Computer Science for Practicing Engineers

Computer Science for Everyone Else

CMU-ISR-11-115

October 5, 2011

David Garlan

Anthony J. Lattanze

Table of Contents

Overview and Purpose

The Computer Science for Practicing Engineers (CS4PE) course is a breath-oriented course covering key topics in computer science that represent critical foundational knowledge for anyone developing software. The course is designed for students and professionals who do not have a formal background in computer science, but currently write, or anticipate writing software as a key part of their job. CS4PE represents one important facet of Computation Thinking [Wing 2006]. Computational thinking is a fundamental skill for everyone, not just for computer scientists. To those core concepts essential to a particular professional discipline, we need to add computational thinking to every professional's analytical ability.

The purpose of this report is to describe the motivation for, and the development of, the CS4PE course. The aim is to describe why such a course is needed, the design of the course, our experience in delivering the course since its inception, and feedback received from a peer review of the course content and assignments during the 2010 Conference on Software Engineering Education and Training. We will also discuss how and why the course has evolved since its initial development and delivery, as well as future directions that we expect the course to move in.

Background and Motivation

In 2007 we created a new professional masters program in embedded software engineering, housed in the Institute for Software Research in the School of Computer Science at Carnegie Mellon University. Our purpose in creating this program was to address the fact that embedded systems are pervasive today: from control systems for chemical processing, production lines, automotive, and aerospace, to cell phones, appliances, and entertainment systems. Engineers building these systems must bring together complex software, computing hardware, and peripheral devices to satisfy demanding requirements, operating under significant physical, technical and environmental constraints.

Despite the challenges inherent in developing embedded systems, today most embedded systems engineers learn how to design and build the software for these systems on-the-job, without any formal background or training. Most of these engineers have a formal background in a traditional engineering discipline such as electrical or mechanical engineering. While these engineers have some of the skills necessary to build complex embedded systems, their background is typically inadequate to meet the interdisciplinary challenges that arise with embedded systems in which knowledge of hardware, software, and specific domains must come together. Today embedded systems software is complex, large, and requires the specific expertise of trained software engineers with a solid background in computer science principles.

Initially we created the CS4PE course to meet the needs of the Masters of Embedded Software Engineering program, one of several professional masters programs in software engineering at Carnegie Mellon [Garlan, Gluch and Tomayko 1997]. Many of the students admitted to this program had a formal background in electrical engineering and some experience writing software, but needed more depth in computer science fundamentals to be prepared for advanced courses and to thrive as practicing embedded software engineers.

However, after delivering a pilot version of the course, we noticed that students from other schools and institutes were interested in taking the course. In our first pilot delivery of the course we had students in the course from the chemical engineering and materials sciences departments. In interviewing these students we learned that they too had demanding computing needs, that they needed to write software to support their work, and that they needed a better understanding of computer science principles to write the software required in their daily work. In fact, we often admit students into our Masters of Software Engineering program who have a great deal of experience writing software, but who have a formal background in academic disciplines other than computer science. So the key motivation for developing CS4PE was to bridge the gap that many students have in their formal backgrounds and their practical need to understand computer science well enough to write software in practice.

More broadly, it became clear that both industry and academia have an increasing need of computer professionals with solid foundations in computational thinking, but who have not had this background in their undergraduate degree program. According to a CRA study the main factors that make fulfilling this need difficult are the decline in enrollment of undergraduates in computer science, and the high rates of migration of computer science undergraduates to other majors [Cohoon and Chen 2003]. The high degree of reluctance of computer science students to choose graduate training over immediate entry into workforce is also causing a shortage in professionals with graduate-level training [Cohoon et al. 2003].

On the other hand, the last couple of decades of expansion in the software industry have produced a large number of experienced practitioners who approach software development without a formal background in computer science. Such practitioners may have many years of software development experience and may have had some courses or technical training with specific computer languages, but they lack training in fundamental aspects of computational thinking such as computing theory, data structures, and design. Without such knowledge it is difficult to engineer appropriate levels of trust, security, and reliability in the increasingly complex systems of today or to fully exploit computational power in non-software domains such as biology, astronomy, medicine, etc.

While there have been efforts to address the decline in the number of undergraduates majoring in computer science, little has been done to provide suitable educational alternatives for practicing engineers without a formal background in computer science. Currently, the only way for practicing engineers to fill the gaps in formal training is by enrolling in undergraduate level classes, which are time-consuming and do not take advantage of their experience. Moreover, lacking prerequisite knowledge for most graduate programs, such engineers are discouraged from seeking graduate training.

The course Computer Science for Practicing Engineers attempts to fill that gap.

Course Philosophy and Learning Objectives

In the development of CS4PE we realized that it would be necessary to strike a balance between theory and practical computer science techniques. Too much theory might alienate practitioners attending the course; too little focus on theory, and the course might not serve students over the years in practice as

technologies evolve and change. To strike and maintain this balance, we developed these guiding principles:

1. Focus on real-world problems: The course is designed to prepare students for immediate competency so that concepts presented in class can be directly applied in real world situations. Much of the course material reflects the types of computational problems practitioners face regularly today, and are likely to face in the future.
2. Theory + Example + Practice: We wanted to avoid long stretches of theoretical. The course is therefore developed around a model where theoretical background is provided in one or two lectures, complemented by practical and/or industrial example of the theory in practice, followed by a hands-on exercise using the principle(s).
3. Domain neutrality: Throughout the course we remain domain-neutral. We did not want to focus solely on information technology, web-based systems, and database-oriented application domains. Nor did we want to focus solely on embedded systems domains. Instead we attempted to select examples from both of these areas, especially where the domain might influence the technical approaches that a student might take. This provides opportunities to teach sound engineering judgment.
4. Programming language agnosticism: We do not use a particular programming language for the course, but rather use three different languages. In practice, engineers are routinely faced with having to quickly learn and be productive with a variety of languages. Languages affect the implementation of data structures and algorithms and exposure to several languages helps reinforce the application of theory in a practical context. Additionally the course addresses both object-oriented and non-object-oriented paradigms in terms of data structures, languages, and application design.

With respect to learning objectives, the primary goal of CS4PE is to provide engineers, who do not have formal training in computer science, with a solid background in the key principles of computer science. As prerequisites to entry, we expect that students have some project, internship, and/or industry experience, and are familiar with at least one programming language. Students may have little or no formal background in algorithms, data structures, analysis, or design techniques and methods. The course is designed to complement the experience that engineers already have in writing software with formal computer science underpinnings that will make them more capable in developing software intensive systems.

