
Experience Paper

For

InSpa Project

Version 1.0 approved

Prepared by:

Isaiah Paradise, Jeffrey Toce, Mehdi Slaoui Andaloussi

Seattle University Software Engineering Program

June 9, 2008

1. Introduction

1.1. Team Members

Mehdi Slaoui Andaloussi: A native of Morocco, Mehdi earned his Bachelor's degree in Electrical and Computer Engineering from McGill University in Montreal, Quebec. After graduation, Mehdi moved to Seattle for a job at Microsoft Software Engineer in Test for Microsoft Game Studios. After 3 years, he moved on to the MSN group as Lead Software Engineer working on Branded Entertainment web site. Mehdi is currently a Program Manager working on the next generation of rich internet application framework called Silverlight.

Isaiah Paradise: Isaiah earned his Bachelors of Science in Computer Engineering at Washington State University, with a focus in ASIC/VLSI design and systems programming. Following graduation, Isaiah went to work for The Boeing Company in 2003, where he served as a lead Software Engineer for Airborne ISR software systems and unmanned air vehicle software integration from 2003-2008.

Jeffrey Toce: Jeff earned his Bachelors of Science in Computer Science at Carleton College in Minnesota. Following graduation, Jeff went to work for Microsoft from 2001-2005, where he took a job at Wizards of the Coast. In 2006, Jeff returned to Microsoft to pursue his master's degree. He is now a lead software developer in test for Microsoft Finance IT department.

1.2. Project Goals

The original project goals for InSpa was to develop a data mining tool that will generate demand forecast projections from InSpa's historical point of sale database. However, after meeting with InSpa, we discovered that InSpa already has a way of generating a demand forecast. What they lack is a way to efficiently build employee schedules around this demand forecast data. The new project goals is to now create an auto-scheduling application that will use demand forecast data and employee data input to build an efficient schedules to maximize revenue. The primary use case of this system will allow spa directors to add/remove/modify employee data and generate schedules of variable length for individual stores and employee types.

1.3. Software Development Process

Because all of the team members work full time, and most of the team members were enrolled in secondary classes, our schedules were very difficult to mix. It was determined that we could only meet once a week on the weekends, and all of our work will be done virtually. All communication will primarily be done through e-mail. It was also determined that this project was going to be much larger than originally estimated. The minimum 100 hours per quarter quota for the students wasn't going to be enough to complete a project of this size on complexity. Because of these reasons, the team needed a lightweight software development process that focused on maximizing coding while minimizing metrics, documentation, and administrative overhead.

The team chose to adopt a lightweight SCRUM process for the InSpa project. We broke the tasks down based on functionality and developed a sprint backlog. We used one-month developmental sprints to try and keep our iterations short. Each week, the team lead would write up a progress status for the customer and project sponsor.

Having a good configuration management tool was critical for this project. Not only is the code base large, the team members were all independently working on the code. Having a quality CM tool and code merge tool was very important. Our choice in CM tools came down to Subversion SVN. We choose Subversion because it is a free open source tool that is widely used. It also had an excellent client tool called Tortoise SVN, which integrated directly with Windows Explorer. We hosted our code repository on a secure project server at Seattle University, and used VPN to connect to the repository and upload/download current versions of the software.

1.4. Challenges and Risks

The challenges of this project were clear from the beginning. The new tasks of creating a software system for generating efficient schedules is a much more complex task than data mining. The problem space of solving scheduling problems is extremely large and difficult to formally test. The biggest risk was the amount of effort that was required to complete this project. The team members did not have a lot of spare time to devote to this project outside of the 100 required hours per quarter.

The complexity of the scheduling program grew once we realized the challenge of this problem. Simply generating schedules to maximize on revenue is only one component of the problem. The hard part is generating reliable and predictable schedules for the employees. Employees have their own view of what a “good” schedule is. Every employee has different preferences on hours and days they want to work vs. hours and days they are available to work. Some employees have demands for consecutive days off (i.e. weekends). Most employees do not want to even show up for work unless they were given at least a 4 hour long shift. To further complicate the problem, different employee types have different scheduling restrictions. For example, massage therapists cannot work more than a certain number of hours per day because of the physical demand of their skill. Assigning “bad” schedules to the employees will just disrupt the employees lives and will likely result in employee turnover. All of this adds up to be a very complicated problem to solve on paper, let alone solve in software. Therefore, the complexity of this problem was a risk to the project schedule. To mitigate this risk, we relied on the expertise of Dr. Joslin to help solve the scheduling problem.

