Advisor Initials: GTH
Project Number: GTH-SN01

Project Darkstar

A Major-Qualifying Project Report
submitted to the faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science
by

Paul Gibler ______________________

Robert Martin ______________________

Christopher Scalabrini ______________________

Date: December 15th, 2008

Approved: George T. Heineman, Major Advisor

____

Abstract

Project Darkstar is a game-server infrastructure developed by Sun Microsystems to support on-line massively scalable game environments. The WPI project team's primary goal was to increase the usability and attractiveness of the introductory documentation and tutorials for Project Darkstar to appeal to a wider set of developers.

Table of Figures

Contents

1 Introduction

The project team wrote this paper to fulfill the undergraduate requirement at Worcester Polytechnic Institute (WPI) for the Major Qualifying Project. The students collaborating on this project are Paul Gibler, Robert Martin, and Christopher Scalabrini. The team's advisor is George Heineman, and the team's professional liaison at Sun Microsystems is Jim Waldo. The project took place during A-term 2008 with a brief follow-up in B term for completing this report.

1.1 About WPI

WPI is a private university located in Worcester, Massachusetts. As a core part of the undergraduate curriculum, all students pursuing degrees complete a number of projects. Students typically complete the Major Qualifying Project (MQP) in their senior year. The project represents capstone work for each prospective graduate in his or her major. For this project, each student is pursuing a bachelors of science in Computer Science.

WPI offers a unique opportunity in which to complete the MQP. Known as the Global Perspective Program, it allows students to collaborate with companies and organizations around the world. In this case, the project team worked at the Sun Microsystems campus in Burlington, Massachusetts.

1.2 About Sun Microsystems

Sun Microsystems is a computer hardware and software development company based in Santa Clara, California. Sun, a Fortune-500 company,[1] has been a key contributor in technological advancements in the last decade, most notably its server systems and the Java programming language. Sun is developing a variety of open-source projects; Project Darkstar is one such project and is the focus of the project team's MQP. A new technology still in development, the Project Darkstar team envisioned a 15-minute out-of-the-box experience to demonstrate the capabilities of Project Darkstar and to attract potential developers.

1.3 The Project

The project team worked directly with both the Project Darkstar team as well as the project's advisor at WPI. The original goal of the project was to create an out-of-box experience for users downloading Project Darkstar. These early requirements envisioned by the Project Darkstar team were vague and undefined. As such, much of the project work involved experimentation, brainstorming, and proposals to the Project Darkstar team. In Section 1.4, the project team outlines the definition and requirements of an out-of-the-box experience.

[1] "184. Sun Microsystems." CNN Money. Fortune 500. 2008 <http://money.cnn.com/magazines/fortune/fortune500/2008/snapshots/881.html>.

1.4 15-minute Out of the Box Experience

The Project Darkstar team required a way to highlight Project Darkstar's capabilities. They decided that there should be a 15-minute out-of-the-box experience beginning with a simple installation process. Following the installation, the out-of-the-box-experience would include a short and graphically appealing demonstration of the technology. This concept was inspired by out-of-the-box experiences present in technologies such as MySQL, SunSPOTs, and Eclipse.

An out-of-the-box experience is a quick demonstration that will attract potential users to the technology. The project team envisioned that the user would install Project Darkstar and then launch a demonstrative application highlighting of the capabilities of Project Darkstar. In completing this project, we focused on a core set of guiding principles

1.4.1 Simplicity

The demonstration application should be simple and intuitive to use. This goal was particularly difficult, as the main feature the Project Darkstar team wanted to highlight was its simple application programming interface (API). As a result, the project team had to assume all prospective users were programmers. However, the team still wanted to target a larger audience because the goal was to demonstrate not only the ease of the Project Darkstar API, but the power of the technology to game designers as well. The team also sought a way to explain Project Darkstar to managers and project leaders who may not necessarily know Java focusing on how Project Darkstar could reduce total development time of a project. This additional requirement led to the development of a multi-staged experience that starts with a simple demonstration accessible to a large audience and then expands increasingly in-depth to the level of Java developers.

The project team's main goal throughout the out-of-the-box experience was simplicity as it is a key issue in the success of the out-of-box experience. When people attempt to use a product, the inability to make progress or perform a certain task can frustrate users. For Project Darkstar, a potential developer that becomes frustrated through the course of the out-of-the-box experience is lost. During the team's first experience with the provided Project Darkstar binary download, there was a large time investment before the team had anything functional. As a result, the project team's first goal, the Installer (described in section 4), alleviates this problem. The team realized early on that keeping the experience simple, easy to use, and problem-free was instrumental to the success of the project.

1.4.2 Performance

The time investment required by an out-of-the-box experience is a primary determining factor that guides a majority of the experience itself. Originally, the Project Darkstar set-up process was arduous. It required users to perform manually several error-prone steps that could be easily automated. A delayed start due to a drawn-out set-up presents the initial impression of complexity, which would drive away potential users. As the out-of-box experience is simply a demonstration and nothing more, keeping any set-up to an absolute minimum is very important. The length of the total out-of-box experience itself should be brief. The original plan was to keep the total length around 15 minutes. In the team's case, the final duration of the out-of-box experience was altered as requirements changed. While the final product contains an estimated total experience lasting around 2-3 hours, we created a number of short, encapsulated tutorials each of which can be completed in approximately 15 minutes.

1.4.3 Appeal

An out-of-the-box experience needs to be appealing. First and foremost, it generates interest in potential users. Users that are unfamiliar with Project Darkstar are key members of the project's target audience. The out-of-box experience should draw their attention by presenting appealing material. Secondly, it persuades users to use the product. While creating decent software is helpful in accomplishing this, people are going to see the demonstration before they use the final product. As such, creating a demonstration that astounds prospective users is vital. Content is less important here; a person is more likely to be drawn to the demonstration with more flair regardless of the overall product content. Similarly, a successful out-of-box experience will also provide some degree of flashiness. The project team attempts to appeal to users in multiple ways, the best example being the Project Snowman game (see section 6). Here, users see a completed game that uses Project Darkstar rather than the API. What is important is not that Project Darkstar is fun and exciting, but that it can be used to create a game that is. The SunSPOTs out-of-box experience is another good example, discussed Section 1.5.2.

1.5 Examples

Several instances of modern out-of-box experiences exist. Presented below are some of the more relevant examples, specifically those related to technology and programming including: MySQL, the SunSPOTs sample program, and Eclipse (as an out-of-box experience for the Java language).

1.5.1 MySQL

MySQL is a relational database server program. The out-of-box experience provided is very basic. Upon downloading, the user can install the program, which is done through the use of a GUI dialog box. This dialog walks the user through simple steps such as choosing an install directory and selecting which components to install. Once the install is completed, the user is prompted with the option to run and configure immediately. From here, the user can immediately open the MySQL command prompt. The idea behind this out-of-box experience is to create a straightforward set-up that can be achieved with minimal effort, and allow users access to the application as soon as possible. This concept embodies the principle of simplicity, and is a prime example of what the project team wanted to accomplish with the Project Darkstar installer.

1.5.2 SunSPOTs

SunSPOTs are hand-held, mobile, embedded development platform created by Sun Microsystems. The SunSPOT out-of-box experience does not highlight any development tools; rather, it is an expressive program that displays the device's features in a visually appealing fashion. The device includes the following peripherals: an array of LED lights, a set of buttons, an accelerometer, and a wireless mechanism capable of connecting to other SunSPOTs. When the device is activated, one of the LEDs lights and acts as a "bouncing ball." It travels across the LED array until it reaches the end, and then rebounds in the other direction. The SunSPOT's accelerometers, which can detect movement changes, demonstrate this functionality. If a user shakes the device, the "ball" LED will bounce around faster. When brought into close proximity with other SunSPOTs, the "ball" can travel between them, showcasing the wireless support. This sort of example is a great way of generating a lot of appeal to new users by presenting the capabilities of the device in an amusing fashion. This can serve to leave a lasting impression on users and cajole them to use the product.

1.5.3 Eclipse

Eclipse is an Integrated Development Environment (IDE) for multiple languages, most notably Java. Eclipse includes several convenient features including project build management, refactoring, syntax auto-completion, dynamic error messages, and plug-in. Previously, when writing any code with Java, extra work via the command line (using javac and the Java virtual machine) was required in order to compile and run programs. Eclipse effectively creates a simpler environment for development by abstracting out complications that are not always necessary. This greatly reduces the strain of Java programming for newcomers. These qualities embody the essential elements of a successful out-of-box experience: simplifying the experience by increasing production speed and efficiency.

2 Project Darkstar Overview

Project Darkstar is designed to eliminate complexities in software systems that attempt to take advantage of the powerful computers and high-peed networking available today. In the past, software was typically installed as stand-alone applications designed to run on a desktop machine. However, the Internet revolutionized software development and, with it, gaming technology. Taking advantage of the ability to connect users on a large scale, early massively multiplayer online games (MMO) such as Ultima Online and EverQuest created entire persistent worlds of human and computer players, introducing a new style of game-play never before seen. However, the requirements of such systems were immense. Complex distributed systems were designed to support the vast number of players. These systems were difficult and costly to create and maintain; a shortcoming of MMO's that is still relevant today. When World of Warcraft launched, the user base reached the projected twelve-month figure of 300,000 in six weeks.[2] As a result, a variety of technical difficulties and server mishaps ensued, causing uproar from the player community. Server outages in Europe were so bad that Blizzard gave subscribers two days of play time for free as an apology[3]. With game titles as lucrative as the Warcraft franchise, setbacks can cost companies potentially millions of dollars.

Project Darkstar is an open-source solution initiated by Jeff Kesselman at Sun Microsystems to address many of the problems encountered in massively multiplayer online games. Similar to the way developers use game engines to reduce development time, Project Darkstar stands as a similar technology used to assist the development of the distributed server structure for games. Freely available, it is a distributed server system currently in development that provides the following feature set: Persistent and crash-proof transactions, server scalability, massive-scaled concurrency, a lightweight server design, and a simple, well-designed API. While not all games are designed as MMO games, a large set of games exist that make it worthwhile to pursue this line of research and development.

2.1 Persistent and Crash-Proof

Everything that takes place in the Project Darkstar system is persistent, including executing tasks. This is an important feature for many online games. MMO role-playing games (MMORPGs) feature characters that exist in persistent fantasy worlds (e.g., World of Warcraft, Star Wars Galaxies, EverQuest). Many of these games, particularly the more popular commercial variants, require a monthly fee in order to play. Should a server crash, paying subscribers could lose player-specific data that represents a large time investment. This could potentially result in a loss of players, which results in revenue losses. Project Darkstar guarantees that actions will be atomic; i.e., they either happen completely or they are terminated with no effect . These actions are persistent, meaning that even in the event of a server crash, once the system is rebooted the game world will resume where it left off, without losing precious player data. The underlying abstraction that ensures persistence is the well-known concept of transactions. The end-user is intentionally protected from having to know that transactions are executing within the Darkstar infrastructure.

2.2 Scalability

Project Darkstar provides scalability across multiple servers. Game servers store data for online worlds across several systems in order to mitigate network traffic from a single computer to a cluster of machines. In order to accommodate changes in requirements in terms of player

[2] "WoW Downtime Interview at Penny Arcade ." Slashdot. 24 Jan. 2005. 11 Dec 2008 <http://games.slashdot.org/article.pl?sid=05/01/24/1855218&tid=209>.

[3] Gibson, Ellie. "Blizzard to compensate players for World of Warcraft problems." gamesindustry.biz. 09 Aug. 2005. <http://www.gamesindustry.biz/articles/blizzard-to-compensate-players-for-world-of-warcraft-problems>.

capacity, these clusters must have the ability to expand or contract accordingly. Otherwise, the system is prone to overloading, in the case of too many players, or underutilization, in the case of too few. The scalable features of Project Darkstar attempt to avoid issues such as this to help create a more robust network environment.

2.3 Massively Concurrent

In massively-scaled online games, a game server attempts to host a large number of players in a single game world. For this approach to be feasible, the server system must handle data contention on an equally massive scale. A core feature of the Project Darkstar API is the concept of managed reference objects – Java objects that are persistent (to a database back-end storage) that can be read and written to without worrying (in general) about dealing with concurrency overhead.

2.4 Lightweight

The layout of a traditional server system requires that the server takes care of most data processing and that the client is light and distributable. However, in the gaming industry, the model is exactly the opposite. Gaming computers are incredibly powerful systems designed for handling extra processing necessary for real-time interaction. In addition, since the servers have to service a massive number of clients, extra processing power is a scarce and valuable resource. Project Darkstar aims to minimize the required computations necessary for MMO interaction so the client gaming applications experience low latency.

2.5 Simplicity in API

Finally, while the feature set of Project Darkstar is important, it ultimately is an API designed to be used by other programmers to implement their games; as such, the API is designed to be straightforward and easy to work with. The simplicity in which the API is constructed allows for the impressive server elements of Project Darkstar to be utilized effectively and efficiently, giving developers the power to create an incredibly complex distributed system with ease that may not have been practical otherwise.

3 The Problem

Sun and the Project Darkstar community have developed several case studies that demonstrate the effectiveness of Project Darkstar, such as *Hack* and *Bunny Hunters*[4]. There is a missed opportunity, however, in that there is no initial user experience that appeals to the larger demographic of individuals that might be interested in Project Darkstar. Unfortunately, the

[4] "Project Darkstar Community - Projects". Sun Microsystems. 07 December, 2008 <http://www.projectdarkstar.com/external/projects.html>.

existing demonstration applications seem to be appealing primarily to the original demographic for which the Project Darkstar team was aiming, namely, system administrators[5]. This group was assumed to be mostly UNIX-based developers who are familiar not only with using open source technologies but with using the command line for executing applications. The initial growth of Massively Multiplayer Online games was supported by complex and powerful server infrastructures to support the games, which were maintained by system administrators themselves.

However, as the popularity of MMO games has increased, the project has attracted a different demographic, that of independent game developers (also known as Indie developers). The popularity of MMO games even resulted in IMGDC, the Independent MMO Game Developer's Conference.[6] Indie developers tend to look for pre-built technologies to facilitate game development, as they are generally composed of small teams, most without external funding or publisher support. As a result, Project Darkstar has attracted a following of indie game developers.

The Indie game development demographic is less familiar with UNIX systems, preferring Windows due to native DirectX support and other technologies available for Windows game development, such as the XNA Game Development Platform[7]. Project Darkstar's initial user experience was geared toward the original demographic. As a result, the larger Indie developer demographic was less attracted to actually using it[8].

The Project Darkstar installation process was sparsely documented, and offered few instructions to guide users. Ideally, a user would unzip the binary downloadable, extract the Project Darkstar main binary downloadable, and then extract the remaining three libraries to the external library folder. The user would then go into the runtime folders for the provided tutorials, and create a folder named "dsdb." The user would then create the SGS_HOME environment variable. The creation of the "dsdb" folder was absent from the included readme.txt file.

Included with the Project Darkstar binary were two shell scripts. One was for UNIX-based systems, and the other was for Windows systems. The Windows script did not run properly due to incorrectly-placed quotation marks, which required manual editing in order to restore functionality. Assuming the user completed all the steps documented in the instructional document, running the script would result in a console Java application that throws exceptions, then exiting. Most often, the exceptions were a result of the "dsdb" folder not being created.

---

[5] Personal Conversation with Seth Proctor. 27 August, 2008.

[6] "Independent MMO Game Developer's Conference 3.0." IMGDC. 7 Dec 2008 <http://www.imgdc.com/>.

[7] "XNA Creator's Club Online." 7 Dec 2008 <http://creators.xna.com/en-US/>.

[8] Personal Conversation with Seth Proctor. 27 August, 2008.

Once a user figured out where to make folders and where to place libraries, the various tutorials ran with little trouble.

The README file was very lacking in general information and reasons why steps were performed. Most files of the binary download and installation instructions aren't necessary to Project Darkstar development: the batch script was simply for running tutorial code, and the SGS_HOME environment variable was only used by the batch script, and the tutorials are wholly independent. None of the instructions pertained to getting Project Darkstar itself working; their actual purpose was to run pre-made source file tutorials.

Project Darkstar also suffered from lack of marketability. There were no impressive projects completed with Project Darkstar. Furthermore, while Project Darkstar performed well on its unit testing, it suffered from having no benchmarks and there was no evidence that it performed anything it claimed to[9].

Despite its small API, Project Darkstar's tutorials offered very little in the way of interactive learning. The pre-packaged server tutorials consisted of one PDF document that described what the tutorials did and instructed a user to read the provided code as a way of learning. In addition, the tutorials offered very little in instructing a user how to get up and running quickly in any major IDEs, including NetBeans and Eclipse.

