Applying Software Engineering Principles in Developing Safety-Critical Software Systems: A Class Project

Tyler Kaye, Sudhanwa Kholgade, Jeffrey Knutz, David Lannoye,
and John Sartori
Department of Computer Science
University of North Dakota
Grand Forks, North Dakota 58202-9015
grante@cs.und.edu

Abstract

The application of software engineering principles in software development is growing in occurrences and importance. This growth is driven by the increasing number of safety-critical software systems being developed. Development of safety-critical software systems require the use of dependable, reliable, and standardized methodologies, tool support, and modeling and programming languages.

One such set of methodology, tool, and language is an object-oriented model-driven software development methodology, Rational Rose CASE tool, the Unified Modeling Language (UML), and Java programming language, respectively. The growth and dominance of object-oriented development is based on the view that this approach to software development offers a better representation of real world problems and solutions than other methodologies.

In this report we document the work done, and experience gained by teams of upper-class undergraduate students on a one-semester long software engineering course project. The project involves the development of a safety-critical software system using the aforementioned development methodology, tool and languages.

1. INTRODUCTION

The problem of diabetes is a growing concern in the world, especially among Americans. According to the American Diabetes Association (ADA) diabetes is a "disease in which the body does not produce or properly use insulin."[1]. Currently, this disease is believed to affect 20.8 million Americans – almost 7 % of the total population. Within these numbers, the ADA also includes the estimated 6.2 million people that have yet to be officially diagnosed. The diagnosis of diabetes falls into two categories – Type 1 or Type 2. Type 1 diabetes, which accounts for about 10 % of the total patients, occurs when the body stops producing insulin or produces too little insulin to regulate the glucose. Type 2 diabetes occurs when the body is partially or completely unable to use insulin. With this disease, there are several medical complications that can arise. Some of the most common health problems that affect both types of diabetics are heart disease, strokes, high blood pressure, blindness, kidney disease, and damage to the nervous system. Currently, diabetics practice several methods to help prevent or lessen the effects of these complications.

The easiest way for Type 2 patients to manage their health is through a healthy diet and exercise plan. For Type 1 patients, treatment almost always involves the daily injection of insulin, which is the focus of the Automated Insulin Pump System (AIPS). Currently, there are two ways in which a patient can administer insulin. In the first method, the user must check his or her blood sugar with a glucose sensor, calculate the appropriate amount of insulin to administer, and personally inject the insulin. In the second method, the patient uses a AIPS. The AIPS detects the level of glucose in the user's blood, calculates the amount of insulin need, then triggers a pump to administer the correct dosage to the user via a needle that inserted into the user. Both of these methods require the user to play a critical role in his or her treatment. Using the AIPS minimizes the possibility of errors occurring.

To accomplish this, the AIPS integrates the blood glucose sensor and the insulin pump into one system. Integrating these two processes allows the autonomous delivery of insulin to the user. This ability of the system to remove the user from the glucose self monitoring and injection process allows diabetics to live a healthier and more enjoyable lifestyle.

1.1 Problem Description

The spring 2005 Software Engineering undergraduate class at the University of North Dakota, Department of Computer Science was assigned the goal of developing a simulation of a Automatic Insulin Pump System that would demonstrate the usefulness of such a device to diabetic patients.

The class of twenty two students was invited to form teams of two (2) to four (4) students that worked independently as a software development project group. Each team elected a team leader, whose responsibilities including, consulting with the customer (instructor)

outside of class time, ensuring that all deliverables were handed in on time, and schedule meeting times and assignment of work for the team. The course instructor acted as the customer of the product, and the work had to be completed over the course of the semester. In the last week of the semester each team made a presentation of their work to the class and answered questions field from both the instructor and other students in the class. The teams had to submit a series of progress reports, during the semester, which outlined (1) work accomplished to date; (2) problems encountered and their solutions; and (3) tasks to be completed by the next reporting period. Along with a working application each team had to submit write-up of the work done, object-oriented models of the system at various stages of development, and instruction on using the application.

The theoretical course material was presented in sequence with the project such that the students learnt the fundamental software engineering principles and concepts before needing to apply them on the project. The material used in this course was mainly from the required course text [2], and the suggested reference text [3]. Teams were expected to source additional information to supplement not only the project requirements, but also the theoretical foundation of the course.

1.2 Significance of the Work

As software development moved from the realm of a science to an engineering process, the importance of systematic software development became more and more apparent [2]. Today's software development takes place in a distributed, multinational environment where hundreds of people may be simultaneously working on the same project. A methodical approach to requirement capture, system specification, design and implementation is an approach that supports development on such a scale, interaction and co-operation. History has taught us how crucial systematic software development can be [4].

