

A Client-Server Architecture for the Collection of Game-based Learning Data

James Robert Jones

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Osman Balci, Chair
James D. Arthur
Anderson H. Norton III

December 8, 2014

Blacksburg, Virginia

Keywords and phrases: Digital educational games, game-based learning, client-server architecture, iOS mobile software engineering, real-time data collection and display

A Client-Server Architecture for the Collection of Game-based Learning Data

James Robert Jones

ABSTRACT

Advances in information technology are driving massive improvement to the education industry. The ubiquity of mobile devices has triggered a shift in the delivery of educational content. More lessons in a wide range of subjects are being disseminated by allowing students to access digital materials through mobile devices. One of the key materials is digital-based educational games. These games merge education with digital games to maximize engagement while somewhat obfuscating the learning process. The effectiveness is generally measured by assessments, either after or during gameplay, in the form of quizzes, data dumps, and/or manual analyses. Valuable gameplay information lost during the student's play sessions. This gameplay data provides educators and researchers with specific gameplay actions students perform in order to arrive at a solution, not just the correctness of the solution.

This problem illustrates a need for a tool, enabling educators and players to quickly analyze gameplay data. in conjunction with correctness in an unobtrusive manner while the student is playing the game. This thesis describes a client-server software architecture that enables the collection of game-based data during gameplay. We created a collection of web services that enables games to transmit game-data for analysis. Additionally, the web application provides players with a portal to login and view various visualization of the captured data. Lastly, we created a game called "Taffy Town", a mathematics-based game that requires the player to manipulate taffy pieces in order to solve various fractions. Taffy Town transmits students' taffy transformations along with correctness to the web application. Students are able to view several dynamically created visualizations from the data sent by Taffy Town. Researchers are able to log in to the web application and see the same visualizations, however, aggregated across all Taffy Town players. This end-to-end mapping of problems, actions, and results will enable researchers, pedagogists, and teachers to improve the effectiveness of educational games.

ACKNOWLEDGMENTS

My wife, Christie, has given me the strength and confidence that I can do the impossible. I thank her for her unconditional support, uplifting words, and many sacrifices.

Dr. Osman Balci deserves a medal for all the professional guidance given to me at Virginia Tech. He is not only my advisor, but an invaluable repository of amazing advice and support. Dr. Balci took a risk by being my adviser, even though I worked full-time. Thank you for seeing the potential in me. Thank you for pushing me to release that potential.

I would like to also thank my committee members Dr. James D. Arthur and Dr. Anderson Norton for the guidance in creating this body of work. Dr. Norton's passion for improving our educational system is inspiring.

Lastly, I would like to thank my mom and dad. Throughout this process, my mom has given nothing but unconditional support. Without my dad enabling exploration in the field of computer science at a young age, I would not be here today.

TABLE OF CONTENTS

ABSTRACT	ii
ACKNOWLEDGMENTS	iii
LIST OF FIGURES	vi
LIST OF TABLES	vii
LIST OF ACRONYMS	viii
CHAPTER 1: Introduction	1
1.1 RELATED WORK	1
1.1.1 Existing Digital Educational Games	2
1.1.1.1 Early Jamestown	2
1.1.1.2 MathBoard.....	2
1.1.2 Researching Assessment in Games	3
1.1.2.1 Crystal Island.....	3
1.1.2.2 Cool-it.....	4
1.2 STATEMENT OF THE PROBLEM	4
1.3 STATEMENT OF OBJECTIVES	5
1.4 OVERVIEW OF THESIS	5
CHAPTER 2: Software Architecture	6
2.1 ARCHITECTURAL PATTERNS	6
2.1.1 Client-Server Architecture	6
2.1.2 Two-Tier	6
2.1.3 Multi-Tier.....	7
2.1.4 Service-Oriented Architecture	8
2.1.5 Representational State Transfer.....	9
2.2 MOBILE APPLICATION STATISTICS SERVICE	9
2.2.1 Java Enterprise Edition	11
2.2.2 IBM WebSphere Application Server.....	12
2.2.3 IBM DB2	12
2.2.4 Mongo DB.....	13
CHAPTER 3: Taffy Town Educational Game	14
3.1 TITLE SCREEN.....	15
3.1.1 Player Account Creation	15
3.1.2 Player Authentication	16
3.2 TOWN SCREEN.....	19
3.3 WORKSPACE SCREEN	21
3.3.1 Taffy Interactions	21
3.3.1.1 Translation.....	21
3.3.1.2 Slice.....	23
3.3.1.3 Stretch and Shrink.....	23
3.3.1.4 Copy and Paste.....	23
3.3.1.5 Delete.....	23
3.3.1.6 Painting.....	23
3.3.1.7 Glue.....	27
3.3.1.8 Workspace Reset.....	27
3.3.1.9 Patching the Hole.....	27
CHAPTER 4: MASS Web Application and Services	29
4.1 MASS AUTHENTICATION	29
4.1.1 Account Types.....	30

4.1.1.1	Player.....	30
4.1.1.2	Administrator.....	30
4.2	MASS DASHBOARD.....	30
4.2.1	<i>Data Visualizations</i>	32
4.2.1.1	Actions Stacked Bar Chart.....	32
4.2.1.2	Accuracy Pie Chart.....	34
4.2.1.3	Question Type Stacked Bar Chart.....	34
4.3	WEB SERVICES.....	34
4.3.1	<i>App Web Service</i>	35
4.3.1.1	Create.....	35
4.3.1.2	Retrieve.....	36
4.3.1.3	Delete.....	37
4.3.2	<i>Group Web Service</i>	37
4.3.2.1	Create.....	37
4.3.3	<i>User Web Service</i>	38
4.3.3.1	Create.....	38
4.3.3.2	Retrieval.....	39
4.3.3.3	User Authentication.....	40
4.3.3.4	User App Registration.....	41
4.3.3.5	List User Registered Apps.....	42
4.3.4	<i>Statistics</i>	42
4.3.4.1	Create.....	42
4.3.4.2	Retrieval.....	44
CHAPTER 5: Self-Evaluation of Client-Server Solution.....		45
5.1	FUNCTIONALITY.....	45
5.2	USABILITY.....	45
5.2.1	<i>Taffy Town</i>	45
5.2.2	<i>MASS</i>	46
5.3	RELIABILITY.....	46
5.3.1	<i>Taffy Town</i>	46
5.3.2	<i>MASS</i>	46
5.4	PERFORMANCE.....	46
5.4.1	<i>Taffy Town</i>	46
5.4.2	<i>MASS</i>	47
5.5	MAINTAINABILITY.....	47
CHAPTER 6: Conclusions and Future Research.....		49
6.1	CONCLUSIONS.....	49
6.2	CONTRIBUTIONS.....	49
6.3	FUTURE RESEARCH.....	49
6.3.1	<i>Taffy Town</i>	49
6.3.2	<i>MASS</i>	50
REFERENCES.....		51

LIST OF FIGURES

FIGURE 1. A SCREENSHOT OF EARLY JAMESTOWN GAME [APPLE 2010]. HTTPS://ITUNES.APPLE.COM/US/APP/EARLY-JAMESTOWN/ID395229194?MT=8 USED UNDER FAIR USE, 2014.	2
FIGURE 2. MATHBOARD BY PALASOFTWARE [PALASOFTWARE 2014]. HTTP://WWW.PALASOFTWARE.COM/MATHBOARD.HTML USED UNDER FAIR USE, 2014.	3
FIGURE 3. IMAGE OF CRYSTAL ISLAND [NORTH CAROLINA STATE UNIVERSITY 2014]. HTTP://WWW.INTELLIMEDIA.NCSU.EDU/CRYSTAL-ISLAND-OUTBREAK USED UNDER FAIR USE, 2014.	4
FIGURE 4. TWO-TIER CLIENT-SERVER ARCHITECTURE.	7
FIGURE 5. A MULTI-TIERED ARCHITECTURE.	7
FIGURE 6. SERVICE-ORIENTED ARCHITECTURE PRIMARY COMPONENTS.	8
FIGURE 7. MASS ARCHITECTURE.	10
FIGURE 8. JAVA EE CONTAINER ARCHITECTURE [GONCALVES 2010].	11
FIGURE 9. JAVA EE SPECIFICATIONS SUPPORTED BY WAS [ALBERTONI ET AL. 2013]. USED UNDER FAIR USE, 2014.	12
FIGURE 10. TAFFY TOWN’S SCENE STORY BOARD.	14
FIGURE 11. TAFFY TOWN TITLE SCREEN.	15
FIGURE 12. AUTHENTICATION POP-UP WINDOW.	16
FIGURE 13. LOGIN ERROR POPUP.	17
FIGURE 14. TAFFY TOWN ACCOUNT CREATION POP-UP WINDOW.	18
FIGURE 15. SUCCESSFUL ACCOUNT CREATION ALERT PANEL.	18
FIGURE 16. ACCOUNT CREATION FAILURE ALERT PANEL.	19
FIGURE 17. TAFFY TOWN ON A BRIGHT, SUNNY DAY.	20
FIGURE 18. TAFFY TOWN BEING ATTACKED.	20
FIGURE 19. WORKSPACE SCREEN ELEMENTS.	22
FIGURE 20. TRANSLATING A PIECE OF TAFFY.	22
FIGURE 21. PERFORMING THE SLICE GESTURES ON A PIECE OF TAFFY.	24
FIGURE 22. VERTICAL STRETCH INTERACTION.	24
FIGURE 23. COPY AND PASTE USER INTERACTION.	25
FIGURE 24. PAINTING A PIECE OF TAFFY GREEN.	25
FIGURE 25. ACTIVATED COLOR PALETTE.	26
FIGURE 26. TAFFY PIECES PAINTED WITH SEVERAL COLORS.	26
FIGURE 27. CORRECTLY PATCHING THE HOLE CAUSES THE HOLE TO FADE AWAY.	28
FIGURE 28. MASS WEB APPLICATION LOGIN PAGE.	29
FIGURE 29. MASS AUTHENTICATION ERROR.	30
FIGURE 30. MASS DASHBOARD.	31
FIGURE 31. MASS DASHBOARD WITH NO GAME DATA COLLECTED.	31
FIGURE 32. RESEARCH ADMIN’S AGGREGATE RESULTS ON THE DASHBOARD.	31
FIGURE 33. STACKED BARCHART DIPCING VARIOUS ACTIONS A TAFFY TOWN PLAYER PERFORMED.	33
FIGURE 34. PIE CHART SHOWING THE DISTRIBUTION OF CORRECT AND INCORRECT ANSWERS.	33
FIGURE 35. STACKED BAR CHART CORRELATING QUESTION TYPE AND ACCURACY.	33

LIST OF TABLES

TABLE 1. RESTFUL WEB SERVICE HTTP INTERFACE METHODS.....	9
TABLE 2. HTTP VERB DESCRIPTIONS.....	34
TABLE 3. CLASSIFICATION OF HTTP STATUS CODES.....	35
TABLE 4. HTTP POST REQUEST TO CREATE AN APP.....	35
TABLE 5. HTTP RESPONSE CREATING AN APP.....	36
TABLE 6. GET REQUEST TO RETRIEVE APP FROM THE APP WEB SERVICE.....	36
TABLE 7. GET RESPONSE FROM RETRIEVING AN APP FROM THE APP WEB SERVICE.....	36
TABLE 8. HTTP DELETE REQUESTING TO DELETE THE SPECIFIED APP.....	37
TABLE 9. HTTP DELETE RESPONSE INDICATING THE RESULT OF THE DELETE REQUEST.....	37
TABLE 10. HTTP REQUEST TO CREATE A GROUP IN MASS.....	38
TABLE 11. SUCCESSFUL GROUP CREATION RESPONSE IN MASS.....	38
TABLE 12. HTTP REQUEST CREATING A PLAYER USER ACCOUNT.....	39
TABLE 13. HTTP RESPONSE FROM A SUCCESSFUL USER CREATION REQUEST.....	39
TABLE 14. HTTP REQUEST TO FETCH A USER ACCOUNT FROM MASS.....	39
TABLE 15. SUCCESSFUL RESPONSE CONTAINING A USER DOCUMENT.....	40
TABLE 16. AUTHENTICATION REQUEST SENT TO MASS.....	40
TABLE 17. AUTHENTICATION RESPONSE FROM MASS.....	41
TABLE 18. REGISTER APP TO USER ACCOUNT HTTP REQUEST.....	41
TABLE 19. SUCCESSFUL RESPONSE WHEN REGISTERING AN APP TO A USER.....	41
TABLE 20. HTTP REQUEST TO GET A LIST OF REGISTERED APPS FOR A USER.....	42
TABLE 21. HTTP REQUEST SENT BY TAFFY TOWN TO CREATE A STATISTIC.....	43
TABLE 22. HTTP REQUEST FROM MASS WEB APPLICATION TO GET ALL STATISTICS FOR TAFFY TOWN.....	44
TABLE 23. HTTP REQUEST FROM MASS WEB APPLICATION GETTING ALL STATISTICS FOR A SPECIFIC USER OF TAFFY TOWN.....	44

LIST OF ACRONYMS

2D	Two-dimensional
API	Application Programming Interface
CORBA	Common Object Request Broker Architecture
CSA	Client-Server Architecture
CSS	Cascading Style Sheets
DCOM	Distributed Component Object Model
DOM	Document Object Model
EJB	Enterprise JavaBean
ESB	Enterprise Service Bus
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IBM	International Business Machine
iOS	Apple's Mobile Operating System
JMS	Java Message Service
JSF	JavaServer Faces
JSON	JavaScript Object Notation
JSP	JavaServer Pages
MASS	Mobile Application Statistics Service
OAuth	Open standard to Authorization
P2P	Peer-to-Peer
RDBMS	Relational Database Management System
REST	REpresentational State Transfer
RPC	Remote Procedure Call
RMI	Remote Method Invocation
SDK	Software Development Kit
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SVG	Scalable Vector Graphics
TTF	Taffy Task Force
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
WAS	WebSphere Application Server
XML	eXtensible Markup Language

CHAPTER 1: Introduction

“The worst thing a kid can say about homework is that it is too hard. The worst thing a kid can say about a game is it’s too easy.”

-Henry Jenkins

In the past few years, we have seen a huge technical migration from stationary computers to ubiquitous mobile devices. More students have access to smartphones and tablets. In 2012, 39% of students’ grades 6 through 8 use smartphones for homework and 31% use tablets for homework [Khadaroo 2012]. The device ubiquity enables educators to craft new tools that utilize this technology to deliver a better educational experience. One of those tools is the employment of digital games in various subjects. The number of published articles in seven technology-based learning journals from 2006 to 2010 has increased four times greater than the previous five years [Hwang and Wu 2012].

Apple states that over 80,000 educational apps exist on the App Store today [Apple 2014a]. It has been acknowledged that utilizing educational apps can help students learn complex subjects like fractions in a more engaging and fun manner. As we explore later, several of these apps offer some type of assessment, showing the player how they are performing various tasks.

Another emerging technological interest is *big data*, a massive collection of data which traditional data applications cannot effectively handle. The goal of this field of research is to discover methods to analyze these large sets of information to produce some type of *business intelligence*. The first conference for IEEE Big Data was held in 2013 with 259 paper submissions [Drexel University 2014]. IEEE suggests big data is transforming science, business, and healthcare.

Unfortunately, the union of big data solutions and education has had a rocky road. In 2012, a group of educators created inBoom, an open-source computer system that stored student data in a secure, common format that enabled schools to control what was collected and who had access to it. Less than two years later, inBloom has been abandoned due to student privacy concerns [C 2014].

With this evident transition into large bodies of students playing and interacting with digital apps, how can we analyze large datasets of gameplay data with big data analytics?

This thesis investigates the ability to capture *what* players are doing during gameplay and not just the assessment results. This gameplay data can then be used to produce visualizations using big data techniques. The result is not only assessment information, but also building a better understanding of *what* students do to get those results.

1.1 Related Work

Today many educational games exist that offer some evaluation on how well the player is performing. Most of the time, this calculation is the percentage of incorrect answers. This section explores existing apps that offer some type of assessment feedback to understand how this information is captured and used.

1.1.1 Existing Digital Educational Games

1.1.1.1 Early Jamestown

Figure 1 shows Early Jamestown, a game enabling an interactive experience exploring the early days of Jamestown. The game features video clips, interactive timelines, interactive keywords, and pronunciations. Additionally, it features embedded assessment in the form of quizzes to gauge the understanding of the material [Apple 2010].

The assessment data is not available outside the context of the game. This individualistic view enables players to understand where they are lacking in understanding of Jamestown history. If the data could be extracted for a large number of players, then we could potentially correlate problematic areas with gameplay data.

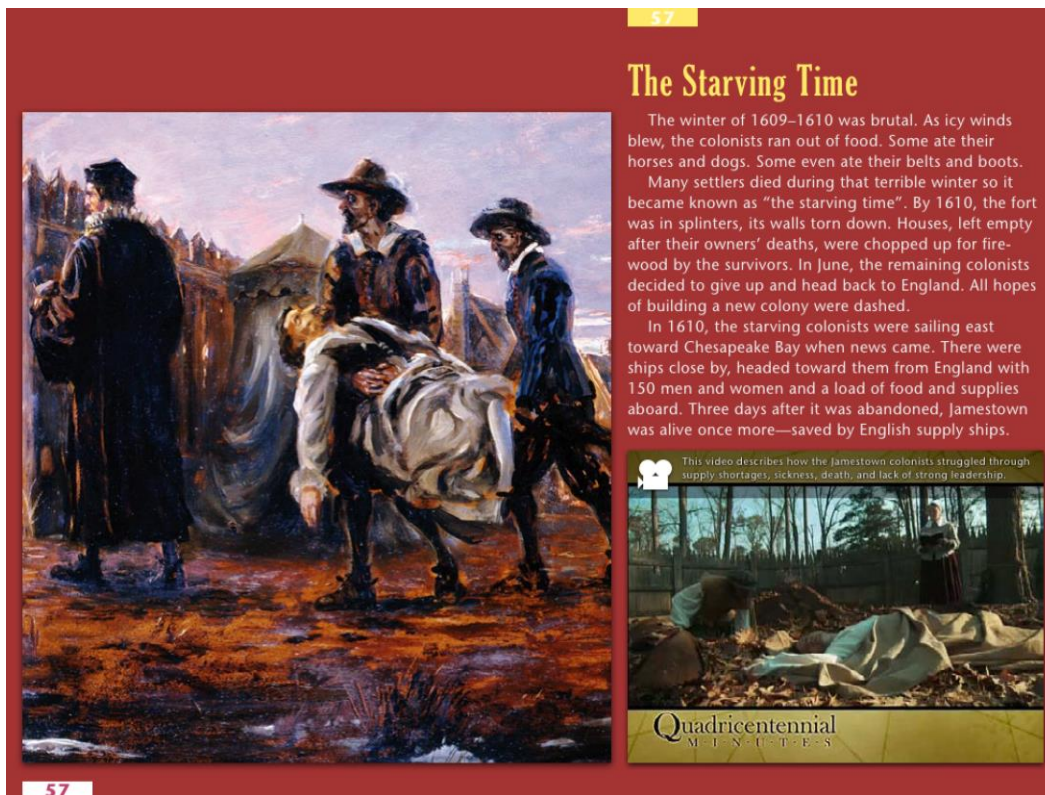


Figure 1. A screenshot of Early Jamestown game [Apple 2010]. <https://itunes.apple.com/us/app/early-jamestown/id395229194?mt=8> Used under fair use, 2014.