Because InSpa using Windows based technologies, we saw it fitting to using Microsoft .NET technologies in C# to design and implement the InSpa scheduling application. This appeared to be the lowest risk for moving forward with the project.

2. First Demo Prototype

In the spirit of agile development, the first deliverable was to develop a prototype demo for the customer. The first demo prototype was just a dummy shell that was meant to show the customer what the interaction would look like and what the output format looked like. This section talks about the various artifacts which were developed for the prototype.

2.1. Database Design

The first artifact we needed to produce was a database design that best represents the relationships between the data entities. We also needed a database design so we could start working with real data from InSpa. Our original database schema was designed in Visio and then exported over to Excel DB tables to make it easy for the rest of the team to integrate with. This database would later be ported over to SQL Server.

2.2. Software Object Model

We needed an object model that made it easy to represent and query work schedules. Our approach was to create classes to represent the entities that make up a schedule. For example, an arbitrary schedule is made up of scheduled weeks. Each day in the week has a list of shifts and a demand forecast. Each shift is assigned to an employee. We ended up using a lot of C# Lists and Dictionary collections to organize the schedules internally. In order to analyze the efficiency of a schedule, one would just need to iterate through each day of the week and compare the realized revenue with the expected revenue from the demand forecast.

2.3. User Interface

The initial User Interface was a hot topic for discussion. We needed a quick and simple UI that allowed us to easily render a visual representation of a schedule. What we wanted was a grid schedule format with painted bars to represent shifts. However, developing this kind of display presentation from scratch would be a large effort, which we didn't have time to do. To mitigate this risk, we leveraged the Microsoft Excel Object Model library to generate schedules to Excel. Excel also allows us to write embedded GUI controls in the presentation layer using Visual Basic. It was decided that we would use Excel as the user interface and database all in one file. The actual scheduling algorithm would be invoked as a separate process in C#, where it would interop with the Excel sheets to draw the schedules.

The essence of the User Interface was to have 4 views (i.e. Excel Sheets). The main view would be the program start page, where the user will select the store, employee type, scheduling start date, and the number of weeks to generate. The scheduling output will be written to a Report view on a separate sheet. We also designed two other sheets for viewing Employee and Store data. The "Employee View" allowed the user to select and modify the scheduling preferences for an employee. The "Store View" provided a view of the employee preferred and available schedules for a store. The basic functionality of the UI will be to configure employee scheduling parameters, invoke the scheduling algorithm, and view the result.

3. Core Software Development

Following the initial prototype demonstration with our sponsors, our next step was to actually design a full release application. The customer liked the idea of using Excel as the user interface because it was simple and is an application they understand very well. However, the hard part of solving the actual scheduling problem was just beginning. This section describes the core software development for the InSpa project.

