Re-Engineering the AlgorithmA Project for Long-Term Maintenance

Willie James
Defense Services
Environmental Systems Research Institute
Redlands, CA
(909) 793-2853
wjames@esri.com

Phil Lucas
Department of Computer Science and Engineering
California State University, San Bernardino
(619) 750-9482
phlucas@acm.org

John O'Connor
Mission Systems
Northrop Grumman
Redondo Beach, CA
(310) 813-0120
john.o'connor@ngc.com

Arturo I Concepcion
Department of Computer Science and Engineering
California State University, San Bernardino
(909) 537-5330
concep@csci.csusb.edu

Abstract

The AlgorithmA project has been in existence since 1991 and is used as an on-going project in the software engineering class taught at CSUSB. In 1998, the project was first implemented on the Internet using Java. In 2007, the maintenance of the project was a big challenge because of the size of the project and the complexity of the architectural design. This paper talks about the decision process of the software engineering class that led to the re-engineering of the entire AlgorithmA project, which is a situation that actually happens in a real software company. The observations and experiences of the project manager, the team leader of the maintenance team, and the team leader of the Java team that implemented the re-design of the project, are discussed in this paper. In 2008, the project was continued by forward engineering the original functions of AlgorithmA. The re-design followed the Model/View/Controller (MVC) model and using the Observer and Factory patterns, made the AlgorithmA project maintainable and extensible for many more years to come.

Categories and Subject Descriptors D.2.9 [**Software Engineering**]: Management - software management, software development, programming teams, software maintenance, software process. D.2.11 **[Software Engineering]**: Software Architectures - information hiding, patterns.

General Terms Management, Documentation, Design, Human Factors.

Keywords Software engineering, software process, software architecture, management.

Copyright is held by the author/owner(s).

OOPSLA'08, October 19-23, 2008, Nashville, Tennessee, USA.
ACM 978-1-60558-220-7/08/10.

1. Introduction

AlgorithmA (Algorithm Animation) is a Web-hosted application to help computer science students learn algorithms through visual demonstration of the data structures and step-by-step execution of the corresponding algorithm written in pseudocode. AlgorithmA is being developed as an on-going project in the upper-division software engineering class in the Department of Computer Science & Engineering, California State University at San Bernardino (CSUSB). The course has been conducted annually for 17 years. It is offered in the winter quarter which consists of 10 weeks of class with one week of finals. A significant challenge for this project has been the difficulty in maintaining quality control over many years when the development team is completely changed with each class enrollment.

There is a long standing problem of instructors teaching what software engineering is but lacking the methods of how to teach it effectively. The AlogrithmA project is used as the project-centric course whose students maintain the project written by previous students who have undergone the course.

Shown in the Appendix is the history of the AlgorithmA project from 1991 to 2008. From 1991 to 1996, the project was implemented in the C language, using different graphics packages such as cursor graphics, XGKS and SPHIGS, and implemented in a structured approach. In 1997, the first version in Java was implemented and for the first time AlgorithmA, although not complete, was accessible from the Internet. This was also the beginning of the use of the

object-oriented approach in the implementation phase. The first complete Web implementation of AlgorighmA was done in the following year, 1998.

The AlgorithmA project has been described in four papers over the past ten years. The first paper [2] describes the software life-cycle/software process that was used in the development of the project. The second paper [3] talks about what is the goal and purpose of the project and the three user interfaces: animation, walk-through, and the authoring system. The third paper [4] discusses the use of the object-oriented paradigm in the project development and the fourth paper [5] shows how the software engineering course is taught as a mock software engineering company.

In this paper, we discuss the difficulties of maintaining the project, which has grown to over 161,000 lines of Java code, excluding PHP and HTML codes. In 1998, we had the first implementation of AlgorithmA on the Web and so there is still Java code in the current project that is as old as 1998! This is the experience of the software engineering class of 2007 which had to deal with the dilemma: continue the development of the current project or scrap the current project and re-engineer the design and its implementation. This paper discusses the software process and decisions that led to re-engineering the project and the comparison of the previous 2007 version to 2008 AlgorithmA in terms of maintenance. Re-engineering of the AlgorithmA project happened in 2007 while forward engineering happened in 2008.

2. Project Organization

The AlgorithmA 2007 project was organized into three main functional groups, with the management team serving a supporting role to all of the other teams. The teams were as follows:

- The Quality Assurance Team
- The Java Implementation and Design Team
- Supporting teams

All teams in the software engineering class, including the management team, are composed of students playing a specific role in the software development process. The management team, specifically, is composed of a project manager and several staffs that support the role of the project manager. The instructor plays the role of the CEO.

