January 2010

EVE Mac Suite Development

Vadim Lozko
Worcester Polytechnic Institute

Project Number: CS-MXC-300

EVE Mac Suite

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

Vadim Lozko
Date: January 12, 2010

Approved:

Prof. Michael J. Ciaraldi, Major Advisor

Abstract

The project was an experience in open source development. A Mac program, using Apple's latest technologies, was written to interact with EVE Online's API. A GPL licensed program was built in conjunction with interacting with players, a dev blog and an open source hosting company. Programming challenges included XML parsing, persistent databases, multithreading, animation and proper model-view-controller implementations. In the end, an application whose functionality is similar to EVEMon was produced.

Acknowledgement

Professor Ciaraldi would be the one I'd like to thank the most. His willingness to take on this project at a late notice and all of his insight is duly appreciated. Quite a bit of volunteer help came from the EVE Online Mac community. Their input helped me know what my requirements and goals were during the development process. Finally, I want to thank CCP for creating a game that gives developers quite a bit to work with.

Table of Contents

Table of Figures

Intro

EVE Online, produced by CCP, is a massive multiplayer online game (MMO) similar to the popular World of Warcraft. It was originally released in 2003 as a Windows-only game before a Mac client was released in 2005. While games like EverQuest, World of Warcraft and others focused on being a fantasy game, EVE was one of the first with a science fiction genre.

Also in 2005 was the release of the EVE API, a way to allow players to monitor progress of their character or corporation (similar to a guild or kin). Of the applications that have take advantage of this API the most popular ones have been Windows-only. The one that stands out the most is EVEMon, a character monitor and skill planner. While EVEMon is open source, it's written almost exclusively in C#, a language that can currently only be compiled in Windows. Another is EVE Fitting Tool, an application that allows a player to compare different fits for a ship and see how well the ship will perform based on the character's status. This one is developed by a single developer and is closed source.

So, why not have such applications for a Mac? They constitute 5-10% of all EVE players out of 300,000 so there's certainly a market for it. If the application was native to Mac OS X, precious resources to play the EVE Online at the same time wouldn't have to be used up for an emulator to run EVEMon. The purpose of this MQP was to create an open-source EVE character monitor application that's native to Macs. To enhance the learning experience of developing an open source application a development blog was created to communicate between the players

and myself. In the future it could potentially do much more than EVEMon, hence the name of the program, EVE Mac Suite.

Background

EVE Online

There are some notable differences between EVE and other online games. While most games have several realms or servers that separate the individual game worlds, the entire EVE universe is located on one server cluster. This means any player can interact with anyone else. Another is that game play is completely player-driven, also known as a sandbox. Territorial claims and the economy are affected by the actions of the players. There is no ultimate "boss" to kill, rank to attain or level to reach.

Perhaps the most notable difference is the way a player advances their character. The vast majority of online games require a player to accomplish something within the game and consume a certain amount of time in the process. In doing so they are able to use new items or use existing ones more powerfully. In EVE, skill books are used to teach skills to a character. These books are readily purchasable in the game's market. Some books aren't learnable until another skill is trained to a certain point. Training skills allow the characters to access more items, use the items more efficiently or advance themselves in some aspect of the game.

Named groups organize the players' skills, although the groups in themselves don't have significance. Each skill has a rank, prerequisite skills, primary attribute and secondary attribute. The primary and secondary attributes are one of five: intelligence, memory, perception, willpower and charisma. A player starts off with

points in each one of them. They can increase them with either items called attribute enhancers or through training specific skills. Prerequisite skills are skills that must be trained to a certain level before the desired skill can be trained. Not all skills have prerequisites, though. Rank is an integer skill training time multiplier with a base of 1. Generally speaking, more advanced skills have higher rank. All skills, when being learned, start from level 0 and have a maximum of 5.

As an example, a player wanted to train to fly a Caracal, a very common ship. To do so they need to have Caldari Cruisers trained to level 2. Caldari Cruisers itself requires Spaceship Command be trained to level 3 and Caldari Frigate trained to level 4. Assuming the player has both prerequisite skills trained to the proper level, they are able to start training the new skill whenever they want, even in the middle of training something else. The primary attribute for this skill is perception and the secondary is willpower. The rank is 5. Using these values, the game has a formula to calculate the training time. With a perception value of 30.8 and a willpower value of 20.9, it would take 2 hours, 17 minutes and 28 seconds total to train.

Therein lies the one greatest difference between EVE Online and other MMOs: no matter how much time one spends playing, their character won't attain skill points any faster. All skill training is real-time based. To fly the most powerful ship in the game, a Titan-class ship, it takes a minimum of about a year and a half of playing. Realistically, since one should attain skills to use a Titan more efficiently for it to be practical, the training time is closer to 3 years. This is in stark contrast to a game like World of Warcraft where a player can feasibly attain max-level epic gear within a month.

One last aspect to skill training is the training queue, which was introduced during the Apocrypha expansion in January of 2009. This feature allows a player to assign skills to be trained that can start within the next 24 hours. This theoretically allows a skill that takes a month or more to train to be added into the queue so long as it starts in less than 24 hours.

EVEMon

In 2005, CCP released its API technology. This technology allowed the download of XML files containing information such as character info, training queue, a character's assets, a character or corporation's wallet transactions and much more. Some of the information is private in nature and thus requires passing in a unique user ID and API key. One important thing to keep in mind is that this API is read-only in nature. A player cannot control any aspects of the game with this API. To change the skill being trained, the player would have to actually log into the game to do so.

One of the first programs to take advantage of the EVE API was EVEMon. Its development continues to this day as a Windows-only application. Initially it started off as a way to monitor the character's current skills. It then added the ability to plan the character's skills for up to several years ahead. Another added feature is the visual skill tree. It allows for easy viewing of a skill and all of its prerequisites in a node-tree form. Within each node is a display of the training time for that particular skill. More recently, the ability to train a group of skills based on certificates has also been added. Certificates are achievements for training a certain set of skills to

certain levels indicating proficiency in an aspect of the game. They in themselves don't provide any benefits.

Methodology

Language

The software was largely written in Objective-C. It's a superset of the C language meaning you can mix and match Objective-C syntax and C in the same line. NextStep heavily developed a library based on that language before Apple purchased the company. Apple took the libraries and further built upon it before releasing it to the general public branded as "Cocoa". A remnant of NextStep's influence can be found within the library's naming convention: every one of Cocoa's classes start with NS, e.g. NSString representing a string class.

While the Objective-C language isn't limited to Apple, it is virtually unused elsewhere. Mostly this is due to the lack of available non-Apple libraries as exhaustive as Apple's. There is a continuing effort called gnuStep that is trying to duplicate the functionality of Apple's libraries so that code compiled for Mac OS X with very little tweaking can just as easily compile on Linux. In doing so, gnuStep is mimicking Apple's naming conventions and uses the exact same names for their methods and variables. While Objective-C is the actual language and Cocoa is the name of Apple's own libraries for the language neither of them really exist much outside of each other and the two terms will be used interchangeably throughout the report.

Apple also offers an alternative, a C++ based programming library known as Carbon. While its use is an option it is more or less considered by developers a

defunct language. Development on it has ceased and all the new technologies created by Apple are Cocoa-only. Several Cocoa technologies used, such as Core Data and Cocoa Bindings, are frequently used within EVE Mac Suite because they provide for much faster application development.