3.1 Proposed Solution

The original solution, presented by Jim Waldo, to the aforementioned problems was to produce a fifteen-minute out-of-the-box-experience for Project Darkstar. Mr. Waldo based his out-of-the-box-experience's design on the MySQL fifteen-minute out-of-the-box-experience, which he considered very effective in attracting users. Mr. Waldo's proposal presented several issues of its own.

Mr. Waldo's basis for comparison was a complete application with which a user could interact. While developing with a SQL database is extremely common (especially in Web-based applications), a database server with a querying language allows for immediate user interaction. Project Darkstar does not allow for immediate user interaction. In order to do anything with Project Darkstar, a user must first develop an application using it, which requires a time investment much longer than fifteen minutes unless the application is trivial.

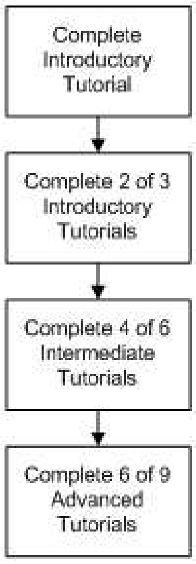
3.2 Revised Solution

The project team agreed that a fifteen-minute out-of-the-box-experience was not a feasible endeavor. However, the project team concluded that a general-purpose out-of-the-box-experience was feasible. The intended goal was for an out-of-the-box experience that would take

[9] Personal Conversation with Jim Waldo. 25 August, 2008.

no more than a single evening to complete, and as such, the team set a three hour upper-bound on the out-of-the-box-experience.

In order to cut out unnecessary time wasting, the team scanned the Project Darkstar community forums in order to find common problems. Problems were identified, including getting a project set up in the major Java Integrated Development Environments (specifically, Eclipse and NetBeans)[10], as well as simply getting Project Darkstar running[11]. The project team decided that a cross-platform installer would be an attainable first step in reducing the amount of time a user needs to spend in order to start working with Project Darkstar.


Complete Introductory Tutorial
Complete 2 of 3 Introductory Tutorials
Complete 4 of 6 Intermediate Tutorials
Complete 6 of 9 Advanced Tutorials
Figure 1: Hierarchical tutorial design

In addition, in order to reaffirm to Project Darkstar is a viable server technology for their projects, the team decided including a small sample game that demonstrates Project Darkstar's features in an aesthetically pleasing and entertaining way. The game needed to be playable by a single player and needed no learning curve.[12]

Finally, in order to familiarize developers with Project Darkstar such that work is not impeded on their personal projects, the initial package needed to include a set of comprehensive, interactive tutorials that explained as many of the Project Darkstar features and nuances as possible. In addition, the tutorials need to include instructions on how to set up a project within the major Java IDEs.

3.3 Advised Solution

In order to devise a solution both Sun Microsystems and the Project Darkstar team would find acceptable, the project team discussed the tutorials and sample game with the project advisor, George Heineman, at length. Several ideas were discussed.

The first idea discussed was the concept of both a tiered, customizable game and tutorial experience. A user would start at a single tutorial. The user would then progress to the first tier and complete a certain number of the tutorials there, although which would be left up to the user. Once the user completed the prerequisite number of tutorials, the user would move to the next

[10] "Project Darkstar Community - Running a server from within NetBeans." 7 Dec 2008 <http://www.projectdarkstar.com/component/option,com_smf/Itemid,120/topic,634.

[11] "Project Darkstar Community - Can't install Project Darkstar?." 7 Dec 2008 <http://www.projectdarkstar.com/component/option,com_smf/Itemid,120/topic,714.0>.

[12] Personal Conversation with Jim Waldo. 25 August, 2008.

tier (which had more tutorials and required a larger number of completed tutorials) and repeat the process, which continued until the user completed the final tier. See Figure 1 for an example.

This set of tutorials was accompanied with a game idea that each subsystem of the game was a modular component that was entirely stand-alone, but the knowledge needed to build the components would be found in previous tutorials. The major issue that arose from this course of action is that no game is truly modular. Every system must at least depend upon a system for data on the server, and systems for communication, user input, and graphical rendering on the client.

This method of tutorials was discovered to be infeasible due to the lack of a guarantee that a user would learn all that is needed to complete tutorials of later tiers. This produced a situation where the tutorials could not build off each other, which defeats the purpose of having tiered tutorials in the first place. After discussing the ramifications of this complication with Professor Heineman, the project team set out to refine the concept.

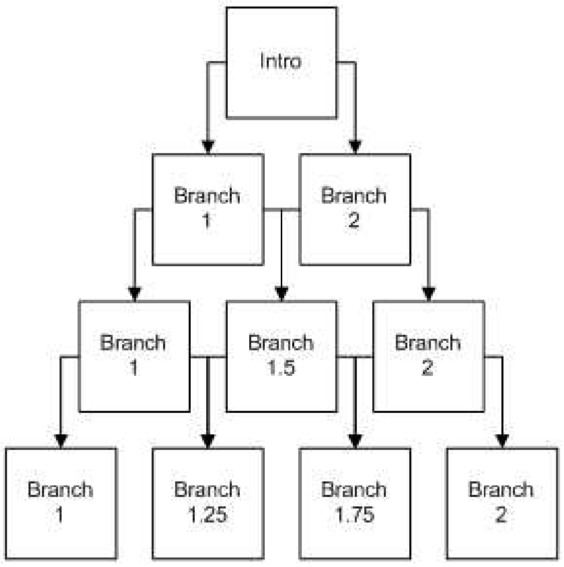


Figure 2: Branching game dependency system

Professor Heineman suggested next a branching set of tutorials, where each tutorial has a set of prerequisite tutorials, any of which could be completed in order to qualify a user for future tutorials. In such a case, a user is not limited by tiers, but would progress along branches of information that was inter-related. The suggestion changed the focus from a breath-first tutorial set to a depth-first tutorial set. See Figure 2 for an example branch set.

The associated game concept with this tutorial set was a game where each component had a clear set of dependencies, with the component depended upon by the second would be completed in both prior tutorials. In the example provided in Figure 2, each box represents a tutorial that implements two components, with horizontally-adjacent boxes each having a single built component in common.

Professor Heineman also suggested the implementation of a system such that a user would not be required to write any code. A user would simply run a program, select a tutorial and the program would scan through the source code and uncomment commented blocks of code. The project team did not implement this concept, as tutorials with no user involvement do not teach anything to the user.

This method as well proved to be infeasible given the project team's available resources. While proper transmission of information is guaranteed for subsequent tutorials, several problems arose. The first problem that arose is the fact that one could make no assumptions about tutorials from more than a single tier prior to the current one. For example, in Figure 2, a user could start at the Intro node, move to Branch 1, then Branch 1.5, and finally Branch 1.75. Given the necessary structure of information to produce such a sequence, it would not matter that the final tutorial in the branch is closer to Branch 2, as the user would have never seen Branch 2. The second problem is that there would be a large amount of duplicated information between tutorials. In order to complete Branch 1.5, information it needs would need to be provided by both Branch 1 and Branch 2, which implies that it should be in the Intro. However, such a progression would imply that Branch 1.5's prerequisite was not Branch 1 or Branch 2, but rather just the Intro, which would collapse the structure of the tutorials.

3.3.1 Refinement of Advised Solution

The project team, upon condensing the concepts presented by Professor Heineman, decided that the simpler solution was the preferred one. The team opted for a single line of tutorials that follow a linear progression, with optional tutorials to include instructions for something a user would not explicitly need, but might be beneficial to know. The Project Darkstar API is far too simple to produce a set of tutorials as extensive as proposed by Professor Heineman, and would very quickly become general game programming guides. The original proposed concepts were on the scale of Dark Age of Camelot, which had a several-year production cycle and a staff of twenty-five individuals[13].

In addition, the team decided that the tutorials should not produce a production-quality game as their capstone, for several reasons. The primary reason is that the project team simply would not have the time and resources to produce a professional, production-quality game. Secondly, the project team decided a tutorial that encompasses a fully featured game would require far too many media resources and far too much client-side game code, which would detract from the quickly learning the Project Darkstar API and completing something meaningful in an evening.

The project team concluded that the tutorials would instead produce a package for a chat program, which would demonstrate all of the features of Project Darkstar in a quick and concise manner. The team also decided that a very small and simple game that uses Project Darkstar would be ideal, but only to show off what Project Darkstar can do.

[13] "Gamasutra - Features - Postmortem: Mythic's Dark Age of Camelot." Gamasutra. 7 Dec 2008 <http://www.gamasutra.com/features/20020213/firor_01.htm>.

4 Deployment

To describe the requirements and goals of the installer developed for this project, the project team briefly outlines the original means by which interested end-users retrieved and installed Project Darkstar.

4.1 Original Method of Deployment

When the project team first began work on the out-of-the-box experience, they evaluated the original method of deployment used by Project Darkstar. Initially, Project Darkstar was installed by having the end-user unzip into a desired location a Text Archive (TAR) file that contained the pre-compiled libraries used to run Project Darkstar. The user then had to set certain environment (or shell) variables that Project Darkstar relied upon, such as `SGSHOME` which had to point to the directory that the files were unzipped to. Once these environment variables were set, the user could launch the server using a Windows batch file or a UNIX shell script. In each case, the user had to supply command line arguments pointing to both the `CLASSPATH` of the server implementation and to a properties file that configured the server.

This process was not detailed in the installation guide for Project Darkstar, and thus a need for a tool to manage the installation process. The primary goal of the new deployment strategy was to automate the process as much as possible, whether by developing an installation tool or acquiring for use such a tool (whether freely or at cost). Additionally, some of the problems with the existing deployment were the current legal restrictions imposed by Project Darkstar's use of the freely available Berkley Database System, known as Berkley DB. Specifically, the Berkley DB license did not permit the reorganization of the files included in the Berkley DB distribution – it had to be deployed in its original distribution TAR as-is.

4.2 Characteristics of Installers

The common goal of all installers is to alter the user's file system and settings such that the target product is able to function without any alterations to the system directly by the user. The changes an installer could make to a user's system include: changes to the file system (which can include the addition of files, folders, shortcut, and links), changes to system environment variables (to add a program's installation path to the main path variable, for example, although the application could make use of several custom environment variables), and – in the case of Windows systems – changes to the registry.

The most common type of installer for Windows-based systems is a wizard installer. A wizard installer is a program that guides a user through a series of steps, with each step representing some sort of request for information from the user. Typical steps include selecting an installation directory, agreeing to a terms-of-service agreement or an end-user license agreement, and configuring the installation. This type of installer is also the most familiar to the Independent game developer demographic.

Another common form of installer is the unattended command-line installer. This type of installer is common among systems where a command-line shell is the norm. In this type of installation, any step that requires user confirmation is assumed to have it, and all the information the user needs to provide is either provided as a parameter, or the default value is inserted. This type of installation process would appeal to the system administrator demographic.

4.3 Requirements

The requirements for the project changed as the project team worked to complete the application. Initially, it only had two requirements: properly relocate the Berkley DB files into their correct location; and provide a graphical interface and command-line interface. The team developed an early prototype that mimicked exactly the user operations required by the original Project Darkstar deployment. While the functionality was correct and its design was simple, the installer was fragile (since it only worked for the exact set of files in the original installation) and it did not provide for the "one-click install" that Jim Waldo was looking for.

Eventually, the initial version of the project was scrapped, as the requirements changed drastically, once the project team along with Jim and George realized that the initial requirements wouldn't work for the final installer. The new requirements demanded robustness in the form of easily maintainable installation paths, as well as system specific variables so that only the proper files are installed depending on the operating system and CPU architecture that the installer is run on.

4.4 Pre-Existing Solutions

Rather than writing an installer, which would take at least some time, the project team evaluated pre-existing installer creators and frameworks. The project team first reviewed InstallShield™[14], a popular installer generator. The biggest issue with InstallShield™ is that it is proprietary software and required an up-front investment. The least expensive version of InstallShield™ detailed on the shopping page on Acresso's website as InstallShield 2009 Express (Windows) + InstallShield Express (Windows) Maintenance – Silver is priced at $949.00[15]. Additionally, since Project Darkstar is an open source project, the team required that only open-source available installer frameworks would be used. Another disadvantage of InstallShield™ is that it only generates installers for Microsoft Windows.[16]

[14] "InstallShield - MSI Windows Installer and InstallScript Installation Tool - Acresso". Acresso Software. 15 Dec 2008 <http://www.acresso.com/products/is/installshield-overview.htm>.

[15] "Acresso Software eShop - Buy Acresso Software Products". Acresso Software. 15 Dec 2008 <http://shop.acresso.com/product/fullproducts.asp#q>.

[16] "InstallShield Features - MSI Software Installation Tools - Acresso". Acresso Software. 15 Dec 2008 <http://www.acresso.com/products/is/installshield-features.htm>.

The project team decided to take a look at some open-source solutions, such as IzPack[17] and InstallJammer[18]. A major problem with the open-source tools were that they did not deliver the level of customizability and the features needed for the installer until they were actually used. Rather than settling with the lack of robustness of the open-source solutions, the team instead decided to build an installer.

4.5 Installer Features

The installer the team developed is simple to use, meaning that the graphical interface resembles ones that users would be accustomed to; the command-line installer also operates as expected. Importantly, both of these versions had to share the same back-end functionality, so that the execution of both the GUI and command-line versions of the installers would behave similarly. The Installer is designed to support a generic installation (within reason) which provides the required flexibility should the installation artifacts change in the future. It also supports a set of customizability options.

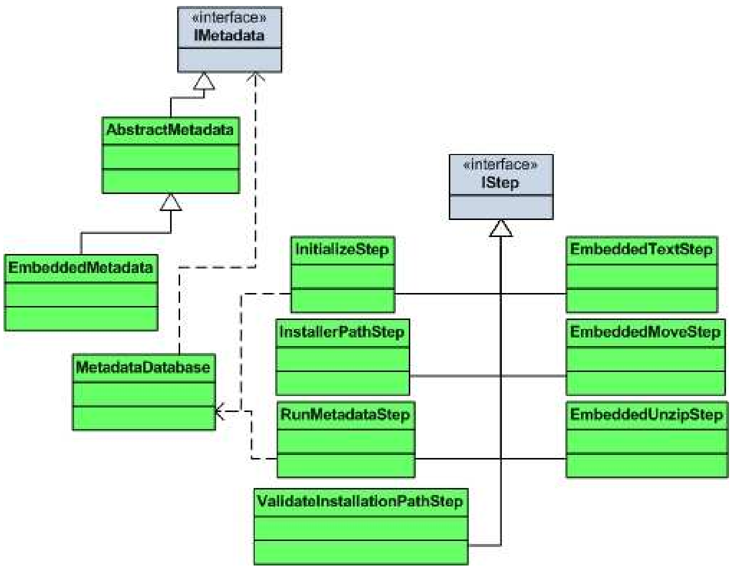


Figure 3: Installer Back-end UML Diagram

4.5.1 Overall Design

The project team designed the installer to allow the user to navigate backwards to earlier steps of the installation process, to review past decisions, for example. The team decomposed the installation process into *steps* – sequences of actions involving processing user input.

Each step implemented the IStep interface, as shown in Figure 3. In this way, the project team easily implemented the command-line version of the installer and GUI version of the installer using the same backend, using the Model-View-Controller[19] as the basis for the installer's design. The new design is generic so it can be used for any project, and it is flexible enough so that if someone wanted to make enhancements to the various steps, they

[17] Ponge , Julien. "IzPack - Package once. Deploy everywhere.". 15 Dec 2008 <http://izpack.org/>.

[18] "InstallJammer - A free, open source, multiplatform installer - Home". 15 Dec 2008 <http://www.installjammer.com/>.

[19] "Model-View-Controller Pattern". eNode. 15 Dec 2008 <http://www.enode.com/x/markup/tutorial/mvc.html>.

would propagate to both the graphical and command-line versions of the installer. The project team also devised a metadata to model the full information about the installation process.

4.5.2 Customizability

The metadata information for the installation process contains all information about the tasks to be accomplished by the installer. It determines the final location of all files put onto the file system during the installation process. In addition, it uses pattern matching with custom variables to maintain the directory structure of both embedded artifacts and the installation directories. A special type of settable variable was the system-specific variable, which would be set to a certain variable depending on the operating system and architecture that the installer was running on. This was useful in the case of system specific binaries that needed to be installed to a certain directory. The installer would automatically determine which variable to use and would replace all instances of the system-specific variable with the proper one at runtime.

See Appendix A for the final version of the installer metadata.