Each of these teams had a specific role in the development and maintenance of the project. However, that role was not completely realized until the middle of the first iteration of the project. The separation of the project into two iterations made possible a re-allocation of resources. The software engineering class follows the iterative approach in the development of the project.

2.1 The Quality Assurance Team

The QA team was originally created to implement and design functional and unit tests for the software being developed by the Java implementation and development (JID) team. However, as the first iteration of the project commenced, the role of the team was modified to focus on legacy system maintenance. This allowed the continued use of the previous system while the new implementation was being designed and developed. The QA team was also tasked with becoming familiar with the legacy system in order to aid in the documentation effort.

2.2 The Java Implementation & Design Team

The JID team originally started as a coding and development team, however their role in the project transformed during the first iteration of the project into research and development (R&D). The JID team was given the task of implementing a new system using the functional requirements defined by the CEO (professor) and an architecture designed by the Software Architect and his assistant. The Software Architect and Assistant Software Architect were tasked with creating a scalable and maintainable design that met all of the requirements of the project.

The decision to have the JID team begin re-engineering the AlgorithmA project was reached after collaboration with members of the QA team who had performed an analysis of the existing AlgorithmA software. It was decided that the complexity and redundancy of the existing code was incompatible with achieving the goals of the management team. The re-engineering would be based on a strict adherence to the MVC architecture using well known design patterns. This serves the function of ensuring that the new system stays maintainable and scalable. The MVC design was completed by the Software Architect and Assistant Software Architect, and the implementation was started by the JID team. The MVC architecture was fully implemented in the second iteration of the project, and the final version is the one that future software engineering classes will use as a development base.

2.3 Support Teams

There were several support teams that were created as a way of meeting the project requirements. One of these requirements was to create a better documented system and to attain a Capability Maturity Model (CMM) level of 3. The Documentation Team was created to oversee the documentation of each facet of the project and to provide a means of accessing that documentation. The final versions of all documentation were distributed and published by the Documentation Team.

Since the project required complex servers to be available, with programs such as Subversion (SVN), Apache,

and other Web software installed, a Server Team was created to oversee all server-related issues. In addition to its principal role, the server team created a Wiki and migrated the previous versioning system to SVN. The Wiki turned out to be a valuable project-wide communication tool and was used for both inter- and intra-team communications, as well as the project-wide publishing of documentation.

The external interface of the project consists of a Web-based Java application. A Web Team was created to re-design and improve the external user interface. Additionally, as part of the re-factoring and re-engineering of the system, the Web team had to discover different methods of implementing the completed Java class files into the external interface. As a non-functional requirement and to ensure accessibility, the team had to design Web interfaces that were XHTML and CSS compliant.

3. Software Maintenance

The original organization had a quality assurance team, responsible for designing and implementing a QA plan. Once the idea to re-engineer the product was implemented, we knew it would take quite a bit of time before the final architecture was approved and implemented. The software requirements specification (SRS) only included modules that needed repair or completion (no new modules) and the QA team became the software maintenance team.

3.1 Team Organization

It was decided by the management team that there would be two iterations, one 6 weeks in length and the other 4 weeks. The first iteration was slightly longer because it included the prerequisites of building the company and staffing the various teams. Before the maintenance team could begin work on the system, the server team had to be established to give secure access to the source code for the programmers. Additionally, it was decided to convert the current CVS system to SVN to gain additional functionality. Finally, the first SRS needed to be developed before any programming could be started. During this time, most of the class worked independently simply running the existing system and cataloguing bugs or discrepancies they thought should be addressed. Advice from former students suggested we ignore any existing bug reports. Their argument was that with their constrained time limits, most students put their emphasis on the SRS and few gave the bug reports much notice. Also, due to different descriptions or different ways to invoke the same bug, there were many bug reports that were redundant. Finally, the SRS enhancements often required changes that fixed existing bugs without the programmer realizing it, so many of the bug reports were obsolete. We took the former students advice and started with an empty Bugzilla database. It seems likely that this logic has been used in previous classes as well, which eliminate the ability to do any analysis on the historical trends of bugs.

3.2 Issues Faced During Maintenance

Each student filled out a checklist of software skills, and the few students who categorized themselves as knowledgeable about Java were put on the development or design teams, leaving the maintenance team with few Java skills. For this reason, we chose to implement Extreme Programming (XP) pairs in which two programmers shared a module; while one typed the other gave suggestions. Pairing students like this allowed them to bounce ideas off each other while avoiding most of the roadblocks that can arise when working alone. Eighteen people on the team meant 9 XP pairs. Coincidentally, the SRS chosen for the first iteration had 9 modules. Each team chose a module and was expected to make whatever enhancements were required by the SRS as well as correct the catalogued bugs.