Tools

Unlike Microsoft, Apple doesn't charge any licensing fees or a one-time charge to obtain and use their development software. All of their applications are available via download from their developer website with a simple registration.

Application development primarily starts with Xcode. By itself it's a nearly complete IDE that provides all the tools to build, debug and interact with a code repository. It's designed to handle a multitude of languages, such as Python and Ruby, and can be scripted to meet the needs of a large developer base. It is the only IDE available for Cocoa-based applications and tools.

Development of the GUI is done with Interface Builder (IB). IB is very tightly coupled with Xcode, so much so that a new IBOutlet variable or a method that returns an IBAction created in Xcode will right away appear in IB. Unlike a language such as Java, IB doesn't insert any code for the GUI elements. Instead, all the elements are stored in a NIB (or XIB) file and can be instantiated multiple times. These GUI elements don't behave any differently whether originating from code or a NIB file.

One small addition to IB is the use of BWToolkit [1]. It's a plug-in for IB that adds quite a few new components and is free of charge. Apple's NSSplitView, a view that splits two separate views and allows for resizing between them, was pretty

lackluster compared to BWToolkit's. Other items used were the button bar and sheet controller.

Performance testing is done by two different apps, Instruments and Shark, each serving a unique role. Instruments has a large assortment of tools to help with memory management, most popular being ObjectAlloc. This tool helps track the creation and destruction of objects while the application is running to help discover memory leaks. Shark is a single purpose tool in that it provides CPU sampling of a running application. It's helpful in finding out what process consumes the most amount of CPU time and can even go down to the assembly code level.

Cocoa Technologies

Core Data

> The Core Data framework provides solutions to common tasks associated with object life cycle and object graph management, including persistence. Its features include:
>
> - Built-in management of undo and redo beyond basic text editing
> - Automatic validation of property values to ensure that individual values lie within acceptable ranges and that combinations of values make sense
> - Change propagation, including maintaining the consistency of relationships among objects
> - Grouping, filtering, and organizing data in memory and in the user interface
> - Automatic support for storing objects in external data repositories [2]

Object-oriented applications are known to have three qualities: state, behavior and identity. What Core Data does is simplifying the process for two of the three qualities (state and identity) and adds an entirely new quality: relationships. Developing a Core Data based application is done by including and editing a Core

Data Model. Editing this model is done via a graphical interface. Providing a name, class type and default value easily sets states, i.e. instance variables, of these Core Data objects. Identity is automatically done when the application asks Core Data to instantiate a new object with the predefined default values. All Core Data model objects inherit from NSManagedObject or are otherwise referred to as a managed object.

Although Core Data simplifies the process of creating and managing objects, it does by no means limit the developer. Xcode allows the developer to use the Core Data model to define the classes or gives you the option of creating class files based on the model. From there, these class files can be modified to add behavior.

Lastly, Core Data is designed to handle relationships between model objects. Mapping these relationships is also done graphically. Relationships can be set to be a to-1 or to-many, of which the latter optionally allows a range. One of the great things about the relationship handling is the delete rule. If a managed object is to be destroyed, this object can be optionally set to automatically destroy other objects to which it has a relationship with.

In essence, looking at a Core Data model is almost the same as looking at a class diagram, the exception being methods aren't shown. There are other aspects to this technology. For instance, Core Data allows for automatic persistence storage and an undo manager. It also provides for model versioning to transition from an older model to a new one via a mapping model.

Core Data is composed of several interworking objects when used. As described before, there is the model. At run time, Core Data creates what's known as

a context at run time. The context is a copy of model objects stored in the RAM. The context interacts with a persistent store coordinator. This object is the intermediary between the context and the actual persistent store and performs all sync and save operations. The persistent store can be an XML, SQLite or a binary formatted file. An application has one coordinator per persistent store. A store technically can have multiple models so long as there aren't any class name conflicts but this is highly discouraged. A store can also have multiple contexts, which is often the case for applications that write to Core Data from multiple threads.

Cocoa Bindings

> Cocoa bindings is a collection of technologies that reduces the code dependencies between models, views and controllers, automatically synchronizing views when models change. You can eliminate most of your glue code by using bindings available in Interface Builder to connect controllers with models and views [3].

Cocoa Bindings are an implementation of the observer pattern. It heavily relies on Apple's key-value observing technology. This technology allows for objects to be "bound" to a variable to observe any changes to it. Whenever a proper call to change a variable's value is sent, a notification is sent to all objects that are bound to, it notifying of the change. From there, these objects have an opportunity to perform whatever action necessary.

Bindings can be setup either in Interface Builder or in code. The advantage of using IB is that it's much simpler and practical for UI elements that never need to have the variable its observing to be unbound. One doesn't need to dig through a

large amount of code to discover what action a binding is performing. They can be set much faster and thus are far more efficient to work with.

Doing it through code is a bit more complicated as not only is there code to do the binding and unbinding, specific delegate methods have to be implemented on the observing object. One advantage of going with this method is that bindings are far easier to debug; any exceptions with bindings done in IB are only listed in the console. One can step through the previously mentioned delegate methods via a debugger if one chose to bind with code. Another advantage about using code is that non-UI elements can be bound together.

Core Animation

> Core Animation is a collection of Objective-C classes for graphics rendering, projection, and animation. It provides fluid animations using advanced compositing effects while retaining a hierarchical layer abstraction that is familiar to developers using the Application Kit and Cocoa Touch view architectures.
>
> Dynamic, animated user interfaces are hard to create, but Core Animation makes creating these interfaces easier by providing:
>
> - High performance compositing with a simple approachable programming model.
> - A familiar view-like abstraction that allows you to create complex user interfaces using a hierarchy of layer objects.
> - A lightweight data structure. You can display and animate hundreds of layers simultaneously.
> - An abstract animation interface that allows animations to run on a separate thread, independent of your application's run loop. Once an animation is configured and starts, Core Animation assumes full responsibility for running it at frame rate.
> - Improved application performance. Applications need only redraw content when it changes. Minimal application interaction is required for resizing and providing layout services layers. Core Animation also eliminates application code that runs at the animation frame-rate.

- A flexible layout manager model, including a manager that allows the position and size of a layer to be set relative to attributes of sibling layers.

Using Core Animation, developers can create dynamic user interfaces for their applications without having to use low-level graphics APIs such as OpenGL to get respectable animation performance. [4]

Originally, Core Animation wasn't in the plans to be used. It was only used in rendering the skill tree, which will be discussed later on. The original intention was to use the standard Cocoa NSView subclasses. However, there immediately came problems of redrawing. Whenever the view that contained the NSView subclasses representing the different nodes on the skill tree were scrolled around, there was so much fragmenting with the redrawing and such a high CPU usage that going this route immediately became unfeasible. The resultant tree would be drawn with vertical or horizontal chunks missing, depending on the direction of the scroll.

The biggest advantage of Core Animation from a programming perspective was its simplicity in programming. Given the name of the technology, one would suspect that there would actually be animation. However, the technology can be just as useful for static drawings, too. The fact that it's such a lightweight drawing architecture allows a user to browse through drawings at a very quick rate.

Dev Blog

By and large, a dev blog is used to communicate to the audience of the progress of the application. Alternatively, a forum could have been used but EVE Online already provides a Macintosh and an API forum. On top of that, there wasn't much anticipation for other user posts other than my own. Even though this

program is open source, it isn't uncommon for a company to have a dev blog of their own. A well known one is Google's.