Specific teaching objectives include:

- familiarize students with the key problems in computer science and established approaches to solve them (problem-solution paring)
- provide students with a broad background in families of algorithms used to solve various classes of problems commonly encountered in practice
- improve the student's ability to recognize and analyze critical computational problems in practice, generate alternative approaches, and judge among them through hands-on exercises

- enable students to better understand, analyze, and characterize those factors that influence algorithmic computational complexity, performance, and memory consumption
- provide students with a palette of options and techniques for analyzing, measuring, and reasonably ensuring correctness in algorithms and applications
- increase student awareness and understanding of data structure families and their underlying strengths and weaknesses for a given computational and application context
- improve the student's ability to performed detailed code-level design, analysis, and documentation
- provide students with a broad survey of operating systems principles, types of operating systems, and their capabilities and limitations
- provide students with a firm background in computer language principles to help students learn and transition into new program languages quickly (as often required in practice)

Course Structure

Given this ambitious set of objectives, it should be apparent that a primary impediment to developing CS4PE is calendar time. The course is structured around a full semester model (about 30 lectures) where each student can be expected to devote approximately 12 hours a week to course work. The course is constructed around five major topic areas. Each topic area features a set of lectures with one or more exercises. The major topic areas are described below:

Topic Area	Description
Algorithms	This section begins with an introduction to algorithm representation and basic analysis. A key theme is that algorithm representation approaches should capture the essence of an algorithm and should be simple, clear, and intuitive in order to support analysis. We introduce the idea that our representation and analysis should be as rigorous *as is practical*. To these ends we avoid deep mathematical theory and introduce formal concepts and techniques that can be directly used in common situations in practice. A key point that we stress in algorithmic representation and analysis is that we remain language neutral and factor out the hardware and operating systems in representing and analyzing algorithms. We stress the point that scope is a critical element of algorithmic representation and subsequent analysis. Several techniques are presented to represent algorithmic pre/post conditions and dynamic behavior such as: state diagrams and basic predicate logic. We provide notation and guidelines for using pseudo code and flowcharts. In terms of algorithmic analysis we focus on correctness and measurement at this point in the course; however we introduce more analysis techniques throughout the course. In this section, we use the representation techniques discussed and show various techniques and examples for algorithmic analysis. In terms of correctness, we introduce concepts of partial and total correctness. We look at various techniques of using assertions to check for correctness. Finally we take a broad survey of verification and validation (V&V) techniques and issues associated with V&V. A second facet of analysis is that of algorithmic measurement. Big-Oh analysis is introduced and used to analyze algorithmic complexity in terms of performance and

	resource consumption (primarily memory). Finally this section concludes with a broad survey of specific algorithmic strategies and the kinds of problems that they are used to solve. The goal is to build a problem-solution pairing in the minds of the student. Families of algorithms that we introduce include: Brute force, Divide and Conquer, Branch and Bound, Dynamic Programming , Greedy, Heuristic, and Probabilistic. Examples of each are introduced and discussed in class and are revisited throughout the course.
Data Structures	The purpose of this section is to introduce students to fundamental data structures, common operations on them, and principles of encapsulation. We also focus on the concepts of composition of applications from fundamental data structures and operations. We build on representation and analysis concepts by analyzing data structure operations in terms of performance and memory utilization. The aim is to arm students with tools to guide their judgment when selecting fundamental code structures to solve problems most efficiently. Specific abstract data types (ADTs) included in this section include: Lists, Stacks, Queues, Trees, Heaps, Graphs, Hashes, and strategies for efficiently structuring information on secondary storage. For all of these ADTs, we analyze operations for ordering, traversing, searching, inserting, deleting, merging, and other fundamental operations as applicable.
Concurrency	In this section we discuss algorithms and techniques for designing concurrent systems and managing concurrent execution. We discuss thread versus process models of concurrent execution. The concepts of fairness, deadlock, and livelock are introduced, and specific algorithms and techniques for managing fairness and avoiding deadlock and livelock are introduced. Various mechanisms for ensuring mutual execution are presented. We introduce scheduling concepts and strategies that range from stochastic to hard deadline scheduling , together with various techniques for analyzing schedulability.
Computer Languages and	In this section we introduce fundamental principles of programming language principles, language structure, and fundamental features of computing languages. We also familiarize the student with basic concepts underlying the design and implementation of programming languages. Students are introduced to imperative, functional, and logical programming paradigms. We compare and contrast structured and object-oriented programming paradigms. In addition to computer language concepts, we discuss the mechanics of interpretation and compilation and the basic elements of concepts of parsing, lexicographical analysis, code generation, optimization, and linking. The goals of this section are to provide students with 1. Information that will ease the many transitions they will make in their careers from one language to another. 2. The ability to discern between various languages and understand how languages can promote or inhibit broader design goals and strategies.
Design Concepts	In this section we introduce detailed software design concepts and provide an overview of software design techniques. We discuss structured and object oriented design concepts and philosophical underpinnings of both. We introduce detailed design methodologies, notations, and specific techniques for analyzing structured and object oriented designs as well as techniques for documenting detailed designs. We also continue the comparison and contrast of structured and object-oriented design approaches introduced in the language section of the course and discuss the

	strengths and weaknesses of both methods.
Operating Systems and Concurrency	In this section we introduce basic operating system concepts and discuss the differences between general and special purpose operating systems. The aim is to equip students with the ability to discern between various types of operating systems and match the appropriate operating system for a given application/domain.

Textbooks and Readings

Finding the proper textbooks for this course was a major challenge during the initial course development. We wanted to identify one or two textbooks that address both data structures and algorithms that would be used as the sole source of readings for the entire course. There is no shortage of data structures and algorithm textbooks on the market. The challenge was to find textbooks that were domain, language, and operating system agnostic. Many data structures and algorithm textbooks feature a specific language such as Java, C#, or C/C++. We even found some data structures and algorithm textbooks that were domain-specific. This would not work for this course because our students come from many domains and will return to work in many domains. These are fine textbooks if you need to learn a specific language in a specific context, but not for teaching general principles of data structures and algorithms. We firmly believe that in order to best prepare our students, we need to prepare them for the dynamism they are likely to face in industry in terms of shifting languages, hardware, operating systems, and varying domain computing needs. Since languages and technologies come and go, our goal was to identify textbooks that helped us teach enduring principles in computer science that would serve our students today and in the long term.