The perfect division of labor of 9 modules between 9 XP pairs worked well on paper. However, it left only the team leader to do anything beyond the programming. This became a problem when two issues needed resolving simultaneously, and at the end of the iteration when the team leader was responsible for 18 individual software engineering student performance evaluations. For the second iteration, the entire team was divided into 3 sub-teams, each with its own lead. However, the XP concept proved to be so popular, that the sub-team leaders still divided their teams into XP pairs and work on iteration 2 continued as in iteration 1.

Unlike a real company, our mock company was completely new. With the exception of some advice from former students, there was nobody with a complete understanding of the existing system - the original architects graduated years ago. Each team had to discover independently how best to proceed. When confronted with building a new module, most chose to copy code from an existing module and modify it appropriately. This resulted in an “inheritance via cut and paste” system. While modules such as the Queue, Dequeue and Linked List were practically clones of each other, they actually shared no code, so the lines of code count (LOC) climbed excessively with each new module. The previous system is approximately 161,000 lines of Java code. There are 66 modules, and not surprisingly, many modules contain very similar LOC counts. For example, of the 66 modules, 40 have between 1000 and 2500 lines of Java code.

Like a real company, the pressure in the class was real. To get a decent grade, the team needed to meet the requirements of the SRS by the deadline, and with so much time spent upfront building the company, the pressure on the teams was to complete the 6-week iteration in 2 weeks. The second iteration, while 4 weeks on the calendar, also needed

an SRS before programming could begin, so was also about 2 weeks in length. Standard coding practices and proper methodologies were often overlooked if there were faster ways of doing things. While it might have been possible (and desirable) to modify an existing class to serve a new module, fear of breaking the existing use of the class meant it was often faster to create a new class than debug the old, increasing the existing code in order to meet the deadline. Additionally, the programming style of the original program was often ignored and new enhancements were written in the style of the new programmers. A single module could contain several different styles. A program that was originally written in an MVC format could have "view" methods added to the "container" module, for example. This latter issue was most likely the result of the time pressure, but it was exacerbated by the limited Java experience of many of the programmers.

It should be completely expected for programmers to make "rookie" mistakes when programming Java (or any language) for the first time. While a good QA team should catch these types of errors, the team leader succumbed to the same pressure as the programmers - produce or die (in this case, have something presentable or get a bad grade). The QA was left to the individual teams who were responsible for the error in the first place. Therefore, these types of bugs have been accumulating in the product for almost 10 years! To make matters worse, Java continued to develop and evolve, with new features added and old features deprecated. The current release of AlgorithmA has so much deprecated code that it will not compile with a Java compiler newer than version 1.4.

3.3 The Decision to Re-Engineer

From the beginning, it was obvious that the current system was at a breaking point. Almost all of the causes of software aging that David Parnas identified [7] applied to AlgorithmA. The lack of movement (deprecated code), rookie programmers with a lack of proper QA, no access to the original system architect and limited time constraints all contributed to a system that only "mostly" worked. New modules would continue to be cut and paste clones of other modules, propagating known bugs (as well as the unknown bugs) into each new module. Focusing on bug fixes would not solve the problem of the lack of proper inheritance, multiple programming styles or lack of documentation. Refactoring was considered and has been shown to improve source code [9], but with so many different program styles implemented, each program would need to be refactored independently. Maintaining consistency, then, would still be a daunting task.

It was therefore decided that a completely new architecture was required. An architecture that considered not only the SRS, but allowed for future maintenance and growth while considering the lack of experience of future programmers.

4. Re-Engineering

What is software re-engineering? According to Linda Rosberg [8], Engineering Section head at the Software Assurance Technology Center for Unisys Federal Systems, "Re-engineering is the examination, analysis, and alteration of an existing software system to reconstitute it in a new form. It involves re-design and re-implementation of the software but keeping the original functions for which the software was developed." The job of re-engineering the AlgorithmA Project for 2007 was to be the job of the software architect and his assistant along with the Java team.

4.1 Team Organization

The architect and his assistant were in charge of creating the base classes and interfaces that would be the groundwork for implementing the MVC pattern along with the Observer and Factory patterns. They would then pass along the results of their work to the Java team. The Java team would design classes and write code that would implement the higher level interfaces. The architect would review the work of the Java team and make sure that the coding adhered to the appropriate design pattern. The Java team consisted of a team leader and eight developers.

The Java team leader was responsible for assigning work to each team member. He was to consider the experience and skill level of each team member as he assigned the various tasks that needed to be performed. The team leader was also available to assist the team members via email, phone calls, a "Java code camp", and regular class lab time. Most of the time assignments were handed out to individual team members but it was not unusual for team members to work in tandem in carrying out a particular task. The team members would submit their code and both the Java team leader and the software architect would review it and suggest changes if they felt it was required.

