Simply: Simple Supply

Team Simple

Michael Yeaple
Mustafa Al-Salihi
Curtis Burtner
Jeremy Shulman

Wegmans Team

Mike Shannon
Wendy Altamura
Matt Campbell
Dave Texter
Tony Diaz
Jessi Chichelli
Elier Ruiz

Faculty Coach

Kenn Martinez
Table of Contents

Project Overview
Basic Requirements
Constraints
  Design & Implementation Constraints
  Technology Constraints
Development Process
  Process
  Roles
  Communications
Project Schedule: Planned and Actual
System Design
  System Architecture
    Implementation Design
    Rationale and Breakdown
  Engine
    Implementation Design
    Rationale and Breakdown
  Algorithms
    Sequence Diagram
    Rationale and Breakdown
  Wegmans Import Job
    Implementation Design
    Rationale and Breakdown
  Plan-o-gram Explorer
    Implementation Design
    Rationale and Breakdown
  Customize Explorer Views
    Implementation Design
    Rationale and Breakdown
  Modify Algorithm Parameters
    Sequence Diagram
    Rationale and Breakdown
  Modify and Save Item Distribution
    Implementation Design
    Rationale and Breakdown
  Work from Item Explorer
    Implementation Design
    Rationale and Breakdown
Selecting Rows
Finalizing Items
Run Algorithms

Process and Product Metrics
Overview
Velocity
Hours Invested
Burndown Charts
Cyclomatic Complexity

Product State at Time of Delivery
Project Reflection
Project Overview

Wegmans hires merchant analysts to collate historical item information in order to calculate projected distributions of inventory to push to individual stores proactively rather than waiting until a store is out of stock and needs to have certain products replenished. This is done through the use of a system of spreadsheets and email communication. These spreadsheets contain such item information as how well each store is selling that particular product in addition to how well it has sold during previous seasons. The merchant analysts currently make use of a few calculations based on how far they are into a season to help them build distributions for each product. Once a distribution is created it is literally copied and pasted into another application in order to finalize the distribution. This is a very lengthy process as it needs to be done for every item that Wegmans holds in their warehouse.

Simply aims to help streamline this process by providing merchant analysts a web application to replace this series of spreadsheets which will present the results calculated from their standard distribution calculations in an automated fashion. Additionally, the use of this application will remove the need for copy/pasting finalized distributions into another application as Wegmans will already have this information in their databases.

Basic Requirements

At the beginning of the project, we outlined the basic functionality of our projected final product and created a list of these functional requirements. Through various discussions, some of these requirements were deemed to be out of scope. Our final product includes the following functionality:

1. A web role which allows users to view and interact with the system
2. An engine role which performs calculations for existing algorithms
3. A customizable “portal” for the merchant analysts and warehouse employees to view and interact with the system
4. A means of searching for specific items by attributes deemed appropriate (ex. item name, department, etc.)
5. A means of viewing results of the distribution algorithms
6. A means to manipulate the inputs for the existing distribution algorithms
7. A means of importing data into the system at a single entry point
8. Our working product in the Wegmans environment so they can expand upon/maintain it.

Some of the requirements that we documented in TFS but labeled as out of scope stretch goals include:

1. A means of authenticating users
2. A customizable “portal” for the store operations employees to view future shipments of product to their specific store
3. A means of sorting items by how statistically anomalous the data is for the item
There were also some requirements that were elicited at the beginning of the project that were
decidedly out of scope from the beginning. These included:

1. The ability to add new algorithms through the web interface
2. Localization as Wegmans is based in the East Coast of the United States, and the users
   of this system are expected to have a primary understanding of English
3. Creation of the data itself and/or any test data because it is expected to be delivered by
   the company
4. Creation of methods to import the data from other formats than what is originally
   provided
5. Running the web application under a secure protocol (HTTPS) as the team expects the
   customer will have its own production environment for the final product

Constraints

Design & Implementation Constraints

The design of this project was, for the most part, left in our hands. We did have a meeting with
our sponsor’s architecture team to ensure that our bird’s eye view of the system followed proper
standards that could be easily maintained after the project was handed over to the Wegmans
team. In addition to that meeting, we had a couple code reviews with the Wegmans team in our
first few iterations to generate rapport with the Wegmans team and demonstrate our
implementation of the architecture we had presented to them before. After these meetings, we
took full control of the project but made sure to continue following our upfront design and
architecture.

Technology Constraints

While nothing was mandated in terms of technologies we should use, Wegmans strongly
suggested some that would make our final handoff and the integration to their environment
easier. These technologies included Team Foundation Server, Microsoft Visual Studio,
ASP.NET MVC 4, and Microsoft SQL Server. We looked at potentially using other technologies
but ended up using these as making sure it was easy for them to spin up our application in their
environment was an important concern.

These technologies did not make up our full tech stack. The other technologies we decided to
use include AutoFac, Dapper, and Google Charts. We researched comparable technologies and
presented our findings to our sponsor but ended up choosing these technologies based on their
feedback.
Development Process

Process

Our team’s process for developing the system was Extreme Programming (XP). We added an initial conception phase to account for understanding the problem and business domain. We chose this process because it places a strong emphasis on customer satisfaction and teamwork. The process was approved, but not mandated, by the sponsor. The sponsor was used to more traditional processes such as waterfall. So, the team had a chance to introduce the benefits of iterative processes to the sponsor through this project.