1.1.1.2 MathBoard

Palasoftware created MathBoard, shown in Figure 2, to be a highly configurable math app targeting kindergarten through elementary school students [Palasoftware 2014]. In order to challenge players, MathBoard provides multiple answer styles. Notable features include:

- The ability to omit negative numbers.
- Random equation generation for addition, multiplication, subtraction, cubes, and square roots.

- Intelligent problem and wrong answer generation prevents guessing.
- A problem solver that walks through each solution.
- A quick reference guide.
- Multiple profiles, allowing players to save, review, and share results from quizzes.

MathBoard takes assessment data exploration a bit further than the Early Jamestown game by allowing players to review and share quiz results. This enables manual analysis on performance data from more than a single player, but the scale could only be limited to a low number of students. In addition, this obtrusive way of requiring a large number of players to send quiz results for analysis is not as trustworthy. The results can be manipulated before sent for analysis.

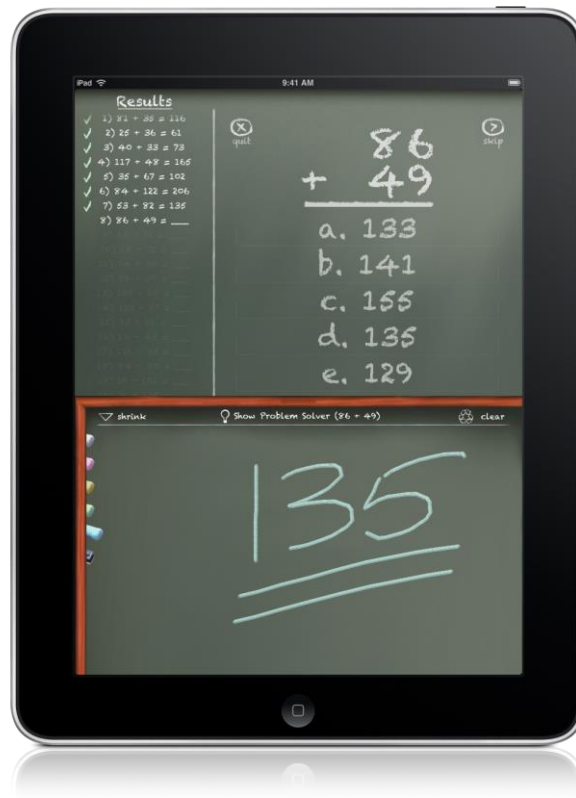


Figure 2. MathBoard by Palasoftware [Palasoftware 2014].
<http://www.palasoftware.com/MathBoard.html> Used under fair use, 2014.

1.1.2 Researching Assessment in Games

We explore several research articles in order to gain insights on how some researchers assess effectiveness of educational games. This section provides an overview of some of our findings.

1.1.2.1 Crystal Island

[Meluso et al. 2012] explore the effect of collaborative and single game player conditions on learning science by playing a game called Crystal Island. Funded by the National Science Foundation, Crystal Island is an online game in which the player interacts with characters on an island in order to learn about various landforms by using narrative-centered learning. A screenshot of Crystal Island is shown in Figure 3.

All data collection is completed *after* gameplay through the utilization of pre- and post-assessments. No data is collected during gameplay around the actions the student performs. Having this data may be useful to shed light on differences in gameplay actions between single versus collaborative play. By expanding the population, the online features can be utilized to capture a very large sample size without having manual assessments.



Figure 3. Image of Crystal Island [North Carolina State University 2014].
<http://www.intellimedia.ncsu.edu/crystal-island-outbreak> Used under fair use, 2014.

1.1.2.2 Cool-it

[Pfothenauer et al. 2009] created an online game called “Cool-it” with the objective to teach engineering concepts. This instructional game teaches the principals of cryogenic design by transforming players into cryogenic consultants. Each player is challenged to design a solution in several categories of cryogenic engineering including space, medicine, communications, electric power, and defense. The design is assessed based on a quantitative comparison of the player’s design against an optimal solution. The game is accessed through a web browser which is connected to a back-end server that provides a Simple Object Access Protocol (SOAP) wrapper to a Matlab library for resource-intensive calculations. Additionally, clicks and keystrokes are recorded during gameplay and stored in a database. Cool-it is designed to capture gameplay information and assessment data. However, the problem with this approach is that the captured data is manually sanitized and graphs are manually built after all targeted gameplay is performed. This reduces the scalability of this system as the more players of Cool-it, the longer this manual analysis process will take. Lastly, players are not provided any mechanism to view their aggregated gameplay progress after analysis.

1.2 Statement of the Problem

The transition to mobile-based digital educational games has increased over the past few years, however, unobtrusively gathering assessment and gameplay data for real-time data analysis is non-existent. Research suggests gameplay data is collected by either large log dumps after gameplay has occurred or the evaluation of various questionnaires and tests.

By utilizing multi-tiered client-server architecture, we can develop a software solution for collecting live gameplay data as well as assessment data for big data analysis and visualization. This approach allows researchers to collect gameplay data in a non-invasive manner with assessment results while thousands of students are actively playing the game.

Educators and researchers can use the proposed software-based solution to discover various correlations between gameplay and assessment results, which can facilitate better lesson plans and improved game-based learning.

1.3 Statement of Objectives

The objectives of the research described herein consist of the following:

1. Develop an educational game on an iPad mobile device, which unobtrusively collects and transmits gameplay data over the Internet to a server computer for analysis.
2. Develop a software application running on a server computer, which receives the data collected during the iPad gameplay over the Internet.
3. Engineer the software application executing on a server computer to generate effective data visualizations from the data collected during gameplay by individual players for teachers and researchers to use.

1.4 Overview of Thesis

This thesis is organized as follows: Chapter 2 provides an overview of network-centric software architectures and describes the architecture chosen for our solution. Chapter 3 describes the client software, an iPad educational game called Taffy Town. Chapter 4 presents the server software, a Java EE-based web application with its web services. Chapter 5 presents a self-evaluation of the client and server software applications. Lastly, Chapter 6 states conclusions, contributions, and future work.

CHAPTER 2: Software Architecture

“A design is an instance of an architecture like an object is an instance of a class.”

-Anonymous

Over the years, technical solutions have become increasingly complicated. Instead of single-client software solutions, the need for interconnected, distributed systems has dramatically increased the complexity of software. This added complexity facilitates the need for a standardized definition of a baseline application that serves the role as a template for all other solutions. This template is known as the *software architecture* [Erl 2005]. Software architecture operates at a generic high-level, breaking a system in components and focusing on how those components interact. This template can be physical and logic abstractions or more detailed diagrams. When these templates can be utilized to satisfy the same occurring problem in software engineering, an *architectural pattern* or style is created. Many architectural patterns exist, including: client-server, event-driven, peer-to-peer (P2P), representational state transfer (REST), and service-oriented (SOA) [Pressman 2010].

This chapter explores existing architectural patterns in order to determine the best software architecture to satisfy the objectives of this thesis.

2.1 Architectural Patterns

2.1.1 Client-Server Architecture

Client-server architecture (CSA) is a distributed architecture that separates various concerns of the application into a client-host and server-host. The client application sends various requests to the server over a computer network. These requests can be data lookups or computationally-intensive processes the server is able to perform [Pressman 2010]. After the server receives the request and executes the requested task, the results are sent back to the client. Decomposition of the various concerns of the application can reside on either the client or server. Depending on the degree of application segregation and the resulting dissemination, several classifications of client-server architecture exist. The most common are two-tier CSA and multi-tiered CSA in which a tier represents one of the separated layers of the application [Erl 2005].

2.1.2 Two-Tier

In two-tiered client-server architecture, the application is divided into two tiers; the client tier and data tier. The client tier, or application, contains all logic for the application including graphical user interfaces (GUI), processing or business logic, and server connection management. The server's responsibility is to manage all the data the client needs to execute. Typically, a relational database management system (RDBMS) is hosted on the server that enables the client to request the execution of structured query language (SQL) statements [Fowler 2003]. Figure 4 depicts a two-tier CSA.

Two-tiered CSAs have several disadvantages. As the number of clients increase, the server's performance slows down as each client request must be performed in a transactional manner. Transactions ensure multiple clients are using the most current data by locking information in the database. This prevents dirty reads, lost updates, and non-repeatable reads [Bolton et al. 2012]. Additionally, changes to the client application will require administrators to update each client with the new application. This increases the cost of software maintenance and decreases the maintainability of the solution.

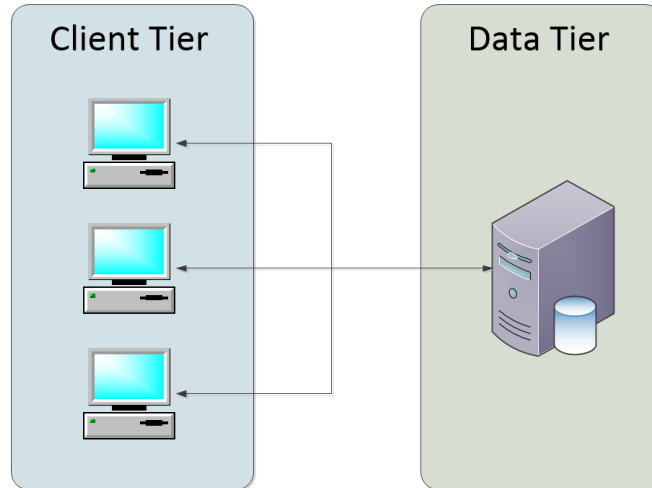


Figure 4. Two-tier client-server architecture.

2.1.3 Multi-Tier

A multi-tiered CSA is achieved by further decomposing the software into multiple physically separated layers. One of the most common types of multi-tiered architecture is when the application is segregated into three tiers: the presentation tier, the application tier, and the data tier as shown in Figure 5 [Pressman 2010].

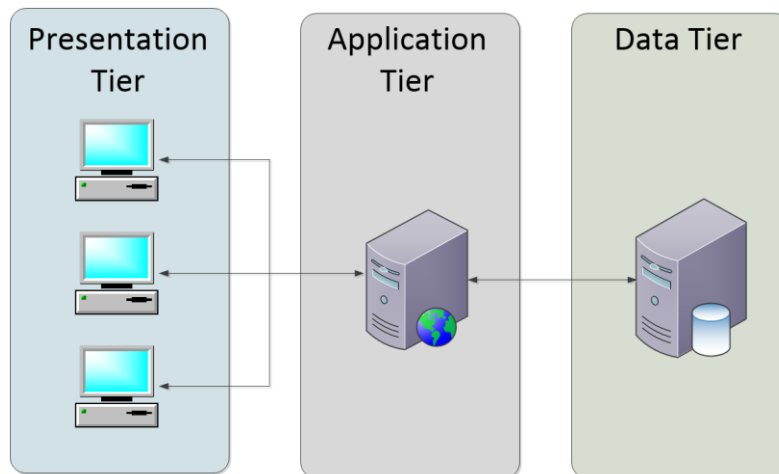


Figure 5. A multi-tiered architecture.

The presentation tier is responsible for display and collection of user data. This presentation can be in the form of a simple text-based terminal or complicated GUI. For example, an Internet browser is concerned with loading a web page and displaying it to the user. The user is able to input information and send it to the server for processing. The browser is not responsible for processing the data, but only the display and collection of that data.

The application tier or business logic tier sits between the presentation and data tiers. This layer is responsible for taking requests for the client's presentation tier, performing some type of business logic or process, and returning the results back to the client for presentation. The data layer is utilized by this layer in order to load or store any data needed for the invoked computations.

The data tier is responsible for the storage and retrieval of information. Data can be stored on the file system or a RDBMS. Data is retrieved by the application tier for various processing and returned for storage. Only the application tier should be allowed to request information from this tier.

Further separating applications into physically separated layers solves several disadvantages of a two-tiered architecture. The application layer acts as a mediator between the presentation layer and data tier, increasing the number of clients without affecting the performance. This is achieved by allowing the application tier to maintain pools of connections to the database instead a single connection per client. This increase of scalability provides better performance for a large number of clients. Another benefit is the increased ability to maintain the application logic. Instead of having to update every client, only the application tier needs to be upgraded.

These benefits come with several disadvantages. As we increase the number of separated tiers, the complexity of the software solution increases. This complexity adds to the initial setup of the application. Additionally, more software is needed for the application tiered in order to host the services for the clients [Fowler 2003].

2.1.4 Service-Oriented Architecture

Service-oriented architecture (SOA) has emerged to be the best architecture to use to solve the problem of interoperability among systems of systems. SOA specifies the system using course-grained, loosely coupled, autonomous modules called *services*. A service exposes behavior through a *contract*. A contract defines the discoverable *endpoints* and *messages* a behavior needs for execution. *Service consumers* send and receive messages to the service, invoking behavior defined in the contract. A *policy* is concerned with security specifications, auditing, and other cross-cutting concerns that govern the service. Any service consumer interacting with a service must adhere to the policy [Rotem-Gal-Oz 2012]. Figure 6 depicts the relationship of the components of SOA.

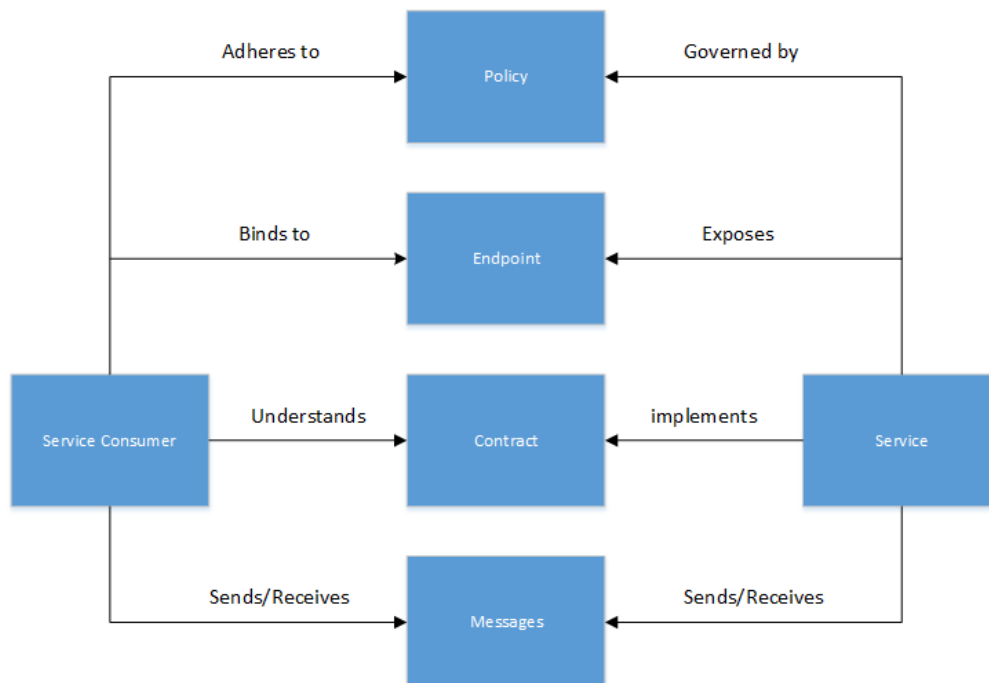


Figure 6. Service-Oriented Architecture primary components.

2.1.5 Representational State Transfer

The representational state transfer (REST) architectural style imposes several design rules or constraints to ensure distinct architectural properties are achieved. These constraints are applied to the components, interactions, and data required by the software solution. The constraints for REST are:

1. *Client-Server Model* – requires support for independent evolution of client and server logic. Clients send requests to the server in order to invoke a task. The task is either executed or rejected before a response is sent to the client.
2. *Stateless* – no state should be kept between communication between the client and server.
3. *Cacheable* – the server is able to mark messages as cacheable or non-cacheable so clients can reuse responses.
4. *Layered System* – no layer can have knowledge of any layer past the directly connected layer.
5. *Uniform Interface* - all clients and services adhere to a uniform interface.

When applying REST constraints to web service application programming interface (API), the HTTP based web services are considered *RESTful*. By utilizing RESTful web services, several properties of the software are improved. These properties include: performance, scalability, simplicity, modifiability, visibility, portability, and reliability [Erl et al. 2014].

A RESTful web service consists of a service consumer and service hosted on a web application server. Clients utilize the uniform interface of HTTP to send requests that interact with a resource. The HTTP methods are described in Table 1.

Table 1. RESTful web service HTTP interface methods.

HTTP Method	Description
GET	Returns the resource location at the uniform resource identifier (URI).
POST	Creates a new resource. A URI is returned that represents the newly created resource.
PUT	Modify an existing resource.
DELETE	Deletes a resource at the provided URI.

The next section describes the architecture proposed for the collection of gameplay data.

2.2 Mobile Application Statistics Service

In order to satisfy the objectives of this thesis, we need a server application that will enable a large number of mobile clients to send gameplay data for analysis. Additionally, this server application needs to be able to support the ability for players to log in and view this performance data in real-time. These functional requirements are the basis of our server application, named Mobile Application Statistics Service (MASS).

MASS will be responsible for the following key requirements:

1. Enable games to send gameplay data in a light-weight manner.
2. Analyze and generate visualizations of the gameplay data.
3. Provide a web site for players to view generated visualizations of their gameplay in real-time.

By identifying these functional requirements of MASS, we evaluated various architectural styles against the following architectural requirements:

1. A light-weight asynchronous communication from a mobile device to a server application over the Internet.
2. A robust infrastructure that can handle many clients sending large datasets simultaneously.
3. Since we do not know what data client games send, we need a means to store some type of *unknown data structure*.