A second criterion we considered in selecting course textbooks were practical examples and industrial case studies. Finding and agreeing upon adequate textbooks became one of the most daunting early challenges in developing this course. After a lengthy search we settled on the following:

- David Harel, Yishai Feldman, *Algorithmics: The Spirit of Computing (Third edition),* Addison-Wesley, 2004
- Alfred V. Aho, Jeffrey D. Ullman, John E. Hopcroft, *Data Structures and Algorithms,* Addison-Wesley, 1983

The Harel-Feldman textbook is a wonderful text that focuses broadly on Algorithms, but also strategies for solving practical problems. It is concisely written, includes many practical examples, and is domain and language agnostic. While its focus is algorithms, representation, and analysis, there is some coverage on data structures woven throughout the text. The Aho et al. text is a long-used text on data structures. The best feature of this text is that it is language neutral and uses a standard pseudo-code to illustrate data structures - this was very difficult to find in a data structures book. The down side is that the book is old, but the material is still relevant. One problem with this text is that we explore a few newer data structures that are not in the book, so we must compliment the text with supplemental readings. There are newer data structures textbooks, but while a newer text would be highly desirable, any data structures book that is language biased would be unacceptable for this course. While data structures are the focus of the Aho et al. text, it also includes decent coverage of algorithms woven

throughout the book. While not perfect, these two textbooks complement each other nicely, and at this point, they seem to be an optimal choice. While we are still using these texts, we are always on the look-out for better texts that are newer, more relevant, and include industrial examples and/or case studies.

Neither of these textbooks, however, provides adequate industrial case study examples. So in addition to these textbooks, we learned that we would need to include supplemental readings to compliment the textbooks that illustrate the use of theory in practice. Many of the supplemental readings change from year to year, but are often industrial case studies or lessons learned that directly illustrate the use of concepts and techniques presented in the classroom. In some cases, the supplemental readings are part of "thinking and analysis assignments," where students read the case study and are asked to analyze some aspect of the case study using course concepts and techniques.

Experience and Feedback

The initial version of the CS4PE course was prepared in the summer of 2008, and a pilot version of the course was first delivered in Fall semester of 2009. The course included students with backgrounds in electrical engineering, materials science, and chemical engineering.

The first version of the course started with lectures in algorithmics, correctness, efficiency, automata, formal languages, and computability. The first three topics were reasonably well accepted and relevant to the practitioner-oriented audience. However, the topics on automata, formal languages, and computability were far too theoretical for the audience, and were not sufficiently related to real-world problems that practitioners face. For example, while Turing machines are certainly of theoretical, historic, and research interest, they are not often of practical use to engineers designing and building software applications. The number of lectures on these formalisms (6 in total) was delivered in one continuous sequence, with no industry examples or exercises to illustrate key concepts in a practical way. At the end of this series of lectures an assignment was given to the students that required them to apply a combination of the techniques from these lectures. They struggled with this assignment academically. But more importantly, the assignments did not sufficiently include realistic problems that students could relate to their work experience. The students were a bit overwhelmed by the experience and a few students dropped the course at this point. When these students were interviewed, they indicated that the course was too theoretical and "computer sciencey" for their needs. They indicated that they did not know how the concepts related to application development.

As noted earlier, striking a balance between theory and practice has been the biggest challenge in this course and this part of the course has been adjusted the most to better strike this balance. Based on this experience, much of this material was "lightened up," and specific topics in these lecture where kept or removed based upon their relevance to practitioners. In the second delivery, however, we found that we had removed too much. By the third delivery of the course we felt we had found the right balance. One technique that seems to work very well is to limit how much theory is presented in a single block. Presenting small "chunks" of theory, with practical examples is most effective. Now in the course, when theoretical material is presented in the class, it is done in no more than two lectures immediately followed by industrial examples, in-class exercises, and hands-on homework. So a lesson learned and a

good general heuristic for preparing lectures is that 1) you must be able to sufficiently present the theory in 1 to 3 lectures, and 2) you must be able relate the theory to an industry case study and/or a realistic exercise.

For example, in the current course we introduce algorithmic representation techniques to include practical techniques in predicate logic, temporal logic, and state machines. This is done as a small chunk of two lectures that includes practical examples and discussion woven into the lectures (see the course syllabus in Appendix 1). We follow these lectures with a simple assignment (see Appendix 2) that challenges students to apply the representation concepts on a realistic problem. In the next small chunk, we present analysis techniques, followed by an exercise that combines representation and analysis techniques. This continues until we work our way through all of the formal material. With each chunk we build on previous knowledge of the course material. Each assignment (e.g., see Appendix 3) challenges students to use the concepts and techniques of all former chunks to a large extent. The capstone project (see Appendix 4) changes from year to year, but in general asks a student to use nearly all of the concepts of the course on a team-oriented application development project.

In 2010 we had the opportunity to present the CS4PE course to a working group of software engineering educators at the Conference on Software Engineering Education and Training (CSEET – for more information visit: http://conferences.computer.org/cseet/2012/CSEET_2012/Index.html). Members of the working group included educators with considerable experience in computer science and several early founders of Software Engineering Education programs throughout the United States. Most of the feedback for the CS4PE course was positive and most of the attendees indicated that such a course has been needed for some time. All indicated that balancing practicality with theory and finding high quality "industrial strength" exemplars would be a central challenge. There was a great deal of discussion regarding how much formalism should be included in such a course and how it could be conveyed to students, who may have little background at all in discrete mathematics. It was because of this feedback that we added a lightweight, practical module on predicate logic to the course.

Another point of discussion was how much (if at all) should software engineering concepts be introduced to the course. In the design of the course, we focused on computer science principles alone. However, some of the attendees indicated that software engineering concepts could also be woven into the course as well. Most thought that basic lifecycle concepts should be discussed and basic software engineering principles could be introduced in a few lectures and reinforced in exercises. For example, to stress the importance of requirements and design, students could be asked to track their time in each phase: requirements, analysis, design, implementation, test, and repair. The instructor would collect and consolidate and present the data for the class. Trends would show where students are spending their time in the lifecycle, highlighting the importance of early lifecycle work.

Future Development and Directions

In the last year of the course we have felt confident enough in the lecture materials and the assignments to package the course for distance delivery and export to other organizations. Such packaging includes taping of all of the lectures, exercises, instructor guidelines and grading rubrics, and exams. At the

writing of this report, the course is packaged for distance delivery, and we will be offering it remotely to a broad set of practitioners in our distance versions of our software engineering masters programs.