One large component of XP is that you have a customer on-site at all times. However, this was not feasible for us or Wegmans to do due to the nature of the project and their need to continue doing their normal jobs at Wegmans. Instead, we focused on communicating often with the customer in order to display progress and get feedback as early as possible so that changes could be made to accommodate the customer.

Another component of XP is its test-driven nature, this nature manifests itself in the form of unit testing. XP dictates that unit tests shall be written before a feature is developed, ensuring that the feature is properly designed and maintained through the lifetime of the product. At the outset of the project, we believed that unit testing would lead to a higher quality end product and would be well worth the extra effort. While we still believe that consistent unit testing is good practice and leads to a higher quality end product, we abandoned the notion that it was worth the effort as we realized we were not developing quickly enough. We spoke with the sponsor about abandoning unit testing in favor of increasing our velocity and they were in agreement. Our sponsor also admitted that any unit tests were likely to fall by the wayside after the product was delivered.

XP also emphasizes pair programming. During the first semester, most of the product was pair programmed. However, we realized that it also slowed us down. At the end of the first semester, we had a strong framework for us to work within, and we realized that it was worth trading in pair programming on non-complex features of the system for increased speed of development by allowing them to be programmed individually. This allowed us to complete the entirety of the core product by the project hand-off. Had we not done this, we would likely not have made as much progress. Additionally, we made sure that any complex areas of the system were still pair programmed in the second semester, because we did not want to sacrifice quality in these areas.

Each sprint typically flows in the following way:

1. Sprint Planning
   a. Choose user stories
      i. Create validation criteria for each story
      ii. Create tasks and perform estimations

2. Design
Each sprint begins with planning—which actually occurs both before and during the demo of the previous sprint. This involves reprioritizing the backlog, choosing which user stories the team will commit to completing by the end of the sprint, creating validation criteria for each one, and creating the necessary tasks—with estimates of how long it will take to complete the respective task—within TFS. Team members volunteer for the user stories they want to take on so that they own the user story. Only one team member can own a user story; however, multiple people may contribute to a user story when necessary (for example, pair programming is necessary due to the complexity of the story). The planned user stories and their validation criteria are then presented to the sponsor for approval during the planning meeting.

Once the stories have been selected and validation criteria has been created for each story, the design phase begins. At the outset of the project, design always included wireframes for the user-interface and class diagrams for the new features. As the UI and design solidified, we stopped creating wireframes and focused on only diagramming new features if the feature was considered sufficiently complex. During this phase, if any questions or concerns come up, communication occurs with the customer.

After the designs are created, they are then reviewed and approved by the rest of the team. More time is allocated for features that are complex, and less time is given to fixes or adjustments to current features. Occasionally, a design is turned away at this stage and required to be improved and reviewed again.

Once the designs are reviewed, implementation occurs, and any questions or concerns are once again communicated to the customer. Integration occurs as each user story is completed, and then all features of the system are tested individually and together. Any necessary bug fixes are implemented. Then, a demo of each user story is given to the customer, and acceptance testing occurs to ensure the validation criteria has been met. Should the acceptance tests fail in any manner, the user story is carried into the next sprint and completed in addition to the planned user stories for the next sprint.

We initially started with four week sprints in the first semester. These sprints felt long, drawn-out, and resulted in us not making nearly as much progress as we had hoped. It was also difficult to make process improvements to address issues or react to customer feedback that occurred mid-sprint. So, we re-evaluated our process and cut the length of each sprint down to two weeks. This allowed for us to have meaningful meetings with the customer more often, get
better feedback on the implementation of features, and adjust rapidly to any changes deemed necessary.

Roles
There were several initial roles identified by the team. The first of these was the team coordinator/leader, which Michael took on. His responsibilities included coordinating team efforts, identifying work that needs to be done, and ensuring that all necessary work is completed by the appropriate team members. The second of these was the role of sponsor communicator, which Curtis took on. This meant that Curtis became the primary point of contact with the sponsors outside of sponsor meetings. He was in charge of providing status updates to the sponsors via email and communicating any urgent concerns/questions to the sponsor outside of meetings. Initially, Mustafa planned to take on the role of website coordinator—which meant he would have been in charge of updating the artifacts on the department server; however, Jeremy ended up stepping into this role. Mustafa took on the role of the scheduler. He was responsible for creating and updating the project timeline/schedule. Additionally, Curtis and Michael both took on the role of scribe during any meetings that occurred. This meant that they were able to keep track of new user stories, assignment of tasks, and any other relevant information. Information such as new user stories and tasks were then added to TFS by Curtis and Michael as soon as possible. The reason we chose to have two scribes was due to the volume and speed of information that needed to be captured during meetings.

In addition to these roles, each team member had artifacts and responsibilities that they “owned.” Michael owned the deployment document. Curtis owned the risks document and the project metrics. Mustafa owned the testing document. Jeremy owned the design document. Each team member contributed to the previously mentioned documents, but the owner of each document was responsible for making sure that the content was high quality and up-to-date.

