PitFail Report 3

An Online Financial Engineering Game

December 12, 2011

Software Engineering I, Group 3
https://github.com/pitfail/pitfail-reports/wiki

Michael Koval, Cody Schafer,
Owen Healy, Brian Goodacre
Roma Mehta, Sonu Iqbal
Avanti Kulkarni

Table of Contents

1 Individual Contributions

All team members contributed in ways which are too layered and interlinked to quantify.

2 Summary of Changes

1. The domain model has been massively expanded to reflect the domain-specific aspects of PitFail. This is described in How the Domain Model Has Changed
2. The interaction diagrams have also reached further into the domain-specific areas of PitFail, to cover the new domain concepts. See Perturbations and Interactions. Note that these are for the most part not changes in the actual software, just changes in how we understand the role of the documentation.
3. PitFail has largely stuck to its original requirements; however, many more of those requirements have been implemented than had been at the time of report#2. Requirements that have been implemented since then:
    1. The addition of leagues.
    2. The addition of Teams (portfolios with multiple owners).
    3. The ability to view a portfolio's history
    4. It is possible to sell only part of your holdings of a stock via the website.
    5. The ability to view league standings.
    6. The ability to comment on trades.
    7. The ability to vote on trades.
    8. The ability to receive notifications about stocks one cares about via the android client.
    9. The ability to receive an email newsletter about one's portfolio.
    10. Dividends are now paid to users' portfolios (This is important. See dividendmotive).
    11. The ability to trade stocks directly with other players.
    12. The ability to place limit orders.
    13. Trading of stocks is now limited to available traders (which is far more realistic).
    14. Execution of market orders is now more realistic -- orders are executed at the asking price, not the last traded price.

3 Glossary

Here we attempt to clarify possibly ambiguous words as they are used in PitFail. We are not defining these words in general; just what they mean in this document.

Ask Price

: The price at which a trader is willing to sell a stock. [Ask]

Asset

: Either a stock or the buyer end of a derivative contract.

Bid Price

: The price a trader is willing to pay for a stock. PitFail has its own opinion (independent of Yahoo Finance) of what the bid price of a stock is, because PitFail players can trade with one another [Bid].

Client

A system that is not running on PitFail's server. An example is a webbrowser, the Android phone, or the Facebook connector.

Company

A synonym for Portfolio.

Consistency

Obeying all explicit or implied contracts (e.g., types describe a form of consistency).

Controller

The part of MVC that operates on the Model but is not represented by a domain concept.

Form

The main mode of interacting with the website frontent. User enters values and submits them.

Last Traded Price

The price which Yahoo Finance considers to be the last traded price of a stock.

Leaderboard

A list of the highest ranked portfolios in a League.

League

A group of portfolios that compete against each other. Users, Portfolios, and Leagues

Liability

The seller end of a derivative contract.

Limit Order

An order to buy or sell a stock at any available price within some bound [Limit].

Model

The part of MVC that maintains the state of the system. The Model is where concepts from the Domain Model are realized as code.

Player

A human being interacting with PitFail.

PitFail

The entire trading simulation, including the backend, the various clients, and the players playing it.

Portfolio

A portfolio in pitfail is the primary concept in trading; the one that makes trades, owns stocks, etc. See Users, Portfolios, and Leagues.

Price

Dollars per share.

Scale

A unitless number.

Side-effect

A result of invoking a function that is hidden by the function's signature. [SideEffects]

Stock

PitFail recognizes as a stock anything that Yahoo Finance is willing to assign a price to.

Team

A synonym for Portfolio, specifically used in the case where more than one user is in control of said portfolio.

User

A user is a single account in the PitFail system. A single user typically corresponds to 1 player. See Users, Portfolios, and Leagues.

View

The part of MVC that is visible to the user. The View may include HTML files, presentation-specific code, protocols for communicating with the user (e.g. HTTP), stylesheets.

4 General Information

4.1 References to the code

References into the code are given with a filename and an id such as ref_254, which appears in the code as a comment. line numbers can change but these should be constant.

4.2 Browsable Source Code

We used Mark Harrah's "browse" plugin [Browse] to generate HTML hyperlinked doc for the Scala code. The doc resides under the main source tree in `server/browsable-doc`. Hopefully this will make the code easier to follow.

4.3 Some general points about the code

Here we attempt to pre-clarify some aspects that might be confusing or unexpected in the report that follows. Some of this is due to our choice of programming languages; some of it is peculiar to our own code.

4.3.1 Lambda expressions

Scala has, and we often use, lambda expressions (example website/view/CommentPage.scala ref_524):

```
val postSubmit = Submit(postForm, "Post") { case text =>
    currentUser.userPostComment(ev, text)
}
```

The expression in curly braces:

```
{ case text => currentUser.userPostComment(ev, text) }
```

is a lambda expression [Lambda] (anynomous function). It becomes a function that can be treated like a value, and is passed to the Submit object, to be called when the form is submitted.

Some consequences of this:

1. Many of our functions do not have names. Their role is evident by the context.
2. Inversion of control [Inversion] is easy and so we use it often.

4.3.2 Traits

A `trait` in scala is similar to a Java interface, except that it can have concrete code in it as well [Traits]. Traits in scala can be used to

1. Split functionality into multiple units (See for example Organization of the Model into traits).
2. Provide a common interface to several classes (like how you'd use a Java interface).
3. Group together a set of disjoint cases, similar to an enum [ADTs] (example website/view/StockSeller.scala ref_104).

4.3.3 Option Types

Many of our functions return a type like `Option[Int]`. (example model/auctions.scala ref_188) `Option` is a Scala type (based on the ML type by the same name [ML]) that can be either present or absent [Option1] [Iry1]. So for example:

```
def sumOption(l: List[Int]): Option[Int] =
    if (l.isEmpty) Some(l.sum)
    else None
```

4.3.4 Monads

Some of our code is monadic [Monads1] (example model/stocks.scala ref_745, website/jsapi/jsapi.scala ref_618, model/magic.scala ref_650).

4.3.5 Applicative Functors

Some of our code uses applicative functors [Applicative1] provided by the Scalaz [Scalaz] library (example model/magic.scala ref_853).

4.3.6 Typesafe numbers

In the report there are many references to the types `Dollars`, `Shares`, `Price` and `Scale`. These are our own classes, defined in model/model.scala ref_868. They represent numbers, where the number represents a dollar value, a shares value, a price, or a unitless number (scale).

The purpose of these classes is to check at compile time that we are using the correct units. You can do (example model/stocks.scala ref_325):

```
val cost = price * shares
```

and get the right type. But if you accidentally do:

val cost = price / shares

you will get a compile-time error.
We did this after making too many math mistakes. It was a huge improvement.

4.3.7 HLists

Some of our code uses HLists (examples model/spser.scala ref_718, website/intform/branches.scala ref_575) (and other heterogeneous collections) [HList]. Our use of these is described more thoroughly in the sections that use them.

5 Architecture

5.1 Overall Architecture

PitFail follows roughly an MVC [MVC] architecture. David Pollak (Lift's author) believes that Lift is a "View First" architecture [View], but since none of use are familiar with "View First", nor do we really know what it means, we stuck to MVC.

5.1.1 Model

The model contains the most domain-specific parts of PitFail. These are classes that represent trades (model/stocks.scala ref_225), portfolios (model/stocks.scala ref_204), derivatives (model/derivatives.scala ref_807), etc.

The Model provides a large set of public methods for extracting data and performing operations. These are described in more detail in the section on interactions.

The model resides in `model/`.

5.1.1.1 Organization of the Model into traits If you look at the class `Portfolio` (model/users.scala ref_204) you will see it defines no methods, but it does mix in many traits (`PortfolioOps` (model/users.scala ref_782), `PortfolioWithStocks` (model/stocks.scala ref_569), `PortfolioWithDerivatives` (model/derivatives.scala ref_789), ...). This allows us to separate the many responsibilities of a portfolio (because there are very many) without having to expose that decision to client code: the client can use a `Portfolio` like a `PortfolioWithStocks`.

This made the model code much easier to work with. It's hard to imagine working in a language that doesn't have this feature; either you'd have to make client code aware of how you've broken up responsibilities (which is sometimes appropriate, but not usually), or have a few massive classes that are hard to work with.

5.1.2 View

PitFail has multiple view components that all refer back to the same model. These are the website, Android, Twitter and Facebook clients.

The Views use the public accessor functions in the Model to retrieve data and perform actions.

- Website view in `website/view/`
- Twitter view in `texttrading/twitter.scala`
- Android view in `android/Pitfail_Android_Cleint/src/com/pitfail/`
- Facebook view in ??????

5.1.3 Controller

To preserve the relationship between the Domain Model and the code, it is better to have few controller classes [Controllers]. A controller class is one that does not represent a concept in the domain (e.g. a Derivative is a domain-specific concept, but a DerivativeTradeOps is not (See also "Anemic domain model" [Anemic])). However, a few controller classes did sneak in:

- `Checker` (website/control/Checker.scala) runs a timer to perform periodic checks (dividend payments etc).
- `LoginManager` (website/control/LoginManager.scala) holds a current login
- `Logout` (website/control/Logout.scala) performs a logout
- `Newsletter` (website/control/Newsletter.scala) sends the newsletter

- `OpenIDLogin` (website/control/OpenIDLogin.scala) handles OpenID protocol
- `PortfolioSwitcher` (website/control/PortfolioSwitcher.scala) keeps track of a user's "current" portfolio
- `TwitterLogin` (website/control/TwitterLogin.scala) handles the OAuth protocol to log in with Twitter

5.1.4 All external libraries that our code uses

(*Note you don't need to install these dependencies because "sbt" will do that automatically. This list is provided to give a clearer idea of how our code is structured*).

- sbt as the build tool.
- Lift as the web framework.
- H2 Database as the database.
- Joda Time for time.
- Dispatch for HTTP.
- up for heterogeneous lists.
- SLF4J for logging.
- Scalaz Library for miscellaneous functional programming features.
- Scalatest for unit testing.
- Java Servlet API for servlets.
- GSON for JSON serialization.
- Scribe for OAuth protocol.

5.2 The Website

The website is one of PitFail's Views in the MVC architecture. It makes calls into the Model to perform specific operations (example website/view/CommentPage ref_458).

5.2.1 Overall Website Architecture

The website is built on top of the Lift Web Framework [Lift1]. It runs on the Jetty Web Server [Jetty1].

5.2.1.1 Performing actions (Buy/Sell/...) via the Web frontend Suppose the user has filled out a form like this one (Figure 1):

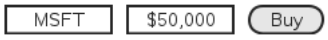


Figure 1: A form for buying stock.

and presses "Buy".
In order to process that request, the following must happen:

1. An HTTP post is sent from the browser to the server (Jetty).
2. Jetty delegates the request to the web framework, Lift.
3. Form data is parsed and processed.
4. A call is made to the model to perform the operation.

These steps are described in more detail below.

5.2.1.2 When Lift gets an HTTP POST The sequence of messages for an HTTP Post are (Figure 2):

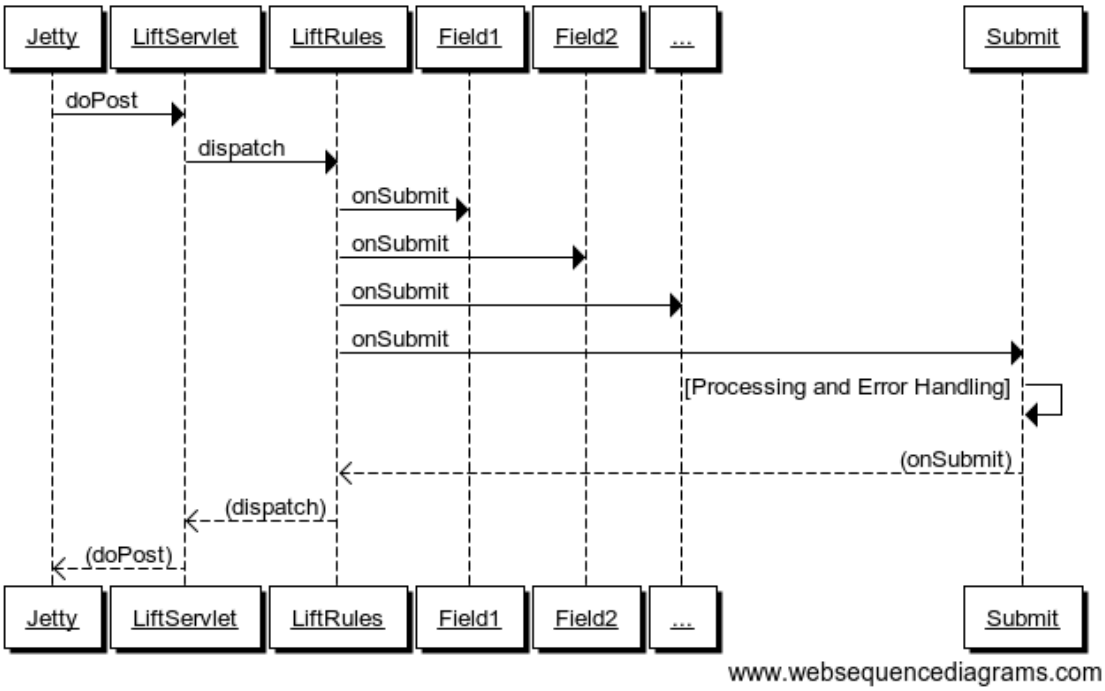


Figure 2: Form submission

PitFail is currently using jQuery to submit forms (website/html/templates-hidden/default.html ref_325). Ideally we'd like our forms to work using either jQuery or traditional HTML forms, but we got this working first so it's what we're using for now.

When the user hits "Buy", JavaScript in the page generates an HTTP POST directed at PitFail's server. The server Jetty receives the POST, and calls LiftServlet.doPost() (actually there are some other steps involved because LiftFilter must first filter the requests but these are all internal to Lift). LiftServlet passes the request on to LiftRules to dispatch it.

LiftRules recognizes that this is an Ajax request coming from an HTML form, and extracts the form fields out of it. LiftRules keeps a table of onSubmit callbacks indexed by field name. For all the incoming fields, Lift calls the onSubmit callback, and then finally the onSubmit callback for the submit button -- that way, by the time the submit button's callback is invoked, all the fields will have been invoked first.

We have written a significant amount of code to interface with Lift forms, which is described in Improving Lift Forms.

5.2.1.3 Checking for Consistency Scala is a statically typed functional language that has a lot in common with ML, where the philosphy is that you should use the type system to prove the consistency of your data at compile-time, eliminating the need for run-time checks [Typing].

Unfortunately, this is web programming, where your data is regularly sent to domains outside of your control. It appears that a strong type system relies a good deal on trust, which you simply don't have when half your program lives in a web browser. We found most of our work was spent meticulously pulling untrusted data back into a strongly typed format, only to have it be clobbered again at the next page reload.

When a form is submitted, we have to do 2 things with the data:

1. Convert the user's loosely structured input into a strongly-typed internal representation (example website/view/ModelFields.scala ref_717).

2. Perform the action requested (example website/view/CommentPage ref_458).

At either stage something can go wrong.

Because we wrote our own form handling wrappers (Improving Lift Forms), we wrote error handling code for our form wrappers, using a trait called `BasicErrors` (website/intform/intform.scala ref_293). `BasicErrors` checks each of the fields in the form for errors; if there are any errors these are reported to the user, and if all are consistent, it builds a single object containing all the form data (which is elaborated in Improving Lift Forms).

The process of structuring data and checking for input errors looks like this (Figure 3):

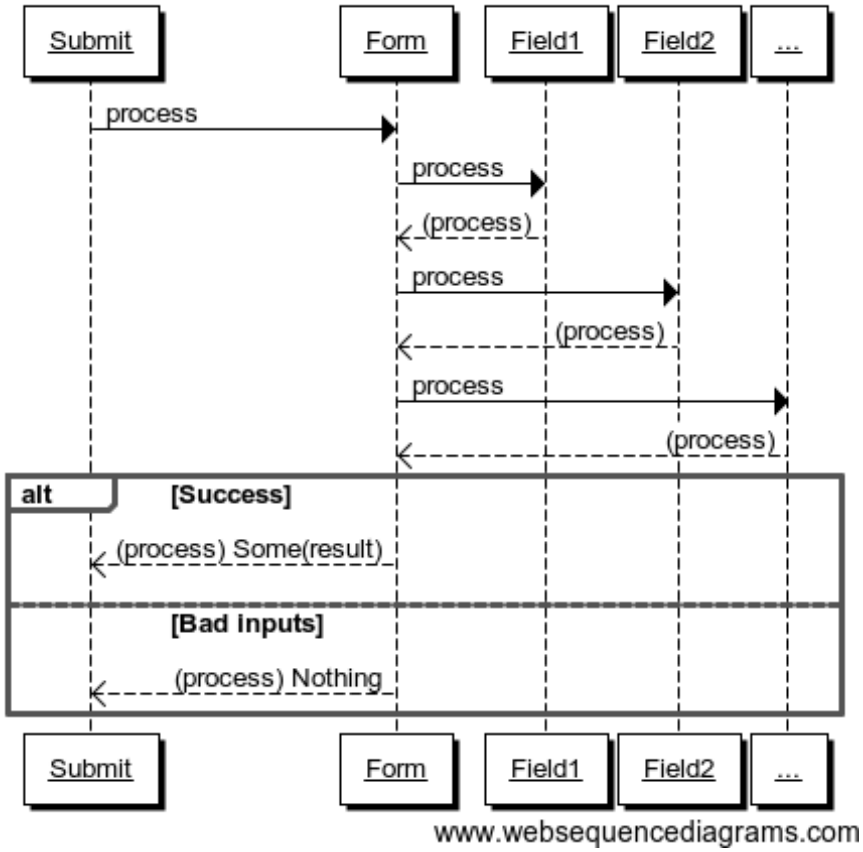


Figure 3: Checking for input errors

If the data makes it past input checking, the operation must be sent to the domain-specific parts of the code, such as `Portfolio` or `StockAsset`. These operations are described in detail in interactions.

If the operation fails because of something more fundamental -- say, for example, the user attempts to buy more of a stock than is being offered for sale -- the operation will throw an exception (`NoBidders` in this case) (model/stocks.scala ref_478). The View catches the exception and converts it to a message that will be displayed to the user (example website/view/StockSeller.scala ref_736).

We like this system because:

1. The Model (`Portfolio`, `StockAsset`, ...) do not have to duplicate the checks made in the view. For example, the model never needs to check that a string is formatted correctly like a number [Dry].

2. The Model does not have to provide human-readable error mesages; it mearly throws exceptions, which the View then decides an appropriate message for. This keeps our code to the MVC pattern.

5.2.2 Overall Android Client Architecture:

The Android client runs on the Android phones (Android version 2.2 and above). The development for the same is done using Android Development framework (Android SDK) which is basically Android library written in Java language. The Android library allows the developer to create screens, manage flows among the screens and also define connection with server (if required). In Pitfail, the connection from Android client can be made to either Yahoo! Finance to get the latest stock rates and other information or to the PitFail server, to update the database information about the user and also to retrieve user information according to the flow.

5.2.2.1 Android Frameworks used:

5.2.2.2 Activities: An Activity is an application component that provides a screen with which users can interact in order to do something. We created activities to perform different tasks like Sell Stock, LeaderBoard, New Team. Each activity is given a window in which to draw its user interface.

5.2.2.3 Services: A Service is an application component that can perform long-running operations in the background and does not provide a user interface. Android provided two types of services. Bounded and Unbounded. We created an Unbound Polling service to receive stock updates from the server. An Unbound service will continue to run in the background even if the user switches to another application. The Polling service hits the Jetty server periodically to recieve stock updates on any of the stocks held by user. Our Polling service starts as soon as the User starts the PitFail Application on his device.

5.2.2.4 Notifications: Notification is a special feature of the Android smart phones, where the user can receive important updates about the account even when the application is not in the front screen. We used this feature to provide notification to the user when the rates of the shares held by the user change in the market. This will help the user to receive automatic updates, rather than checking the statistics from time to time. The Polling service passes any stock updates as new notifications with a unique ID to the Notification Manager. The Notification Manager then displays the stock update message as a New Notification on Android Status Bar. The user can clear the Notifications whenever he wants.

5.3 Interacting with a Trading Simulation over Twitter

5.3.1 Motivation

Twitter is a service that is already widely used by many people, so there is a lower threshold of learning and discovery to play a game over Twitter than to use a dedicated website. It is not expected that the Twitter

interface will duplicate all features of the website; rather users will be able to perform their most common tasks from an interface they are familiar with.

The bulk of the proposal is a syntax that represents the operations of the game. This syntax could integrate into any system that allows sending brief messages from named accounts. However, since Twitter is already well integrated this extra flexibility may be unnecessary.

5.3.2 Implementation

5.3.2.1 Accounts The game has an account, tentatively named `pitfail`, and will listen for user tweets sent to `@pitfail`.

A user may *start* playing PitFail over Twitter. This lets the user start playing faster and with no setup -- the first message they send to `@pitfail` creates an account. There's no way to automatically associate this with an OpenID login (that I know of) -- if the user later wants to use the PitFail website

The program may respond to tweets that require a response by sending tweets back to users.

5.3.2.2 Syntax of the commands

5.3.2.2.1 View Portfolio

```
@pitfail portfolio
```

PitFail will respond with assets and liabilities in a human-readable form.

5.3.2.2.2 Buy a Stock

```
@pitfail buy 100 shares of HP
```

or:

```
@pitfail buy HP * 100
```

(See [[Products # A language for securities]])
or:

```
@pitfail buy $250 of HP
```

PitFail will respond with an ACK if successful, or an error if the trade failed.

This implicitly places a market order. PitFail currently does not support setting limits on the price at which the trade is executed.

5.3.2.2.3 Sell a stock

```
@pitfail sell 100 shares of HP
```

```
@pitfail sell HP * 100
```

```
@pitfail sell $250 of HP
```

5.3.3 Reflections, now that we have tried it

Being able to specify trades as text commands is *very* convenient. Yes, you have to learn the syntax of the commands, but once you do, it is much faster, clearer, less awkward, and generally more pleasant than using a website.

5.4 Architectural Styles

PitFail is composed of a large number of pieces of code which provide a wide range of functionality. This necessitated using different achitectural styles for various portions of the program. Some sections of the code are independent of other portions to the degree that they can be viewed as libraries. This is particularly the case with the Stock Database code, which presents itself as a library from which different querying archetectures may be constructed.

The Stock Database library heavily follows the pipe and filter achitectural style. Each class (also called a Stock Database, SDB) either links other SDBs together or communicates with a concrete SDB. In practice, many more of the SDBs form the interior "filtering" functions rather than the endpoint "driving" functions. The filtering SDBs impliment collating of requests, caching of results, various forms of rate limiting, and fallback between different pipelines of SDBs.

5.4.1 Identifying Subsystems

PitFails subsystems:

1. A *server*; this houses the Model (from MVC), the server for the website, a set of servlets that can invoked by the Android and FB clients, and the listener for the Twitter frontend.
2. The Android UI, which runs on the player's phone, and makes calls into the model via servlets.
3. The FB UI, which lives on its own server and makes calls into the model via servlets.

Note that the decision to house the website and Twitter UIs in the same running process and servlet container as the Model was purely at our discretion; on the other hand the Android UI has no choice but to run on a separate machine. Having the website run in the same process as the model meant that the website could make calls in-language to the model, which is much easier to work with.

Key "client side" portions of the code are the Android application and the Javascript code generated by Lift which notifies server side handlers of some event/change or makes a request which runs a server side handler. None of the *WebPage* Javascript code should be considered a subsystem of PitFail.

The interaction between subsystems is show in Figure 4.

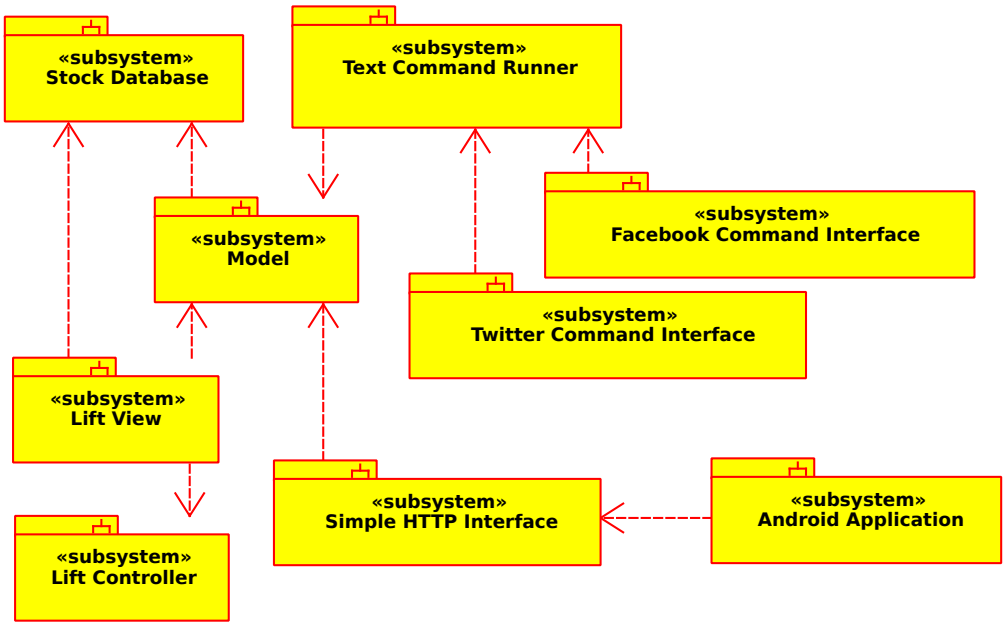


Figure 4: Major Subsystem Diagram of PitFail

6 Domain Model

6.1 How the Domain Model Has Changed

The original domain model diagram looked like this (Figure 5):

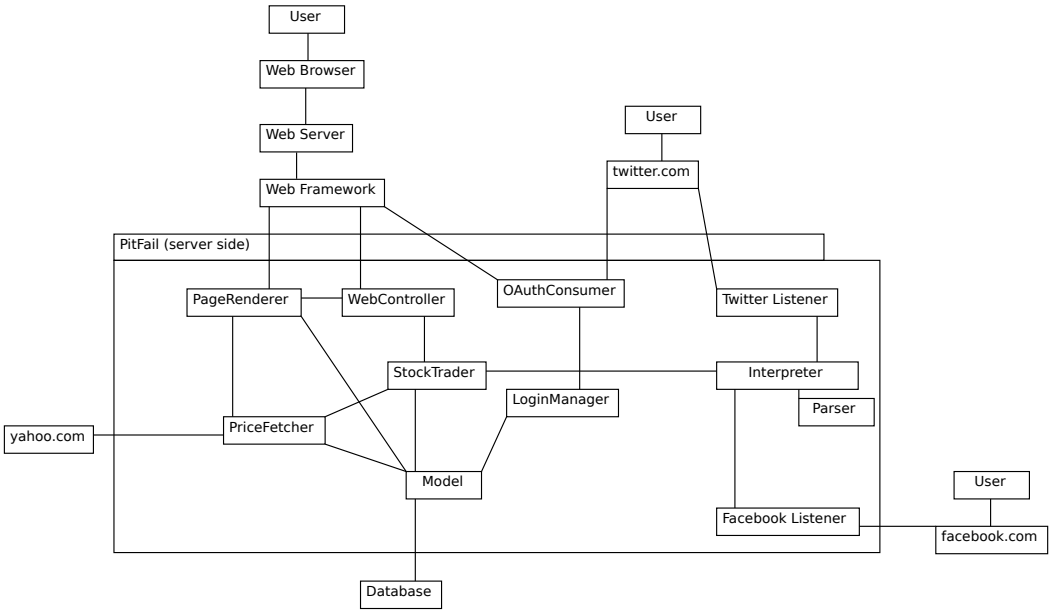


Figure 5: The original domain model.

The single biggest change has been the addition of many more domain specific concepts to the model. This is mostly due to our improved understanding of the role that a domain model plays, namely to clarify and define the domain that the system lives in. The old domain model was much less domain-specific and more software-specific, which does not serve the original purpose.

The concepts that have been added are domain-specific concepts such as `StockAsset`, `DerivativeAsset`, `NewsEvent`, `EventComment`, etc. (which appear below in the diagrams).

The new domain model has become too complicated to show in a single diagram. The various pieces of it are diagrammed and explained in the following sections. What has been *removed* from these sections are architecture-specific aspects of the system; these have been moved to other sections. The reason is that the architecture specific parts (how a request comes in, HTTP and AJAX protocols) are not domain-specific.

One consequence of working more domain concepts into the model is that we had to include some concepts that do *not* correspond to software objects. This is noted where it occurs.

6.2 Users, Portfolios, and Leagues

6.2.1 Basic Definitions

- "User" -- A human player of PitFail. A user may manage more than one portfolio.
- "Portfolio", aka "Team" aka "Company" -- A made-up PitFail entity that *owns* and *trades*. Many times in this document it may be mentioned that a "portfolio" places an order. The reason for this phrasing is that the order is associated with a portfolio, not with a user. The primary traders in PitFail are portfolios. A portfolio may be owned by more than one user.

- "League" -- a collection of portfolios competing against each other. A league is managed by a User, but participated in by Portfolios. Hence a single user may have portfolios that belong to different leagues.

An example might help to illustrate what is going on here (Figure 6):

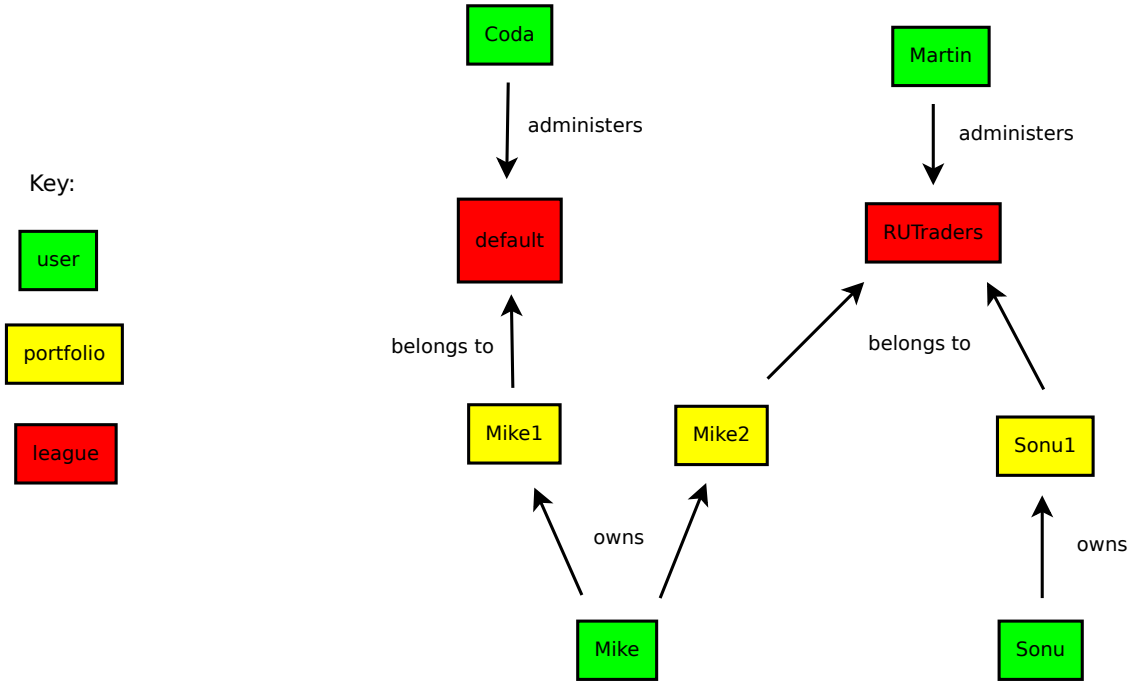


Figure 6: An example of users, portfolios and teams.

In this example, Mike and Sonu are users. Mike has two portfolios, named Mike1 and Mike2; Sonu has 1 portfolio, named Sonu1. Mike1 belongs to a league named "default"; Mike2 and Sonu1 belong to a league named "RUTraders".

Coda and Martin are users that administer the "default" and "RUTraders" leagues. Coda and Martin might have portfolios of their own, but this is not relevant to the business of administering leagues.

The reasons for the existence of each of these concepts is:

- "User" -- This provides a way for an actual human user to log into the site, to have an experience that is tied to them.
- "Portfolio" -- These actually do the trading. A Portfolio is the one actually credited with owning assets and being responsible for the payment of liabilities, *not* the user.
- "League" -- The purpose of a league is to represent "competition" between portfolios. Hence rankings are done within a league, and "rules" are set within a league. Trading, however, happens globally, among all leagues.

In the report we will often say that "a portfolio does this" and "a portfolio does that"; the action is being initiated by a human, but we model it as if the portfolio is the doer of an action: a portfolio buys a stock, a portfolio sells a stock. If we want to refer to a real human being we will use the word "player".