The intention of this dev blog is two-fold. First, it's to let users of the software know the status of the app, including upcoming features. Second, it's to truly enhance the experience of developing an open source application via input and interactions from the users.

Open Source

The idea of something being open source can mean a variety of things, largely dependent on the license chosen. A BSD license is the easiest to work with for other developers. Others can be much more restrictive. A very common one is the GPL public license, which comes in three different versions. For this project, I chose a GNU public license version 2 [5]. I generally wanted a license that allowed people to use my code so long as such people freely distributed their source but still retain ownership of the original project.

Requirements

Use Cases

It is assumed that the main user is the game player because this application does not have any practical use for non-EVE Online players.

Character Management

- **Main Actor:** Player
  Goal: Add new character to monitor.
  Conditions: Character isn't already being monitored; Internet connection is available.
  Outcome: Application loads information of desired character into a new

view.

Scenario:

- Player prompts application to add a new character.
- Player enters in their user ID and API key.
- Application connects to EVE API to download character list.
- Application loads a list of characters for the user ID and API key.
- Player selects desired character(s).
- Application connects to EVE API to download data for character(s).
- Application stores information for character(s) into database.
- Application creates new view(s) for character(s).
- Application loads information for character(s) into view(s).
- Application sets update timer for 1 hour or earlier if necessary.

Exceptions:

- Internet connection is unavailable.
  - Prompt player that application is unable to connect to EVE API.
- EVE API is unavailable.
  - Prompt player that application is unable to connect to EVE API.
- Player enters in a bad user ID/API key combination
  - Prompt player of incorrect user id or API key use.
- Player cancels adding new character.
  - Return to the main application's window with nothing done.
- Player attempts to add an already monitored character.
  - Prompt player that character is already monitored. Return to the main application's window.

- **Main Actor:** Player.
  Goal: Remove a monitored character.
  Conditions: Character isn't already being monitored.
  Outcome: Application closes the view for the character and deletes all information of that character from the database.
  Scenario:
  - Player selects view of desired character to remove as the current active one.
  - Player tells the application to delete the character.
  - Application prompts the player if they're sure they want to remove the character.
  - Player confirms the prompt.
  - Application, if necessary, closes the skill planner window for the character.
  - Application closes the view of that character.
  - Application removes information of that character from the database.
  - Application selects another existing character as the active one or blank if there is no other characters being monitored.

  Exceptions:
  - Player cancels the delete character confirmation.
    - Return to the main application's window with nothing done.

- Player attempts to remove a character with no active characters being monitored.
    - Menu option to remove a character is unavailable.

- **Main Actor:** Player.
  Goal: View different monitored character.
  Conditions: Player has more than one monitored character.
  Outcome: Application will load the view of desired monitored character.
  Scenario:
    - Player prompts application to show list of monitored characters.
    - Application presents list of monitored characters with the currently active one highlighted.
    - Player selects desired character to view.
    - Application updates main window to show the newly selected character's information and makes that character current.
    - Application highlights the new active character in the list of all monitored characters.

  Exceptions:
    - Player has only one character.
        - Player has no other characters to select and thus nothing happens.
    - Player selects current active character to be the newly active one.
        - Nothing happens.

- **Main Actor:** Player.
  Goal: Update a monitored character.
  Conditions: Internet connection is available.
  Outcome: Application will synchronize with latest character information from EVE API.
  Scenario:
    - Player selects view of desired character to update as the current.
    - Player tells the application to reload the character.
    - Application connects to EVE API to download data for character.
    - Application updates character's information in the database.
    - Application loads new information for character into view.
    - Application resets update timer for 1 hour or earlier if necessary.

  Exceptions:
    - Internet connection is unavailable.
        - Prompt player that application is unable to connect to EVE API. Character monitoring resumes as before.
    - EVE API is unavailable.
        - Prompt player that application is unable to connect to EVE API. Character monitoring resumes as before.
    - Player cancels updating.
        - Character monitoring resumes as before.

- **Main Actor:** Monitored character.
  Goal: Update character.
  Conditions: Internet connection is available; application launched, skill training has completed or 1 hour since last update has elapsed.
  Outcome: Application will synchronize with latest character information from EVE API.
  Scenario:
    - Application connects to EVE API to download data for character.
    - Application updates character's information in the database.
    - Application loads new information for character into view.
    - Application resets update timer for 1 hour or earlier if necessary.

  Exceptions:
    - Internet connection is unavailable.
      - Prompt player that application is unable to connect to EVE API. Character monitoring resumes as before.
    - EVE API is unavailable.
      - Prompt player that application is unable to connect to EVE API. Character monitoring resumes as before.
    - Player cancels updating.
      - Character monitoring resumes as before.

Skill Planner - General

- **Main Actor:** Player.
  Goal: Launch skill planner.
  Conditions: An active character.
  Outcome: A new window is launched with a skill planner for the currently active character.
  Scenario:
    - Player, if necessary, selects the desired character.
    - Player prompts the application to launch skill planner for that character.
    - Application creates a new skill planner window for that character and moves it to front of screen.
    - Application selects first skill plan on the list to be active.

  Exceptions:
    - No characters are available.
      - Skill planner button is unavailable.
    - A skill planner window is already open for the current active character.
      - Application brings the character's skill planner window forward.
    - Player opens skill planner for the first time for this character.

        - Application creates a new, blank skill plan with a default name for this character and selects it as active.

- **Main Actor:** Player.
  Goal: Close skill planner.
  Conditions: An open skill planner window.
  Outcome: The desired skill planner window is closed.
  Scenario:
    - Player prompts the window to close the way windows in Macs are traditionally closed.
  Exceptions:
    - None

- **Main Actor:** Player.
  Goal: View skill tree or description for a skill.
  Conditions: An active skill planner window.
  Outcome: A skill tree is shown for the selected skill.
  Scenario:
    - Player, if needed, opens up the skill group in whose list the skill is contained within.
    - Player selects the desired skill to view in the browser.
    - Application will draw skill tree for the selected skill elsewhere within the skill planner window.
  Exceptions:
    - Player selects a skill group rather than an actual skill.
        - Nothing gets drawn. If another skill tree is drawn from a previous viewing, it still remains.

Skill Planner - Skill Plans - General

- **Main Actor:** Player.
  Goal: Create a new skill plan.
  Conditions: An active skill planner window.
  Outcome: A new skill plan is produced for the currently active character.
  Scenario:
    - Player, if necessary, opens the list of current skill plans.
    - Player prompts application to add a new skill plan.
    - Application creates a new, blank skill plan with a default name and makes it active.
    - Application adds this new skill plan to the list of available skill plans and makes it active.
  Exceptions:
    - None

- **Main Actor:** Player.
  Goal: Remove a skill plan.
  Conditions: An active skill planner window; desired skill plan selected.
  Outcome: A new skill plan is removed for the currently active character.
  Scenario:
    - Player prompts the application to remove skill plan.
    - Skill plan and all skills within the plan are removed from list and database.
    - Application selects another skill plan to be active.
    - Application refreshes the view to the newly active skill plan.

  Exceptions:
    - Player attempts to delete the only skill plan available.
      - Option to delete the plan is unavailable. At least one active skill plan must exist at all times.