Communications
Our process only explicitly defined a single point of communication with the sponsor: the weekly customer meeting. It soon became clear that this was not enough and a weekly email was required to supplement our progress reporting. This email typically contained metrics such as our burndown chart and the number of hours invested for that sprint. It also contained the status of our work, an agenda for the upcoming customer meeting, and any questions which we needed answered. Beyond our regular reporting, emails were leveraged to ask urgent questions, provide support to the sponsor's technical team, and to elicit feedback for certain artifacts. These artifacts were typically academic requirements and necessitated the sponsor’s approval, as it would have a permanent reflection on the sponsor’s image.

Project Schedule: Planned and Actual
Initially, we developed a rough schedule (figure below) based on our understanding of the requirements and scope at the time. The original schedule was developed prior to the establishment of our core process. As a result, the schedule changed entirely. We ended up
with a four week iteration cycle for the first semester and a two week iteration cycle for the second semester.

<table>
<thead>
<tr>
<th>Task Name</th>
<th>Start Date</th>
<th>End Date</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration Zero</td>
<td>08/28/14</td>
<td>09/18/14</td>
<td>16</td>
</tr>
<tr>
<td>General Requirement Elicit</td>
<td>08/28/14</td>
<td>09/18/14</td>
<td>16</td>
</tr>
<tr>
<td>Setup development environment</td>
<td>08/28/14</td>
<td>09/12/14</td>
<td>12</td>
</tr>
<tr>
<td>Formulate team structure</td>
<td>08/28/14</td>
<td>09/05/14</td>
<td>7</td>
</tr>
<tr>
<td>Build website</td>
<td>09/05/14</td>
<td>09/12/14</td>
<td>6</td>
</tr>
<tr>
<td>Choose Technologies</td>
<td>08/28/14</td>
<td>09/12/14</td>
<td>3</td>
</tr>
<tr>
<td>Prioritize features for future iterations</td>
<td>09/11/14</td>
<td>09/15/14</td>
<td>1</td>
</tr>
<tr>
<td>Iteration one</td>
<td>09/15/14</td>
<td>10/02/14</td>
<td>14</td>
</tr>
<tr>
<td>Requirements</td>
<td>09/15/14</td>
<td>09/19/14</td>
<td>5</td>
</tr>
<tr>
<td>Manage requirement change</td>
<td>09/15/14</td>
<td>09/19/14</td>
<td>5</td>
</tr>
<tr>
<td>Project Management</td>
<td>09/19/14</td>
<td>09/22/14</td>
<td>2</td>
</tr>
<tr>
<td>Manage iteration</td>
<td>09/19/14</td>
<td>09/22/14</td>
<td>2</td>
</tr>
<tr>
<td>Reassess project scope and risk</td>
<td>09/19/14</td>
<td>09/22/14</td>
<td>2</td>
</tr>
<tr>
<td>Update software artifacts</td>
<td>09/19/14</td>
<td>09/22/14</td>
<td>2</td>
</tr>
<tr>
<td>Design and Architecture</td>
<td>09/15/14</td>
<td>09/25/14</td>
<td>9</td>
</tr>
<tr>
<td>Prioritize iteration story points</td>
<td>09/15/14</td>
<td>09/22/14</td>
<td>6</td>
</tr>
<tr>
<td>Design system architecture</td>
<td>09/18/14</td>
<td>09/23/14</td>
<td>4</td>
</tr>
<tr>
<td>Review system architecture by sponsor</td>
<td>09/25/14</td>
<td>09/25/14</td>
<td>1</td>
</tr>
<tr>
<td>Implementation</td>
<td>09/23/14</td>
<td>10/01/14</td>
<td>7</td>
</tr>
<tr>
<td>Plan the integration</td>
<td>09/23/14</td>
<td>09/23/14</td>
<td>1</td>
</tr>
<tr>
<td>Implement components</td>
<td>09/25/14</td>
<td>10/01/14</td>
<td>5</td>
</tr>
<tr>
<td>System integration</td>
<td>09/27/14</td>
<td>10/01/14</td>
<td>4</td>
</tr>
<tr>
<td>Testing</td>
<td>09/25/14</td>
<td>10/01/14</td>
<td>5</td>
</tr>
<tr>
<td>Verify Testing approach</td>
<td>09/25/14</td>
<td>09/25/14</td>
<td>1</td>
</tr>
<tr>
<td>Validate build stability</td>
<td>09/30/14</td>
<td>10/01/14</td>
<td>2</td>
</tr>
<tr>
<td>Achieve acceptable mission</td>
<td>09/25/14</td>
<td>10/01/14</td>
<td>5</td>
</tr>
<tr>
<td>Demo and Review</td>
<td>10/02/14</td>
<td>10/02/14</td>
<td>1</td>
</tr>
<tr>
<td>Sponsor demo</td>
<td>10/02/14</td>
<td>10/02/14</td>
<td>1</td>
</tr>
<tr>
<td>Retrospective</td>
<td>10/02/14</td>
<td>10/02/14</td>
<td>1</td>
</tr>
</tbody>
</table>

Original Schedule

We considered the outset of the project to be Iteration 0. During this iteration, the team focused on requirements elicitation and building the framework of the project (tech stack, database, identify the key requirements). This was our first milestone before beginning development work. Our next milestone was to have the skeleton of the project, the engine and web application, built and running on our production server by the end of the term. This would allow for rapid development going into the next term.