4.5.3 Additional Notes

The installer took the project team a week and a half to complete, during which more robustness was expected. In the end, it was finished and it worked, while being flexible, maintainable, and robust. The project team added more features, such as metadata variables for commonly used paths and system-specific variables for installing only the proper binaries depending on the target system.

See Appendix B for an early version of the installer metadata, for comparison purposes to the final version.

See Appendix C for the final version of the installer system-specific and metadata variables.

5 Sample Game

A sample game was a necessary part of the out-of-the-box experience. Assuming the user is already interested in Project Darkstar, given the completed installation process, the sample game would provide the user with a demonstration of how Project Darkstar facilitates rapid development. There is a game written with Project Darkstar called Hack[20]. However, the requirements for the sample game specified that it had to be graphically detailed enough to impress a user, and Hack specifically uses eight-bit graphics. In addition, the game had to be small enough that the user could make noticeable changes to it.

[20] "darkstar-hack: Hack - A Project Darkstar Game." 7 Dec 2008 <https://darkstar-hack.dev.java.net/>.

5.1 Pre-Game Research

The most pressing issue facing the team was whether to select an existing Darkstar application or develop one from scratch for the out of the box experience. The team knew that something flashy that the team could use to show off what Project Darkstar could do was necessary, and the team agreed that a sample game was the best method to accomplish this. The issue here was determining what was feasible within the 7-week timeframe while still fulfilling the requirements of a usable and effective out-of-box experience.

The first major topic under discussion was the dimensionality of the game itself; that is, whether or not it should be 2D or 3D. Each had its own pros and cons. For 2D, game development was much simpler. In order to create a finalized game, a lot of content creation is involved. A 2D game meant that the visual content was limited to textures and flat image files. While the simplicity of a 2D game was alluring, it lacks the visual appeal of standard 3-dimensional games currently on the market. A 3D game has the advantage of appealing more to the current MMO community which is accustomed to games that involve exploration of a 3-dimensional game world. In this respect, it was decided that 3D was the way the project team wanted to approach the team’s demo game.

The team knew at this point that the decision to create a 3D game meant that not only was content creation more difficult, but so was the technicalities of dealing with real-time interactions. In short, the team needed an engine that could do the dirty work for us. With Java as the team’s development platform, options were limited, due to most engines being written in C++ for efficiency. During this time, the Project Wonderland team had been mentioning that they were planning on switching their current rendering system over to one called jMonkeyEngine (jME). One of the more popular 3D game engines available for Java it boasted high performance as well as being an open-source project. In the end, it was decided that if the team was indeed going to create a full 3D game, jMonkeyEngine would be the way to do it.

5.1.1 jMonkeyEngine

The feasibility of jME was unknown before use, and therefore additional research into this topic was carried out. The team investigated the capabilities of jME while the installer was being written, hoping to reach a conclusion within a week. Towards its advantage, the jME web site contains several tutorials available with sample code provided. The team evaluated these tutorials and created a quick demo application as a test case. The process of going through all the tutorials took longer than expected and once this was complete, the project team decided that the demo would be a simple World of Warcraft-style camera, something that should have been a trivial task: it was not. The process of creating a short demo revealed some major problems in working with jME.

First of all, there was very little versioning. Only two versions of jME were available to download; 1.0 and 2.0. There was no intermediate versioning. Any updates to the engine were committed to the latest version meaning that there was no true consistent version of the software available. In addition, jME lacked good design practices, in the team's opinion. The entire structure consisted of a massive tree of derived classes with little organization other than package relation. In order to use or extend any class, you usually had to read down through the various parent classes in order to determine the functionality. This structure was not consistent; seemingly related input classes would be structured in two entirely different manners. In general, the engine was not designed to include an API, meaning that functionality was derived from the use and knowledge of the entire framework as a whole, making it difficult to work with.

In the end, it was clear that jME was causing more trouble than it was helping, and that it would create a great deal of work in order to create the game the the team felt was necessary to generate appeal for new users. In this sense, the use of jME was not advised, and it was decided that creating something decent was out of the scope of a 7-week project.

5.2 Idea 1: Extensible Module-Game

As described previously, one of the ideas proposed by Professor Heineman was a modular component-based game. There were several designs and possible implementations discussed. For example, any module would have both the ability to read information from it and to allow other modules to cause events within the module. Using this assumption, several of World of Warcraft's systems were decomposed into modules in an attempt to see if it would work. An example is shown Figure 4.

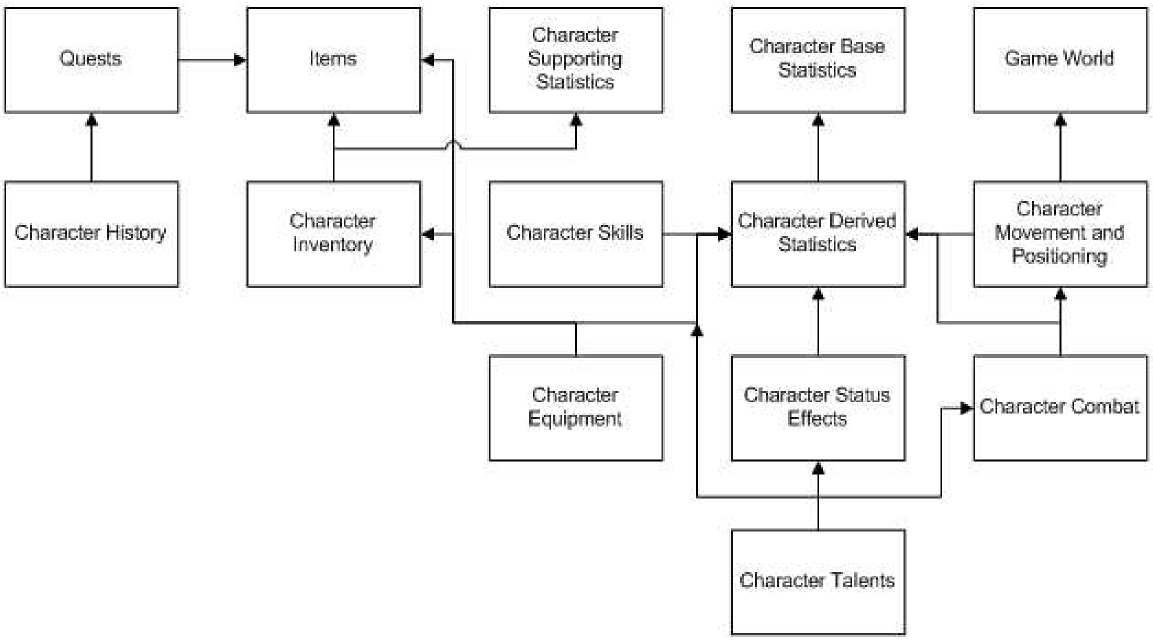


Figure 4: Portions of World of Warcraft decomposed into modules.

5.2.1 Problem

The only problem that arose with a modular-based game idea is that such an idea is far too ambitious and would require too much manpower to produce in the allotted time. The project team has agreed that should the Project Darkstar team have additional time, implementing a sample game a similar manner would be a worthwhile endeavor. A modular game system of this sort also works well with Project Darkstar's architecture, as related pieces of data are kept separate, which would minimize contention.

5.3 Multiplayer Space Shooter

The second idea the project team came up with was an idea for a very simple multiplayer space shooter game. Based on a JMonkey tutorial, players would attempt to shoot each other while moving their avatars around in a spherically bounded game world.

5.3.1 Problem

While this idea was sufficiently small and simple to program, it had a similar issues to Hack; it simply was not impressive enough an idea to impress a user, both graphically and in terms of game-play. In addition, the vast majority of the client code was tied to JMonkey, which meant a user would have to learn the JMonkey engine to do anything with the game beyond playing it. The game also did not lend itself well to single-player game play, which is a necessary part of the sample game due to the fact that there is no public server on which the game could be run.

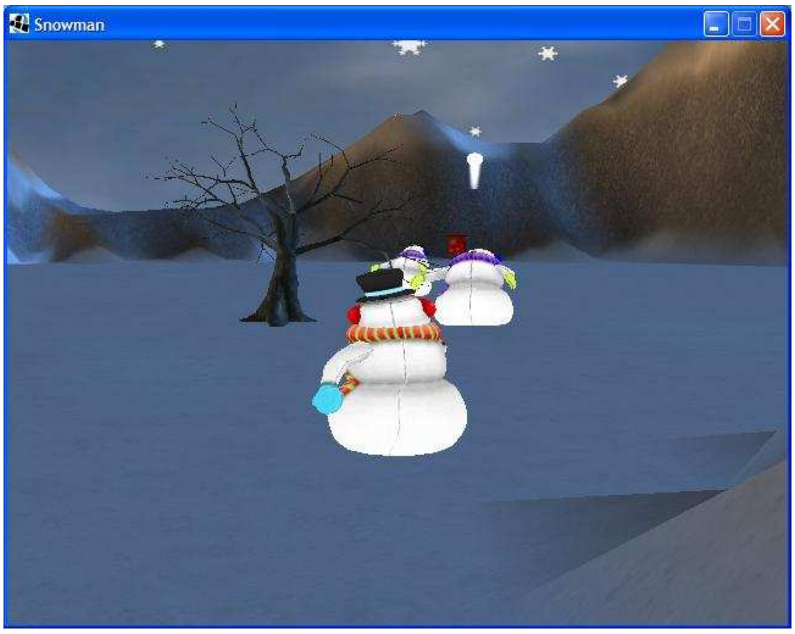


Figure 5: Project Snowman during game-play.

5.4 Solution

For the sample game in the out-of-the-box experience, the project team decided to use Project Snowman[21]. This decision was made since the project team did not have a lot of time to design and develop a quality game in the short period of time that they had left, as Project Snowman

[21] "project-snowman: Project Snowman". Sun Microsystems. 15 Dec 2008 <https://project-snowman.dev.java.net/>.

was an already working game with Project Darkstar and that it looked acceptably pleasing as it was a complete 3D game.

6 Project Snowman

Project Snowman is a third-person shooter where human and AI players are pitted against each other in a snowy tundra wasteland with the goal of each player being to capture the opposing team's flag and return it to that player's base.[22] The game was developed using Project Darkstar as its server technology, and has been stress tested with up to 6,000 clients connected to a single Project Snowman server.[23] Figure 5 depicts an example of game play in Project Snowman

6.1 Game-play

Project Snowman's game play consists of capture-the-flag style objectives with shooter elements in which two teams each consisting of human and/or AI controlled players attempt to steal the opposing team's flag and capture it at their own base. There are a variety of game features that players will utilize and become accustomed to, such as how the rules of the game are defined, the view of the world, and the snowman's capabilities.

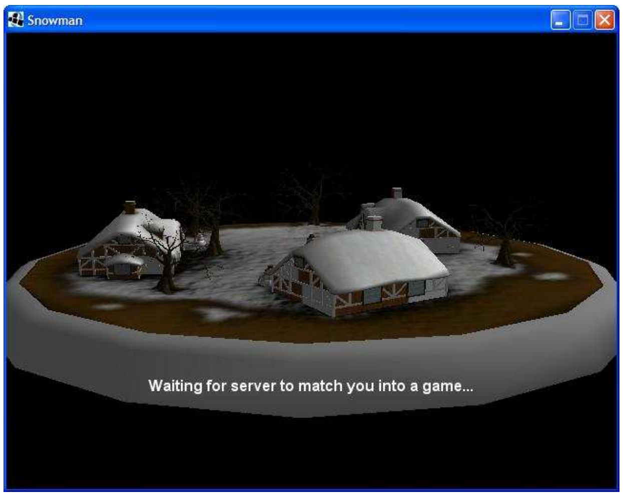


Figure 6: Match-making Screen

In Project Snowman, the goal of the game for each team is to have one of their players capture the other team's flag and return it to their base successfully. If a team succeeds at doing this, then that team wins the round, and a new game is started. Game play usually consists of teams either moving to the enemy flag to retrieve it in tandem, as seen in Figure 5. Each team member is responsible for protecting each other from the volley of snowballs from the other team by returning a volley of their own, or by being sneaky and having an individual player avoid the rest of the calamity and steal the enemy flag.

Each player also has a view of the world, and many of the things that each player sees while playing the Project Snowman is similar. Every player, while starting the game, will see an intro

[22] "TWiki . Games . ProjectSnowman". Sun Microsystems. 15 Dec 2008 <http://wiki.java.net/bin/view/Games/ProjectSnowman>.

[23] Personal Conversation with Owen Kellet. 30 August, 2008.

screen, as seen in Figure 6, of a rotating circular level with text that says, “FINDING A GAME.” Once that player has successfully joined a game, then their view is immediately switched to a game play view, as seen in Figure 7, in which they are viewing the game world behind a snowman in a 3rd person perspective. In the game, players are portrayed as snowmen in an icy tundra that is surrounded by some hills, dead trees, a few houses, and two flag points, one for each team. Each snowman starts off with the same traits, and the only difference between the snowmen on each team is the color of their scarves – this is set this way so that players can easily know which players are on their team and which players are their enemies. Players will see if other players are moving, attacking, running with the flag, or performing any other game related action. When a game ends, the player is returned to the matchmaking screen, and the game start process begins again.

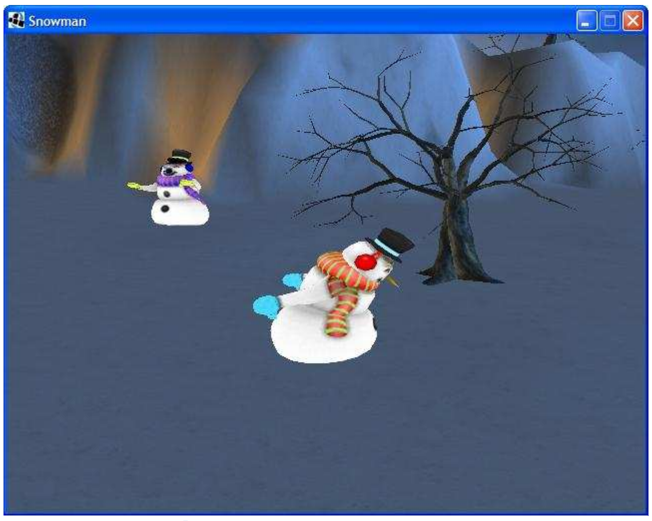


Figure 7: Game-Play View

6.2 Purpose

Using Project Snowman for this project had a clear purpose – the project team needed to answer the question “What does the user do first after running the installer?” Ideally, this was going to be a game. Preferably, it was going to be a good looking game. Project Snowman fit the bill, as it’s a good looking 3D game with fully developed client-server interaction using Project Darkstar, and was being actively maintained by Owen Kellet and Keith Thompson of the Project Darkstar team, which was important to the project team since the example game had to be in a stable state, interesting, and enjoyable. The other option was “Hack”, a rogue-like dungeon crawler, but the game is not stable, enjoyable, or good looking, living up to its namesake of being hacked together.[24] As a simplistic, 2D dungeon-crawler, it did not meet the standards set by the project team for a game that would highlight Project Darkstar’s potential as a large-scale massively multiplayer server. The limitations of the game did not convey the power and scalability of Project Darkstar that a game with higher production value would convey to a potential developer.

[24] "darkstar-hack: Hack - A Project Darkstar Game ". Sun Microsystems. 15 Dec 2008 <https://darkstar-hack.dev.java.net/>.

6.3 Background

Project Snowman is a game devised by the Project Darkstar team to showcase the Project Darkstar server technology.[25] Jeff Kesselman did create a game before it called Bunny Hunters; however, it was a proof-of-concept to show that the Project Darkstar technology could run a game and not something that highlighted the scalability of Project Darkstar.[26] Bunny Hunters has a low player max, while Project Snowman can support a large number of players. Project Snowman was originally intended to be a massively-multiplayer game, supporting non-player characters and an expansive world.[27] It is now a non-massively multiplayer game in the vein of Quake or Counter-Strike, as the game instances are not persistent.[28] Primarily the following people developed the game: Keith Thompson, Yi Wang, and Owen Kellet.

6.4 Development Cycle

The game was created in July of 2008, and was to be ready to showcase for the Austin Game Developer conference two months after that. The art assets were made in this period of time by an unspecified artist contracted by Sun, but it is not known when the art assets were put in the game.[29] Recently, the game has been developed by Owen Kellet and Keith Thompson, who were able to implement the client-server connection within two weeks – an impressive development cycle for any team, considering competitive 3D games have development cycles counted in years rather than weeks[30]. Right now, Project Snowman is receiving incremental updates in the form of dependency removal from NetBeans and Ant. Owen Kellet is actively maintaining it.