- **Main Actor:** Player.
  Goal: Rename a skill plan.
  Conditions: An active skill planner window; desired skill plan selected.
  Outcome: The desired skill plan is renamed.
  Scenario:
    - Player enters in the desired name of the skill plan in name box.
    - Player performs another action, whether pressing the 'enter' key or clicking elsewhere, to update the name.
    - Application updates the name of the skill plan in the list of skill plans and the database.

  Exceptions:
    - Player enters in a name of an existing skill plan.
      - This does not produce errors as skill plans are referenced by name. However, it will likely cause player confusing.
    - Player enters a null name.
      - The skill plan's name will revert to its previous value.
    - Player enters in a really long name.
      - The name will simply be truncated in the list of skill plans but otherwise there are no errors.

Skill Planner - Skill Plans – Planned Skills

- **Main Actor:** Player.
  Goal: Add skills to skill plan.
  Conditions: Desired skill plan selected.
  Outcome: A skill and all of its prerequisite skills are added to the skill plan.
  Scenario:
    - Player locates the desired skill in the skill browser.
    - Player drags and drops the desired skill from the browser into the planned skills table.

- Application calculates the tree of prerequisite skills (if any) based on the character's current trained skills and if the prerequisite skills are already in the skill plan.
- Application places the skills in the proper location, starting with the prerequisite skills and working down. It will do so by providing one listing per skill per level. If a prerequisite skill is already in the plan from a previous placement, the application will continue with placement after the location of that prerequisite skill. Full details on the process will be discussed in the design.
- Application updates the skill planner table.
- Application updates the database of the changes.

Exceptions:

- Skill plan already has the selected skill added to the plan.
  - Application will attempt to add the next available level to the plan. If there is already a level 5 listing of the skill, the application will do nothing.
- Player tries to add a skill or a skill that contains a prerequisite in a position before a prerequisite of the skill being added.
  - Application will add it after the location the prerequisite.
- A learning skill is added to the plan.
  - Application recalculates training time for all skills located after the learning skill.

- **Main Actor:** Player.
  Goal: Remove skill from skill plan.
  Conditions: Desired skill plan as well as skill is selected.
  Outcome: A skill and all skills to which the selected skill is a prerequisite of are removed from the plan.
  Scenario:
  - Player locates the desired skill in the skill planner table.
  - Player prompts application to remove the skill.
  - Application goes through each skill whose location is after the selected one and checks if the selected skill is a prerequisite of that skill. If it is, that skill is placed on a delete list.
  - Application deletes selected skill and all skills on the delete list.
  - Application updates the skill planner table.
  - Application updates the database of the changes.

  Exceptions:
  - No skill is selected or no skills are present in the skill table.
    - Button to delete skill is unavailable.
  - Player deletes a learning skill.
    - All skills whose location is after the learning skill have their training times recalculated.

- **Main Actor:** Player.
  Goal: Move skill in skill plan.

Conditions: Desired skill plan selected.
Outcome: A desired skill is moved within the skill plan.
Scenario:

- Player locates the desired skill in the skill browser.
- Player drags and drops the desired skill from within the table to anther location in the table or player presses the up or down buttons to move it one location at a time.
- Application updates the skill planner table.
- Application updates the database of the changes.

Exceptions:

- Player moves a skill to a position located after a prerequisite of the skill.
  - Application positions the selected skill right above the prerequisite skill.
- Player moves a learning skill.
  - Application updates training time of all skills.

Testing

Testing is one of the more difficult aspects of this application. One obvious need is to have several characters available to test, which costs a minimum of $15/month for an account with three character slots. Another obvious one is an Internet connection to download the API XML data. While this by itself isn't a problem, the API server isn't always available. It's guaranteed to be down at least an hour every day due to CCP's regularly scheduled maintenance. On the bright side, the downtime is during a time of day with the least amount of traffic.

The more difficult problem with the XML data is that some XML data is difficult to produce and duplicate. For example, trying to test the skill queue with ten or more skills in the queue may be impossible if I don't have that many skills readily available to train. This problem extends to any one skill that another player has a problem with that I've either already trained or am unable to do so for a long time. The solution to this problem is to somehow allow a way for the application to manually use XML files rather than to automatically download them. Going this

route, though, was not built into the application's user interface but rather required compiling the application using debug pre-processors written into the code.

Releases

When the project was up on SourceForge, I heavily relied on build numbers as my release numbers. I could have gone with the traditional 1.0 and incremented it from there but I felt that such a system was dated. It's hard to tell at what point does a piece of software's version go up an integer. Build numbers provided a clean, easy to understand way of referencing without feeling tied down to a system. Not every build was actually released, however.

Design

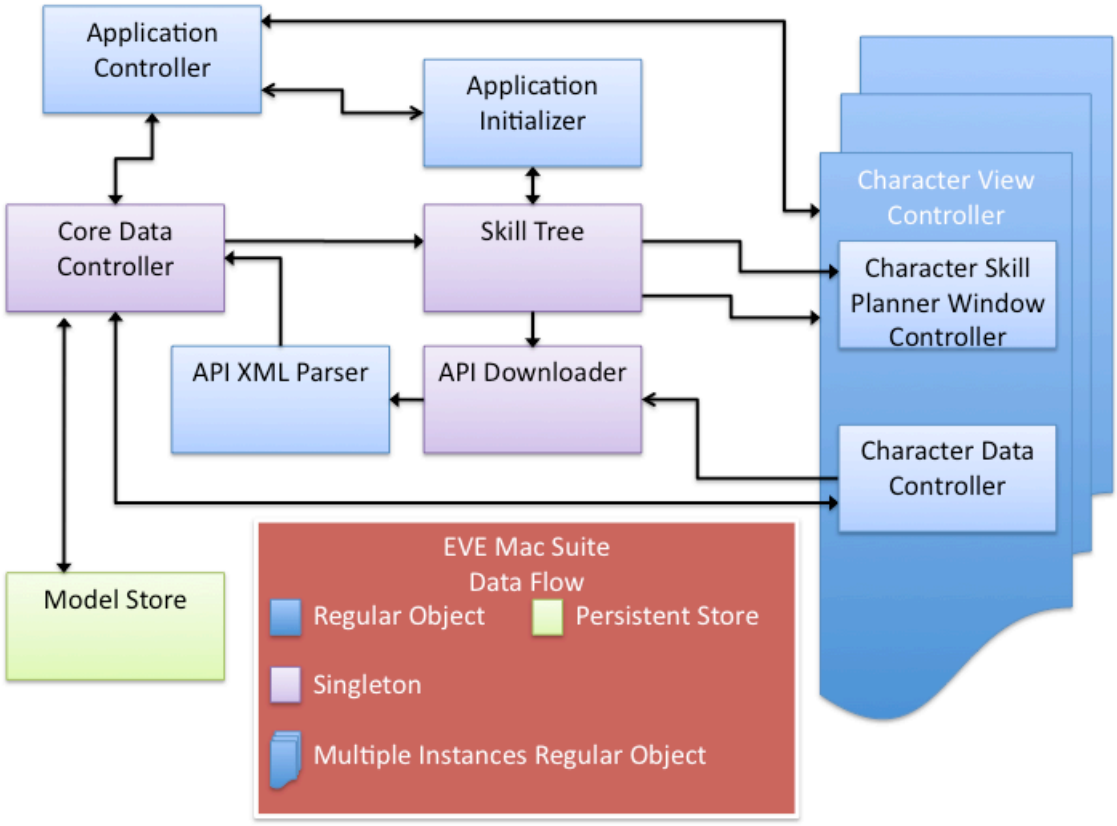


Figure 1 Data flow with the application components.