At the start of the second semester, we had well established requirements and a solid architecture in place. We modified the schedule to focus on implementation and testing (black box and white box testing). We didn’t formally define milestones. Instead, we used user stories as our milestones. The goal of each iteration was to commit to and complete 25 story points worth of user stories. At the end of nearly every two week iteration, we successfully completed all of the user stories that we committed to. In the event that we did not complete one of the
user stories, the user story got carried into the next iteration and was labeled as a high priority item to make sure it was completed during the iteration in addition to our planned 25 story points for that iteration.

**System Design**

In this section, we shall discuss the overall architecture of the system and the design of some of the important features in the application. Please note that the UML models in these diagrams may not show every attribute and method of the respective class in the codebase. If we were to show everything, the images would be too cluttered to display the important and relevant points. As a result, irrelevant attributes and methods to the feature being discussed are not displayed in the diagrams.

**System Architecture**

**Implementation Design**

![Diagram of System Architecture](image.png)

**Rationale and Breakdown**

Above is a genericised diagram of the architecture. The three main pieces of our architecture are the web application, the engine, and our database. We split it up this way because the web application should be able to be used independently of the engine. This also allows for the application to be more performant and for it to scale easily.
The engine is responsible for running the computations required for determining each item’s store distribution via the job subsystem—which is described in further detail in the Engine section. We chose to design a job subsystem because we wanted to be able to kick off events with a “fire and forget” type pattern. Each job asynchronously completes one action which is the responsibility of the engine. These jobs are each treated as a transaction. This way we do not have to worry about the web application hanging while the engine calculates. It is not expected that anyone should be accessing the engine via the web browser; hence, the lack of other controllers and views. The API will provide an access point for the Wegmans scheduler and the web app to easily kick off jobs.

The web application is responsible for allowing users to view detailed data, distributions calculated by the engine--which have been saved in the database by the engine--and saving finalized distributions after they have been modified and approved by a merchant analyst. It follows a standard MVC model. MVC is a standard, tried and true approach to building stable web applications within a framework. One of the main reasons for this is that Wegmans commonly uses MVC for their applications so designing it in this way will allow them to more easily integrate our application into their code base as we'll be following similar conventions. Additionally, ASP.NET comes pre-configured with the MVC framework which is very easy to get up and running with.

Providers are responsible for all interaction with the database. They query the database via stored procedures and use Dapper to map the results into C# objects (data models) for us to use within the application. Data models simply hold data and have very little to no functionality. Our decision to separate database operations into the providers allows us to keep the data conceptually separate from the population of data objects. It also isolates our database interactions to one area of the system, creating a better separation of concerns. It should be noted that providers and data models are a shared class library between the web app and the engine. The separation of the providers and data models into separate class libraries is done to adhere to the DRY principle, because the web application and engine use a lot of the same data and database interactions. That being said, at runtime, the web app and engine have separate instances of these class libraries, because the web app and the engine are deployed as completely separate web applications.

The decision to use an ORM was for ease of use and speed of development, without sacrificing much control, if any, over the database interactions since we chose to use stored procedures. Dapper was chosen as our ORM because it was very easy to get up and running and to work with. It was also approximately twice as fast as other comparable ORMs such as NHibernate--which was considered, but had too much setup overhead and would have added unnecessary complexity to the project. The only downside to using Dapper is that complexly nested data requires some manual effort to map into C# objects.

Stored procedures were chosen for all database interaction for a few reasons. The first is for security. They reduce the possibility of SQL injection. Additionally, they are faster, cacheable, reduce network traffic, and make it easier for the developers to work with data in the database.
Finally, Wegmans recommended the use of stored procedures since they typically use them in their applications as well.

**Engine**

**Implementation Design**

**Rationale and Breakdown**

Within the engine, there is a web API—the JobController—so that both the web application and the Wegmans scheduler can kick off jobs. A command pattern used for the scheduling and execution of jobs. This was chosen to encapsulate the functionality of each job so the invoker
could determine how to schedule the job for execution (this could be done with the use of a message queue, threaded requests, etc.). The data access providers are reused from the web application.

We wanted to decouple the computation from the presentation of the distribution information which is why the engine is separate from the web application, but we also wanted to have the web application be able to ensure it had up-to-date information. The solution we came up with was to allow the web application to start jobs. The jobs would log their progress in a job table in the database via a job provider, which would allow the web application to poll for the progress but still keep our separation of concerns. There is no communication from the engine to the web application other than to give a success response when a job starts, meaning that jobs can run asynchronously when kicked off from the web application and the end-user can continue to work while the job is running.

**Algorithms**

**Sequence Diagram**

![Sequence Diagram](image)

**Rationale and Breakdown**

We wanted an outline of exactly how a job would execute an algorithm and log its progress in the database specifically. The resulting sequence diagram follows the design of the engine as expected.