The Denver International Airport's failed baggage delivery system is a prime example of how expensive an improperly undertaken software development process can be [5]. After spending $230 million and a decade to fix the system which was not built properly in the first place, the project was given up. Another example is the recall of Abbott Labs Blood Glucose Meters [6], which could be inadvertently commanded to display glucose measurements from US standards to European standards. As a result, the user could take possible harmful actions to 'control' the incorrect sugar level in his or her body. These examples underline the importance of building a system correctly, and in this report we outline our attempt to do by following established software engineering principles.

1.3 Scope of the Work

The development of the AIPS was constricted by the time available to complete the project. As the culmination of a semester long introductory course in software engineering principles, work on the project was accomplished alongside concepts as they

were taught as part of the coursework. Consequently, some elements of a real life software development cycle were not performed. Particularly, there was no feasibility study performed to ascertain the merits/demerits of this software engineering effort. Although requirements for the project were provided by the contractor and not captured by the project team, requirement analysis was performed to better understand the project. An object-oriented iterative waterfall software development process was then employed to develop the AIPS. After implementation and testing, the project was deemed to be complete. System maintenance was not performed to enhance the system after testing.

The remainder of this report structure is as follows: Section 2 presents the fundamental theory and concepts used in the project; Section 3 documents the actual work done for this project; and Section 4 presents our conclusions.

2. BACKGROUND

Students in this course were introduced to a number of theories and concepts that were not encountered in previous computer science courses. The course structure exposed the students to an equal quantity of theory and practical applications of the theory, by way of the team project. In this section we outline the key topics covered and used in the team project.

2.1 Software Engineering Principles

The first weeks of the course covered the set of software engineering principles [7] from which were a focus in carrying out the team project. The software engineering principles theory covered in the course were:

- Rigor and Formality – focuses on the accuracy and precision associated with the system, and the use of mathematical theory and concepts in measuring and ensuring this accuracy and precision.
- Separation of Concern – has to do with the developers' ability to focus on a particular view of subset of issues of the overall system.
- Modularity – is related to the separation of concerns, as it seeks to compartmentalize related concerns into modules.
- Abstraction – is a technique that is also related to separation of concern as it allows developers to concentrate levels (layers) of understanding of the problem and its solution.
- Anticipation of Change – is the ability to factor into the system its evolution, from the initial system to one that will incorporate additional functionality and features, some of which are unknown at the initial stage.
- Generality – involves the ability of being able to identify this specific problem as an instance of a more general class of problems – This is related to the principle of abstraction.
- Incrementality – is the act of building the system in deliverable modules and grow the complete system in increments.

During the development cycle in this project these principles acted as guides, with the exceptions of formality, generality and incrementality. The nature of this project being completed in one semester did not lend itself to apply these principles.

2.2 Development Process

The next few classes covered the topic of process development. The class was introduced to a number of development processes, namely: the classical Waterfall and Spiral approaches. We learnt about component-based software development, process iteration, and the object-oriented approach to software development. Fundamental to all these approaches is the stages (phases) of the respective cycles.

In this course we adapted an iterative waterfall approach that was object-oriented based. This development process is graphically depicted in Figure 1.

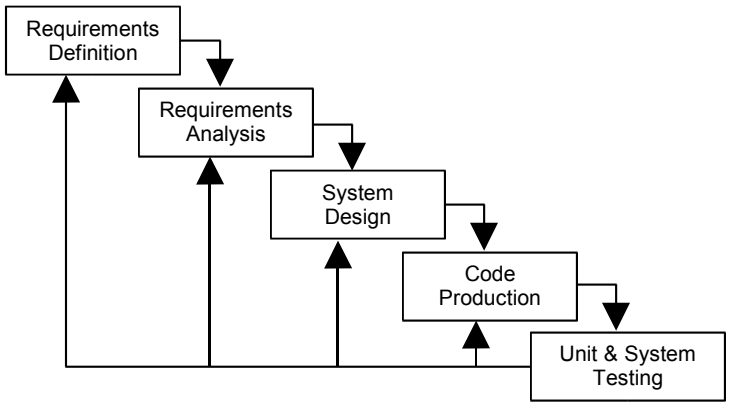


Figure 1 – Project Development Process

The process of Figure 1 allowed us to return to a previous development phase to correct, modify or add information based on our discovery at a later stage.

2.3 Modeling Notation

In object-oriented development, the Unified Modeling Language (UML) [8] is the *de facto* modeling language and is supported by major corporations. The UML is a set of graphical and textual notations for modeling various views of software systems, using object-oriented (00) concepts. The UML specification offers a set of syntactically well-defined modeling notations, and is a general purpose modeling language. The UML's EMs are used to tailor the standard UML concepts for specific enterprise, domain or application requirements. The UML EMs were specifically defined to meet the requirements of some projects which require features beyond those defined in the UML specification. The UML defines a set of models of which the following ones are used in this project:

- Class Diagram – is used at all stages of the development cycle and captures the relationships between the static entities of the system.