Application Components

From a functional standpoint, the final application is broken down into the following parts, as shown by the data flow diagram:

- Application Controller – This portion maintains the main window's view and tracks the active characters.
- Application Initializer – Registers any value transformers and loads the saved characters when the application gets launched.
- API Downloader – This part is responsible for handling a series of XML file and picture downloads. It would then pass the XML files to a parser.
- API XML Parser – This portion of the program would take the XML files and parse through the data to store into an XML database.
- Core Data Controller – This portion bridges the persistent Core Data store with the application
- Core Data Store – There is one database store for the entire application.
- Skill Tree – This is part of the model that gets loaded at application launch. The information within it is static and used throughout the application.
- Character Data Controller – There is one per character. The data handler keeps a timer as to when the next XML update is to be done. It is directly responsible for interacting with the Core Data store pertains to the character's information. It also sends out notifications

whenever changes in the character's info are made. Notably it actively monitors the current skill in training and the training queue.

- Character view controller – There is one per character. This portion interacts with the main application's view and the character's data handler.
- Character Skill Planner Window Controller – There is one per character. This portion is responsible for drawing the skill node tree, show description of each skill, calculate training times and maintain the plan queue.

Data Flow

To help understand how all these interact, a description of the flow is warranted. Assuming this is the first time the application gets run, the application initializer first connects to the EVE API to download the skill tree XML. The downloader makes a request, performs the download and passes it on to the skill tree parser. The parser goes through the XML file and creates a Core Data context of the data. A tree gets built of every skill group, skill and skill requirements. The application now is ready to be used by the player.

The player then prompts the application to add a new character. They enter in the user ID and API key unique to their account. The application controller (AC) queries the EVE API and brings back a list of characters under that account. The player, in this case, selects just one character to monitor. The AC then creates a character view controller (CVC) for that character. The view controller right away creates a character data controller. The data controller makes an API request for a

character sheet. Like the skill tree, the process is sent through the downloader and then the XML parser. In this step, the character's learned skills are linked to the skill tree. The XML parser stores the information of that character into Core Data context.

When the character sheet is finished, two things happen. First, the CVC notifies the AC of it and the AC places that view controller to the main window. Second, the character data controller then makes a request for the character's image. Rather than going through an XML processor this time, the image downloader saves the data to the context directly. Lastly, the data controller makes a request for the skill queue. The skill queue follows the same route as the character sheet.

Because the UI elements in the CVC use Cocoa Bindings and are bound to the Core Data values of the character, no notification needs to be made to the view controller that the data controller is finished. The CVC will automatically show the skill in training as well the character portrait.

Now, say the player wants to use the skill planner. The player prompts the application to launch it. The AC tells the active CVC to launch a skill planner window controller (SPWC). The character view controller passes along its character data controller to initialize it. Just like the CVC, the SPWC uses Cocoa Bindings extensively. As soon as it is launched, the SPWC right away is populated with values of the bound character. However, the SPWC is also bound to the skill tree. Unlike most of the rest of the application, the functionality of the SPWC is almost entirely self-contained and thus doesn't require a descriptive data flow. The only exception is its ability to write to the Core Data context of any skills they add to the skill plan.

Figure 2 Main window with an attached character view controller.

Application Controller

EVE Mac Suite runs with principally one main window. This window contains placeholder UI elements whenever there aren't any characters listed. It's split up into 3 parts. First, there is the upper section that contains a toolbar. This toolbar contains sections to show the list of characters, to reload a character's data and to launch the character's skill planner. The button to show the list of characters toggles a drawer of the list, the second part of the main window. Each listing in the drawer shows the character's portrait, the current skill being trained and the time to completion.

The last section is more of at the heart of the application. While it's completely blank when the application gets first launched, the portion underneath the toolbar is the location each character's view controller gets displayed. Only one view controller gets loaded into this position at a time.

Within the application controller there is also the functionality to manage characters. The ability to add and remove characters is controlled by menu

functions. The application controller launches a new window to enter in a user ID/API key whenever the player asks to add a new character but otherwise nothing unique happens whenever a character is deleted. The application controller has direct access to the Core Data controller (as does much of the app) to add/remove character.

The application controller has a tight coupling with the application initializer, which will be discussed further on. Besides creating, deleting and managing them, the application controller has no direct access to a character's view controller. Whenever a different character gets selected, the application controller releases the old one and loads up the new one. While the data flow diagram does show a two-way connection from the application controller to a character view controller, it's not quite the tight coupling of the initializer. The application controller sends commands to the view controller via direct methods whereas the character view controller sends calls back via delegate methods.

Application Initializer

As one can imagine, the application initializer is the first item that gets run when EVE Mac Suite launches. The class structure is setup in such a way that both the app controller and the initializer have knowledge of each other's public methods and call them, hence the tight coupling. Once the initializer gets run, it is never needed or called on again.

The very first order of business when the object gets created is to setup the value transformers. These are meant for converting one type of value to another in bindings and initialization has to be done when custom ones are being used as they

are frequently in this app. Next it launches the EVE Skill tree static information, which will be discussed later on. Lastly, it opens up the application's preference file (a file that persistently stores the list of characters along with any other preferences) to load up the characters and initializes them one by one. The initializer doesn't actually create the character data controller. Doing so would actually duplicate functionality. Instead, it passes the character data to the application controller to do the actual creation.

API Downloader

The downloader is in charge of connecting to the EVE API, downloading the appropriate XML document by, if necessary, providing a user ID/API key in an HTTP POST request. It's a singleton instance because the intention is not to send off several simultaneous HTTP requests but to download each XML one at a time. Requests are made with a special downloader request object that provides the character's information, the type of download (character list, character sheet, skill tree, etc) and what object will be in charge of handling the downloaded XML API. Downloads are handled in a queue fashion where as soon as one download is finished, another one begins.

All downloaded XML files get stored locally in EVE Mac Suite's Application Support folder. While it technically isn't needed after it gets processed, it's ideal to have around as it helps in debugging possible problems.

The downloader itself went through several different iterations. Cocoa provides two different ways to download: synchronously and asynchronously. With the latter option the thread gets completely locked up. The former allows the

download to come through as a serious of delegate calls that allows for tracking progress. While it may seem that going with synchronous downloads would be the ideal way to go, a design decision was made to have the downloading and XML processing be done on a thread separate from the main thread. While I was able to work with multiple threads on a series of function calls, I was unable to find out how to keep a thread alive to be able to receive a series of function calls. In the end, an asynchronous download option on a background thread was the decision made. The trouble with this method, though, is the difficulty in canceling a download. The only real way to do it is to kill the actual thread.

There is one special subclass of the regular downloader known as the image downloader. The EVE API, given a character ID and one of two different pixel sizes in a POST request will provide an image rather than an XML download.