6.5 Platform-Independent Launcher

Launching Project Snowman was still going to be a problem. It required several VM settings to be set on the command-line and also required the arguments that it itself used. Rather than giving the user instructions with command-line options to run, the project team created a command-line argument running application.

---

[25] Kellet, Owen. "Project Snowman: Lessons Learned". 15 Dec 2008 <http://owenkellett.com/2008/09/23/project-snowman-lessons-learned/>.

[26] "bunny-hunters: Project Home Page ". Sun Microsystems. 15 Dec 2008 <https://bunny-hunters.dev.java.net/>.

[27] Project Snowman Code. 2 September, 2008.

[28] "TWiki . Games . ProjectSnowman". Sun Microsystems. 15 Dec 2008 <http://wiki.java.net/bin/view/Games/ProjectSnowman>.

[29] Personal Conversation with Keith Thompson. 4 September, 2008.

[30] "Gamasutra - Features - Postmortem: Mythic's Dark Age of Camelot." Gamasutra. 7 Dec 2008 <http://www.gamasutra.com/features/20020213/firor_01.htm>.

6.5.1 Requirements

There were a few things required of the launcher. First of all, it had to be platform independent such that it would run on all platforms that Project Darkstar ran. This was the most important requirement for the launcher. For Project Darkstar, this meant choosing the correct Berkley DB binary to use depending on the operating system and architecture that the launcher was run on. For Project Snowman, this meant choosing the appropriate LWJGL binaries. As well, it had to be incredibly easy to use – a "one-click launch" or a simple launch via command-line was important to Jim Waldo.[31]

6.5.2 Design

The launcher retrieved the command to run from a .properties file, which is the Java standard for program settings.[32] The .properties file contained settable variables and system-specific variables that were set depending on the system that the launcher was run on. The libraries created to perform this duty were originally in the installer but were later removed, put into the platform-independent launcher, and eventually copied back over to the installer as the requirements of the project changed.

6.5.3 Purpose

The original purpose of the launcher originally was to run Project Darkstar without shell or batch scripts, since the ones supplied with Project Darkstar contained bugs such as incorrect path settings to binary files and proved to be a mess to maintain, causing several end users to run into errors.[33][34][35] This was put on hold for a large stretch of the project team's time at Sun. Sun later revived the project, as it would be useful for launching the Project Snowman client and server. The project team used it with Project Snowman was to remove the dependency on NetBeans and Ant – these were dependencies that the project team did not want their OOTBE to have, as requiring multiple installations to get Project Darkstar up and running would have hindered the quality of the out-of-the-box experience.

[31] Personal Conversation with Jim Waldo. 30 August, 2008.

[32] "Properties (Java Platform SE 6) ". Sun Microsystems. 15 Dec 2008 <http://java.sun.com/javase/6/docs/api/java/util/Properties.html>.

[33] "Server Installation not working on WindowsXP 32 bit". Sun Microsystems. 15 Dec 2008 <http://www.projectdarkstar.com/component/option,com_smf/Itemid,120/topic,720.0>.

[34] "Project Darkstar Community - Can't install Project Darkstar?". Sun Microsystems. 15 Dec 2008 <http://www.projectdarkstar.com/component/option,com_smf/Itemid,120/topic,714.0>.

[35] "Project Darkstar Community - Problems running HelloWorld example". Sun Microsystems. 15 Dec 2008 <http://www.projectdarkstar.com/component/option,com_smf/Itemid,120/topic,672.0>.

6.5.4 Result

The platform-independent launcher streamlined the process of running Project Snowman's client and server portions, and it is flexible enough to be used for running Project Darkstar itself. It fulfilled the need for a customizable, lightweight, cross-platform program launcher.

7 Tutorials

In order to provide Project Darkstar users with a method to begin work rapidly on their own development projects, the project team designed a set of tutorials to demonstrate the Project Darkstar API.

7.1 Current Server Tutorials

The original Project Darkstar server tutorials consisted of a set of six wholly unrelated tutorials, each with the intent to teach a user part of the Project Darkstar API. All six tutorials were in a single PDF document, with SwordWorld as an appendix.[36] The project team briefly summarizes these tutorials here to show their limited scope.

7.1.1 HelloWorld

A user completing the first tutorial would simply construct a Project Darkstar application to output "Hello World". Much of the tutorial was devoted to repeating what was stated in the original installation document, as well as how to run the shell script that came with the original binary download. As such, the tutorial document (a separate download) was dependent on not only Project Darkstar itself, but also a set of files separate from the code itself.

7.1.2 HelloLogger

Project Darkstar's second tutorial had very little to do with Project Darkstar itself. It simply described a basic use of the Java `Logging` package provided by the Java Software Development Kit. While this tutorial has no dependencies separate from the first tutorial, it still does not cover anything related to Project Darkstar.

7.1.3 Tasks, Managers, and HelloTimer

This tutorial's purpose is to introduce tasks and all of the managers that come with Project Darkstar's API. This tutorial, in the main driver class, provides code for a repeating task. Ignoring the improper delegation of functionality of the tutorial's code, the tutorial's sole

[36] Kesselman, Jeffrey. "Project Darkstar Server Application Tutorial." 24 April 2008 7 Dec 2008 <http://www.projectdarkstar.com/ccount/click.php?id=29>.

purpose is to describe a single method call. This tutorial could fold in elsewhere.
HelloPersistence

This tutorial's purpose is to introduce persistent storage. This tutorial is actually three miniature tutorials. The first is similar to the Hello Timer tutorial, where it instead stores the previous timestamp without using the `DataManager`. In the second tutorial, it abstracts the task to a new class, and in the third tutorial, it instructs the user to rewrite the class in order to use the `DataManager`. This tutorial provides a roundabout way to instruct the user in performing a simple task.

7.1.4 HelloUsers

This tutorial introduces the developer to user sessions, known as `ClientSessions`. It is split up into a smaller set of tutorials, each of which requests the user to rewrite the same code several times. The code ends with a server application that echoes all messages back to the client. However, the tutorial provides no client-side code to demonstrate that it works.

7.1.5 HelloChannels

This tutorial introduces the developer to the broadcasting API, known as channels. This tutorial shows the user how to make small chat program using channels. Most of the code is in the form of Java logging code, rather than actual Project Darkstar code. This tutorial bears a striking resemblance in concept to Hello Users; however, in code it bears little resemblance.

7.1.6 SwordWorld

This tutorial is supposed to act as a sort of capstone to the tutorials. Unlike the previous two tutorials, which have associated client-side code in a separate file, there is no tutorial around the construction of a client for SwordWorld, which produces a situation where a user cannot interact with the server.

7.2 Current Client Tutorials

The original Project Darkstar client tutorials consist of two tutorials, each of which describes the construction of a Project Darkstar client. The tutorials are provided in a single PDF document. Each tutorial's focus is specifically on writing the client associated with the server tutorials previously described. While the server tutorials do have associated client tutorials, they are not in a single location, and thus provide a frustrating user experience.

7.3 Problems with Current Tutorials

In the first week of the project, the team carried out these tutorials as the team worked to understand Project Darkstar. Because these tutorials provided the team its first real experience to

Project Darkstar, the team is able to evaluate them from the perspective of the end-users for which they were written originally.

The team found that the primary weakness of the tutorials is they represent separate tasks without a clear context for how these tasks interrelate. There is no notion of iterative development, and no notion of how to develop the respective programs properly. They simply provided a completed source file and a shell script for running it. Individual lines, left unexplained and undocumented, leave inexperienced users with little actual knowledge about the system.

In addition, several of the server tutorials require clients in order to demonstrate that they are functional. The client tutorials and client binaries are completely separate from the servers, which produces a disconnection in the development process.

7.3.1 Revisions to Tutorial Structure and Content

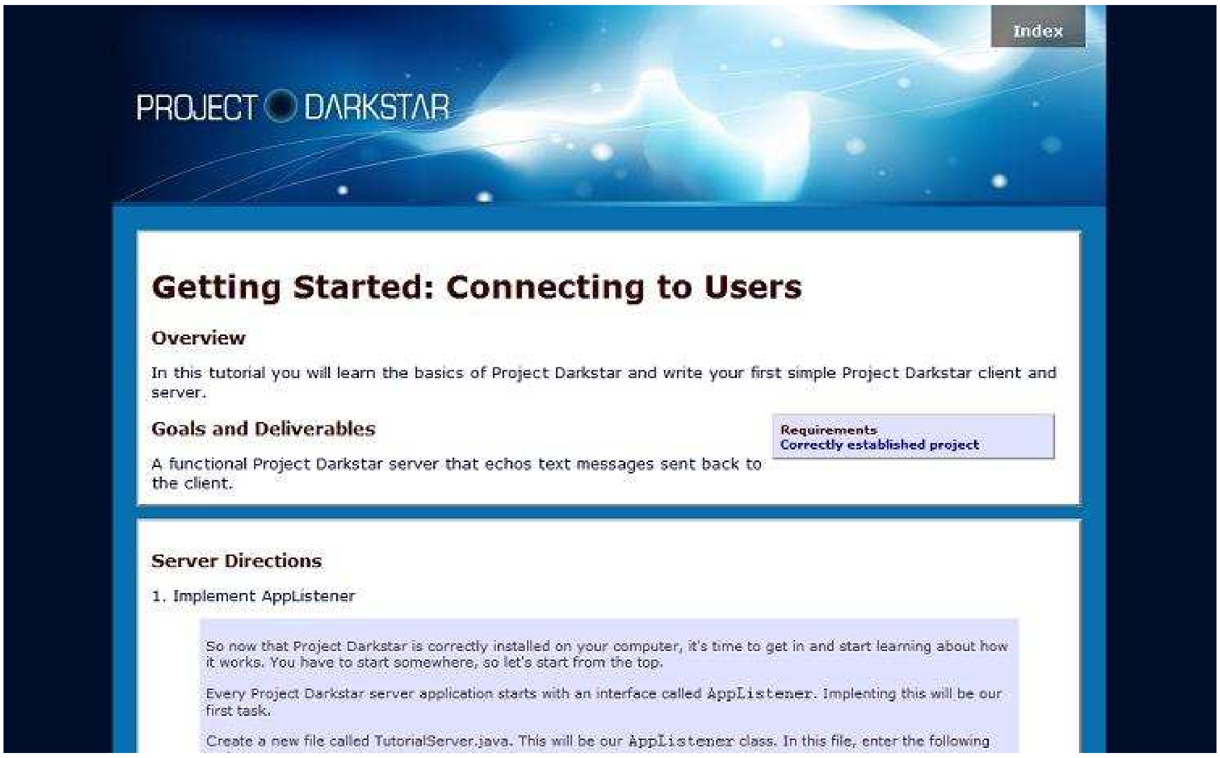
The project team decided to keep the linear progression of tutorials present in the original Project Darkstar tutorials. In addition, the project team felt the need to add several optional tutorials that instructed users in basic pieces of information not directly related to Project Darkstar, but which are otherwise extremely useful. For example, the project team felt an effective optional tutorial would instruct the user in producing usable NetBeans and Eclipse projects.

In addition, the project team felt that cutting out as much information pertaining to topics other than getting started with Project Darkstar was necessary. As such, the original Hello World tutorial and the Hello Logger tutorial were entirely cut, leaving only information directly pertaining working with Project Darkstar.

7.4 New Tutorial Outline

The basic idea with the new set of tutorials was to create a simple environment for users to learn the essentials of Project Darkstar while simultaneously creating a client/server basis for future projects. Individual tutorials also build upon one another iteratively, culminating in a complete system that demonstrates all aspects of the Project Darkstar API. Once completed, the user can then choose to partake in a challenge tutorial that walks through the development of the server system for an actual game. Specifically, this is the Snowman Game, as mentioned in Section 6. Participants are encouraged to code along with the tutorials. This has the added benefit of involving the users immediately and keeping users involved while holding their interest.

All tutorial layouts use an `HTML` format created using Cascading Style Sheets (CSS). The team does this to simplify tutorial creation for us during development as well as for Sun employees should they desire to make changes or additions after the team departs Sun. The CSS is designed to fit the aesthetic standards of the Project Darkstar website for consistency, as shown in Figure 8.



Getting Started: Connecting to Users

Overview

In this tutorial you will learn the basics of Project Darkstar and write your first simple Project Darkstar client and server.

Goals and Deliverables

A functional Project Darkstar server that echos text messages sent back to the client.

Requirements
Correctly established project

Server Directions

1. Implement AppListener

So now that Project Darkstar is correctly installed on your computer, it's time to get in and start learning about how it works. You have to start somewhere, so let's start from the top.

Every Project Darkstar server application starts with an interface called `AppListener`. Implenting this will be our first task.

Create a new file called TutorialServer.java. This will be our `AppListener` class. In this file, enter the following

Figure 8: New Project Darkstar Tutorial

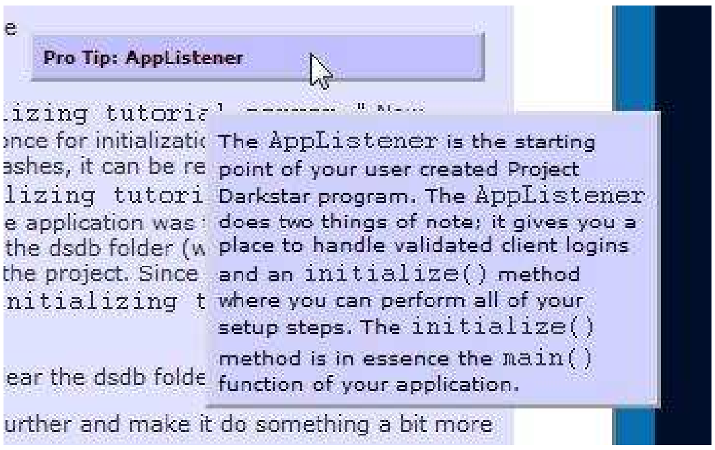
The tutorials are hosted on a web page which contains links to the various documents, including: two introductory articles defining Project Darkstar installation procedures as well as how to create a project; the four basic tutorials; and the challenge tutorial. Every page has a link that leads back to the index, and a link that leads to the main Project Darkstar website.

Each individual tutorial has three major sections. First is the overview section. This contains the title, an overview, and a description of goals and deliverables. In addition, it describes the prerequisite knowledge and skills required to complete the tutorial. These refer to previous tutorials and are represented as links for the convenience of the user. The other two sections are the client and server sections. These are mainly identical in terms of features; the only difference is the separation of client and server units for each tutorial. Both contain a list of directions and conclude with a “wrap-up” that recapitulates the content of the section. A few special properties of these sections are worth mentioning. First, all the text is written in a simple style that defers from a more formal style to better connect with users. Second, all code is formatted in its own font to differentiate between it and normal text. Large blocks of code are placed in specialized code blocks that employ syntax highlighting. Finally, major Project Darkstar API interfaces and concepts are displayed in small pop-up boxes that the team called “Pro Tips.” These pop-up boxes appear on a mouse-over event and list detailed information on their respective subjects. Figure 9 shows a sample of a Pro Tip.

7.5 Basic Tutorial Outline

The final set of basic tutorials is as follows:

7.5.1 Tutorial 0: Project Set-up


Pro Tip: AppListener

The AppListener is the starting point of your user created Project Darkstar program. The AppListener does two things of note; it gives you a place to handle validated client logins and an initialize() method where you can perform all of your setup steps. The initialize() method is in essence the main() function of your application.

Figure 9: Pro-Tip Box

Tutorial 0 is an optional set-up tutorial. It is optional because the project team provides a set of pre-made projects that allows users to bypass this step if they so choose (More about the pre-made projects are discussed in detail in section 7.7). Complete Project Darkstar applications require the construction of a development environment in order to properly run and build. Specifically, the user must first install Project Darkstar to get the required libraries and create project files if using an IDE. If no IDE is to be used, the semantics of writing the actual Java code is left to the user. Project Darkstar applications also require two additional components. First, a properties file that specifies various variables of the Project Darkstar stack must be created, during which project-specific settings are defined. Also, the user must create a folder named “dsdb” that will serve as the location for all database information. These requirements detail the necessary prerequisites to build. To run, the Java virtual machine needs a specific set of arguments. These arguments include setting the main class to be the Kernel class that is provided by Project Darkstar (All applications are a set of classes that implement the interfaces of the Project Darkstar API, i.e., applications are components run on top of the Project Darkstar stack). Additionally, the location of the properties file and Berkeley DB native libraries must be specified, as well as any project-specific command-line arguments. These steps are all detailed in full in this tutorial. The team knew that users would be using a variety of different IDE’s for this process, and the team did not want to create any limitations or dependencies on any singular IDE or method, therefore, this tutorial has three different sections: Using Eclipse, using NetBeans, and using no IDE.