- Use Case Diagram – is used at the requirement stage of the development cycle and captures the functional (dynamic) properties of the system, and the relationships between the users and other systems.
- Activity Diagram – are used at the early design stage to define the task involved in carrying out a system activity. An activity diagram implements a use case of the use case diagram.
- Collaboration Diagram – details the communication sequences between objects in carrying out a task.

We also include a static and a dynamic dictionary as a deliverable item. These dictionaries are textual representations of the static and dynamic aspects of the system.

2.4 Development Tool

The project was developed using the computer aided software engineering too Rational Rose® [9]. Rational Rose is an object-oriented UML software design tool intended for visual modeling and of enterprise-level software applications. Rational Rose provide iterative development and round-trip engineering. It allows designers to take advantage of iterative development because the application can be created in stages with the output of one iteration becomes the input of the next. Rational Rose can also perform "round-trip engineering" by allowing the developer to trace changes from the code back to the models from which the code was derived.

3. PROJECT DEVELOPMENT

In this section we present some of the models developed in the project. As was mentioned earlier – different models were developed at various stages of the development life cycle. We built system models at the requirement analysis stage, high level design and detail level design stages of the development life cycle. These models were UML objected-oriented models.

3.1 Requirement Models

At the requirement stage of development two models were developed. The first was a Use Case Diagram, that captured the services (functionalities) offered by the system to the user. Along with the use case diagram a requirements level class diagram was developed. This class diagram eventually evolves into the detailed design level class diagram. The requirements level class diagram contains the classes that are known to the user of the system and the relationships between them. In the final detail design class diagram all known attributes and methods of the system are included. The use case diagram is presented in Figure 2 and the requirements level class diagram is presented in Figure 3.

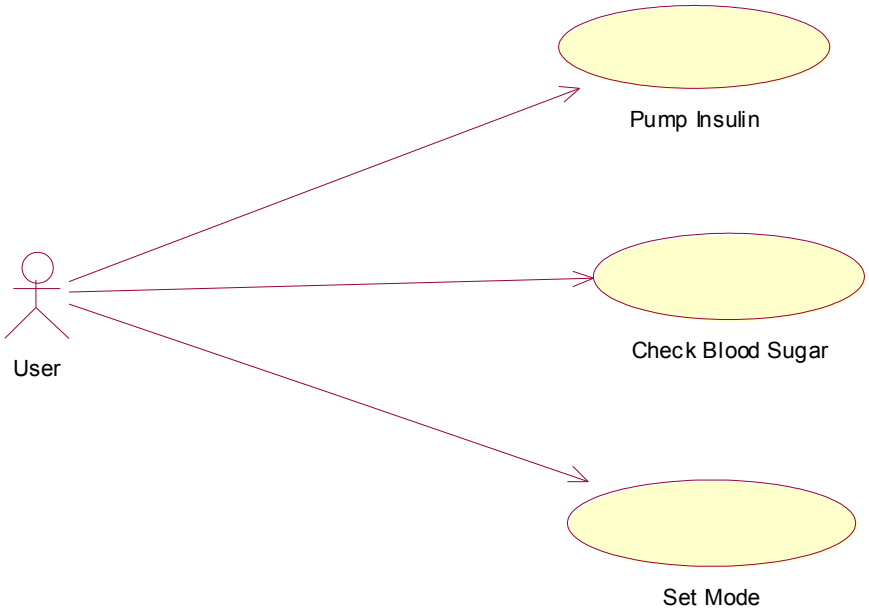
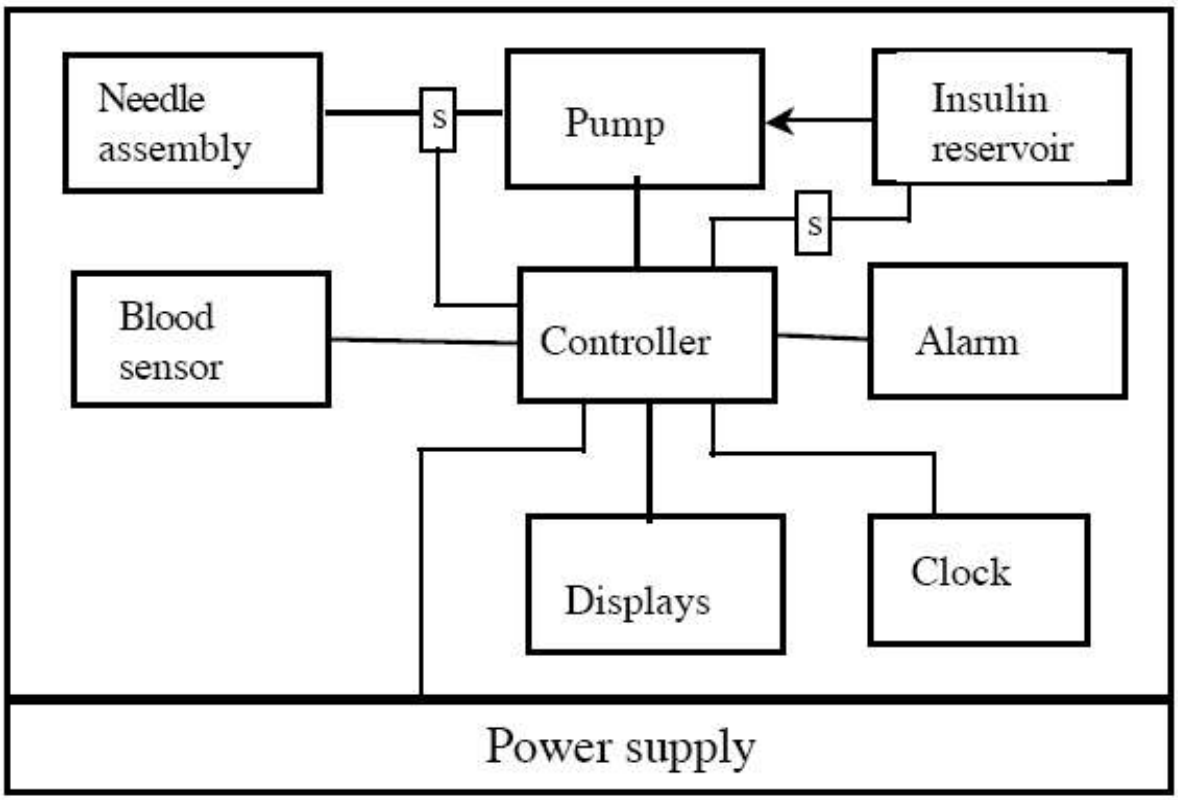