Given these constraints, we chose to utilize a mixture of multi-tiered client-server architecture and a collection of RESTful web services as shown in Figure 7. By using RESTful web services for the clients, the light-weight asynchronous communication requirement is satisfied. Light-weight communication allows statistics to be created in MASS using simple HTTP commands, reducing bandwidth and hardware resource requirements of the client. Additionally, multi-tiered CSA will allow MASS to scale at each tier independently. Scaling each tier will enable a large number of players to send gameplay data to MASS. Lastly, we incorporated a NoSQL database to store gameplay information to satisfy the “unknown data structure” requirement. NoSQL databases enable the storage of schema-less information to be retrieved and stored. The game is responsible for defining the structure of the gameplay data transmitted to MASS, removing the need for modifying the server when the gameplay data’s structure changes.

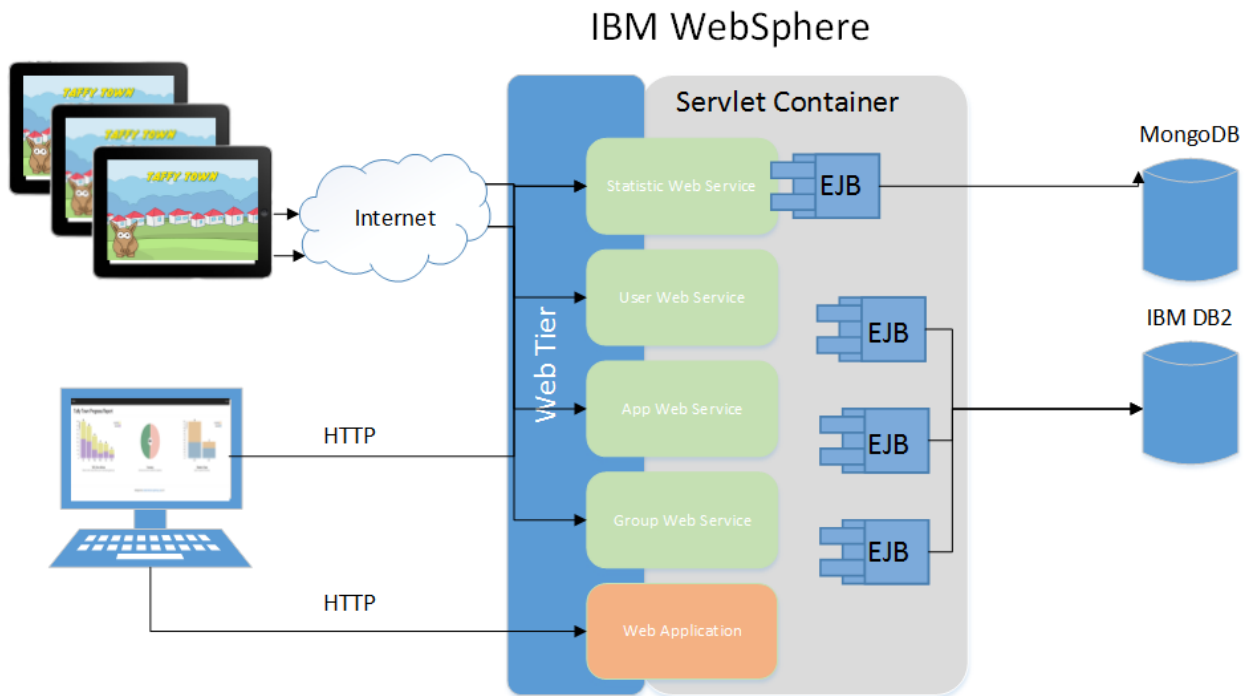


Figure 7. MASS Architecture.

The various components of this architecture are described in the next several sections.

2.2.1 Java Enterprise Edition

The Java platform, Enterprise Edition (Java EE) is a distributed, multi-tiered application model designed to handle large enterprise applications. Java EE is a set of specifications implemented by different containers as shown in Figure 8. Each container provides services to hosted components like transaction handling, dependency injection, and life cycle management. The Java EE runtime environment is comprised of four component types:

1. Applet – Swing-enabled applications that are executed in a browser.
2. Applications – programs executed on the client such as batch programs or GUI applications.
3. Web applications – applications that execute in a web container comprised of servlets, web event listeners, JavaServer Pages (JSP), and JavaServer Faces (JSF). Servlet support SOAP and RESTful web services as of Java EE 6.
4. Enterprise Applications – executed in an Enterprise JavaBean (EJB) container that utilize various supplemental APIs like the transactional API, timer services, Java Messaging Services (JMS), remote method invocation (RMI).

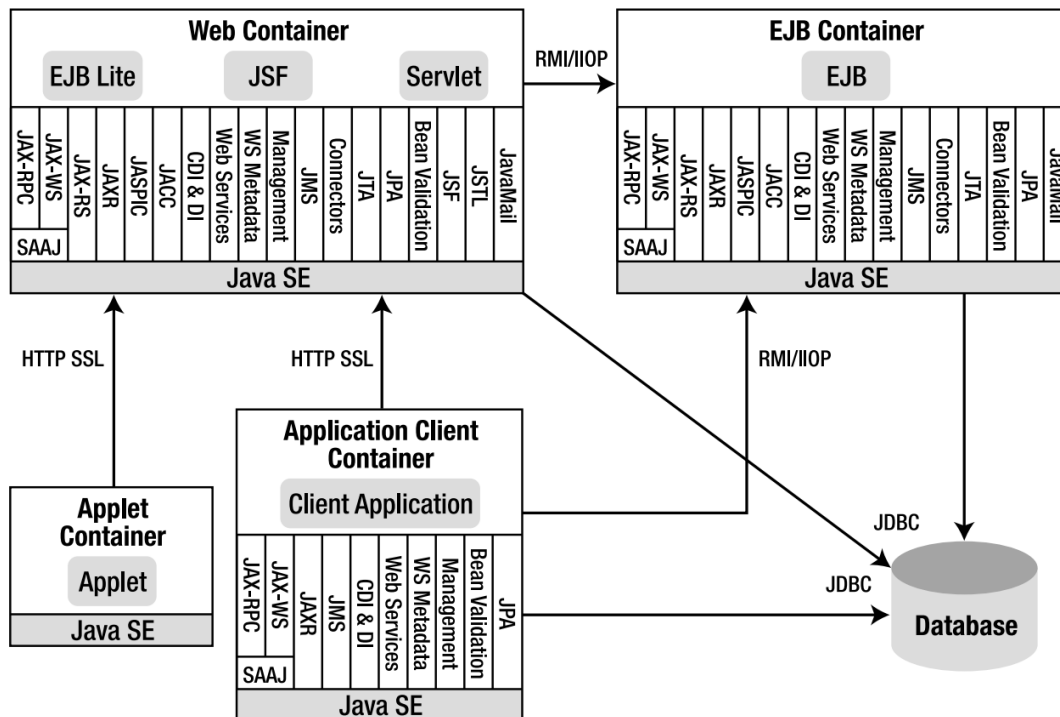


Figure 8. Java EE container architecture [Goncalves 2010].

We chose Java EE to take advantage of the services offered in each of the Java EE containers. Java EE provides an application tier software package that enables us to build a web site and to create RESTful web services required by MASS.

2.2.2 IBM WebSphere Application Server

IBM’s WebSphere Application Server (WAS) is one of the leading enterprise-level Java EE application servers on the market. WAS supports all Java EE container specifications, allowing all functional requirements of MASS to be implemented. Figure 9 lists all Java specifications supported by WAS.

Technology	Specification reference	Full profile	Liberty profile	Liberty Core
Java Platform, Enterprise Edition 6 (Java EE 6)	JSR 316	X		
Java Platform, Enterprise Edition 6 Web Profile	JSR 316	X	X (8.5.5)	X
Web services technologies				
Java API for RESTful Web Services (JAX-RS) 1.1	JSR 311	X	X	X
Java APIs for WSDL 1.2	JSR 110	X	X	X
Implementing Enterprise Web Services 1.3	JSR 109	X	X (8.5.5)	
Java API for XML-Based Web Services (JAX-WS) 2.2	JSR 224	X	X (8.5.5)	
Java Architecture for XML Binding (JAXB) 2.2	JSR 222	X	X (8.5.5)	
Web Services Metadata for the Java Platform	JSR 181	X	X (8.5.5)	
Java API for XML-based RPC (JAX-RPC) 1.1	JSR 101	X		
Java APIs for XML Messaging 1.3	JSR 67	X		
Java API for XML Registries (JAXR) 1.0	JSR 93	X		
SOAP with Attachments API for Java (SAAJ) 1.3	JSR 67	X	X (8.5.5)	
Web application technologies				
Java Servlet 3.0	JSR 315	X	X	X
JavaServer Faces 2.0	JSR 314	X	X	X
JavaServer Pages 2.2/Expression Language 2.2	JSR 245	X	X	X
Express Language (EL) 2.2	JSR 245	X	X	X
Managed Beans 1.0	JSR 316	X	X	X
Standard Tag Library for JavaServer Pages (JSTL) 1.2	JSR 52	X	X	X
Debugging Support for Other Languages 1.0	JSR 45	X	X	X
Enterprise application technologies				
Contexts and Dependency Injection for Java (Web Beans 1.0)	JSR 299	X	X (8.5.5)	X (8.5.5)
Dependency Injection for Java 1.0	JSR 330	X	X (8.5.5)	X (8.5.5)

Figure 9. Java EE specifications supported by WAS [Albertoni et al. 2013]. Used under fair use, 2014.

2.2.3 IBM DB2

We chose IBM's DB2 as our RDBMS. DB2 offers enterprise-level performance, scale, and reliability without a lot of configuration. Additionally, WAS has built-in DB2 driver support facilitating faster integration with DB2 as opposed to MySQL [\[IBM 2014\]](#).

2.2.4 *Mongo DB*

In order to eliminate the coupling between the gameplay data sent from the structure of the data, MongoDB is utilized for the storage of all gameplay data. MongoDB is a document-oriented database as opposed to a relational database. This replaces the concept of a "row" with a document. Documents allow for complex hierarchical relationships in a single entity. Most importantly for MASS, document-oriented databases do not require a schema definition that defines the various fields and structure of the data. This allows JavaScript Object Notation (JSON) data containing gameplay information to be added dynamically, eliminating the inflexibility caused by relational models [\[Chodorow 2013\]](#).

By utilizing MongoDB, the client can send any JSON object to MASS without having to make any changes to MASS itself. This is achieved by structuring the JSON documents in the following structure.

```
{
  "Application ID" : "An ID representing the application sending data to MASS",
  "User ID" : "The ID of the user that the gameplay data belongs",
  "JSONMetadata" :
  {
    // Any game-specific data
  }
}
```

CHAPTER 3: Taffy Town Educational Game

We developed Taffy Town as the client software application for our client-server solution. This digital educational game has two primary responsibilities: (a) engaging students in game-based learning mathematics and (b) unobtrusive data collection about player interaction during gameplay. Taffy Town is designed and developed to run on the Apple iPad mobile device by using the iOS Software Development Kit (SDK) and Apple’s 2D game engine and framework called Sprite Kit [Apple 2014b].

In Taffy Town gameplay, the player is tasked to save Taffy Town from taffy-eating aardvarks known as the “Varks”. The player, as part of the Taffy Task Force (TTF), patches holes in various town buildings with vark-resistant taffy. The player observes Taffy Town and when a building is attacked, the player stops the attack tapping the building. This presents the player with a piece of taffy, hole graphic, and a fraction question. The player needs to answer the presented question by manipulating the taffy to patch the hole. The game allows manipulation of taffy by moving, stretching, cutting, gluing, deleting extra pieces, and painting taffy pieces. After the player transforms the presented taffy, s/he attempts to patch the hole. If the player gets the question correct, the hole is patched. Otherwise, the hole shakes signaling that the patch is unsuccessful. Regardless of correctness, the game transitions back to viewing Taffy Town in anticipation for another attack by the Varks. Figure 10 illustrates an overview of this flow of actions during gameplay.

As the player answers questions, information about each taffy manipulation is sent to the server application, MASS. The data collected includes various facets of information about the question and all transformations performed on the taffy. In order for this data to correctly map to the current player, users are required to create and login to their account before playing. Authentication and account creation features are offered to the user with pop-up windows to ensure their experience is contained within the game.

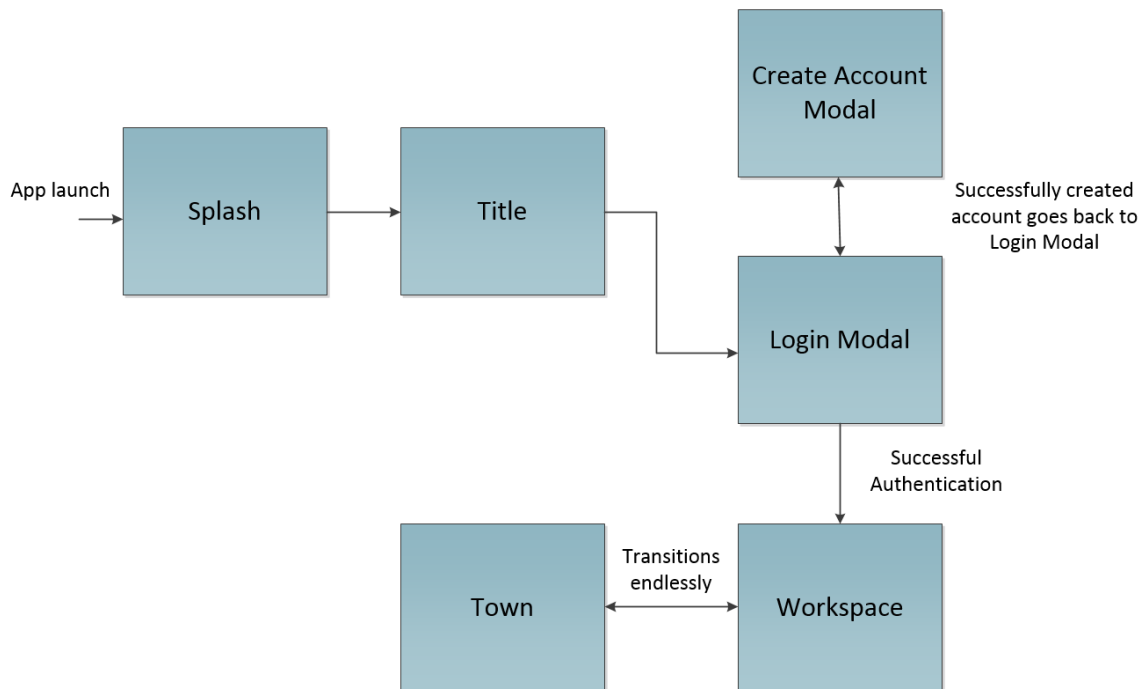


Figure 10. Taffy Town’s scene storyboard.

3.1 Title Screen

After launching Taffy Town on the iPad, the Title Screen is displayed as shown in Figure 11. The title screen has a static background with the title and a single button labeled “Play”. Pressing Play initializes and displays the authentication pop-up window, which contains an option to create an account.



Figure 11.Taffy Town title screen.

3.1.1 Player Account Creation

Before a player can login to Taffy Town, an account must be created. Pressing the “Create Account” button on the authentication pop-up window displays the account creation window to the player shown in Figure 14. The seven graphical elements are described below:

1. *First Name text field*: Responsible for collecting the first name of the player.
2. *Last Name text field*: Responsible for collecting the last name of the player.
3. *Email text field*: An email address associated with the account. This is used as part of the authentication process of Taffy Town.
4. *Password text field*: A string of letters and numbers that is used for protecting the account. This is used to log into the game.
5. *Confirm Password text field*: To ensure the password is not mistyped, the confirm password must match the password text field.

6. *Register Button*: Submits the information supplied to the server. Upon successful account creation, the game displays a pop-up window as shown in Figure 15. If an error occurs when creating the account, a pop-up window is displayed with error information shown in Figure 16.
7. *Cancel Button*: Dismisses the account creation pop-up window. The authentication pop-up window is redisplayed.

3.1.2 Player Authentication

In order to collect data about a player's actions in real-time, each player is required to create an account and then login. As shown in Figure 12, the authentication pop-up window has five graphical elements as described below:

1. *Email Address text field*: Email address the player used when creating his or her account.
2. *Password text field*: Password used for account creation to protect the player's account.
3. *Login Button*: Sends the email and password supplied by the player to the server for authentication. A successful login transitions to the Town Screen described in section 3.2.
4. *Create Account Button*: Displays the "Create Account" pop-up window described in 3.1.1.
5. *Cancel Button*: Dismisses the pop-up window.

Supplying an invalid email and password combination results in an error as depicted in Figure 13.

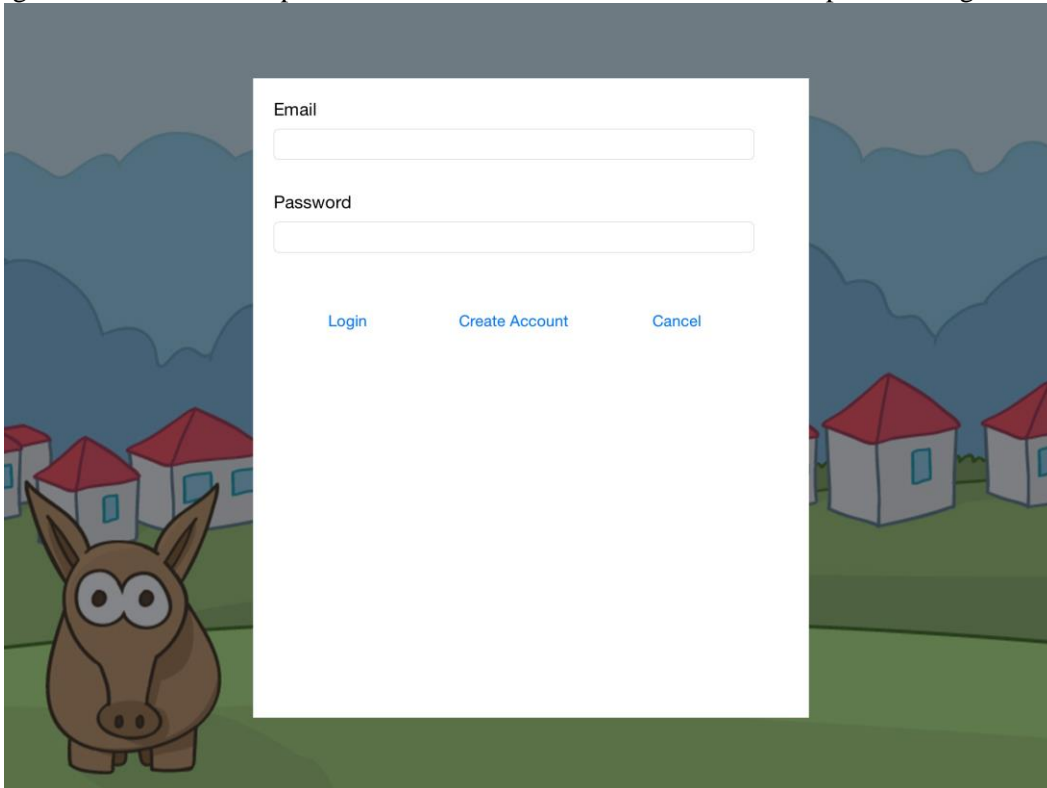


Figure 12. Authentication pop-up window.