7.5.2 Tutorial 1: A Simple Echo Server

This is first of the four “basic” tutorials. It starts by introducing the user to the basics of the Project Darkstar API. This starts with the explanation of the `AppListener`, `ClientSession`, and `ClientSessionListener` interfaces. This tutorial also introduces the `AppContext` static class, a class which provides an access point to a variety of manager classes that are introduced and applied in later tutorials. A small introduction to client/server architectures is also discussed. By the end of the tutorial, the user had a functional Project

Darkstar server application that allows users to connect and send text messages, which are echoed back to the client. The tutorial also provides instructions to create a client application capable of communicating with the server application. Note that the pre-made NetBeans project allows users to run both the client and server, allowing them to test the results of the tutorial immediately.

7.5.3 Adding Persistence

The persistence tutorial attempts to familiarize users with the concept of persistence in online applications. This begins with a discussion of the difference between the `ManagedObject` and `Serializable` interfaces in Project Darkstar applications; these interfaces are necessary for storing data persistently in the database and have subtle differences when writing Project Darkstar applications. The `ManagedReference` and `DataManager` objects are introduced as ways to access the data stored in these objects. These objects and interfaces form the core of persistence in Project Darkstar.

In addition to server applications and client/server architectures, users new to creating these systems need to know about effective communication between the client and server. Therefore, the team acquaints users with the concepts of operation codes and wire protocols. The project team does this once persistence is explained. At this point users will implement a simple wire protocol that they will use throughout the remainder of the tutorials.

At the conclusion of this tutorial, users will be left with an upgraded version of the client and server applications that will store that last message sent by clients. Using the new wire protocol, users can send messages that will retrieve this message, which will be echoed back to the user.

7.5.4 Allowing User Interaction

Tutorial 2 ended with a discussion of communication between client and server through the use of wire protocols. In tutorial 3, the team discusses more communication; this time within the server application itself. The echo server application written in tutorial 1 communicates messages back to the user. The server is also capable of controlling direct communication between users fairly easily; however, in online games and applications there are times when users may wish to speak to a group of people at once. An example of this would be simple online instance messaging programs. To facilitate message communication in the server, Project Darkstar employs the concept of channels. This concept in introduced to the user through the explanation of the `Channel` interface and the `ChannelManager` class. For the client, the `ClientChannel` interface is also covered. At the conclusion of this tutorial, the client and server applications are upgraded to echo messages sent from a given client to all connected users.

7.5.5 Tasks

Tutorial 4 marks the final tutorial in the first set of "basic" tutorials. Here, users are introduced to the `Task` interface and its uses. Server applications have to make large numbers of computations based on data received from connected users. Server crashes, if gone unchecked, can interrupt running processes which can ultimately lead to lost player data. To protect against this, Project Darkstar uses the concept of tasks. These tasks will remember the current state of execution in the event of a crash and, when the server is restarted, will continue from the point of interruption. Tasks are implemented through the use of the `Task` interfaces and are scheduled and controlled using the `TaskManager` object, both of which are explained in this final portion of the introductory tutorials. At the completion of this tutorial, users will have upgraded their previous client and server applications to include a periodic task that echoes the current server time to all connected users.

7.6 Challenge Tutorial

A challenge tutorial was devised that would give the user a chance to muck around in the Project Snowman code and implement various parts of its wire protocol on the Project Snowman client and Project Darkstar implementation. This tutorial was important to the project-team's OOTBE, but it was a secondary objective in comparison to the basic tutorials. Fortunately, time permitted and the project team was able to create this tutorial for the user.

7.6.1 Tutorial Set-up

The challenge tutorial came with two a pre-packaged NetBeans project – a fully complete version, which could be run as-is, and a scratch version in which the user implemented the material that the tutorial covers.

7.6.2 Tutorial Design Goals

It was important that the user not only got to implement the server and client code, but that they understood how it worked. It would have been easy to simply tell the user to implement various chunks of code and not explain it, but that would have defeated the purpose of the tutorials. Fortunately, this was kept in mind, and as a result the way in which the tutorial was written reflected the conscious decision of teaching the reader.

7.6.3 Purpose

Once the user of the tutorials has completed them, they ideally understand of how to send messages to and from a Project Darkstar server implementation and a client that uses the Project Darkstar wire-communication protocol. The challenge tutorial exists for users that want a more in-depth look at how Project Darkstar would work with a fully featured game. As such, the challenge tutorial built on what the user learned in the basic tutorials. It takes it a step further, with the user making changes and additions to the internals of the Project Snowman client and its server implementation that uses Project Darkstar to facilitate communication between them.

7.6.4 User Tasks

When the user begins the tutorial, they first create a `MatchmakerTask` to perform the matchmaking duties that are necessary for games to be created when users join the server. Once that is complete, they go to the Project Snowman client and implement logging in, using the `SimpleClient` class to create a connection with the server. They then implement on the server a way of handling clients connecting by creating `Game` instances and putting clients in those `Game` instances. Also implemented on the server is the dispatching of a game ready message, sent to each client connected to the game with a socket message. After this is implemented, the user then implements on the client a way of handling a game ready message from the server by generating a local view of the game. The user also implements the sending of a movement update message on the client, sent to the server when the user clicks on the ground. Back on the server, the user implements the handling of client position update messages, and relays those back to all clients connected to the game, including the client that sent the message. Finally, the user implements on the client a way of handling player position update messages sent from the server, to update the local position of the client.

See Appendix D for a detailed outline of the user tasks in the Challenge Tutorial.

7.6.5 Result

By the end of the tutorial, the reader of the tutorial should have a fully version of Project Snowman. The goal here was to give the user a sense of accomplishment and show them that it is not difficult to implement client/server communication using Project Darkstar.

See Appendix E for the full text of the Challenge Tutorial.

7.7 Pre-Made Projects

In addition to the tutorial documents, a set of supplementary pre-made NetBeans projects are included. This section discusses the reason for their inclusion in the final set of deliverables and the importance of these projects as an addition to the out-of-box experience.

7.7.1 The Problem

Project Darkstar applications require some prior set-up that is specific to the application. This is beyond the team’s control and cannot be automated through the use of the installer. Instead, the user has to perform this set-up. This is not a trivial task. In order to start coding with Darkstar, users must download and install Darkstar, create projects with build scripts, and create properties files. Also, in order to run the application users have to run the java virtual machine with a long list of specific but required arguments. Ideally, new users should be able to skip this step, as forcing them to do it can potentially cause setbacks if they can't adequately create the project themselves. Note that adding another level of simplicity is advantageous for the success of an out-of-box experience, as described earlier in section 1.4. Pre-made projects hide this set-up from the user and provide them with immediate access to a working programming environment once Project Darkstar is installed.

7.7.2 Issues and Controversy

In order to abstract the set-up process, the team originally chose to provide various IDE project files. Using a single IDE created a dependency on the IDE that the team would rather have avoided, if possible. The team wanted to provide project files in Eclipse and NetBeans, two popular IDE's for the Java language. The project team was able to create NetBeans projects without trouble, but Eclipse gave us an issue. The team needed multiple project files to cover all tutorials, each of which had to have a reference to the required Project Darkstar libraries. While NetBeans enables project files to use relative path names for libraries, Eclipse will only allow relative paths for projects within the workspace. This limitation meant that the team had to side with NetBeans for all the team's project files, creating an unwanted dependency. The alternative to using project files was a simple project set-up tutorial that explains this process. Note that this information will have to taught to the user eventually, as this set-up is specific to the application; thus, always hiding it from the user would create issues when users attempt to create a project from scratch on their own.

7.7.3 The Solution

In the end, it was decided to keep the pre-made project files. The main reason behind this was because that although the team could only support a single IDE, the benefits in terms of usability outweighed the disadvantages of a NetBeans dependency. The addition of the pre-made projects grants a critical improvement in the ease of which users can begin learning about Project Darkstar. Along with the project files, it was decided to also create a tutorial that explains the set-up procedure, since this will have to be discussed at someone regardless of whether or not the user chooses to use the pre-made projects. This also meant that this tutorial could be listed as optional for users who wish to skip it.

The pre-made project files and set-up tutorial together give the user an option as to how to go about the tutorials, increasing the positive aspects of the out-of-box experience by adapting to various user demands.

8 Conclusion

Much progress was made over the course of this MQP. The team concludes by discussing the final set of deliverables and an evaluation of the project as a whole. The project team then makes its final statement.

8.1 Final Deliverables

At the time of project completion, the project team developed the following set of deliverables: An installer for Project Darkstar, A platform independent launcher for the Snowman game, and an extensive set of tutorials. The final out-of-box experience uses these in a step-by-step fashion, moving from simple steps that can be performed by any user with basic computer knowledge, to advanced steps designed for implementers of game server systems. Here the team recapitulates the various deliverables and the methodologies behind the creation of each.

8.1.1 Installer

The installer program was written to solve the problem of a complicated set-up. Originally, users had to perform a large set of monotonous tasks, such as the relocation of files and folders, before any work with Project Darkstar could ensue. This influenced the effectiveness of an out-of-box experience in a negative fashion because it prevented users from immediate access to Project Darkstar and created the potential for errors to occur during the set-up process. The installer alleviates these issues. It is written in Java and therefore does not discriminate against the operating system of the user. Previous monotonous tasks could now be performed by the installer. This increases the ease of set-up by making this process faster and more efficient. The installer is also highly extensible and easily modified by Sun employees should Project Darkstar requirements change in the future.

8.1.2 Launcher and Project Snowman Demo

After installation, the next logical step in the out-of-box experience was to appeal to users by demonstrating Project Darkstar in action. Project Snowman is a simple MMO game developed at Sun Microsystems to serve two purposes: Provide the Project Darkstar development team with a complete application with which to test the performance of their software, and to use as a demonstration for game conferences. The team realized the potential for using Project Snowman as a demonstration for the Project Darkstar out-of-box experience as it simplified the development of the out-of-box experience and relieved us of the task of creating an entire game from scratch, something which would have taken a considerable effort and would have been arguably infeasible. An issue with using Project Snowman remained, however. The game used Apache Ant in order to run. This created a dependency that the team would rather not burden users with. To overcome this, the team created a platform-independent launcher program, written in Java. This was easily feasible because the Ant script used to run simply ran the game via command line arguments that varied based on the operating system of the user. With the addition of the launcher application, the team added an easy point-and-click method for users to get a first-hand experience of Project Darkstar applications, appealing to users and increasing the effectiveness of the out-of-box experience.

8.1.3 Tutorials

The tutorials are the final step in the out-of-box experience, designed for users familiar with Java that will use Project Darkstar to create their own game server systems. A set of tutorials already existed prior to the onset of this MQP; however, they were large inadequate for a variety of reasons. Consistency in design was a major issue. The tutorials were originally separated into two documents: one for server tutorials and one for client tutorials. The final product of each set of tutorials was a functional client or server application for the client and server tutorials, respectively. The issue here was that these applications were not designed to interact with each other. This gave users a set of examples, but did not address such issues as coherency and communication between the client and server. The pre-existing tutorials also were provided in separate downloads from the Project Darkstar website. The project team felt that this detracted from the overall experience as it could potentially frustrate users who would want to view tutorials that explained the creation of a client-server architecture. Finally, each individual tutorial was unrelated with other tutorials, even those within the same client or server document. The team's revamped version of the tutorials combined both client and server aspects of the tutorials into one document. Also, each sequential tutorial builds off the last, reducing iteration and simplifying the coding process. Tutorials are also reformatted into colorful html pages, which employ a simple and easy-to-read design that is more appealing than the black and white format of the pre-existing tutorials.

Along with the tutorial documents, supplementary code for every tutorial is provided. Pre-made projects are included to give users immediate access to working Project Darkstar code. These projects eliminate required set-up that would normally have to be performed by the user, enhancing the out-of-box experience by allowing them to bypass the initial set-up tutorial.

Tutorials are split into two major sections. First, a set of basic tutorials walks users through the Project Darkstar API and introduced the various interfaces and client/server architecture concepts incrementally. An optional "challenge" tutorial is also provided following the completion of the basic tutorials. The challenge tutorial walks users through the production of the server system for the Project Snowman game, giving users both a first-hand experience at what is required to create a complete MMO game using Project Darkstar, as well as demonstrating the ease as which this system can be implemented.

The challenge tutorial is the final deliverable and also represents the conclusion of the out-of-box experience.

8.2 Evaluation

The final deliverables represent an out-of-box experience, a usability device aimed at increasing awareness and interest for Project Darkstar. As such, a comprehensive evaluation would include the results of a usability study. The requirements necessary for completing such a study were not feasible within the seven-week timeframe of the project, nor did the team's Sun liaison request it.

An effective evaluation of the current status of the project is left to the improvements made over the existing system to create a more usable environment and the comments received from various Sun employees.

The first major improvement was the addition of the installer program. There have been many complications regarding the previous installation procedure[37], which have been eliminated with this program. Second, the team has included a packaged game that utilizes the Project Darkstar server system and an included launcher, giving users a first-hand experience at the capabilities of Project Darkstar that was not included otherwise. Finally, the tutorials improve on the existing tutorial set by uniting the server and client sections into a complete representation of a full Project Darkstar system with provided code. All deliverables are designed to be easily adaptable by Sun employees for future use.

The team presented the result of the project to a group of Sun employees at the conclusion of A-term. The comments the team received were very positive[38] [39] and the general consensus was that the project was beneficial to Sun and the Project Darkstar team. John Crowell, a Sun employee, stated the following in an email to the project team: "You guys did a great job. I'm looking forward to being able to test it myself and using it as a springboard to introduce newbie users to Darkstar."[40]

8.3 Future

As Project Darkstar continues to grow and develop, changes may need to be made to the various components of the out-of-box experience. Changes to the installer need only be minor, as the installer was designed to be extensible and easily adaptable to various changes. One possible adaptation to the user interface could include a form to choose which components to install; e.g. users could elect to not install the Project Snowman game to save hard disk space. The launcher application also needs little to no adjustment, as it simply hides JVM command-line arguments, which a user can re-enter as necessary. Most future work will lie in the tutorial documents and supplements. Sun Microsystems will add new features over time, and the tutorial documents will need to change accordingly. The project team projected the aesthetic style associated with the Project Darkstar logo and website onto the design of the tutorial HTML layout, which will also need necessary changes in order to maintain consistency. Content changes as mandated by Sun and/or the community members using the tutorials are a possibility, including various edits to

[37] "Project Darkstar Community - Can't install Project Darkstar?." 7 Dec 2008 <http://www.projectdarkstar.com/component/option,com_smf/Itemid,120/topic,714.0>.

[38] Personal Conversation with Jim Waldo. 15 October, 2008.

[39] Kotzen, Jennifer. "Re: Stay in touch!" Email to Gibler, et al. 15 October 2008.

[40] Crowell, John. "Re: Stay in touch!" Email to Gibler, et al. 15 October 2008.

both the tutorial text and supplementary code. Finally, concrete evidence as to the effectiveness of the out-of-box experience can be obtained through the employment of a usability study. This study is, at this point in time, undefined; it must, however, must be comprehensive enough to encapsulate all areas of the out-of-box experience as well as comparisons with the previous set-up. This can be represented informally (most likely) by feedback from community members or more formally as a professional analysis obtained from a larger audience.

8.4 Final Word

Following the conclusion of this project the team hopes that it has made a profound impact on Project Darkstar and that the team's work will be useful in generating attention for the system and improving the way developers create MMO applications. The impact of this project has potential far-reaching effects. The game industry, as well as several other applications, can greatly benefit from the proliferation of Project Darkstar, the advent of which could change the focus of MMO development from a costly venture to a reasonable effort. For games, this creates more competition for large game corporations and enhances the quality of MMO applications as a whole. The team hopes that Sun utilizes Project Darkstar for the betterment of the MMO developer community and that the project has been beneficial in working towards this goal.

9 References

"184. Sun Microsystems." CNN Money. Fortune 500. 2008 <http://www.wpi.edhttp://money.cnn.com/magazines/fortune/fortune500/2008/snapshots/881.html>.

"Acresso Software eShop - Buy Acresso Software Products". Acresso Software. 15 Dec 2008 <http://shop.acresso.com/product/fullproducts.asp#q>.

"bunny-hunters: Project Home Page ". Sun Microsystems. 15 Dec 2008 <https://bunny-hunters.dev.java.net/>.

"darkstar-hack: Hack - A Project Darkstar Game." 7 Dec 2008 <https://darkstar-hack.dev.java.net/>.