6.2.2 The User-Portfolio-League domain model

The basic concepts and relationships for the system in its idle state are (Figure 7):

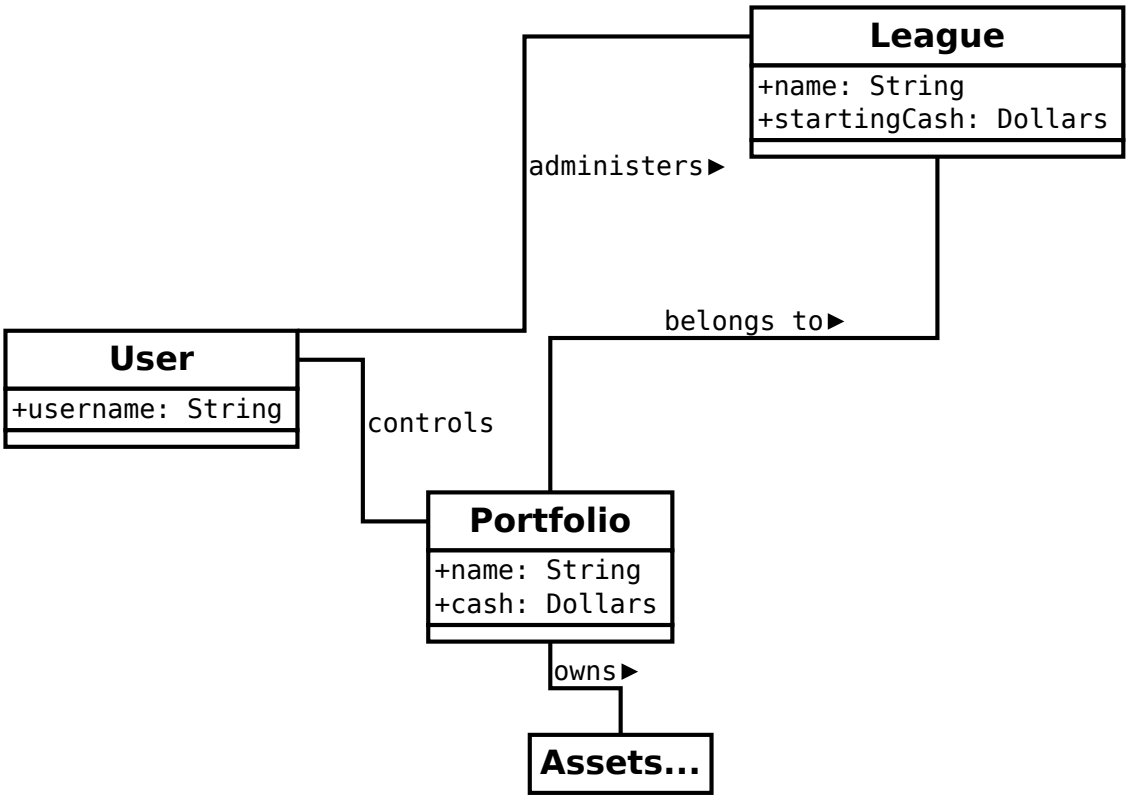


Figure 7: User/Portfolio/League concepts

Precisely what assets are indicated by "Assets..." will be described later (Assets and Liabilities). The relationships between these concepts are:

- *User controls Portfolio*: When a user controls a portfolio, a player who is logged in as that user can perform actions with that portfolio. e.g. if user Mike controls portfolio Mike1, then when human Mike is logged in as user Mike, he can buy stock with portfolio Mike1.
- *User administers League* The administrator(s) of a league can invite people to the league, and set the starting cash.
- *Portfolio belongs to League* Portfolios in the same league are roughly playing "against" each other, in that rankings are done per-league. But PitFail is not really an adversarial game.
- *Portfolio owns assets* See Assets and Liabilities.

Adding some of the creation/joining operations, this becomes (Figure 8):

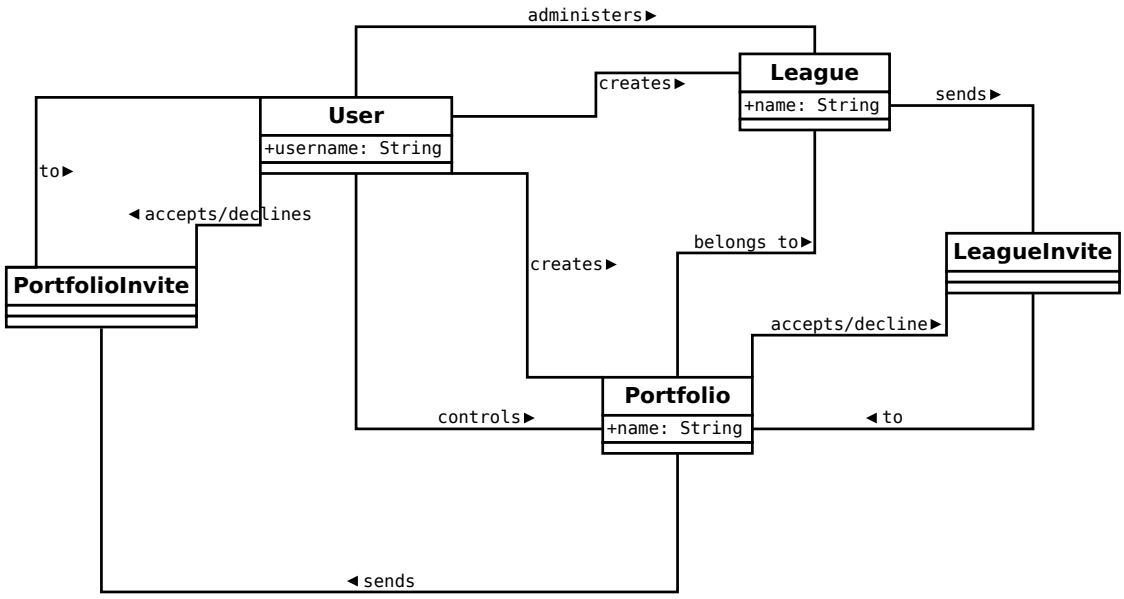


Figure 8: User/Portfolio/League concepts, with creation/joining operations

Note a few potentially surprising things about this model:

- PortfolioInvites are sent to Users, and LeagueInvites are sent to Portfolios. This is because it is a User who will control a portfolio, and a Portfolio that will join a league (users do not join leagues).
- Even though, in reality, a human user initiates the action of "sending" an invite, it is shown in the diagram as originating from a Portfolio or a League, because that is how we interpret it; invites come from the concepts that can be joined.

In the actual code, some of the "many-to-many" relationships acquired an extra class (the association class). Such as (model/users.scala) (Figure 9):

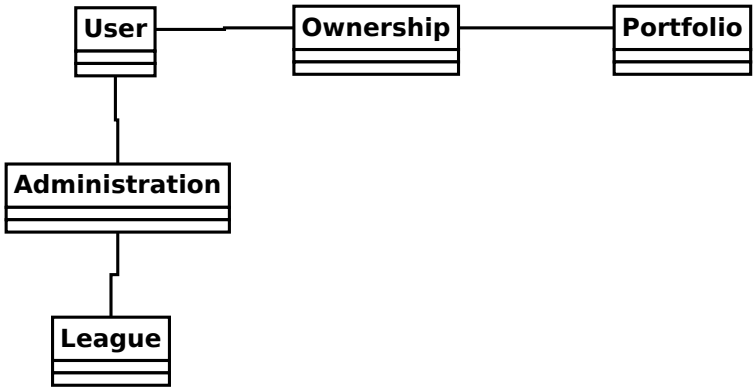


Figure 9: Some association classes.

But this is a detail of the implementation and not part of the domain model; no meaningful attributes are stored with Ownership and Administration.

6.3 Assets and Liabilities

This part describes only the *ownership* aspect of assets and liabilities. The trading and exercising aspects will be described later.

The diagram below shows only the part of the domain model that relate to the ownership of assets and liabilities (Figure 10):

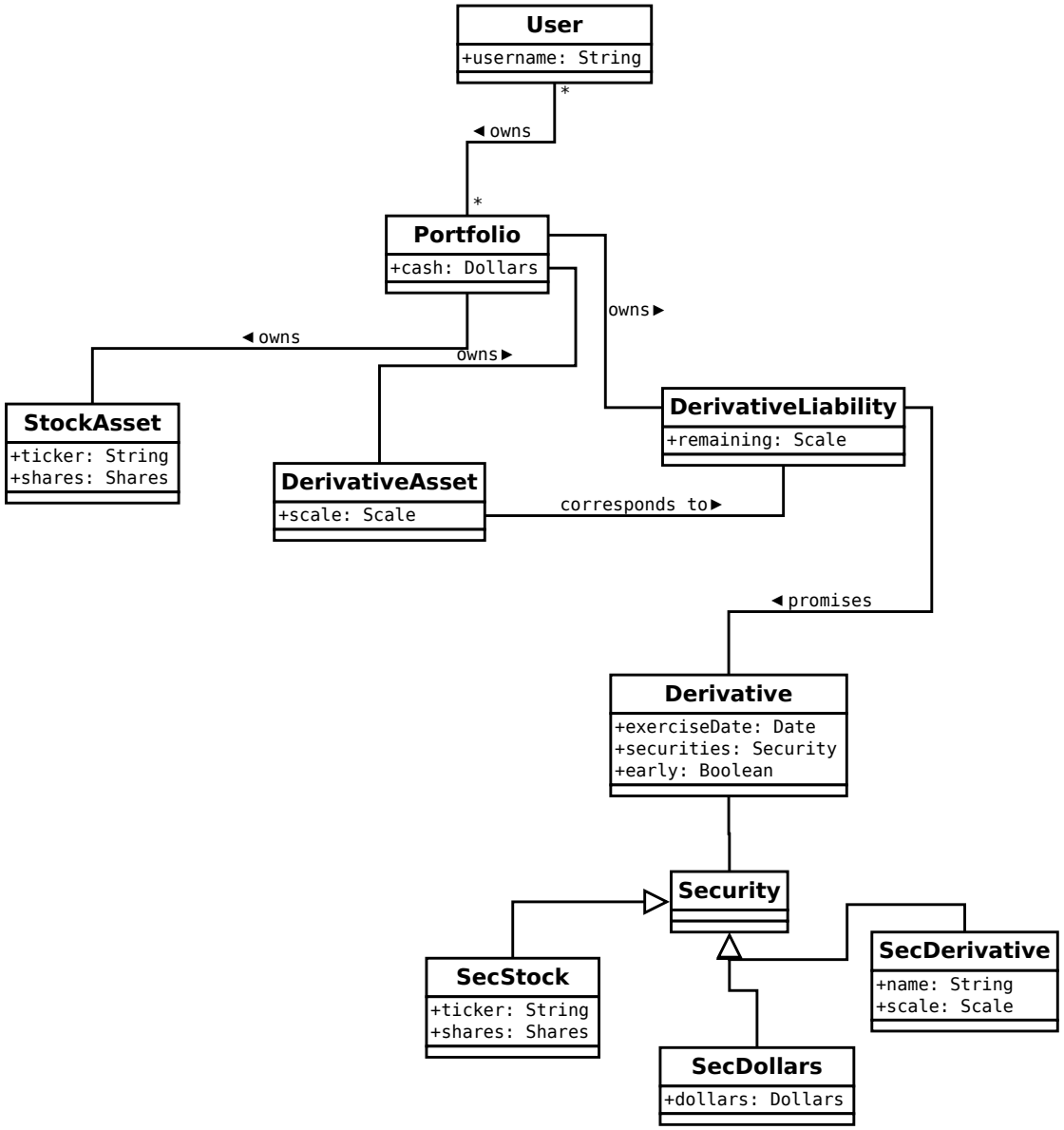


Figure 10: Assets and Liabilities

There are two kinds of assets: StockAssets and DerivativeAssets, and one kind of liability: a DerivativeLiability.

6.3.1 How StockAssets work

A stock asset is simply a number of shares of a particular stock. So for example, 30 shares of MSFT is a stock asset.

6.3.2 How Derivative Assets/Liabilities work

A derivative, in PitFail, is a promise to exchange a list of assets on or before a specified date. There are 3 parts to this contract:

1. The *Derivative* is the statement of the contract; that is, it is the list of assets to be exchanged, the date on which it is to occur, and whether the contract may be exercised early (See for example [American]). The exact nature of how the contract is specified is described in the section on derivativeexp.

2. The *DerivativeLiability* is the statement by one portfolio that they will offer up the assets specified in the Derivative.

3. The *DerivativeAsset* is a promise to a portfolio that they will be able to collect the assets promised in the Derivative.

Each DerivativeAsset corresponds to exactly 1 DerivativeLiability, and each DerivativeLiability corresponds to 1 or more DerivativeAssets. Each DerivativeAsset has a property called `scale` which is the portion of the liability this asset has a claim on. A DerivativeLiability has an attribute `remaining` which is the fraction of the contract that has *not* been exercised (Figure 11):

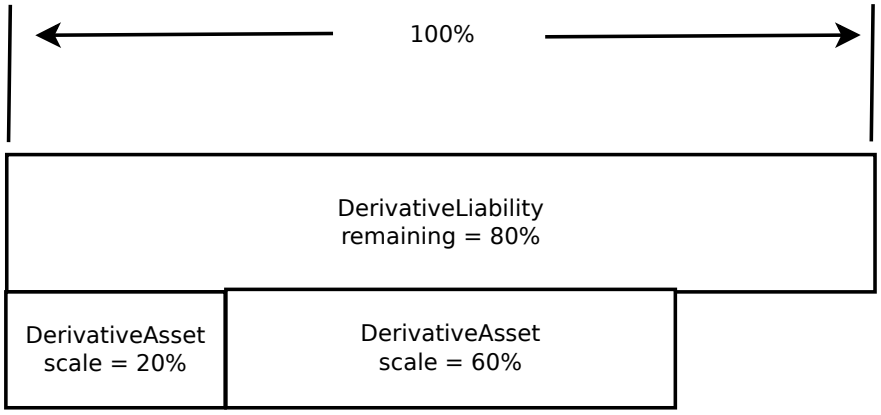


Figure 11: The relationship between the sizes of DerivativeAssets and DerivativeLiabilities.

Every time a DerivativeAsset is exercised, it is deleted, and the `remaining` of the corresponding DerivativeLiability is reduced by the `scale` of the DerivativeAsset. It is an invariant of the system that the sum of of the scales of all DerivativeAssets for a particular DerivativeLiability must equal the `remaining`.

6.4 Derivatives

The parts to a derivative contract are:

1. A list of securities to be traded.

2. A date on which this is to occur.

3. Whether it may be exercised early.

4. A condition that decides (automatically) whether the derivative will be exercised on the scheduled date.

(2) and (3) are just a `DateTime` and a `Boolean` respectively; (1) is more complicated.
The list of securities is represented as a list, where each element may be one of:

1. A "stock" security, `SecStock`, which holds a ticker symbol and a number of shares.
2. A "dollars" security, `SecDollar`, which holds a dollar amount.
3. A "derivative" security, `SecDerivative`, which holds a named liability and a scale (see the section on Scaling Derivatives). (At the moment there is no way within the PitFail UI to create a `SecDerivative`. However, since the theoretical concepts behind it are complete, we describe it anyway).

If any of the quantities are negative (eg negative shares, negative dollars, negative scale), that means that the securities are supposed to move from the buyer to the seller.

For a descripton of how derivatives are exercised see Exercising Derivatives.

6.4.1 Scaling Derivatives

Many aspects of PitFail require that derivatives be scaled. That is, given one derivative, create a new one with identical terms, but "smaller" or "larger" (Figure 12):

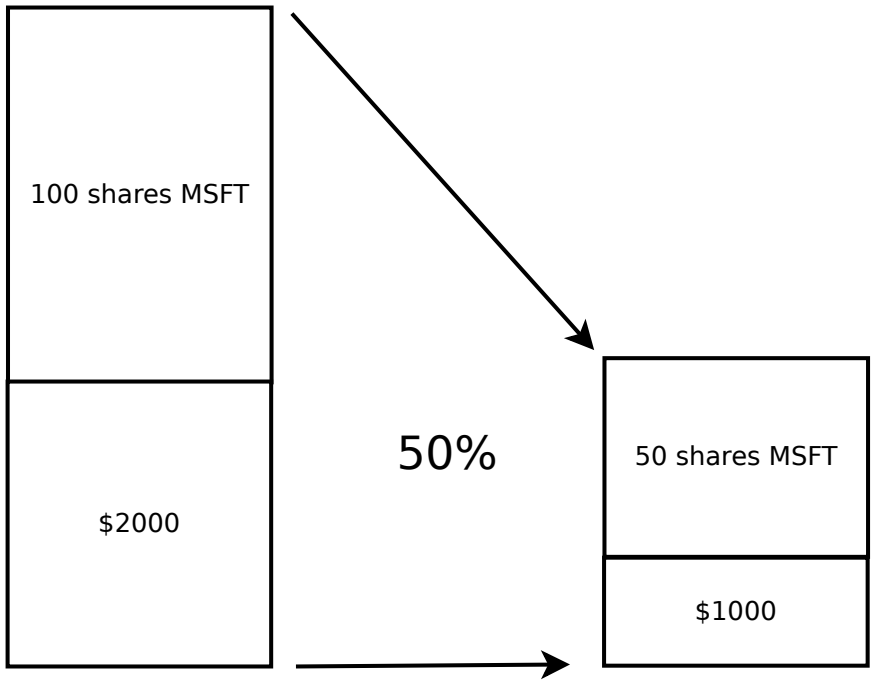


Figure 12:

Scaling is done by scaling each security promised:

1. For SecDollar, scale the dollar amount
2. For SecStock, scale the share amount
3. For SecDerivative, scale the scale amount

and leaving the date and early exercise the same.

6.5 Trading Stocks

The diagram below represents the "idle state" of the system with respect to stock trading (13):

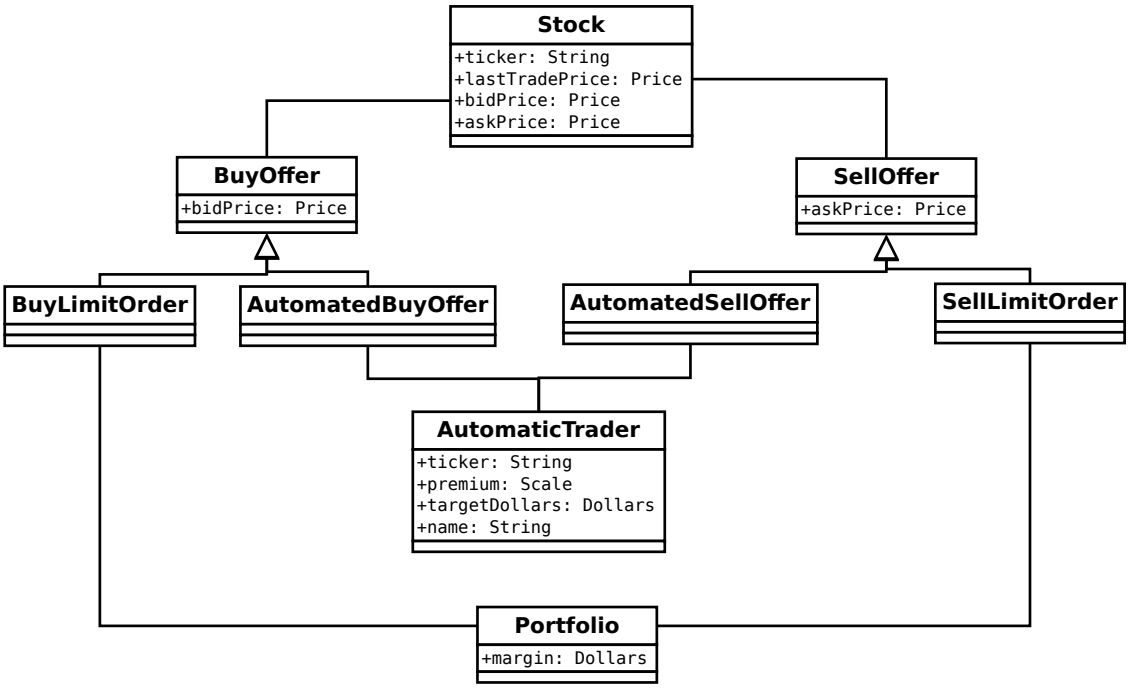


Figure 13: Stock trading at idle.

When the system is is idle, no trades are taking place; all that exist are orders that have yet to be fulfilled. PitFail allows only two kinds of orders to sit idly. These are

1. Limit orders
2. Automated (synthetic) trading orders.

Market orders do not exist when the system is idle because market orders are executed at the offering price as soon as they are created. PitFail does not provide explicit support for stop orders, but it would be easy for a user to create one using the javascript automated trading API (and, when a Stop is triggered, it becomes a market order [Stop], and so will be executed immediately).

All orders in the idle state have two important properties: the available number of shares, and the limit price. This will allow PitFail to form automatic matches, as described later.

An invariant of the system is that when the order system is Idle, there are no orders that can be matched with one another.

6.5.1 When a new order comes in

When a new order comes in, it has a desired number of shares, and it may or may not have a limit price. First, all existing orders for the same stock are collected, and sorted by desirability (ie, best price to worst price) Figure 14:

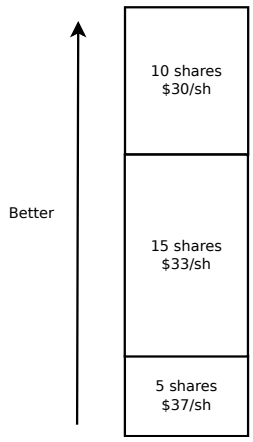


Figure 14: Comparing available with desired shares.

The incoming order is matched up against the best orders possible (that are below its limit price, if any). Those orders are then completely or partially executed (Figure 15):

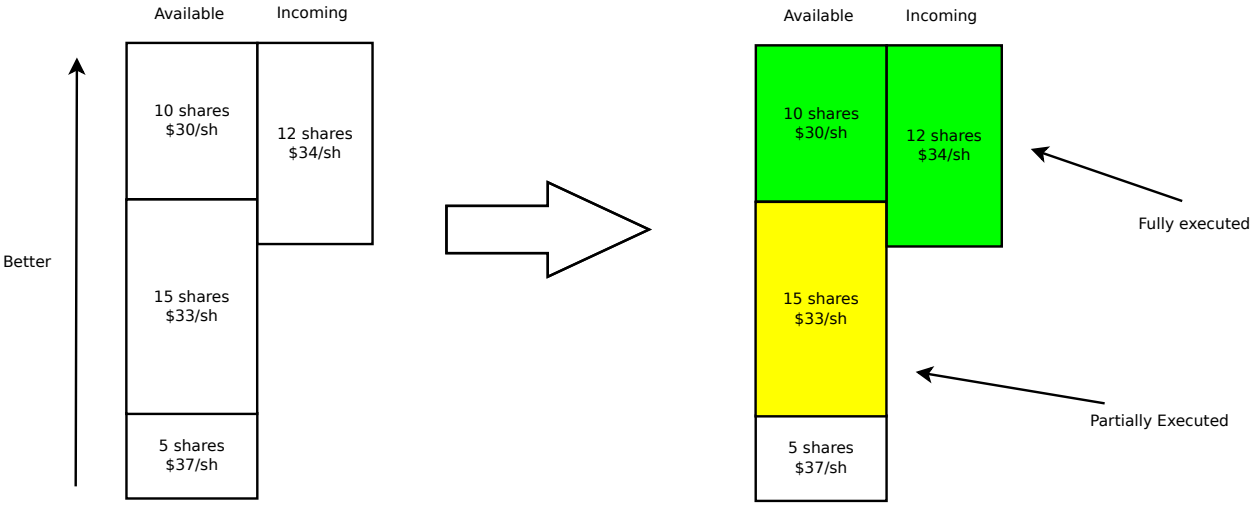


Figure 15: Which orders are partially or fully executed.

In this example, 10 shares will be purchased at 30/sh, and 2 shares at 33/sh.

You will notice that the orders already *in* the pool pay a price in not being able to negotiate -- since the buyer is willing to pay 34/sh, they would, if they could, increase their limit to 34/sh to take advantage. However, by having orders in the pool that are *not* negotiated, there is a benefit in liquidity; hence traders who place orders unexecuted into the pool will change a liquidity premium in the trade (which is why there is a spread between the bid and ask price for a stock as offered by the same trader [Makers]).

If the newly placed order is not fully executed, and the trader specified a limit, it will become part of the pool of unexecuted orders.

6.5.2 Margin

In order to ensure the smooth execution of orders, when a user places an order that is not executed immediately, they must set aside margin so that the order can be executed later. For a buy order the user sets

aside cash that will be used to buy the shares when the order is executed, and for a sell order the user sets aside the shares that will be sold.

If the order is cancelled or not fully used the margin will be returned.

6.5.3 Domain model for trading

The model below does not correspond 1-1 to actual software classes because our architecture is not entirely object-oriented. For example, there is no class called Execution; execution of orders is procedural (Figure 16).

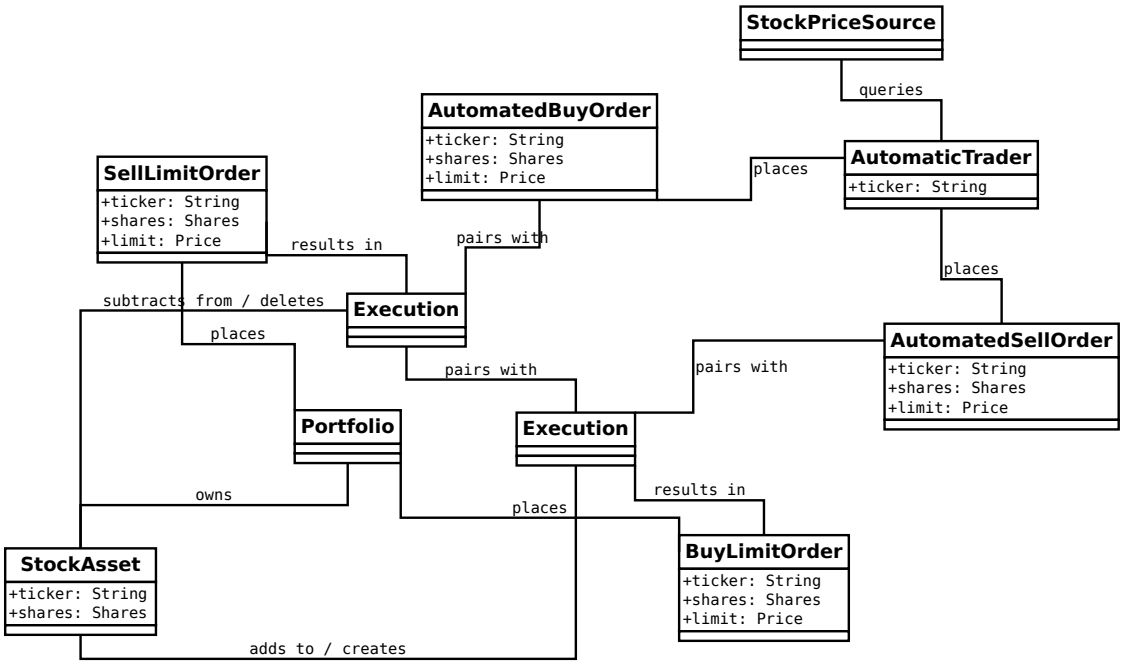


Figure 16: The execution of a trade.

The association of AutomaticTrader with StockPriceSource is meant to convey that the automatic traders use real-world bid and ask prices to set their bid and ask prices.

Because there is too much to fit on one diagram, here is the part of the domain model that deals with cash and margin (Figure 17):

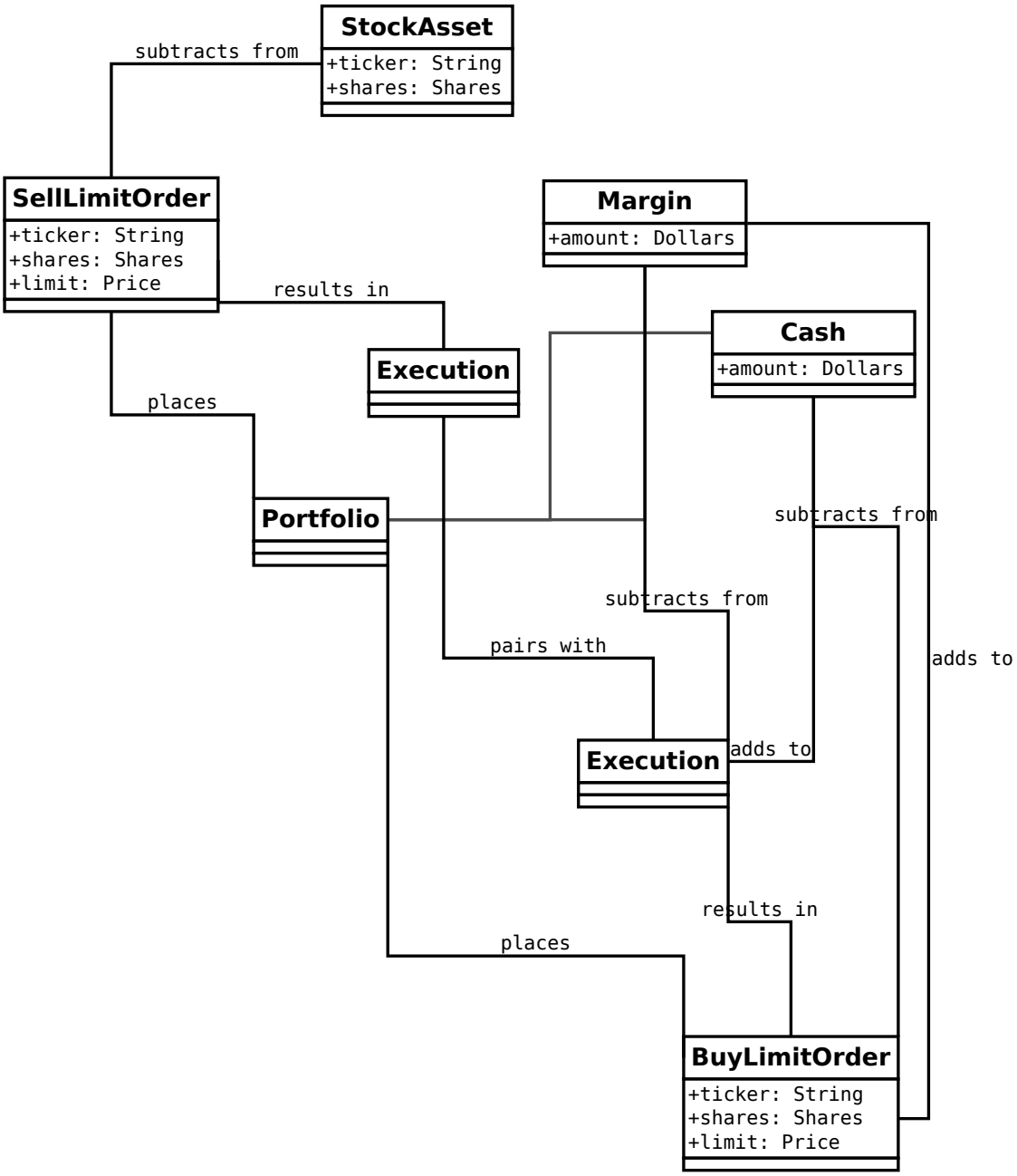


Figure 17: How cash moves when trading.

(In the code, there is no object called Cash, rather it is an attribute of Portfloio; but it is helpful to show it as such for the domain model).

The reason that the execution of a BuyLimitOrder "adds to" Cash is that all the necessary cash has already been set aside in Margin; the cash that is being added is the leftover margin.

When an order is cancelled (by its owner), all that must happen is that the margin is restored (Figure 18):

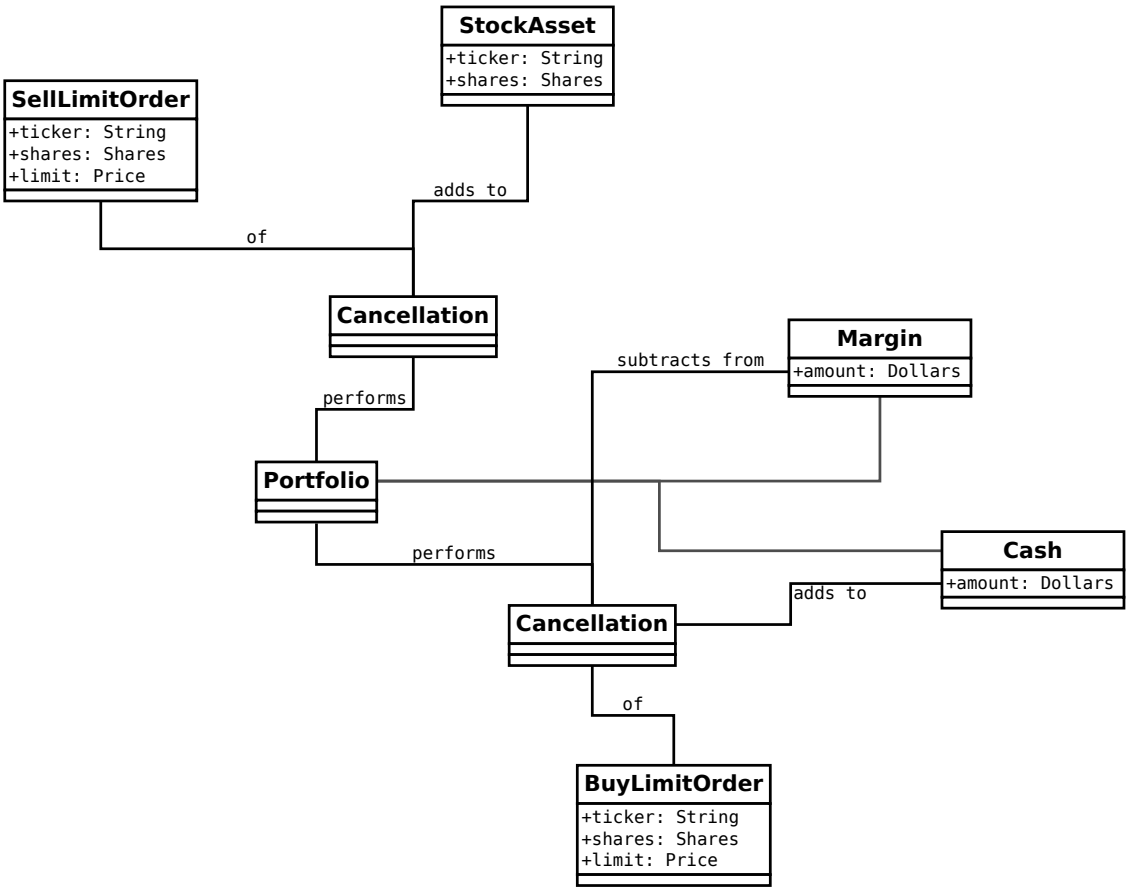


Figure 18: Cancelling and order and restoring margin.