The Java team leader at two intervals during the quarter would do a performance evaluation on each team member. Likewise each team member on two occasions during the quarter would also evaluate the work of the team leader. All evaluations were passed on to the professor in charge of the class. He would then meet individually with each team member, including the team leader, and discuss the results of the evaluations.

4.2 Training the Java Team

While all the members of the Java team had previously taken courses in C++, only a couple of team members had experience with Java. It was decided that to help the team members become more productive, a Java code camp would be setup. The camp would assist team members in

transferring their C++ coding skills to Java. Some of the topics that were to be covered were: Java's Single Inheritance Structure, Implementing Java Interfaces, Applets vs. Applications, Java Event Dispatching, and Java's Error Handling Mechanism. In addition to these general topics, the camp would be a place, outside of class, where team members could ask questions and get answers regarding any development issues they were facing.

The code camp would be held Monday thru Thursday for five consecutive weeks. The daily sessions ranged from one to two hours. The student developers met in the computer labs during open lab sessions. Attendance was on a strictly voluntary basis and it ranged from one to five team members per session. Even though topics were pre-planned the structure was quite informal. Different team members were free to ask questions about specific topics that were of interest to them. While the team leader was primarily responsible for instruction, team members also worked together to help one another with issues they might be having.

4.3 Implementing the MVC Design Pattern

MVC is concerned with the separation of software components so that they may be changed, altered, or improved more easily. Components are grouped into the three categories based on how they function in the application.

Steve Burbeck notes that "the view manages the graphical and/or textual, the controller interprets the mouse and keyboard inputs from the user, commanding the model and/or the view to change as appropriate, finally, the model manages the behavior and data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller)." [1]

As noted above the AlgorithmA project had evolved into hundreds of classes and tens of thousands of lines of code. A single class could be dealing with both drawing animations to the screen and also be responsible for creating and manipulating data. Moving the architecture to the MVC pattern would be a giant step to organizing the code. All classes that dealt with the view that the user would see would be logically and physically grouped together, the same with the classes handling the data. Then all the interaction between the data and the view would be managed by the controller classes and they too would be grouped together.

Figure 1 is the class diagram that became the basis for development in the new re-engineered AlgorithmA. The diagram shows the class structure and how the classes fit together in the context of the MVC, the Observer, and the Factory patterns. In Figure 1 there are five classes that comprise the view. MasterView is the superclass and the other four are its subclasses.

4.4 Implementing the Observer Pattern

The purpose of implementing the Observer pattern was to ensure that the classes were loosely coupled. From the implementation of the MVC pattern there were classes that represented the view, the model or data, and classes that controlled the interaction between the view and data classes. To make the controller more efficient the Observer pattern was used as a bridge between the data and the view. The view classes could subscribe to be notified if the data changed. The data or model would create a notify event if it changed, there was no concern from the perspective of the data classes about how change would affect the other classes. When data changed, the view would be alerted because it was an observer of the data's notify events. The view could then take whatever action was deemed necessary to react to the change in the data. This action could range from updating the graphical user interface components, speeding up the animations, or take no action. The key classes comprising the observer pattern are designated in Figure 2 with the label Observer Pattern Classes. The interfaces, Subscriber and Publisher, are the super-classes for the classes that are observers of events or those that publish events.

4.5 Implementing the Factory Method Pattern

The factory method pattern is a design pattern that allows for an unknown class to be created from a superclass [6]. This creation process happens similar to the "abstract factory" pattern but instead of relying on a separate factory class the existing object of a substantiated class is able to use one of its existing methods to return a new class that implements one of its own interfaces [6]. Users of the application would interact with Java applets embedded in Web pages. The Factory pattern enabled the use of a single Web page entry point and a single Java applet. The controlling applet would be passed parameters from the Web page. Based on those parameters, at runtime, the applet would call the appropriate method to create the classes needed to carry out the request. In Figure 1 in the section labeled View Classes we see the class MasterView. This is the class that contains the methods needed to do the dynamic class creation at runtime. This was a remarkable improvement over the previous approach that used multiple Web pages and distinct separate applets to handle each user request.