"darkstar-hack: Hack - A Project Darkstar Game." Sun Microsystems. 15 Dec 2008 <https://darkstar-hack.dev.java.net/>.

"Gamasutra - Features - Postmortem: Mythic's Dark Age of Camelot." Gamasutra. 7 Dec 2008 <http://www.gamasutra.com/features/20020213/firor_01.htm>.

Gibson, Ellie. "Blizzard to compensate players for World of Warcraft problems." gamesindustry.biz. 09 Aug. 2005. <http://www.gamesindustry.biz/articles/blizzard-to-compensate-players-for-world-of-warcraft-problems>.

"Independent MMO Game Developer's Conference 3.0." IMGDC. 7 Dec 2008 <http://www.imgdc.com/>.

"InstallJammer - A free, open source, multiplatform installer - Home". 15 Dec 2008 <http://www.installjammer.com/>.

"InstallShield Features - MSI Software Installation Tools - Acresso". Acresso Software. 15 Dec 2008 <http://www.acresso.com/products/is/installshield-features.htm>.

"InstallShield - MSI Windows Installer and InstallScript Installation Tool - Acresso". Acresso

Software. 15 Dec 2008 <http://www.acresso.com/products/is/installshield-overview.htm>.

"Model-View-Controller Pattern". eNode. 15 Dec 2008 <http://www.enode.com/x/markup/tutorial/mvc.html>.

Kellet, Owen. "Project Snowman: Lessons Learned". 15 Dec 2008 <http://owenkellett.com/2008/09/23/project-snowman-lessons-learned/>.

Kesselman, Jeffrey. "Project Darkstar Server Application Tutorial." 24 April 2008 7 Dec 2008 <http://www.projectdarkstar.com/ccount/click.php?id=29>.

Personal Conversation with Jim Waldo. 25 August, 2008.

Personal Conversation with Jim Waldo. 30 August, 2008.

Personal Conversation with Keith Thompson. 4 September, 2008.

Personal Conversation with Seth Proctor. 27 August, 2008.

Ponge, Julien. "IzPack - Package once. Deploy everywhere.". 15 Dec 2008 <http://izpack.org/>.

"Project Darkstar Community - Can't install Project Darkstar?." 7 Dec 2008 <http://www.projectdarkstar.com/component/option,com_smf/Itemid,120/topic,714.0>.

"Project Darkstar Community - Problems running HelloWorld example". Sun Microsystems. 15 Dec 2008
<http://www.projectdarkstar.com/component/option,com_smf/Itemid,120/topic,672.0>.

"Project Darkstar Community - Projects". Sun Microsystems. 07 December, 2008 <http://www.projectdarkstar.com/external/projects.html>.

"Project Darkstar Community - Running a server from within NetBeans." 7 Dec 2008 <http://www.projectdarkstar.com/component/option,com_smf/Itemid,120/topic,634.

Project Snowman Code. 2 September, 2008.

"Properties (Java Platform SE 6) ". Sun Microsystems. 15 Dec 2008 <http://java.sun.com/javase/6/docs/api/java/util/Properties.html>.

"Server Installation not working on WindowsXP 32 bit". Sun Microsystems. 15 Dec 2008

<http://www.projectdarkstar.com/component/option,com_smf/Itemid,120/topic,720.0>.

"TWiki . Games . ProjectSnowman". Sun Microsystems. 15 Dec 2008 <http://wiki.java.net/bin/view/Games/ProjectSnowman>.

"WoW Downtime Interview at Penny Arcade ." Slashdot. 24 Jan. 2005. 11 Dec 2008 <http://games.slashdot.org/article.pl?sid=05/01/24/1855218&tid=209>.

"XNA Creator's Club Online." 7 Dec 2008 <http://creators.xna.com/en-US/>.

10 Appendix A – Final Metadata

```
      EMBEDDED
name:PDS_SERVER
source:/targets/
file:pds-server.zip
destination:./#PDS_BINARY_DIR#/server/

      EMBEDDED
name:PDS_SERVER_BDB
source:/targets/
file:bdb.zip
destination:./#PDS_BINARY_DIR#/server/lib/bdb/

      EMBEDDED
name:PDS_SERVER_MINA
source:/targets/
file:mina-core.zip
destination:./#PDS_BINARY_DIR#/server/lib/

      EMBEDDED
name:PDS_SERVER_SLF4J
source:/targets/
file:slf4j.zip
destination:./#PDS_BINARY_DIR#/server/lib/

      EMBEDDED
name:PDS_CLIENT
source:/targets/
file:pds-client.zip
destination:./#PDS_BINARY_DIR#/client/

      EMBEDDED
name:SNOWMAN_EXEC
source:/targets/
file:snowmanexec.zip
destination:./#SNOWMAN_DIR#/

      EMBEDDED
name:SNOWMAN_CLIENT_LIB
source:/targets/
file:sm_clientlib.zip
destination:./#SNOWMAN_DIR#/Client/lib/

      EMBEDDED
name:SNOWMAN_CLIENT_LIB_LWJGL_BIN
source:/targets/
```

```
file:lwjgl_bin.zip
destination:./#SNOWMAN_DIR#/Client/lib/lwjgl/

      EMBEDDED
name:SNOWMAN_CLIENT_LIB_LWJGL_JAR
source:/targets/
file:lwjgl_jar.zip
destination:./#SNOWMAN_DIR#/Client/lib/

      EMBEDDED
name:SNOWMAN_CLIENT_LIB_MINA
source:/targets/
file:mina-core.zip
destination:./#SNOWMAN_DIR#/Client/lib/

      EMBEDDED
name:SNOWMAN_CLIENT_LIB_SLF4J
source:/targets/
file:slf4j.zip
destination:./#SNOWMAN_DIR#/Client/lib/

      EMBEDDED
name:SNOWMAN_CLIENT_LIB_JORBIS
source:/targets/
file:jorbis.zip
destination:./#SNOWMAN_DIR#/Client/lib/

      EMBEDDED
name:SNOWMAN_SERVER_LIB
source:/targets/
file:sm_serverlib.zip
destination:./#SNOWMAN_DIR#/Server/lib/

      EMBEDDED
name:SNOWMAN_SERVER_LIB_BDB
source:/targets/
file:bdb.zip
destination:./#SNOWMAN_DIR#/Server/lib/bdb/

      EMBEDDED
name:SNOWMAN_SERVER_LIB_MINA
source:/targets/
file:mina_core.zip
destination:./#SNOWMAN_DIR#/Server/lib/

      EMBEDDED
name:SNOWMAN_SERVER_LIB_MINA
source:/targets/
file:slf4j.zip
```

```
destination:./#SNOWMAN_DIR#/Server/lib/

      EMBEDDED
name:PDS_TUTORIALS
source:/targets/
file:tutorials.zip
destination:./#TUTORIAL_DIR#/


      EMBEDDED
name:PDS_TUTORIALS_MINA
source:/targets/
file:mina-core.zip
destination:./#TUTORIAL_DIR#/tutorial_projects/tutorial_libs/

      EMBEDDED
name:PDS_TUTORIALS_BDB_JAR
source:/targets/bdb.zip/
file:db.jar
destination:./#TUTORIAL_DIR#/tutorial_projects/tutorial_libs/

      EMBEDDED
name:PDS_TUTORIALS_BDB_BIN
source:/targets/bdb.zip/
file:#BDB#
destination:./#TUTORIAL_DIR#/tutorial_projects/tutorial_libs/bdb/

      EMBEDDED
name:PDS_TUTORIALS_SGS
source:/targets/pds-server.zip/lib/
file:sgs.jar
destination:./#TUTORIAL_DIR#/tutorial_projects/tutorial_libs/

      EMBEDDED
name:PDS_TUTORIALS_SLF4J
source:/targets/
file:slf4j.zip
destination:./#TUTORIAL_DIR#/tutorial_projects/tutorial_libs/

      EMBEDDED
name:PDS_TUTORIALS_SGS_CLIENT_JAR
source:/targets/pds-client.zip/lib/
file:sgs-client.jar
destination:./#TUTORIAL_DIR#/tutorial_projects/tutorial_libs/

      EMBEDDED
name:PDS_TUTORIALS_TUTORIAL_PROJECTS
source:/targets/
file:nb_projects.zip
```

```
destination:./#TUTORIAL_DIR#/tutorial_projects/
```

11 Appendix B: Early Metadata

```
        EMBEDDED
name:PDSInstall
source:/targets/
file:pds-server.zip
destination:./pds-server/

        EMBEDDED
name:BDB
source:/targets/
file:bdb.zip
destination:./pds-server/lib/bdb/

        EMBEDDED
name:MINA
source:/targets/
file:mina-core.zip
destination:./pds-server/lib/

        EMBEDDED
name:slf4j
source:/targets/
file:slf4j.zip
destination:./pds-server/lib/

        EMBEDDED
name:CLIENT
source:/targets/
file:pds-client.zip
destination:./pds-client/
```

12 Appendix C: Metadata and System-Specific Variables for Installer

```
# This file contains all of the installer wide variables used in the
metadata.
# This section contains the system specific variables. If the variable
follows the pattern
# [varname].sys.[os].[architecture], then the system will store in the
variable database
# a variable named [varname] who is set to the proper value based on the
operating system
# that the application is run on. As a side effect, you cannot have a
property named sys,
# and you should not have a variable being set that has the same name as a
system variable.

LWJGL.sys.windows.x86 = win32-x86
LWJGL.sys.macosx.x86 = macosx-x86
LWJGL.sys.linux.x86 = linux-x86
BDB.sys.windows.x86 = win32-x86
BDB.sys.macosx.x86 = macosx-x86
BDB.sys.linux.x86 = linux-x86
BDB.sys.linux.x64 = linux-x86_64
BDB.sys.macosx.ppc = macosx-ppc
BDB.sys.solaris.sparc = solaris-sparc
BDB.sys.solaris.x86 = solaris-x86

# Variables that do not start with sys. are treated as regular variables.

TEST_DIR = test
PDS_BINARY_DIR = binaries
SNOWMAN_DIR = snowmanexec
TUTORIAL_DIR = tutorials
```

13 Appendix D: Challenge Tutorial Summary

1) On the server, spawn a MatchMakerTask so that players can join a game.
    a. Have the server store joining players in an Array of Deques, which are ManagedObjects.
2) On the client, implement logging in.
    a. Have the reader use the SimpleClient to connect to the server.
    b. Once connected, the user authenticates their username and password, and then logs in once they're authenticated.
3) On the server, implement the processing of login requests.
    a. When a client joins, create a SnowmanPlayerListener to listen to messages from that client.
    b. Put the SnowmanPlayerListener into the Deque of players waiting to join a game, that is held in the MatchmakerTask.
4) On the server, implement the startup of a game.
    a. Spawn a new instance of a Game using a GameFactory.
    b. Add players to the Game.
    c. Inform all players in the Game that they're connected to it and tell them that the game is ready.
    d. Also, send each of them information about everyone else's starting position.
5) On the client, implement the startup of a game.
    a. When a user receives a game ready packet from the server, the client game spawns a new Game instance.
    b. The client Game world is populated with all of the players it's informed are in the current instance of the Game.
6) On the client, implement the sending of packets containing the updated position of the player.
    a. Have the client generate a packet containing the x and y coordinates of their Snowman.
    b. Send that packet to the server.
7) On the server, implement the processing of client sent packets, namely, position update messages.
    a. When a packet is received from a client, the server validates the packet and determines what kind of message it contains.
    b. Once the message is determined, the packet information is extracted.
    c. The goal here is to implement the processing of a position update message. When the server receives one, it simply repackages the packet into a packet the client can process and relays the position update to all clients connected to the Game where the packet was received from.

8) On the client, implement the processing of server sent packets, namely, position update messages.
    a. When a packet is received from the server, determine the type of packet.
    b. Once the type has been determined, the information can be extracted from it.
    c. Since the user is implementing the position update packet, they make it so that when the message is received, the entity to update is identified and their position in the client game is updated.

14 Appendix E: Challenge Tutorial Text

The project team includes the full text of one of the completed tutorials to show the level of detail which is asked and to demonstrate the time investment necessary for the longest tutorial offered.

```
TITLE
Implementing multiplayer into Project Snowman

OVERVIEW
This tutorial covers all of the key aspects for implementing multiplayer into
Project Snowman, from writing the login code for client and server to
invoking a game state update when a move message is received. However, it
does not cover all aspects - many of the details are skimmed over. Feel free
to peruse further into the Project Snowman code, but remember that the
purpose of this tutorial is to give a quick introduction into how to get
things to work in a "real" game.

REQUIREMENTS
Project Snowman source code with code segments stripped and dependencies
NetBeans 6.1+ (for building and running the client and server)

GOALS & DELIVERABLES
A complete Project Snowman that allows logins and multiplayer.

STEPS
Before we get started, it should be noted that you will be asked to fill in a
lot of code in a lot of empty methods in many different classes. It’s
important that whenever you make a change to a class that you save the file
you made a change to, so you don't experience any strange side-effects later.
Another thing that should be pointed out is that the tutorials are structured
such that you give the server functionality to handle messages from the
client, and then implement client-side the ability to send the corresponding
messages to the server to be handled. With that said, we can now continue
onto the tutorial.

Your first task is to load the Project Snowman files into NetBeans. This is
accomplished by opening NetBeans, selecting File->Open Project, navigating to
the folder where you unpacked your Project Snowman projects, and while
holding down the Ctrl key, click each project to select them, and then click
the 'Open Project' button. The projects you should select are snowman-client,
snowman-common, and snowman-server

Your 'Projects' box in NetBeans should now be populated with 3 projects -
SnowmanCommon, SnowmanClient, and SnowmanServer. The SnowmanClient and
SnowmanServer contain code unique to the client and server portions of
```

Project Snowman, respectively. SnowmanCommon is a library that contains data structures and processing logic that is common to both the client and server.

All 3 projects are incomplete. Much of the code needed to run Project Snowman is present, but a few things are missing that we are going to implement - the ability to login and the ability to send "move" messages to keep characters updated across all client computers.

What we want to do first is make it possible for players to log in. The first thing we want to do in this case is prepare the server to receive login requests from the client. This is handled in the SnowmanServer project in the class [com.sun.darkstar.example.snowman.server.SnowmanServer]. Go down to the [initialize] method, and insert the following code into it.

```
[
this.appContext = SnowmanAppContextFactory.getAppContext();
this.gameFactory = new GameFactoryImpl();
this.entityFactory = new EntityFactoryImpl();
this.waitingDeques = new ManagedReference[NUMDEQUES];
for(int i = 0; i < waitingDeques.length; i++) {
 Deque<ManagedReference<SnowmanPlayer>> deque = new
ScalableDeque<ManagedReference<SnowmanPlayer>>();
 waitingDeques[i] = appContext.getDataManager().createReference(deque);
}

this.config();
appContext.getTaskManager().scheduleTask(new
MatchmakerTask(numPlayersPerGame,
               numRobotsPerGame,
               robotDelay,
               gameFactory,
               entityFactory,
               appContext,
               waitingDeques));
]
```

This code looks complex, but it’s really not doing much right now.

This line

```
[
this.appContext = SnowmanAppContextFactory.getAppContext();
]
```

just gets the [AppContext] of the SnowmanServer, which gives you access to all of the [Manager] objects in Project Darkstar.

These lines

```
[
this.gameFactory = new GameFactoryImpl();
this.entityFactory = new EntityFactoryImpl();
]
```

are implementations of the factory design pattern and are used to spawn games and entities.

This line

```
[
this.waitingDeques = new ManagedReference[NUMDEQUES];
]
```

creates an array containing a certain number of [Deque]s, which by default is 10. A [Deque] is a double-ended queue, and is used for storing clients who are trying to join a game. The advantage of a Deque over simply a list of joining clients can only be understood if you understand the relationship between [Task]s and [ManagedObject]s. The [MatchmakerTask] that is created uses these Deques to grab clients who want to play a game, and if there are many [Task]s pulling from a single data structure, then contention occurs in the Project Darkstar transaction system, which can bring the overall system speed way down. Fortunately, Project Snowman has a [ScalableDeque] class that is designed to prevent this problem – it’s written such that it’s possible for many [MatchmakerTask]s to pull players from it efficiently.

The loop

```
[
for(int i = 0; i < waitingDeques.length; i++) {
 Deque<ManagedReference<SnowmanPlayer>> deque = new
ScalableDeque<ManagedReference<SnowmanPlayer>>();
 waitingDeques[i] = appContext.getDataManager().createReference(deque);
}
]
```

populates the array of [Deque]s by instantiating [ScalableDeque]s containing [ManagedReference]s to [SnowmanPlayer]s and storing a [ManagedReference] to the [ScalableDeque]s into the [waitingDeques] array.

