen, M., "*Assessing process and product-a practical lab exam for an introductory programming course"*, Frontiers in Education 2006

[2] Cockburn, A., *"Designing an incremental-iterative one-semester, undergraduate course in software engineering",* Humans and Technology Technical Report HaT TR 2006.03, Sept 6, 2006

[3] Hou, L. and Tomayko, J., *"Applying the Personal Software Process in CS1: an Experiment"*, in Proceedings of 29th SIGCSE Technical Symposium on Computer Science Education, Atlanta, Feb 26-Mar 1 1998, pp. 322-325

[4] Humphrey, W.S., *"The Personal Process in Software Engineering*", Third International Conference on the Software Process, Reston, Virginia, October 10-11, 1994, pp 69-77.

[5] Humphrey, W. S., *PSP: A Self-Improvement Process for Software Engineers*, Addison Wesley, 1997.

[6] Hunt, A. & Thomas, D., *The Pragmatic Programmer*, Addison-Wesley, 2000

[7] Hunt, A. & Thomas, D., *"Learning to Love Unit Testing",* Software Testing and Quality Engineering (STQE), Jan/Feb 2002

[8] Maletic, J.I., Howald, A, Marcus, A., *"Incorporating PSP into a Traditional Software Engineering Class: An Experience Report*", 14th Conference on Software Engineering and Training, Charlotte, NC, 2001.

[9] McConnell, S., *Code Complete 2*, Microsoft Press, 2004

[10] Millis, B. & Cotteel Jr, P., *Cooperative Learning for Higher Education*, American Council on Education Oryx Press, 1998

[11] Williams, L. & Kessler, R., *Pair Programming Illuminated*, Addison-Wesley, 2003

[12] Williams, L. & Kesler, R., *"The Effects of Pair-Pressure and Pair-Learning on Software Engineering Education"*, Conference of Software Engineering Education and Training, 2000

Appendix A – Sample Student Project

Sudoku Project Description

For this project you will complete a skeleton C program that presents a sudoku puzzle to a player and provides commands for the player to incrementally solve the puzzle. Here is a tutorial on the rules of sudoku for those who are unfamiliar with the game.

Setup (Provided Skeleton Code)

Download the file sudoku.zip from the software engineering website to a clean working directory for this project. At this point you should see the following files & directories:

bool.h
: A header file declaring a pseudo-type for `bool` along with constants `FALSE` and `TRUE`.

main.c
: The main driver function: It parses the command line arguments, initializes the puzzle, loads the puzzle configuration from a file provided as an argument, and, if the load succeeds, reads user commands (one command per input line) and calls the appropriate processing function in the **puzzle** module.

arguments.h & arguments.c
: Interface to and implementation of a module processing the command line arguments to (a) determine whether or not the input commands should be echoed (optional **-e** flag), and (b) open the puzzle configuration file, which defines the initial state of the puzzle. Exits with appropriate messages if there are problems with the arguments provided.
Note: You need not concern yourself with this module, as it is only used by the main function. On the other hand, perusing it would give you examples of using some standard C functions and idioms.

puzzle.h & puzzle.c
: The **puzzle** module you will complete. The header, **puzzle.h**, defines the interface you ***must*** adhere to, as the only source file you will submit is the corresponding implementation, **puzzle.c**. In particular, **puzzle.c** is the *one file you must edit* and the *only file you may edit*.

solve_sudoku & skeleton_sudoku
: Two executable files, compiled and linked for Linux. **solve_sudoku** represents a full, complete implementation of the requirements, including error handling; use it as a reference against which to compare your program as you go along. **skeleton_sudoku** was compiled directly from the files we provide; all the functions in the **puzzle.c** file have been stubbed out, and those functions with return values return whatever is required to

make it appear that everything proceeds normally. This can be used as a baseline for comparison with your version.

Makefile

A file making it easy to compile the source code and link the object files. To create an executable file named **sudoku** use the command:

```
make sudoku
```

As you add functionality to the **puzzle.c** implementation, this executable will be used to test your changes. **Makefile** also has a series of targets to run your program using different valid and erroneous puzzle configurations, as well as command scripts that are error free or seeded with specific types of errors. More on these tests below.

p+s

A directory with (**p**)uzzle configurations and command (**s**)cripts for use in testing.

Description.html

This file (in case you are working remotely and can't access the SE web servers).

Track+Reflect.doc

An M/S Word file in which you record your estimate and actual time it took to do each level, along with reflections on why they diverge (if they do) and your perception of the project as a whole.