API XML Parser

The XML parser acts as a mediator between the EVE API and the Core Data store. Cocoa provides two different methods of XML parsing: tree and stream. The tree method was used for a considerable amount of time before a switch to the stream method. With the tree method XQuery commands were used. These proved to be far too slow so Cocoa's native NSXMLNode and NSXMLElement objects were used. However, even these proved to be a problem as the speed improvement, while noticeable, still wasn't great. Worse, using such methods required that the XML nodes stayed in the same position every time. This method would run into difficulties whenever CCP made a small change to its API.

Much later on, a switch to the stream method was made. With stream XML parsing, the application takes a top-down approach to reading the XML. Whenever various XML items are read, various calls are made to the delegate with different methods for each of the different types. The advantage to this method was that it was incredibly faster. Whereas before when the program launched for the first time and a skill tree xml had to be downloaded, it would take upwards of 10 to 15 seconds to process it. With stream processing it was cut down to 2 to 3. The downside was there was quite a bit of tedious and almost redundant code to write, particularly with the character sheet.

Excluding the skill tree and brand new characters, every XML parse replaces existing data within the Core Data store. Methods are built in to retrieve the existing objects for whom will have their data replaced.

Core Data Controller

Since there is only one Core Data store and Core Data is accessed throughout the application, the Core Data controller is a singleton instance. The controller only creates a single context despite the fact that there are multiple threads. Whenever the XML parser writes to the controller, all writing to the store is blocked. Otherwise, all other writing is done on the main thread. Both the main thread and background threads can read from a Core Data context without problems.

Core Data Store

Managing the Core Data Store aspect of this application was by the easiest part of the development process. When the development of this application started, an XML format was chosen for the store to be saved as. This provided an easy way to

browse through what actual values were being saved and track relationships when debugging. XML stores, however, are the slowest of the three. When compared to an SQLite or a binary store, it can often be 2 to 3 times slower to both read and write from.

Later on, a decision to go with an SQLite store was made. This, along with XML stream parsing, provided a near instantaneous application launches and character updates. SQLite stores, however, had one significant shortfall: they didn't support fetched properties. Fetched properties are properties much like an attribute or relationship in a Core Data model. However, the user can create queries, much like an SQL query, to request data elsewhere within the context for a model object. Due to the complexity and entirely different technology of these queries, Core Data is unable to translate them to SQLite queries. The only other option available was a binary store. Fortunately, there wasn't any significant loss of speed switching from SQLite to binary.

Skill Tree

The skill tree, much like the Core Data controller, is accessed throughout the application and is used constantly. The skill tree is actually a portion of the Core Data model that's initially loaded up into the context, regardless if there aren't any characters monitored. Part of the reason for having one instance is that it cuts down on redundancy, which in turn uses less processing power. Structurally, it is built to be a tree of skill groups and within them are the skills. Within each skill, besides the expected attributes, such as name, description and rank, there are 0 to 3 skill requirement relationships. These skill requirement objects then connect to a

required skill and have a level attribute. The skill tree directly interfaces with a character's dynamic data.

Character Data Controller

A single, unique data controller is created per monitored character and is initialized via an NSDictionary (a glorified hash map) passed in from the enclosing view controller. The data controller, while mediates between the Core Data controller and the UI, is also responsible for several other functions. Whenever a character is reloaded (or first loaded, for that matter), the data controller first requests a character sheet download and parse. If this is a first-time character load, the controller then requests for the character image. The last request made is for the character's training queue.

Upon finishing the character's skill queue, a timer is initialized to fire in an hour. The timer can fire earlier, though, if certain conditions are met. One other timer the controller starts is the skill points update timer. This frequency of this timer firing is based on a formula, usually in the range of 1 to 3 seconds. Calculating training time starts off with an array of numbers, 250, 1414, 8000, 45255, and 256000. These numbers represent the amount of skill points training required for a rank 1 skill. Rank is a multiplier so, for example, with rank 3 skills these values are tripled. The time, in minutes, to calculate the training time for a skill is equal to the following formula:

$$\text{training time (minutes)} = \frac{\text{(skill points to next level)}}{\text{(primary attribute value)} + \frac{\text{(secondary attribute value)}}{2}}$$

All other formulas, such as the finish date and seconds per skill point can be calculated based off of the above formula. The data controller calculates the exact values based on the current skill being trained and the character's attributes. With every firing of this timer, the character gains a skill point in the skill being trained. This is, within a few seconds, supposed to by in sync with the skill actually being trained in game. In a sense, EVE Mac Suite guesses the current amount of skill points a skill has. At every update, the application resynchronizes with the EVE API to ensure accuracy of the data.

Back to the character sheet update timer, there is a chance that this update timer can fire before the usual frequency of once every hour. This happens when the skill being trained is scheduled to finish within the hour. When the application adds enough points to a skill for the skill to be at the next level, there is a pause for 5 seconds. This is because there is an occasional in-game delay for training completion. After, another update is done and once again the API update timer is reset for one hour.

Character view controller

There exists one character view controller per character and is a subclass of NSViewController. This class, introduced in Apple's Leopard release, is similar to an NSWindowConroller but instead controls NSViews. Passing in an NSDictionary containing the character's account information creates an instance of this controller. The controller doesn't actually do anything with the NSDictionary except pass it along to its data controller. Because of the extensive use of Cocoa Bindings, this view

controller actually has some of the least amount of functionality. In fact, half the lines of code within this view controller's class are to support the bindings.

Within the actual NIB file are placeholders for the character's values and portrait. There is, however, an NSTabView (as the name suggests, a view that contains tabs) that has two tabs, one for the current skills of the character and one for skill queue.

The only other functionality this view controller provides is to pass commands between its data controller and the application controller. Specifically, it tells the data controller to perform an update whenever the player prompts the application to do so or tells the application controller whenever the data controller finishes an update. Finally, the character view controller is in charge of managing the skill planner window controller. It opens it upon the player's request and releases it upon close.

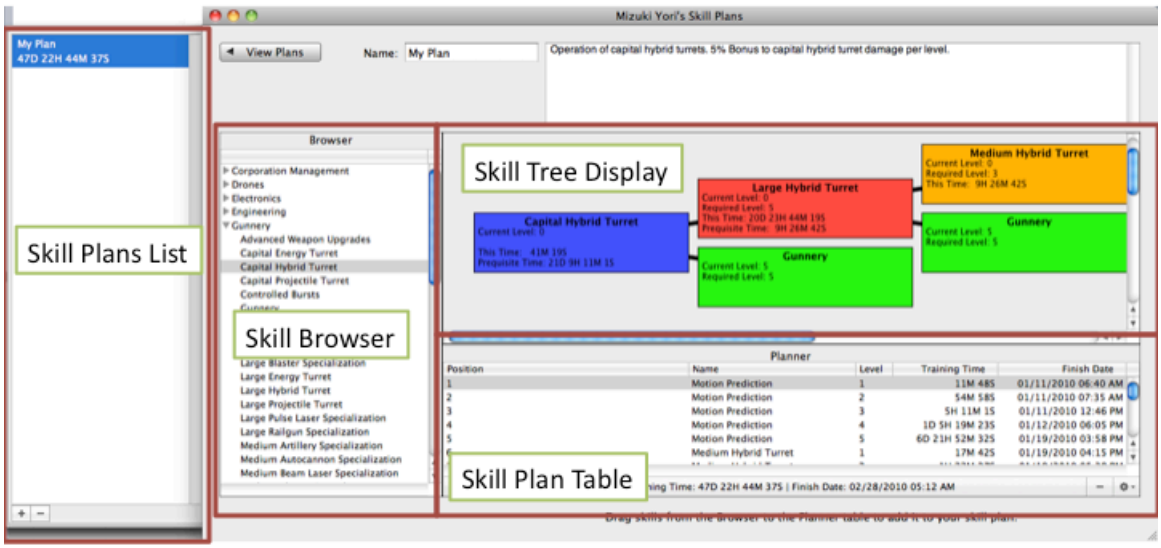


Figure 3 The four major components of the skill planner window.