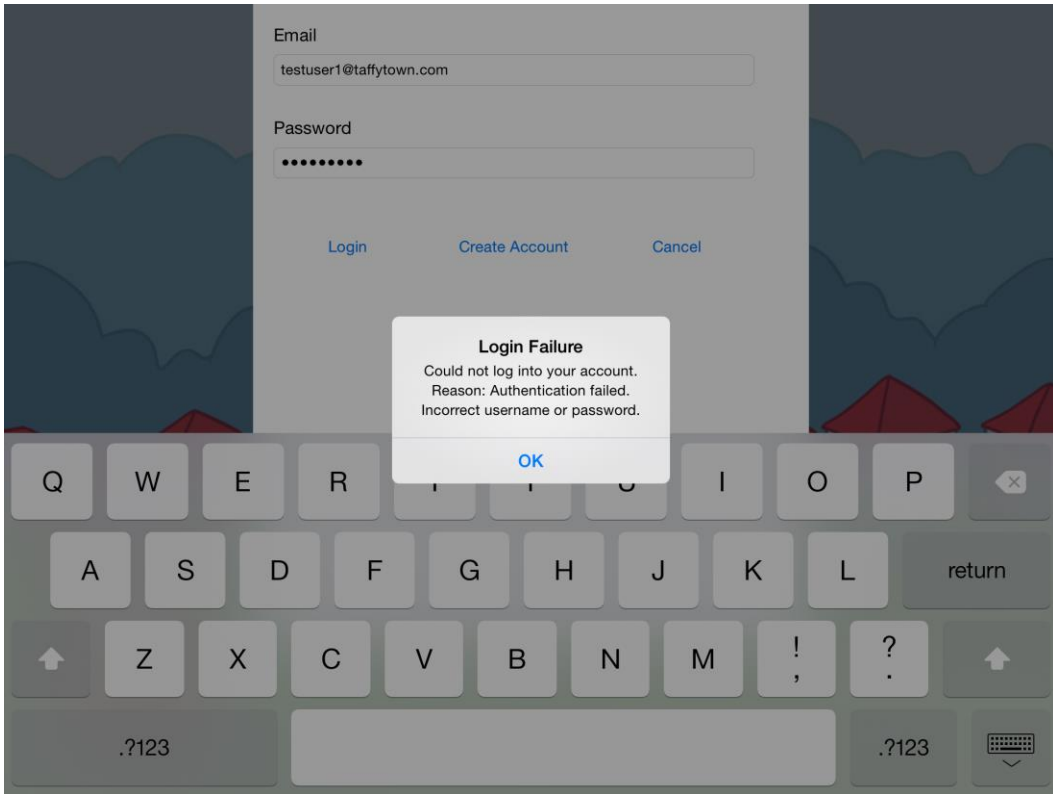
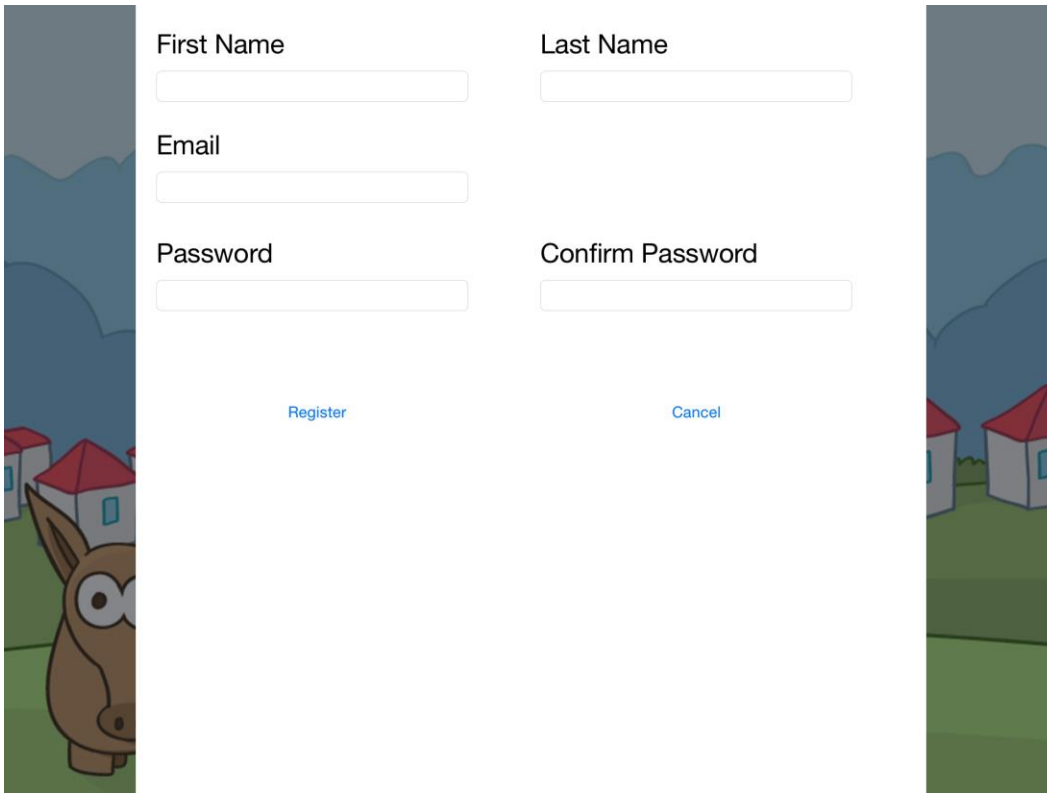


Figure 13. Login error popup.



First Name

Last Name

Email

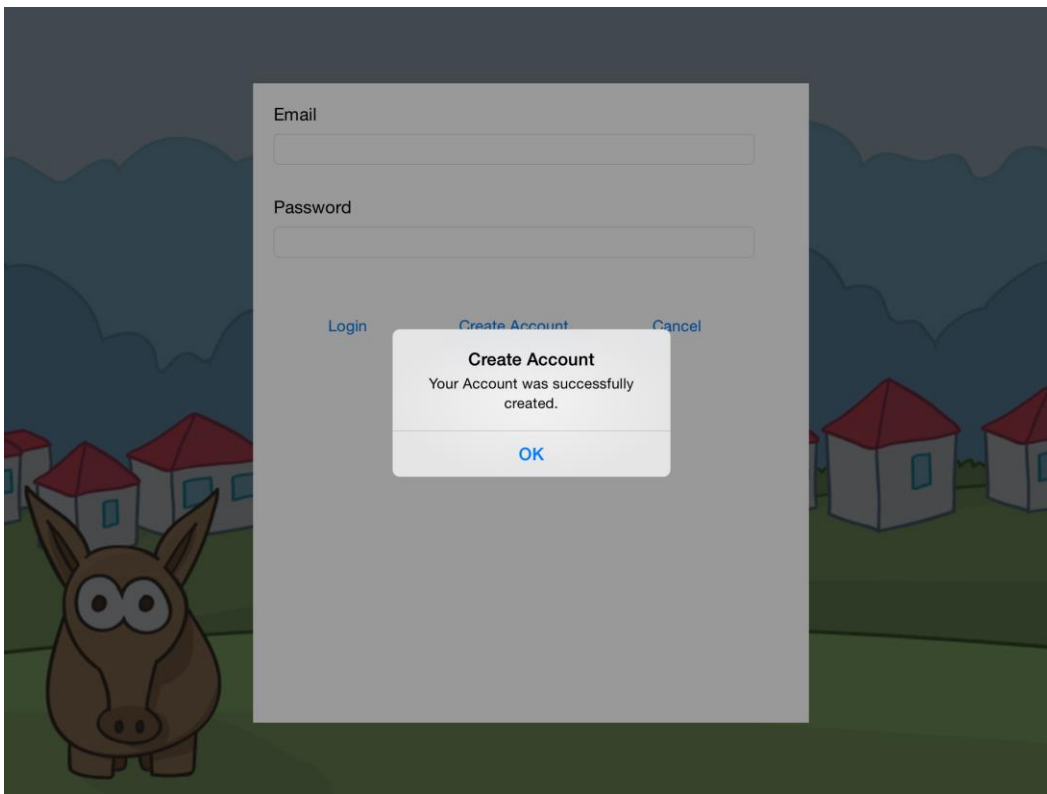
Password

Confirm Password

[Register](#) [Cancel](#)

The form is set against a background illustration of a town with houses and a donkey character in the bottom left corner.

Figure 14. Taffy Town account creation pop-up window.



Email

Password

[Login](#) [Create Account](#) [Cancel](#)

Create Account
Your Account was successfully created.

[OK](#)

The alert panel is overlaid on a dimmed version of the account creation form and background illustration.

Figure 15. Successful account creation alert panel.

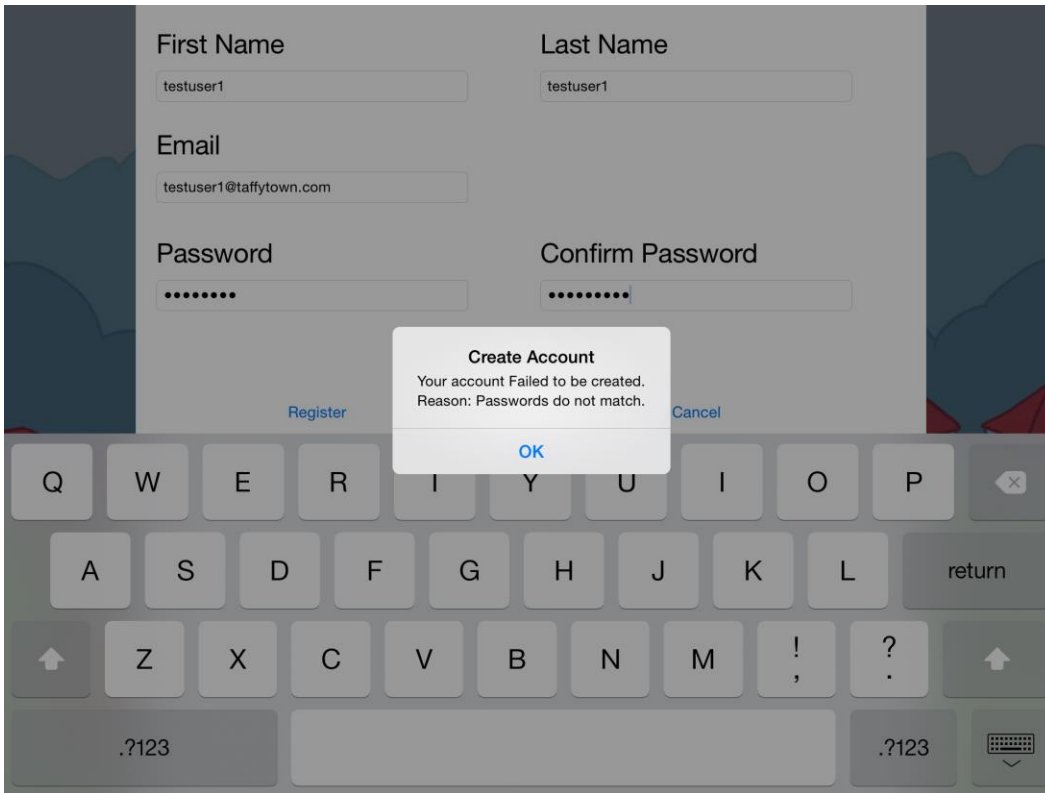


Figure 16. Account creation failure alert panel.

3.2 Town Screen

The town screen, as shown in Figure 17, is the start of Taffy Town’s gameplay in which the town is displayed to the player. The background consists of animated clouds, a few grassy hills, and the three buildings of Taffy Town; a house, library, and fire station.

Player interaction on the town screen is achieved by single tapping on one of the three buildings. In random intervals, different buildings enter an “attack” state as shown in Figure 18. In order to stop an attack, the player must tap a building that has the dust cloud image superimposed. If the player delays in tapping an attacked building, more buildings can also be attacked. There is not a penalty for neglecting to tap an attacked building; however, the game does depend on this user interaction to transition to the workspace screen described in the next section.



Figure 17. Taffy Town on a bright, sunny day.

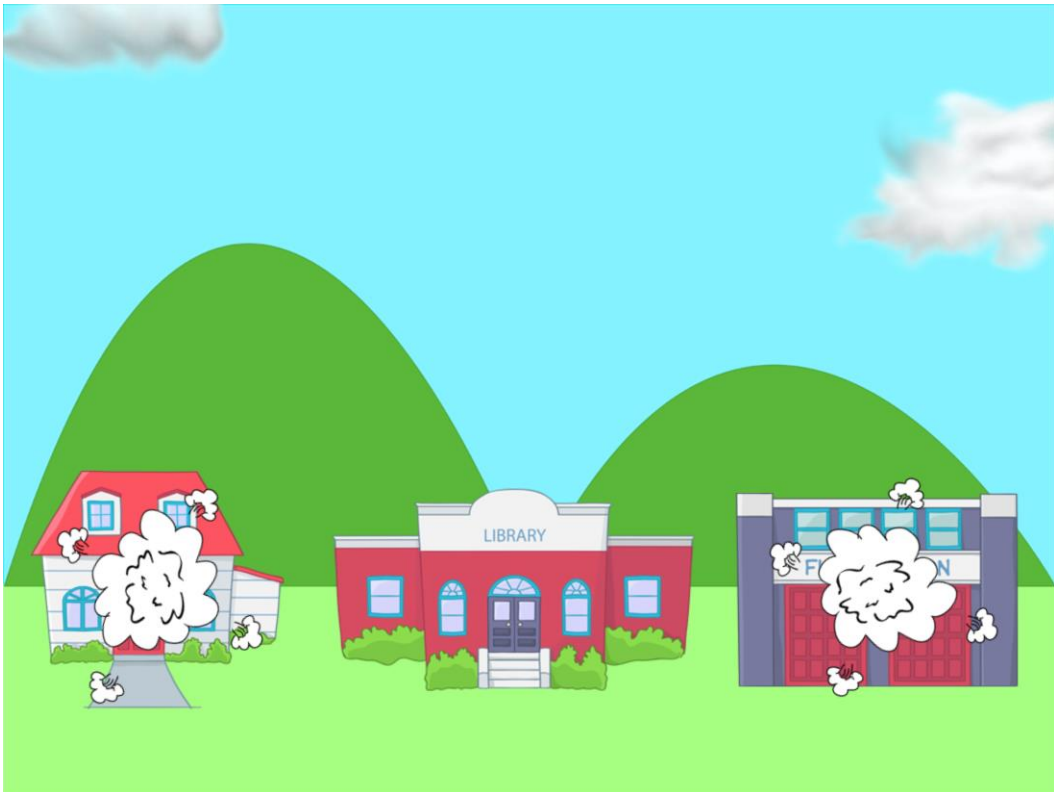


Figure 18. Taffy Town being attacked.

3.3 Workspace Screen

The workspace screen is where the core interaction with Taffy Town takes place. In this scene, the player manipulates the piece of taffy in order to answer a question by “patching” the hole. The screen is divided with an image of the damaged building on the left and a grid with a piece of taffy on the right. The core elements of the workspace scene, depicted in Figure 19, are described below:

1. *Question text display:* A randomly generated fractional question in which the player must answer by manipulating the piece of taffy.
2. *Hole:* This image represents the hole the aardvarks ate and corresponds to the fractional question displayed in the question display field. A player uses this hole to “patch” or submit the answer to the problem.
3. *Taffy:* Players can perform a specific set of gestures to manipulate the taffy.
4. *Paint brush:* Taffy pieces can be painted by dragging this image over taffy pieces.

Each time the workspace screen is displayed, a random question is generated for the player to answer. A question can be one of two types. Type 1 takes the form of “The Varks ate a hole that is n/d the size of this piece of taffy. Change the taffy to patch the hole!”. Type 2 is formatted, “This piece of taffy is n/d of the hole the Varks made. Make the taffy as a whole!”. The difference between question types is the target object in which the fraction applies. Type 1, the fraction is applied to the hole. Conversely, in Type 2 the fraction is applied to the taffy piece. Each question type can be either a proper or improper fraction.

3.3.1 Taffy Interactions

As the player manipulates the taffy, the game is collecting and saving user interaction data. When the player attempts to patch the hole, the user interactions, question-specific data, and results are sent to the server. This information includes question type, correctness, question text, and all the transformations applied to the taffy piece for the currently logged-in player. The next section describes all the supported user interactions on the workspace screen and the data sent to the server application related to the action.

3.3.1.1 Translation

Taffy pieces can be moved around on the screen by placing a single finger on the targeted taffy and dragging it around. The taffy follows the finger all around the screen. When the player stops touching the taffy, it stops in its current place. The player is allowed to move taffy pieces to any place on the screen, including over the hole. Figure 20 shows a single taffy piece that has been moved to the top of the grid.