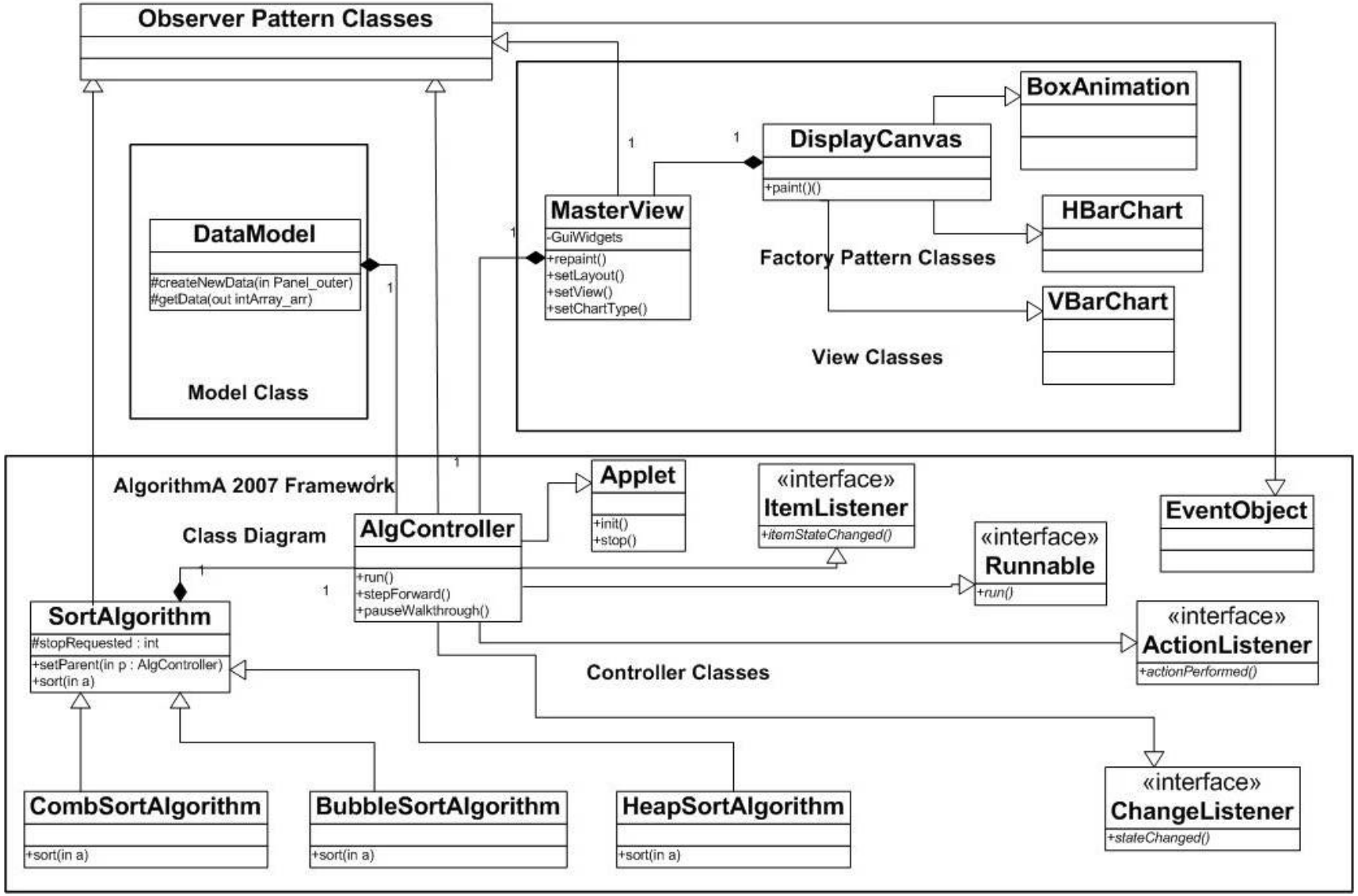

Figure 1. MVC and Factory Pattern Classes

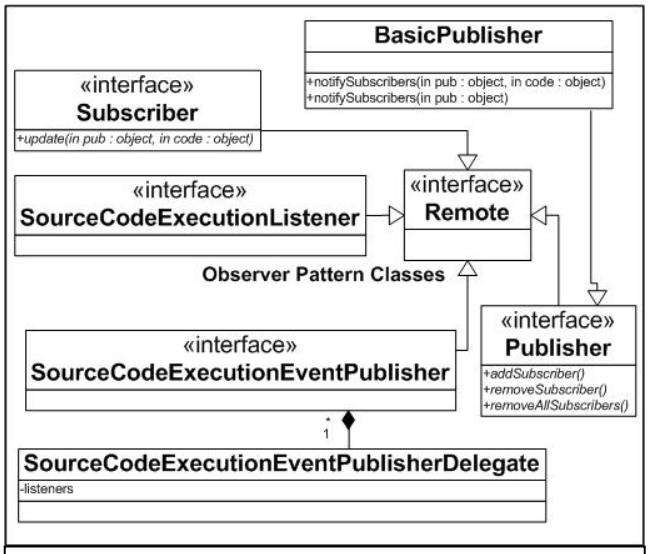

Figure 2. Observer Pattern Classes

4.6. MVC Benefits

Figure 3 shows differences in the number of lines of code (LOC) and the number of files used for many of the algorithms for the 2007 and 2008 versions. 2007 introduced the new architecture, but the class of 2008 was the first to actually implement it. In most cases, the changes in complexity were marked. Many algorithms, such as the Dequeue, Linked List, Queue and Stack dropped to a size 1/3 of the original, in both lines of code and number of files. Comparing the *quality* of the new code to that of the old is difficult to judge, and it is entirely possible that the new code is more complicated to understand than the previous. While this may be true (*quality* is a very subjective concept), the new system architecture is fully documented and the same architecture has been successively applied across several types of algorithms. Anyone understanding one algorithm should be able to assist others. It will be up to future managers to ensure individual teams do not deviate but remain committed to the current architecture. Unfortunately, historical perspective suggests that deviations *will* occur and the system will eventually grow in complexity, requiring another reengineering iteration.

5. Lessons Learned

The students gained a better appreciation of what it means to work together as a team. Teams were put together by the students who were assigned roles as managers. As a result, students found themselves having to work closely with students that they did not know very well. They then had a team leader appointed over them and they had no voice in that decision. Hence there were personality conflicts between team members and team leaders, yet the work had to be done. There was one opportunity granted to change teams but surprisingly only a couple of students took advantage of the opportunity. Communication is the key to working together and they realized this when working within their teams, working with other teams, or working with the management team.

	2007		2008	
	LOC	# Files	LOC	# Files
Data Structures				
Dequeue	3408	31	1109	3
Heap	1902	15	385	5
Linked List	3604	31	1115	3
Priority Queue	1715	12	748	3
Queue	3076	31	672	3
Stack	3880	33	1200	6
Recursion				
Factorial	1690	22	337	3
Fibonacci	1624	22	353	3
Inorder	2002	22	524	3
Maze	1427	10	627	4
Post-order	1657	24	541	3
Pre-order	2050	24	519	3
Rule	2708	24	341	3
Star Fractal	1490	21	497	4
Tow. of Hanoi	1205	18	1061	5
Search				
Binary Search	2934	22	529	3
Breadth First	2048	23	631	4
Depth First	1946	23	535	3
Sequential			550	3
Tree				
2-3-4	3821	26	927	5
AVL	4025	23	1344	9
B-Tree	4668	21	767	5
M-Way	1542	16	1961	10
Red-Black	2125	5	956	4

Figure 3. Comparison Between Versions