6.6 Dividends

It is very important for PitFail to keep track of dividends paid by stocks, for two reasons:

1. It would be unrealistic in a particularly unsettling way: stocks that will never pay dividends have no value; why are we trading them?

2. Because PitFail players will own stocks that pay dividends, and every time a dividend is paid the stock price drops abruptly, players would not appreciate having the price drop if they do not receive a dividend in return.

Periodically, PitFail queries Yahoo Finance to see if stocks owned by the players have paid dividends. If they have, the system will pay dividends to the player, in what is represented here (though not in the code) as a `DividendEvent` (Figure 19):

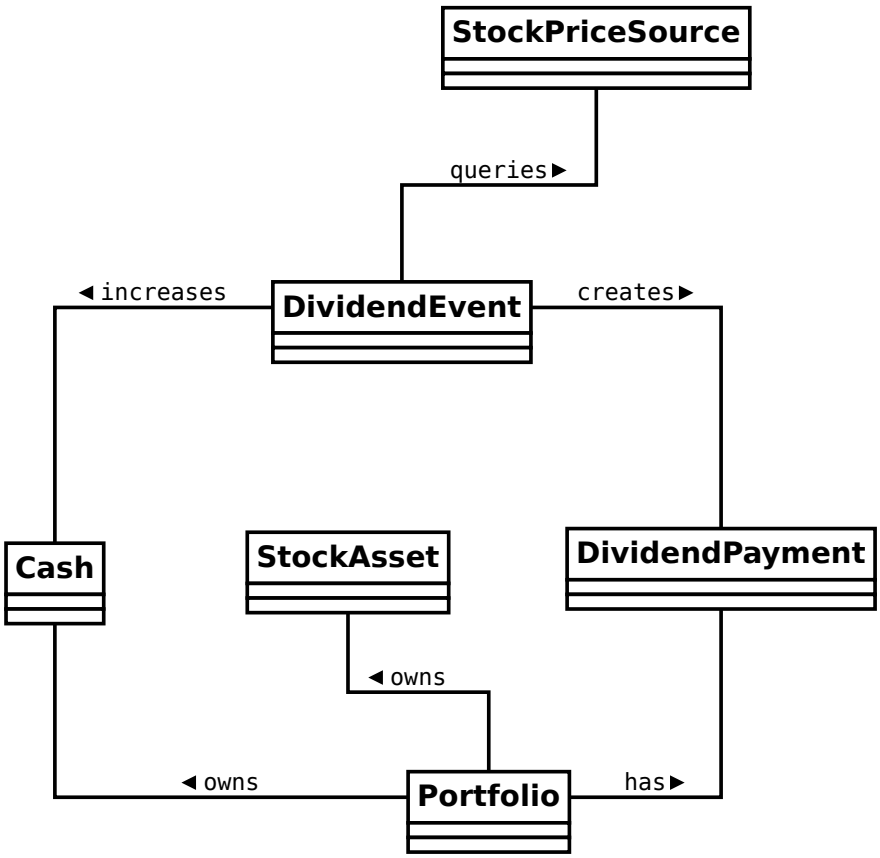


Figure 19: When dividends are paid.

The `DividendPayment` object is created only to allow the user to view the history of their dividend payments.

6.7 News

The purpose of "news" is to show PitFail players to see what other PitFail players have been doing. Importantly, News is not part of actual trading; this is just for seeing what's going on.

This means that a single news event has associations with a lot of other concepts, but not in a way that affects the rest of the program: it's just point out, for example, which derivative was traded when reporting that a derivative was traded.

The basic concept domain for News is (Figure 20):

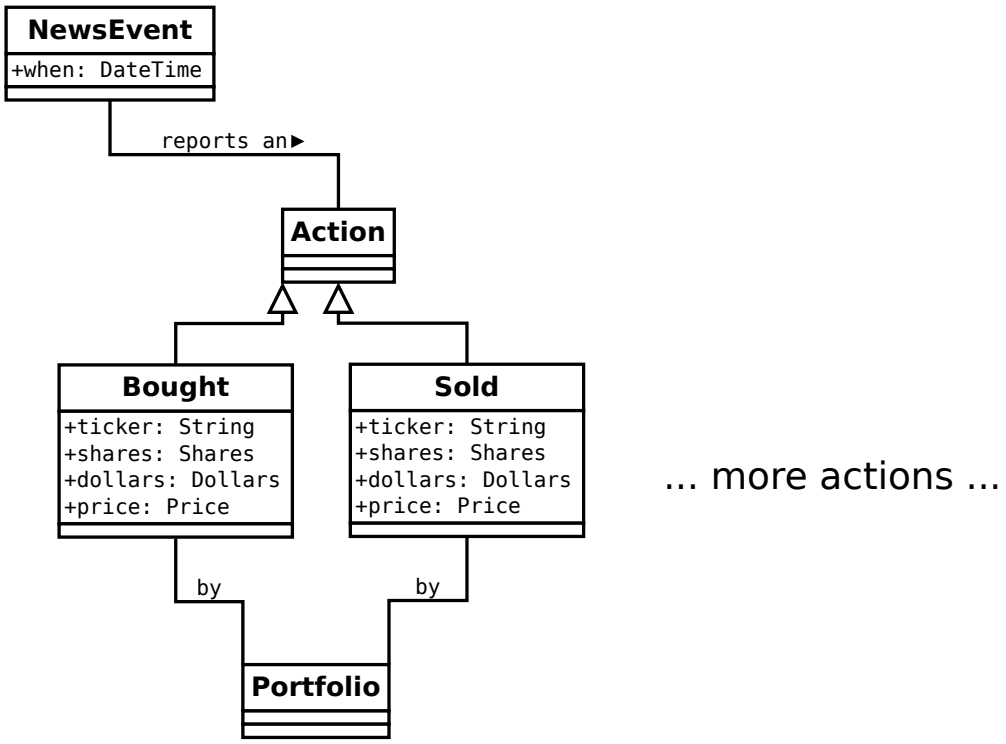


Figure 20: The news Domain.

only two actions are shown here; there are a lot so they are split up across multiple diagrams.

Buying and selling stocks, as shown above, refer to the Portfolio who "did" the action, and the information about what was bought or sold. This only applies to orders that are executed (either immediately or later). Orders that are delayed will generate another kind of an event.

Derivative Trading has the following kinds of events (Figure 21):

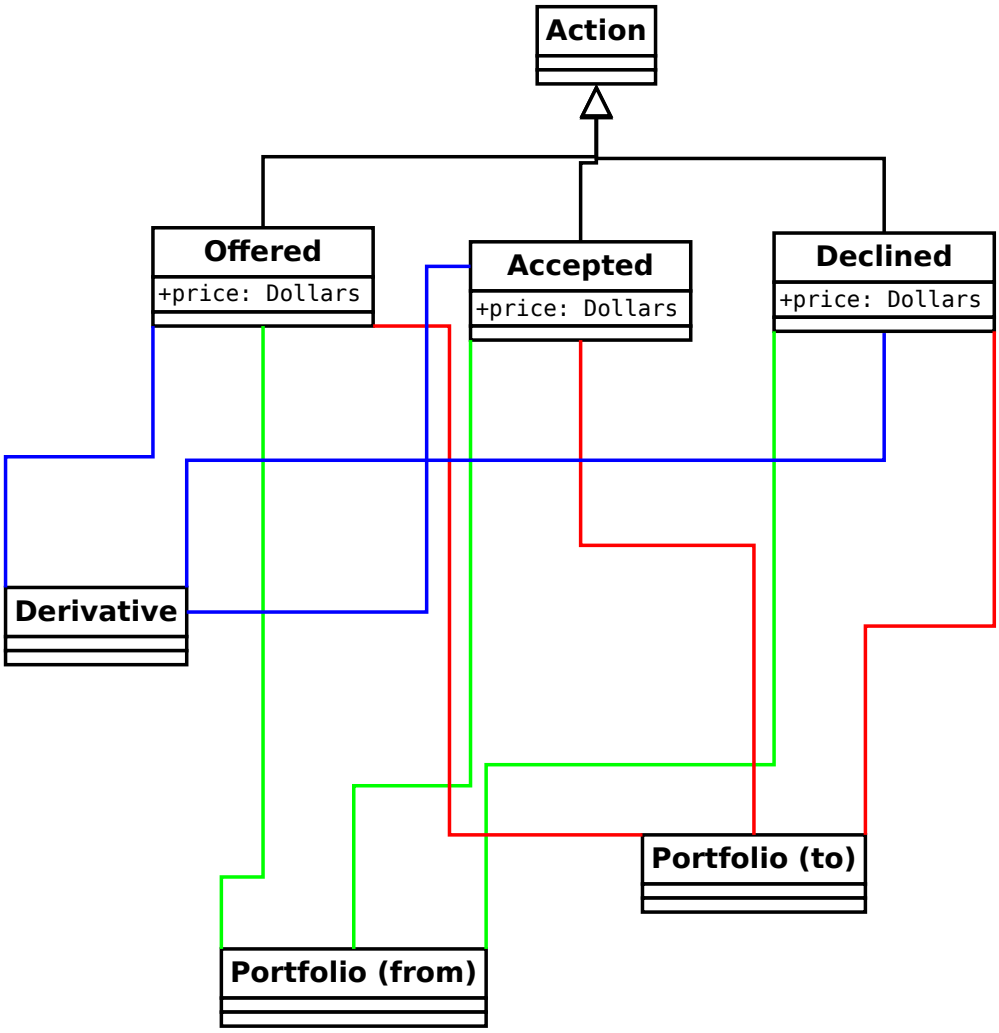


Figure 21: News for derivative trading.

`from` and `to` are shown as separate concepts even though they are instances of the same class, because they play a different role in these events: one is the portfolio making the offer, the other is the portfolio receiving, and possible accepting, the offer.

For Auctions we have (Figure 22):

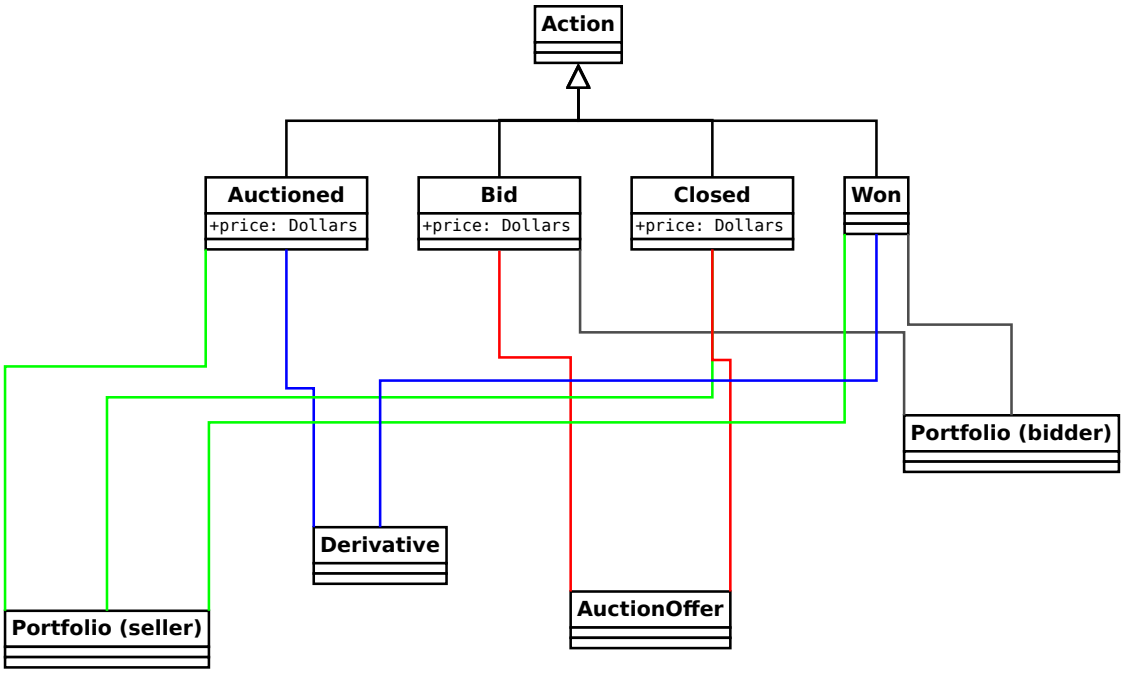


Figure 22: News for auctions.

There are other associations which are not shown, that relate to voting. These are described in the section on voting.

Placing orders that get delayed are described by (Figure 23):

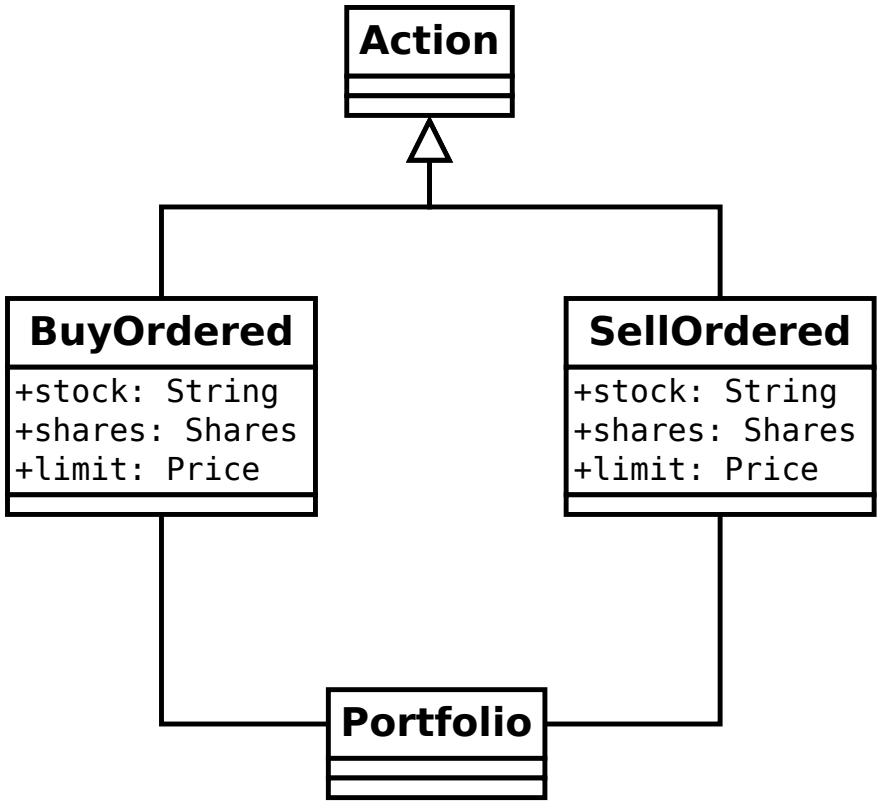


Figure 23: News for orders.

Where the associated portfolio is the one who performed the buy or sell.
There is one more event for exercising derivatives (Figure 24):

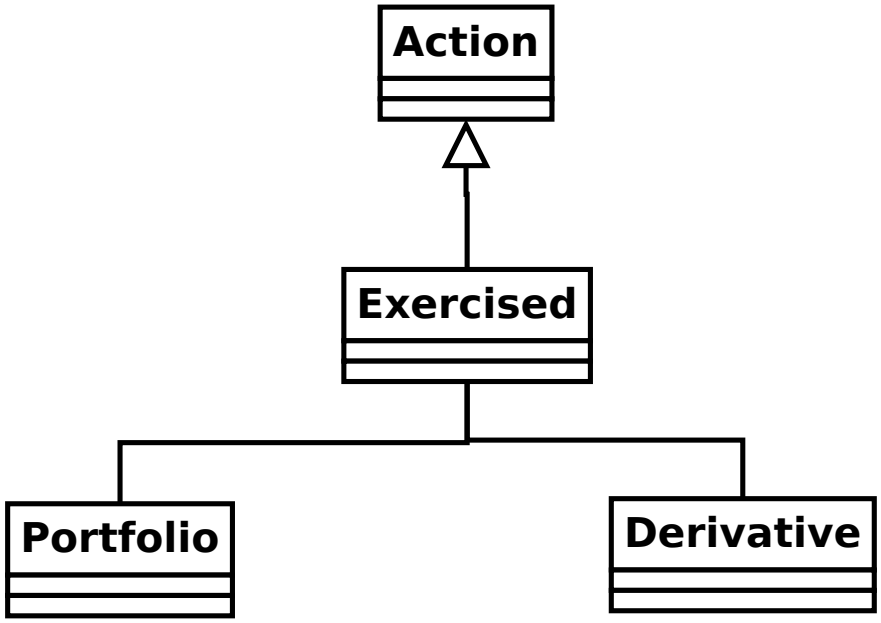


Figure 24: News for exercising derivatives.

Where the associated portfolio is the one who did the exercising.

6.8 Voting

When players enter into a contract (not executing it yet, just entering it) involving a derivative, the following assets are moved (Figure 25):

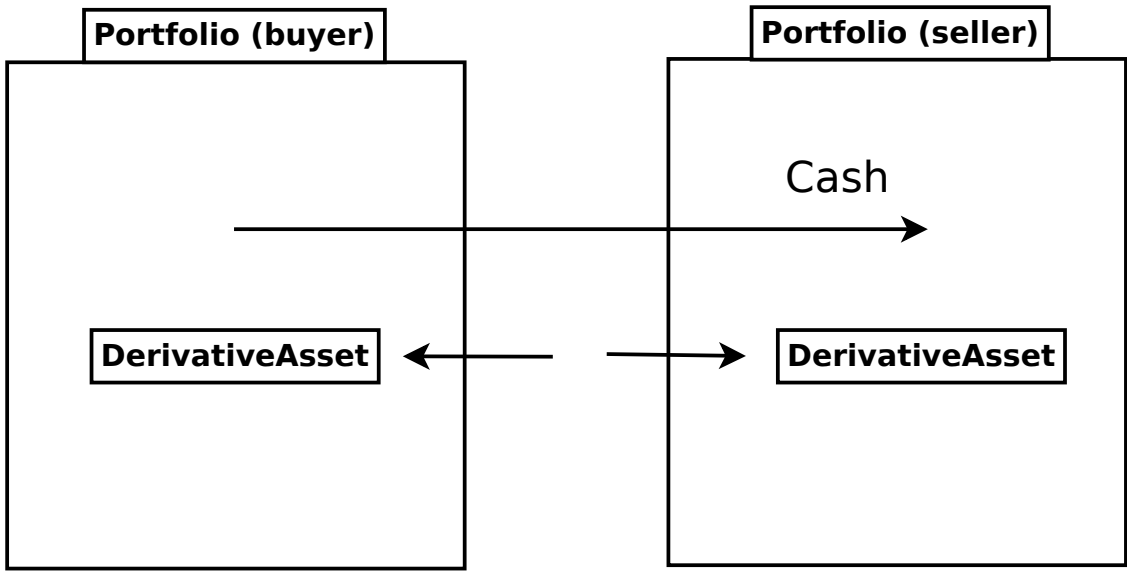


Figure 25: How assets and liabilities change when a contract is entered.

If owning the asset (being in the buyer side of the contract) pays off more than the cash payed, the buyer is happy. If owning the liability (being in the seller side of the contract) is not bad enough to negate the cash received, the seller got a good deal. These are not necessarily mutually exclusive.

Now, say a third player, the Voter, looks at his news feed and thinks that the buyer got a good deal (and maybe the seller too, but that is not relevant yet). The Voter would be happy with an arrangement like the following (Figure 26):

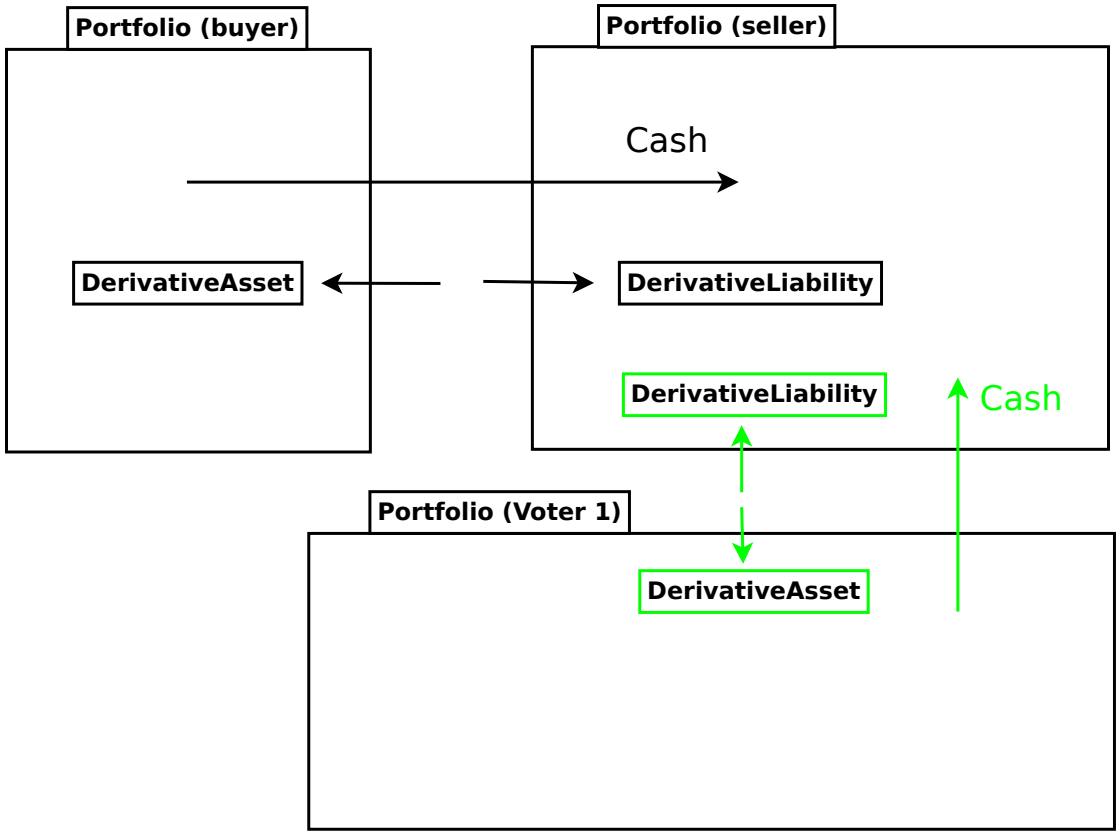


Figure 26: Another player makes a similar deal.

where the derivative in green resembles the derivative in black, and the cash in green resembles the cash in black. (As in, if it was a good deal for him, it's a good deal for me too. Not necessarily true, but it could be true sometimes).

When two portfolios enter a derivative, an object is created called `DerivativeBuyerSetAside` (there is a nearly identical process for sellers) (Figure 27):

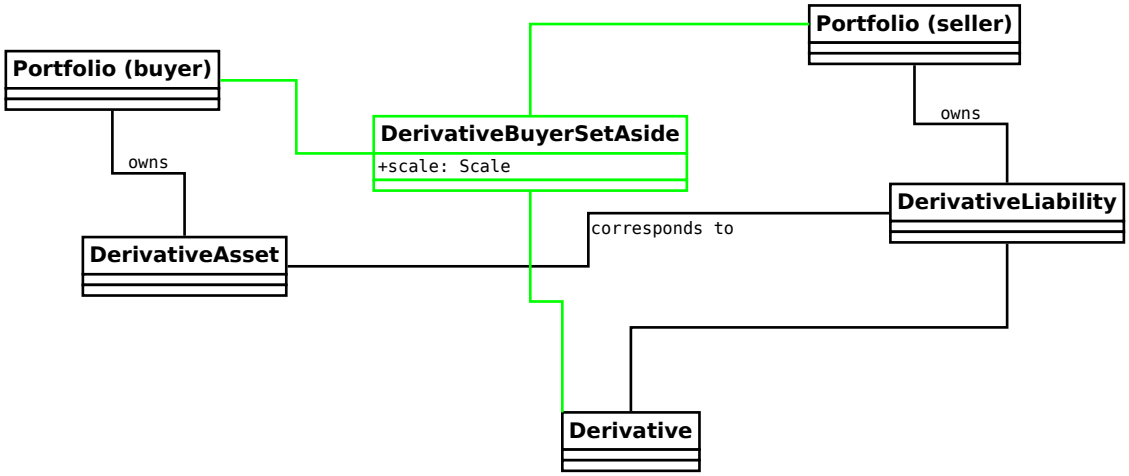


Figure 27: DerivativeBuyerSetAside

(remember, the `Derivative` holds the terms of the contract, and the `DerivativeAsset` and `DerivativeLiability` show who owns which end).

The `DerivativeBuyerSetAside` holds one attribute, which is the "amount" left to be voted on. For the precise meaning of this scale, see the section on Scaling Derivatives.

The `scale` remaining starts out at `3%`. When the first voter votes in favor of the buyer, they enter into a contract with the seller that is identical to the original derivative, but scaled to 1.5% (= 3%/2). He also pays the seller 1.5% of what the original buyer paid (Figure 28):

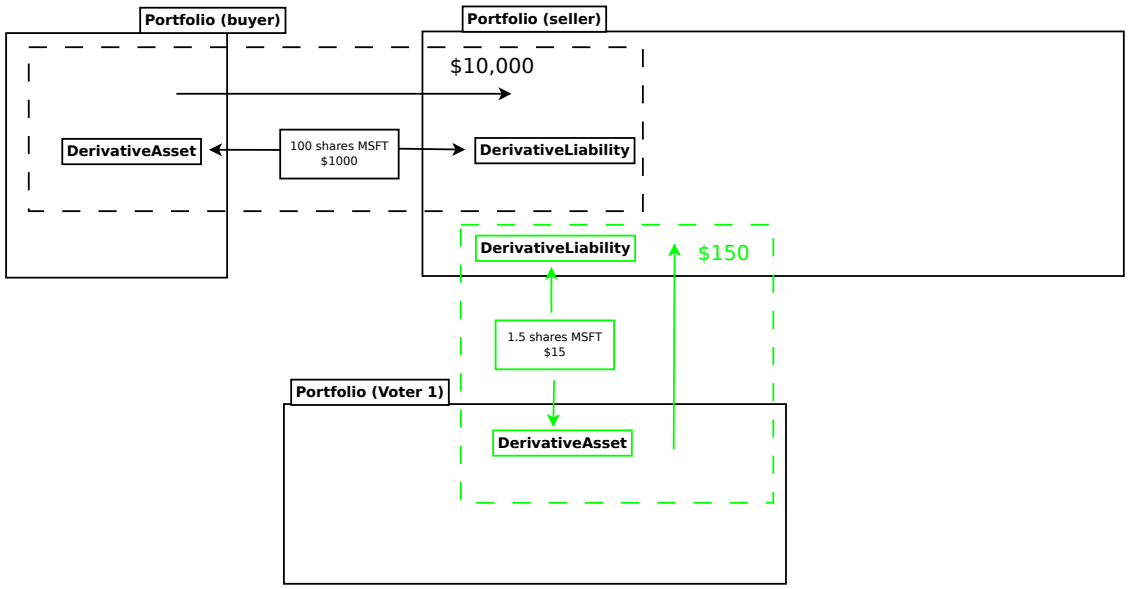


Figure 28: A voter enters into a contract.

The `scale` remaining is then cut by half to 1.5% (The interpretation of this is that the original `3%` is the total amount that will be allocated after infinitely many votes are made).

Now if another player votes, they will realize 0.75% of the original trade (Figure 29):

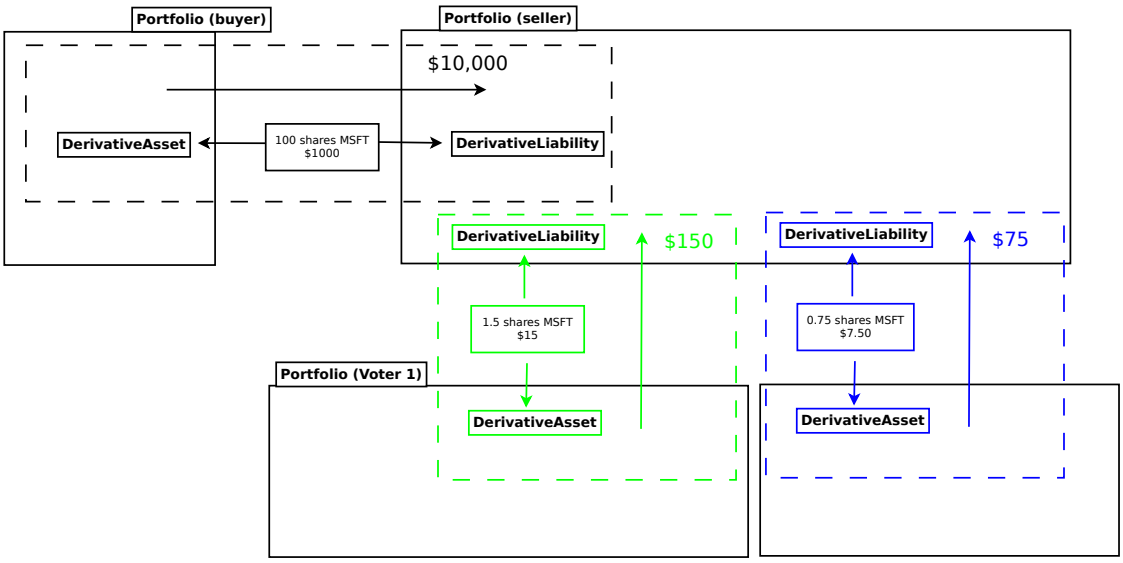


Figure 29: Another voter casts a vote.

Votes are recorded and associated with the origanal NewsEvent, so that a score of buyer-votes and seller votes can be calculated (Figure 30):

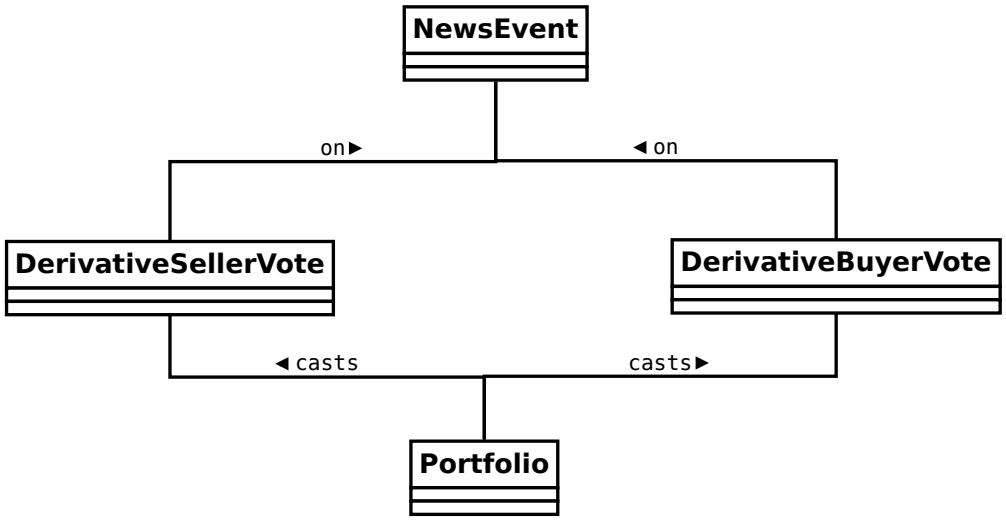


Figure 30: Scoring events.

6.9 Comments

Compared to voting, comments are refreshingly simple.

Users, not portfolios, cast comments. A comment is associated with a news event (Figure 31):

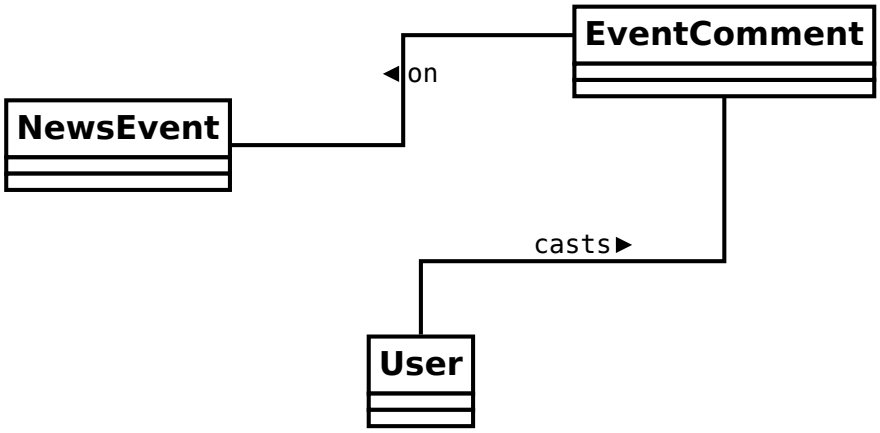


Figure 31: Comments on a news event.

6.10 Auto Trades

While the system is idle, an auto-trade is represented as (Figure 32):

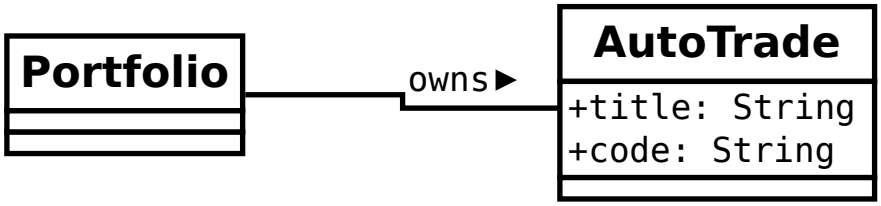


Figure 32: An auto trade while the system is idle.