The [config] method sets the various game settings by grabbing system properties and setting the corresponding variables to the corresponding system properties.

This chunk of code schedules a [MatchmakerTask] that knows about the waiting players and the game settings. The [MatchmakerTask] will be explained in further later.

```
[
appContext.getTaskManager().scheduleTask(new
MatchmakerTask(numPlayersPerGame,
               numRobotsPerGame,
               robotDelay,
               gameFactory,
               entityFactory,
               appContext,
               waitingDeques));
]
```

The next step is to add players to the [Deque] when they log in. This will be done in the [loggedIn] method in [SnowmanServer].

Uncomment the following code inside of the [loggedIn] method:

```
[
if (logger.isLoggable(Level.FINE))
 logger.log(Level.FINE, "Player {0} logged in", session.getName());
SnowmanPlayerListener player =
  new SnowmanPlayerListener(appContext,
          entityFactory.createSnowmanPlayer(appContext, session));
BigInteger id = player.getSnowmanPlayerRef().getId();
BigInteger index = id.mod(BigInteger.valueOf((long)NUMDEQUES));

waitingDeques[index.intValue()].get().add(player.getSnowmanPlayerRef());
return player;
]
```

The first set of lines

```
[
if (logger.isLoggable(Level.FINE))
 logger.log(Level.FINE, "Player {0} logged in", session.getName());
]
```

is used to log that a player has joined the server. This is useful - in a multi-threaded architecture like Project Darkstar, it can be hard to debug when there is a problem, and often times printing to stdout is a waste of time since the console can be filled with text quite quickly. Logging events is a very useful way of drilling down to the cause of a problem in Project Darkstar.

The next code chunk

```
[
SnowmanPlayerListener player =
  new SnowmanPlayerListener(appContext,
          entityFactory.createSnowmanPlayer(appContext, session));
]
```

creates a new [SnowmanPlayerListener]. This class is an implementation of [ClientSessionsListener], and its purpose is for listening for messages from its corresponding client. There is an instance of [SnowmanPlayerListener] on the server for each connected client.

The rest of the code

```
[
BigInteger id = player.getSnowmanPlayerRef().getId();
BigInteger index = id.mod(BigInteger.valueOf((long)NUMDEQUES));

waitingDeques[index.intValue()].get().add(player.getSnowmanPlayerRef());
return player;
]
```

ensures that each player in each [Deque] will have a unique id. This is vital for a multiplayer game, as an id number lets the server differentiate between the various entities present in the current game. Once a unique id has been generated, the player is added to a [Deque], and their [SnowmanSessionListener] is returned.

The [SnowmanServer] class is now complete. Support for client login has been implemented, but not much else is capable. We would like to be able to put the players into a game. To do this, we're going to have to set up [MatchmakerTask] to do this very thing.

Now, open the class [com.sun.darkstar.example.snowman.server.tasks.MatchmakerTask]. You will notice that most of the code is filled in except for the [run] and [startGame] methods. These are the "meat" of the [MatchermakerTask] - they are where the magic (and matchmaking) is made.

First, lets implement the [run] method. Uncomment the following code into that method:

```
[
boolean playersFound = false;
for(int i = 0; i < waitingDeques.length; i++) {
 ManagedReference<SnowmanPlayer> nextPlayer = waitingDeques[i].get().poll();
 if(nextPlayer != null) {
  playersFound = true;
  waitingPlayers.add(nextPlayer);
 }
 if(waitingPlayers.size() == numPlayersPerGame) {
  startGame();
  break;
 }
}

if(playersFound)
 appContext.getTaskManager().scheduleTask(this);
else
 appContext.getTaskManager().scheduleTask(this, POLLINGINTERVAL);
]
```

The [run] method initially assumes that no players have been found. Then, it begins iterating through the [Deque] containing the waiting players. If there are waiting players, it adds that players to a local [List] of waiting players, to be used later. If it has found enough players, it starts a game. If it fails to start a game, it reschedules itself immediately if it has players waiting, since it does not want the currently waiting players to wait any further. Otherwise, it does a periodic schedule since its not urgent that it run immediately - this is useful if there are other tasks running as the [MatchmakerTask] will not "hog" the scheduler to itself.

When a game is started, the [startGame] method is called. Uncomment this code in that method to give it some life:

```
[
boolean needMore = false;
for(Iterator<ManagedReference<SnowmanPlayer>> ip = waitingPlayers.iterator();
ip.hasNext(); ) {
 try {
  ip.next().get();
 } catch(ObjectNotFoundException e) {
  ip.remove();
  needMore = true;
 }
}
if(needMore)
 return;

String gameName = NAME_PREFIX + (gameCount++);
SnowmanGame game = gameFactory.createSnowmanGame(gameName,
             numPlayersPerGame + numRobotsPerGame,
             appContext,
             entityFactory);
ETeamColor color = ETeamColor.values()[0];
for(Iterator<ManagedReference<SnowmanPlayer>> ip = waitingPlayers.iterator();
ip.hasNext(); ) {
 game.addPlayer(ip.next().get(), color);
 color = ETeamColor.values()[(color.ordinal() + 1) %
ETeamColor.values().length];
}
for (int i = 0; i < numRobotsPerGame; i++) {
 game.addPlayer(entityFactory.createRobotPlayer(gameName + "_robot" + i,
robotDelay),
       color);
 color = ETeamColor.values()[(color.ordinal() + 1) %
ETeamColor.values().length];
}
game.sendMapInfo();

waitingPlayers.clear();
]
```

The first chunk of code

```
[
boolean needMore = false;
for(Iterator<ManagedReference<SnowmanPlayer>> ip = waitingPlayers.iterator();
ip.hasNext(); ) {
 try {
  ip.next().get();
 } catch(ObjectNotFoundException e) {
  ip.remove();
  needMore = true;
 }
}
if(needMore)
 return;
```
]
is there to check for any players that disconnected while the [run] method was grabbing players. If any players are missing from the [Deque], the [startGame] method returns and the [run] method continues execution, waiting for more players to join before calling [startGame] again.

Uncomment the following code in the [startGame] method:
[
```
String gameName = NAME_PREFIX + (gameCount++);
SnowmanGame game = gameFactory.createSnowmanGame(gameName,
             numPlayersPerGame + numRobotsPerGame,
             appContext,
             entityFactory);
ETeamColor color = ETeamColor.values()[0];
for(Iterator<ManagedReference<SnowmanPlayer>> ip = waitingPlayers.iterator();
ip.hasNext(); ) {
 game.addPlayer(ip.next().get(), color);
 color = ETeamColor.values()[(color.ordinal() + 1) %
ETeamColor.values().length];
}
for (int i = 0; i < numRobotsPerGame; i++) {
 game.addPlayer(entityFactory.createRobotPlayer(gameName + "_robot" + i,
robotDelay),
       color);
 color = ETeamColor.values()[(color.ordinal() + 1) %
ETeamColor.values().length];
}
game.sendMapInfo();
```
]

This part
[
```
String gameName = NAME_PREFIX + (gameCount++);
SnowmanGame game = gameFactory.createSnowmanGame(gameName,
             numPlayersPerGame + numRobotsPerGame,
             appContext,
             entityFactory);
```
]
creates a unique game instance which is to be populated with players. The following code does just that.
[
```
ETeamColor color = ETeamColor.values()[0];
for(Iterator<ManagedReference<SnowmanPlayer>> ip = waitingPlayers.iterator();
ip.hasNext(); ) {
 game.addPlayer(ip.next().get(), color);
 color = ETeamColor.values()[(color.ordinal() + 1) %
ETeamColor.values().length];
}
for (int i = 0; i < numRobotsPerGame; i++) {
 game.addPlayer(entityFactory.createRobotPlayer(gameName + "_robot" + i,
robotDelay),
       color);
 color = ETeamColor.values()[(color.ordinal() + 1) %
ETeamColor.values().length];
}
```
]
It looks complicated, but don't worry - it really isn't. The confusing lines here are
[
```
color = ETeamColor.values()[(color.ordinal() + 1) %
ETeamColor.values().length];
```
]
and
[
```
color = ETeamColor.values()[(color.ordinal() + 1) %
ETeamColor.values().length];
```
]
All these lines do is ensure that each team will get the same number of players in the case that the number of players is divisible by the number of the teams - in the other case, some teams may have one more players and others may have one less. The rest of the lines in this chunk of code add the players to the game instance. The method [game.sendMapInfo();] sends the map information to each connected player. The final line, [waitingPlayers.clear();] simply clears the list of waiting players so that a new game with new players can be spawned.

So now we have an initialized game that players can join. But while the server may be able to handle players joining, the client still can't connect. Time to switch gears for a moment and hop to the client side of things. Open up the SnowmanClient project, and navigate to [com.sun.darkstar.example.snowman.ClientApplication]. This is the starting point of the client. You will notice that a [ClientHandler] and [Client] object are created - these give you a way of communicating with the server. Navigate to both classes, and you will see that they have been filled in - this is intentional. As you will notice in [Client], most of the methods simply make calls to the [SimpleClient] class, significantly reducing the pain of communicating with the server. Once initialized, the [Client] object passes in a [MessageListener] to the [SimpleClient] constructor. The [MessageListener] is the yin to the [Client]'s yang - it receives messages from the server and channel rather than sending them out to either. Navigate to the class at [com.sun.darkstar.example.snowman.client.handler.message.MessageListener] and lets take a look at some of the things it does.

First of all, you'll notice that it has the [loggedIn], [loginFailed], [getPasswordAuthentication], [joinedChannel], and two [receivedMessage] methods all implemented. Note that we are only interested in the [receivedMessage] method that accepts only a [ByteBuffer], as it is the only one utilized in Project Snowman. When [SimpleClient] receives a message from the server, it contains an OpCode. That OpCode indicates which method is to be invoked, and [SimpleClient] does just that. The [loggedIn] and [loginFailed] methods both spawn a [Task] when they are invoked, to handle the various aspects of each. We will not concern ourselves with this for now, but its important to understand that both simply update the state of the game client. The method [joinedChannel] just returns the [MessageListener] object, as it is both a [ClientChannelListener] and [SimpleClientListener]. What we're more concerned about is the the [MessageHandler] that is called to parse packets from [ByteBuffer]s. Before we get into that, though, we should make sure that the client is logged in.

To accomplish this, navigate to the class [com.sun.darkstar.example.snowman.game.AuthenticateTask]. This class is a client side [Task] - its much different than a Project Darkstar [Task], but shares some similarities. Client side [Task]s, like Project Darkstar [Task]s, do some sort of job. They differ in that all Project Darkstar [Task]s are handled concurrently, whereas the client [Task]s are handled synchronously. Still, it gives us a useful way of splitting up processing logic, and [AuthenticateTask] is a good example of this. Uncomment the following code to the [execute] method in [AuthenticateTask]:
[
```
final LoginGUI gui =
((LoginState)GameStateManager.getInstance().getChild(EGameState.LoginState.to
String())).getGUI();
gui.setStatus(gui.getDefaultStatus());
InputManager.getInstance().setInputActive(false);
this.game.getClient().getHandler().authenticate(this.username,
this.password);
Properties properties = new Properties();
properties.setProperty("host", System.getProperty("host", "localhost"));
properties.setProperty("port", System.getProperty("port", "3000"));
this.game.getClient().login(properties);
```
]

Initially, the [execute] method tells the gui to go to its default status, which is attempting to log in. You should not concern yourself too much with the next line, [InputManager.getInstance().setInputActive(false);]. It's a jMonkeyEngine related code snippet - all it does is disable user input, since the game will automatically log the player in with no user action required. The next line is used to generate a [PasswordAuthentication] object in the [ClientHandler], which is used when logging into the server.
[

```
this.game.getClient().getHandler().authenticate(this.username,
this.password);
```

]

The next series of code attempts to login to the server.

[

```
Properties properties = new Properties();
properties.setProperty("host", System.getProperty("host", "localhost"));
properties.setProperty("port", System.getProperty("port", "3000"));
this.game.getClient().login(properties);
```

]

The [properties] object simply holds settings about the host to join. It then passes this information to the [login] method in the game [Client], which is the class used for sending data and messages to the server. The [login] method in the [Client] class wraps in the [SimpleClient]'s [login] method, the [SimpleClient] being part of the standard Java Project Darkstar API. When the client is successfully logged in, the [loggedIn] method that we covered in [MessageListener] is invoked, and the game state is updated to reflect the newly created connection with the server. Now that we're logged in, the server is going to want to tell us to start a game once enough people have joined. Recall that the [receivedMessage] method is invoked when data is received from the server. Open up the class [com.sun.darkstar.example.snowman.client.handler.message.MessageListener] and go to [receivedMessage] method with the stub [receivedMessage(ByteBuffer arg0)] and uncomment the following code in it.

[

```
SingletonRegistry.getMessageHandler().parseClientPacket(message,
this.handler.getProcessor());
```

]

The code [this.handler.getProcessor()] retrieves the [IClientProcessor] instance used by this [MessageHandler]. The [IClientProcessor] implementation used by the [MessageHandler] is the class [MessageProcessor]. After the processor is retrieved, the rest of the code makes a call to part of the SnowmanCommon library, and to get logins working correctly, we're going to dive into it. Open up ProjectSnowman and navigate to the class [com.sun.darkstar.example.snowman.common.protocol.handlers.MessageHandlerImpl]. This is the class whose purpose is to determine the message type for both the client and server, and react accordingly. In the case of starting a new game, the server must first ask the client if its ready, and then each client must respond that its ready before the game can begin. This is handled in the [parseCommonPacket] method. Uncomment the following code in the [parseCommonPacket] method:

[

```
switch (code) {
 case READY:
  logger.log(Level.FINEST, "Processing {0} packet", code);
  processor.ready();
  break;
 default:
  this.logger.warning("Unsupported OPCODE: " + code.toString());
}
]
```

Notice that the [parseCommonPacket] method takes an [IProtocolProcessor] as a parameter. That interface has one method, [ready], which is implemented in both the [IServerProcessor] and [IClientProcessor] classes. Both classes simply fire off a ready packet so that the other can react accordingly. Open up the SnowmanClient project and go to the class [com.sun.darkstar.example.snowman.client.handler.message.MessageProcessor]. Uncomment the following code in the [ready] method:

```
[
public void ready() {
      TaskManager.getInstance().createTask(ETask.Ready);
}
]
```

It creates a task, [ReadyTask], which when executed by the client sends a ready packet back to the server. Once the server receives that ready packet, it invokes its own [ready] method. The [IServerProcessor]'s [ready] method is in the class [com.sun.darkstar.example.snowman.server.impl.SnowmanPlayerImpl]. Uncomment the following code in it's [ready] method:

```
[
public void ready() {
      if (gameRef != null) {
       setReadyToPlay(true);
       gameRef.get().startGameIfReady();
      }
}
]
```

If the game exists, the server is ready to start the game. The [gameRef] variable refers to an instance of a [SnowmanGame]. That interface is implemented in the class [com.sun.darkstar.example.snowman.server.impl.SnowmanGameImpl]. Open that class now, and uncomment the following code in the [startGameIfReady] method:

```
[
public void startGameIfReady(){
 appContext.getDataManager().markForUpdate(this);
 readyPlayers++;
 if(readyPlayers >= realPlayers)
  send(ServerMessages.createStartGamePkt());
}
]
```

The method prepares its own object for an update, as data is about to change server-side. It then uses [ServerMessages] to create a start game packet. We will create a packet later, but for now you just need to understand that it’s simply creating a [ByteBuffer] that contains a NEWGAME OpCode and some other data about the game. It then uses the [send] method, which fires off a packet to the game channel, to let all clients know about the new game simultaneously.

This means that each client is going to receive a NEWGAME message. For this next part, open the SnowmanCommon project, and open the class [com.sun.darkstar.example.snowman.common.protocol.handlers.MessageHandlerImpl], which inherits the [MessageHandler] interface. In [MessageHandlerImpl], note that the [parseClientPacket] method with the method stub of [parseClientPacket(ByteBuffer packet, IClientProcessor processor)] is the only one used by external classes - the method simply extracts the OpCode from the packet and calls a more extensive [parseClientPacket] method with the stub [parseClientPacket(EOPCODE code, ByteBuffer packet, IClientProcessor unit)]. In that method, uncomment the following [case] in the [switch] statement:

[
```
case NEWGAME:
 int myID = packet.getInt();
 byte[] mapname = new byte[packet.getInt()];
 packet.get(mapname);
 String mapString = new String(mapname);
 logger.log(Level.FINEST, "Processing {0} packet : {1}, {2}", new
Object[]{code, myID, mapString});
 unit.newGame(myID, mapString);
 break;
```
]

What this [case] does is retrieve the players id as set by the server and the name of the map. Finally, a new game is started.