Character Skill Planner Window Controller

The skill planner window controller is one of the most intricate parts of the application. So much so that it can be broken up into 4 major sections. It's composed of the skill plans list, the skill browser, the skill tree display and skill plan table.

The easiest is the skill plan list. Much like the characters list in the main window, this list is located in a drawer toggled by a button. Each listing shows the name of the plan and the total training time of the skill plan. Each listing here is bound to the character's skill plans.

The skill plan list, located in the upper left corner, is an NSOutlineView that lists all the groups of skills and within them the actual skills. Such a view is ideal for displaying a tree of data. Because of the way Cocoa Bindings work where traversal down the tree required identical method calls, a unique solution had to be devised to allow everything to bind correctly using Cocoa Bindings. Within the NIB is an NSTreeController, a proxy object for tree structures, that's bound to the skill tree singleton object.

The Core Data model contains a "Group or Skill" object that happens to be subclassed by the "Skill Group" and "Skill" object. While within the code the superclass doesn't get used, it provides a common set of attributes between the skill groups and skills. Not shown on the Core Data model but is shown within the classes for these objects is a category (objective-c terminology for an extension of a class) that has a single method named identically in both subclasses but return a different array of items.

Next is the skill tree display. This view is bound directly to the selected item in the skill browser. Whenever the player clicks on a skill, a three-step process

occurs in drawing the tree. First, the application goes through the static skill tree to get the tree of skill requirements for the selected skill, generating a new tree in the process. The application then goes and checks against what is already trained and updates this new tree. Lastly, the tree gets drawn out in a left to right manner, branches first, using Core Animation.

The drawing code, while simple in comparison to other options such as openGL, is still quite intricate. The nodes in the newly created tree are used for node layer objects. Everything in Core Animation is drawn with layers, hence the name. Though the drawing starts from the right side, this is actually the end of the tree rather than the beginning. The layer onto which these node layers are drawn against is separated into five columns (5 is the largest depth of all the skill trees).

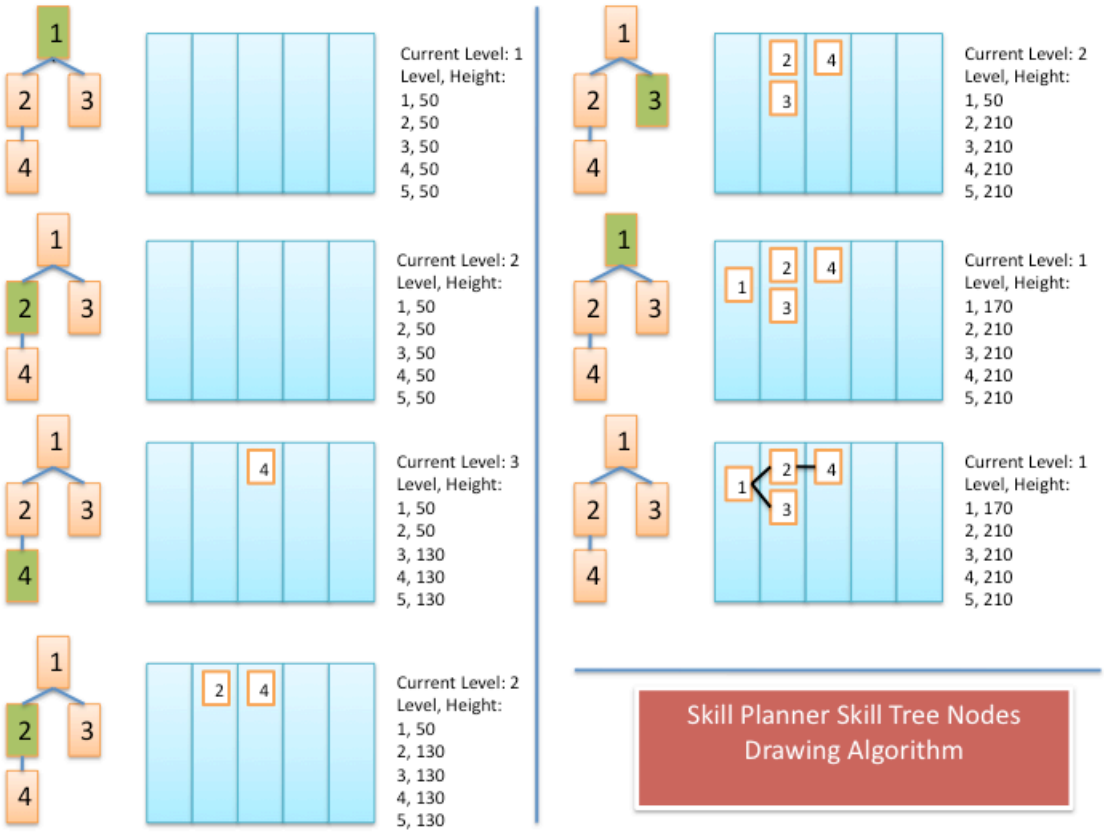
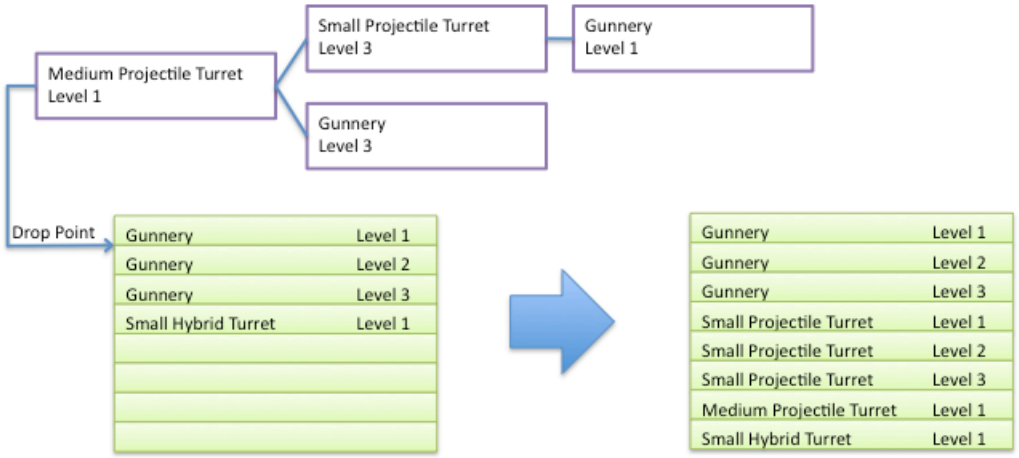


Figure 4 Tree node drawing algorithm sequence.

The algorithm itself is too complicated to summarize within a couple of paragraphs. Figure 4 above essentially shows step by step how it works. The current level represents the column being drawn into. The "level, height" is an indicator of the minimum height to draw the next node at that level. What's notable is that when a minimum height is set for one level, all subsequent levels, if they haven't reached the minimum height, are adjusted to that minimum also. This algorithm lays out a tree structure that's simple and easy to look at.