Figure 2 – Requirement Use Case Diagram

Figure 3 – Requirement Class Diagram

3.2 Desig

At the desi
developed.



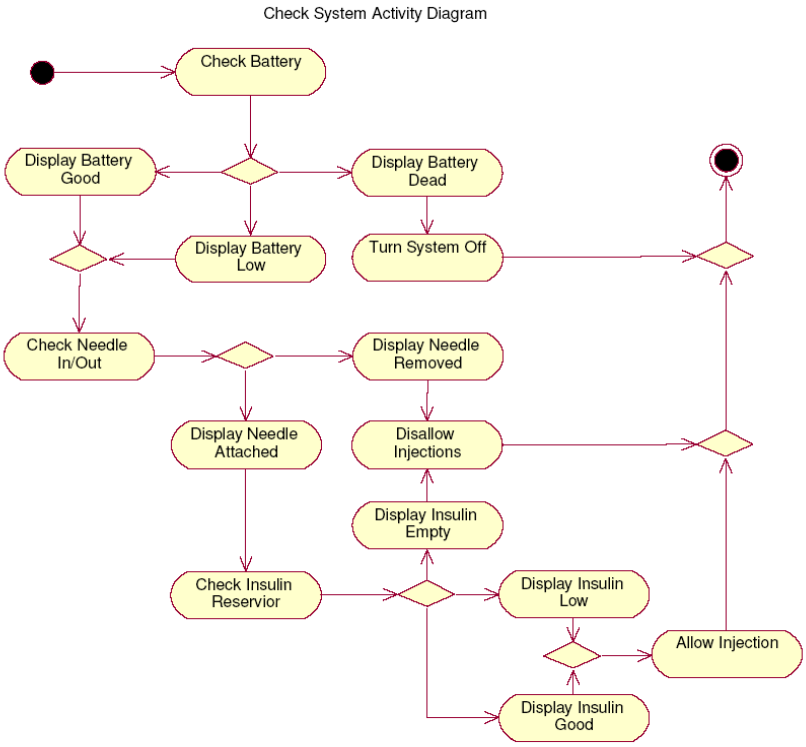


Figure 4 – Check System Activity Diagram

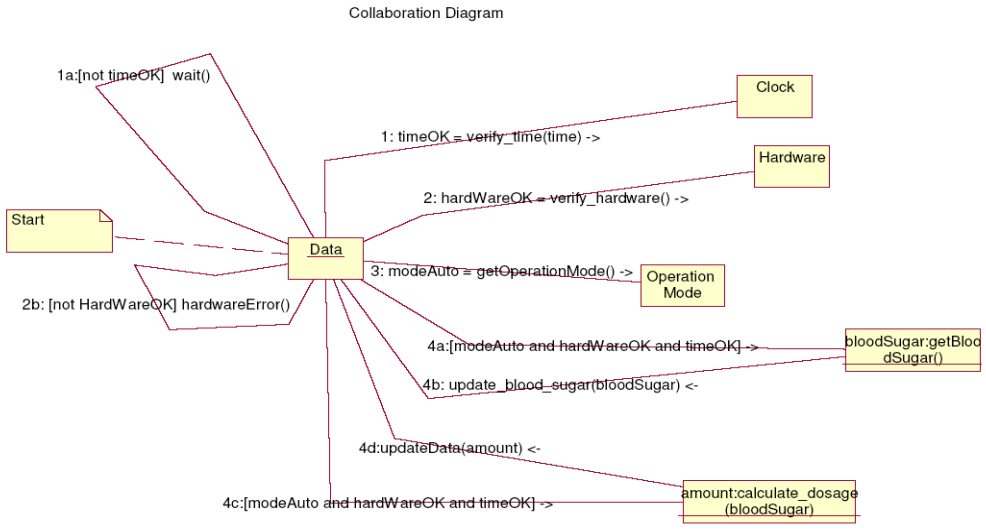
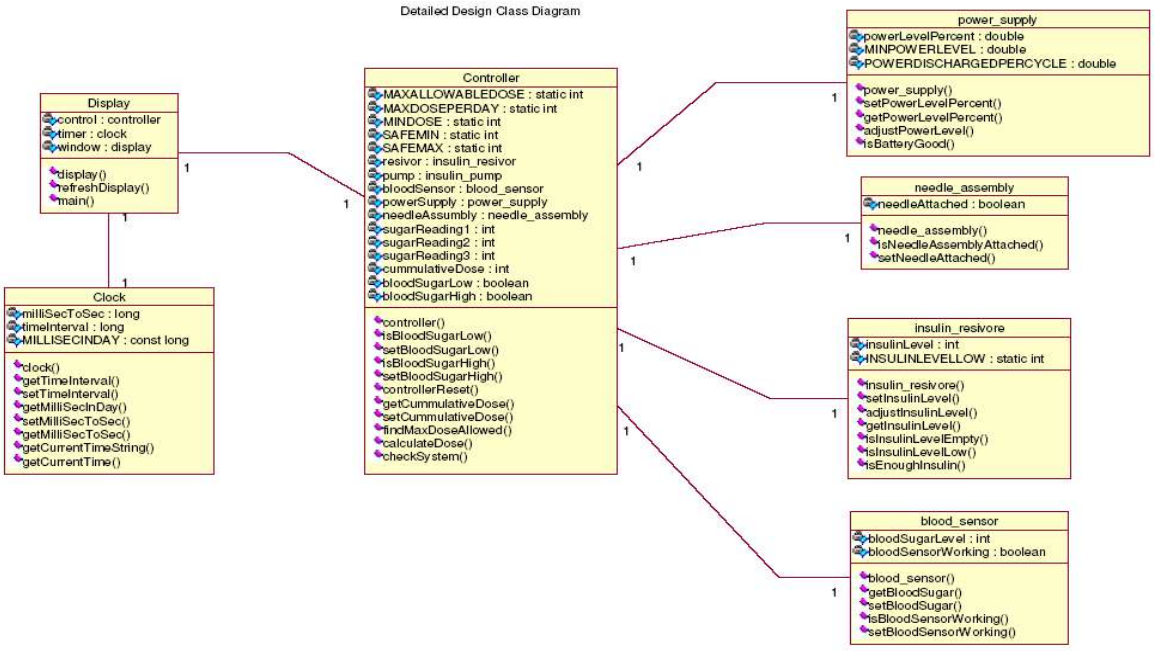


Figure 5 – Collaboration Diagram

Figure 5 above illustrates the collaboration for checking the system. The detailed design class diagram is illustrated in Figure 6 below.

Figure 6 – Detail Design Class Diagram



3.4 Code Production

The software coding for the insulin pump system was written in the Java programming language. Java was used for the ability to create classes which closely mimicked those from the Detail Design Models. Java also allowed us to code more like we would for a true embedded system. We had class of objects that formed two major parts of the system, a backend where all the hardware interaction takes place and a user interface. This user interface, which represents the front panel display that can be accessed by the system user, contains all relevant controls and indicators necessary to monitor and administrate system operation. The graphical user interface (GUI) for the system is presented in Figure 7 and 8.

The above user interface displays all relevant system information to the user, as well as all controls needed to operate the system in "manual" mode. In accordance with the project requirements, the GUI displays the current time, the last time a dose of insulin was administered, and the corresponding amount of that dose. If any hardware component malfunctions while the system is running, a system alarm indicator activates, prompting the user to check the system messages. This alarm is both auditory and visual. By scrolling through the system messages, the user can isolate the source of the error and take appropriate measures. Other indicators on the GUI show the level of charge in the battery and the amount of insulin remaining in the reservoir, there is also a history button which displays a table containing a history of blood sugar values and doses.