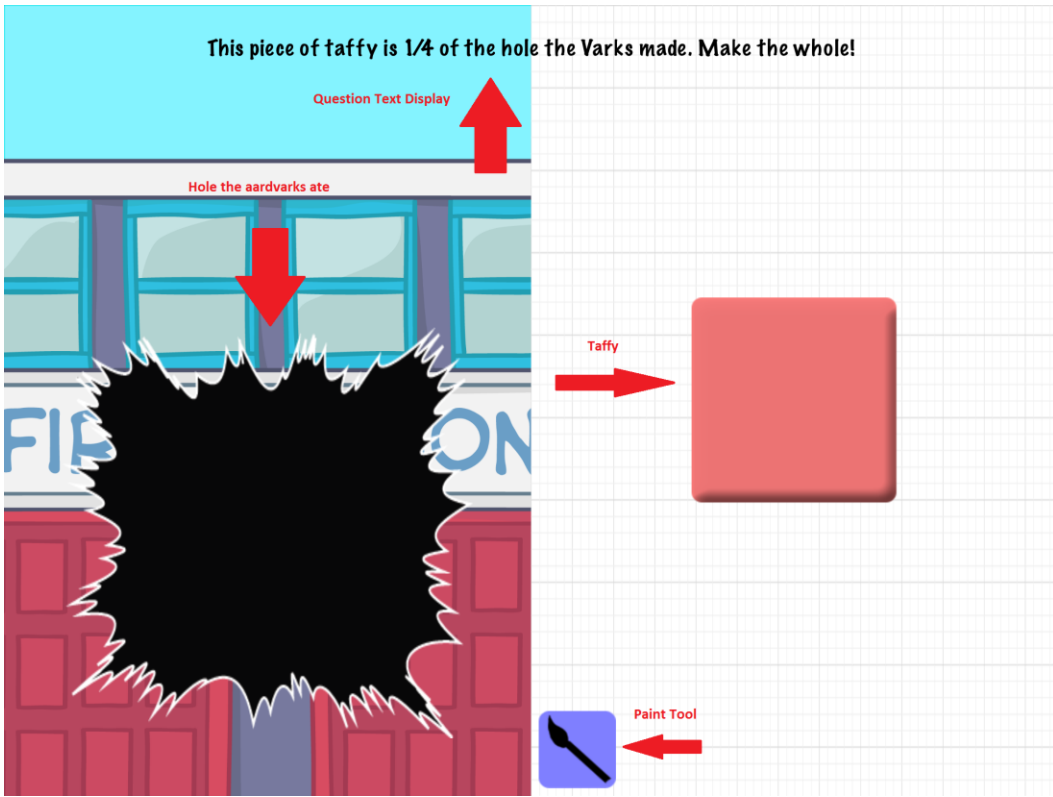


Figure 19. Workspace screen elements.

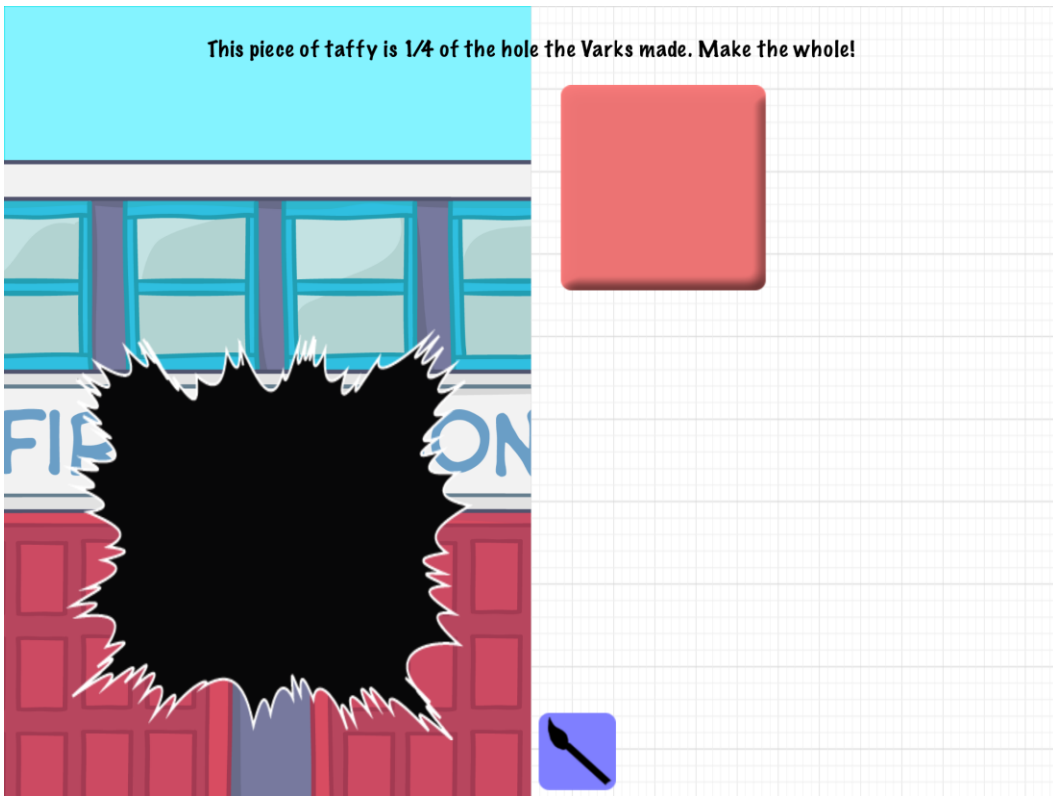


Figure 20. Translating a piece of taffy.

3.3.1.2 *Slice*

A player can slice or partition a piece of taffy by swiping, using a single finger, through the taffy piece. The player must start on the grid, swipe through the taffy, and end on the grid in a single motion. The taffy piece is divided on a successful swipe either horizontally or vertically. Diagonal slicing is not supported. Figure 21 shows a several pieces of taffy after various horizontal and vertical slices.

The slice data sent to the statistics web service is the X coordinate for vertical slices and the Y coordinate for horizontal slices.

3.3.1.3 *Stretch and Shrink*

Stretching the taffy can be accomplished by placing two fingers at opposing ends of the taffy piece and moving them apart in a stretching motion as demonstrated in Figure 22. Bringing the fingers closer together in a pinching motion shrinks the taffy accordingly. Additionally, stretching and shrinking diagonally is not supported.

Performing the stretch and shrink actions records the direction and rate in which the taffy piece is manipulated. Subsequent stretches or shrinks are recorded as independent interactions.

3.3.1.4 *Copy and Paste*

Players can copy a piece of taffy by double tapping the target taffy with a single finger then double tapping the grid with a single finger. As shown in Figure 23, you can copy and paste a single piece of taffy multiple times.

The only data sent to the server for copy/paste interactions is an interaction called “copy”. No other supplemental data is needed.

3.3.1.5 *Delete*

Extra pieces of taffy can be removed from the screen by double tapping the target piece with two fingers.

No data except for the interaction itself is sent to the server for delete taffy interactions.

3.3.1.6 *Painting*

Each taffy piece can be “painted” by placing a single finger on the paint brush icon then dragging over pieces of taffy. The taffy’s color changes to the color of the paint brush icon as shown in Figure 24.

Players can change colors by activating the color palette with a single-finger, double-tap on the paintbrush icon. Figure 25 illustrates the activated color swatch. The player selects the color by tapping on the targeted color. Each taffy piece, as demonstrated in Figure 26, can be painted independently of each other.

Painting taffy pieces is not recorded and subsequently sent to the statistics service.

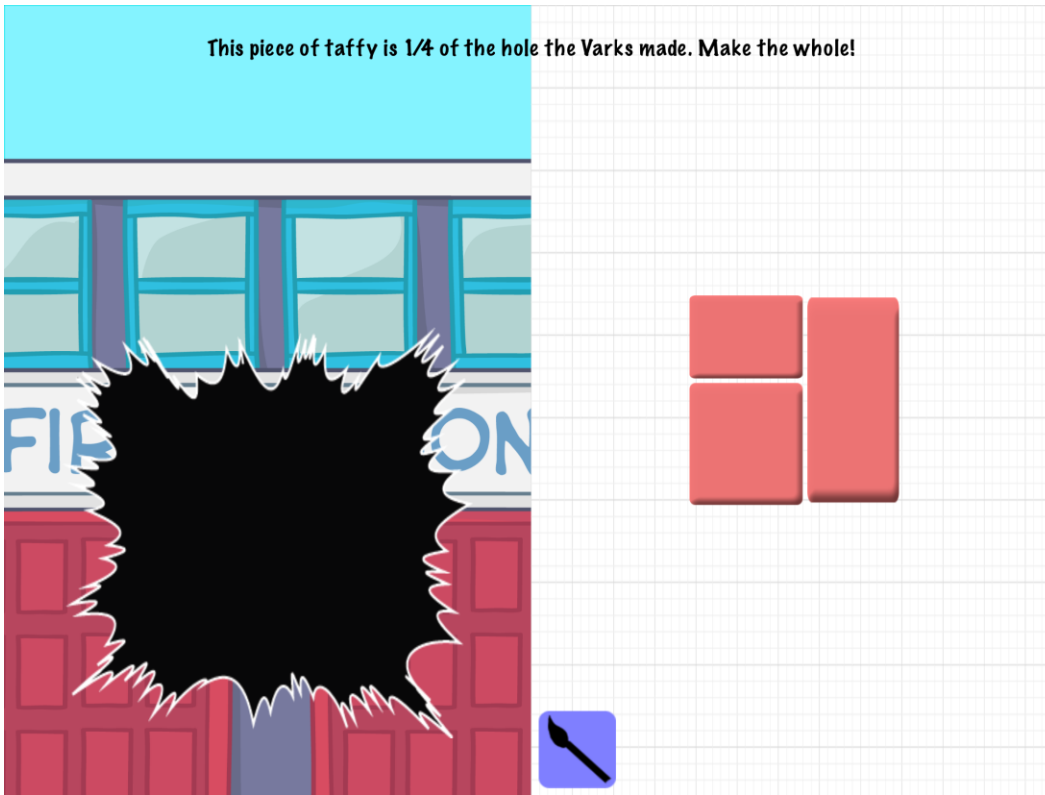


Figure 21. Performing the slice gestures on a piece of taffy.

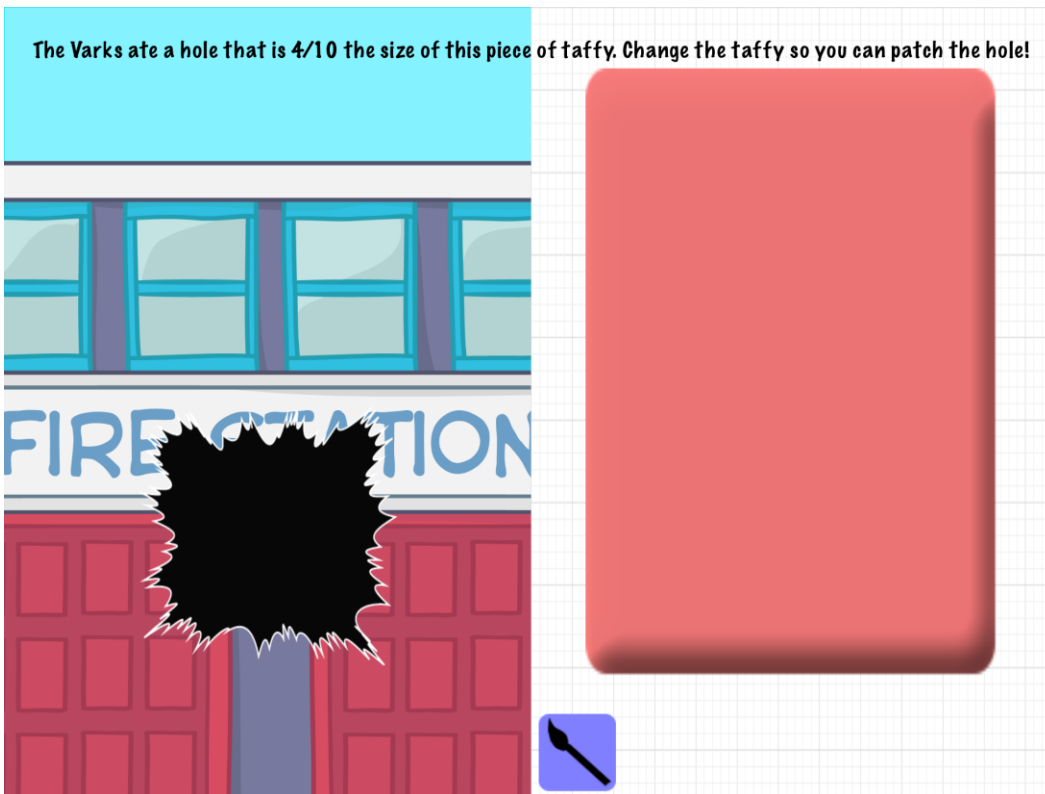


Figure 22. Vertical stretch interaction.

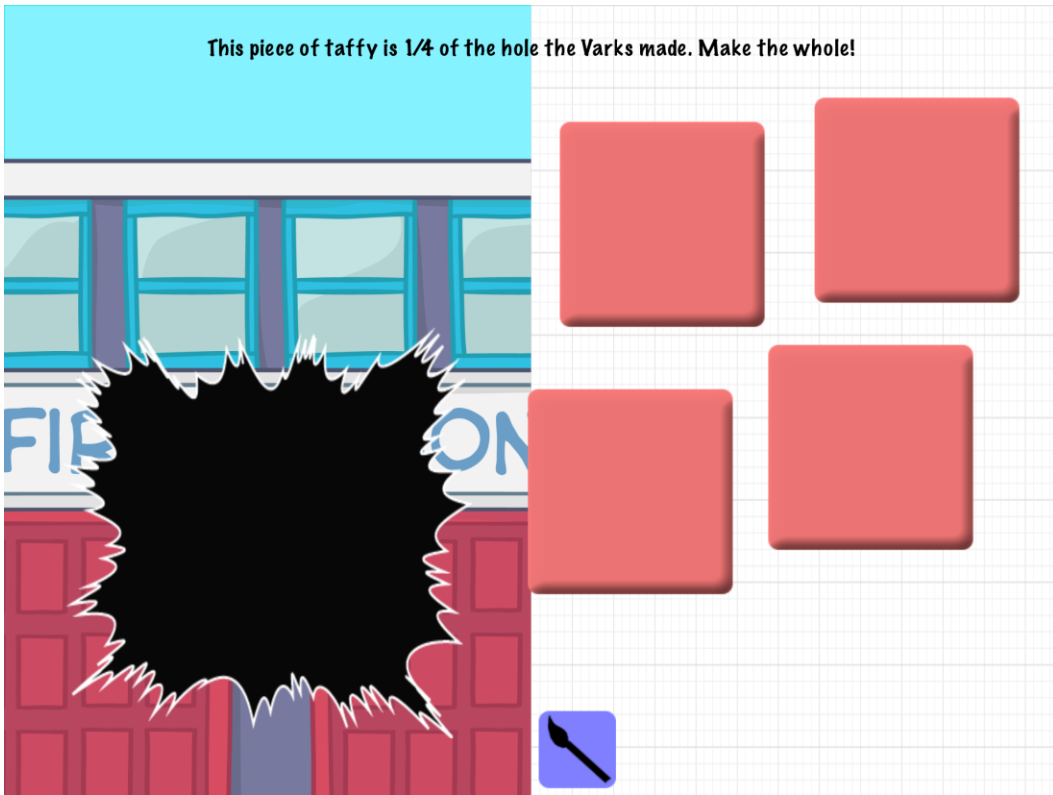


Figure 23. Copy and paste user interaction.

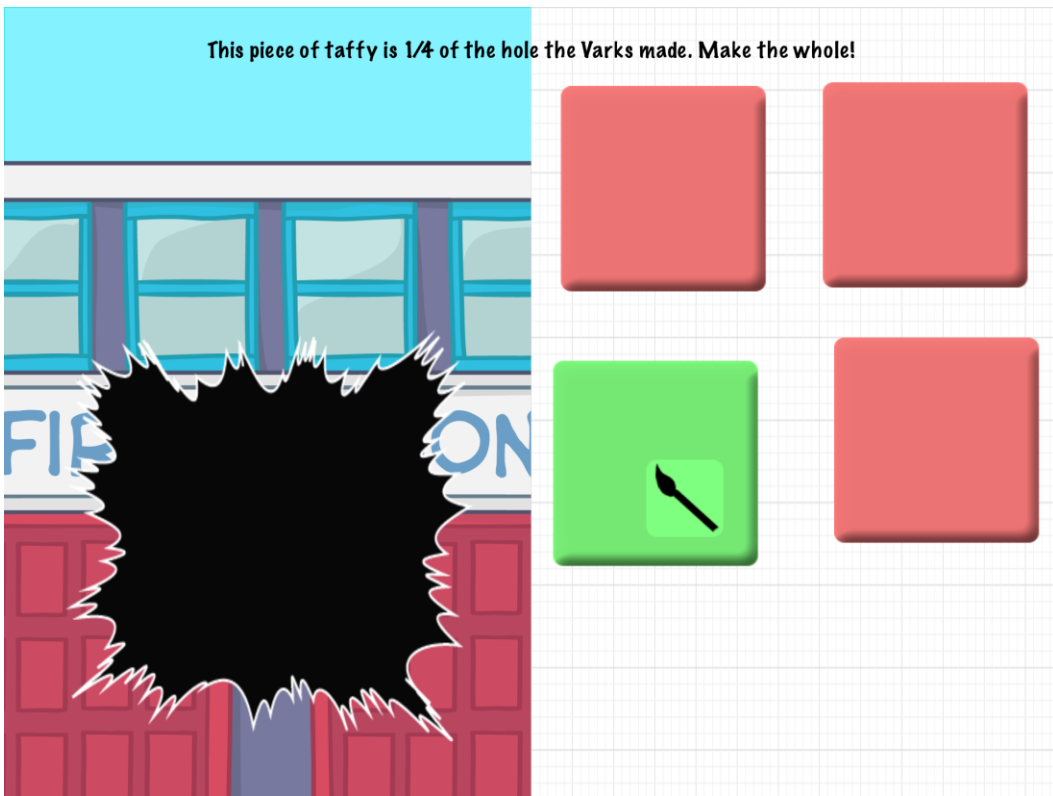


Figure 24. Painting a piece of taffy green.

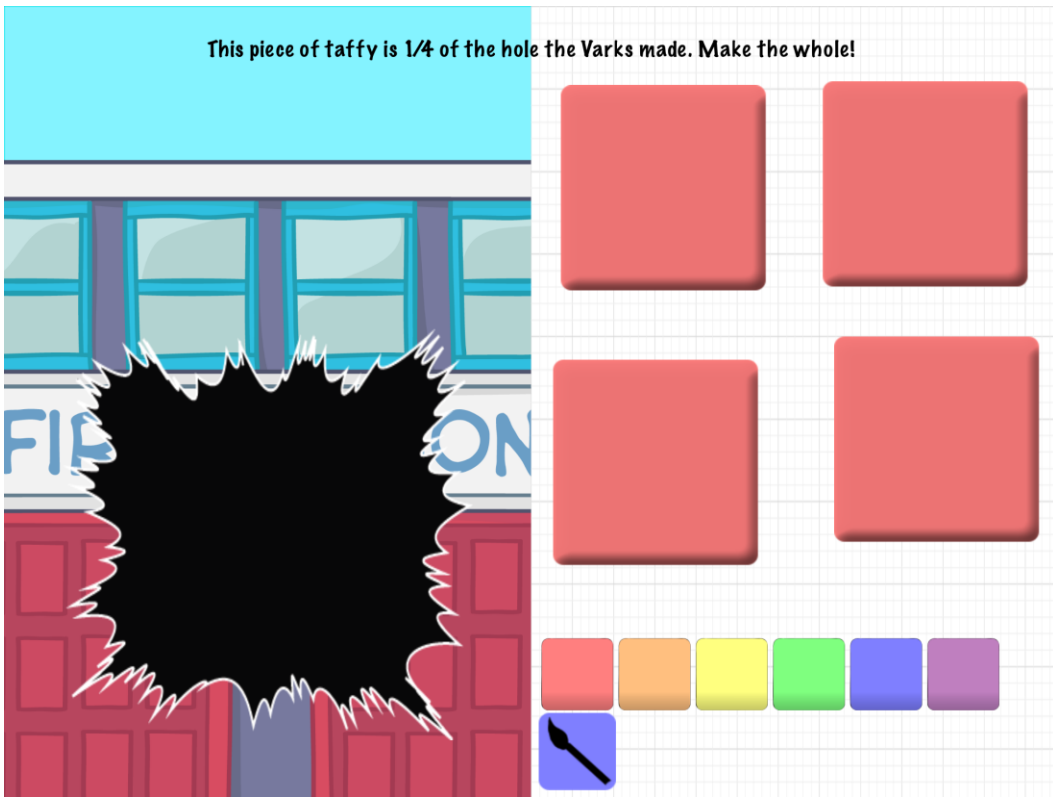


Figure 25. Activated color palette.