As we continue to deliver the course on campus, refinements continue. The struggle to balance theory and practical application is ongoing. Each year we review our choice of formalisms and work hard to understand which will best serve the practitioner once they leave the course and the University. We constantly review the examples, case studies, and the exercises to ensure they are relevant and serve to illustrate the principles in our lectures.

Moving forward, we plan to add other topics to the course. Specifically, students have indicated that they would like to see the following topics worked into the course.

- Networking: basic concepts; packet structure; protocol development; the details of socket level programming
- Secondary storage: how to effectively store information on disk; on-disk data structures; I/O performance considerations; binary file format; stream data storage
- Databases: basic concepts; introduction to exemplar technologies; schema design strategies; optimization strategies

All of these are reasonable topics that we would like include. However, we must balance this with material already in the course and live within the one-semester time constraint. In addition to adding these topics, we would like to include assignments that would provide opportunities for students to practice these concepts. The course currently has five individual assignments and one team assignment. In the future, as we add these topics, we will also need to refactor the assignments to include concepts and techniques presented in these additional lectures.

Acknowledgements

The development and initial delivery of this course, as well as support for the workshop on this topic, was supported by the National Science Foundation under Grant No. 0836133. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. We also would like to thank the staff and faculty who assisted with the development and delivery of the course, including Orieta Celiku, Mel Rosso-Llopart, and Mark A. Ardis. We would also like to acknowledge the organizers of the 2010 Conference on Software Engineering Education and Training (CSEET), who allowed us to host our workshop on this topic as part of their annual conference. Finally, we thank the students who took the course and provided invaluable feedback during its formative stages.

References

D. Garlan, D. Gluch and J. Tomayko. “Agents of Change: Educating Software Engineering Leaders.” In IEEE Computer, Vol. 30/No. 11, pages 59-65, 1997.

J. McGrath Cohoon and Lih-Yuan Chen. “Migrating out of Computer Science.” Computing Research News, Vol. 15/No. 2, pages 2-3, March 2003.

J. McGrath Cohoon, K. Baylor and Lih-Yuan Chen. “Continuation to Graduate School: A Look at Computer Science Departments.” Presented at the Association for Institutional Research Reform, May 2003.

J. Wing. “Computational Thinking.” Communications of the ACM. Vol. 49/No. 3, pages 33-35, March 2006.

Appendix 1

The course syllabus.

17-630 Course Syllabus – Fall 2011

Course Title: Computer Science Principles for Practicing Engineers
Meeting Times: Monday and Wednesday, 3:30PM to 4:50PM
Classroom: SCR 265

Course Description:

Software is a ubiquitous part of modern society. Of course software is most visible in information technology and web oriented applications, but software cuts across nearly every scientific and engineering discipline known to mankind. Software is a critical element in aerospace and automotive systems, chemical engineering and processing, medicine, manufacturing and production lines, cell phones, appliances, entertainment systems, and more. Engineers building these systems must bring together complex software, computer hardware, and peripheral devices to satisfy demanding requirements often operating under significant physical, technical and environmental constraints. Because of the challenges inherent in developing these systems organizations in these domains often hire engineers and scientists with formal backgrounds in disciplines in domains other than software engineering and computer science. Practicing engineers and scientists have deep domain training and experience, but their background is often inadequate to meet the interdisciplinary challenges that arise in building complex systems where knowledge of hardware, software, and domain expertise must come together. Most of these professionals learn how to design and build the complex software for these systems on-the-job, with little formal background or training. Today software intensive systems are complex, large, and require the specific expertise of trained software engineers with a solid background in computer science principles. This process can involve considerable trial and error, resulting in poorly designed and documented systems; defect laden software, bloated product development costs, unmaintainable software, and missed opportunities to leverage software development investments. The result is that many of these engineers and scientists are not fully realizing their potential as practicing software engineers. This course is designed for practicing software engineers to bridge these gaps in formal computer science training and prepare engineers and scientists to design and build better software intensive systems.

Prerequisites:

Previous coursework in computer science (such as data structures or algorithms) is not necessary. However, students should have at least 1 year of experience writing software applications preferably in an industrial setting. Academic project experience may suffice for industrial experience. Students in doubt regarding their experience should obtain instructor's permission.

Course Objectives:

The primary objective of the course is to provide engineers and scientists without formal training in computer science, a solid background in the key principles of computer science. The primary purpose of this course is to complement any experience that students may already have in writing software with formal computer science underpinnings, making those engineers and scientists more capable in developing software intensive systems. Specific learning objectives include:

- preparing students for immediate competency so that course material can be directly applied in real world situations
- improving the student's ability to recognize and analyze critical computational problems in the course of their work, generate alternative solutions to problems, and judge among them
- enabling students to better understand, analyze, and characterize those factors that influence algorithmic computational performance and memory consumption.
- increasing student's awareness and understanding of detailed code structures and their underlying strengths and weaknesses
- improve the student's ability to performed detailed, code-level design and document the design in an understandable way

Textbooks and Readings:

Required:

1. "Algorithmics: The Spirit of Computing (Third edition)," David Harel, Yishai Feldman
2. "Data Structures and Algorithms," Alfred V. Aho, Jeffrey D. Ullman, John E. Hopcroft)

A variety of papers and URL readings will be provided in addition to these required textbooks.

Instructors:

Anthony J. Lattanze:

Teaching Professor, Institute for Software Research International
Senior Member of the Technical Staff, Software Engineering Institute
300 South Craig Street, Room 278, 412-268-4736, lattanze@cs.cmu.edu

Manuel Rosso-Llopart:

Associate Teaching Professor, Institute for Software Research International
300 South Craig Street, Room 270, 412-268-4629, rossollo@cmu.edu

Brief Description of Course Assignments:

This course includes numerous hands-on programming and analysis assignments. Students will program in a variety of languages. The programming assignments will be a combination of individual assignments and a team oriented final capstone project. In addition to programming assignments, students will be assigned readings to support the lecture materials.

Assignment	Percentage	Description
Individual Assignments	50%	These are small assignments that can vary: readings Q&A, assignments based on lecture, coding, and so forth. These assignments will be done individually.
Final Capstone Project	50%	The course project will be completed in 2-3 person teams.