The students learned the importance of maintenance in software engineering. They were given a code base that was very difficult to locate and isolate bugs and quite difficult to add new functionality. In the first week, all students were assigned to bug patrol. There was a lot of frustration in trying to understand the code. During the course, the students learned that software maintenance is a large part of the expense of a software product. This point was driven home by working with the AlgorithmA code base, which has over 161,000 lines of code.

Peer pressure is often times a bad influence, but this could be one of the reasons why many of the students did over and beyond their assigned tasks in the project. The majority of students spent more hours in the software engineering class

than in other classes they were currently taking that quarter term.

The complexity of the task and the short time constraint helped the students learn how to work under pressure. One team leader would later say that this was by far the most difficult and pressure filled quarter in his academic career. He said it felt like a real job except there was no paycheck. He stated that he learned to work under pressure, learned to delegate responsibilities by evaluating skills in others and assigning work accordingly. "When my team was feeling overwhelmed I'd try and lift their spirits, assuring them that we can do this. I learned that sometimes you do more than is expected, you do what needs to be done even though it was assigned to someone else, because in the end you're responsible for it being done", he admitted.

The management team would learn lessons about leadership along with software engineering. They had been hand picked by the professor based upon their submitted resumes and personal interviews. This meant that they would have to lead other students that had tried to obtain these positions but were turned down. There was some resentment. Some felt that they should have been placed in management positions How the managers would garner the support of these students was a question each team leader had to ask as well as answer on their own. They pulled it off. There were no mutinies and very few complaints were launched about their leadership style. In the end each student engineer provided to the professor a private evaluation of the management team members. Their leadership was rated outstanding.

The students working on AlgorithmA 2007 understood the importance of design and architecture. Without this step in software engineering, large software projects cannot be carried out in terms of implementation and maintenance. The MVC design pattern was a good architecture to use. After arriving at the new architecture, there was little time left except to produce some prototypes for the sorting algorithms. The following year, 2008, the students followed this design in re-implementing about half of the original functions of AlgorithmA. In other words, the students experienced forward engineering. And now it is expected that maintenance will be easier for the students who will work on AlgorithmA 2009.