When a player runs an AutoTrade, we have what we conceptually (though not in the code) call an AutoTradeEvent (Figure 33):

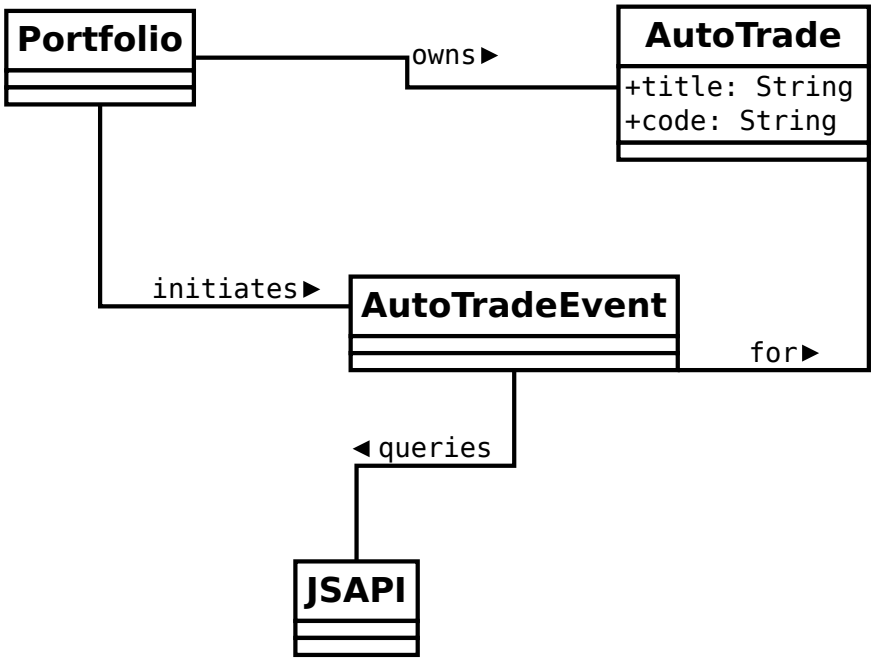


Figure 33: An auto trade being run.

The `JSAPI` is a set of JavaScript functions and corresponding server-side handlers that allow the Auto Trade to actually perform actions. See Running an Auto Trade.

7 Perturbations and Interactions

7.1 Stocks

7.1.1 `allStockHoldings`

Gets all stocks held in PitFail (model/stocks.scala ref_158) Figure 34.

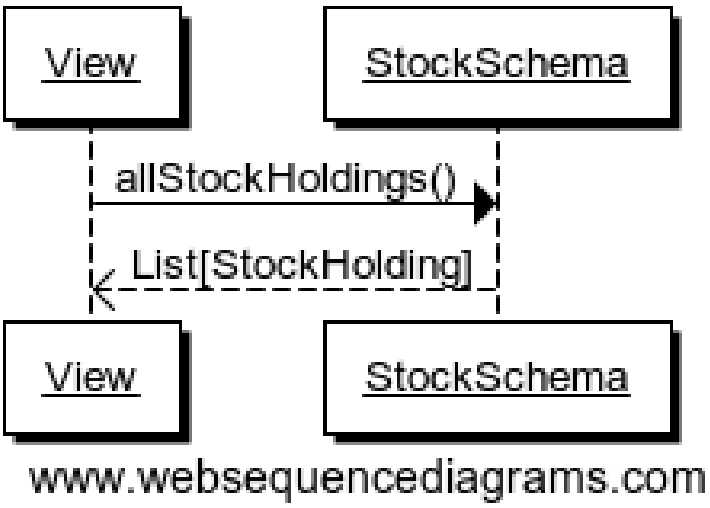


Figure 34: allStockHoldings

7.1.2 `Portfolio.myStockAssets`

Gets stock assets from this portfolio (model/stocks.scala ref_937) Figure 35.

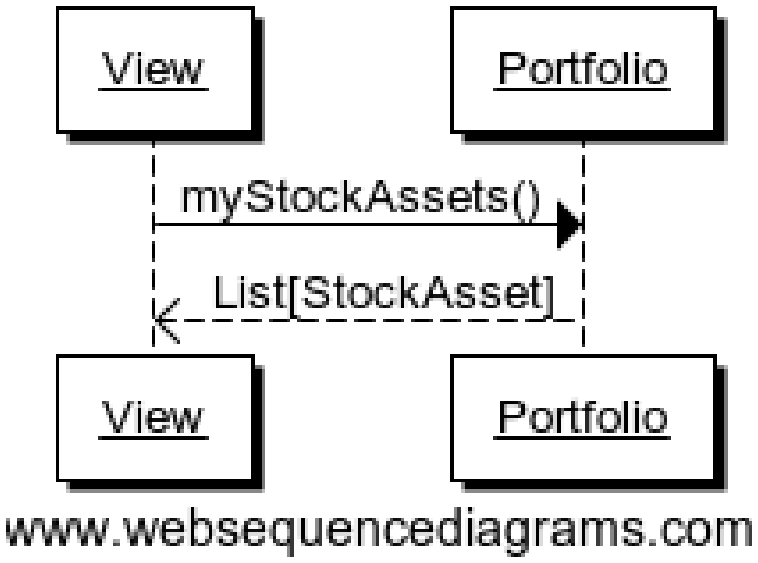


Figure 35: myStockAssets

7.1.3 `Portfolio.haveTicker`

Gets an asset for this stock if we have one, None otherwise (model/stocks.scala ref_407) Figure 36.

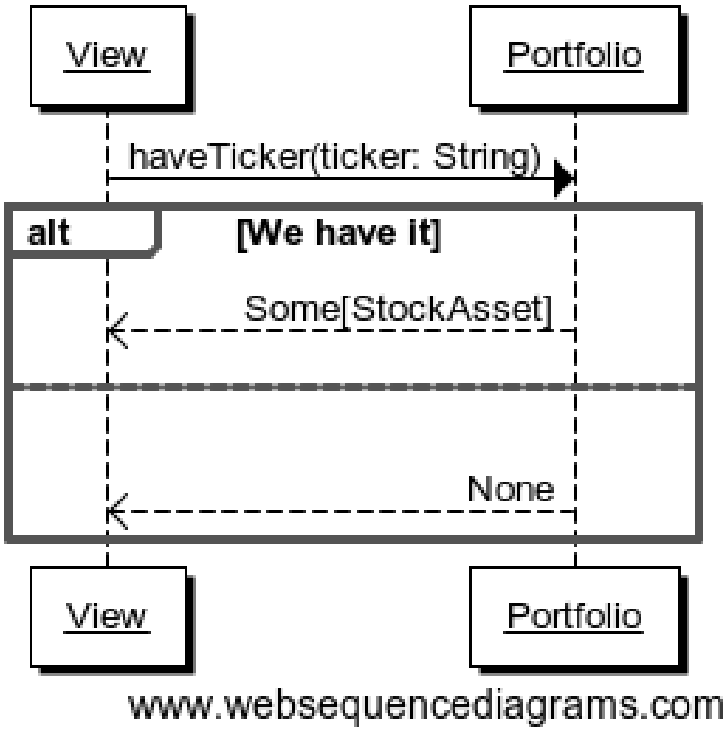


Figure 36: haveTicker

7.1.4 `Portfolio.howManyShares`

Gets how many shares of this stock do we have (model/stocks.scala ref_666) Figure 37.

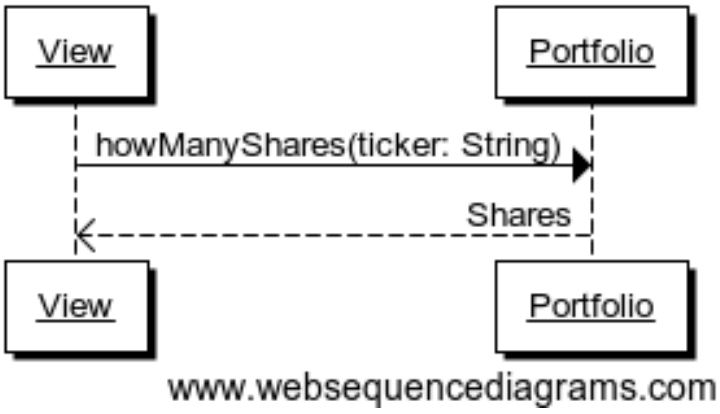


Figure 37: howManyShares

7.1.5 `Portfolio.howManyDollars`

Gets how many dollars (at last traded price) of this stock we have (model/stocks.scala ref_873) Figure 38.

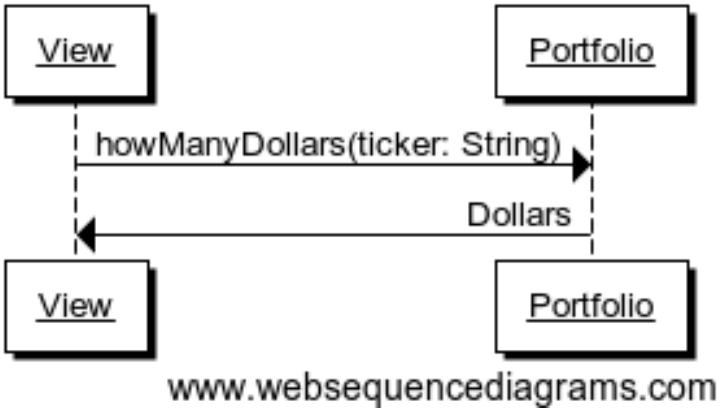


Figure 38: howManyDollars

7.1.6 `Portfolio.userBuyStock`

Attempts to make a market-order purchase of a stock (model/stocks.scala ref_850) Figure 39.

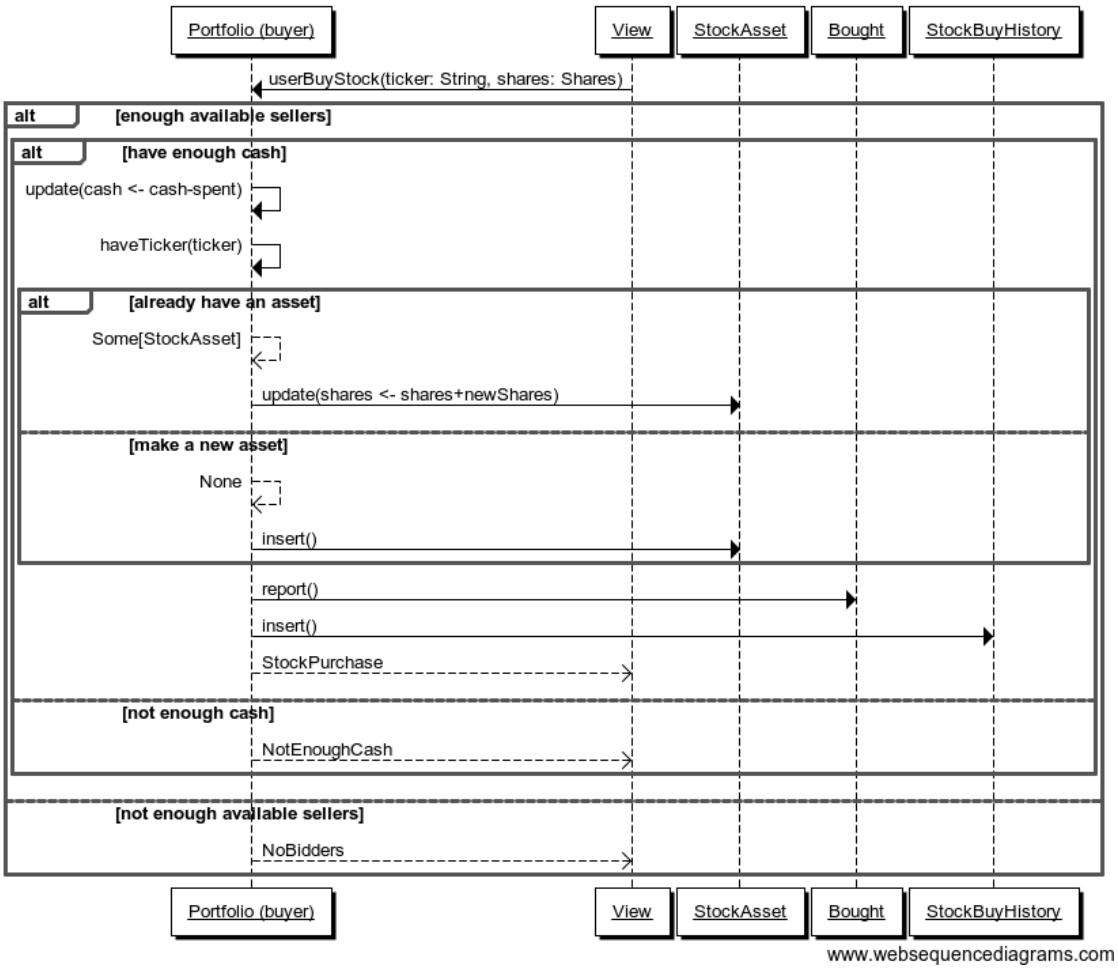


Figure 39: userBuyStock

7.1.7 `Portfolio.userSellStock`

Makes a sell market order for a stock (model/stocks.scala ref_620) Figure 40.

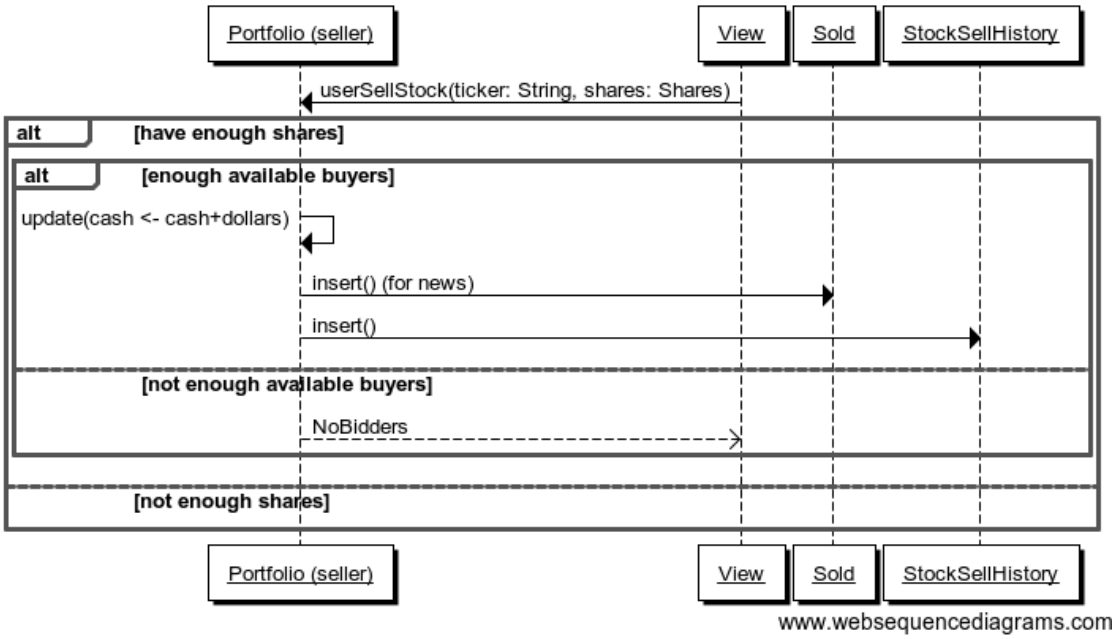


Figure 40:

7.1.8 `Portfolio.userSellAll`

Sells all of the shares we own (with a market order)(model/stocks.scala ref_306) Figure 41.

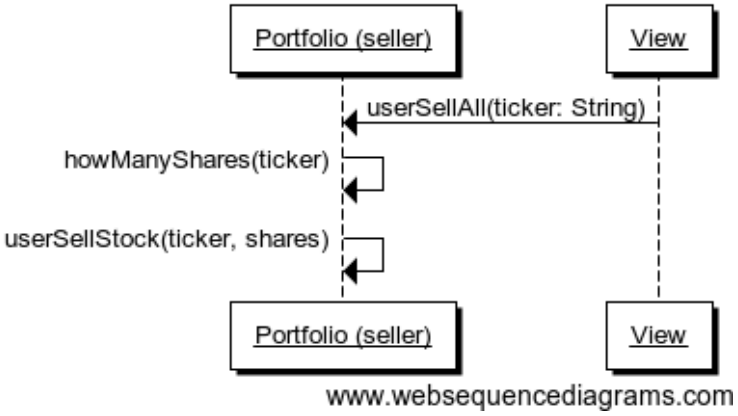


Figure 41: userSellAll

7.1.9 `Portfolio.userMakeBuyLimitOrder`

Places a buy limit order. This involves first executing all of the order that can be executed immediately (ie there are available sellers below the limit) and then deferring the rest until another available seller comes in (model/stocks.scala ref_184) Figure 42.

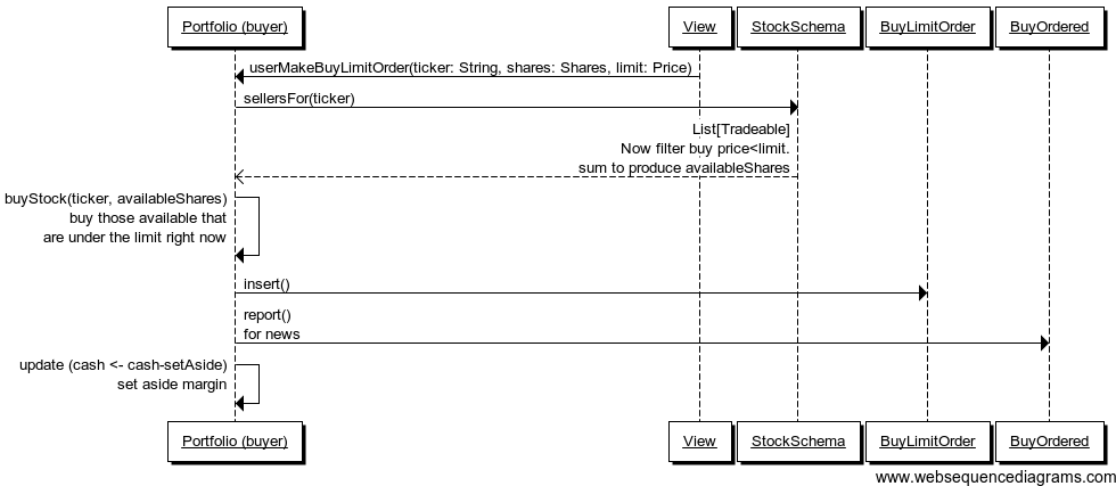


Figure 42: userMakeBuyLimitOrder

7.1.10 `Portfolio.userMakeSellLimitOrder`

Places a sell limit order. This involves executing all that can be executed immediately (where ther are available buyers above the limit) and then defers the rest (model/stocks.scala ref_939) Figure 43.

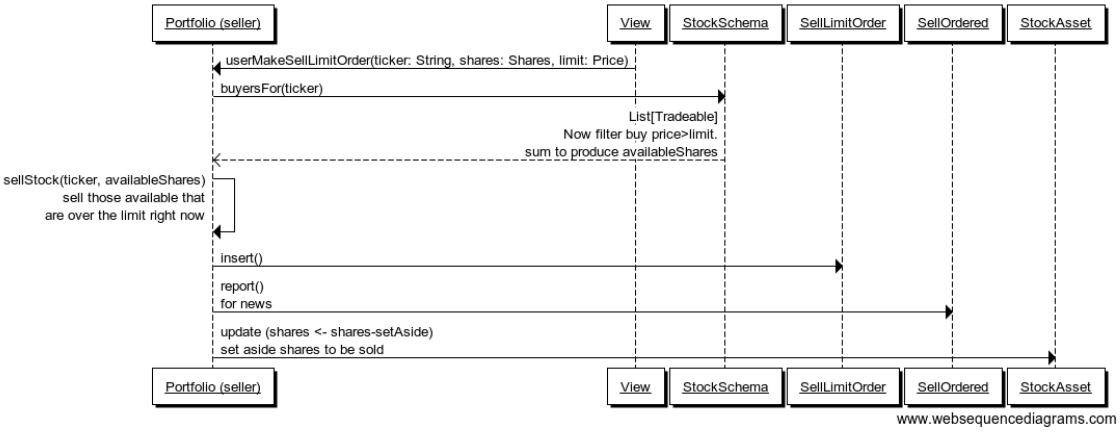


Figure 43: userMakeSellLimitOrder

7.1.11 `Portfolio.myBuyLimitOrders`

Gets all pending buy limit orders (model/stocks.scala ref_734) Figure 44.

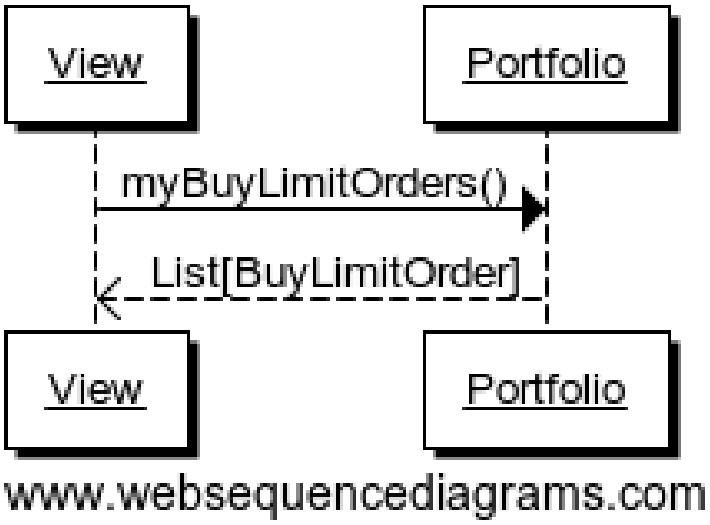


Figure 44: myBuyLimitOrders

7.1.12 `Portfolio.mySellLimitOrders`

Gets all pending sell limit orders (model/stocks.scala ref_680) Figure 45.

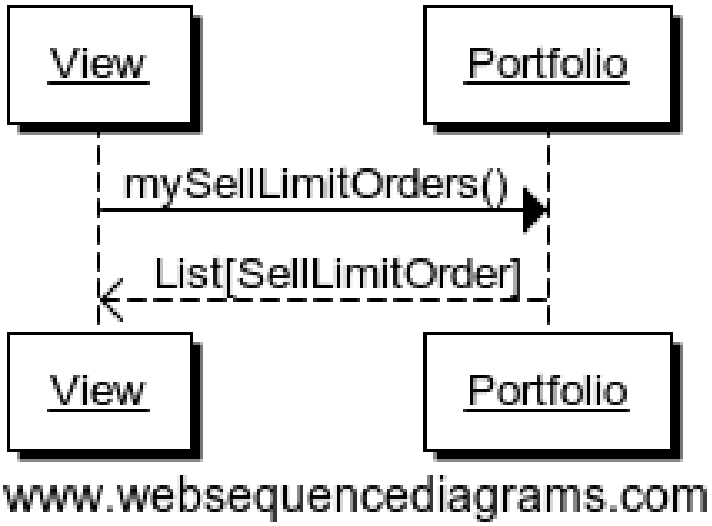


Figure 45: mySellLimitOrders

7.1.13 `Portfolio.margin`

Calculates the current margin that has been set aside (model/stocks.scala ref_224) Figure 46.

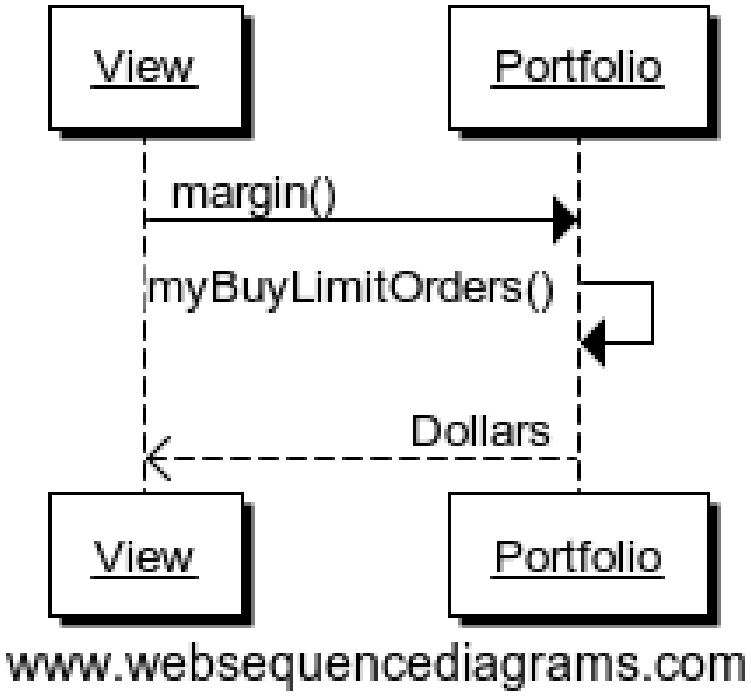


Figure 46: margin

7.2 Derivatives

7.2.1 Exercising Derivatives

When a derivative is exercised, the goal is to move the securities from their source (seller or buyer's portfolio) to their destination (buyer or seller's portfolio). When this is possible, the procedure is easy; the only complications that arise are when this is not possible (model/stocks.scala ref_519).

7.2.1.1 Moving Dollars Say $100 dollars needs to move from A to B. If A has $100, $100 is deducted from A's cash, and added to B's cash.

If A does not have $100, as much as possible is deducted and added to B's cash. this should begin a process of margin call and forced liquidation, but PitFail does not support this feature at this time (model/derivatives.scala ref_392).

7.2.1.2 Moving Stocks Say 100 shares of MSFT need to be moved from A to B. If A has 100 shares of MSFT, they are deducted from A's portfolio and added to B's.

If A does not have 100 shares of MSFT, the following steps are taken:

1. First, A (under the control of the system, not the human player) attempts to buy 100 shares of MSFT at 15% above the last traded price. This is similar to a limit order in that the trade will execute at the ask price if the ask price is less than 1.15*(last trade price). This attempt to buy may be partially or completely executed (if there are shares available), or not at all.

2. If, after attempting to buy the remaining shares, A *still* does not thave 100 shares MSFT, pays the remaining debt to B in cash, at 1.15*(last trade price)*(shares unaccounted for).

3. If A does not have enough shares *or* enough cash, this should generate a margin call and A's assets should be liquidated, but PitFail does not support this feature.

This procedure for moving stocks differs significantly from the old procedure (as of demo #1), because in the old version it was always possible to buy an unlimited amount of a stock. When this became no longer possible, it was necessary to design a system that would respect the limited volume available but still be largely automatic; since we do not expect PitFail players want to be bothered by an online game to resolve the issue. Hence the 15% premium -- high enough to give a user an incentive to actually own the stocks promised, but not so high as to make it a disaster if they do not (model/derivatives.scala ref_411).

7.2.1.3 Moving Derivatives This feature was removed from the most recent version of PitFail because the UI still does not support creating a derivative that refers to another derivative (making the support in the backend moot). In the old version, the way this worked was that, if A owned the specified amount of the specified derivative, it would be moved. If not, a *new* derivative would be created with terms identical to the desired ones, for which A would hold the liability and B the asset.

7.2.2 `Portfolio.myDerivativeAssets`

Gets all derivative assets we own (model/derivatives.scala ref_74) Figure 47.

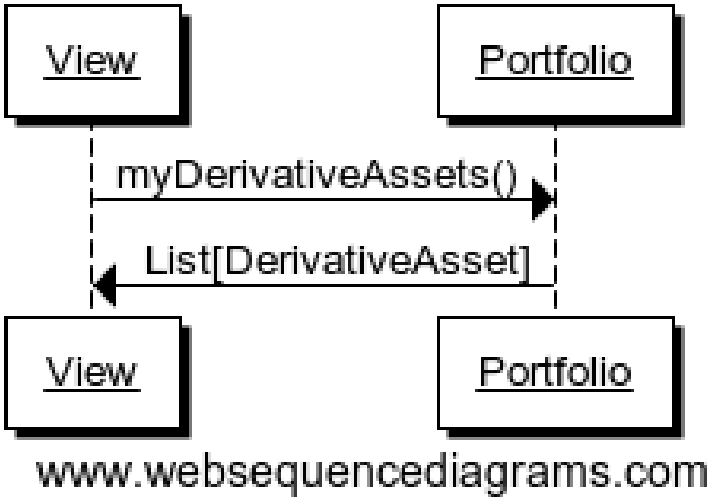


Figure 47: myDerivativeAssets

7.2.3 `Portfolio.myDerivativeLiabilities`

Gets all deriavtive liabilities we own (model/derivaives.scala ref_484) 48.

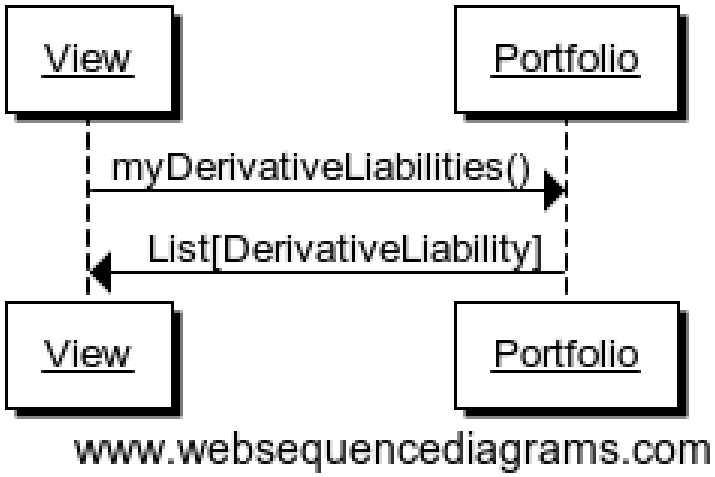


Figure 48: myDerivativeLiabilities

7.2.4 `Portfolio.myDerivativeOffers`

Gets all derivative offers that have been sent to us and not yet accepted/rejected (model/derivatives.scala ref_462) 49.

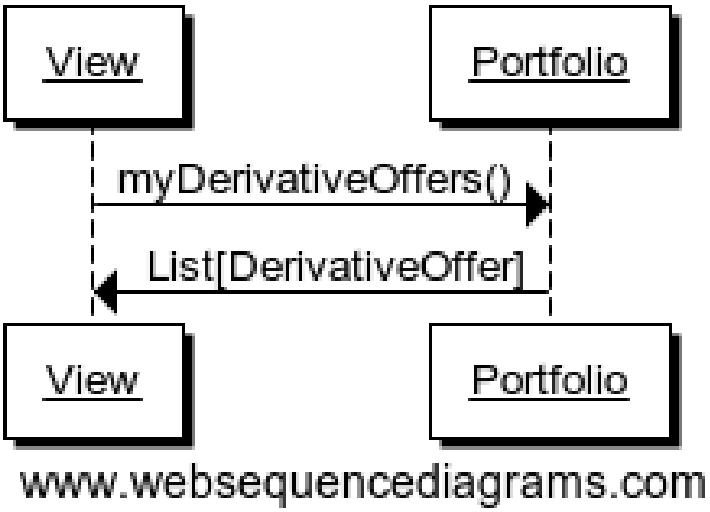


Figure 49: myDerivativeOffers

7.2.5 `Portfolio.userOfferDerivativeTo`

Offers a derivative to another user (model/derivatives.scala ref_6) 50.

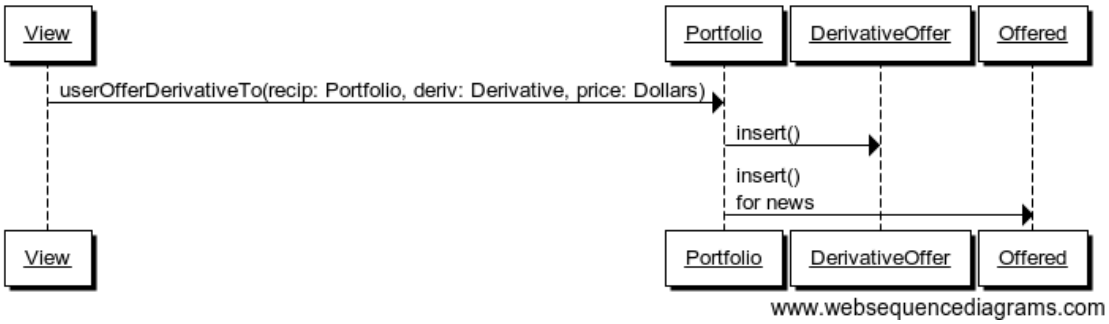


Figure 50: userOfferDerivativeTo

7.2.6 `Portfolio.userOfferDerivativeAtAuction`

Offers a derivative at auction (model/derivatives.scala ref_674) 51.

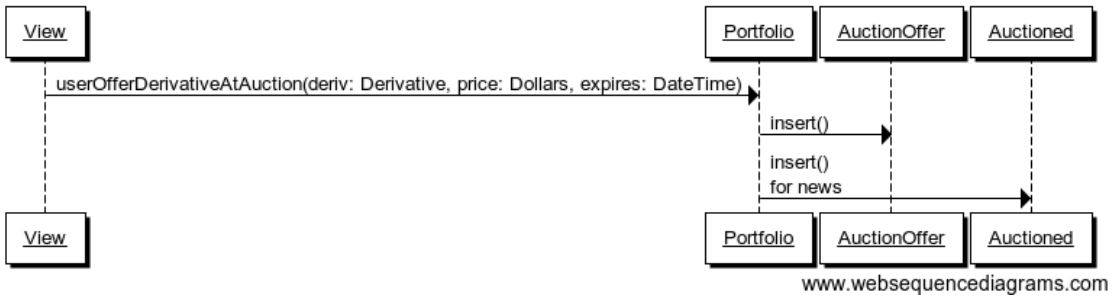


Figure 51: userOfferDerivativeAtAuction

7.2.7 `Portfolio.userAcceptOffer`

Accepts a derivative offer (model/derivatives.scala ref_699) Figure 52.