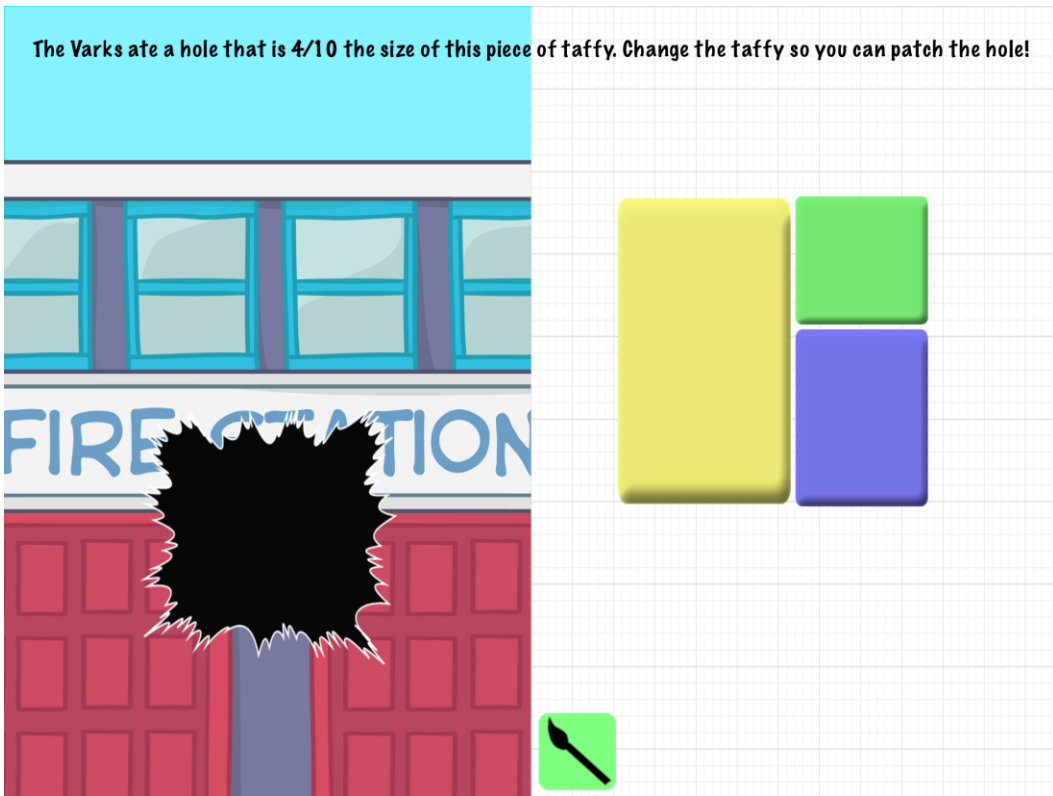
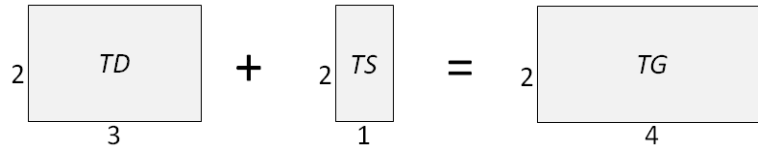


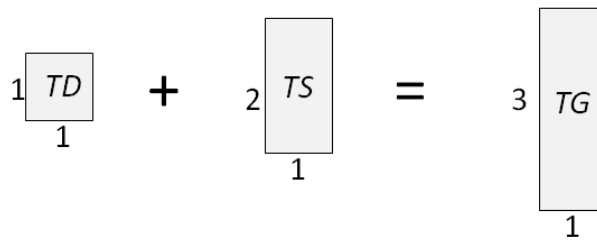
Figure 26. Taffy pieces painted with several colors.

3.3.1.7 Glue

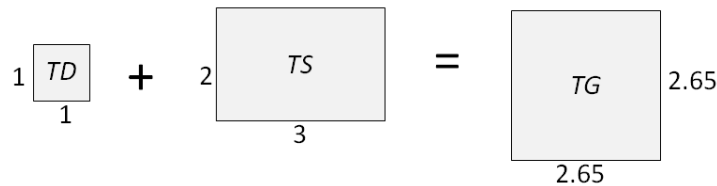
Multiple taffy pieces can be combined or “glued” together by dragging a piece of taffy TD into a stationary piece of taffy TS . As the pieces touch, they are combined into a single larger piece of taffy TG . Since taffy of various sizes can be glued together, the game performs the following algorithm to glue the taffy.



1. If the heights h of TD and TS are equal, $w^{TG} = w^{TD} + w^{TS}$ and $h^{TG} = h^{TS}$.



2. If the widths w of TD and TS are equal, $w^{TG} = w^{TS}$ and $h^{TG} = h^{TD} + h^{TS}$.



3. Otherwise, $w^{TG} = \sqrt{A^{TD} + A^{TS}}$, $h^{TG} = \sqrt{A^{TD} + A^{TS}}$.

Gluing two pieces of taffy with different colors does not matter. The color of the dragged taffy is chosen as the color of the final taffy piece.

Information sent to MASS when gluing taffy only consists of the action itself.

3.3.1.8 Workspace Reset

In order to avoid situations where all taffy pieces are deleted by accident or the player wants to start over, the state of the workspace scene can be reset by shaking the iPad. The taffy is replaced just as it originally started. All previous user interaction is removed from the interaction tracking system.

3.3.1.9 Patching the Hole

The last player interaction is “patching” the hole created by the Varks. As the player moves taffy pieces over the hole, they become transparent. This allows the player to see the hole underneath the potential answer. When taffy piece is over the hole and transparent, the player can swipe the taffy using three fingers. This action signals the game to determine if the submitted taffy piece satisfies the question. If the question is correct, the game fades out the hole showing the building as if it was patched by the taffy

piece as shown in Figure 21Figure 27. **Correctly patching the hole causes the hole to fade away.** Correctness is determined by calculating the area of the submitted taffy. If that area satisfies the fractional question within an 8% error threshold, the patch is considered correct. Conversely, any taffy piece outside the 8% error threshold is incorrect and does not cause the hole to fade. Regardless on the results of the patch, the game transitions back to the town scene.

Patching the hole also causes the game to send collected data to MASS. The data sent as a result of this interaction includes the question text, question type, correctness, and all manipulations performed. This transmission is automatic and requires no explicit effort from the player.

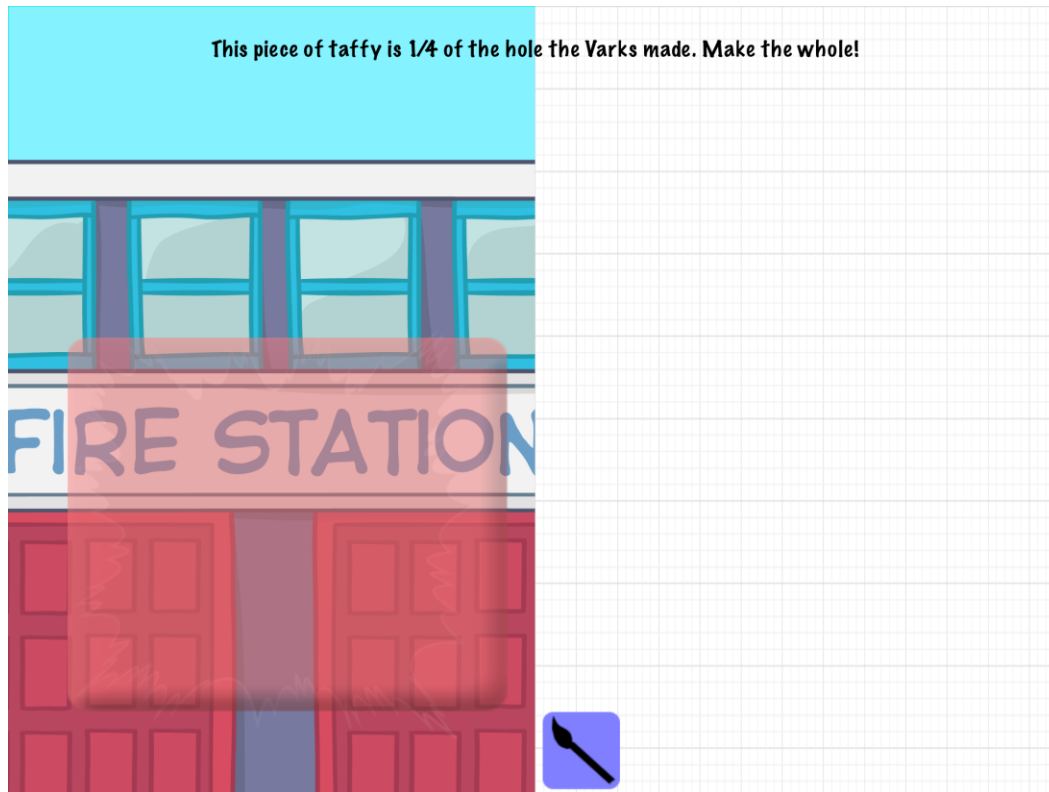


Figure 27. Correctly patching the hole causes the hole to fade away.

CHAPTER 4: MASS Web Application and Services

A software application running on a server computer named *Mobile Application Statistics Service* (MASS) is developed to serve as the server-side application in our client-server solution. The web application displays and updates graphs of gameplay data on the dashboard as the player is playing Taffy Town. MASS is composed of two primary components; the web application and web services.

The web application provided by MASS allows registered players to log into a website to view various graphs depicting all the data collected and transferred from Taffy Town. Players automatically have an account created whenever they create a new account in Taffy Town. After a player logs into the web application, they are presented with three data visualizations corresponding to that player’s specific gameplay. Additionally, an administrator can log in to MASS to obtain aggregated results across all registered Taffy Town players. These visualizations do not uncover any specific individual’s performance or display information that would identify a player.

The second component of MASS is three RESTful web services [Fielding 2000]. The *Authentication Service* provides a mechanism for Taffy Town to authenticate players when logging in to play the game. The *User Service* allows clients to create accounts, modify account information, and retrieve specific user data. Lastly, the *Statistics Service* is responsible for processing and creating graphical visualizations of the data collected during Taffy Town gameplay. In the following sections, each component is described in detail.

4.1 MASS Authentication

When a student or research administrator navigates to MASS using a web browser, an authentication web page is displayed as shown in Figure 28. The page consists of a header, authentication form, and footer as described below.

1. *Header:* Contains a link to MASS’s home page and a logout hyperlink.
2. *Authentication form:* The form allows the user to input the email address and password linked to his or her account. Clicking the “Sign in” button submits the form. Successful login forwards to the user’s dashboard described in 4.2. Erroneous login attempts result in a message portrayed in Figure 29.
3. *Footer:* Copyright information and a hyperlink to the Mobile Software Engineering Lab.



Mass Login

Please sign in

Email address

Password

Sign in

© Virginia Tech. Mobile Software Engineering Lab. 2014

Figure 28. MASS Web Application login page.

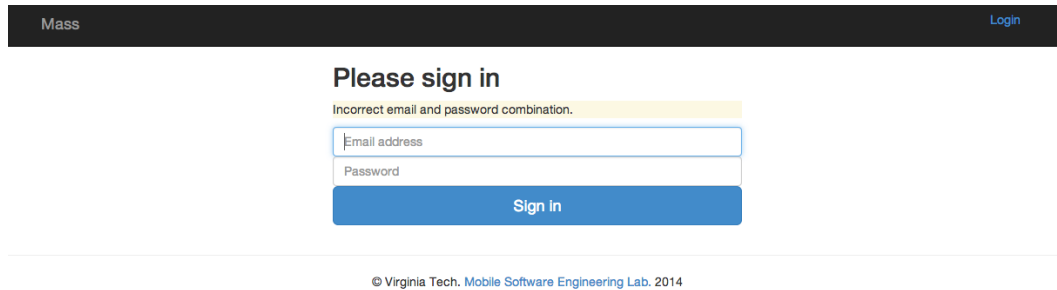


Figure 29. MASS Authentication Error.

4.1.1 Account Types

A user is any person that can log into MASS’s web application. MASS supports two user account types; Player and Administrator. This thesis uses the term “user” when discussing topics that applies to both Player and Administrator.

4.1.1.1 Player

A player account is created during Taffy Town account creation described in 3.1.1. A player is able to view performance data related to Taffy Town gameplay in real-time. No other actions are available to the player account.

4.1.1.2 Administrator

The Administrator account type enables researchers to log into MASS and view gameplay data captured and aggregated across all Taffy Town players in real-time. No information is presented to the research admin that could identify any of the individual players.

Only a single research admin account exists in MASS that is specific to Taffy Town gameplay information. Creating this account type is not exposed in MASS’s web application.

4.2 MASS Dashboard

Figure 30 shows the dashboard presented to a user on successful login. This page has a main content section that visually organizes information captured from Taffy Town to produce three graphs explain further in 4.2.1. The page updates every thirty seconds automatically to ensure the visualizations are up to date. If the MASS has no gameplay information collected from this user, the Dashboard displays as shown in Figure 31.

The research admin’s dashboard is visually the same, however, displays aggregated results from all players of Taffy Town. Figure 32 shows the research admin’s dashboard with combined data.

Lastly, the dashboard features the same header and footer as in 4.1. By reusing these elements, the experience using MASS is a cohesive web application.

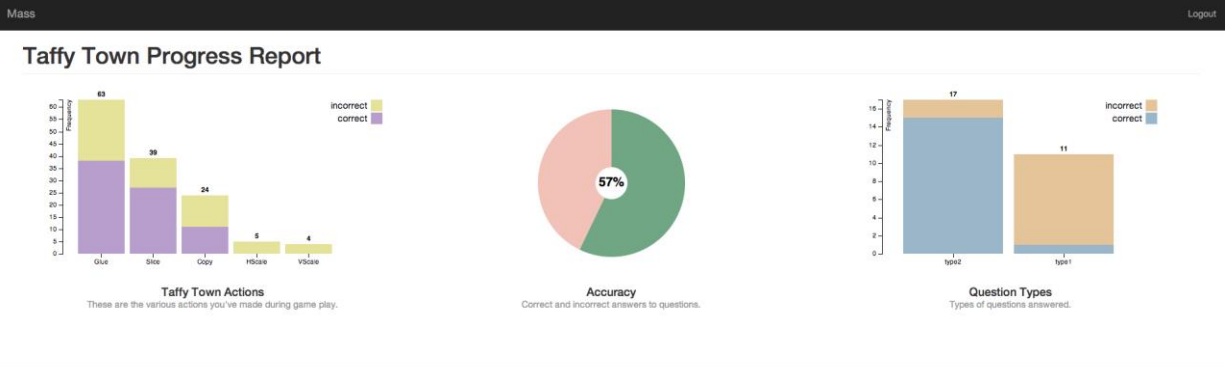


Figure 30. MASS Dashboard.

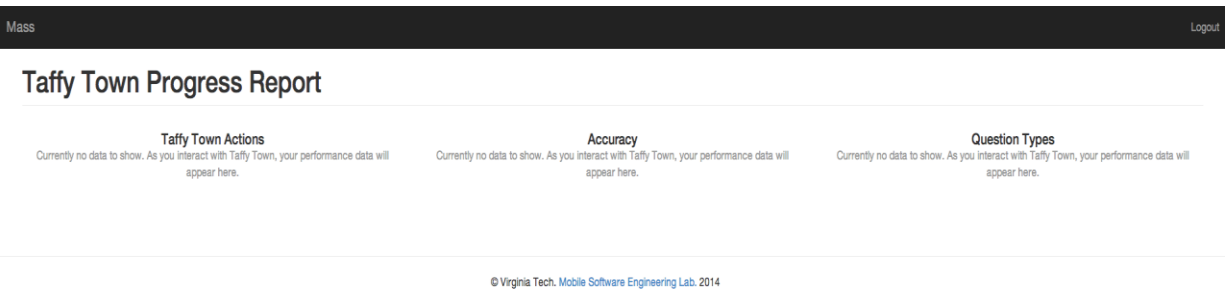


Figure 31. MASS Dashboard with no game data collected.

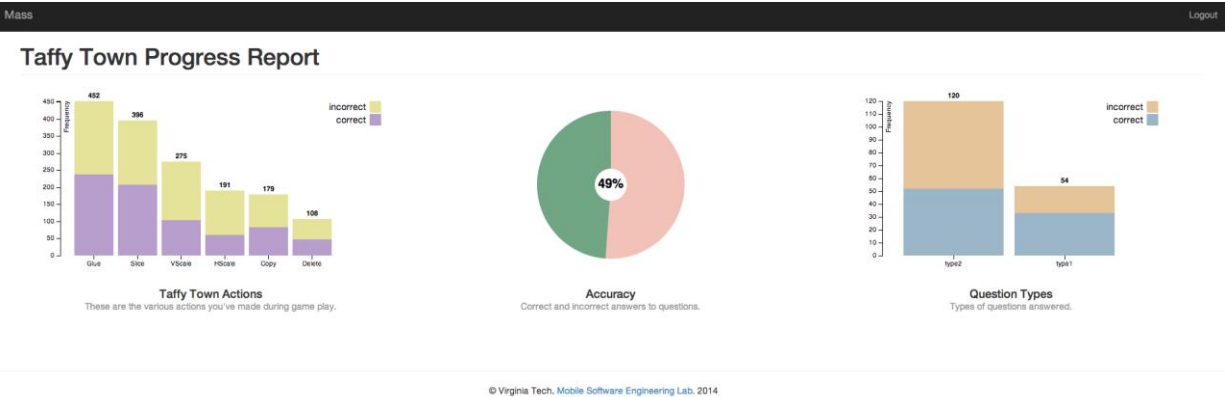


Figure 32. Research Admin's aggregate results on the dashboard.