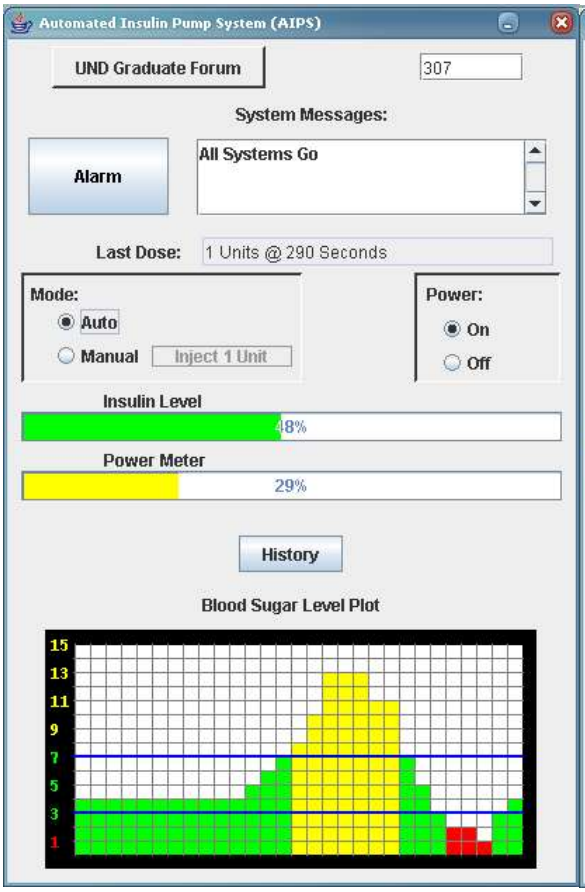


Figure 7: System User Interface.

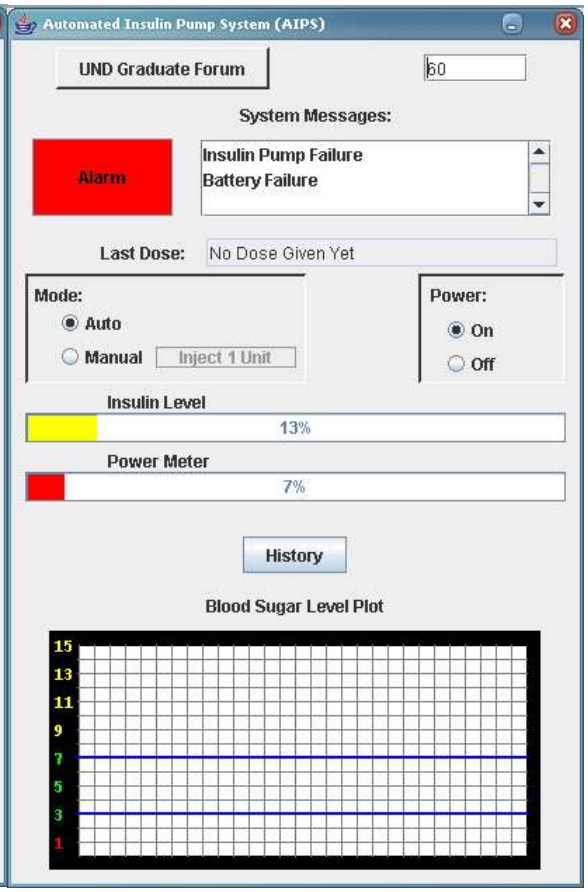


Figure 8: System User Interface During Failure.