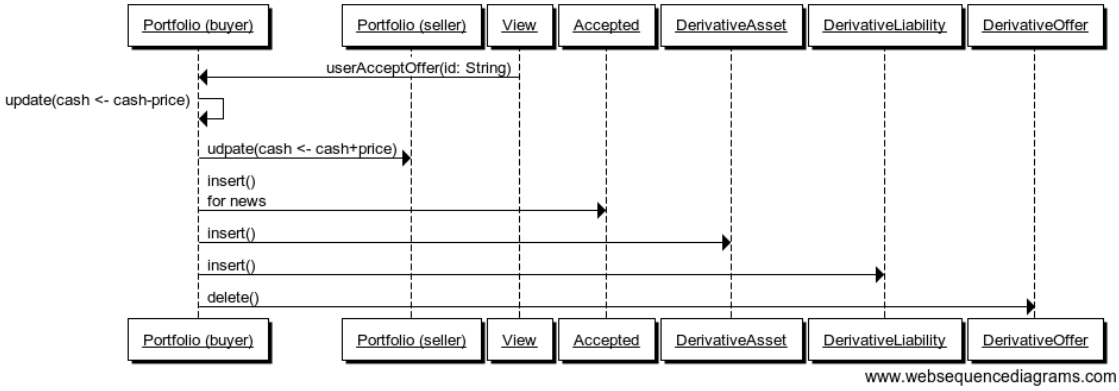


Figure 52: userAcceptOffer

7.2.8 `Portfolio.userDeclineOffer`

Declines a derivative offer (model/derivatives.scala ref_650) Figure 53.

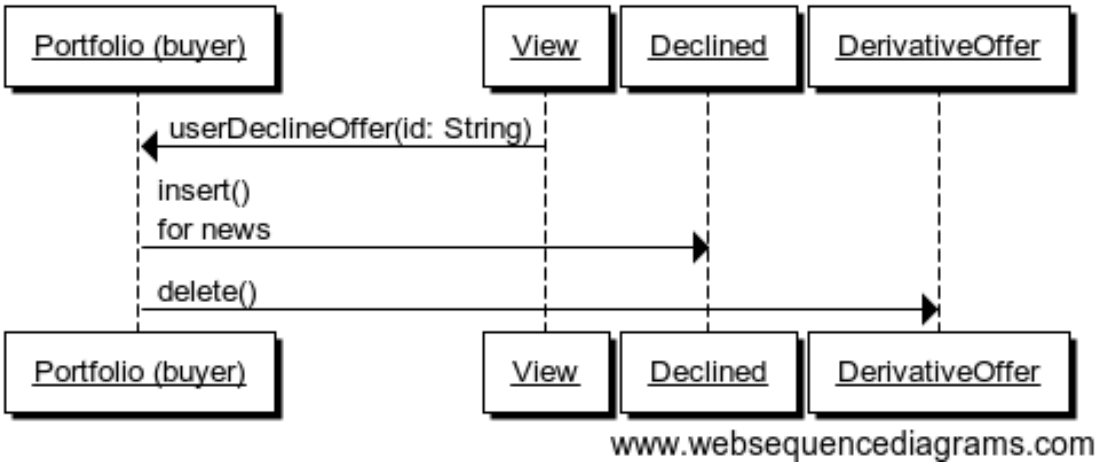


Figure 53: userDeclineOffer

7.2.9 `DerivativeAsset.userExecuteManually`

Exercise a derivative before its scheduled exercise date (model/derivatives.scala ref_583) Figure 54.

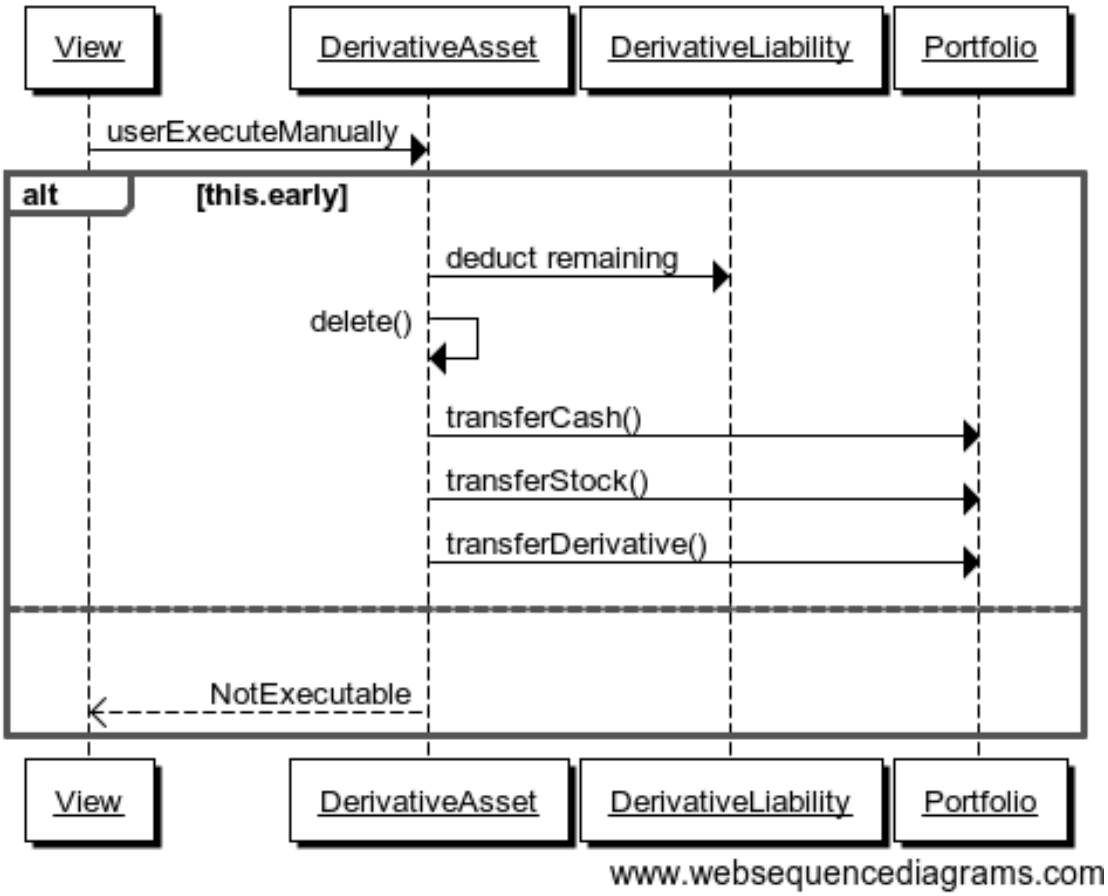


Figure 54: userExecuteManually

7.2.10 `DerivativeAsset.systemExecuteOnSchedule`

Executes a derivative on its scheduled exercise date, provided that the contracted condition holds (model/derivatives.scala ref_289) Figure 55.

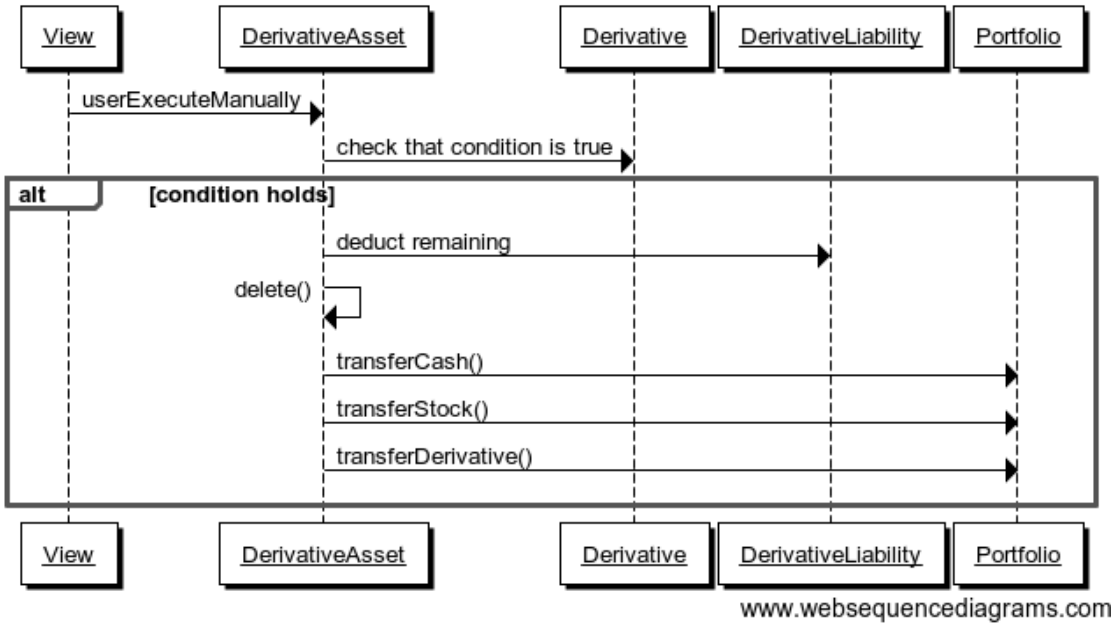


Figure 55: systemExecuteOnSchedule

7.2.11 `DerivativeAsset.spotValue`

Gets how much a derivative would be worth should it be exercised today (model/derivatives.scala ref_319) Figure 56.

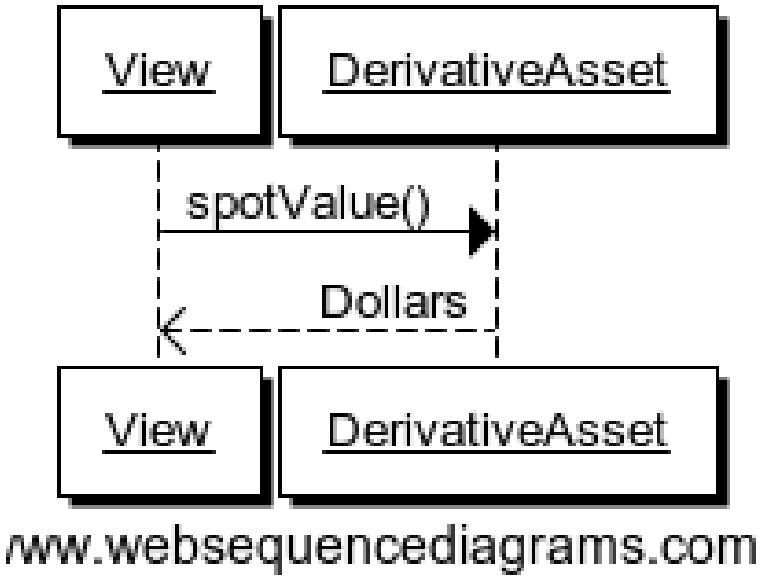


Figure 56: spotValue

7.3 Dividends

7.3.1 `DividendSchema.systemCheckForDividends`

Checks for new dividends, and credits them if there are (model/dividends.scala ref_789) Figure 57.

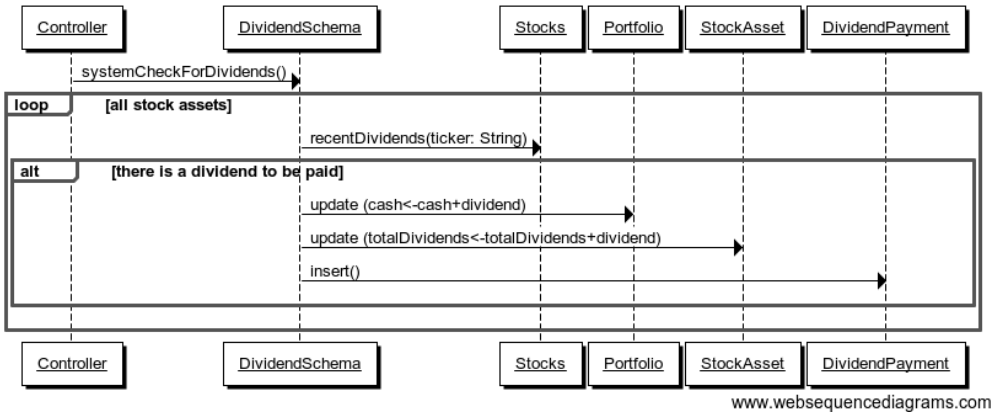


Figure 57: systemCheckForDividends

7.3.2 `Portfolio.myDividendPayments`

Gets a list of dividend payments that we have received (model/dividends.scala ref_489) Figure 58.

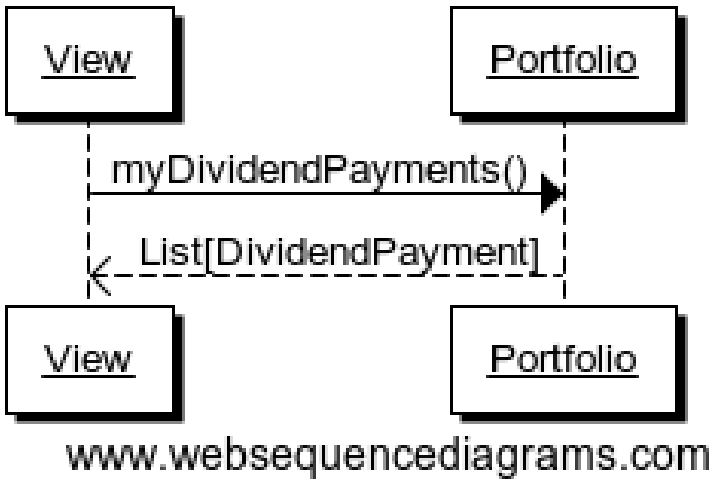


Figure 58: myDividendPayments

7.4 Voting

7.4.1 `Portfolio.userVoteUp`

Casts an up-vote on a trade (model/voting.scala ref_805) Figure 59.

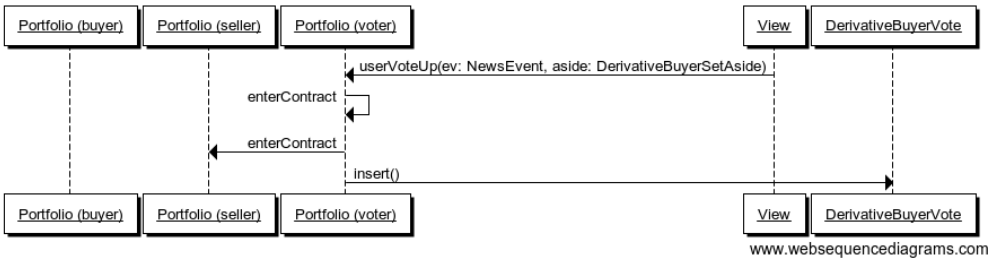


Figure 59: userVoteUp

7.4.2 `Portfolio.userVoteDown`

Casts a down-vote on a trade (model/voting.scala ref_940) Figure 60.

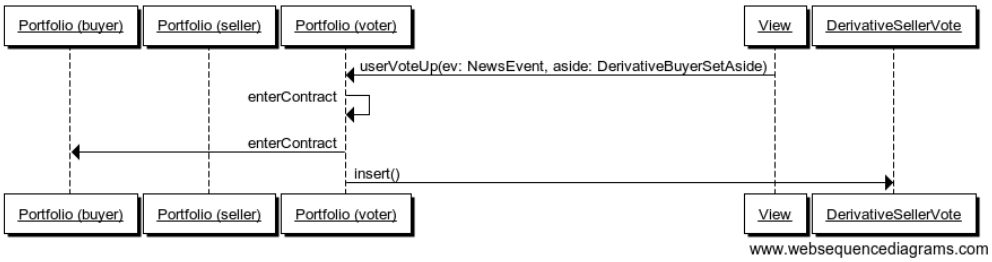


Figure 60: userVoteDown

7.4.3 `NewsEvent.buyerVotes`

Gets all for-buyer votes on this event (model/voting.scala ref_146) Figure 61.

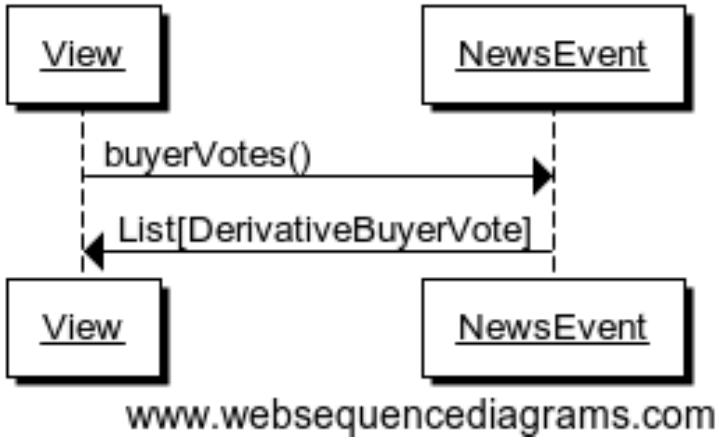


Figure 61: buyerVotes

7.4.4 `NewsEvent.sellerVotes`

Gets all for-seller votes on this event (model/voting.scala ref_405) Figure 62.

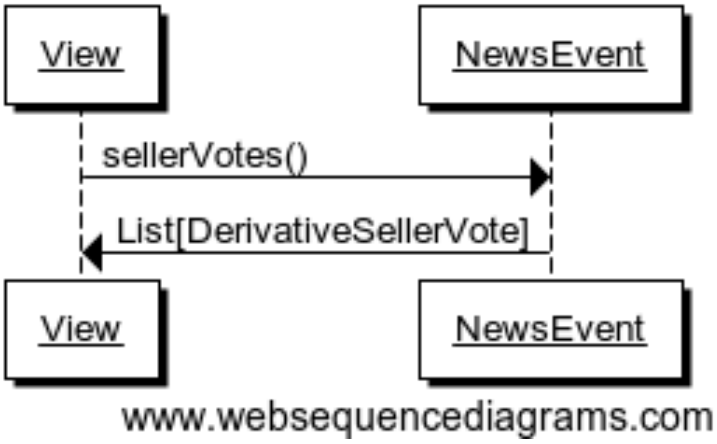


Figure 62: sellerVotes

7.5 Comments

7.5.1 `User.userPostComment`

Posts a comment on an event (model/comments.scala ref_494) Figure 63.

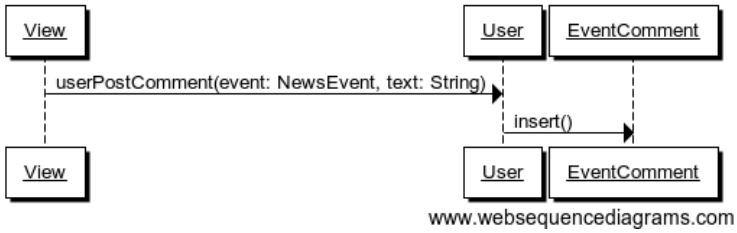


Figure 63: userPostComment

7.5.2 `NewsEvent.comments`

Get comments associated with this event (model/comments.scala ref_449) Figure 64.

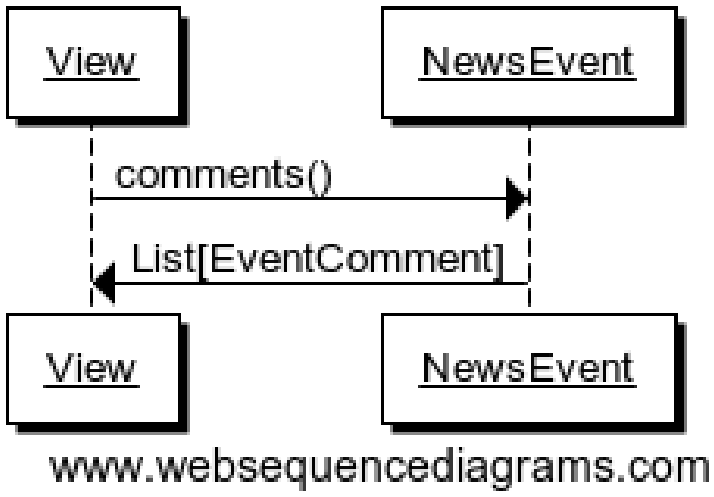


Figure 64: comments

7.6 Auto Trades

Auto trades have a more complicated flow of control than other parts of the code, because execution is split between the server and the client (website/jsapi/jsapi.scala).

7.6.1 Running an Auto Trade

I'm hoping the following diagram is clearer than it would be as a sequence diagram Figure 65:

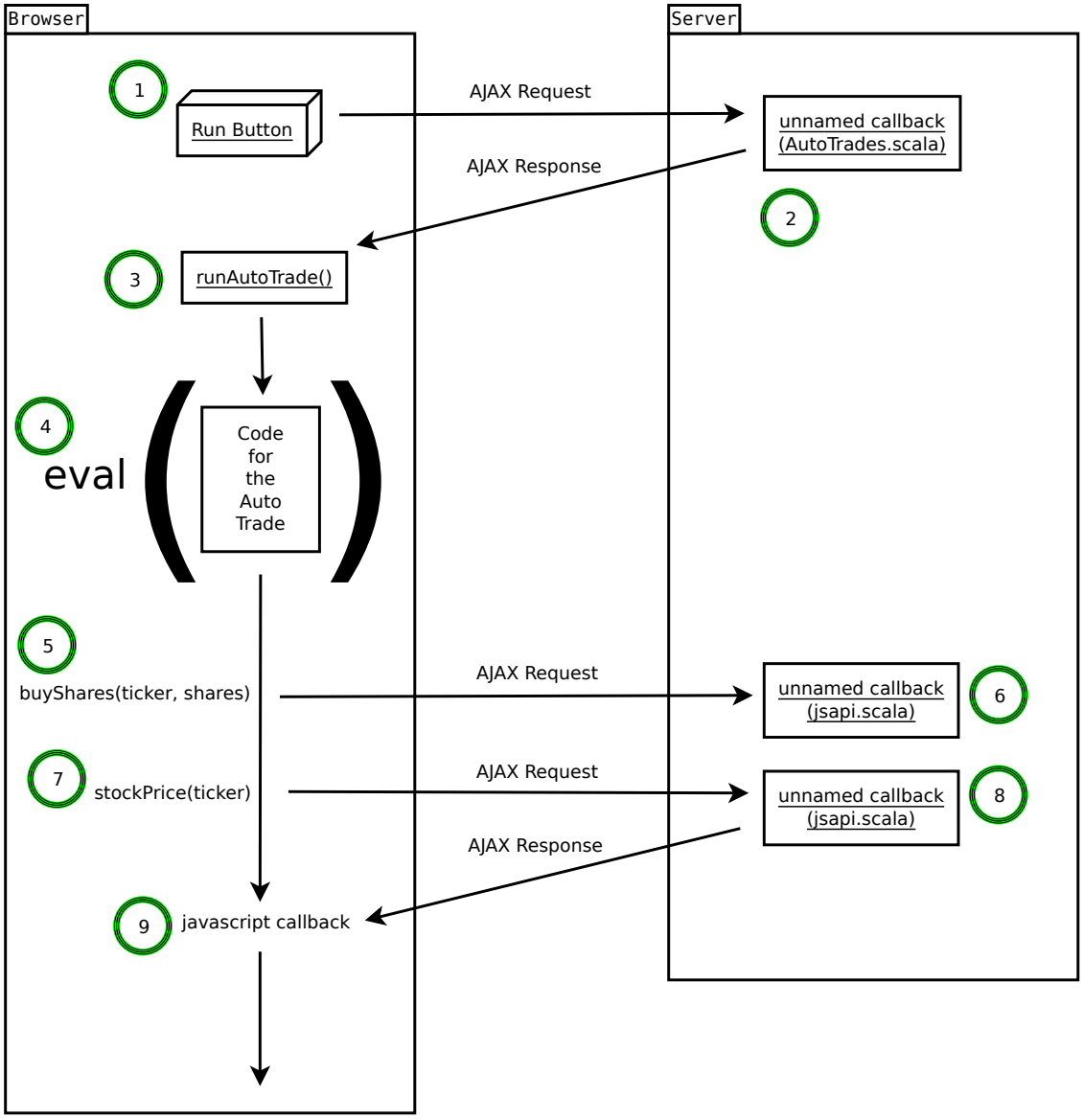


Figure 65: The full sequence of running an auto-trade.

This corresponds to the following Auto-Trade code (in JavaScript -- what the user types in):

```
buyShares('MSFT', 100)
stockPrice('MSFT', function(price) {
    alert(price)
})
```

The steps are:

1. The user presses the "Run" button. This sends an AJAX request to the server.
2. A callback in the Scala code (website/view/AutoTrades.scala ref_73) receives the AJAX request and sends a response in the form of a JavaScript command to be executed on the client [Ajax].

3. The JavaScript command gets the users AutoTrade out of the textarea, which is also a segment of JavaScript (website/jsapi/jsapi.scala ref_188).

4. The user's code is evaluated with `eval()` (website/jsapi/jsapi.scala ref_188).

5. The user's code makes an API call -- in this case `buyShares(ticker,shares)`. `buyShares()` is a JavaScript function that lives in the client (website/jsapi/jsapi.scala ref_405), and that makes an AJAX request to the server (website/jsapi/jsapi.scala ref_867).

6. The server receives the AJAX request and performs the operation (buying a stock) (website/jsapi/jsapi.scala ref_645).

7. The user's code makes another request -- but this one is different because the user's code needs a reply.

8. A callback in the Scala code receives the request, gets the data, and constructs a response that consists of a JavaScript object (the price) (website/jsapi/jsapi.scala ref_18).

9. The user's callback is invoked with the response (website/jsapi/jsapi.scala ref_867).

7.6.2 Creating

This creates a new (blank) auto trade (model/auto.scala ref_168) 66.

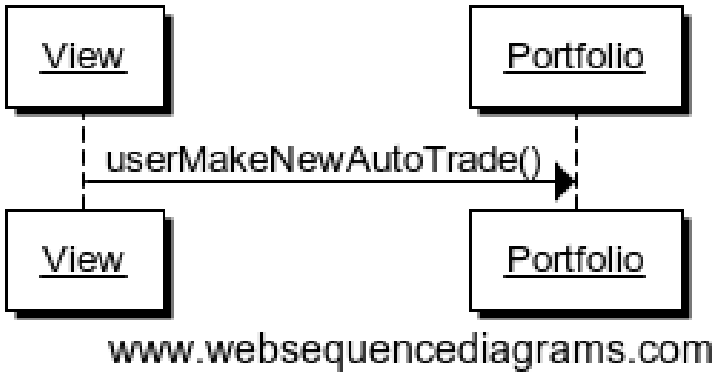


Figure 66: userMakeNewAutoTrade

7.6.3 Modifying

This updates the stored information about an auto-trade (model/auto.scala ref_337) Figure 67.

[Ajax] `http://exploring.liftweb.net/master/index-11.html`

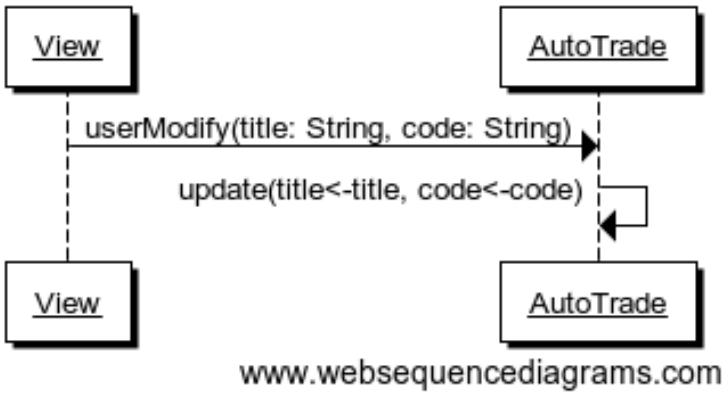


Figure 67: userModify

7.6.4 Deleting

This deletes an auto trade (model/auto.scala ref_309) Figure 68.

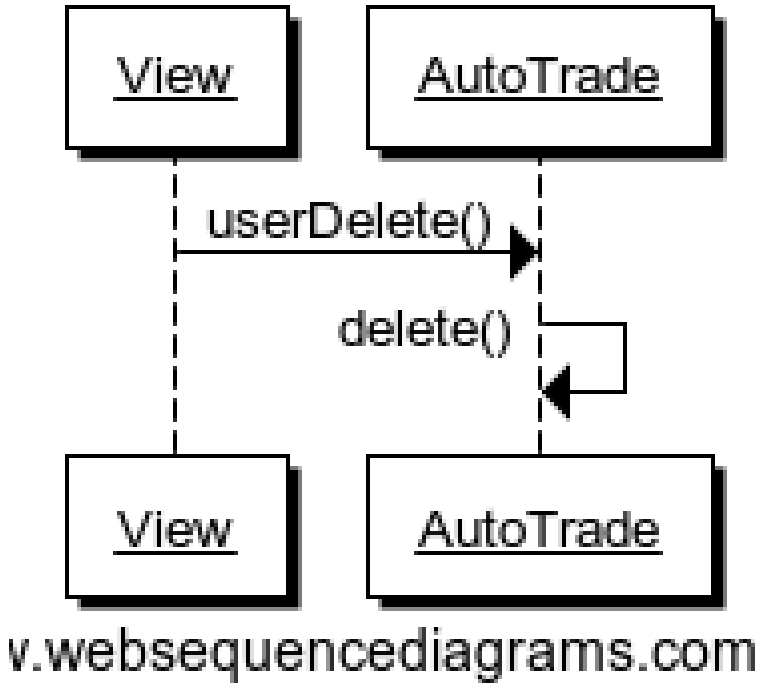


Figure 68: userDelete

7.6.5 Getting all auto trades

This gets all the auto trades associated with a portfolio (auto trades are associated with portfolios, not users (see the domain model)) (model/auto.scala ref_900) Figure 69.

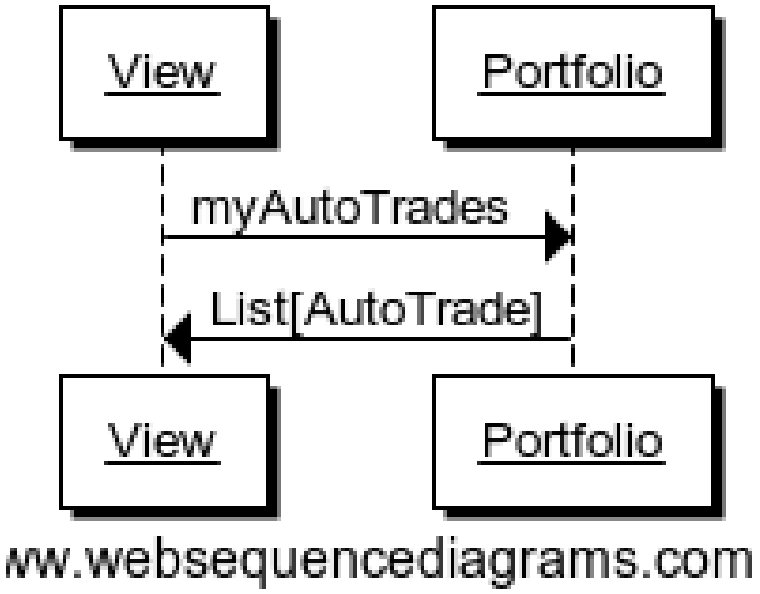


Figure 69: myAutoTrades

7.7 News

7.7.1 Getting recent news events

This gets the most recent events that have been reported (model/news.scala ref_531) Figure 70.

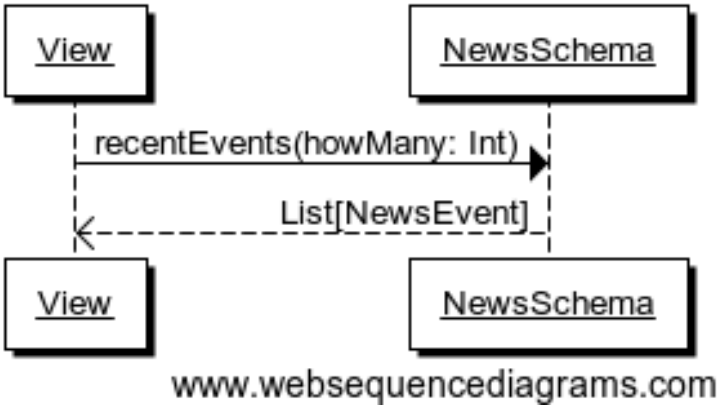


Figure 70: recentEvents

7.7.2 Reporting an event

The API into reporting events is the `report()` method in the class `Action`, which takes the action, associates a timestamp with it, and adds it to the list of all events that have occurred (model/news.scala ref_121) Figure 71.

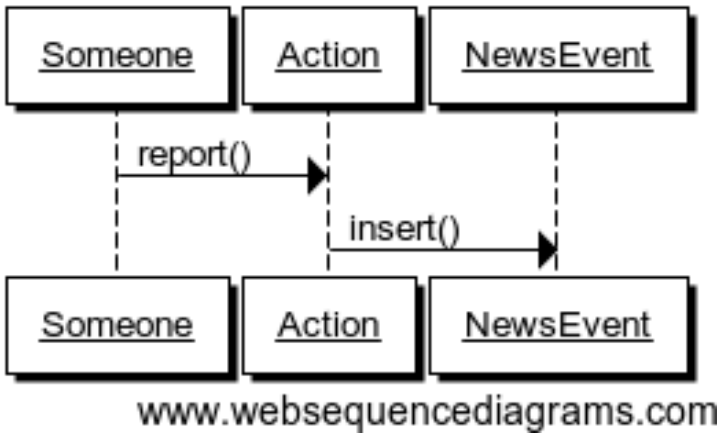


Figure 71: Reporting a news event.

7.8 Auctions

7.8.1 Offering a derivative at auction

This creates a new auctioned item (model/derivatives.scala ref_674) Figure 72.

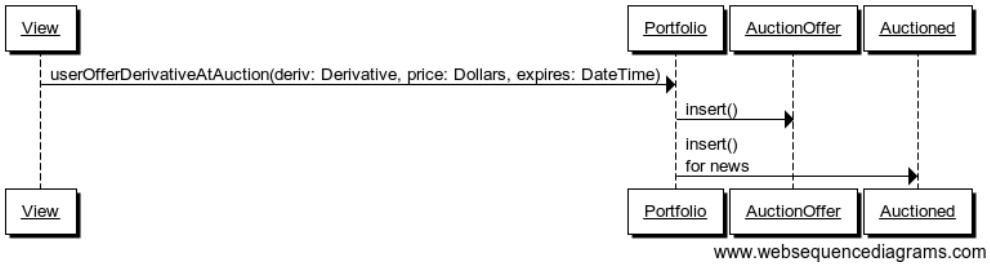


Figure 72: userOfferDerivativeAtAuction

7.8.2 Bidding on an auction

This casts a bid on an auction item (model/auctions.scala ref_861) Figure 73.