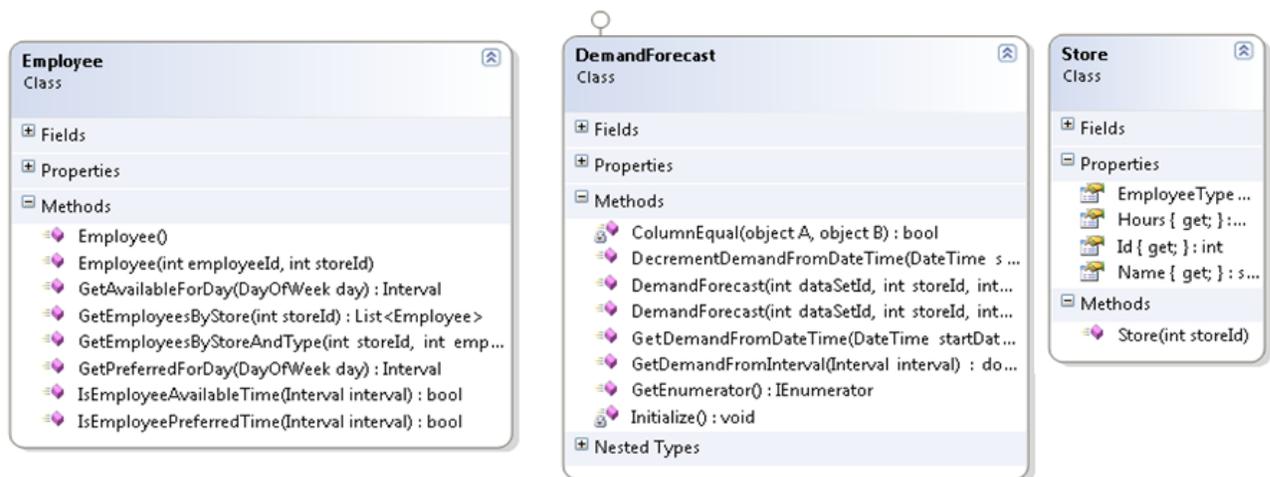
3.1. Scheduling API and Algorithm

Our goal from the beginning was to create a set of very generic APIs that will be used by different algorithms. These APIs are helper functions that allow us to retrieve different information about the following entities:

- Employee
- Store

- Demand Forecast
- Schedule

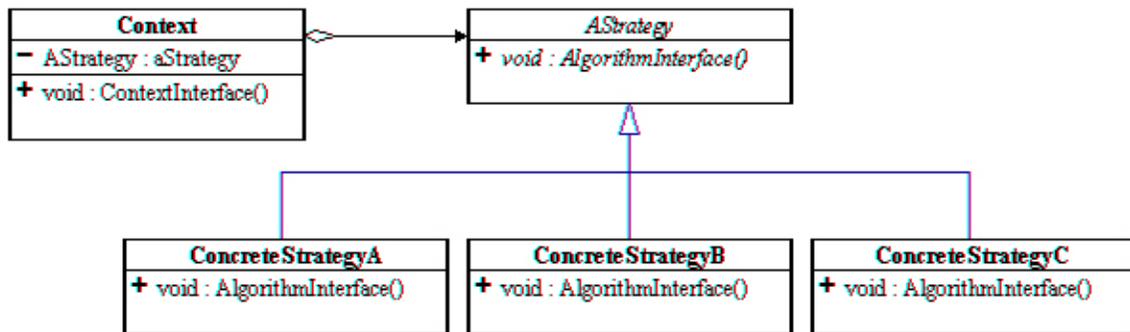
The following diagram shows a high level view of the object model:



We used a number of data structures in our project. During the development we were concerned about the efficiency of our algorithm and we knew from the get go that many retrieval function needed to be as fast as possible. Here is a summary:

- A **Schedule** is basically a **List** containing objects of type **Week**
- A **ScheduledWeek** is an **array** containing objects of type **ScheduleDay**
- A **ScheduleDay** is a **List** containing objects of type **Shifts**
- A **DemandForecast** is a **dictionary** using dates as key and demand as the corresponding value