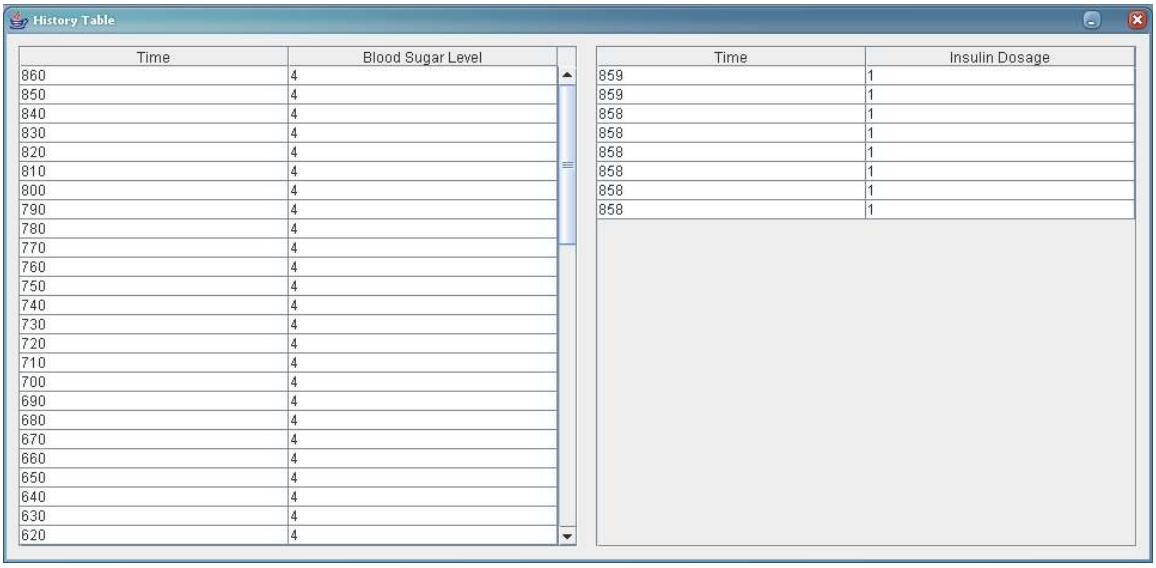


Figure 9: Table of history of the system.

Besides displaying relevant system information, the user interface also functions as a control panel for manual operation of the insulin pump. To deliver a manual dose of insulin, the toggle switch controlling the operation mode must be set to manual. Once the system is running in manual mode, the user may press the "Inject 1 Unit" button to deliver one unit of insulin. Even in manual mode the system users decisions are checked make sure they do not exceed the maximum daily dose. Figure 10 illustrates the error message from attempting to deliver more than the maximum daily dosage.

Figure 10: Visual Notification of Maximum Insulin Dose for the Day.

In addition to the User Interface a hardware simulator was design and coded to run on beneath the insulin pump and provide the backend with different state levels for the various internal variables. Figure 11 displays the portion of the GUI that displays the internal variables.

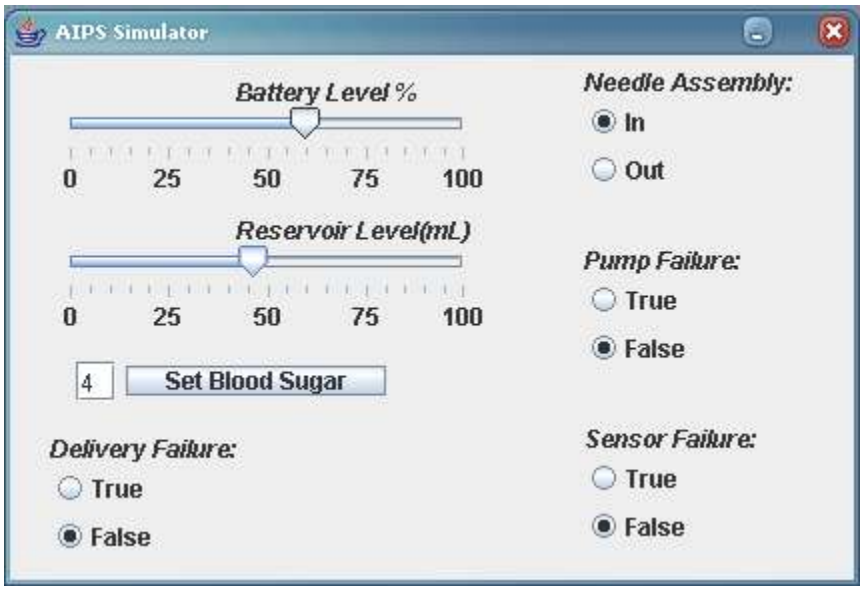


Figure 11: Visual Representation of Internal Variables.

This interface provides greater control over the system during testing. The tester can change the levels for both the battery and insulin reservoir. The interface allows the tester to manually specify blood sugar levels to test if the system accurately responds to changes in sugar levels. Finally the panel allows the tester to simulate a variety of different system failures to test the systems response to these.

3.5 Testing

Upon completion of the coding of each subsystem they were tested for correct operation. Upon passing all specified test cases, the subsystem was integrated into the larger system model. Finally, when all subsystems were completely specified and integrated, the entire AIPS model was tested rigorously. To ensure proper functioning of the system, a test plan was created, describing each case to be tested, the expected result, and the observed result. Verification of the test plan led to changes in the system architecture. Once all test cases passed together, the system was deemed ready for delivery. The test plan matrix used in system verification can be found in **Appendix A**.

4. CONCLUSION

The process of software development has become ubiquitous. Only a few decades ago, most software development was accomplished by small teams and was considered part of computer science. Most of today's software systems are extremely complex, high-speed, distributed systems. As complexity has increased, the art of building software has evolved into the software engineering discipline. Without the techniques, protocols and methods of software engineering, designing and implementing today's software systems would be extremely challenging if at all possible. This software engineering project enabled the team to get a taste of each of the phases of the software development cycle and helped put textbook principles into a real life perspective.