Schedule of Lectures

Lecture Number	Major Topic	Lecturer and Topic	Assignments
1	Overview of Computing	Course Intro and History of Computer Science	**Read:** Har04: Ch1–Ch2
2		Algorithm Representation and Analysis	**Read:** Har04:Ch4, Ch5) **A1:** Analysis and Representation Exercise
3		Algorithmic Strategies	**Read:** Har04:Ch9, Ch14(pp359-362), Assertion Essentials
4		Correctness	**Read:** Har04:Ch6, BigOhPrimer.pdf (p129-143) **A2:** Ball Shuffle Puzzle
		NO CLASS – LABOR DAY	
5		Algorithmic Measurement and Analysis	**Read:** AHU83: Ch1, Parnas72.pdf
6	Data Structures Algorithms	Introduction to ADTs	**Read:** Har04:pp35-38, LinkedListBasics.pdf, **A3:** Simplex (refer to PracticalOptimization.zip)
7		Introduction to Lists	**Read:** AHU83:Ch2, Har04:pp92-94
8		Lists and Stacks	**Read:** Har04:pp71-74, QueueADT.pdf
9		Queues	**Read:** AHU83:Ch8
10		List Sorting	**Read:** AHU83:Ch3 **A4:** Virtual Rube Goldberg Machine
11		Trees I	**Read:** AHU83:Ch5, Har04:pp39-43
12		Trees II	**Read:** AHU83:Ch12, Har04:pp91-92
13		Heaps I	**Read:** Har04:pp87-91
14		Heaps II	**Read:** Har04: ch7: pp168-170 **FINAL PROJECT:** The Crawler
15		Graphs I	**Read:** Digraph Data Structures.pdf **A5:** Six Degrees
16		Graphs II	**Read:** Read: "How Hashes Really Work", Abhijit Menon-Sen; http://www.perl.com/pub/2002/10/01/hashes.html

17		Hashes I	**Read:** "Selecting a Hashing Algorithm", B. J. MCKENZIE, R. HARRIES AND T. BELL
18		Hashes II	**Read:** Har04 Ch13, StructureDesign.pd
19	Computer Languages and Design Concepts	Design – I	**Read:** Har04 Ch14, OO&UML.pdf
20		Design – II	**Read:** SevenDeadlySins.pdf, ProgrammingLanguages.pdf
21		Secondary Storage and Databases	**Read:** FundimentalsOfStorageCh13.pdf
22		Computer Languages I	**Read:** SD&OO.pdf
23		Computer Languages II	**Read:** Har04 Ch10
24	Miscellaneous Topics	Introduction to Concurrency I	**Read:** concurrency.pdf
25		Concurrency and Scheduling	**Read:** Tanenbaum A.S, "Operating Systems Design and Implementation, Third Edition"; Sections: 1.1-1.3 and 3.1-3.3; http://www.flylib.com/books/en/3.275.1.1/1/
26		Operating Systems I	**Read:** Tanenbaum, Sections 4.1-4.3
27		Operating Systems II	**None**

Appendix 2

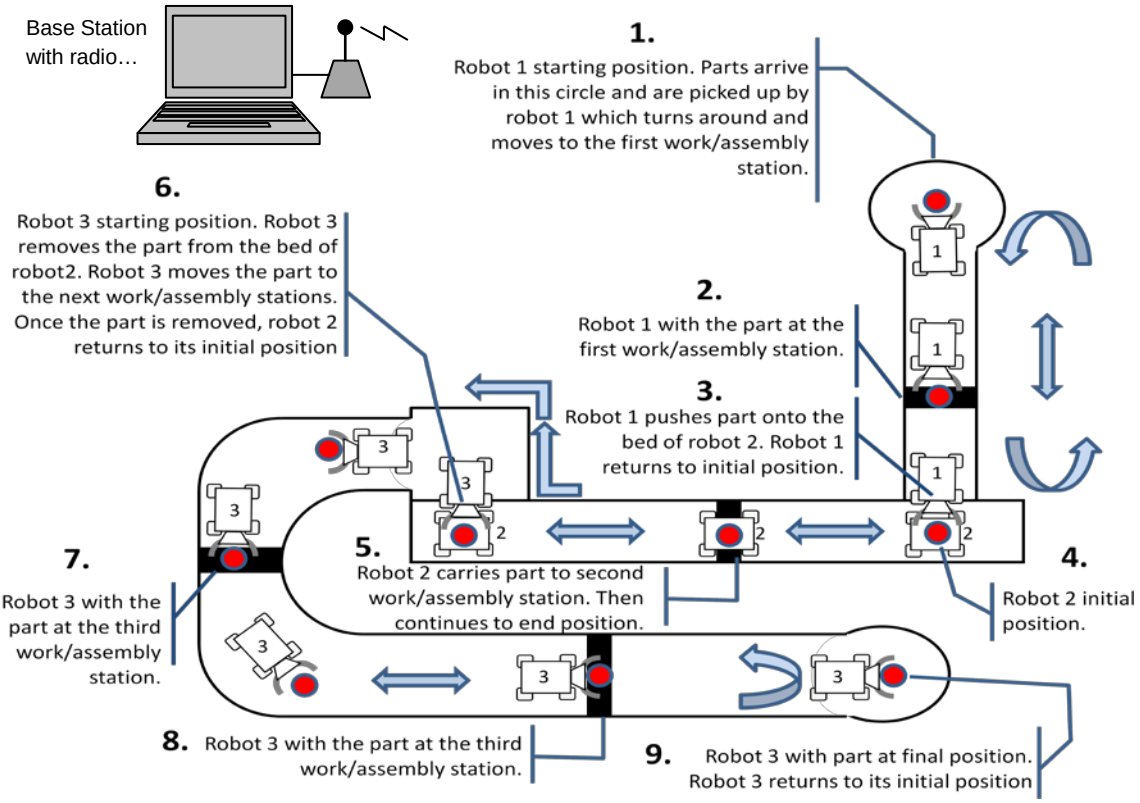
Example Assignment – This assignment is designed to help students practice with basic notation and analysis techniques presented in class.

ASSIGNMENT 1

Protocol Modeling

Assignment Description:

This is an individual assignment. Assume that you are a software engineer responsible for implementing a protocol for communication to coordinate the activities of interoperating robots used in a manufacturing application. The robots are equipped with a simple radio that enables the bots to send and receive information from a base station. Robots are not permitted to communicate with each other and only may communicate with a base control station. All the robot radios are on the same frequency, so bots cannot transmit data at the same time. This also means that when the base station transmits, all robots receive the message. The customer has provided the description below to describe how the robots work together to move parts around a factory assembly line using the drawing below:



Other Assumptions:

- all messages are delivered accurately – e g. no need for CRC checks or retransmits
- the bots will be in their proper initial positions
- robots “know” how to do their specific tasks: pick up parts, move from initial positions to final positions and return to initial positions
- robots must be instructed to begin their tasks, return to initial positions, or wait as necessary

Your assignment is to analyze the protocol and communications requirements for this application and create a precise specification for how the robots will communicate with the base station. Using notation techniques presented in class, precisely describe the communication protocol that will satisfy this application. Make sure that you describe the message alphabet, the meaning of the messages, how robots discern messages, control mechanisms (e g. end of message, message IDs, etc.), and the overall dynamic behavior of the system. Do not worry about code, data structures, or other design or implementation details. What is needed is a precise description of the communication protocol.