Note: If you have problems running either **solve_sudoku** or **skeleton_sudoku**, this is probably because the files are exectuable. To fix this, run the following command on Linus:

```
chmod +x solve_sudoku skeleton_sudoku
```

This will set the e(**x**)ecute permission on these files, at which point you should be able to run them.

Using the Example Programs

All the examples will be given using **solve_sudoku**. You can substitute **skeleton_sudoku**, but the program will look like it did nothing (which, in fact, is what happens in the skeleton). And, of course, as you implement the required functionality, you can use your compiled version, **sudoku**.

The program is invoked in one of two ways:

```
./solve_sudoku configfile
./solve_sudoku -e configfile
```

In both cases, **configfile** is a text file giving the initial placement of digits in the puzzle. The optional **-e** flag determines whether or not the user commands are echoed to standard output. If you're running on the console and typing in commands, you should probably not use the flag. On the other hand, if you are running commands from a script (a text file), then **-e** lets you see what commands are executed in sequence.

Example

```
./solve_sudoku -e p+s/good_puzzle.txt <
p+s/script_good_solve_puzzle.txt
```

runs the program using a legitimate puzzle configuration in **p+s/good_puzzle.txt** with echoing turned on and the commands coming from the script file **p+s/script_good_solve_puzzle.txt** - this script has commands that add digits to the puzzle to solve it. Along the way, the partial puzzle solution is printed at several points.

You could run the puzzle interactively as follows:

```
./solve_sudoku p+s/good_puzzle.txt
```

In this case, the main program directs the initialization and loading of the puzzle configuration, prints the initial board, and enters the command loop. Each time through the loop the program prompts command: to which you respond with a single lower-case letter command and possibly digits used by the command. Spacing is important: the command letter must be the first character on the line, and the command and arguments are separated from each other by a single digit; each command is terminated by a newline. The commands are:

`p`	Print the puzzle
`q`	Quit (also on end-of-file)
`a` ***r c d***	Add digit ***d*** to the puzzle at row ***r*** column ***c***
`e` ***r c***	Erase the digit at row ***r*** column ***c***

Rows, columns, and digits are all in the range `1..9`.

Design Notes

Puzzle Configuration File Format

The configuration file comprises a series of 0 or more rows, each row beginning with three digit characters giving the row, column, and value for one of the initial placements in the puzzle. All three digits must be in the range `1..9`. For example, the line `135` means `5` is to be placed at row `1`, column `3` in the initial configuration.

Puzzle Data Structures

The puzzle implementation is built on two 10x10 matrices: **puzzle** and **fixed**. The matrices are 10x10 to permit 1-based indexing into the 9x9 array defining the puzzle proper; the 0th row and column are unused.

The **puzzle** matrix holds the values placed so far, with 0 representing a blank (available) location or cell. After initialization to all zeros, this matrix is filled in first from the configuration file and then via commands read from standing input. In later stages of the implementation, the program

will enforce the Sudoku consistency constraints - a row, column, or 3x3 region may not contain the same value in two cells.

The **fixed** matrix is a boolean matrix initialized to all `FALSE` values. When a puzzle is configured (via data in a configuration file), the row and column for each value so placed is set to `TRUE` in **fixed**. In later stages of the implementation, attempting to erase a fixed value is an error and the cell is not changed.

Puzzle Module Interface

The public interface has of an enumeration, **op_result**, which is used to report back the status of each add or erase command. **OP_OK** signals success; the other values in the enumeration reflect the different errors that may be detected. **OP_BADARGS** means the command gave a row, column, or digit that was not in the range `1..9`, **OP_OCCUPIED** is for attempts to place something at a location with an existing value, **OP_ILLEGAL** is for placements that would violate the Sudoku rules (a duplicate value in a row, column, or region), **OP_EMPTY** is for attempts to erase an empty cell, and **OP_FIXED** is for attempts to erase a fixed cell (one set by the puzzle configuration). Only **OP_OK** results in a change to the puzzle layout; all erroneous commands have no effect.

The interface proper is defined by five visible functions:

`init_puzzle( )`	Initialize the **puzzle** and **fixed** matrices.
`configure( pf )`	Read configuration lines from the file whose handle is ***pf*** and use this to fill out the **puzzle** and **fixed** matrices. If any configuration errors are encountered, this function prints a message and exits - thus a return from this function represents successful configuration.
`print_puzzle( )`	Print the puzzle - The first line is 25 dashes (`-`). - A line of 25 dashes is also printed after the 3rd, 6th, and 9th row of the puzzle. - Each puzzle row begins with a vertical bar (`|`) - Each cell in the row is printed as a space and the cell's value (space for a 0 value). - After the 3rd, 6th, and 9th colum, a space and a vertical bar are printed.
`add_digit( r, c, d )`	Add digit ***d*** to the puzzle cell at row ***r***, column ***c***, returning success or error status.
`erase_digit( r, c )`	Erase the puzzle cell at row ***r***, column ***c***, returning success or error status.