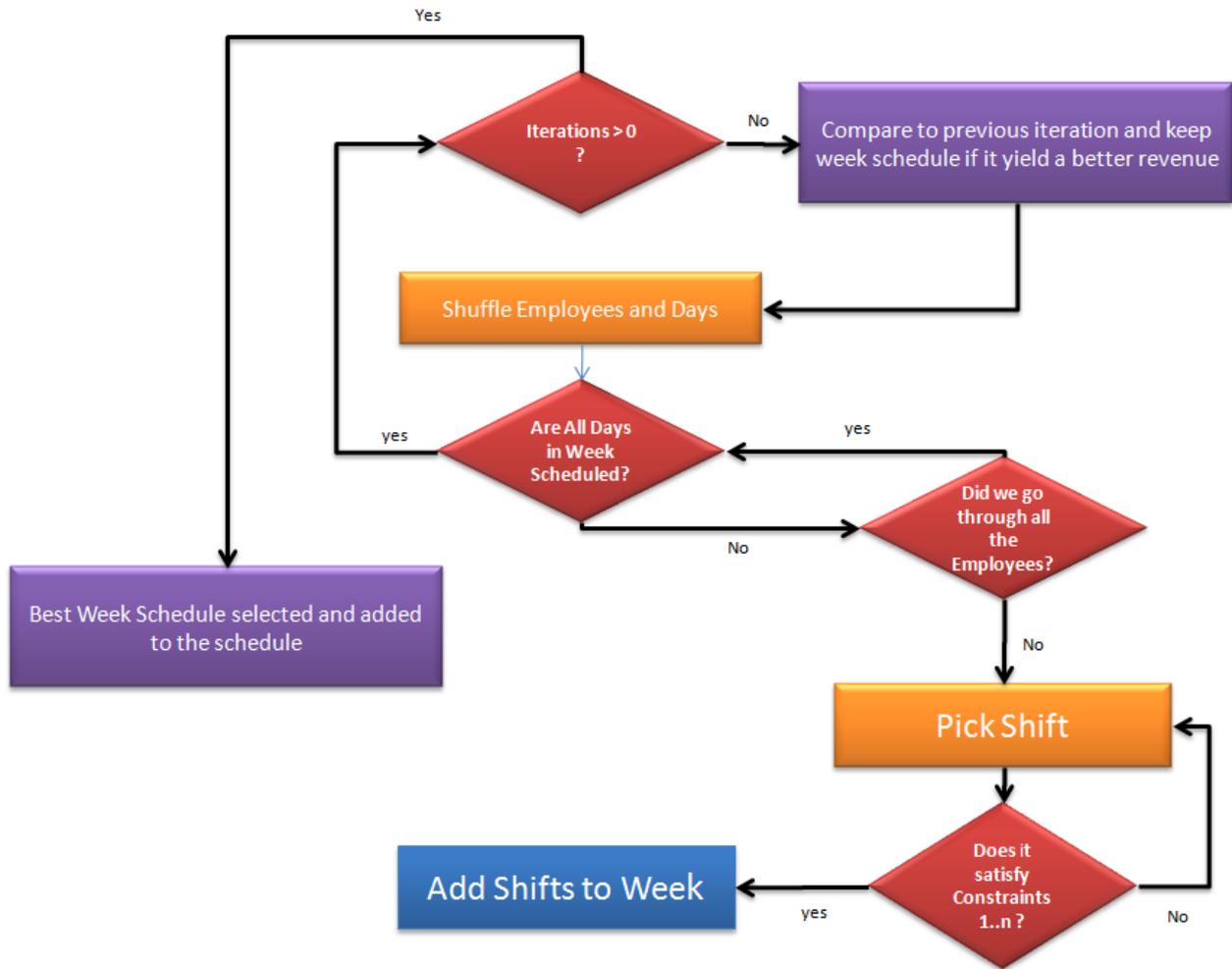
Another important design goal we had during this project is to allow for future developers to create new algorithms and easily plug them without any change to the architecture. For that purpose, the strategy design pattern came in handy:



The following diagram shows the class diagram for the scheduling algorithm:



Our team focused in the implementation of the GreedyAlgorithm which is basically a very brute force way of generating a large number of schedules by introducing each time some randomness and picks the best schedule. Here is a diagram showing the flow of the GreedyAlgorithm:



3.2. Data Access Layer

Any properly designed application requires a componentized data access layer to bridge the gap and translate data between the database layer and object model types with the rest of the system. Our design goal was to implement an abstract data storage layer that is separate from provider implementation (Excel, SQL Server, etc...). We wanted to provide a local in-memory copy of data that can be modified without affecting the master source. A typical use case for the scheduling application would be the spa director changing around employee information and checking the scheduling efficiency to see if they need to hire or lay off an employee. During this activity, we do not want to commit any changes back to the database because the user will be generating “what-if” scenarios for scheduling. Ultimately, we would want to track changes to the data and write them back to the master source on command. The first Data Access Layer mainly served as a simple data importer that translated data from generic DataSet types to our object model types. A true generic database factory pattern was needed to fully achieve the desired behavior we wanted. This transition didn’t take place until we were faced with some large refactoring decisions later in the project.