The development of a new software system begins with the capturing of system requirements. In many senses, this can be the making or breaking portion of the project. More often than not, the real life process that is being modeled or automated is not a part of the software engineer's field of knowledge. It thus becomes necessary that the software engineer understands the system and its requirements in the same manner that the customer understands them. Although the contractor provided the system requirements, this team extracted additional requirements from a diabetic. This exercise helped the team understand the project from an end user's perspective and get a feel for what an ideal system should be like. It also forced the team to make a decision between a system that is good enough and will satisfy all requirements and an ideal system. Given time, budgetary and resource constraints, demarcating this boundary can be a critical decision in real life.

The team relied heavily on the Unified Modeling Language for the entire design process. The project was iteratively broken into smaller and smaller, more manageable chunks until all the major components and processes were identified. These components could then be interpreted in terms of software components and implementation could begin. In the implementation phase, the team had to make another important decision, the choice of a language and platform to implement the design. The purpose of a software design via UML is such that it can be implemented in any language that supports the object-oriented paradigm. Choices such as this one will definitely benefit a lot in real life projects, and aid in completing the project in a conservative timeframe.

The testing phase immediately followed the implementation phase. Creating a realistic and at the same time thorough test plan is extremely important – especially while testing a life support system such as the AIPS. After the system was found to behave as required, the system was ready for release. In the real world, beta testing would follow this stage and gather statistics and human sentiments about the product so that it can be further improved. This would also help sort out any bugs that went undiscovered. A final version of the product would then be released. Maintenance and release of improved versions of the system would then continue until a new technology is discovered and the product is retired.

This project has been a valuable experience that closely followed a real life scenario. The core software engineering concepts that were learned in the classroom were applied in this project, and insight about their power or shortcomings was gained. One of the most important lessons learned was that of time management and delegation of tasks. The importance of breaking down tasks into modules that can be assigned to different development teams also became very evident.

REFERENCES

[1] http://www.diabetes.org/home.jsp

[2] Sommerville, Ian *Software Engineering* 7th Edition, Addison Wesley, ISBN 0-321-21026-3, 2004

[3] Larman, Craig *Applying UML and Patterns* 3rd edition, Prentice Hall, ISBN 0-13-148906-2, 2005

[4] Shaw, Mary; Garlan Gavid, *Software Architecture: Perspective on an Emerging Field* Prentice Hall, 1996

[5] Weiss, Todd *United to Scrap Baggage System at Denver Airport* Computerworld, June 13, 2005

[6] Recalls & Safety Alerts *Blood-Glucose Meter could give Incorrect Readings* Consumer Report, December 2005

[7] Ghezzi, Carlo; Jazayeri, Mehdi and Mandrioli, Dino *Fundamentals of Software Engineering* 2nd edition, Prentice Hall, 2003

[8] http://www.uml.org

[9] http://www-306.ibm.com/software/rational/

APPENDIX A

Test description	Component being tested	Test Input	Expected Response	Observed Response
The battery alarm shall be set if battery level falls below battery alarm level.	Power Supply	BatteryAlarmLevel = 10 BatteryLevel = 50	No alarm	No alarm
	Power Supply	BatteryAlarmLevel = 10 BatteryLevel = 5	Alarm set	Alarm set
The blood sensor alarm shall be set if the blood sensor malfunctions.	Blood sensor/Controller	SensorStatus = false	Alarm set	Alarm set
The needle assembly alarm shall be set if the needle assembly malfunctions.	Needle assembly/Controller	NeedleStatus = false	Alarm set	Alarm set
The pump alarm shall be set if the pump malfunctions.	Pump/Controller	PumpStatus = false	Alarm set	Alarm set
The insulin alarm shall be set if the insulin level falls below the maximum allowable dose	Insulin reservoir	InsulinAlarmLevel = 10 InsulinLevel = 15	No alarm	No alarm
	Insulin reservoir	InsulinAlarmLevel = 10 InsulinLevel = 8	Alarm set	Alarm set
The system shall administer an insulin dose if the blood sugar, is in the safe zone but increasing at an increasing rate	Controller	CurrentReading = 115 PastReading1 = 100 PastReading2 = 90	Issue dose	Issue dose
The system shall administer an insulin dose if the blood sugar is above 120 mg/dl and steady or increasing, but not is the rate of decrease is increasing	Controller	CurrentReading = 140 PastReading1 = 140	Issue dose	Issue dose
	Controller	CurrentReading = 130 PastReading1 = 150 PastReading2 = 160	Do not issue dose	Do not issue dose
The system shall not administer a dose if the blood sugar is in the safe region and the rate of increase is not increasing.	Controller	CurrentReading = 110 PastReading1 = 100 PastReading2 = 85	Do not issue dose	Do not issue dose
The system shall not administer a dose if the blood sugar is less than 80 mg/dl	Controller	Current Reading = 60 TriggerDose = true	Do not issue dose	Do not issue dose
There will be no dosage of insulin after the daily maximum allowable limit has been reached.	Controller	MaxDailyDose = 25 DailyTotal = 25 Dose = 5	Do not issue dose	Do not issue dose