Deliverables:

Write a paper that clearly describes your solution. Please be thorough, but concise in your answers and discussion. Note that grammar, good style, and format counts. Any reasonable formatting style and structure is acceptable.

Grading:

This assignment is worth 20 points as follows:

- **Correctness and completeness:** 15 points. Your solution includes all aspects of the problem and is correct in its representation.
- **Quality of report:** 5 points. Quality in terms of grammar, format, conciseness, thoroughness.

Appendix 3

Example Assignment – This assignment is designed to exercise the students understanding of ADTs, data structures, and separation of concerns concepts presented in the class. While not as practical as other assignments, students often indicate that this assignment is fun as well as instructive. Note that students are required to perform and utilize analysis techniques presented earlier in the course.

CMU 17-630 Computer Science for Practicing Engineers Fall 2011

ASSIGNMENT 3

The Virtual Rube Goldberg Machine

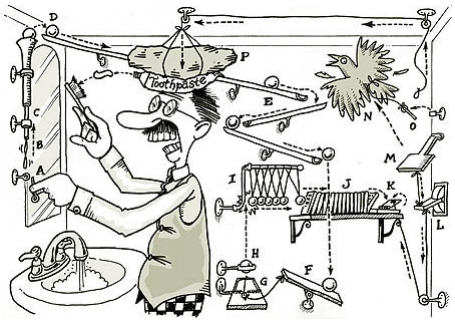
Introduction:

To this point, we have discussed Linked Lists, Queues, Stacks, Hash Tables, and Trees in detail. This exercise is designed to give you some practical, hands-on practice in putting together a few of these data structures.

References: Aho et al., Harel

What is a Rube Goldberg Machine?

In this assignment you will build a virtual Rube Goldberg machine to practice with some of the data structures we discussed in class thus far. A Rube Goldberg Machine is a complex device that performs simple tasks in indirect and convoluted ways Reuben Goldberg was an American



engineer who changed his career to cartoonist (only in America would you do this ☺). He was famous for his political cartoons and satire. However, the works which would lead to his lasting fame involved a character named *Professor Lucifer Gorgonzola Butts* shown in the cartoon on the left. In this series, Goldberg drew absurd inventions that would later bear his name: *Rube Goldberg Machines*. Rube Goldberg received the Pulitzer prize for his satire in 1948. Today there are several contests around the world known as Rube Goldberg contests, which challenge high school students to make complex machines to perform a simple tasks. These contests are not only fun, but they challenge students to practice fundamental engineering, physics, and mechanical principles as well as encourage creativity. They must utilize a specified number of various simple machines, using common junk and household items to perform absurdly simple tasks such as cracking open and egg, opening a door, ringing a bell, and so forth. You can check out a real Rube Goldberg machine at http://www.flixxy.com/best-rube-goldberg-machine.htm. This is one of the best I have seen – *enjoy it!*

Assignment Description:

If the MechEs and Physicists can do it, so can computer scientists (without getting our hands dirty). Your job will be to create a virtual Rube Goldberg Machine with ADTs. The ADTs that you will use will include the queue, stack, binary tree, heap, and many of the associated operations on these ADTs. One way to approach this assignment is to think about the functionality from the user's perspective. Certainly this is a reasonable place to start, however you should think in terms of the application being composed of ADTs and operations upon them. Note that the ADTs will contain the data and with each ADT we studied there were lots of operations upon them. Think as if you were a small team and that you had a few engineers creating the ADTs, and others that would compose the application using these ADTs. Remember the adage that I keep referring to in class, "*I don't want to know how to build a color TV set, I want to watch Sesame Street.*" This is the mind set we should have as the engineer responsible for composing the application from the ADTs. Those building the ADTs should worry about the details of the ADT structures and operations in isolation from other ADTs and in isolation from application specifics as much as possible. There should be as little dependency between the ADTs and the application as possible – you will have to use your judgment. In this assignment, you will play the role of ADT builder and application builder. This is an individual assignment.

Your Virtual Rube Goldberg machine will do the following:

- You will read from a provided space-and-comma-delimited text file list of data in the following format:
    - first name (30 characters)
    - last name (30 characters)
    - age
    - day of birth (mm-dd-yy)

  An example would be as follows: Harry Bovik, 49, 1960
- Your program should support any number of entries. As the data is read from a file, initially store the data in a queue.
- Dequeue each element from the queue, print each item, and requeue each item. Ask the user to press any key to continue the processing.
- Next, you will reverse the order of the data in the queue by dequeuing each element and pushing them onto a stack. Once all the data is dequeued from the queue and pushed on to the stack, pop off each element of the stack and re-queue each element back into a queue ADT, reversing their order in the queue. Once completed, dequeue each element from the queue and print each item for the user - ask the user to press any key to continue the processing. Make sure you requeue each item to preserve the queue.
- Next, dequeue the elements from the queue and place them into an unordered binary tree. Add the items into the tree ADT in the order they are dequeued, adhering to the binary tree shape property.
- Print the contents of the tree in pre-order and ask the user to press any key to continue the processing. Print the contents of the tree in post-order and ask the user to press any key to continue the processing. Ask the user to press any key to continue the processing.
- Move the data from the unordered binary tree into a linked list ADT using an in-order traversal of the tree. Print the contents of the list and ask the user to press any key to continue the processing.
- Sort the contents of the list using a quick sort in ascending order. Print the contents of the list and ask the user to press any key to continue the processing.
- Allow the user to interactively enter another name, age, and birthday. Add this to the list in the proper location to maintain the sorted order. Print the contents of the list and ask the user to press any key to continue the processing.
- At this point you are done with the processing.
- You will use C or C++ for this assignment – you may use any compiler you like. Please do not use any prepackaged, open-source, or commercial libraries for the list, stack, queue, or tree ADTS. Maintain a clean separation of these ADTs. Make sure that you handle all possible errors within the ADTs and the end-user application. Your code MUST BE WELL STRUCTURED AND DOCUMENTED! Ensure that you include headers and comments in your code, properly indent code blocks, and otherwise use good coding practices. If we can't read and understand your code, we will not grade it.