3.3. User Interface

The user interface didn't change much from the original prototype demo. We still had the program launcher view, employee view, store view, and scheduling report. The format was modified to display the employee and store data differently per customer request. Besides these changes, there were some additional "extra credit" features we were going to add. Hosting the UI and database within Excel gave us a significant advantage of utilizing the charting and graphing capabilities of Excel. We were proposing to add a chart wizard GUI form that will allow the user to select data categories and plot the data in a floating graph as well as other schedule reporting presentations. The advantage with this is the user could plot financial data within the UI using the generated schedule data vs. the original demand. There were numerous options to this feature. Although this fell outside the scope of the original goals and objectives, it would server a nice "free" feature for using Excel as the presentation layer.

However, like most original plans, requirements creep and feature creep tend to lead to refactoring work down stream. Often features which are not required end up getting cut due to schedule and resource constraints, which was the case with this feature.

4. Requirements Creep and Redesign

Requirements creep is a constant threat for projects that do no sufficiently lock down requirements properly. This section will discuss the various requirements creep and redesign that took place.

4.1. User Interface

If there's one place where requirements creep tends to show up, it's in the most visible part of the code, which is the user interface. When presenting a product demo 6 months into the project, we learned that InSpa expected concurrent user access to the application by their multiple spa managers. Because our user interface is embedded within a flat excel file with numerous VBA macros, allowing multiple user access simply wouldn't be possible with the UI and database embedded together in the same file. A work-around solution would be to create copies of the excel file for each store. Unfortunately, duplicating the files means duplicating the database as well. This would require additional effort to update the demand forecast for each database. Because of this missed usability requirement, the entire user interface had to be scrapped in favor of a real desktop application that communicates with a database server.

The user interface was quickly ported over to C#, however we still generated the scheduling output in excel files, which users can just save off to disk or delete. Rewriting the user interface also meant that we need a real database server to interact with. We made a decision to port the data base schema over to SQL Server. Rewriting the user interface cost us about a month in redesign and about 4,000 lines of code. However, in the end, it resulted in a much more robust application that was easier to debug and manage vs. keeping the embedded VBA macro-driven Excel application.

4.2. Data Access Layer

The transition from an Excel based application to a C# application with SQL Server database required a significant rewrite of the data access layer (DAL). Because we needed to migrate to a new database, it was decided that a more abstract, transparent solution was needed to avoid future DAL rewrites. What we wanted was a neutral query language that will work with all data providers. Furthermore, we needed

to track data changed in an in-memory copy of the data so we know what data needs to be written back to the Database on a transactional commit. Our solution to this was to utilize the Microsoft Enterprise Library to abstract the database and implement a factory pattern. We wanted a configuration file to hold the simple database connection configuration and the factory to handle the rest of the details. To track data changes, we needed a strongly-typed DataSet to model our data in-memory, manage changes, and bridge between data sources. The answer to all of these requirements is the .NET Language Integrated Query (LINQ) technology. This technology allows for strongly-typed C# queries that are language neutral and use a SQL-like syntax. The LINQ technology turned out to be excellent for our needs. New to the .NET Framework 3.5, LINQ adds strongly-typed reflection to database queries with the syntax and data types checked at compile time. LINQ also allows for anonymous types and in-line lambda expression functions. LINQ ultimately defined our DAL to be a truly transparent data access layer. The strongly-typed lists and DataSets also made it trivial for other software components to pull in and iterate through our in-memory data copy. The DAL database objects are widely used throughout the User Interface as a means of achieving the load/modify/commit/cancel behavior we wanted with the employee and scheduling parameters.