Tasks & Deliverables

The project is organized as a sequence of six functionality levels. Each level requires you to extend the work done at the previous level. For each level 1 through 6 you will fill out the associated estimating, tracking, and reflection activity outlined in the file **Track+Reflect.doc**. 15% of your project grade rides on your attention to the details of estimating the time it will take you to complete the activity, track that time and accurately record it when the level is done, and provide thoughtful reflection on the work you did. Some levels are relatively straightforward and require less in the way of reflection, however they should not be ignored completely.

Similarly, 15% of your grade is based on the quality of your implementation. Aspects considered include:

- **Naming:** Are variable, function and other names meaningful? Do they have a clear relation to their role in the computation? Would a competent developer be able to pick up this code and understand why you chose the names you did?
- **Documentation:** Do you use comments to simply repeat the code or to provide insight into its context and intention? Are comments, clear, concise, and correct? Are comments readable - do they avoid distracting grammatical and spelling errors? Are the aligned with the code to make it obvious which comments refer to which sections of code?
- **Structure**: Are the bodies of functions organized into clear, coherent sets of statements that clearly contribute to the function's goals? As appropriate, are sections of code factored out into local functions of their own, either to simplify the original function, to enhance readability, or to adhere to DRY (Don't Repeat Yourself)?
- **Style**: Does the layout of the code enhance its readability? Do you use consistent indentation in line with examples done in class? Do you use blank lines to separate distinct blocks of code? Do you use spaces to enhance readability (or are expressions scrunched together, making it difficult to distinguish variables from operators from constants)?
- **Source Code Control**: Did you demonstrate appropriate us of source code management during the development of the project? Did you provide svn comments for each commit of an update file? Is your test code under source control?

Level 0 (Compiles & Links)

This is a pseudo level, in that all your puzzle.c must do is compile and link against the rest of the program, and your Track+Reflect.doc must contain a reflection on whatever you did. As these conditions can be met by simply submitting the puzzle.c skeleton in the zip file and almost anything for thetracking and reflection, this is really a test of whether you can submit to myCourses.

Level 1 (Puzzle Module Initialization and Printing of an Empty Puzzle)

Fill in the body of the **init_puzzle()** function and write the **print_puzzle()** function. Skeletons for two static methods related to printing are there for your completion and use: **print_dashes()** and **print_row(row)**. When your modifications compile and link, test them using make:

make test_l1

Level 2 (Initialization, Configuration and Printing of a Real Puzzle)

Fill in the body of the **configure(puzzle_file)** function, ignoring any error conditions for now. This requires you to

- Read three digit characters and a newline from the file whose handle is the argument (see the **Puzzle Configuration File Format** section).
- Convert each digit character to the number it represents,
- Place the value in the puzzle cell designated by the row and column, and finally,
- Mark the selected cell as fixed.

When your modifications compile and link, test them using make:

make test_l2

Level 3 (Add and Erase Commands)

Fill in the bodies of **add_digit(row, col, digit)** and **erase_digit(row, col)**, ignoring any error conditions for now.

When your modifications compile and link, test them using make:

make test_l3

This runs three tests: one a simple test of adding, one a simple test of erasing, and one which solves a real puzzle - the last printout shows a completed puzzle.

Level 4 (Syntax and Simple Add/Erase Errors)

Change the **configure** function:

1. Discard everything after the three digit characters up to and including the next newline or **EOF**.
2. Check to ensure they are all digit characters in the range '1' through '9'; if any are not in range, print the message:
   Illegal format in configuration file at line *N*, where ***N*** is the line number, and call **exit(1)**.
   You may find it useful to complete the skeleton helper function **in_range(value)**.
3. Ensure that the cell at the selected row and column is not already filled with a value; if it is, print the message:
   Illegal placement in configuration file at line *N*, where, once again, ***N*** is the line number, and call **exit(1)**.

Change the **add_digit** function:

1. If any of the three arguments is not in the range 1 .. 9, return **OP_BADARGS** without changing the puzzle. You may find the helper function **in_range(value)** useful.
2. If the selected cell already has a value in it (i.e., it is non-zero), return **OP_OCCUPIED** without changing the puzzle.