For instructors wishing to duplicate these experiences in their own classrooms, we suggest the following: First, make sure that the project is large enough and complex enough that it cannot be done by a small team of students -- it needs the whole class to undertake the project. Second, since the project will take a lot of time from the students, the instructor must find ways to motivate the students to work beyond the hours required by the class. Third, emphasize the design and architecture step of the software development process. This will ensure that the project can be maintained for a long period of time.

Peer pressure, mentioned earlier as an explanation as to why students did above and beyond their required hours for this class, was brought about by the competition among teams – nobody wanted to be the only one who did not finish their assigned module or task. In XP programming, the students did not want to let their partner down or their team down. But the most compelling motivation for making the students work more is the ownership of the project. Each software engineering class creates their own unique interface, improves or adds more functions, or fixes major problems or bugs. For example, it was the 2007 class that re-engineered the project and the 2008 class that performed the forward engineering. See the Appendix for the major achievement of each class in software engineering.

Regarding project team organization, the basic organization is one management team, several programming teams, and support teams (server, Web, architectural design, documentation, quality assurance. etc.). Care must be taken in selecting the members of the management team. This consists of the project manager and two to three management staff (depending on the size of the class). The management team must be good in both management and technologies, and most important is that the members must be able to work closely together and spend longer hours working on the project than any other students. This is where ownership comes into play – responsibility for the success or failure of the project lies with the management team and all the rest of the teams, most especially the project manager. The students learn from their mistakes of what works and what does not work. The project is implemented in two iterations and so most mistakes are done in iteration one and corrections and adjustments done in iteration two.

As the CEO of the mock software company, the instructor meets the management team and some selected team leaders after every laboratory period. This simulates the executive meetings done to report the progress of the project to the CEO. This meeting also discusses problems and issues generated from the development of the project, such as personnel, architectural design, and implementation. The CEO holds the management team, in particular the project manager, responsible for fulfilling the milestones specified in the Software Project Management Plan (SPMP) document and adhering to the software quality processes specified in the Software Quality Assurance Plan (SQAP) document.

At the end of the class, there is a formal presentation by all the students. This simulates the activity where they become salespeople and try to sell the project. Each team and each team member are given the opportunity to present the component they did and why the project is better. To add realism to the formal presentation, representatives from two local software companies (ESRI and Optivus) are invited to the occasion.

The class size varies from year to year, from 30 – 50 students. The minimum prerequisite for the software

engineering class is data structures, which means the students have completed their first two courses in C++ programming. Other students may also have completed upper-division courses such as, Web programming, server programming, compilers, database, and programming languages. The AlgorithmA project is written mostly in Java and so a majority of the students coming to this class may not know Java beforehand. Therefore a real life situation is simulated here, learning new programming languages and technologies when one joins a software company.

A survey was given to the class to determine their level of skills at the beginning and end of the quarter. The survey is a list of different programming languages, technologies, and tools that may be used in the project. Among the important skills are: Java proficiency, UML Rational Rose or DIA, CVS/SVN, and Bugzilla. The survey is filled out by the student with a 0 (if the student does not have any skill in this item) up to 5 (if the student is at the expert level for this item). At the beginning of the 2008 class, the survey shows a majority of 0s and 1s on the above skills. Figure 4 shows the average proficiencies at the end of the class.

Java Proficiency	2.85
UML/ Rational Rose or DIA	2.47
CVS/SVN	2.73
Bugzilla	2.04

Figure 4. End Proficiencies

For software engineering skills, the class was surveyed to determine if they have learned software engineering concepts and principles. The survey consists of the key process areas (KPA) of the Capability Maturity Model (CMM). The students were asked if they have used the particular KPA or they have observed it being used by a team member or another team. The following KPAs were selected by the 2008 class as either used or observed by 60% or more of the class:

- Peer reviews
- Intergroup Coordination
- Software Product Engineering
- Software Process Definition
- Software project Tracking and Oversight
- Software Project Planning
- Requirements Management

6. Conclusion and Future Directions

AlgorithmA 2007 was re-engineered using the MVC model, the observer and factory patterns. Using these design patterns most of the sorting algorithms feature was re-implemented. The result was amazing. The resulting source code is very clean and completely separated the three major components of a Java applet: the model, the view, and the controller. Now the animation of a sorting algorithm can have any implementation of animation, such as scatter graph or bar graph without affecting the model or the controller components. We can add any other sorting algorithm without affecting the view or the controller components. And we can change the events that trigger animation without affecting the model or the view components.