4.3. Final Integration and Testing

After making the changes to the Data Access Layer (DAL) and the user interface, we needed to switch the Scheduling API over to use the new LINQ-based DAL instead of the old data access layer. Immediately after this was done, we ran into new integration bugs. Upon inspection, we discovered that the new DAL only pulled in the demand forecast needed for the scheduling time frame requested by the user. The old data access layer would actually read in the entire demand forecast model into memory. Because our data scope was reduced down to more efficient levels, we uncovered a fundamental flaw in the object model and scheduling algorithm. We were iterating past the end of our user-defined scheduling time frame. This resulted in the application crashing. After debugging the problem we realized a number of problems in the organization of scheduled weeks that begin on Monday and extend until Sunday, which is what InSpa wanted. However, internally, we were organizing weeks from Sunday to Saturday, which is the default for the C# DayOfWeek enumeration object. This discovery triggered refactoring effort of our object model to correct the scheduling overrun and to get the application working as expected. Unfortunately, this last refactoring effort cost us a week of effort right before we were scheduled to hand the project over to InSpa. This resulted in little or no customer beta testing due to schedule constraints.

5. Conclusion

This section covers the final deliverables to InSpa and some final observation on our design and lessons learned.

5.1. Final Product Deliverable

Our final release to InSpa consists of a 2-tier desktop application written in C# with a SQL Server database back end. In total there were over 25,000 lines of code written for the InSpa Scheduling application. The LOC distribution is as follows:

- 50% of code for the Data Access Layer
- 20% of code for Scheduling API and Algorithm
- 16% of code for the User Interface
- 7% of the code for unit tests.
- 7% of the code for prototypes.

5.2. Design Advantages

There are some good examples to point out about the final design of the InSpa Scheduler. The team utilized a lot of standard design patterns to make it extendable and scalable for future development. The simple Scheduling API and strategy design pattern make it easy for a new algorithm to be developed in the future. We would like to see a variation of our greedy algorithm be implemented as a Genetic Algorithm with Local Search optimizations. The API itself is also logically modular for unit testing.

Using an all-Microsoft suite of technologies definitely improved integration. We rarely encountered any problem with the tools and everything worked pretty well together. Using LINQ for our abstract data access layer was a huge advantage for future database platform support. The data access layer will never have to be rewritten when changing databases.

Finally, leveraging the Excel object model for rendering schedules did save us a lot of time and development efforts. In the end, producing excel-based schedules benefits the customer more since it's a common file type that everyone understands.

5.3. Lessons Learned

Over the process of the last 3 months, there were a number of lessons learned from this project. The 3 main lessons learned were in our project management, requirements gathering, and use of technology.

Project Management:

It is difficult to develop a larger scale application with a team that can only work part time virtual with few meetings. Problems would often arise when team members would become overwhelmed with their jobs and personal life that they wouldn't have time to complete their tasks on time as expected. It is entirely unrealistic to estimate completion due dates for software in this kind of environment. All project members worked full time jobs and had busy personal lives, so a project of this size and scope is better suited for fulltime grad students rather than part time grad students. Therefore, the only management solution that would work was to work virtual and coordinate through e-mail. A lesson to be learned here is when you change the entire direction of a project from data mining to solving a scheduling problem; you're opening up a huge schedule risk.

Requirements Gathering:

Probably the biggest lesson learned is to lock down the usability requirements as soon as possible. Requirements creep caused a lot of redesign on this project. We should have elicited hard requirements from the customer and had them sign off on what was being delivered early on the project. We did elicit requirements, but unspoken "assumptions" on concurrent user interaction required some drastic changes to the application late in the game. Agile was a good methodology for showing the customer the application and getting them actively involved in the direction of the project. However, if you don't lock down exactly what the software is and isn't supposed to do; you are just setting yourself up for more work down stream.

Technological Challenges:

The scheduling problem is a difficult one to solve. As such, it took a lot of discussion with Dr. Joslin and InSpa to properly capture the scope of the problem. The biggest lesson learned here is to definitely leverage a domain expert in this field. Another challenge was learning the new technology for this project. The InSpa project is a good showcase of the latest and greatest .NET technologies. As such, there was a slower learning curve for this new technology. That cost the project a lot of time and effort that could have been used for implementing another algorithm, extended software testing, or a heavier research project. In the end, all efforts had to be spent on getting the software up to speed and working.