Change the **erase_digit** function:

1. If either of the two arguments is not in the range 1 .. 9, return **OP_BADARGS** without changing the puzzle.You may find the helper function **in_range(value)** useful.
2. If the selected cell is already empty (i.e., its value is 0), return **OP_EMPTY** without changing the puzzle.
3. If the selected cell is fixed (non-eraseable), return **OP_FIXED** without changing the puzzle.

When your modifications compile and link, test them using make:

```
make test_l4
```

This runs a series of tests, the first few with different combinations of bad configuration files, and a last one with a valid puzzle but bad (**a**)dd and (**e**)rase commands.

Level 5 (Row and Column Rule Violations)

Change the **configure** function:

1. Check to ensure that the value being placed in the puzzle is not already in the row or column specified. If it would be a duplicate, print the message:
   Illegal placement in configuration file at line *N*, where ***N*** is the line number, and call **exit(1)** - you may want to add this check onto the previous one that produces this message. You may find it useful to complete the skeleton helper functions **row_contains(row, digit)** and **col_contains(col, digit)**.

Change the **add_digit** function:

1. Check to ensure that the value being added in the puzzle is not already in the row or column specified. If it would be a duplicate, return **OP_ILLEGAL** without changing the puzzle.
   You may find the helper functions **row_contains(row, digit)** and **col_contains(col, digit)** useful.

When your modifications compile and link, test them using make:

```
make test_l5
```

This runs a series of tests, the first few with different combinations of bad configuration files, and a last one with a valid puzzle but bad (**a**)dd commands.

Level 6 (Region Rule Violations)

Change the **configure** function:

1. Check to ensure that the value being placed in the puzzle is not already in the region specified. If it would be a duplicate, print the message:
   Illegal placement in configuration file at line *N*, where ***N*** is the line number, and call **exit(1)** - you may want to add this check onto the previous one that produces this message. You may find it useful to complete the skeleton helper function **region_contains(row, col, digit)**.

Change the **add_digit** function:

1. Check to ensure that the value being added in the puzzle is not already in the region specified. If it would be a duplicate, return **OP_ILLEGAL** without changing the puzzle. You may find the helper function **region_contains(row, col, digit)** useful.

When your modifications compile and link, test them using make:

```
make test_l6
```

This runs a series of tests, the first few with different combinations of bad configuration files, and a last one with a valid puzzle but bad add commands.

Assessment (100 pts total)

Level 0 (puzzle.c compiles and links)	10
Level 1 (initialize and print empty puzzle)	10
Level 2 (initialize, configure and print a real puzzle)	15
Level 3 (add, erase and solve a real puzzle)	15
Level 4 (detect bad syntax and simple add / erase errors)	5
Level 5 (detect adding duplicate values in row or column)	5
Level 6 (detect adding duplicate values in a region)	10
Estimation, tracking, analysis and reflection	15
Implementation quality, source control	15

Appendix B – Sample Practicum

(Also representative of a typical class activity performed in pairs)

Instructions

This practicum has a firm 50 minute time limit and will be taken in lab during class time only. The practicum is open book, open notes, open Internet, open everything – except help from your neighbors or instructor.

Problem

This exercise requires you to write a C function that deletes a symbol / value pair from a hash map. The prototype (signature) of the function is

void drop(char *symbol)

The given code hashmap.c includes the following functions:

insert( ) - inserts a symbol / value pair into the hash map using symbol as the key
hash( ) – uses the given symbol to perform a hash function which computes the index (bin) into the hash table
lookup( ) – returns the hash table bin entry (symbol / value) for a given symbol
dump( ) - print all the symbol / value pairs for the hash table
main( ) – edit this function to include any tests you created to verify your solution.

The following standard string functions are also used:
int strcmp(char *s1, char *s2) – compares two strings, returns zero if strings are equal
char *strdup (char *s1) – returns an ***allocated*** block of memory containing a duplicate of string s1

1. Download the starter code and makefile:
   - hashmap.c
   - Makefile
2. READ the existing code before you begin writing the **drop()** function. Draw diagrams as needed to (re)familiarize yourself with the hash map data structure.
3. Run the sample program which inserts symbol/value pairs and prints the contents of the hash map.
4. Write the **drop()** function and modify the **main()** function as needed to test your solution. A sample test case is included in main() to get you started.
5. Enter your name in the header block of the program.
6. You do not need to modify any existing functions other than main().
7. When you are ready to submit your solution, drop your file into the Practicum One drop box in myCourses. You may make multiple submissions; I will only look at the last one submitted.

NOTE: In writing **drop()**, it is your responsibility to ensure that all the allocated memory resources/areas are no longer needed in the hash map are properly released in the proper order.