Of course your program must work and it must be easy to follow what the program is doing. The user must be given simple, easy to follow feedback in order to follow the progress of the operation. You will also be graded on how well structured your ADTs are, how well you have maintained a clean separation of concerns between the various ADTs and the application itself, and how the overall structure of the program. Note that a working program alone is not enough to get a good grade on this assignment – structure and separation of concerns and code quality matter as much as functionality.

What to Turn In:

Post your source code and executable code (.c and .exe) on blackboard – note that we will not recompile programs. We will not install any special environments to run your code. We expect to run your code from a simple command window and observe the operation. All I/O from the user to the application shall be from the keyboard. We will run your program and observe its operation – if we cannot run your program, you will lose points depending upon the reason for the problem (instructor's discretion). Assume that the target OS platform is the Windows XP or better.

Deliverables:

1. Report that includes:
    - Describe the design of your various ADTs, the data structures you selected, what information and operations you hid and your reasoning for your decisions.
    - Describe how to use your program.
    - Show the analysis of your application in terms of correctness and complexity (performance and memory) as described in class. You may utilize any of the notation and analysis techniques presented in the class.
2. Demonstration program.

NOTE: Please be thorough, but as concise as possible in your discussion. Note that grammar, good style, and format counts. Any reasonable formatting style/structure is acceptable.

Provide your source code, data files, and executable program.

Put all of these materials (described above) into a zip file and turn it in electronically (blackboard or email). Use Windows file compression or Winzip to compress your files. Include a "readme" file in your archive that explains the contents of your archive. Name your archive as follows: A3+YourFamilyName. So my assignment archive would be named: A3+Lattanze.

Grading:

This assignment is worth 50 points as follows:

- **Report content:** 20 points total for content.
- **Implementation:** Correctness of the application for a total of 20 points. Note the design and structure of your API, application, and how well it matches your design is an important part of the evaluation.
- **Ease of use and write-up quality:** Your application must be easy to use – easy to enter input, easy to read and understand your output (5 points). The write-up will also be evaluated in terms of grammar, format, and readability will also be considered (5 points).

Appendix 4

This is an example of one of our capstone projects. Note that students are required to pull together many of the concepts in the course from notation, to analysis, to data structures, to design.

Capstone Project

Specialized Search Engine

Overview

In this assignment, you will build a Specialized Search Engine that will search the text files of information archived on the site: *http://textfiles.com*. As a starting point you will be provided with a basic web crawler engine written in Java to illustrate basic web crawling mechanics.

Textfiles.com Background

The file archives on *textfiles.com* are collection of documents from the 1980-1995 timeframe. The archive contains a variety of esoteric topics and related articles in good old fashion text files. The documents are typically maintained in the ASCII document format. The file names tend to follow the 8.3 file naming standard, also called "short filename" standard or SFN. This convention was used by old versions of DOS and Microsoft Windows prior to Windows 95/NT 3.51. Filenames following the 8.3 standard have at most eight characters, optionally followed by a period "." and an extension of at most three characters. Typically file and directory names are case-insensitive. In addition to the article document files, the *textfiles.com* website maintains an index of document genre or topic area. Within each topic area there are lists of articles. However, there is no overarching search engine.

Project Requirements

In this assignment, you will build a Specialized Search Engine that will search the text files on Textfiles.com. The goal is to create a fast search engine that can search *textfiles.com* in two ways: 1) for document genres, or 2) search the documents themselves for specific key words.

A basic web crawler engine written in java to illustrate basic web crawling mechanics is available on the course Blackboard site. The application is implemented in Java. The entire basic web crawler source code is in a single source file named: *CrawlerEngine.java*. You should make sure that the latest version of the JDK is loaded on your system. To compile the example crawler, download the source file into a local directory and type the following in a command window:

javac CrawlerEngine.java

You may get some compilation warnings depending upon the version of Java compiler you are using, but these can be ignored. To run the application, type the following:

java CrawlerEngine <starting URL> <number of pages>

This program illustrates the basic concepts associated with crawling through web pages. It is strongly recommended that you become intimately familiar with the example application – it is small, so it should not take too long. This will start at <starting URL>, each file and look for URLs, and crawl each of the URLs it finds in each file. It will look <number of pages> deep for URLs. As a test for the CrawlerEngine demo try using http://textfiles.com as follows:

java CrawlerEngine http://textfiles.com 10

This will crawl the *textfiles.com* web pages (that permit crawling) and list them as they are found. Note that if you increase the depth, the application may seem to "hang" at the end of the application, but it is looking at each and every file for URLs at the specified depth. This gets close to $O(n^X)$ – so you want to be careful with the depth of your searches and keep well within the bounds of the *textfiles.com* web pages.

Your task will be to create an application that will do two different searches of the *textfiles.com* archives. The user will provide a search string, and then they will select one of two searches: and 1) *index search* and 2) a *document search*. The *index search* will look through the topics index on *textfiles.com* for any topics and the URL to the index topic that match the string. The matching indexes will be listed for the

user. The *document search* option will look through the documents to search for matching strings within the document. If there is a match the document will be listed on the screen with the URL to the document.

Teams will have to constantly balance performance and accuracy in order to build the most efficient application as possible. It is important that your search engine is as fast as possible. Think in terms of strategies to maximize the performance of your search engine such as: concurrency, caching, hashing, and so forth, although you don't have to use these. However, you will have to aggressively manage resources (especially memory) to; maximize performance, prevent buffer overflows, or hangs due to resource deadlocks.

Project Constraints and Assumptions

Safe and Courteous Crawling

Crawlers can be dangerous and it is easy to be mischievous with them. Therefore, teams must adhere to the *robot exclusion protocol* (http://en.wikipedia.org/wiki/Robots.txt) for all crawlers built as part of your search engine. An example check for robot exclusion has been built into the example crawler engine. Obviously mischievous behavior conducted as part of this course will not be tolerated.

Crawling Limits

By default your crawler has an upper bound of 20 on the total number of files to be downloaded and checked for URLs to explore. This is a lot and may dramatically impact the performance of your search engine without improving the search. The team is free to establish and tune this variable to optimize between accuracy and efficiency – however, do not compromise thoroughness of the search.