When the web application is polling for job status, we wanted the result to be as close to a concrete percent as possible. Brainstorming how to do this, we found that it would be difficult to estimate that. Instead we came up with a solution that provides useful status information even though it isn’t necessarily exact. A job that is computing an algorithm will update its percent completion every time it processes an item. It will begin by logging that the job has started through the use of the job table. After getting any required information from the providers it will begin to iterate over each item being processed. After a distribution has been created for an
item, the job’s status will be updated. Finally, the job will log when it is done saving any distribution results. Note that saving will be completed in one batch job so progress will not be updated while saving is taking place. Rather, it will occur after the database transaction is complete.

Wegmans Import Job

Implementation Design

Rationale and Breakdown

In order to move Wegmans’ data to our database, Wegmans is responsible for inserting the data they want into their schema in our DB. Then the WegmansParseData job is kicked off on the Engine through the Engine’s API. This job will call a stored procedure which is responsible for wiping our databases of their data, and then replacing it with the new data. The data wipe clears everything Wegmans is responsible for providing, from item data, to sales data, to historical data, to store data. We preserve distribution information, so as not to erase any work done by buyers. After the Simply tables have been wiped and the new Wegmans data has been imported, the job then kicks off the other algorithm jobs with default settings to create default distributions for all items across all stores.

The advantage of this system is that it provides Wegmans with an easy point of integration. If they want data to appear, they need only to insert it into our staging tables. Bad data does not have to be traced back to a bug in a complex parser in our system, which means Wegmans maintains code which they’ve written.
This system has the added benefit of being simple. The SQL scripts are easily understood and easy to modify. Any schema changes will most certainly require an adjustment in any of the import scripts, but this cost is mitigated by the fact that the required change will certainly be easy to find and easy to make.

**Plan-o-gram Explorer**

**Implementation Design**

```
PlanogramController
  PlanogramExplorer(PlanogramProvider)
    + PlanogramExplorer(): ActionResult

PlanogramProvider
  Use
    + GetPlanogram(instrActionOrder: x, int season: x, string searchText: x)

PlanogramExplorerModel
  Use
    + PlanogramExplorerModel: PlanogramProvider

Planogram
  Use
    + ID: int
    + name: string
    + SaleStart: DateTime
    + EndSale: DateTime
    + Aggregates: Data
    + Aggregates: Data

Columns Preferences
  Use
    + Grid: GridModel
    + ColumnPreferences: ColumnPreferencesModel

GridModel
  Use
    + Grid: GridRowModel
    + Header: GridRowModel

GridRowModel
  Use
    + Cell: GridCellModel
    + Active: string
    + InHeader: bool

GridCellModel
  Use
    + Cell: string
    + Active: string
    + Data: string
```

**Rationale and Breakdown**

The plan-o-gram explorer is a fairly straightforward component. The design of this component is extremely similar to that of the item explorer and the season explorer. When a request is made to the PlanogramController’s PlanogramExplorer method, the planogram data and aggregate data is requested from the PlanogramProvider’s GetPlanogram method. The data is then sent to the PlanogramExplorerModel, which compiles the data into an easily usable format for the view. The GridModel is used for creating the consistent grid seen across each of the explorer views (and the grids seen in the item details page). Column preferences are also taken into account, which are discussed in another section of this document.
Rationale and Breakdown
Customizing explorers within the application entails allowing a user to change the order and visibility of columns on each explorer view. We wanted a user to not only be able to do this for the current session, but we wanted the changes to persist over many sessions. This required a structure that allowed us to store this configuration in the database. We also wanted to make the configuration generic enough that it could apply to many of the views within the application and not be limited to one single explorer.

The general way in which the column preferences work is that a form is posted to the AccountController’s SaveColumnPreferences method, which then tasks the ColumnPreferenceProvider with saving these preferences into the database. Then, when a user makes a request the respective controller and action method associated with the column preferences, the ColumnPreferenceProvider’s GetColumnPreferences method is called to return a list of ColumnPreference objects determining how the view should be displayed. These are sorted into shown and hidden columns lists in the ColumnPreferenceModel within the view model associated with the controller and action method being called so that they can be used easily within the view for organizing the view and for populating the edit column preferences form.
It is important to note that user accounts are considered to be out of scope for this project. That being the case, the current implementation will result in global changes whenever someone changes the column preferences. In the future, ColumnPreference objects will have a reference to a specific user ID so that the changes are only visible to the user who made the changes.

The alternative to this was to use JavaScript to rearrange the columns on the fly within the page and store the settings for the duration of the user’s session; however, that did not offer the persistence that we wanted to offer to the users.

**Modify Algorithm Parameters**

Sequence Diagram

```
Modify Algorithm Parameters
Rationale and Breakdown
The above sequence diagram depicts the process for modifying an algorithm’s calculation parameters. First, the user clicks a button on the page which calls a jQuery function to create a popup from a hidden form. On this form, the user chooses the algorithm he or she wishes to use, and then fills in the inputs for the calculation parameters he or she wishes to overwrite. The user then clicks the “Run Algorithm” button, which validates the form and then makes an AJAX POST request to the engine to kick off a job with the supplied calculation parameters. The engine kicks off a job and returns a JobResponse with the ID of the job which was just created.
```
Finally, the Job ID is stored in JavaScript session storage for use elsewhere (e.g. getting the job status and creating a progress bar to display the job’s progress).