The experiences and decision-making process that the software engineering class underwent are important experiences that can only be gained by working on a real live project, such as AlgorithmA. The students realized that more work will be required in maintaining the project than re-engineering the entire project.

In AlgorithmA 2008, we re-implemented several of the other algorithms, such as several Data Structure, Recursion, Search and Tree algorithms using the new architecture. It was shown that maintenance was easier because of the adherence to the MVC design and keeping all components in this design pattern. We plan on researching the feasibility of using Ruby or Python to implement the authoring feature of AlgorithmA. It is expected that the current implementation will be more maintainable and extensible for many more years to come. The URL for the most current version of AlgortithmA is https://algo.ias.csusb.edu.

Appendix

Year	Features and Environments
1991	Software design was completed but no implementation.
1992	The first version was completed on two environments: DOS and Unix on 386 machines. It was written in C and using cursor graphics.
1993	The second version was completed on IBM RS/6000, AIX, and using C and XGKS graphics.
1994	The third version was completed on SGI Indigo workstations, Irix, and using SPHIGS graphics. First implementation of the authoring system, which includes a C source code generator.
1995	The fourth version still runs in the same environment as previous with the added features of hypertext interface and multimedia. An R&D team was formed to explore OO paradigm and C++.
1996	The fifth version still runs in the same environment as previous with more sorting algorithms and doubly linked lists and more multimedia added. The R&D team was successful in implementing a prototype entirely written in C++ and using OO approach.
1997	The first version in Java and OO approach was

	implemented using Java 1.0 and awt. For the first time the project was accessible on the Web. Only the BubbleSort walkthrough and animation were completed.
1998	AlgorithmA 98 was implemented using Java 1.0 and awt to be browsed on Netscape 4.0 or newer.
1999	AlgorithmA 99 was implemented using Java 1.2 and plug-ins for Windows 98/NT running on Netscape 4.0 and Internet Explorer 4.01 and Solaris appletviewer.
2000	AlgorithmA 2000 was implemented using Java 1.2, Swing library, and plug-ins for Linux.
2001	AlgorithmA 2001 still runs in the previous environment. Additional features include: pattern matching algorithms, network flow algorithm, AVL tree, B-tree, and authoring system now includes pointer variables.
2002	AlgorithmA 2002 was implemented to run in both Windows and Linux environments. Re-engineered the authoring system for extensibility. It can walkthrough an algorithm and show it's animation.
2003	The authoring system was re-engineered again to allow for extensibility and better animation of the user-defined pseudo-code algorithms
2004	The feature on design patterns was initiated.
2005	Mathematical algorithms were included for the first time and included more design patterns.
2006	Introduced Alma as the cartoon character for exploring the data structures and algorithms.
2007	Re-engineered the entire project in MVC design pattern and started a new version of AlgorithmA.
2008	Forward engineering of project re-implementing about half of original functions.

References

[1] Steve Burbeck, “Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC)” http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html, 1992.

[2] A.I. Concepcion, “Using an Object-Oriented Software Life-Cycle Model in the Software Engineering Course,” In Proceedings of the 29th ACM SIGCSE Symposium, Atlanta, GA, Feb 1998.

[3] A.I. Concepcion, L. Cummins, E. Moran, and M. Do, “AlgorithmA 98: An Algorithm Animation Project,” In Proceedings of the 30th ACM SIGCSE Symposium, New Orleans, Louisiana, Mar 1999.

[4] A.I. Concepcion, N. Leach, and A. Knight, “AlgorithmA 99: An Experiment in Reusability and Component-Based Software Engineering,” In Proceedings of the 31st ACM SIGCSE Symposium, Austin, TX, Mar 2000.

[5] A.I. Concepcion, M. Bernstein, K. Fitzgerald, and J. Macdonell, “ AlgorihmA Project: A Ten-Week Mock Software Company,” In Proceedings of the 36th ACM SIGCSE Symposium, St. Louis, MO, Mar 2005.

[6] Brina Ellis, Brad Myers, and Jeffrey Stylos, “The Factory Pattern in API Design: A Usability Evaluation”, Carnegie Mellon University, http://www.cs.cmu.edu/~NatProg/papers/Ellis2007FactoryUsability.pdf2007

[7] David L. Parnas, “Software Aging,” International Conference on Software Engineering, Sorrento, Italy, 1994.

[8] Linda Rosenberg, “Software Re-engineering”, http://satc.gsfc.nasa.gov/support/reengrpt.PDF

[9] Tom Mens and Tom Tourwe, “A Survey of SoftwareLRefactoring”, IEEE Transactions on Software Engineering, Vol 30 No 2, Feb 2004.