4.2.1 Data Visualizations

The user's dashboard features three distinct data visualizations; actions stacked bar chart, accuracy pie chart, and question type distribution stacked bar chart [Murray 2013]. We utilized D3.js and crossfilter.js, two JavaScript libraries in order to display these graphics in real-time.

D3.js is a library that allows HTML documents to be manipulated based on a given dataset. It uses HTML, SVG, and CSS, allowing a data-driven approach in creating and manipulating various DOM elements [Bostock 2014].

Crossfilter.js is a multi-dimensional processing library created by Square to allow merchants to dissect payment history datasets [Square 2012]. It allows websites to manipulate, reduce, and filter extremely large datasets.

MASS uses a four-step process in conjunction with D3.js and crossfilter.js in order to generate each SVG graph. MASS polls the statistics service every thirty seconds in order to load new information. If the dataset is stale, these steps are triggered causing the charts to animate with the new information.

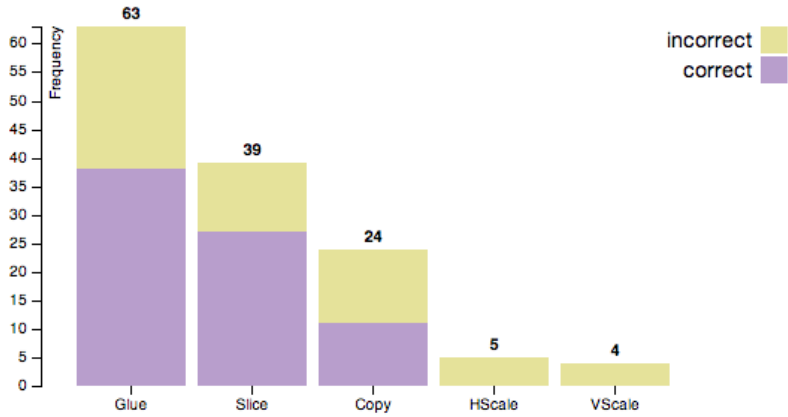
1. *Dataset Retrieval*: MASS connects to the statistics web service and pulls down a set of data specific to the chart being generated. This is called "raw player game data".
2. *Crossfilter Transformation*: The dataset is then grouped, aggregated, and filtered on various indices custom to the chart being generated. This set of data is referred to as "filtered player game data".
3. *Graph Transformation*: Each graph expects the data to be in a specific format. This step in the process allows the filtered player game data to be transformed into a format D3.js understands. Data after this process is referred to as "transformed player game data".
4. *Graph Rendering*: Transformed player game data is loaded into custom D3.js scripts that generate the SVG and displays it on the dashboard.

The next three sections will describe each chart in detail.

4.2.1.1 Actions Stacked Bar Chart

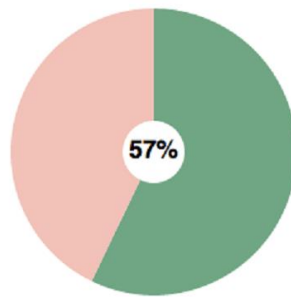
Figure 33 illustrates the first Taffy Town graph displayed on the far left of the MASS dashboard. This graph correlates actions performed by players and accuracy of the answer. The chart's components are described below [SAS 2014].

1. *X-Axis*: Represents the action performed by the player.
2. *Y-Axis*: The *total* number of times the action was performed for all Taffy Town gameplay.
3. *Groups (Bars)*: Each group or bar is composed of *correct* and *incorrect* subgroups stacked.
4. *Legend*: Distinguishes between the *correct* and *incorrect* subgroups by using different colors.



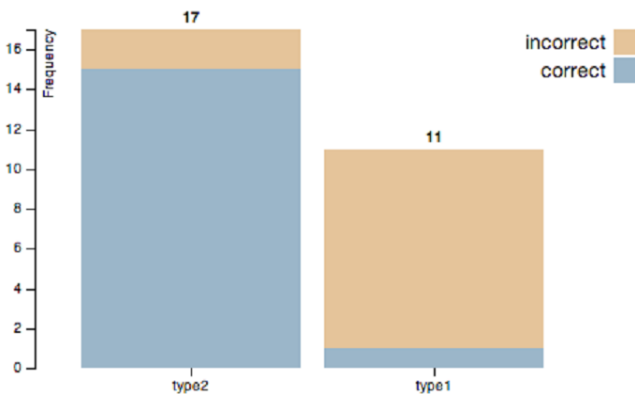
Taffy Town Actions
 These are the various actions you've made during game play.

Figure 33. Stacked barchart depicting various actions a Taffy Town player performed.



Accuracy
 Correct and incorrect answers to questions.

Figure 34. Pie chart showing the distribution of correct and incorrect answers.



Question Types
 Types of questions answered.

Figure 35. Stacked bar chart correlating question type and accuracy

4.2.1.2 Accuracy Pie Chart

The middle pie chart, Figure 34, shows the percentage of *correct* answers. Only two slices exist, one light red representing the incorrect proportion and a green slice representing *incorrect* answers. For easier readability, the exact percentage of correctness is shown in the middle of the chart.

4.2.1.3 Question Type Stacked Bar Chart

In order to understand which type of question leads to more correct answers, Figure 35 shows the relationship between question type and accuracy. 3.3 describe the two question types implemented in Taffy Town. The question type chart is composed of several components described below.

1. *X-Axis*: Represents the type of question answered by the player.
2. *Y-Axis*: The *total* number of times the question was answered by a player.
3. *Groups (Bars)*: Each group or bar is composed of *correct* and *incorrect* subgroups stacked on top of another.
4. *Legend*: Distinguishes between the *correct* and *incorrect* subgroups by using different colors.

4.3 Web Services

The second component of MASS is several RESTful web services. These web services provide a lightweight conduit for client-server communication. Each service provides a specialized set of functionality, separating various logical concerns.

All services operate by using *Hypertext Transfer Protocol* (HTTP) over a TCP/IP network. Clients send various HTTP requests to these web services in order to request data or to invoke some type of functionality. All of MASS's web services support only two *Multipurpose Internet Mail Extension* (MIME) types; "application/xml" and "application/json". These MIME types specify the format of the entity-body. The MIME type "application/xml" signifies the entity-body to be encoded in *Extensible Markup Language* (XML). Likewise, "application/json" indicates *JavaScript Object Notation* (JSON) represents the entity-body [Gourley 2002]. Each web service conforms to a RESTful architecture by exposing an interface invoked by various HTTP verbs described below [Richardson and Ruby 2007].

Table 2. HTTP Verb descriptions

HTTP Verb	Description
GET	Retrieves an object from a web service. The entity depends on what service the request is sent to.
POST	Add the object contained in the request's body to MASS.
PUT	Update a specific object.
DELETE	Removes the target object from MASS.

Each MASS web service offers a "health" check for clients in order to ensure the service is operational. The service does not return any document, however, *success* is determined by the HTTP status code. HTTP status code classification ranges are presented in Table 3 [Gourley 2002].

Table 3. Classification of HTTP status codes

Defined Range	Category
100-101	Informational
200-206	Successful
300-305	Redirection
400-415	Client Error
500-505	Server Error

The next section describes each of the web services by describing supported HTTP actions. Each action provides example request and response messages along with a description of the purpose of the action in relation to Taffy Town.

4.3.1 App Web Service

Before MASS can collect information from a client application, the *app* is required to register with MASS. An *app* is an object that represents a registered application with MASS that wants to have information collected. The app web service provides an interface to add, retrieve, and delete applications from MASS. An *app* has the following information:

1. *Name*: The name of the application that is registered with MASS.
2. *UUID*: The *universally unique identifier* that is associated with the application.

Requests to the app web service are sent to “http://hostname:port/Mass/api/app” in order to invoke various functions related to apps. No clients use the App web service directly. Taffy Town is the only application that is registered with MASS, however, more could be added.

4.3.1.1 Create

In order to create an app with MASS, a HTTP POST is sent to the App web service. The POST request contains a representation of the app to create as well as various HTTP headers as depicted in Table 4. Creating an app requires only a single datum of information, the *name* of the application to create.

Table 4. HTTP POST Request to create an app.

POST /Mass/api/app HTTP/1.1 User-Agent: Chrome/38.0.2125.122 Host: hostname Content-Type: application/json { “name”: “Taffy Town” }
--

If the app is successfully created, the response includes the URI of the app as shown in Table 5.

Table 5. HTTP Response creating an app

```
HTTP/1.1 201 CREATED
Location: http://hostname:port/Mass/api/app/9148e793-9ff1-4cae-b125-717fb43b4d38
Date: Mon, 2 Jun 2014 12:28:53 GMT
Server: WebSphere Application Server/8.0
X-Powered-By: Servlet/3.0
Content-Length: 0
```

4.3.1.2 Retrieve

App retrieval is accomplished by clients issuing a GET request to the app service. The client must know the UUID associated with the target app. The URI to send the request takes the form of “http://hostname:port/Mass/api/app/{APP_UUID}” . Table 6 and Table 7 show example request and responses respectively.

Table 6. GET request to retrieve app from the app web service.

```
GET /Mass/api/app/9148e793-9ff1-4cae-b125-717fb43b4d38 HTTP/1.1
User-Agent: Chrome/38.0.2125.122
Host: hostname
Accept: application/json
```

Table 7. GET response from retrieving an app from the app web service.

```
HTTP/1.1 200 OK
Date: Mon, 2 Jun 2014 12:28:53 GMT
Server: WebSphere Application Server/8.0
X-Powered-By: Servlet/3.0
Content-Type: application/json
Content-Length: 93

{
  "id": 351,
  "name": "Taffy Town",
  "uuid": "9148e793-9ff1-4cae-b125-717fb43b4d38"
}
```

4.3.1.3 Delete

Removing an app is achieved by sending a DELETE HTTP request to the app web service. This permanently removes the app matching the UUID from MASS, including any associated data collected. Example request and response messages are shown in Table 8 and Table 9.

Table 8. HTTP delete requesting to delete the specified app.

```
DELETE /Mass/api/app/9148e793-9ff1-4cae-b125-717fb43b4d38 HTTP/1.1
User-Agent: Chrome/38.0.2125.122
Host: hostname
Accept: application/json
```

Table 9. HTTP delete response indicating the result of the delete request.

```
HTTP/1.1 200 OK
Date: Mon, 2 Jun 2014 12:28:53 GMT
Server: WebSphere Application Server/8.0
X-Powered-By: Servlet/3.0
Content-Type: application/json
Content-Length: 0
```

4.3.2 Group Web Service

A *group* is an object that is used to classify users during authentication and authorization mechanics used in MASS [Credle et al. 2007]. Each group corresponds to a security role defined in WebSphere Application Server that is used for declarative security. MASS only has two supported groups; *Player* and *Researcher*.

A group only has two members contained in its representation.

1. *ID*: is the primary key used by the RDBMS
2. *Name*: The name of the group. This value must be unique to MASS.

Even though CRUD operations are not exposed to clients, a group web service was still created to maximize extensibility with two supported operations; creation and retrieval. Delete and update was not exposed to minimize the risk of deleting required security roles [Sadler, et al. 2009].

4.3.2.1 Create

A security group can be created for MASS by sending a POST request to <http://hostname/Mass/api/group> as depicted in Table 10.

A successful group creation request results in a response similar to Table 11.

Table 10. HTTP request to create a group in MASS.

```
POST /Mass/api/group HTTP/1.1
User-Agent: Chrome/38.0.2125.122
Host: hostname
Content-Type: application/json

{
    "name": "Group Name"
}
```

Table 11. Successful group creation response in MASS.

```
HTTP/1.1 201 CREATED
Location: http://hostname:port/Mass/api/group/34
Date: Tue, 3 Jun 2014 10:12:03 GMT
Server: WebSphere Application Server/8.0
X-Powered-By: Servlet/3.0
Content-Length: 0
```

4.3.3 User Web Service

The user web service provides functionality related to user account objects. A *user* corresponds to a unique account that is able to log into MASS. Additionally, the user is associated with statistics as described in 4.3.4. A user has several fields of data as described below.

1. *First Name*: The first name of the user.
2. *Last Name*: The last name of the user.
3. *Password*: An encrypted version of the user's password.
4. *Email*: The email address of the user. This is used during the login process only. MASS does not send any emails to users.
5. *Group Name*: The name of the group the user belongs to.
6. *UUID*: A universally unique identifier that is associated with the user.

The user service allows clients to create and retrieve user objects. Additionally, the services support other business processes including; user authentication, user app registration, and getting a list of registered apps a user is registered to.

4.3.3.1 Create

Creating a user in MASS is achieved by sending an HTTP POST request to the user web service. The data required in the document included in the request has several required fields.

- Email
- Password
- Confirm Password
- First Name
- Last Name
- Group Name

Table 12 shows an example request and Table 13 shows an example successful response from MASS.

Table 12. HTTP request creating a player user account.

```
POST /Mass/api/user HTTP/1.1
User-Agent: Chrome/38.0.2125.122
Host: hostname
Content-Type: application/json

{
  "email": "someone@somewhere.com",
  "firstName": "Jane",
  "lastName": "Doe",
  "password": "P@ssword123",
  "confirmPassword": "P@ssword123",
  "groupName": "Player"
}
```

Table 13. HTTP response from a successful user creation request.

```
HTTP/1.1 201 CREATED
Location: http://hostname:port/Mass/api/user/2543a793-9bb1-5cce-b754-132fb64b8d43
Date: Tue, 3 Jun 2014 10:12:03 GMT
Server: WebSphere Application Server/8.0
X-Powered-By: Servlet/3.0
Content-Length: 0
```

4.3.3.2 Retrieval

Retrieving a specific user from the web service only requires the UUID associated to the targeted user. Table 14 shows the request needed in order to get a user from MASS.

Table 14. HTTP request to fetch a user account from MASS.

```
GET /Mass/api/user/2543a793-9bb1-5cce-b754-132fb64b8d43 HTTP/1.1
User-Agent: Chrome/38.0.2125.122
Host: hostname
Accept: application/json
```

Upon receiving a valid request from a client, MASS sends a successful response containing the user document formatted as specified in the *Content-Type* HTTP header as illustrated in Table 15. Notice the password is not contained in the response in order to increase the security of MASS.

Table 15. Successful response containing a user document.

```
HTTP/1.1 200 OK
Date: Mon, 2 Jun 2014 12:28:53 GMT
Server: WebSphere Application Server/8.0
X-Powered-By: Servlet/3.0
Content-Type: application/json
Content-Length: 93

{
  "firstName": "Jane",
  "lastName": "Doe",
  "email": " someone@somewhere.com ",
  "uuid": "2543a793-9bb1-5cce-b754-132fb64b8d43"
}
```

4.3.3.3 User Authentication

Clients can authenticate a user through the user web service by sending email and password. The server responds with a unique token that is subsequently used when sending statistic information [Sadler, et al., 2009]. This unique token uniquely identifies an authenticated user. Table 16 depicts an example HTTP request performing a user authentication.

Table 16. Authentication request sent to MASS.

```
POST /Mass/api/user/auth
User-Agent: Chrome/38.0.2125.122
Host: hostname
Content-Type: application/json

{
  "email": " someone@somewhere.com ",
  "password": " P@ssword123"
}
```

Notice the password is not encrypted. This is a security vulnerability that is addressed in the future. If the authentication request is successful, the server sends a response containing a token as shown in Table 17.

Table 17. Authentication response from MASS.

```
HTTP/1.1 200 OK
Date: Mon, 2 Jun 2014 12:28:53 GMT
Server: WebSphere Application Server/8.0
X-Powered-By: Servlet/3.0
Content-Type: application/json
Content-Length: 93

{
  "token": "2543a793-9bb1-5cce-b754-132fb64b8d43",
}
```

4.3.3.4 User App Registration

Before statistic collection for a specific app can occur, a user must be registered with an app. The user web service provides this functionality by enabling clients to send requests as in Table 18.

This associates the user's UUID provided in the URI with the app document contained in the request payload. If either the user or app does not exist, an error response is returned to the client. Table 19 shows the response returned by the server when the request is successful. This contains all registered apps of the specified user.

Table 18. Register app to user account HTTP request.

```
POST /Mass/api/user/2543a793-9bb1-5cce-b754-132fb64b8d43/apps
User-Agent: Chrome/38.0.2125.122
Host: hostname
Content-Type: application/json

{
  "uuid": "9148e793-9ff1-4cae-b125-717fb43b4d38"
}
```

Table 19. Successful response when registering an app to a user.

```
HTTP/1.1 200 OK
Date: Mon, 2 Jun 2014 12:28:53 GMT
Server: WebSphere Application Server/8.0
X-Powered-By: Servlet/3.0
Content-Type: application/json
Content-Length: 93

[[
  {
    "id": 51,
    "name": "Taffy Town",
    "uuid": "4d4c80bf-8940-4ae2-bf59-766865118f6c"
  }
]]
```

4.3.3.5 List User Registered Apps

The last method of the user web service is listing the apps the user currently has registered. Table 20 show a request made to MASS in order to get a list of all apps registered to a user. The response is the same as Table 19 when the request is successful.

Table 20. HTTP request to get a list of registered apps for a user.

```
GET /Mass/api/user/2543a793-9bb1-5cce-b754-132fb64b8d43/apps
User-Agent: Chrome/38.0.2125.122
Host: hostname
Accept: application/json
```

4.3.4 Statistics

The statistics API is responsible for the submission and retrieval of data collected in MASS. A *statistic* can be *any type* of information in conjunction with an “appID” and “userID” pair. The following describes a *statistic* entity-body’s fields needed for logging information in MASS.

1. *appID*: The UUID of the app the data is coming from.
2. *userID*: The UUID of the user that statistic represents.
3. *JSONMetadata*: An object serialized in JSON that represents the data to send from the client to MASS.

MASS is designed to allow client applications to define and send data specific to the client app. Taffy Town’s targeted data is actions performed by players during gameplay. A different app could send custom information to MASS’s statistics web service in addition to Taffy Town. This is achieved by utilizing MongoDB, a NoSQL database server for storage of client/app information [Copeland 2013].

Three methods are available to invoke on the statistics API; create a new statistic, retrieve all stats for a specific user and app, retrieve all statistic objects for an app.