The code for this user story is all front-end (HTML, CSS, JavaScript) code. Reusable functions are all located in the app-function.js script file, whereas, one-off logic such as attaching these functions or managing form elements with JavaScript is done within the respective view.

It made a lot of sense for us to keep the logic for this all on the front-end in order to create a more responsive experience. We decouple the engine from the web application, and we also gain performance in the web application through this implementation. Jobs can run asynchronously while the user does other work in the system and then returns to the item(s) they ran an algorithm on.

The alternative to this design was to separate out the algorithm code into a reusable class library and run the algorithm through the web application upon a user request. That being said, there would be no performance gain or improvements from doing it this way. In fact, we could introduce concurrency issues that are not easily handled by doing it this way (e.g. a race condition resulting from a job running an algorithm an item and a user running the same algorithm on the same item with different parameters).

**Modify and Save Item Distribution**

**Implementation Design**
Modifying and saving an item distribution is slightly tricky. To start with, we wanted to be able to modify a distribution within our GridModel. The reason for this is that each row in the GridModel is associated with a different store. This allowed us to keep our distribution amounts in the same rows as their store information, providing the user with a more cohesive and fluid experience. In order to accommodate this need, the GridModel had to be modified to be able to hold input fields and be placed within a form, if necessary. To make the GridModel into a form, three attributes were added to the GridModel class: FormController, FormAction, and FormMethod. If FormController and FormAction are both populated, the GridModel in the view will be wrapped in a form, using the FormController, FormAction, and FormMethod (GET, POST [This is a System.Web.Mvc enum]) attributes to populate the form HTML element appropriately. The other change that had to be made was to GridCellModel. ModelName, ModelRow, and GridCellType were all added to GridCellModel. ModelName, ModelAttribute, and ModelRow will be used to...
appropriately attach the correct name for the HTML input so that it posts to the server correctly as can be seen in the following article:

http://www.binaryintellect.net/articles/b1e0b153-47f4-4b29-8583-958aa22d9284.aspx

GridCellType is simply an enum which indicates whether a cell is a Text value, an Input, or a HiddenInput (the previous Pascal case words indicate the enum names). This allowed us to generate the correct HTML in the View when using the CellModel. Saving the item distribution is done by posting the GridModel form to the /Item/SaveDistribution method (POST only). This method takes a list of DistributionModel. The list of DistributionModel is then converted to a List of Distribution and saved to the database via the ItemProvider. The saved distribution is assigned a DistributionType of 4 (WorkInProgress) in the database, meaning that the distribution has not been finalized yet. Validation of the distribution in order to make sure that it is a valid distribution is done on the client side via jQuery and also on the server side.

**Work from Item Explorer**

**Implementation Design**

```plaintext
ItemController
- ItemProvider : ItemProvider

+ FinalizeItems(string itemIds, DistributionType distributionType, FinalizedDistributionType finalizedDistributionType, int pickWeek, int shipWeek)

ItemExplorerModel
+ Grid : GridModel
+ ColumnPreferences : ColumnPreferenceModel
+ IsFilteredToPlanogram : bool

ItemProvider

+ FinalizeItems(List<int> itemIds, DistributionType distributionType, FinalizedDistributionType finalizedDistributionType, int pickWeek, int shipWeek, string currentUser)

DB
```

**Rationale and Breakdown**

The class diagram above depicts the back-end (C#) classes involved in working from an ItemExplorer level. There are several components to this user story. They have been broken down into the sections that follow.

**Selecting Rows**
The ItemExplorerModel was the only class affected by this. Selecting rows was done by prepending checkbox inputs into the header and the item rows upon creating the GridModel for the Item Explorer. Hence, the reason there are no extra attributes or methods shown in the ItemExplorerModel. These checkboxes are disabled if the user is not filtered to a single planogram or the item has been finalized. A JavaScript function (Simpply_GridModelSelectAll in app-function.js) is used to select all non-disabled checkboxes if the checkbox in the table header was clicked. Another JavaScript function (Simpply_GridModelGetSelectedRows in app-function.js) is used to get all selected rows for use in the following sections.

**Finalizing Items**

Finalizing items is fairly straightforward. A JavaScript function (Simpply_FinalizeSelected in app-function.js) is used to first get the selected rows (using Simpply_GridModelGetSelectedRows in app-function.js) and then to create a pop up prompting the user to choose a distribution algorithm to finalize for the selected items. When the user chooses an algorithm, an AJAX request is fired off (Simpply_Finalizeltems in app-function.js) calling the ItemController’s Finalizeltems function. This, in turn, calls the ItemProvider’s Finalizeltems function, which calls the Finalizeltems stored procedure on the database.

**Run Algorithms**

Running algorithms was developed from the ground-up with working from both the item explorer and the item detail page in mind. A JavaScript function (Simpply_RunAlgorithmFromItemExplorer in app-function.js) gets the Item IDs from the selected rows and passes them to the run algorithm JavaScript function (Simpply_RunAlgorithmPopup in app-function.js). This function renders a modal window from a hidden form for users to select an algorithm and change the inputs. When the submit button is clicked, an AJAX request is fired off to the engine’s JobController—which is an API controller—and then the JobController kicks off the respective job with the parameters passed to it via the job subsystem, described elsewhere in this document.