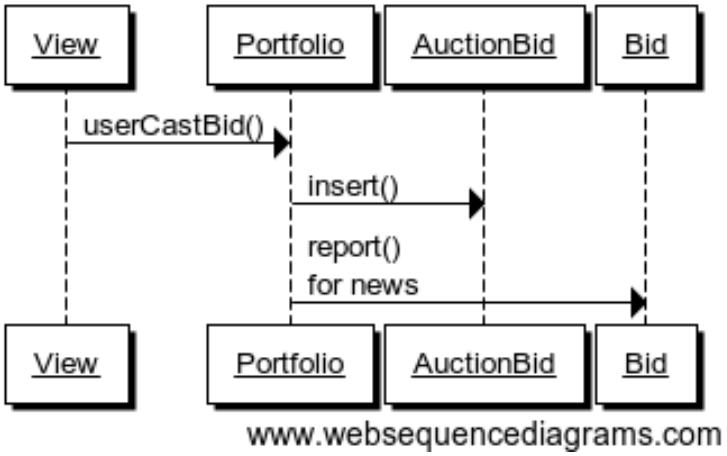


Figure 73: userCastBid

7.8.3 Getting the current high bid

This gets the current high id, if there is one (if no bids have been cast, there will be no high bid) (model/auctions.scala ref_188) Figure 74.

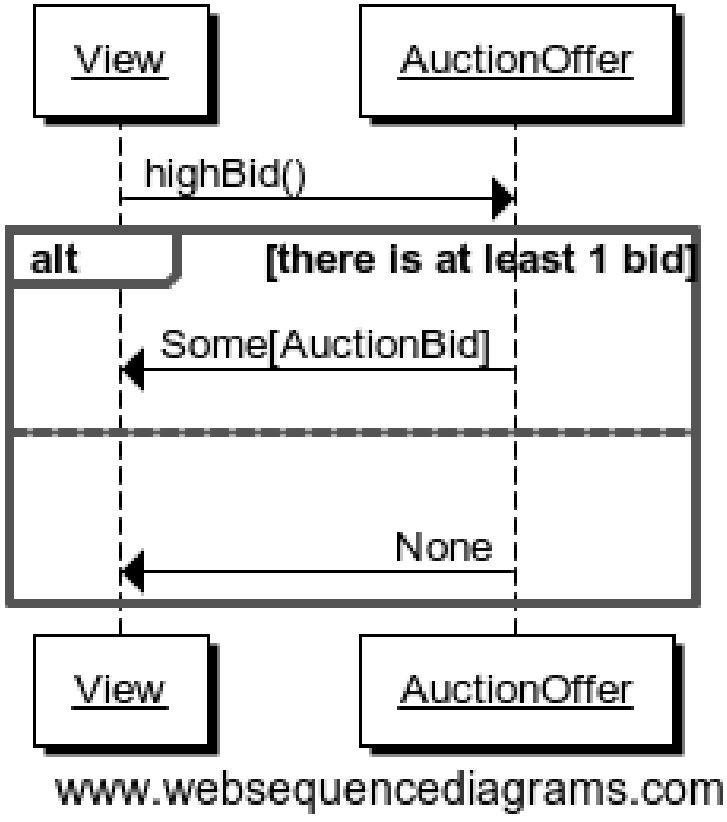


Figure 74: highBid

7.8.4 Closing an auction

Closing an auction results in entering a derivative contract. See the sections on derivatives for an explanation of what this means (model/auctions.scala ref_870) Figure 75.

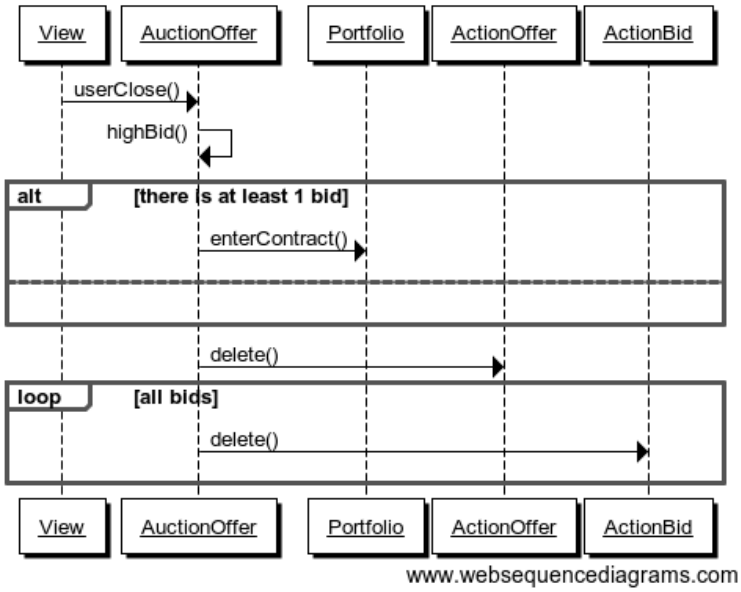


Figure 75: userClose

7.8.5 Buy Via Android Cleint

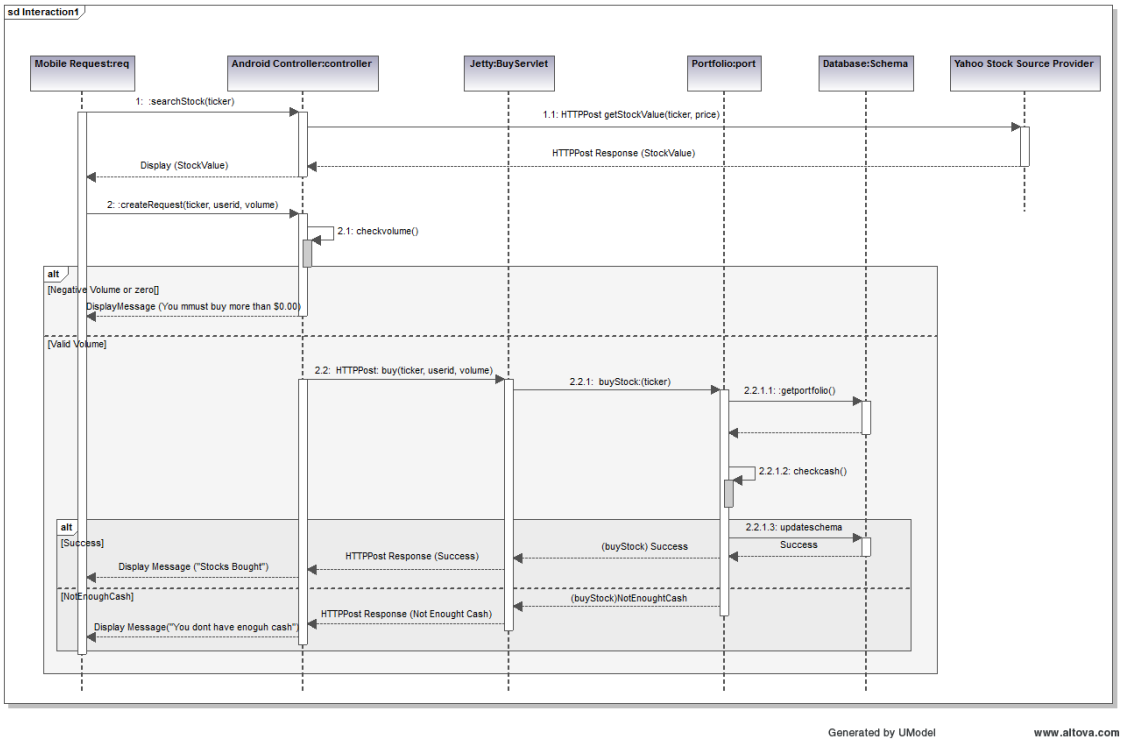


Figure 76: Buy Stocks via Android Client

The diagram above (Figure 76) is the interaction sequence diagram for UC Buy Stocks from an Android Mobile Client. This Interaction diagram is the extension of System sequence Diagram for UC-1 Buy Stocks. As shown, first the search action is initiated by the Android Controller which requested by the Android user. The Android controller sends an HTTP Post request to Yahoo Stock Source. This request specifically asks for the Stock Value of the stock ticker by sending the corresponding tag with the request. Once the response is received, the Mobile Client creates the Buy request. The Android controller calls the BuyServlet using an HTTP Post request via the Jetty Server.The Jetty server has capability to support both Scala and Java sources as it runs on a JVM. All the servlets for Android are written in Java which internally calls functions from Scala classes. The reason for choosing Java for Android client is for its compatibility.The BuySerlvlet internally makes use of the Portfolio class the extract the user info from the Database. If the Volume to be bought is correct, user's portfolio is updated and results are sent back to the user.

7.8.6 Sell Via Android Cleint

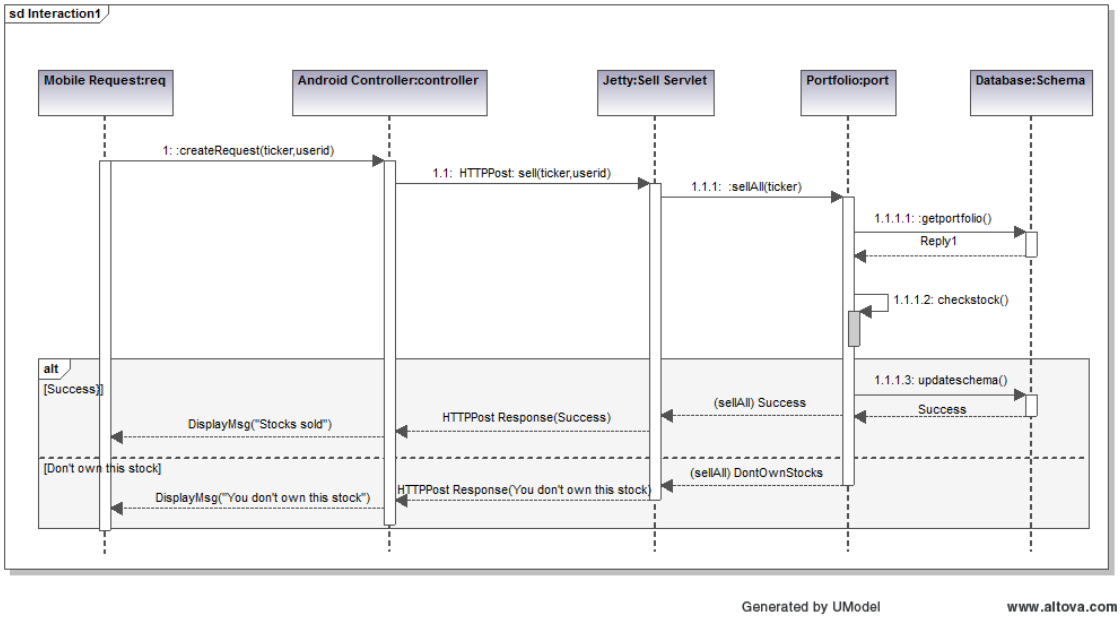


Figure 77: Sell Stocks via Android Client

The diagram above (77) is the interaction sequence diagram for UC Sell Stocks from an Android Mobile Client. The user initiates the action by creating a request by providing the Stock ticker name he intends to sell off. The Android controller sends an HTTP Post request to SellServlet via the Web Server. The BuyServlet makes use of portfolio class and call the function to update the user profile.Because we expect asynchronous requests there is a possibility that by the time a SellStock is completely executed there can be another asynchronous call from some other client interface by the same user. Such a situation is handled by throwing back an exception message “You dont own this stock” and corresponding appropriate message back to the user.Currently, we sell off all the corresponding stocks.

7.9 Notifications for Android Client

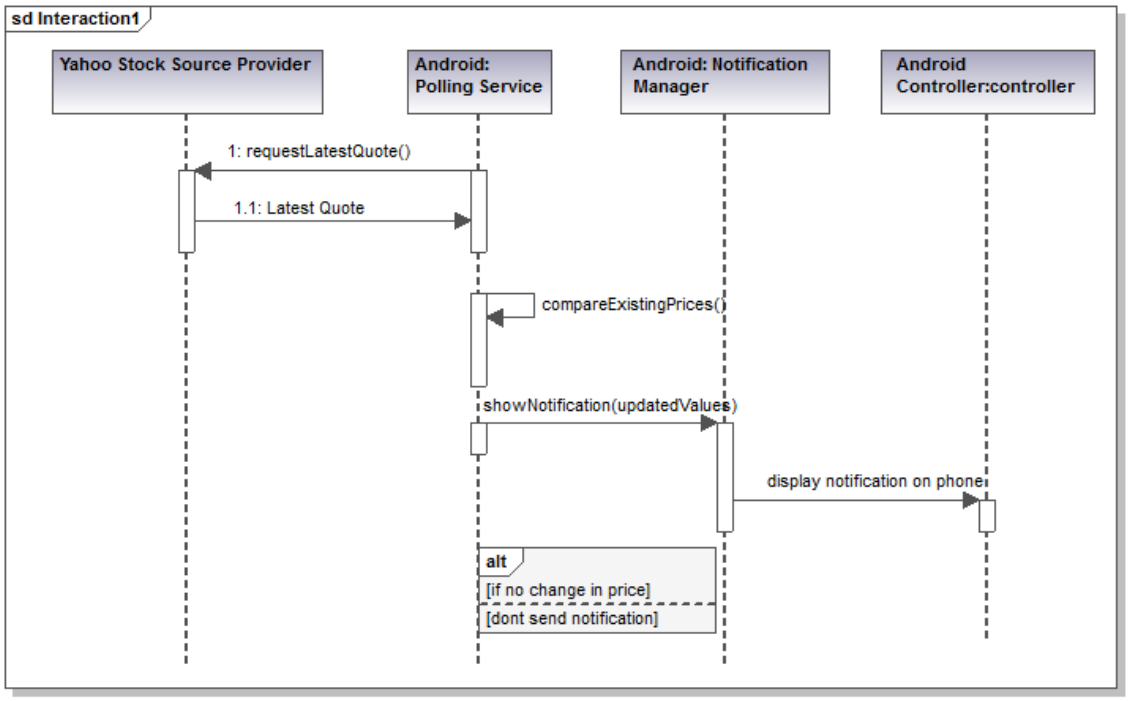


Figure 78: Sell Stocks via Android Client

When the user starts the Pitfail Application for the first time, a background service is started with it which is not bounded to the application. This is a Polling service which polls the Web Server periodically. On receiving the request from the service, the server executes the Stock updates Servlet which collects information on any change in the price of all the stocks the user owns. If the margin of change is equal to more than 1 dollar, the corresponding updates are sent to the polling service. The Polling service then sends those messages to the Android Notification manager. The Notification manager then display new notifications as Stock updates for the user. If there is a previous notification which is not yet viewed by the user, the previous notification is updated and there is just one latest notification available for the user to view.

7.10 FaceBook Operations:

Facebook interface currently supports 4 operations:

1. Buy Stocks.
2. Sell stocks.
3. View Portfolio.
4. View Leaderboard.

If a player wants to access PitFail via Facebook, he or she can post the request on PitFail's wall in the following format:

Username: Operation(Buy/Sell):[volume]:[Ticker]

Arguments in square brackets are optional. For example, View portfolio and view leaderboard operations do not take volume and ticker as arguments.

The request posted on the wall needs to be processed. To process this request :

1.This request should be listened to and FB app should be notified of the wall post

2.The wall post should be read and parsed.

3.The request should invoke appropriate module from server to get the operation done

4.The player should be notified of the status of the request (successful/failed)

The operations takes place partly at Facebook client side and partly at server side.

Here is a description in detail:

7.10.1 FaceBook Client:

Facebook client includes mainly two operations:

1. FBListener -- FBListener listens to our facebook page pitfail and notifies the app controller of any incoming request (a wall post) to be processed.

2. ParseMessage -- ParseMessage parses user's wall post to multiple token , checks if the message follows the required syntax and decides if the message is good enough to be processed. Figure 79

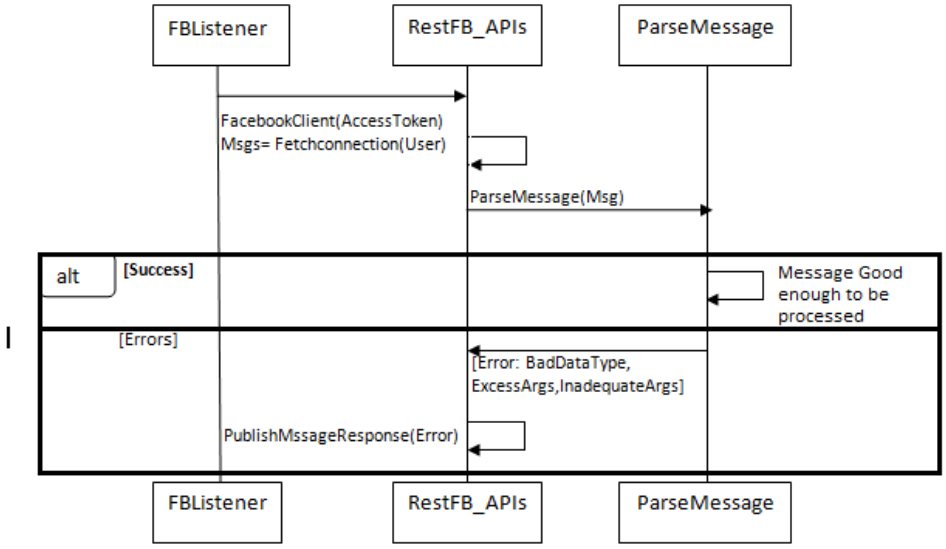


Figure 79:

FBListener listens to the wall post of our account and notifies pitFail FB app of any new wall post. We use RestFB APIs that access Facebook account of PitFail using the unique access token provided by FaceBook. API fetchConnection(User) reads the new wall post and passes it to ParseMessage module. ParseMessage processes the wall post, extracts the information required to process the request. It also checks for the right number of arguments and the data type (e.g. Volume has to be a number, a request to view portfolio does not take more than two arguments).

If the message is good enough to be processed (no errors), client controller calls appropriate functions from the server, otherwise the player is notified of the error by commenting on player's wall post.

7.10.2 Server Operations:

Now once the message is retrieved and parsed at the client side, the server functions are invoked with the parsed tokens as arguments.

Before processing any request, we always check if the username that is requesting this operation is valid or not. Therefore before invoking any other method client invokes EnsureUser method to enusure the authenticity of the user.

7.10.2.1 Ensure User: Facebook interface of PitFail does not (for now) support registration. The player has to be already registered to the system to play the game via FB interface. Figure 80

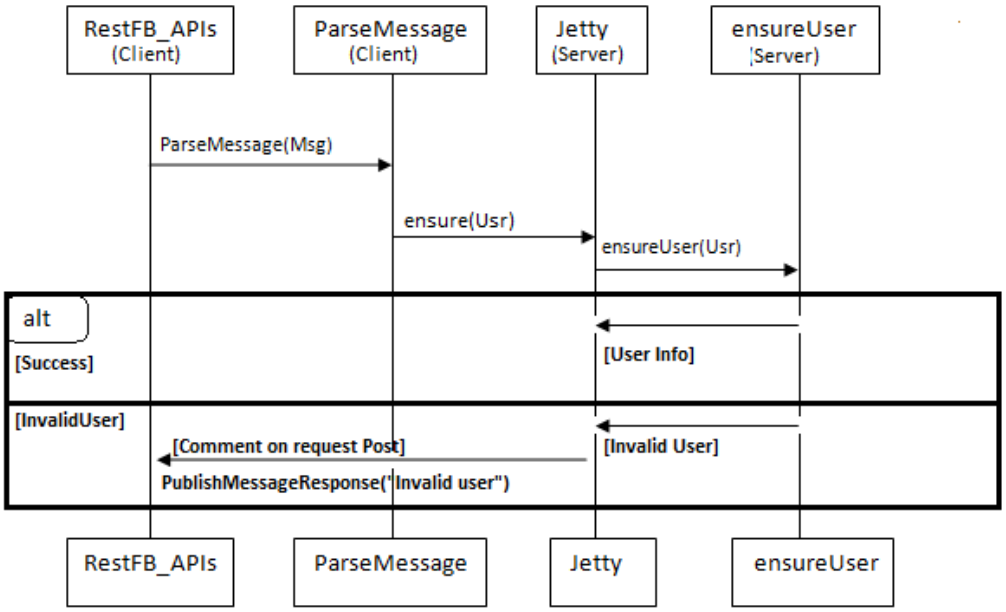


Figure 80:

ensureUser ensures the existence of a user before the user's request tries to access portfolio. If the user exists, the request is processed further otherwise the player is notified of the error occurred by posting a comment on his wall post.

Once the user is checked for his/her authenticity, we can proceed further with the actual operation requested by the user. Below are the operations user can execute.

7.10.2.2 Buy Stock: for all the operations below, once the ensureUser confirms the authenticity of the user, FaceBook client invokes a Java servlet on Jetty server. The main task handled by this java servlet is to accept arguments from Facebook client and invoke appropriate scala mothods to perform task requested by facebook client Here the servlet is: FBBuyServlet(Username) Figure 81

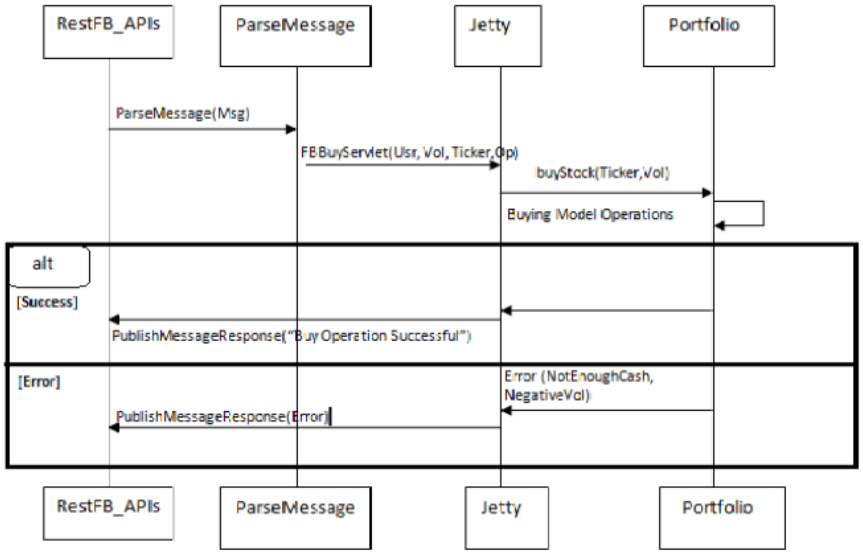


Figure 81:

7.10.2.3 Sell Stock: In sell stock , FBSellServlet() is the Java servlet that accepts arguments from Facebook client and invokes scala method to sell stocks. Figure 82

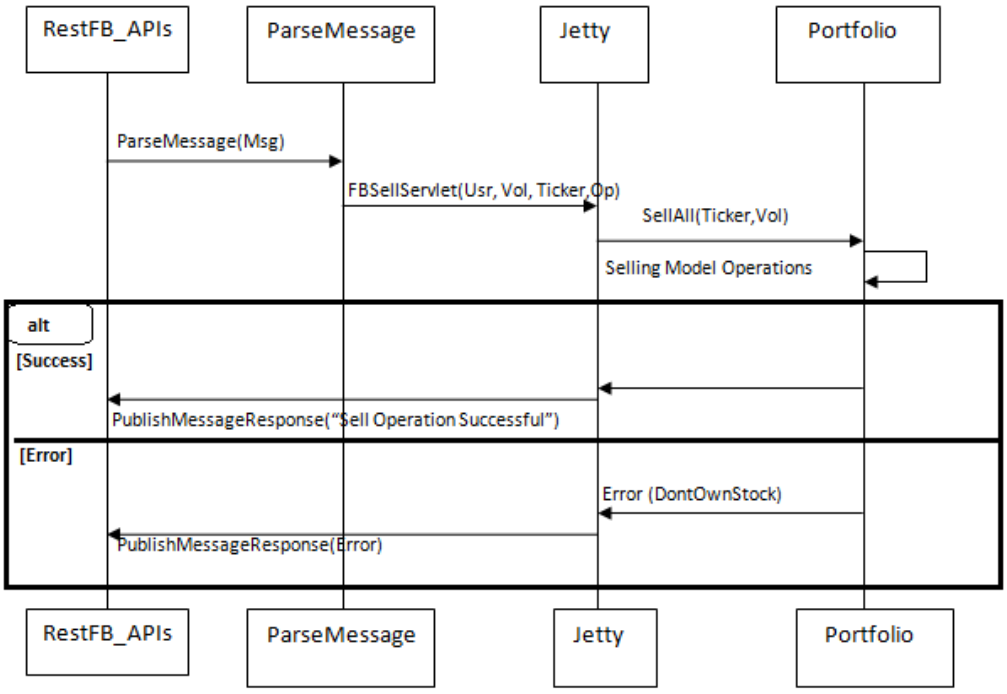


Figure 82:

7.10.2.4 View Portfolio: Before processing any request , we make sure (by invoking ensureUser) that the username exists. Therefore there is no failure flow (alternate flow) for portfolio view. We will invoke this funtion only if the ensureUser confirms that the user exists. Figure 83

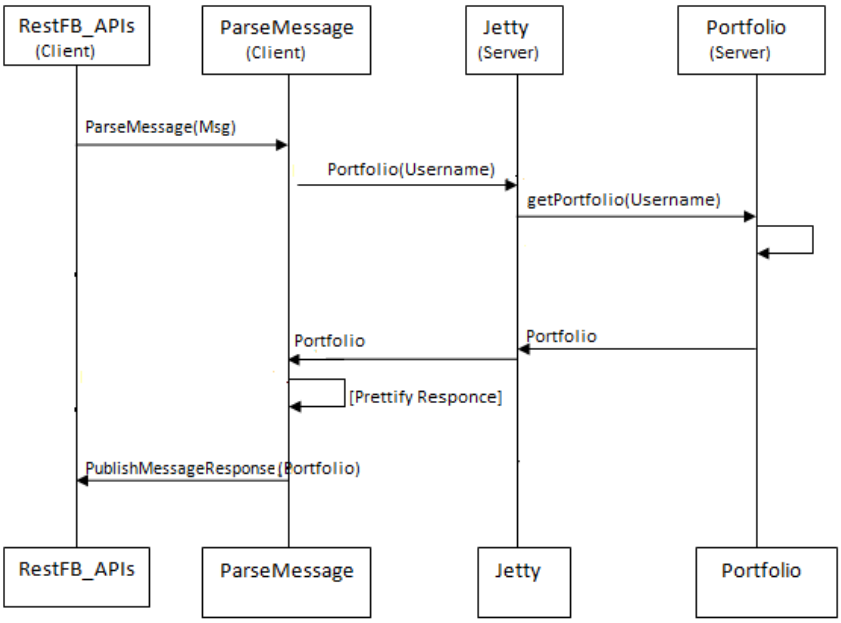


Figure 83:

Once client receives response (portfolio for the username) from server, client prettifies the response make it look better as FaceBook wall post.

7.10.2.5 View Leaderboard: Apart from the leagues created by different users, we have a global league. Players playing via facebook can view the leaders of global league by using operation - view leaderboard.

Here too, we dont have a alternate (failure) flow, as this method will be invoked only once ensureUser confirms that the username exists. Figure 84

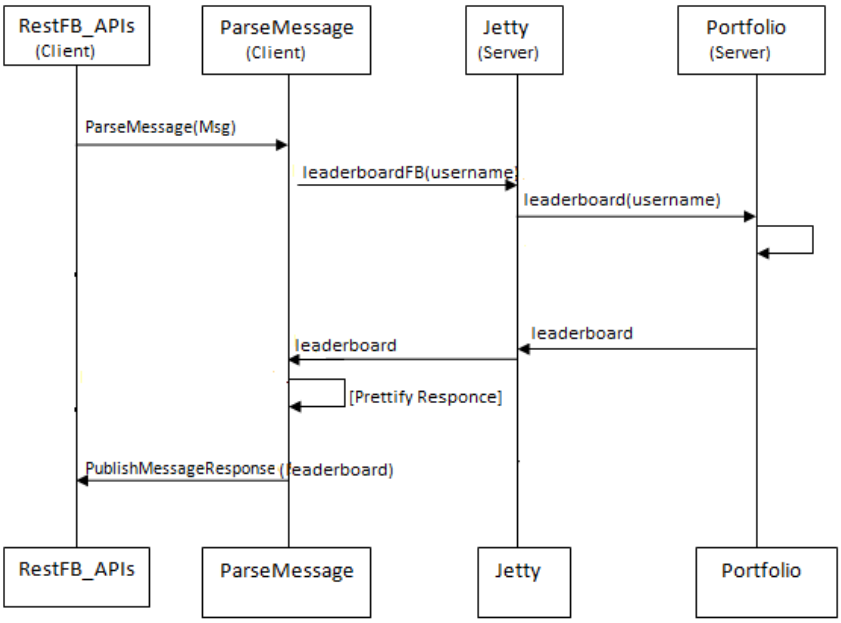


Figure 84:

8 System Architecutre and System Design

8.1 Templating

David Pollak, who developed Lift, believed that it was better not to mix code and HTML[Pollak]_. This is because code is too powerful -- you may initially set out to include only View code in the HTML, but it's too easy to accidentally slip in some functionality that actually belongs in the Model[Pollak]_.

In lift templates, you write HTML code like:

```
<lift:NewsEvent>
    <param:subject/> <param:action/> on <param:when/>
</lift:NewsEvent>
```

and then bind values to it in the Scala code like:

```
class NewsEvent {
    def render(in: NodeSeq) = bind("param", in,
        "subject" -> "joe",
        "action" -> "Bought 100 shares of MSFT",
        "when" -> "Today"
    )
}
```

David Pollak may be right, but we found that the drawbacks of using Lift's templates did not end up being worth the extra help in separating View from Model, and converted most of our template code to raw Scala code. Some reasons for this were:

- We have 4 different views attached to our model -- this means that we already have a really good idea when when we are putting model code into the view, because it gets duplicated among the several Views. Having more than one frontend is a great way to enforce good MVC design.
- Scala has inline XML literals [XML].
- Lift templates cannot do 1 really important thing. Consider the following made-up template code:

```
<lift:Dashboard>
  <lift:Portfolio/>
  <lift:Offers/>
</lift:Dashboard>
```

  This code inserts 3 objects: the containing Dashboard, and inside it a Portfolio and a list off incoming Offers. Now the question is: how do you make the Portfolio code aware of the enclosing Dashboard code?

  In Lift there is no way to do this. Using XML literals this is trivial:

```
class Dashboard {
    def render =
        <div>
            {Portfolio(this).render}
            {Offers(this).render}
        </div>
}
```

- Transforming XML is not type-safe, so errors are not caught until the page is loaded. This wastes a lot of time debugging, and could potentially miss errors forever.

Considering these factors, we wrote our HTML using Scala's XML literals (example website/view/DividendChart.scala ref_44).

8.1.1 Improving Lift Forms

8.1.1.1 Limitations of standard lift forms Lift provides some abstractions for getting data out of a submitted form [Lift2]. It is done in a callback-manner:

```
var ticker: String
var shares: String

bind("param", html,
    "ticker" -> SHtml.text(ticker, { t => ticker = t }),
    "shares" -> SHtml.text(shares, { s => shares = s })
)
```

That is, when the form is submitted, the callbacks are called, and they are passed the data that was submitted.

There are a few reasons we wanted to improve on this system:

- Because callbacks are called individually, you have to use side-effects to build up the complete structure of the data. We like to avoid side-effects when possible [SideEffects].
- Because callbacks are called individually, you have to wait until all have been called to do checks that synthesize multiple values.
- Lift forms deal almost entirely with Strings. This is awkward in a statically typed language. We'd rather worked with typed fields.

To address these concerns we wrote `intform`, which is a wrapper around lift forms (website/intform/).

8.1.1.2 Typed form fields

Every field in `intform` has a type (website/intform/intform.scala ref_727). This is the type of the value that is produced when the form is submitted. So for example a StringField produces a String, a UserField (where you type in a user's name) produces a User, a DollarsField a Dollars, and so on. The Field class has a `process()` method (website/intform/intform.scala ref_997) that produces a value of the correct type.

Once you have introduced typed fields you have to deal with the fact that you might not be able to produce a value of the type you want. Say you have a `DollarsField` and the user types in "one-hua,s.chuetnouhscrasc.hua". You can't convert that to a number. So the `process()` method has to have the type:

```
def process(): Option[A]
```

where `A` is the type that the field produces (see the section on Option Types).

8.1.1.3 Aggregating multiple fields together

Say you have two `IntFields` and a class:

```
case class Point(x: Int, y: Int)
```

and you want to use these to build a `PointField`. We use the same method we used in Serializing objects without using reflection: we treat `Point` as a product type, which be built from a heterogeneous list of fields (website/intform/branches.scala ref_575).

8.1.1.4 Hiding side-effects When an `intform` is submitted a callback is called with the submitted data (example website/view/CommentPage.scala ref_524). At first this seems no different than what Lift forms do. The improvement is that while in Lift forms you have multiple, separate callbacks that are passed the individual fields, in `intform` you get the entire data as a single object, so you do not have to deal with *interaction* between the callbacks. Consider (psuedocode):

```
var x: Int
var y: Int

IntField() { newX =>
    x = newX
}
IntField() { newY =>
    y = newY
}
```

Now say you want to add a check that `x < y`. Where do you add it? If you add it here:

```
var x: Int
var y: Int

IntField() { newX =>
    x = newX
}
IntField() { newY =>
    y = newY
    if (y >= x) throw BadInput
}
```

you are assuming that the `x` callback happens before the `y` callback -- but this is not at all obvious from the code. On the other hand, if your callback takes all data together:

```
PointField() { p =>
    if (p.y >= p.x) throw BadInput
}
```

now you are not relying on the order of any side-effects.