Now, we are able to have a player log into the server and a game can be started. Our last goal is to add in the ability to handle players moving around and keep player positions synchronized across all game clients. Fortunately, it’s very similar to what we just did. Recall that there is a [SnowmanPlayerListener] instance for each connected client, and that the methods in [SnowmanPlayerListener] are invoked when certain messages are received from the client. Open up the SnowmanServer project and open the class [com.sun.darkstar.example.snowman.server.SnowmanPlayerListener]. Go to the [receivedMessage] method and uncomment the following code.

[
```
try {

SingletonRegistry.getMessageHandler().parseServerPacket(arg0,playerRef.get().
getProcessor());
} catch (ObjectNotFoundException disconnected) {}
```
]

This code makes a call to the [MessageHandler] that we dealt with previously, but this makes a call to [parseServerPacket] rather than [parseClientPacket]. Both functions are very similar in nature - they handle network messages but deal with them in a different manner, as [parseClientPacket] causes client side code to be executed, and [parseServerPacket] causes server side code to be executed. Open up [com.sun.darkstar.example.snowman.common.protocol.handlers.MessageHandlerImpl] again and go to the method with the stub [parseServerPacket(ByteBuffer packet, IServerProcessor processor)]. Note that the method simply extracts the OpCode from the [ByteBuffer] and then calls the [parseServerPacket] method with the stub [parseServerPacket(EOPCODE code, ByteBuffer packet, IServerProcessor unit)]. That method has a [switch] statement that executes different behaviors depending on the OpCode that is passed to it. Uncomment this [case] to the [switch] statement:

```
[
case MOVEME:
 float moveStartX = packet.getFloat();
 float moveStartY = packet.getFloat();
 float moveEndX = packet.getFloat();
 float moveEndY = packet.getFloat();
 logger.log(Level.FINEST, "Processing {0} packet : {1}, {2}, {3}, {4}",
      new Object[]{code, moveStartX, moveStartY, moveEndX, moveEndY});
 unit.moveMe(moveStartX,
    moveStartY,
    moveEndX,
    moveEndY);
 break;
]
```

What occurs here is quite simple - the server pulls out the new position of the player from the packet, logs the occurrence, and then tells the players server-side instance to move. The [moveMe] method can be found in the class [com.sun.darkstar.example.snowman.server.impl.SnowmanPlayerImpl]. Go to the class and go to the [moveMe] method with the stub [moveMe(float startx, float starty, float endx, float endy)]. Uncomment this code in the method:

```
[
Long now = System.currentTimeMillis();
moveMe(now, startx, starty, endx, endy);
]
```

The server stores the time when the [moveMe] method is invoked. This is useful in the case of lag, so that the server may predict where a player is even if the player hasn't sent an update message to the server. Uncomment the code in the method with the stub [moveMe(long now, float startx, float starty, float endx, float endy)].

```
[
//no op if player is dead or not in a game
if(state == PlayerState.DEAD || state == PlayerState.NONE)
 return;

appContext.getDataManager().markForUpdate(this);

//verify that the start location is valid
Coordinate expectedPosition = this.getExpectedPositionAtTime(now);

if (checkTolerance(expectedPosition.getX(), expectedPosition.getY(),
       startx, starty,
       POSITIONTOLERANCESQD)) {
 //collision detection
 Coordinate trimPosition = appContext.getManager(GameWorldManager.class).
   trimPath(new Coordinate(startx, starty),
      new Coordinate(endx, endy));

 this.timestamp = now;
 this.startX = startx;
 this.startY = starty;
 this.destX = trimPosition.getX();
 this.destY = trimPosition.getY();
 this.state = PlayerState.MOVING;

 sendAll(ServerMessages.createMoveMOBPkt(id, startX, startY, destX, destY));
}
else {
 logger.log(Level.FINE, "move from {0} failed start position check", name);

 this.timestamp = now;
 this.setLocation(expectedPosition.getX(), expectedPosition.getY());
 sendAll(ServerMessages.createStopMOBPkt(id,
          expectedPosition.getX(),
          expectedPosition.getY()));
}
]
```

Initialy, the method checks to see if the player is dead or that there is no player.

```
[
//no op if player is dead or not in a game
if(state == PlayerState.DEAD || state == PlayerState.NONE)
 return;
]
```

In that case, there is no need to do any further processing and you can simply return from the method - you won't be able to move the player anyways. If the player is around and alive, its time to prepare the object in the data store for an update. This is accomplished by this line:
[appContext.getDataManager().markForUpdate(this);]
The next block of code is for collision detection.

```
[
Coordinate trimPosition = appContext.getManager(GameWorldManager.class).
  trimPath(new Coordinate(startx, starty),
     new Coordinate(endx, endy));
]
```

The [trimPath] method determines if there is anything on the map that would impede movement. If so, the [trimPath] method returns a [Coordinate] on the path before the collision with the map would occur. The server, in this case, is allowing only a valid move to occur, and preventing the player from moving through things. This is a good example of the server holding the "truth" in that the server controls the constraints of the game, not the client.

```
[
this.timestamp = now;
this.startX = startx;
this.startY = starty;
this.destX = trimPosition.getX();
this.destY = trimPosition.getY();
this.state = PlayerState.MOVING;

sendAll(ServerMessages.createMoveMOBPkt(id, startX, startY, destX, destY));
]
```

The [SnowmanPlayer] whose [moveMe] method was called has their state updated to [MOVING], and their current position and destination updated. Then, the [sendAll] method is used to broadcast a move packet to the game channel. Let’s take a look at how this packet is formed. Open up the project SnowmanCommon, and then open up the class [com.sun.darkstar.example.snowman.common.protocol.messages.ServerMessages] and go to the method [createMoveMOBPkt]. Uncomment the following code:

```
[
byte[] bytes = new byte[1 + 8 + 28];
ByteBuffer buffer = ByteBuffer.wrap(bytes);
buffer.put((byte) EOPCODE.MOVEMOB.ordinal());
buffer.putInt(targetID);
buffer.putFloat(startx);
buffer.putFloat(starty);
buffer.putFloat(endx);
buffer.putFloat(endy);
return buffer;
]
```

This code creates a [ByteBuffer] and fills it with the data to be transmitted, namely the OpCode (MOVEMOB), the id of the character to be updated, the starting position of the character, and the ending position of the character. This information is then broadcast to the channel and to each client. Now that the server is capable of handling client movement, we will now add the client-side capability to send out move messages to the server. Open the SnowmanClient project and then open the class [com.sun.darkstar.example.snowman.game.entity.controller.SnowmanController]. This class handled events on the client-side, such as button presses. Go to the method [onButton] and uncomment the following code:

```
[
if(!this.isActive()) return;
if(this.getEntity().getState() == EState.Attacking) return;
  if(this.getEntity().getState() == EState.Hit) return;
if(button == 0 && pressed) {
      TaskManager.getInstance().createTask(ETask.UpdateCursorState,
this.entity, x, y).execute();
      switch(this.getEntity().getCursorState()) {
     case Invalid:
      //do nothing
      break;
     case TryingToMove:
      TaskManager.getInstance().createTask(ETask.MoveCharacter, this.entity,
x, y);
      break;
     case Targeting:
      TaskManager.getInstance().createTask(ETask.Attack, this.entity.getID(),
this.getEntity().getTarget().getID());
      break;
     case TryingToGrab:
      TaskManager.getInstance().createTask(ETask.Attach,
this.getEntity().getTarget().getID(), this.entity.getID(), true);
      break;
      }
}
]
```

The chunk that we're interested in is this one.

```
[
case TryingToMove:
 TaskManager.getInstance().createTask(ETask.MoveCharacter, this.entity, x,
y);
 break;
]
```

This [onButton] method is invoked when the player clicks in the game window. The [case] [TryingToMove] creates a [MoveCharacterTask], which moves the player. Open the class [com.sun.darkstar.example.snowman.game.task.state.battle.MoveCharacterTask], and go to the method [getDestination]. Uncomment the code in it:

```
[
if(this.local) {
      DisplaySystem display = DisplaySystem.getDisplaySystem();
      CollisionManager collisionManager =
SingletonRegistry.getCollisionManager();
      Ray ray = new Ray();
      display.getPickRay(new Vector2f(this.x, this.y), false, ray);
      World world =
this.game.getGameState(EGameState.BattleState).getWorld();
      Vector3f click = collisionManager.getIntersection(ray, world, new
Vector3f(), true);
      if(click == null) return null;
      try {
            Spatial view =
(Spatial)ViewManager.getInstance().getView(this.character);
            Vector3f local = view.getLocalTranslation().clone();

      this.game.getClient().send(ClientMessages.createMoveMePkt(local.x,
local.z, click.x, click.z));
            return collisionManager.getDestination(local.x, local.z, click.x,
click.z, world.getStaticRoot());
      } catch (ObjectNotFoundException e) {
            e.printStackTrace();
            return null;
      }
} else {
      return new Vector3f(this.endX, 0, this.endZ);
}
]
```

Take a look at the try block, specifically, this line.

```
[
this.game.getClient().send(ClientMessages.createMoveMePkt(local.x, local.z,
click.x, click.z));
]
```

This is where the client actually sends a message to the server with its destination. Let’s take a look at how the move me packet is created. Open the SnowmanCommon project and open the class [com.sun.darkstar.example.snowman.common.protocol.messages.ClientMessages]. Uncomment the following code in the [createMoveMePkt] method:

```
[
byte[] bytes = new byte[1 + 8 + 16];
ByteBuffer buffer = ByteBuffer.wrap(bytes);
buffer.put((byte) EOPCODE.MOVEME.ordinal());
buffer.putFloat(x);
buffer.putFloat(y);
buffer.putFloat(endx);
buffer.putFloat(endy);
return buffer;
]
```

You will notice that this function is extremely similar to the [createMoveMOBPkt] method in [ServerMessages], and that it differs only in the enum used for the OpCode and that it does not store its id in the packet. There is a good reason for not storing the id - the server already knows who is sending it a message when it receives one since it has a [SnowmanPlayerListener] instance for each player connected - it would be a waste of data that could be put to better use elsewhere. When writing a scalable online game, you want to send only the minimal amount of data to the server neccessary so that the game state is synchronized properly among all clients, to keep latency low. Our final goal is to give the client the capability to handle move messages from the server to update the positions of other player. Recall the class [MessageListener], which handled logins for us. This class handles all messages received from the server, including move me messages. Open the project SnowmanClient and open the class [com.sun.darkstar.example.snowman.client.handler.message.MessageListener]. Go to the [receivedMessage] method with the stub [receivedMessage(ByteBuffer message)]. Recall the code you uncommented in it:

```
[
SingletonRegistry.getMessageHandler().parseClientPacket(message,
this.handler.getProcessor());
]
```

This is again using the SnowmanCommon class [MessageHandler] to parse the message and use the [MessageProcessor] of the client to execute the behavior

indicated by the message. Open up the SnowmanCommon project and open the class [com.sun.darkstar.example.snowman.common.protocol.handlers.MessageHandlerImpl]. Go to the [parseClientPacket] method with the stub [parseClientPacket(EOPCODE code, ByteBuffer packet, IClientProcessor unit)] and uncomment the following [case] in the [switch] statement:
[
```
case MOVEMOB:
 int moveId = packet.getInt();
 float moveStartX = packet.getFloat();
 float moveStartY = packet.getFloat();
 float moveEndX = packet.getFloat();
 float moveEndY = packet.getFloat();
 logger.log(Level.FINEST, "Processing {0} packet : {1}, {2}, {3}, {4}, {5}",
      new Object[]{code, moveId, moveStartX, moveStartY, moveEndX,
moveEndY});
 unit.moveMOB(moveId,
     moveStartX,
     moveStartY,
     moveEndX,
     moveEndY);
 break;
```
]
This [case] handles movement packets received from the server. The code
[
```
int moveId = packet.getInt();
float moveStartX = packet.getFloat();
float moveStartY = packet.getFloat();
float moveEndX = packet.getFloat();
float moveEndY = packet.getFloat();
```
]
retrieves all of the numerical data from the packet - the id number of the player to move, that players start position, and that players end position. The next chunk
[
```
logger.log(Level.FINEST, "Processing {0} packet : {1}, {2}, {3}, {4}, {5}",
     new Object[]{code, moveId, moveStartX, moveStartY, moveEndX, moveEndY});
```
]
logs the occurence. As stated earlier, this is very useful for debugging when something goes wrong. The chunk
[
```
unit.moveMOB(moveId,
  moveStartX,
  moveStartY,
  moveEndX,
  moveEndY);
```
]
is the part that we're interested in. Recall that the [IClientProcessor] that was passed in was the [MessageProcessor] class from the client, and therefore

the behavior for the [moveMOB] method will be found in it. Open up the SnowmanCommon project, and open the class [com.sun.darkstar.example.snowman.client.handler.message.MessageProcessor]. Go to the method [moveMOB] and uncomment the following code:

```
[
if(objectID == this.myID) return;
TaskManager.getInstance().createTask(ETask.MoveCharacter, objectID, startx,
starty, endx, endy);
]
```

The if statement [if(objectID == this.myID) return;] is there since when the move me message is broadcast to the channel, the client will receive that message back, and there is no purpose to update the clients position based on what the server tells it - the client already knows its local snowman's position. Otherwise, a [MoveCharacterTask] is executed client-side. Open the class [com.sun.darkstar.example.snowman.game.task.state.battle.MoveCharacterTask]. Uncomment the following code in the [execute] method:

```
[
if (this.character == null) return;
      try {
            this.character.resetVelocity();
            this.character.resetForce();
            Vector3f destination = this.getDestination();
            if (destination != null) {
                  this.character.setDestination(destination);
                  Spatial view =
(Spatial)ViewManager.getInstance().getView(this.character);
                  if(!this.local) {
                        view.getLocalTranslation().x = this.startX;
                        view.getLocalTranslation().z = this.startZ;
                  }
                  destination.y = 0;
                  Vector3f lcoal = view.getLocalTranslation().clone();
                  lcoal.y = 0;
                  Vector3f direction = destination.subtract(lcoal);
                  direction.y = 0;
                  direction.normalizeLocal();
                  view.getLocalRotation().lookAt(direction, Vector3f.UNIT_Y);
                  Vector3f force =
direction.multLocal(EForce.Movement.getMagnitude());
                  this.character.addForce(force);
                  PhysicsManager.getInstance().markForUpdate(this.character);
                  // Step 9.
                  this.character.setState(EState.Moving);
                  ViewManager.getInstance().markForUpdate(this.character);
            }
      } catch (Exception e) {
            e.printStackTrace();
      }
```

]
The initial if statement just checks to see if the character actually exists - if it doesn't, there is no reason to execute the task. The rest of the code is there to actually update the position of the snowman, using the jMonkey Engine API. Simply stated, it gets the destination of the snowman to update and moves it there. Feel free to look into the jMonkey Engine API calls, as they are beyond the scope of this tutorial.

Time to give yourself a pat on the back, and to take a break - you just did a lot of work! However, it would be wise to review what you just did - it was actually quite simple. When the client sent out a message to the channel, you had an OpCode specifying the type of message. Then the server validated the message based on the OpCode and data, and repackaged it with the same OpCode and similar data and sent it back out to the clients, who knew how to deal with it based on the OpCode the server sent along with the message. Project Darkstar makes it very easy to implement this functionality. You now have the ability to fully utilize this technology, as many game projects you may work on will implement their messaging system using similar concepts.

To recap:
SnowmanServer - [SnowmanServer] Initializes the server portion of Project Snowman; accepts incoming connections.
SnowmanServer - [MatchmakerTask] Sets up all of the game's on the server.
SnowmanClient - [MessageListener] Listens for messages client-side that come from the server.
SnowmanClient - [MessageProcessor] Contains behavior to handle the various messages that the client receives.
SnowmanCommon - [MessageHandler]/[MessageHandlerImpl] Parses packets both client-side and server-side and makes the proper calls to the [IProtocolProcessor], [IClientProcessor], or [IServerProcessor] supplied.
SnowmanServer - [SnowmanPlayer]/[SnowmanPlayerImpl] Contains behavior to handle the various messages that the server receives from the client.
SnowmanClient - [SnowmanController] Responds to user interaction with the game window.
SnowmanCommon - [ClientMessages]/[ServerMessages] Each contains a set of methods used for generating packets, for the client and server respectively.