**Process and Product Metrics**

**Overview**

At the outset of the project, we selected a handful of metrics to track the quality of product we were delivering, and the quality of the process we were using to deliver it. As the project progressed, we quickly stopped gathering these metrics, as they were not viewed as useful to the improvement of our product or process.

Although we did not gather and maintain all of the metrics we had originally planned to, there were three which we maintained and leveraged throughout the course of the project. Those three metrics were:

- Velocity
- Hours invested
- Burndown charts

We did not collect and log any code metrics as the project progressed. While it would have been useful and interesting to track, it is difficult to say whether we would have acted on the information. Although the project is wrapping up, we have gathered cyclomatic complexity.
statistics across three of our project folders so that we are able to at least reflect on the quality of product we are delivering. Those three systems on which we gathered cyclomatic complexity are:

- Web Role
- Engine
- Data Access

**Velocity**

Velocity, in conjunction with story-points, is a metric typical to agile teams. It represents how many story points are completed within a single sprint, and is a reflection of the team’s development “speed”. For our project, we leveraged user stories and then assigned those stories story-points. We then decided on the velocity we wanted to reach in a sprint and put enough stories into the sprint such that completing all stories would yield the desired velocity. At the end of the first term, we realized we weren’t going to deliver enough of the product at our current development speed. With the velocity metric as our guide, we aimed to double it going into the second term.

Below is a chart measuring our velocity from sprint (iteration) to sprint. However, velocity by itself does not give a full picture of our increase in development speed from term 1 to term 2. Velocity is simply a measure of story points completed per iteration and because we reduced the length of our iterations going into the second term. A change in iteration length implies a change in the number of story points completed per week, but not a change in the velocity itself. Due to this, we have included the number of story points completed per week alongside our velocity measurement.

Included as well is an “Hours Per Point” and “Hours in Product” metric. Hours in product represents the total number of man hours spent to reach the designated velocity, while hours per point represents the number of hours spent per story point. The objective with these metrics is to show that our increased velocity was not just a matter of assigning easier stories a greater value (which would result in a low Hours Per Point measurement), but was a result of an improvement in our development process.

To reiterate, only velocity was gathered on an iteration by iteration basis. This metric was used to aim for the development speed which would deliver a satisfactory amount of product. All other metrics listed were gathered at the end of the term in order to provide a clearer picture of our increased velocity.

<table>
<thead>
<tr>
<th>Iteration #</th>
<th>Date Start/End</th>
<th>Velocity (Story Points)</th>
<th>Iteration Length (Weeks)</th>
<th>Story Points/Week</th>
<th>Hours in Product</th>
<th>Hours Per Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>October 16 - November 13</td>
<td>18</td>
<td>4</td>
<td>4.5</td>
<td>32</td>
<td>1.78</td>
</tr>
<tr>
<td>2</td>
<td>November 14 - December</td>
<td>14</td>
<td>4</td>
<td>3.5</td>
<td>99.170</td>
<td>7.1</td>
</tr>
</tbody>
</table>
The first thing to note from the metrics above is that our velocity did increase from term 1 to term 2. This is seen in the velocity spike from iteration 2 to iteration 3. The story points per week metric shows that we did not just increase our velocity by only a few points from term 1 to term 2, the increase was actually more than double when taken in conjunction with the shorter sprints. Finally, the hours per point metric shows that

The final iteration listed here, iteration 8, represents a break-down in our process as we started wrapping up. We stopped focusing on delivering more features, so we didn’t story-point the academic work and bug fixes we were now investing our time into. This iteration should be considered an outlier.

## Hours Invested

An important, real-world metric which we were required to gather is the amount of time we’ve invested into the product. This metric gave our sponsors, our coach, and the members of our team a clear picture of the effort being put into individual tasks, and the project overall.

While not required by the department, hour estimates for individual tasks drove our burndown charts. They were also easy to do and gave us a general sense of where we were in the iteration.

Hours slipped and percent slippage are not metrics we gathered throughout the course of the two terms. These metrics will, however, allow us to reflect on our estimation accuracy. Hours slipped is the number of hours our actual went over our estimate, and percent slippage is what percentage of our actual the slipped hours make up.
<table>
<thead>
<tr>
<th></th>
<th>Date Range</th>
<th>Total Hours Invested</th>
<th>Remaining Hours</th>
<th>Remaining/Actual Proportion</th>
<th>Percentage Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>October 16 - November 13</td>
<td>108</td>
<td>121.3</td>
<td>13.3</td>
<td>10%</td>
</tr>
<tr>
<td>2</td>
<td>November 14 - December 12</td>
<td>212.7</td>
<td>237.73</td>
<td>25.03</td>
<td>11%</td>
</tr>
<tr>
<td>3</td>
<td>January 26 - February 12</td>
<td>89.58</td>
<td>98.93</td>
<td>9.35</td>
<td>9.45%</td>
</tr>
<tr>
<td>4</td>
<td>February 13 - February 26</td>
<td>68.25</td>
<td>72.58</td>
<td>4.33</td>
<td>5.97%</td>
</tr>
<tr>
<td>5</td>
<td>February 13 - February 26</td>
<td>67.9</td>
<td>97.14</td>
<td>29.24</td>
<td>30.10%</td>
</tr>
<tr>
<td>6</td>
<td>February 13 - February 26</td>
<td>77</td>
<td>86.97</td>
<td>9.97</td>
<td>11.46%</td>
</tr>
<tr>
<td>7</td>
<td>April 02 - April 16</td>
<td>64.7</td>
<td>74.5</td>
<td>9.8</td>
<td>13.15%</td>
</tr>
<tr>
<td>8</td>
<td>April 16 - April 30</td>
<td>112.5</td>
<td>114.05</td>
<td>1.55</td>
<td>1.36%</td>
</tr>
</tbody>
</table>

Note that these numbers do not include tasks which were considered to be “outside” an iteration. These tasks usually came in the form of unplanned project management activities, such as communication efforts.