Lastly, there is the skill planner table. The way it works is by a drag and drop operation from the skill browser to the skill planner table. Like the skill tree display, a node tree object is generated for the skill based on what the character has already

learned. The skills are then added to the table with the prerequisite skills first and the desired skill at the end. Individual levels of the skills are listed. For example, a player wanted to learn to fly Caldari Battleships and they have Spaceship Command trained to level 5 and Caldari Cruiser trained to level 2. In this case, the order of skills added would be Caldari Cruiser level 3, Caldari Cruiser Level 4 and finally, Caldari Battleship. When the skills are finally added, a summary of the total training time and finish date are shown.



Skill Planner Table Mid-Drop Example

Figure 5 Dropping overlapping skills in the middle of a listing.

Things get a bit tricky when skills are placed somewhere other than an empty list or not at the end of one. In this case, there is an extra step that checks to see if a prerequisite skill required for the dragged skill already exists within the

table. Also checked is whether a prerequisite skill of the dragged skill within the table is of a lower level. Figure 5 illustrates what happens. Similar logic applies whenever an individual listing is moved up and down. A check is made to ensure that the position is valid or otherwise ignores the player's commands.

The last point on this table is the effect of learning skills. When added, it is assumed that the player will have their attributes increased as soon as they are learned. Therefore, any skill listed after the learning skills will have to be presumed to use the potential new attribute values. Thus, whenever changes are made with any skills located after the first listed learning skill, a total recalculation is done.

Design Challenges Faced

Repository

Although not quite a design challenge per se, I came to discover that SourceForge is absolutely hideous with its release system. Frequently it times out when attempting to upload files. The menu system to maintain release versions isn't intuitive and once committed, it can't be undone. Eventually, after so much frustration, I moved everything over to Google Code.

Time zone

The training times in the EVE API are all in Greenwich Mean Time. Cocoa's calendar and date API had a sufficient set of methods that allowed for relative ease of converting a string value to a date object. The difficulty came with time zones. The application is expected to run in multiple time zones but GMT doesn't use one. Figuring out Cocoa's handling of time zones was definitely a tricky feat. A date object had to be instantiated with a GMT time zone but display in the local time. This

seemingly easy task took over eight hours of development to learn to parse the date string, instantiate in the proper time zone and use values within the current time zone and testing it each step of the way.

Cocoa Bindings

Cocoa Bindings were a blessing and a curse. No doubt they did remove what would amount to a massive amount of glue code. And just like regular code, problems arose during testing. Unlike code, however, Cocoa Bindings couldn't be debugged; there is no possible way to step through the values being given to the bindings and the output produced. In the end, quite a bit of trial and error needs to be done to pinpoint bindings issues. Occasionally the console or debug log provided hints as to what the problem could be but breakpoints are of no value.

Skill In Training

For a long time, a Core Data modeling decision concerning the concept of a "Skill in Training" ended creating far more problems than it solved. Conceptually, a player has a set of trained skills that are a subset of all the skills. The idea was to use a skill in training for skills to which a player has points in and a static skill for all others. This wasn't a problem when the original table featuring all the trained skills was built. It became somewhat of a problem when the skill queue was built. It became a nightmare when the skill planner was being built. This problem was further compounded by the quirky concept of a "Skill and Level" object that was primarily intended for the skill planner and mapping prerequisite skills.

The problem arose in the fact that they were modeled differently. For a skill in training to know what skill group it's part of, it first has to go to its skill

relationship and then group relationship. A static skill, on the other hand, can go directly to its group relationship. In the skill planner, objects representing all the three different types of skills were used. This eventually led to a massive amount of type checking, a violation of object-oriented principles.

Eventually, the static skills remained static. The skill and level objects were removed and a skill requirement object took its role as an intermediary between a skill and its prerequisite skills. Separate entities for the skill queue and skill plan items were created that both linked to the learned skill. Although in the end quite a few more Core Data model objects were created, these objects were much better suited to their specific task.

Core Data Locking

This problem remains an unsolved problem. A Core Data context can be locked for writing. This is ideal for multithreaded applications. In EVE Mac Suite, when a character is actively training a skill, the skill points they earn automatically get written to the context on every tick. This, however, needs to be paused whenever the application is doing an API update. On occasion, the locking doesn't happen and race conditions occur. No amount of testing was able to discover the source of the problem despite using attempts such as trying to acquire a lock on both threads.

Conclusions

In the end, the work to create this application was highly enjoyable. Being a single developer, there were things in the development process that were skimmed over as it added too much administrative work. These items included change logs,

creating an installer and maintaining proper versioning. The dev blog, on the other hand, was used quite frequently and was a great way to communicate my progress. Player input, though, mostly came from EVE's official online forum.

The experience of developing an open source program was unlike what I anticipated. In the summer of 2009 I requested support from the EVE Mac community to further develop this program on the dev blog. Granted, the pool of EVE Online players who know Cocoa is likely to be extremely small but I still didn't get a single response for non-programming work. The few ISK donations that I did receive were very small (ISK is the currency used within EVE Online). In a sense, I discovered that working on an open source program should be the rewarding part of the process, that programming is its own reward.

Future Work

While quite a bit of time and effort has been put into this project, there still remain quite a few things left from my original goal. For example, the Core Data model contains objects that represent certificates but are never used anywhere within the application. Being able to add skills to a plan simply by adding a certificate would be a powerful feature.

The skill planner, while it's there, doesn't compare to the power of EVEMon's with its ability to provide suggestions, sorting and integration with the EVE Fitting Tool application. One possible improvement would be to brain storm a user interface that would make the job of skill planning much more intuitive than EVEMon's.

A feature that I wish I had time to implement was the fitting tool. Such an addition would have been A LOT of work to do, such as parsing the database dump. It would also require understanding of EVE mechanics far beyond what I currently know.

Lastly, there's the financial planner. This was a feature that was hoped to be added but one that's realistically unattainable. This feature would likely not sync with the other intended features of the application much.

Appendices

References

[1] “BWToolkit: Plugin for Interface Builder 3.” Internet: http://www.brandonwalkin.com/bwtoolkit/ , [Jan. 10, 2010].
[2] “Core Data Overview.” Internet: http://developer.apple.com/mac/library/referencelibrary/GettingStarted/GettingStartedWithCoreData/index.html , March 4, 2009 [Jan. 10, 2010].
[3] “Introduction to Cocoa Bindings Programming Topics.” Internet: http://developer.apple.com/mac/library/documentation/cocoa/Conceptual/CocoaBindings/CocoaBindings.html , March 8, 2009 [Jan. 10, 2010].
[4] “What is Core Animation?” Internet: http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/CoreAnimation_guide/Articles/WhatisCoreAnimation.html , Oct. 19, 2009 [Jan 10, 2010].
[5] “GNU General Public License, version 2” Internet: http://www.gnu.org/licenses/gpl-2.0.html , June 17, 2009 [Jan 10, 2010].

Sites

EVE Online: http://www.eveonline.com

EVEMon: http://evemon.battleclinic.com

Dev blog: http://emsdev.blogspot.com

Source code repository: http://code.google.com/p/evemacsuite/