URL Format

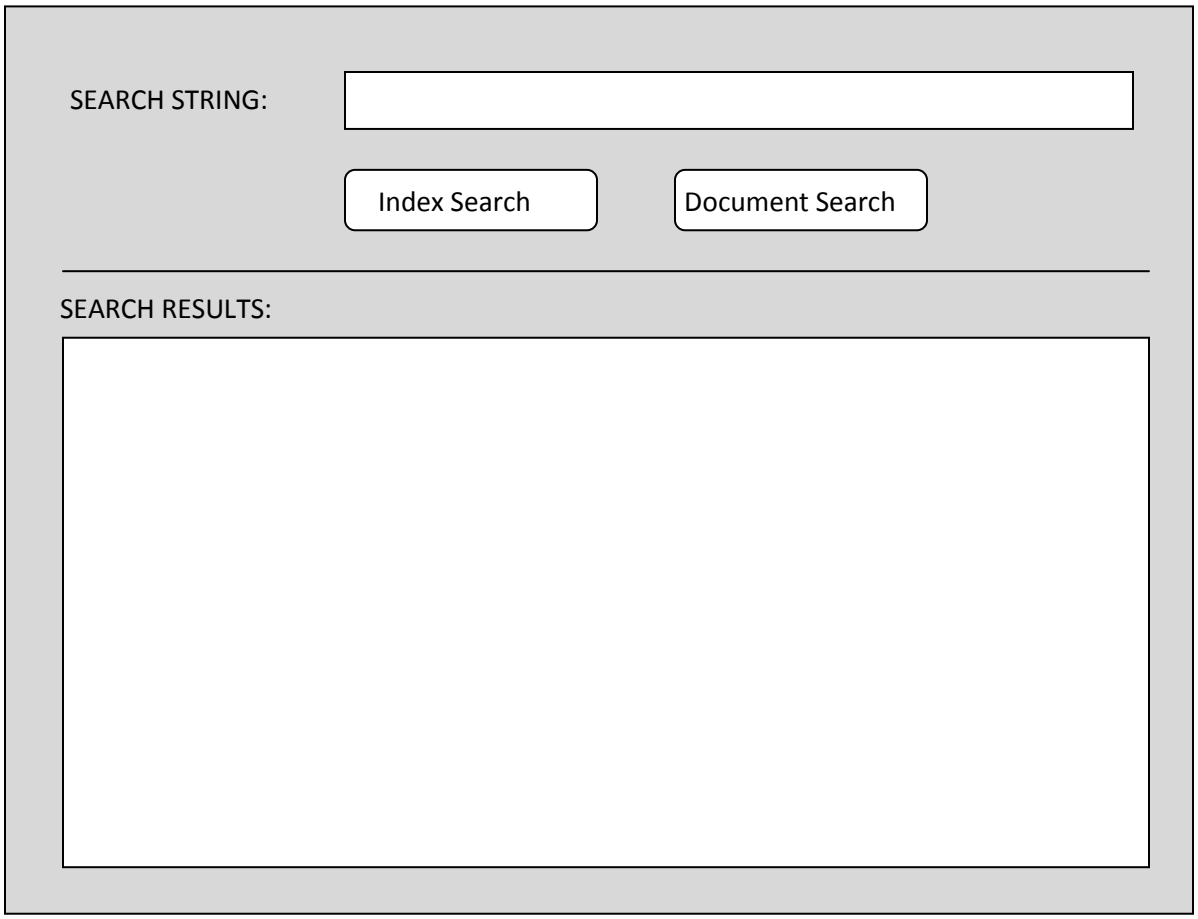
On *textfiles.com* you can assume that things are simple, and that HTML files have URLs listed the form < a href="URL">. You don't have to worry about the case where this appears in the middle of some other HTML tag (such as a comment). Ignore any others forms. Again, refer to the example crawler engine provided.

Platform, OS, Language

You may implement the application on a Windows or Mac system, however you must use the Java language (version 1.6 or later) to promote portability - we will test your application on a Windows based machine. If you are unfamiliar with Java, I have included a Java Short Course on the web page.

Standalone Application

Your application will run on a standalone personal computer that is connected to the internet. You must work within the limitations of the computer you have at your disposal and you many not violate the platform, OS, or languages constraints established above. While the example crawler engine is based on a command line interface, you must build a Graphical User Interface (GUI). While there is no GUI specification, a minimal GUI might look like this:



Here the user enters the search string in the text field labeled “SEARCH STRING” and then pushes the “Index Search” or “Document Search” button. Once one of the buttons is pressed, the search begins. MAKE SURE YOU INDICATE PROGRESS SOMEHOW – inform the user that the application is busy. If possible, show the percent complete,… if you deem that this is not possible, say why in your write up. However, you still must provide an indication that the application is busy with a search. To construct the GUI (actually the entire application) you may use the NetBeans IDE or you may build the application by hand. You may be as elaborate as you like with the GUI, but it must provide the basic functionality listed above.

Project Deliverables

Your team will produce a document that includes your analysis, design, and operation guide. In addition to this document, you will also need to turn in all the application source code and the executable program. Each element is described below:

You should provide a comprehensive analysis of the problem including:

- Functional Analysis: The team should clearly articulate the required system functionality including any clarifications and assumption.
- Computability and Performance Analysis: The team should analyze and estimate to the best of your ability the search engine’s performance using big O analysis.
- Correctness and Robustness Analysis: The team should attempt to anticipate the kinds of errors that might be encountered during operation and think in terms of how the system should react and behave in the presence of these errors.

In addition to this analysis, you must describe the overall design of the application and the general approaches the team used to design and build the system.

- Describe any key data structures that your team utilized to structure the application and why.
- Discuss how you addressed the quality attribute requirements of performance, correctness and robustness – describe how portioned and structured your system to meet these quality attributes and any others of concern .
- You do not have to use any particular notation for the design – we will not evaluate the correct usage of a notation, but you must clearly articulate the design with pictures and prose. It must be understandable.

The final element of your document will be an operation guide that describes how to set up and run your program on a user's machine. Make sure that you clearly describe how to install and set up the application. The only safe assumption that you should make is that the latest Java Runtime Engine (JRE) is installed on the user's computer.

Project Deliverables

This project will be worth 100 points, allocated as follows:

- Operation (50 points)
    - correct operation of the application
    - ease of use
    - ease of installation
- Document (40 points)
    - format, content, completeness, conciseness
- Quality (10 points)
    - code authorship and coding practices (comments, clarity, etc.)
    - how well the code structures adhere to the design