8.2 Serializing objects without using reflection

8.2.1 Why we needed to change

For the first demo, our database backend was written using Squeryl [Squeryl1]. There were some pros and cons to using squeryl, but overall we probably would have kept using it if this were possible.

However, as the model code grew large, we realized we had to reorganize, and one of the ways we reorganized was to split the code into `traits` (See Traits), and mix them together (See Organization of the Model into traits). Unfortunately, Squeryl does not support mapping inner classes, because it does not know how to reconstruct the outer pointer [Squeryl2].

It was more important to us to have a better organized model code than to keep using Squeryl, so we had to change. Initially, we ignored the database backend; our code had no persistence, but this did not make much of a difference in testing and we were able to implement all the important operations with no database. And another benefit of have no database is that we'd keep our code non-specific to the particular database code we used.

But, in the end, we could not actually present a website that lost all its information every time it was restarted. So we wrote "spser" (model/spser.scala).

8.2.2 Product Types

A product type is a type with members [ADTs], e.g. (in Java):

```
class Point {
    public int x;
    public int y;
}
```

is a product type, where the members are `x` and `y`. Another way of saying this is that the `Point` type is a product of `int` and `int`.

So say you wanted to serialize a class:

```
Foo {
    T1 x1;
    T2 x2;
    ...
    TN xN;
}
```

to a database. Well, if you already have a way to turn the types `T1...TN` into database fields, then serializing a `Foo` is just a matter of extracting the members, and converting them to fields (model/spser.scala ref_984). Deserializing is just a matter of extracting the `xk` values, and applying a constructor:

```
(T1, T2, ... TN) => Foo
```

to build a `Foo` object (model/spser.scala ref_704).

8.2.3 Generic representation of products

We use the same representation of products as Mark Harrah's HLists [HList], which in turn is the same representation as Oleg Kiselyov's HList for Haskell [Kiselyov].

A product is either HOne (model/spser.scala ref_220):

```
case class HOne() extends HProd
```

ie, a product of 0 types (the name `HOne` is a reference to the "unit type"[Unit]), or a product of an existing product with one more type added on (model/spser.scala ref_464):

```
case class HTimes[+H,+T<:HProd](head: H, tail: T) extends HProd
```

8.2.4 Looping over products

A common technique when working with Haskell's HLists is to write a typeclass for the loop operation, and then instance declarations for the base- and recursive- cases [Loop]. We use the same technique (as in model/spser.scala ref_231). Where Haskell has typeclasses Scala has implicit parameters [Implicits]. So, for example, to print an `HProd` we can do:

```
trait Display[A] {
    def display(a: A): Unit
}

implicit def displayOne = new Display[HOne] {
    def display(o: HOne) { }
}
implicit def displayTimes[H,T<:HProd:Display] = new Display[HTimes[H,T]] {
    def display(p: HTimes[H,T]) { println(p.head) ; hDisplay(p.tail) }
}
```

8.2.5 Extracting the fields of a product type

Now that we have `HProd`, we have 2 different ways to represent each product type. There's the original, "friendly" way:

```
case class Point(x: Int, y: Int)
```

and the `HProd` "generic" way:

```
HProd[Int, HProd[Int, HOne]]
```

When writing code, we want to use the "friendly" way as much as possible, except in the very backend, where we need to be able to iterate over product fields and so must use the "generic" way. So we must be able to convert between them.

If you look at the class:

```
case class Point(x: Int, y: Int)
```

after it has passed through the first few Scala compiler phases, you will see (among other things; the full output is huge):

```
case class Point extends java.lang.Object with ScalaObject with Product with Serializable {
    <caseaccessor> <paramaccessor> private[this] val x: Int = _;
    <stable> <caseaccessor> <accessor> <paramaccessor> def x: Int = Point.this.x;
    <caseaccessor> <paramaccessor> private[this] val y: Int = _;
    <stable> <caseaccessor> <accessor> <paramaccessor> def y: Int = Point.this.y;
    def this(x: Int, y: Int): Point = {
        Point.super.this();
        ()
    };
    override def productPrefix: java.lang.String = "Point";
    override def productArity: Int = 2;
    override def productElement(x$1: Int): Any = x$1 match {
        case 0 => x
        case 1 => y
        case _ => throw new java.lang.IndexOutOfBoundsException(x$1.toString())
    };
};
```

In other words, the Scala compiler provides some minimal support for extracting elements from product types, in the form of `productElement`. `productElement` is not type-safe, but if we trust the Scala compiler to generate it correctly, we can do some type coercion and create a type-safe extractor (model/spser.scala ref_997).

8.2.6 Re-creating a product type from the fields

How do we go from `HTimes[Int,HTimes[Int,HOne]]` to `Point`? `Point` has a constructor:

```
(Int, Int) => Point
```

which can be used to construct a `Point` given the fields. Unfortunately this is another area where Scala's types are awkward to work with; there is no type-safe way to generalize over function arity. The solution is a set of auto-generated functions for every function arity up to some size (model/spser.scala ref_662).

8.2.7 The advantage to this method of serialization

The biggest advantage to serializing objects using product types is that it works *within* the language, whereas reflection works outside the language. In Scala this is especially relevant because Scala uses Java's reflection API's, which do not know about Scala. The disadvantages to working outside the lanuage are:

- Less type information. JVM type erasure [Erasure] takes away most type information.
- Less type safety. Because reflection operates a run-time and doesn't have static types.
- The chance to conflict with language features, such as how Squeryl cannot pass the outer pointer to a synthesized object. This one was the killer.

8.2.8 Putting this all together

Ideally we would like to add our serialization/deserialization routines to Squeryl. There is no reason this should not be possible. We tried; given more time, we might thave succeeded, but the Squeryl code is fairly set on using reflection to create objects. So we wrote a tiny DSL [DSL] for building SQL queries and attached it to the H2 JDBC library [H2] (model/spser.scala ref_629).

8.3 Applying OO cohesion metrics to our code

A Scala compiler plugin was used to automatically find which methods reference which attributes. This information is used to calculate the cohesion metrics SCOM [SCOM], CC [CC], LSCC [LSCC], and CAMC [CAMC] (the last one uses method signature types and does not look at attributes).

The file `metrics-summary.txt` shows the method-attribute matrix for each class.

A summary of the metrics for all the classes are:

id	name	SCOM	CC	LSCC	CAMC
10749	RunChecks	1.000	1.000	0.000	1.000
245780	$anon	1.000	1.000	0.000	1.000
293016	ResponseAsset	1.000	1.000	1.000	1.000
10548	SharesField	1.000	1.000	1.000	1.000
9558	Security	1.000	1.000	1.000	1.000
10012	TwitterFrontend	0.000	0.000	0.000	0.667
9505	ShortThrowableRenderer	1.000	1.000	0.000	1.000
10459	Direction	1.000	1.000	1.000	1.000
169117	AutoTradeSubmit	1.000	1.000	1.000	1.000
10070	LoginManager	0.000	0.000	0.000	0.333
10162	AggregateField	1.000	1.000	0.000	1.000
9679	UserSchema	0.000	0.000	0.000	1.000
10612	OutgoingOffers	0.000	0.000	0.000	1.000
9623	Dollars	1.000	1.000	1.000	0.146
39042	User	1.000	1.000	1.000	1.000
10544	DollarsField	1.000	1.000	0.000	1.000
9612	Refresh	1.000	1.000	0.000	1.000
10193	Form	1.000	1.000	1.000	1.000
10303	FieldErrorRender	1.000	1.000	1.000	1.000
108737	ResponseOption	1.000	1.000	1.000	1.000
9481	Email_bg	0.000	0.000	0.000	0.667
10046	insertTestData	1.000	1.000	0.000	1.000
9721	GetPortfolio	1.000	1.000	0.000	1.000
9669	StockSchema	1.000	1.000	1.000	1.000
39048	Ownership	1.000	1.000	1.000	1.000
147231	$anon	1.000	1.000	0.000	1.000
10972	MyPage	1.000	1.000	0.000	1.000
9629	Price	1.000	1.000	1.000	0.238
10453	Recipient	1.000	1.000	1.000	1.000
9811	CachedStockDatabase	0.550	0.500	0.350	0.400
39181	DerivativeBuyerSetAside	1.000	1.000	1.000	1.000
9943	TransactionResponse	1.000	1.000	1.000	1.000
246458	$anon	1.000	1.000	1.000	1.000
9796	BatchingStockDatabase	0.056	0.167	0.056	0.250
293013	Response	1.000	1.000	1.000	1.000

145421	__currentLogin	1.000	1.000	1.000	1.000
9993	TextTrader	1.000	1.000	1.000	1.000
10309	AggregateRender	1.000	1.000	1.000	1.000
10305	FormOuter	1.000	1.000	1.000	1.000
10454	SpecificUser	1.000	1.000	1.000	1.000
11044	FormattedDerivative	1.000	1.000	1.000	1.000
157406	$anon	1.000	1.000	1.000	1.000
10871	NoOrder	1.000	1.000	1.000	1.000
9874	StockDatabase	1.000	1.000	1.000	1.000
9921	Buy	1.000	1.000	1.000	1.000
246941	$anon	1.000	1.000	0.000	1.000
10392	ClearDatabase	1.000	1.000	0.000	1.000
9701	BuyServlet	1.000	1.000	0.000	1.000
39167	UserWithComments	0.000	0.000	0.000	1.000
39203	AutoTrade	1.000	1.000	1.000	1.000
10246	StringField	1.000	1.000	0.000	1.000
10163	CaseField	0.667	0.500	0.333	1.000
9604	Link	0.833	0.667	0.500	0.333
10253	ConstField	0.000	0.000	0.000	1.000
265398	HaveCommand	1.000	1.000	1.000	1.000
9889	YahooCSVStockDatabase	0.000	0.000	0.000	0.500
10492	dividendChart	1.000	1.000	0.000	1.000
10033	DBSetup	1.000	1.000	0.000	1.000
10250	IntField	1.000	1.000	1.000	1.000
39168	NewsEventWithComments	0.000	0.000	0.000	0.333
9930	StockAsset	1.000	1.000	1.000	1.000
11020	BigDecimalFormatted	1.000	1.000	1.000	1.000
10211	BadInput	1.000	1.000	1.000	1.000
39137	Declined	1.000	1.000	1.000	1.000
9763	StockUpdates	1.000	1.000	1.000	1.000
39143	Bid	1.000	1.000	1.000	1.000
9574	ComparableSecurity	1.000	1.000	1.000	1.000
10343	AuctionPage	0.000	0.000	0.000	1.000
9931	StockDollars	1.000	1.000	1.000	1.000
9598	Table	0.000	0.000	0.000	0.500
10196	Form	1.000	1.000	0.000	1.000
39140	Auctioned	1.000	1.000	1.000	1.000
142554	accessToken	1.000	1.000	1.000	1.000
39062	OldUser	1.000	1.000	1.000	1.000
10306	SubmitRender	1.000	1.000	1.000	1.000
10300	Refreshable	1.000	1.000	0.000	1.000
9625	Dollars	0.000	0.000	0.000	0.500
10210	FormSubmit	0.000	0.000	0.000	0.583
10410	commentPage	1.000	1.000	0.000	1.000
39110	AuctionBid	1.000	1.000	1.000	1.000
10244	StringField	1.000	1.000	1.000	1.000
10581	News	0.000	0.000	0.000	0.556
9601	KL	1.000	1.000	1.000	0.500
39054	League	1.000	1.000	1.000	1.000
9735	SellServlet	1.000	1.000	1.000	1.000
39194	NewsEventWithVotes	0.000	0.000	0.000	1.000
39146	Won	1.000	1.000	1.000	1.000
11024	BigDecimalOptionFormatted	1.000	1.000	1.000	1.000
157401	NeedRenderable	1.000	1.000	1.000	1.000
9565	SecDerivative	1.000	1.000	1.000	0.500
10971	UserPage	1.000	1.000	0.000	1.000
9699	BuyServlet	1.000	1.000	1.000	1.000
9824	DividendDatabase	1.000	1.000	1.000	1.000
145792	savedPortfolio	1.000	1.000	1.000	1.000
10304	InnerFieldRender	1.000	1.000	1.000	1.000
93022	$anon	1.000	1.000	0.000	1.000
10282	package	0.000	0.000	0.000	0.042
10458	OpenAuction	1.000	1.000	0.000	1.000
9608	Links	0.000	0.000	0.000	0.500
9649	RankingSchema	0.000	0.000	0.000	1.000
10644	PortfolioInvites	0.000	0.000	0.000	0.333
10166	CaseField	1.000	1.000	0.000	1.000
10915	tChart	1.000	1.000	0.000	1.000
10795	SearchPipeline	1.000	1.000	1.000	1.000
265394	Idle	1.000	1.000	1.000	1.000
39051	PortfolioInvite	1.000	1.000	1.000	1.000
10893	SwitchPortfolio	1.000	1.000	0.000	1.000

10807	SelectField	1.000	1.000	0.000	1.000
10808	SelectRender	1.000	1.000	1.000	1.000
9856	QueryService	1.000	1.000	1.000	1.000
9971	package	1.000	1.000	1.000	1.000
9991	Backend	1.000	1.000	1.000	1.000
9937	Response	1.000	1.000	1.000	1.000
10804	SelectField	0.500	0.333	0.333	1.000
39099	DerivativeAssetOps	0.000	0.000	0.000	1.000
10302	FieldRender	1.000	1.000	1.000	1.000
39089	DerivativeAsset	1.000	1.000	1.000	1.000
10264	DateField	1.000	1.000	0.000	1.000
39193	PortfolioWithVotes	0.000	0.000	0.000	0.367
39157	NewsEvent	1.000	1.000	0.000	1.000
10270	TextAreaField	1.000	0.500	0.500	1.000
9849	HttpQueryService	1.000	1.000	1.000	1.000
127504	$anon	1.000	1.000	0.000	1.000
10197	Field	1.000	1.000	1.000	1.000
10140	package	1.000	1.000	0.000	1.000
9673	DividendSource	1.000	1.000	1.000	1.000
9868	QuoteInfo	1.000	1.000	1.000	1.000
10942	UserField	1.000	1.000	0.000	1.000
10631	PortfolioField	1.000	1.000	0.000	1.000
9600	DBMagic	1.000	1.000	1.000	1.000
9488	email	1.000	1.000	0.000	1.000
10660	PortfolioLink	1.000	1.000	0.000	1.000
10227	ChildError	1.000	1.000	0.000	1.000
39025	NoSuchEvent	1.000	1.000	1.000	1.000
9838	FailoverStockDatabase	0.000	0.000	0.000	0.750
9934	StockShares	1.000	1.000	1.000	1.000
9683	VotingSchema	1.000	1.000	1.000	1.000
59900	$anon	1.000	1.000	0.000	1.000
9562	SecStock	1.000	1.000	1.000	0.500
9497	package	1.000	1.000	0.000	1.000
39206	PortfolioWithAutoTrades	0.000	0.000	0.000	1.000
10308	TextRender	1.000	1.000	1.000	1.000
10208	FormSubmit	1.000	1.000	1.000	1.000
39065	PortfolioOps	0.000	0.000	0.000	0.600
10563	NewPortfolio	0.000	0.000	0.000	1.000
9490	email	1.000	1.000	1.000	1.000
39029	NameInUse	1.000	1.000	1.000	1.000
9802	CacheMap	0.167	0.333	0.167	0.417
10307	ErrorRender	1.000	1.000	1.000	1.000
9571	CondGreater	1.000	1.000	1.000	1.000
9538	AuctionSchema	0.000	0.000	0.000	0.333
10687	PrintSchema	1.000	1.000	0.000	1.000
10256	ConstField	1.000	1.000	0.000	1.000
39166	EventComment	1.000	1.000	0.000	1.000
145413	NotLoggedIn	1.000	1.000	0.000	1.000
9789	TestServlet	0.000	0.000	0.000	0.500
10269	DateTimeField	1.000	1.000	0.000	1.000
245766	$anon	1.000	1.000	0.000	1.000
39023	NoSuchDerivativeLiability	1.000	1.000	1.000	1.000
39021	NoSuchDerivativeAsset	1.000	1.000	1.000	1.000
10958	UserLink	1.000	1.000	0.000	1.000
59950	$anon	0.000	0.000	0.000	0.625
11272	NodeSeqPlus	1.000	1.000	1.000	1.000
38939	NotFound	1.000	1.000	1.000	1.000
10267	DateTimeField	1.000	1.000	1.000	1.000
10361	AuctionThumbnail	0.000	0.000	0.000	1.000
93057	$anon	1.000	1.000	0.000	1.000
10978	theirPortfolio	1.000	1.000	0.000	1.000
9483	Email_bg	1.000	1.000	1.000	1.000
10199	BasicErrors	1.000	1.000	1.000	0.333
39092	DerivativeLiability	1.000	1.000	1.000	1.000
246450	$anon	1.000	1.000	1.000	1.000
10207	Submit	0.000	0.000	0.000	0.583
265395	HaveQuote	1.000	1.000	1.000	1.000
39115	PortfolioWithAuctions	0.000	0.000	0.000	0.500
9671	StockPriceSource	1.000	1.000	1.000	1.000
10579	News	0.000	0.000	0.000	1.000
9908	Message	1.000	1.000	1.000	1.000
9554	DerivativeSchema	0.000	0.000	0.000	1.000

10673	PortfolioPage	1.000	1.000	0.000	1.000
39010	NotExecutable	1.000	1.000	1.000	1.000
9644	operations	0.000	0.000	0.000	0.667
10131	TwitterLogin	0.000	0.000	0.000	0.286
39047	Portfolio	0.000	0.000	0.000	0.333
10464	StockInDerivative	1.000	1.000	1.000	1.000
39001	DontOwnStock	1.000	1.000	1.000	1.000
9875	NoSuchStockException	1.000	1.000	1.000	1.000
10011	twit	1.000	1.000	0.000	1.000
9656	schema	1.000	1.000	1.000	1.000
9940	Status	1.000	1.000	1.000	1.000
39070	StockAsset	1.000	1.000	1.000	1.000
10629	PortfolioField	1.000	1.000	1.000	1.000
10847	stockChart	1.000	1.000	0.000	1.000
9862	Stock	1.000	1.000	1.000	1.000
10272	TextAreaField	1.000	1.000	0.000	1.000
9912	Request	1.000	1.000	1.000	1.000
10312	ListRender	0.000	0.000	0.000	0.333
39008	NoSuchOffer	1.000	1.000	1.000	1.000
9626	Shares	1.000	1.000	1.000	0.208
10200	Processable	1.000	1.000	1.000	1.000
9559	SecDollar	1.000	1.000	1.000	0.500
9901	YahooStockDatabase	0.000	0.000	0.000	0.333
10622	portfolio	1.000	1.000	0.000	1.000
10974	myPage	1.000	1.000	0.000	1.000
11001	BoxOps	1.000	1.000	1.000	1.000
10217	SubmitResult	1.000	1.000	1.000	1.000
9631	Price	0.000	0.000	0.000	0.500
39057	UserOps	0.000	0.000	0.000	0.273
9826	CachedDividendDatabase	1.000	1.000	1.000	1.000
10545	PriceField	1.000	1.000	1.000	1.000
293011	Response	1.000	1.000	1.000	1.000
10223	Error	1.000	1.000	1.000	1.000
10976	theirPage	1.000	1.000	0.000	1.000
10301	Page	1.000	1.000	0.000	1.000
39114	AuctionOfferOps	0.000	0.000	0.000	1.000
39013	BidTooSmall	1.000	1.000	1.000	1.000
9622	package	0.000	0.000	0.000	0.500
9542	AutoTradeSchema	1.000	1.000	1.000	1.000
9825	YahooDividendDatabase	1.000	1.000	1.000	1.000
9527	DummySchema	0.000	0.000	0.000	1.000
52247	$anon	0.000	0.000	0.000	0.333
9628	Shares	0.000	0.000	0.000	0.500
10259	BooleanField	1.000	1.000	0.000	1.000
142552	requestToken	1.000	1.000	1.000	1.000
10823	SellThisStock	1.000	1.000	1.000	1.000
10219	OK	1.000	1.000	1.000	1.000
38961	Insert	1.000	1.000	1.000	1.000
10174	DependentListField	1.000	1.000	0.000	1.000
9924	Sell	1.000	1.000	1.000	1.000
10298	Refreshable	1.000	1.000	1.000	1.000
39095	DerivativeOffer	1.000	1.000	1.000	1.000
11275	NodePlus	1.000	1.000	1.000	1.000
10316	CheckBoxRender	1.000	1.000	1.000	1.000
38960	EditOp	1.000	1.000	1.000	1.000
9787	TestServlet	1.000	1.000	1.000	1.000
116811	LazyQuote	1.000	1.000	1.000	1.000
136675	$anon	1.000	1.000	0.000	1.000
39190	DerivativeSellerVote	1.000	1.000	1.000	1.000
108743	ResponseError	1.000	1.000	1.000	1.000
10509	EventPage	0.000	0.000	0.000	0.250
9675	Stocks	0.000	0.000	0.000	1.000
9995	TextTrader	0.000	0.000	0.000	0.750
10734	RefreshHack	1.000	1.000	0.000	1.000
10524	LoginStatus	0.000	0.000	0.000	1.000
10328	package	0.000	0.000	0.000	0.500
10319	TextAreaRender	1.000	1.000	1.000	1.000
9911	Reply	1.000	1.000	1.000	1.000
10079	Logout	0.000	0.000	0.000	0.500
246521	$anon	1.000	1.000	0.000	1.000
246466	$anon	1.000	1.000	1.000	1.000
116813	LazyQuote	1.000	1.000	0.000	1.000

39155	NewsEvent	1.000	1.000	1.000	1.000
9966	ConsoleFrontend	1.000	1.000	1.000	1.000
142492	redirectBackTo	1.000	1.000	1.000	1.000
10249	NumberField	1.000	1.000	0.000	1.000
245750	$anon	1.000	1.000	1.000	1.000
266571	$anon	1.000	1.000	1.000	1.000
108740	ResponseValid	1.000	1.000	1.000	1.000
39080	PortfolioWithStocks	0.000	0.000	0.000	0.412
9878	DatabaseException	1.000	1.000	1.000	1.000
10297	Renderable	1.000	1.000	1.000	1.000
38953	Transaction	1.000	1.000	0.000	1.000
11278	MergeAttr	1.000	1.000	1.000	1.000
10170	ListField	1.000	1.000	0.000	1.000
293014	ResponseAsset	1.000	1.000	1.000	1.000
9532	H2Schema	0.000	0.000	0.000	1.000
10461	ToBuyer	0.000	0.000	0.000	0.250
9737	SellServlet	1.000	1.000	0.000	1.000
38965	Update	1.000	1.000	1.000	1.000
93105	$anon	1.000	1.000	0.000	1.000
9555	Derivative	1.000	0.250	0.250	0.500
10880	StockOrderer	0.556	0.403	0.347	0.200
10712	quoteReport	1.000	1.000	0.000	1.000
39100	PortfolioWithDerivatives	0.000	0.000	0.000	0.278
11015	DateTimeFormatted	1.000	1.000	1.000	1.000
11006	BadUser	1.000	1.000	1.000	1.000
9639	NewsSchema	0.000	0.000	0.000	0.333
10059	Checker	0.000	0.000	0.000	1.000
39164	EventComment	1.000	1.000	1.000	1.000
9632	Scale	1.000	1.000	1.000	0.194
10547	PriceField	1.000	1.000	0.000	1.000
9474	EmailActor	0.000	0.000	0.000	1.000
10101	periodically	1.000	1.000	0.000	1.000
39121	Action	1.000	1.000	0.000	1.000
10470	DerivativeBuilder	0.055	0.109	0.055	0.136
9867	Quote	0.000	0.000	0.000	0.500
39207	AutoTradeOps	0.000	0.000	0.000	0.667
39152	Exercised	1.000	1.000	1.000	1.000
9992	Frontend	1.000	1.000	1.000	1.000
9960	WithUser	0.667	0.667	0.667	0.333
39134	Accepted	1.000	1.000	1.000	1.000
9959	PitFailBackend	0.000	0.000	0.000	0.500
39184	DerivativeSellerSetAside	1.000	1.000	1.000	1.000
38998	NotEnoughCash	1.000	1.000	1.000	1.000
39131	Offered	1.000	1.000	1.000	1.000
11040	CompSecFormatted	1.000	1.000	1.000	1.000
11036	FormattedCondition	1.000	1.000	1.000	1.000
39056	League	0.000	0.000	0.000	0.375
60540	$anon	1.000	1.000	1.000	1.000
39128	Sold	1.000	1.000	1.000	1.000
9765	StockUpdates	1.000	1.000	0.000	1.000
9865	Quote	1.000	1.000	1.000	1.000
9851	HttpQueryService	0.000	0.000	0.000	0.750
10214	BadFieldInput	1.000	1.000	1.000	1.000
10028	Boot	1.000	1.000	0.000	1.000
265392	Status	1.000	1.000	1.000	1.000
10317	DateRender	1.000	1.000	0.000	1.000
9927	SellAll	1.000	1.000	1.000	1.000
38997	NegativeVolume	1.000	1.000	1.000	1.000
39045	Portfolio	1.000	1.000	1.000	1.000
10550	SharesField	1.000	1.000	0.000	1.000
9610	RefreshHub	0.000	0.000	0.000	1.000
39187	DerivativeBuyerVote	1.000	1.000	1.000	1.000
10429	Dashboard	0.000	0.000	0.000	1.000
9586	DividendSchema	0.000	0.000	0.000	1.000
39109	AuctionOffer	1.000	1.000	0.000	1.000
9915	Action	1.000	1.000	1.000	1.000
9578	CompSecDollar	1.000	1.000	1.000	1.000
39079	StockAssetOps	0.000	0.000	0.000	1.000
39076	GroupedStockAsset	1.000	1.000	1.000	1.000
246506	$anon	0.500	0.500	0.500	1.000
10463	ToSeller	0.000	0.000	0.000	0.250
10242	TextField	1.000	1.000	1.000	1.000

10378	AutoTrades	0.500	0.500	0.500	0.250
9942	OK	1.000	1.000	0.000	1.000
11056	ColonEq	1.000	1.000	1.000	1.000
9510	package	0.000	0.000	0.000	0.100
9521	RWTwitter	1.000	1.000	1.000	1.000
256492	Stuff	1.000	1.000	1.000	1.000
39019	NoSuchPortfolio	1.000	1.000	1.000	1.000
39059	NewUser	1.000	1.000	1.000	1.000
10247	NumberField	1.000	1.000	1.000	1.000
9503	ShortThrowableRenderer	1.000	1.000	1.000	1.000
10203	UnitProcessable	1.000	1.000	0.000	1.000
10877	AddToDerivative	1.000	1.000	1.000	1.000
10204	Submit	1.000	1.000	1.000	1.000
39107	AuctionOffer	1.000	1.000	1.000	1.000
9719	GetPortfolio	1.000	1.000	1.000	1.000
9821	Dividend	1.000	1.000	1.000	1.000
167813	Bid	1.000	1.000	1.000	1.000
10113	PortfolioSwitcher	0.000	0.000	0.000	0.333
9519	Twitter	1.000	1.000	1.000	1.000
9607	Link	1.000	1.000	0.000	1.000
9949	Failed	1.000	1.000	1.000	1.000
39012	NoSuchAuction	1.000	1.000	1.000	1.000
11028	FormattedSecurities	1.000	1.000	1.000	1.000
256512	$anon	1.000	1.000	1.000	1.000
10310	CaseRender	1.000	1.000	1.000	1.000
10940	UserField	1.000	1.000	1.000	1.000
59575	$anon	1.000	1.000	0.000	1.000
39004	NotEnoughShares	1.000	1.000	1.000	1.000
39125	Bought	1.000	1.000	1.000	1.000
10167	ListField	0.000	0.000	0.000	0.250
10774	SearchBar	0.087	0.209	0.048	0.136
149581	$anon	1.000	1.000	0.000	1.000
10092	OpenIDLogin	0.000	0.000	0.000	0.250
39058	IsNewUser	1.000	1.000	1.000	1.000
10311	CaseChoices	1.000	1.000	1.000	1.000
39027	NoSuchComment	1.000	1.000	1.000	1.000
10262	DateField	1.000	1.000	1.000	1.000
142543	Auth	1.000	1.000	1.000	1.000
142556	serviceInProgress	1.000	1.000	1.000	1.000
39149	Closed	1.000	1.000	1.000	1.000
39212	DividendPayment	1.000	1.000	1.000	1.000
10997	voteControls	0.000	0.000	0.000	0.467
151983	$anon	1.000	1.000	0.000	1.000
9657	SchemaErrors	0.000	0.000	0.000	1.000
10252	IntField	1.000	1.000	0.000	1.000
9634	Scale	0.000	0.000	0.000	0.500
38969	Delete	1.000	1.000	1.000	1.000
9575	CompSecStock	1.000	1.000	1.000	1.000
10870	StockOrder	1.000	1.000	1.000	1.000
157348	$anon	1.000	1.000	0.000	1.000
39073	StockPurchase	1.000	1.000	1.000	1.000
39017	NoSuchUser	1.000	1.000	1.000	1.000
39094	DerivativeLiability	1.000	1.000	0.000	1.000
10874	BuyShares	1.000	1.000	1.000	1.000
39044	User	0.000	0.000	0.000	1.000
10825	SellThisStock	1.000	1.000	0.000	1.000
10542	DollarsField	1.000	1.000	1.000	1.000
10158	AggregateField	0.667	0.500	0.333	1.000
9570	CondAlways	0.000	0.000	0.000	1.000
9659	NotFound	1.000	1.000	0.000	1.000
142550	returnTo	1.000	1.000	1.000	1.000
9547	CommentSchema	1.000	1.000	1.000	1.000
38950	Transaction	1.000	0.750	0.750	0.250
10721	Refreshable	0.000	0.000	0.000	1.000
108978	Entry	1.000	1.000	1.000	1.000
10257	BooleanField	1.000	0.500	0.500	1.000
157238	$anon	1.000	1.000	1.000	1.000
10467	DerivativeOrder	1.000	1.000	1.000	1.000
11032	FormattedSecurity	1.000	1.000	1.000	1.000
10318	DependentListRender	1.000	1.000	1.000	1.000
9980	parser	0.000	0.000	0.000	1.000
9987	ConsoleTest	1.000	1.000	0.000	1.000

9918	GetInfo	1.000	1.000	1.000	1.000
9917	Portfolio	1.000	1.000	0.000	1.000
10171	DependentListField	0.500	1.000	0.500	1.000
9946	StringResponse	1.000	1.000	1.000	1.000
10313	ItemRender	1.000	1.000	1.000	1.000
9568	Condition	1.000	1.000	1.000	1.000
9609	Transactions	0.000	0.000	0.000	0.125
10933	TestForm	1.000	1.000	0.000	1.000
9603	KL	1.000	1.000	0.000	1.000
10599	Offers	0.000	0.000	0.000	0.500

8.3.1 Decisions that were made about how to calculate the metrics

Fernandez and Peña do not say explicitly whether in the following situtaion method foo() references attribute x [SCOM]:

```
class A {
    var x: Int = _
    def foo() = bar()
    def bar() = x
}
```

Because Scala wraps almost *all* attributes in accessor methods, even internally, these metrics would make little sense unless foo() is considered to reference bar().