We can see that, barring a single outlier, our estimates were never more than 15% off of the actual hour investment required. Also observe that our per-week hour investment remained fairly steady at around 40 hours per 7 days. Recall from the previous set of metrics that we were delivering more story-points per iteration after the first term. These metrics imply that we were not necessarily investing more hours to deliver more story points, we simply improved our process such that we were able to deliver more story points per iteration.

The total number of hours invested into the project as a whole is listed below.

All Iterations: 1214.78

**Burndown Charts**

Burndown charts were leveraged to aid us in visualizing our progress during an iteration. They were shared with our sponsors and our coach. The burndown charts were driven by the hours estimated to be remaining for all tasks assigned to an iteration. From the charts below, we can see that we would not do the estimates for our story tasks until a few days into an iteration, as we typically rested the weekend that an iteration began. During our first iteration, we were not leveraging our project management tool correctly, which made our burndown chart fairly meaningless. Going into the second iteration and beyond, we improved our process such that our tool would provide us meaningful charts.
Cyclomatic Complexity
At the outset of the project, we had chosen this metric as a means to track the quality of code which we were delivering. We did not take the time to gather this metric throughout the course of the project, we have pulled these numbers together for a brief reflection.

Web Role Metrics:

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Complexity Min</th>
<th>Complexity Max</th>
<th>Complexity Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>29</td>
<td>1.86</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>26</td>
<td>1.943</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>31</td>
<td>1.94</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>32</td>
<td>1.955</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>34</td>
<td>1.913</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>48</td>
<td>1.9</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>54</td>
<td>1.93</td>
</tr>
</tbody>
</table>

Engine Metrics:

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Complexity Min</th>
<th>Complexity Max</th>
<th>Complexity Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
<td>1.094</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>8</td>
<td>1.1653</td>
</tr>
</tbody>
</table>
At time of delivery, our main systems have an average cyclomatic complexity below 2. This is a good number which implies a high quality, easy to maintain code base. What the number literally means is that, on average, our methods have very few paths through them. However, within the web role there is a method with an extremely high cyclomatic complexity of 54. This method will be difficult for Wegmans to maintain in the future and should be refactored as soon as possible.

**Product State at Time of Delivery**

At the time of delivery, no planned features are missing. This has been agreed upon by the customer. In addition to the planned features, several stretch features were completed. These include: a mobile-friendly view, finalize item information, and column preferences. All other stretch goals were documented so that they could be accomplished at a later date after the system was handed off. Some examples of these stretch goals include: a store view, view statistically anomalous items, and refactorable areas of the system.

The product is deployed and running within the Wegmans environment. A lot of effort was put into making sure this happened before the project hand off. This was done through the creation of an extensive deployment document, remote deployment support via email, and on-site
meetings to ensure the application was deployed and running, as well as to perform knowledge sharing with the future maintainers of the product. A design document and test scripts have also been generated by the team for the customer to refer to.

Project Reflection

Many things went well during this project, many things were improved from one term to the next, and a few things didn’t go well. Right from the start, the dynamic between our team and the sponsor team was great. Their team was composed of end users who were experienced in the requirements gathering process, and technical members who were familiar with software development. Above all, everyone was excited to be a part of the project and wanted to see it successfully delivered in the end. This relationship with our sponsor was the foundation for our successes throughout the rest of the project.

We were happy that we were able to deliver all the core features, along with some stretch features. Additionally, we made vast improvements to the UI after the end of the first term. The sponsor was extremely pleased with the results of the UI overhaul, as were we. Another success of the project was the code reviews. They went extremely well, to the point that the Wegmans team only pointed out minor alterations from their coding conventions in our code. Finally, we were proud of the fact that that we made our meetings much more productive in the second semester due to better planning.

We have a few regrets with regards to the project. The first of these is that we wish we had done our on-site shadowing sooner. We feel that we would have had a better understanding of the problem and domain earlier on if this had been the case. This could have potentially bought us more time to focus on the product instead of the problem. Another thing we could have done better is testing. We were really focused on developing all of the core features that the customer wanted. As a result, we had to take time away from testing the product in order to deliver the entire core experience. Finally, while we gathered useful metrics throughout the course of the project, some of the metrics we gathered for this document would have been useful to have during the course of the project to aid us in improving our process.

Overall, we strongly believe that this project has been a huge success. Our sponsor has expressed extreme satisfaction to us multiple times with regards to the product's core features and the stretch features. Even more importantly, we are proud of what we have accomplished and what we have learned over the course of this project.