4.3.4.1 Create

Sending an HTTP POST with an MIME type encoded statistics object to the statistics web service creates a single statistics object. Table 21 shows an example Taffy Town request, creating a statistic for answering a question. The JSON contains the unique appID to Taffy Town and the userID that performed the actions on the client.

Successful creation is denoted by a response with a status code of 201.

Table 21. HTTP request sent by Taffy Town to create a statistic.

```
POST /Mass/api/stats
User-Agent: Chrome/38.0.2125.122
Host: hostname
Content-Type: application/json

{
  "appId" : "4d4c80bf-8940-4ae2-bf59-766865118f6c",
  "userId" : "3375f110-db19-4211-a27d-85df1b8dcecc",
  "JSONMetadata" :
  {
    "question" : " This piece of taffy is 1/4 the hole the Varks made. Make the whole!",
    "type" : "type2",
    "correct" : false,
    "transformations" :
    [
      {
        "action": "HSlice",
        "variables":
        [
          { "name" : "y", "value" : 154.45 }
        ]
      },
      {
        "action": "VSlice",
        "variables":
        [
          { "name" : "x", "value" : 417.06 }
        ]
      },
      {
        "action": "Delete",
        "variables": [ ]
      },
      {
        "action": "Glue",
        "variables": [ ]
      }
    ]
  }
}
```

4.3.4.2 Retrieval

By issuing a HTTP GET request, statistic objects can be retrieved from the web service. The API offers two ways of getting statistical information. The first is getting all stats for an entire app shown in Table 22. The second method is getting all stats for a user of a specific app shown in Table 23.

Table 22. HTTP request from MASS web application to get all statistics for Taffy Town.

```
GET /Mass/api/stats/4d4c80bf-8940-4ae2-bf59-766865118f6c
User-Agent: Chrome/38.0.2125.122
Host: hostname
Accept: application/json
```

Table 23. HTTP request from MASS web application getting all statistics for a specific user of Taffy Town.

```
GET /Mass/api/stats/4d4c80bf-8940-4ae2-bf59-766865118f6c/3375f110-db19-4211-a27d-85df1b8dcecc
User-Agent: Chrome/38.0.2125.122
Host: hostname
Accept: application/json
```

The dashboard displaying the three charts uses this endpoint to get the data to produce the visualizations. The user's group determines with endpoint is used to load the data. Any user belonging to the "Player" group sends a request with the UUID of the user and the appID of Taffy Town to load the statistics. Any user belonging to the "Researcher" group, only the appID is used to get all stats for Taffy Town.

CHAPTER 5: Self-Evaluation of Client-Server Solution

This chapter presents a self-evaluation of the client-server architecture for game-based learning data collection. The client is a game called Taffy Town described in CHAPTER 3: . The server application created to capture statistics from Taffy Town is described in CHAPTER 4: .

The self-evaluation is performed by using the following quality attributes: functionality, usability, reliability, performance, and maintainability [Pressman 2010].

5.1 Functionality

Functionality is assessed by evaluating the features and capabilities of the system. This attribute ensures the software solution meets the needs of the customer by ensuring the functional requirements are offered as capabilities in the system.

We utilized the process of *Requirements Engineering* and identified an initial set of user stories for both the client and server applications. As an iteration completed, a demo of functionality was demonstrated to key stakeholders. Any gaps in functionality or changes discovered during the demo led to the creation of new user stories. This cycle of refinement repeated until all features within scope of this document were satisfied [Balci 2014].

5.2 Usability

Usability is the degree to which the software system is easy to learn, easy to use, and promotes high user satisfaction [Rosson and Carroll 2002].

The next two sections discuss how Taffy Town and MASS satisfied the usability quality attribute.

5.2.1 Taffy Town

Taffy Town was designed to be used by middle school students. Influences from other educational mathematics-based games were analyzed in order to understand how to make Taffy Town usable for the targeted audience. One of the key findings was engagement. We needed to develop a story that would keep the student playing Taffy Town to maximize the amount of data collected by MASS. To solve this, we created a story revolving around taffy-eating aardvarks eating holes in the buildings of Taffy Town. Having a meaningful story takes the focus off of education. The players are no longer “just doing math problems”. This leads to a higher user satisfaction as opposed to a strictly mathematics-based game.

Each user interaction available to the player was developed by mimicking the physical world in relation to taffy. Players can stretch, cut, and merge taffy by using simple and intuitive touch-based gestures. This makes Taffy Town easy to use.

One identified area of improvement is the inclusion of a *tutorial* mode in which the player is taught all the available gestures. It’s not very intuitive to understand some of the other actions you can perform on a piece of taffy. For example, three-finger swiping to patch the hole is not shown to the user in game.

5.2.2 MASS

The web application provided by MASS enables players to login and view charts depicting various metrics related to gameplay performance. Each graph has a legend describing what each group represents as well as axis labels. Additionally, a short description is displayed at the bottom of each chart to reinforce the purpose of the corresponding graph. These labels make understand the purpose of the graphics easier.

By employing a uniform webpage layout, MASS's look and feel remains static as the user navigates from the login screen to the dashboard. By having this consistent template the system is easier to use [\[Rosson and Carroll 2002\]](#).

5.3 Reliability

Reliability is the frequency and severity of failure in the software system. Additionally, the accuracy of the results and the predictability of the program also play a key role in the reliability of the application [\[Pressman 2010\]](#).

5.3.1 Taffy Town

The development of Taffy Town included robust error handling in objective-c. This ensures any errors encountered during the game results in a user friendly error message. An example of this occurs when a player does not login using the correct username and password as shown in Figure 13. This error message provides the player with details on why the login did not succeed without catastrophically ending the app.

Additionally, the taffy gestures always perform how they are intended. Stretching always moves in the direction the fingers are moving at the same rate. Patching the hole utilizes the same error threshold value for all questions. This enhances the predictability of Taffy Town.

5.3.2 MASS

The web application also utilizes exception handling in the code. Any error is reported to the user in a friendly consistent manner. Figure 29 shows an authentication error displayed to the user.

All the web services offered by MASS also give meaningful errors to clients. For example, trying to get statistic objects from a user that is not registered to the app results in an appropriate error message and HTTP response code.

5.4 Performance

Performance is measured by the speed in which the software executes, the response times, and resource consumption [\[Pressman 2010\]](#).

5.4.1 Taffy Town

In today's highly complex and graphical games, performance is measured in terms of frames per second (FPS). As FPS lowers, games start to suffer from various graphic anomalies like tearing, jittering, and general slow rendering. This can cause the game to become unplayable.

The performance of Taffy Town was measured by taking the average FPS over a period of ten minutes of gameplay. It performed within an acceptable average FPS of 52.24 with no noticeable dips.

The high performance of Taffy Town is attributed to sprite kit, a 2D gaming engine. Sprite Kit offers performance enhancing features such as sprite sheets, textures, caching, and memory optimization [Berg et al. 2013].

5.4.2 MASS

The performance of MASS's web application was measured in load times for various users. [Work 2014] suggests 40% of users abandon a website that takes more than 3 seconds to load. For MASS to be performant, we ensured the dashboard would load faster than three seconds over a heavy load.

To test this, JMeter was utilized to simulate 20 users loading the dashboard every 5 seconds while we loaded the dashboard as the admin [Apache Software Foundation 2014]. We measured MASS's average load time to be 2.91 seconds. This barely passes the established benchmark of 3 second load times.

Several contributing factors are causing the page to load slower than expected.

1. Accounts with large data sets take longer to load. The admin account loads *all* statistics for Taffy Town.
2. Every time the dashboard loads, all statistics for a user are fetched again.
3. IBM WebSphere, MASS web services, MASS web application, DB2, and MongoDB are all on the same server.

5.5 Maintainability

Maintainability is the degree software can be extended, adapted, and changed due to bugs, new features and enhancements, or external environment evolution.

Both Taffy Town and MASS were designed using SOLID principals [Martin 2000]. SOLID is an acronym representing five principals of object-oriented programming and design.

1. *Single Responsibility* - a class or entity should only have a single responsibility.
2. *Open/Closed Principal* – software components should be open for extension, but closed for modification.
3. *Liskov Substitution Principal* – objects can be replaced with instances of their subtypes without altering the correctness of the program.
4. *Interface Segregation Principal* – many specific interfaces are better than a single general-purpose interface.
5. *Dependency Inversion Principal* – components should depend on abstractions and not concretions.

In addition to adhering to SOLID principals, object-oriented languages were used in the implementation. Taffy Town used Objective-C and MASS used Java Enterprise Edition. Both Taffy Town and MASS employed a domain model in which an object model of the problem domain was created [\[Fowler 2003\]](#). This enabled us to build a highly cohesive and low coupled client-server solution with high maintainability.

CHAPTER 6: Conclusions and Future Research

6.1 Conclusions

This thesis describes the research and software development of MASS, a server solution that collects gameplay-based statistics from Taffy Town, an engaging fractions-based game. Current educational games offer only obtrusive or hard-to-extract performance data from a player's device. Our client-service solution solves this invasiveness by silently pushing gameplay data to MASS's web services. This data is then transformed and animated into insightful charts as part of a dashboard provided by MASS's web application, allowing students to see progress and researchers to gain insights on how students solve particular problems in real-time.

Each system was created by utilizing an agile software development lifecycle. A single phase of architecture design was held which led to choosing IBM WebSphere, IBM DB2, and MongoDB as the server software. We held various design meetings with key stakeholders that drove the creation of user stories. As features were developed and deployed, demonstrations of Taffy Town and MASS were given. Any changes or requests discovered during these demos resulted in new user stories. This cycle of "progressive refinement" lasted until the features exposed by Taffy Town and MASS were within scope of this thesis.

The final client-server system quality attributes are found to be acceptable based on our self evaluations. Taffy Town is an easy to use game in which students solve fractional problems by manipulating taffy. By creating an engaging story, the educational aspects of the game are concealed, further increasing the desire to play the game. MASS's dashboard provides easy to read assessment of gameplay performance for both players and researchers. By being able to correlate gameplay data with performance, researchers can gain insights on how students solve specific fraction-based problems.

6.2 Contributions

The contribution of this body of work is the client-server architecture that provides a means to collect targeted gameplay data and display visualizations of that data in real-time. Players can log in and see personalized game-based collected data. This provides researchers with a tool that facilitates discovery around how students solve problems in educational games. Furthermore, any insights revealed with this tool can be utilized to create more effective lessons and potentially help identify problematic areas for students.

6.3 Future Research

The next two sections describe enhancements and new features that will extend this research even further.

6.3.1 *Taffy Town*

Even though Taffy Town offers students a rich gameplay experience by using multi-touch gestures to perform various taffy manipulations, we have identified the following improvements that push Taffy Town into an even better experience.

1. Add audio elements like music and sound effects.
2. Creating an opening animation, further developing the story of the “Varks” and why you want to save Taffy Town.
3. Include a “training” or “tutorial” mode. This would be a Taffy Taskforce boot camp that is required to complete once on the first play through. After that, the player can choose to go back to this mode and practice.
4. Adding aardvarks to the attacking animation.
5. Enabling the players to “lose” the game. One way of achieving this would be to give each building a “health meter”. Each missed question reduces the meter by a specific amount. If the meter reaches zero, the building crumbles. When all buildings are no longer erect, the student loses the game.

6.3.2 MASS

MASS successfully enabled real-time gameplay data collection, however, we have identified the following improvements that will further enhance MASS into a generic, campus-wide mobile application assessment service.

1. Incorporate a messaging solution by utilizing a service bus. This will provide the mechanism to scale in a fashion that will support hundreds of thousand users and devices simultaneously [[Hohpe and Woolf 2007](#)].
2. Migrate from a polling model when collecting statistics for visualization generation to push architecture. One way of achieving this is to further investigate web sockets.
3. Add the ability to support more than one client app in MASS.
4. Enhance the graphs to support real-time analysis by making them interactive.
5. Enable researchers to create custom graphs or reports to display on their dashboard.
6. Add the ability to create research administrators in the web application.
7. Increase the security by implementing three-legged OAuth [[Allamaraju 2010](#)].
8. Support a teacher-classroom hierarchy in which students register for a class and the teacher can see class performance as well as individual student performance.

REFERENCES

- Albertoni, F., J. Bajerski, D. Barillari, L. Cada, S. Hanson, G. Huang, R. Jain, G. Mendes, C. Mierlea, S. Narain, S. Pinto, J. Ricciuti, C. Sadtler, C. Steege (2013), *WebSphere Application Server V8.5 Concepts, Planning, and Design Guide*, Armonk, NY
- Allamaraju, S. (2010). *RESTful Web Services Cookbook*. O'Reilly Media, Inc., Sebastopol, CA
- Apache Software Foundation (2014), "Apache JMeter," The Apache Software Foundation, <http://jmeter.apache.org/>
- Apple (2010), "Early Jamestown," <https://itunes.apple.com/us/app/early-jamestown/id395229194?mt=8>
- Apple (2014a), "iPad in Education," <https://www.apple.com/education/ipad/apps-books-and-more/>
- Apple (2014b), "SpriteKit Programming Guide," https://developer.apple.com/library/ios/documentation/GraphicsAnimation/Conceptual/SpriteKit_PG/Introduction/Introduction.html#//apple_ref/doc/uid/TP40013043
- Balci, O. (1998), "Verification, Validation, and Testing," In *The Handbook of Simulation*, J. Banks, Editor, John Wiley & Sons, New York, NY, August, Chapter 10, pp. 335-393.
- Balci, O. (2014), "CS3704 Software Engineering Course Notes," <http://manta.cs.vt.edu/cs3704>
- Berg, M., T. Bradley, M. Daley, J. Gundersen, K. Hafizji, and M. Hollemans (2013), "iOS Games by Tutorials," Razeware LLC, Huntingtown, MD
- Bolton, C., J. Langord, G. Berry, G. Payne, A. Banerjee, R. Farley (2012), "SQL Server Concurrency," Logical Read, <http://logicalread.solarwinds.com/sql-server-concurrency-lost-updates-w01/>
- Bostock, M. (2014), "D3.js Overview: D3 Data-Driven Documents," <http://d3js.org/>
- C, K. (2014), "Big Data and Education," The Economist, <http://www.economist.com/blogs/schumpeter/2014/04/big-data-and-education>
- Chodorow, K. (2013), *MongoDB: The Definitive Guide*, O'Reilly Media, Inc., Sebastopol, CA
- Copeland, R. (2013), "MongoDB Applied Design Patterns," O'Reilly Media, Inc., Sebastopol, CA
- Credle, R., T. Chen, A. Kumar, J. Walton, and P Winters (2007), *IBM WebSphere Application Server V6.1 Security Handbook*. IBM, Armonk, NY.
- Drexel University (2014), "Call for Papers for IEEE BigData 2014," <http://cci.drexel.edu/bigdata/bigdata2014/callforpaper.htm>
- Erl, T. (2005), *Service-Oriented Architecture*. Pearson Education, Inc., Upper Saddle River, NJ
- Erl, T., B. Carlyle, C. Pautasso, R. Balasubramanian (2014), *SOA with REST, Principals, Patterns, & Constraints for Building Enterprise Solutions with REST*, Pearson Education, Inc., Upper Saddle River, NJ
- Fielding, R. T. (2000), "Architectural Styles and the Design of Network-based Software Architectures," Doctoral dissertation, University of California, Irvine, CA.
- Fowler, M. (2003), *Patterns of Enterprise Application Architecture*. Pearson Education, Inc., Upper Saddle River, NJ
- Goncalves, A. (2010), *Beginning Java EE 6 Platform with GlassFish 3*. Springer Science+Business Media, LLC, New York, NY
- Gourley, D. (2002), *HTTP: The Definitive Guide*. O'Reilly Media, Inc., Sebastopol, CA
- Hohpe, G. and B. Woolf (2007), *Enterprise Integration Patterns, Designing, Building, and Deploying Messaging Solutions*. Pearson Education, Inc., Upper Saddle River, NJ

- Hwang, G.-J. and P.-H. Wu (2012), “Advancements and Trends in Digital Game-Based Learning Research: A Review of Publications in Selected Journals from 2001 to 2010,” *British Journal of Educational Technology* 43, 1, E6-E10.
- IBM. (2014), “IBM DB2 database software”, IBM, <http://www-01.ibm.com/software/data/db2/>
- Khadaroo, S. (2012), “Not Just 4 Texting: 1 in 3 Middle-Schoolers Uses Smart Phones for Homework,” *The Christian Science Monitor*, <http://www.csmonitor.com/USA/Education/2012/1129/Not-just-4-texting-1-in-3-middle-schoolers-uses-smart-phones-for-homework>
- Martin, R. (2000), “Design Principles and Design Patterns,” http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf
- Meluso, A., M. Zheng, H. A. Spires, and J. Lester (2012), “Enhancing 5th Graders’ Science Content Knowledge and Self-Efficacy through Game-Based Learning,” *Computers & Education* 59, 2, 497-504.
- Murray, S. (2013), *Interactive Data Visualization for the Web*. O’Reilly Media, Inc., Sebastopol, CA
- North Carolina State University (2014), “Crystal Island – Outbreak,” <http://www.intellimedia.ncsu.edu/crystal-island-outbreak/>
- Palasoftware. (2014), “MathBoard,” <http://www.palasoftware.com/MathBoard.html>
- Pfotenhauer, J., D. Gagnon, M. Litzkow, C. Blakesley (2009), “Designing and Using an On-line Game to Teach Engineering,” *39th ASEE/IEEE Frontiers in Education Conference*, San Antonio, TX
- Pressman, R. (2010), *Software Engineering, A Practitioner's Approach*, 7th edition, McGraw-Hill, New York, NY
- Richardson, L. and S. Ruby (2007), *RESTful Web Services: Web Services for the Real World*. O’Reilly Media, Inc., Sebastopol, CA
- Rosson, M. B. and J. M. Carroll (2002), *Usability Engineering, Scenario-Based Development of Human-Computer Interaction*. Morgan Kaufmann Publishers, San Francisco, CA
- Rotem-Gal-Oz, A. (2012), *SOA Patterns*. Manning Publications Co., Greenwich, CT
- Sadtler, C., F. Albertoni, L. Blunt, S. G. Chen, E. Ferracane, and G. Smolko (2009), *WebSphere Application Server V7.0 Security Guide*, IBM, Armonk, NY
- SAS Institute Inc. (2014), “Chart Terminology,” <http://support.sas.com/documentation/cdl/en/graphref/65389/HTML/default/viewer.htm#n0xhef2vixz49cn1ithmurreh5i6.htm>
- Square (2012), “Crossfilter,” <http://square.github.io/crossfilter/>
- Work, S. (2014), “How Loading Time Affects Your Bottom Line,” <https://blog.kissmetrics.com/loading-time/>