Another question is, What is an atttribute? The following decisions were made:

Member	**attribute?**	**Why?**
var	yes	Exact analog of a Java instance variable
concrete val	no	Cannot change
abstract val	yes	Value depends on where it is made concrete
def	no	This is a method

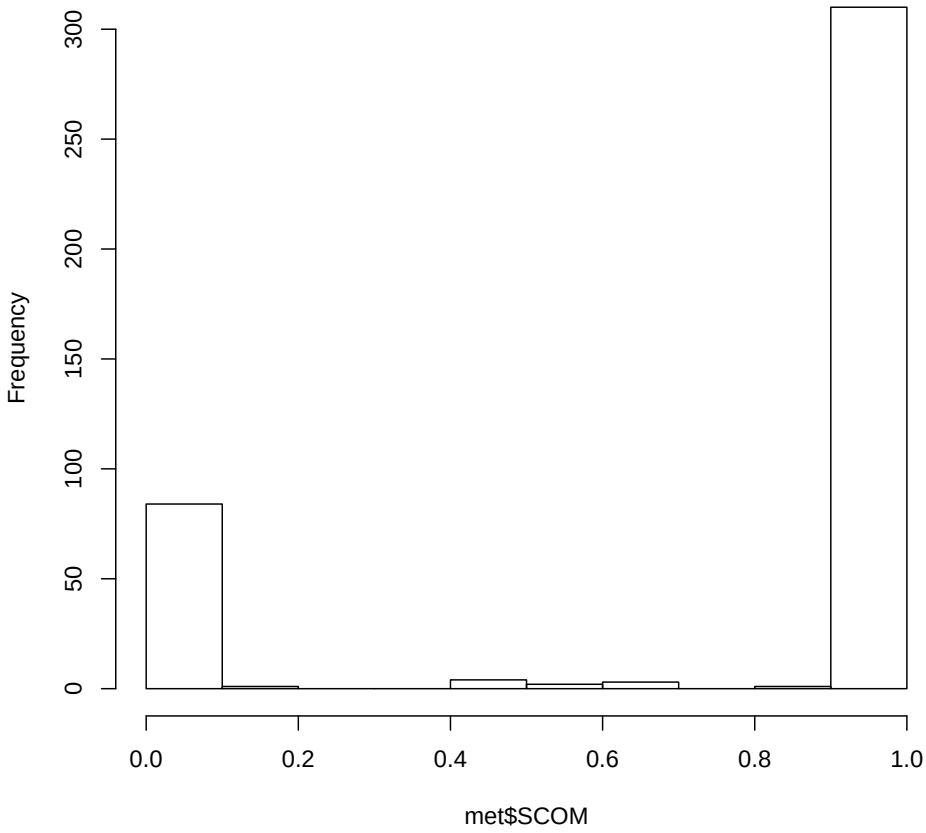
8.3.2 Problems with OO cohesion metrics for Scala code

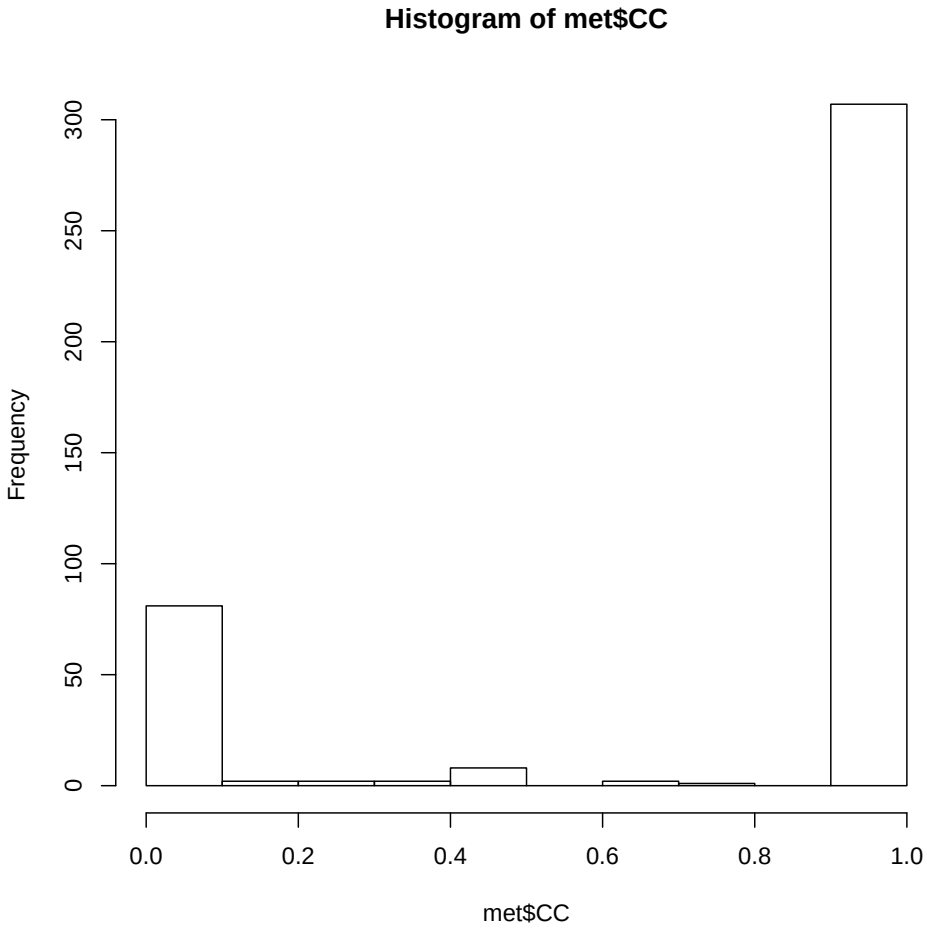
The biggest problem with these metrics is that in Scala it is common (and good practice) to have classes with no methods at all; that act merely as a container for multiple fields [ADTs].

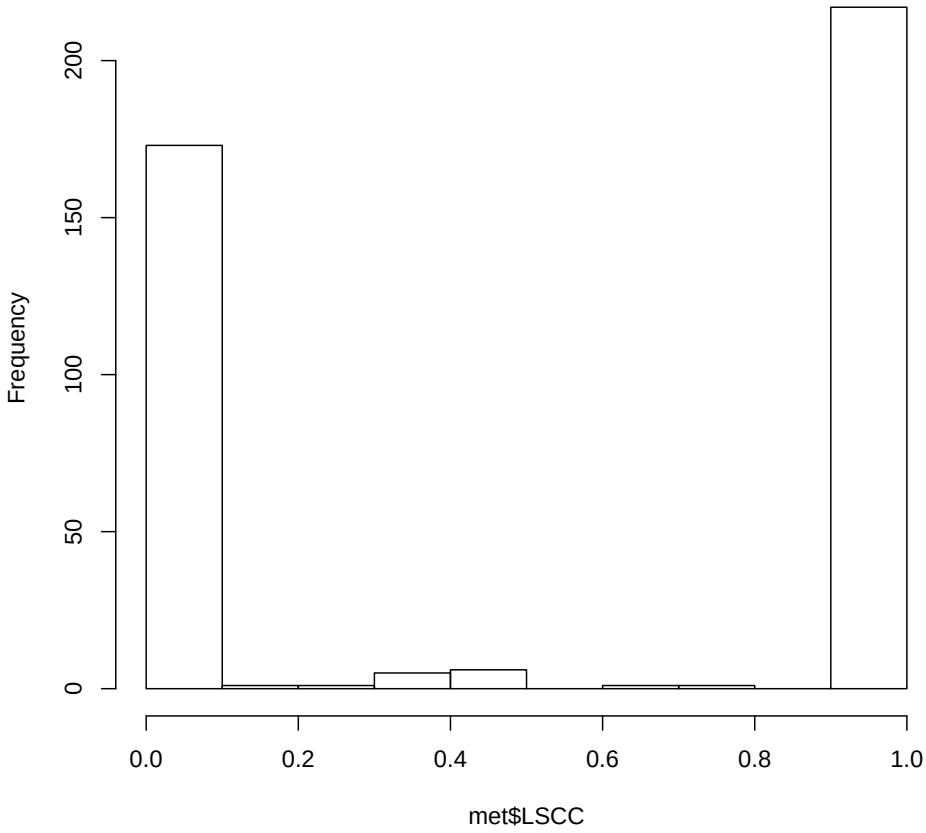
Another problem is that it is common (and also good practice) to abstract methods out into `traits` which contain no fields. Hence a large number of Scala classes contain only fields and no methods, or only methods and no fields.

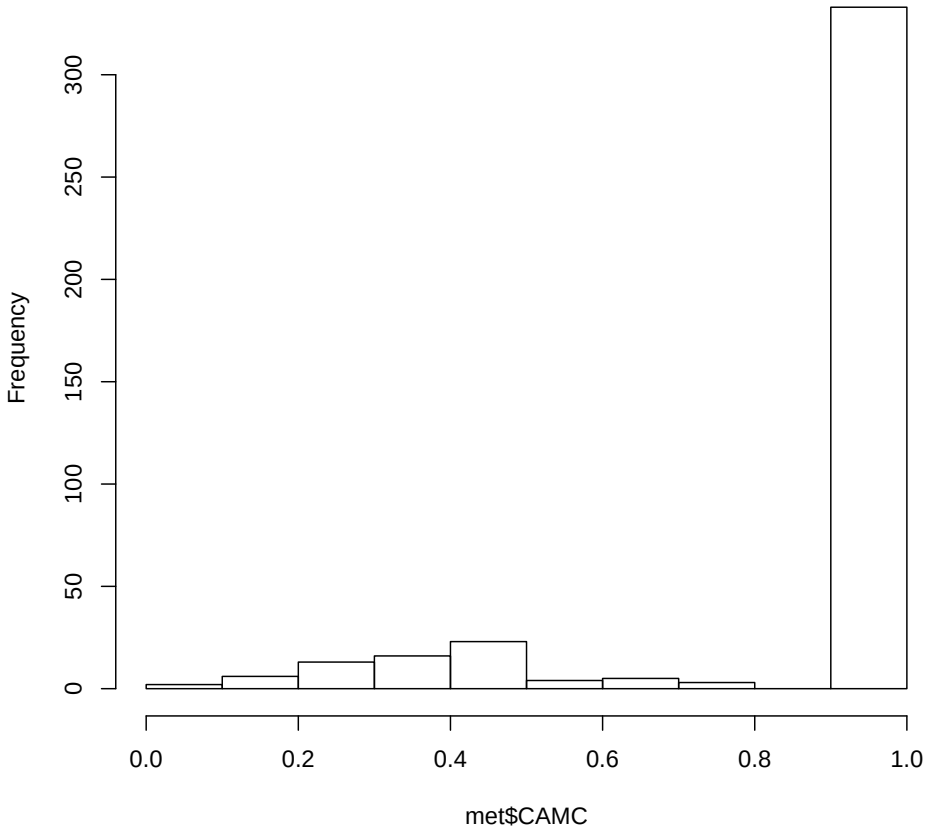
CAMC suffers from the fact that Scala types tend to be more complicated than types in other OO languages, so it is harder for two types to be equal.

These three facts result in the following histograms:


Histogram of met$SCOM
Frequency
0
50
100
150
200
250
300
0.0
0.2
0.4
0.6
0.8
1.0
met$SCOM

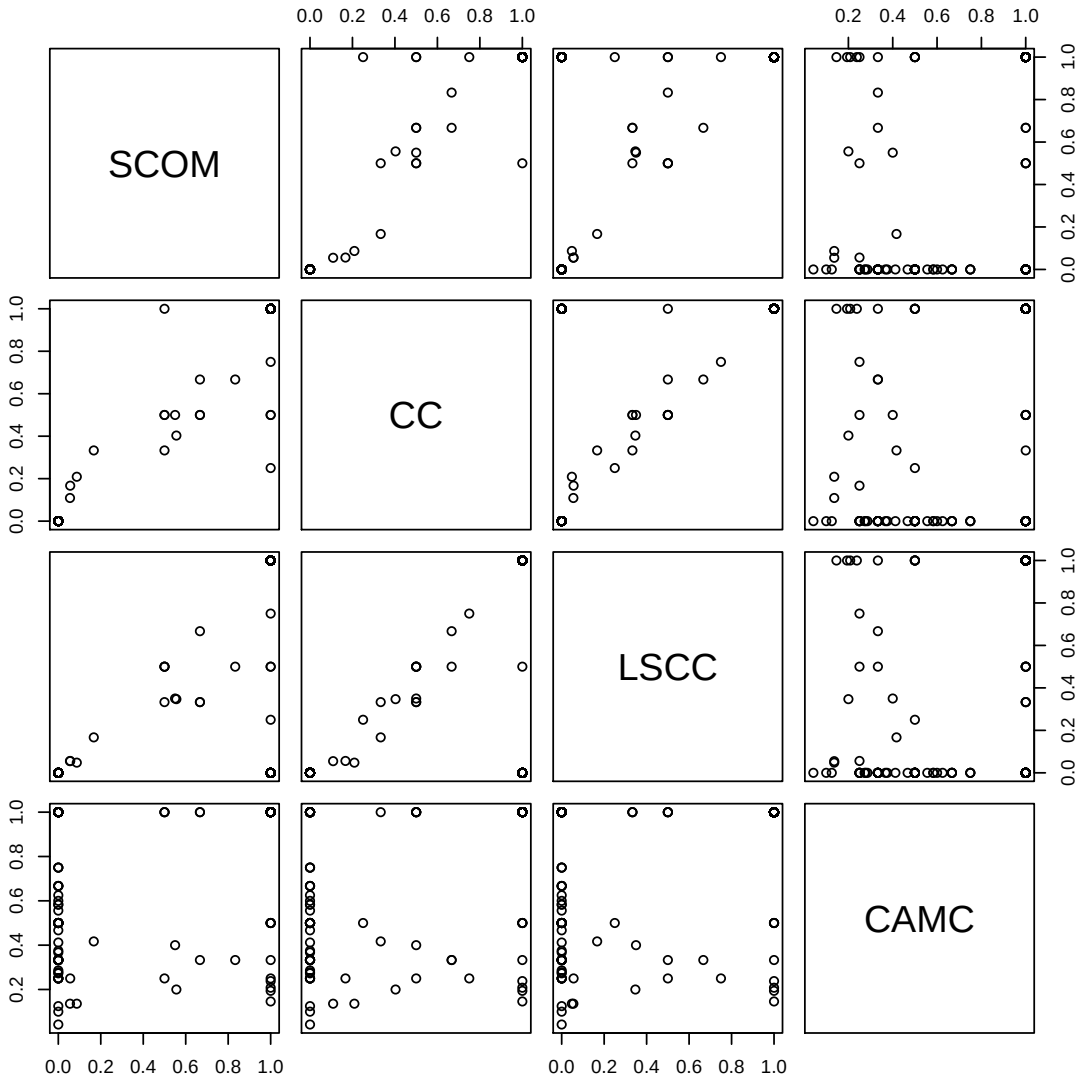

Histogram of met$CC
Frequency
0
50
100
150
200
250
300
0.0
0.2
0.4
0.6
0.8
1.0
met$CC


Histogram of met$LSCC
Frequency
0
50
100
150
200
0.0
0.2
0.4
0.6
0.8
1.0
met$LSCC



Most classes either fall to 0 or 1, with only a few in the middle. Further more it is not clear that those that fall to 0 (classes with no methods) are really bad -- they would be bad Java classes but they are good Scala classes.

It is good to see the above histograms before looking at the below trellis graphic, because otherwise the trellis graphic makes the metrics look more appropriate than they really are:



Most of the metrics corellate well with each other, except CAMC, which is all over the place.

8.4 Evaluating the cohesion of functional code

We do not know how to give a metric nor or we sure that a numerical metric is the right approach, but we have some ideas on what makes functional code more or less cohesive.

8.4.1 Why OO metrics do not work well for functional code

OO metrics do not work well with functional code because they do not give good answers for the following pattern:

```
case class Point(x: Double, y: Double)
```

```
def dist(p1: Point, p2: Point): Double = ...
```

That is, defining classes that only hold fields, and then defining methods outside of those classes. This is common in functional programming [Data]. Scala does incorporate OO, but this pattern is still too common (in our experience) to make OO metrics useful, which would consider the fields of `Point` to be unconnected [SCOM].

Another pattern that OO metrics have trouble with is:

```
def diff(f: Double=>Double) = { (x: Double) =>
    (f(x + 1e-5) - f(x)) / 1e-5
}
```

There is a context being created, in which the variable `f` is visible, almost as if you defined a class like:

```
class Diff {
    val f: Double=>Double

    def apply(x: Int) = (f(x + 1e-5) - f(x)) / 1e-5
}
```

(For many good examples on this pattern see [SICP1]).

OO metrics act as if the only creator of context is a class [SCOM], but in functional programming this is often not true, as above.

8.4.2 Thinking in terms of statements and proofs

The Curry-Howard Isomorphism relates types and data in a programming language to logical statements [CurryHoward]:

- A *type* corresponds to a statement.
- A *value* corresponds to a proof of the statement of its type.

A function with input type `A` and output type `B` has a type written `A => B` which is taken to mean "A implies B" [CurryHoward]. So the actual function (ie the value):

```
def foo(a: A): B = ...
```

can be thought of as the proof that `A` implies `B`. So `A` is the hopethesis in the proof, and `B` is the conclusion.

This gives us a way to describe (not quite define, unfortunately) an idea which we will use to describe cohesion. Say we have a function defined like:

```
def foo(x: String, y: Int): Int = y * 2
```

This has type:

```
(String, Int) => Int
```

(the product type `(String, Int)` is analogous to "and" [CurryHoward]).

So our hypothesis is "`String` and `Int`", but in the proof we only use `Int`. So we have made an unnecessary assumption. And you can see just by looking that the `x` argument to `foo` is superfluous.

So, this gives us a way to say whether a function has superfluous arguments. But that was already obvious, because you don't usually write functions with unneeded arguments anyway: you have to make a conscious effort to put in the `x` argument, and if it's really unnecessary you wouldn't add it in the first place.

But there is another place where hypotheses come from: enclosing scopes. Consider the curried form [Currying] of `foo`:

```
def foo(x: String) = {

    def bar(y: Int) = y * 2

    bar _
}
```

Inside `foo` we define a function `bar`, and then return that function.
What is not so obvious, and easy to miss in actual code, is that `bar` could refer to `x` if it so desires:

```
def foo(x: String) = {

    def bar(y: Int) = x

    bar _
}
```

but it doesn't. This means the assumptions ("`String`") introduced by the enclosing context are not needed in the proof of bar.
There are other ways that unneeded hypotheses can sneak in. Consider:

```
case class Point(x: Int, y: Int)

def projectX(p: Point): Int = p.x
```

`Point` is a product type (See Product Types), but `projectX` uses only one field of the product. A more cohesive design would be:

```
trait HasX { val x: Int }
case class Point(x: Int, y: Int) extends HasX

def projectX(h: HasX) = h.x
```

Or, directly using Scala's structural types [Struct]:

```
case class Point(x: Int, y: Int)

def projectX(p: {val x: Int}) = p.x
```

8.4.3 Evaluating cohesion

Say we want to evaluate the cohesion of the previous code:

```
def foo(x: String) = {

    def bar(y: Int) = y * 2

    bar _
}
```

We would say that the scope created by `foo` has extra things in it that do not belong there, because they make no use of that scope in their code. A more cohesive version is:

```
def bar(y: Int) = y * 2
```

In this sample of code from PitFail (model/auctions.scala ref_823):

```
trait PortfolioWithAuctions {
    self: Portfolio =>

    def auctionOffers: Seq[AuctionOffer] = schema.auctionOffers where
        ('offerer ~=~ this) toList

    def userCastBid(auction: AuctionOffer, price: Dollars) = editDB {
        if (price <= auction.goingPrice)
            throw BidTooSmall(auction.goingPrice)

        (
              AuctionBid(offer=auction, by=this, price=price).insert
            & Bid(this, auction, price).report
        )
    }
}
```

we see that `userCastBid` has `auctionOffers` in scope, but never uses it. We could break it up like:

```
trait PortfolioWithAuctions
    extends PortfolioWithAuctionOffers
    with    PortfolioWithBids

trait PortfolioWithAuctionOffers {
    self: Portfolio =>

    def auctionOffers: Seq[AuctionOffer] = schema.auctionOffers where
        ('offerer ~=~ this) toList

}

trait PortfolioWithBids {
    self: Portfolio =>

    def userCastBid(auction: AuctionOffer, price: Dollars) = editDB {
        if (price <= auction.goingPrice)
            throw BidTooSmall(auction.goingPrice)

        (
              AuctionBid(offer=auction, by=this, price=price).insert
            & Bid(this, auction, price).report
        )
    }
}
```

so `userCastBid` is now more restrictively typed.

8.4.4 Can you assume too little?

We talked about assuming too much, but is it possible to assume too little?

It is possible, if there are holes in your code [CurryHoward] such as exceptions, infinite loops [Iry1] or incomplete case expressions [CurryHoward]. These are regarded in functional programming as a Bad Thing [Iry2] and people already avoid them.

9 Customer Statement of Requirements

Investors today are seeking more effective financial tools that not only motivates them to invest in the stock market and improve their decision making skills but also an application that is interesting enough to keep using. Our goal is to build a system that is less focused on simulation than on playing a game. Existing trading simulations mimic the inconveniences of trading stocks on real markets; while this might help future traders to practice, it is out of place for the typical internet user. PitFail's philosophy is that the market for trading *practice* is already well-handled by games such as Investopedia. PitFail instead believes that it is more important to teach theory than mechanics. In contrast with the existing alternatives, PitFail offers number of differentiating features: while the core program centers around buying and selling of liquid assets (stocks, options; anything with available market prices), PitFail aims eventually to users to trade directly with each other in non-liquid assets such as derivatives. To achieve a low-threshold for getting in to the game, PitFail may be played using users' existing accounts (such as Twitter, smart phones or Facebook) with essentially no setup.

PitFail creates a virtual stock world, creating a network of stock investors, through which they trade real-world stocks without the risk of losing real money. Unlike existing trading simulations, PitFail does not require the players to go through a time-consuming registration process. Players can login to the system using their existing e-mail addresses and the system remembers the players for their next use. As such, PitFail requires essentially no commitment and it is easy for players to get started. Initially, the player is given a fixed amount of startup funds and uses these funds to buy virtual stocks.

You could take a trading game different ways -- Investopedia, which is excruciatingly tied to the real world, or Neopets which is isolated and pristine -- but the nice thing about capitalism is that we can play with any rules, so long as they're consistent. But so many (all that I'm aware of) of the games that have been written so far left out something so important: you can't enter (enforced) contracts with other players.

It's not a trivial detail -- if you can't enter contracts, you can't turn intangible ideas into *assets* -- ie, you can't commoditize all the things you might like to commoditize (well, maybe you can if that's nothing). There's a good reason they don't do this, of course: to enforce contracts you either need a legal system (doable -- Wikipedia has one, but a serious impediment still) or contracts that a computer can enforce. PitFail makes a compromise -- users can enter into contracts (in the form of derivatives), but the rules are reduced to a simple set that the system can enforce, yet that can be combined creatively by the players.

This adds a new aspect to the game -- illiquidity. The PitFail stock exchange is simulated as a perfectly efficient, perfectly liquid market. This is of course unrealistic -- in the real world, trading volume is finite, trades are not made constantly, not all trades are made at the marginal price. Alas, it would be hard for PitFail to simulated illiquidity in stocks -- unless we have access to an actual massive population of traders, it would be simply *too* illiquid to be worth playing.

There are many options for a player to choose from once he/she logs in:

1. Player can join a team (a small group of already registered players). Once player joins a team, the player will buy/sell/compete with other players/teams using collective portfolio of the team.

2. Player can join a league (a group of already registered players) where the members of a league compete with each other using their individual portfolio.

3. Player can play in the "Global League" which includes all players.

When the player trades and builds a portfolio, the system should have access to real-time stock information and should adjust the value of a player's investments based on this real time stock info. PitFail retrieves actual stock prices from a third-party source Yahoo! that monitors stock exchanges and maintains up-to-date (though delayed) stock prices. If the corresponding actual stock loses value on a real-world stock exchange, the player's virtual investment loses value equally. Likewise, if the corresponding actual stock gains value, the player's virtual investment grows equally.

As a game, a crucial part of the application is maintaining player portfolio. The application provides every player with portfolio to view his or her history and modify his or her current investments (i.e. currently

owned stocks and derivatives). In addition to the securities currently owned by the player, the player is able to view a few summary statistics about their portfolio, such as a history of net worth over time, and an indication of which assets have increased in value since their purchase. What the player ultimately cares about, of course, is net worth in the future -- that's what they are trying to optimize. We can't tell them that, of course, nor should we, since it's the whole point of playing the game. We should even be careful in categorizing assets by change in value -- users will of course purchase assets that perform oppositely to hedge risk. Basically, we don't want to decide strategy for the player; we want to give them information and let them decide strategy.

To add a flavor of a game, players can monitor each other's progress by viewing a feed of recent activity and browsing leader boards. PitFail also offers aggregate feeds of recent activity. This allows a group of people to keep abreast of their friends' or enemies' activities. Remember, this is not real personal information we're talking about -- we're willing to sacrifice privacy (if you can call it that) for a competitive spirit. PitFail provides the players with the ability to comment on other's trades when browsing recent activity or viewing another user's portfolio. These comments make players feel involved and part of a larger community. One additional feature PitFail provides is the ability for players to "upvote" and "downvote" trades based on their opinion of trade. PitFail can then rank users and assign status symbols (e.g. badges) to users with the strongest ability to vote predictively. Of course, predicting is only so good if you can't make good trades yourself -- but it's interesting to see both rankings nonetheless. This type of ranking appears to be unique to PitFail. Another feature that appears to be unique to PitFail is that it allows users to design their own securities (i.e futures or options) , thus creating new financial products. Even without a court system to enforce complex contracts, custom securities allow PitFail's users to a new financial environment.

As mentioned, PitFail can be accessed via a website, Twitter, Facebook, or an Android application. Each of these methods have their own purposes. As financial trades are compact and atomic and that they can be expressed through small messages, PitFail provides a Twitter and Facebook interfaces where players can buy/sell securities by tweeting to a particular account/ writing post on Facebook account wall . Twitter and Facebook provide a familiar interfaces to use the system. Also, as no registration is required which makes it easy to use. PitFail can also be accessed via a website that offers additional set of features (In addition to all of the functionality provided by the Twitter interfaces): like view portfolio, design custom securities, interact socially with other users and play against or in co-operation (teams/leagues) with other users. Also, website helps to generate some advertising revenue, making it desirable to attract users to the PitFail website by offering features that are not possible via Twitter/Facebook. Android interface provides features that are similar to that of the website, with the addition of notifications to the user when some event occurs within PitFail.

The motivation for implementing teams/leagues comes from the apparent fact that most (perhaps all) trading games target students and teachers as their principal user base, suggesting this accounts for most of the people who actually play these games. While PitFail is mostly seeking a different niche -- the casual online player -- the classroom market is too big to ignore completely, hence a feature that makes it possible for students to play against each other in a league.

Below is the list of customer requirements:

1. **REQ-1** Stock Market Simulator Website: Investors are looking for an effective tool that allows users to invest and learn without having to invest real money and also allows them to interact with other users more effectively to make the game really enjoyable.

2. **REQ-2** Android Application: Mobile users who like having native applications can use such system with quick access.

3. **REQ-3** Access via Twitter/Facebook: Users who heavily use social networks like Facebook/Twitter can connect to PitFail easily.

4. **REQ-4** Simple User Interface: Users are looking for simple interface that welcomes new users and guides the new user through portfolio management.

5. **REQ-5** Zero-Configuration Setup: Users should not have to set any settings or explicitly create an account to begin playing.
6. **REQ-6** Updated Stock Information: Application should present stock symbols, company names, stock history, updated stock values and prices amongst other details.
7. **REQ-7** Basic Trading: Users should be able to buy and sell stocks whose values change over time.
8. **REQ-8** Large, Liquid, Efficient Market: The simulated "exchange" should present the illusion of a large, liquid and efficient market -- stocks are traded constantly, at marginal price, and each individual trade is small compared to the total trading volume.
9. **REQ-9** Relation to the outside world: The values of stocks should be in some way related to the outside world so that users have information to base trading decisions on.
10. **REQ-10** Player Portfolio: Each player must have separate portfolio that gives him/her option to buy/sell new securities, view currently owned securities.
11. **REQ-11** Evaluate Portfolios: Securities owned by each player should be periodically evaluated and should be updated to their current value.
12. **REQ-12** Advertisements: The website must contain appropriate and interesting advertisements relating to finance and stock
13. **REQ-13** Coordinators for Supervision: Users must be able to create their own leagues.
14. **REQ-14** Summary Statistics: The website should provide users with a few summary statistics about their portfolio -- aggregate value over time, which securities have increased in value. The website shouldn't usurp the role of deciding strategy for the player; only the most basic of stats should be displayed.
15. **REQ-15** Voting: players should be able to up/down-vote each other's trades. Vote tallies should be visible to other users.
16. **REQ-16** Commenting: players should be able to comment (via the website -- you can already comment on anything via Twitter) on each other's trades. Comments should be visible to all users.
17. **REQ-17** Moderation: There should be at least a minimal degree of comment moderation so blatantly offensive comments can be removed.
18. **REQ-18** Designing Derivatives: Players should be able to enter into contracts with each other that will be enforced by the PitFail system.
19. **REQ-19** Guided designing of derivatives: The website should guide players into common formats for derivatives to make it easier for new players to figure out.
20. **REQ-20** Rankings: On the website players should be able to see rankings of all players by portfolio value (liquid assets only), and by voting score.

10 Functional Requirements Specification

10.1 Actors and Goals

- A *Web Player* (or *WebPlayer*) is a *player* who interacts with the *game* via the web browser interface. The web interface also provides access to the command based interface.

  - Buys and Sell Stocks.
  - View and Modify Portfolio.
  - Create League.
  - Participate in Leagues.
  - Wants to effectively administer the tournament to provide either a learning experience to the *players*, or, alternately, an enjoyable experience to the *players*.

- An *Administrator* is a *WebPlayer* who is designated as having administrator control of a *league*. This control allows the *Administrator* to invite addition *players* to the league.
  - Invites players to a *league*.
  - Wants to increase participation in their *league* or add the set of *players* the *league* is intended for.

- A *Twitter Player* (or *TwitterPlayer*) is an indirect *player* who interacts with the *game* via the *Twitter* actor. They are the originator of the commands received from the *Twitter* actor.
  - Buys and Sells Stocks
  - Examines their portfolio

- A *MobilePlayer* is a *player* who interacts with the *game* via the Android interface. This actor contains has limited use cases compared to a Web Player.
  - Buys and Sells Stocks
  - View Portfolio
  - Participate in Leagues

- The *database* is the store for all persistent data on interactions with the *system*. It stores data regarding all user portfolios and the association of authentications with users.

- A *stock information provider* is a supplier of stock pricing data for the present (within the margin of some minutes). They are queried for all data regarding actual market numbers. Currently, *Yahoo* is the *stock information provider* (via its Yahoo Finance API).

- *Authentication providers* allow us to uniquely identify users and associate some stored state with their unique identification.

- *Twitter* is utilized both as a authentication provider (for all *players*) as well as an interface to the service. This actor provides a stream of text based commands from the indirect actor *Twitter Player*.

10.2 Use Cases

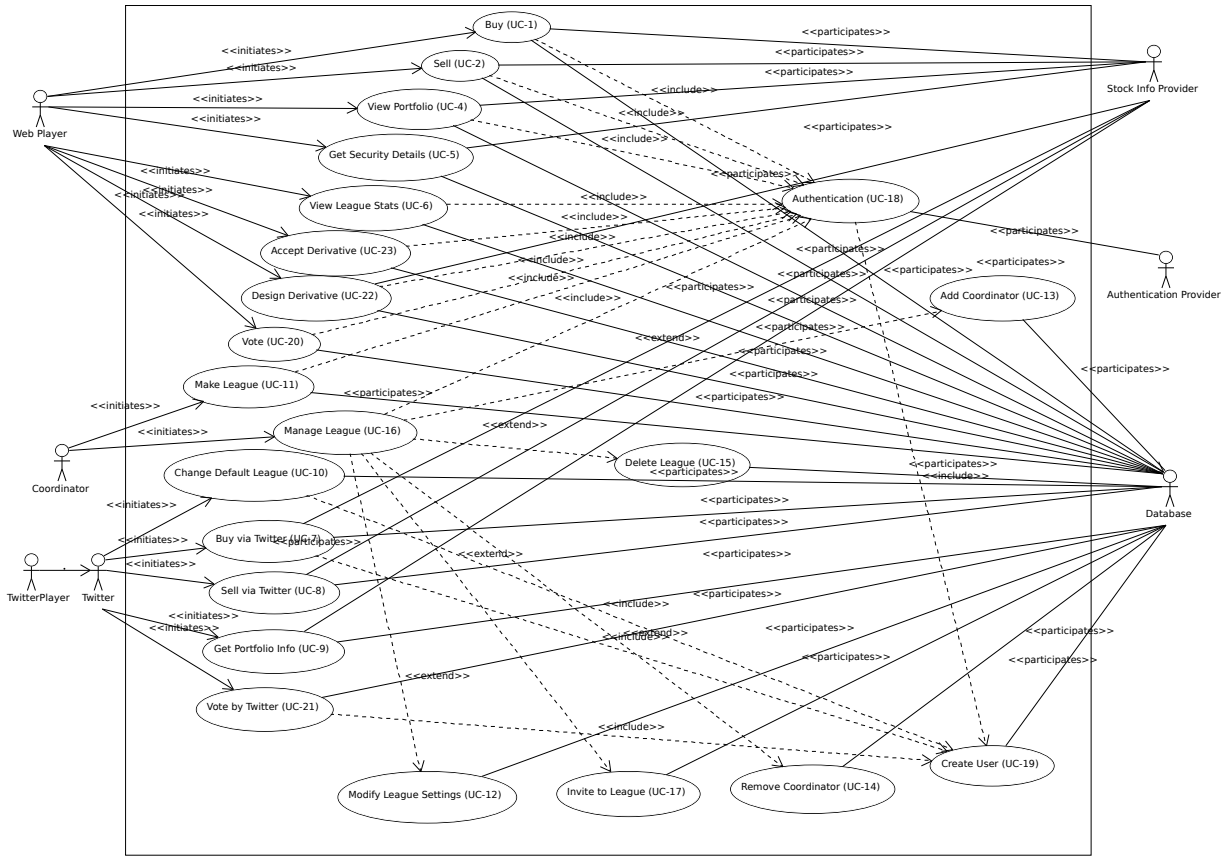


Figure 85: A diagram of use cases and actors showing interactions and relationships.

10.2.1 Listing of Use Cases

1. Buy, Actor: WebPlayer, TwitterPlayer, MobilePlayer Purchases a security from the market at the price listed by the designated market makers.

2. Sell, Actor: WebPlayer, TwitterPlayer, MobilePlayer Sells a held security at the price listed by the designated market makers.

3. View Portfolio, Actor: WebPlayer, TwitterPlayer, MobilePlayer Allow the initiating actor to examine the contents of their portfolio. Information regarding their current assets and liabilities as well as how they have been progressing over time may be displayed.

4. View League stats, Actor: WebPlayer. Display information regarding the entire league including a listing of all portfolios, graphs of the top portfolios, and the distribution of stocks held by the top portfolios

5. Invite to League, Actor: WebPlayer. Send an invitation to a *player* to join a *league* of which the actor is an administrator.

6. Create League, Actor: WebPlayer. Create a new *league* with the creator as the administrator.

7. Accept or Decline League Invitation, Actor: WebPlayer. Acceptance of an outstanding invitation allows the initiating actor to create *portfolios* within the *league* the invite was dispatched from.

8. Create Portfolio, Actor: WebPlayer, MobilePlayer Creates a new *portfolio* associated with a particular *league*. This *portfolio* is created with the cash value indicated by the *league* in which it is created.

9. Get Security Details, Actor: WebPlayer, TwitterPlayer. Display information regarding a particular security (stock or bond) such as historical trends and statistics.

10. View Portfolio, Actor: WebPlayer. Display the contents of one of the initiating actor's portfolios. This includes a listing of assets and liabilities, and graphs showing the change in portfolio value over time.

11. Create Initial Portfolio, Actor: WebPlayer, TwitterPlayer. On the initiation of an action by the *player*, a default *portfolio* is created for them within the default *league*.

12. Authentication, Actor: WebPlayer. The initiating actor authenticates to the server via an external *authentication provider*. *Twitter* is the current supported *authentication provider*. Authentication for the *TwitterPlayer* is provided external to our system.

12. Vote, Actor: WebPlayer The initiating actor votes on a particular trade, investing from a portfolio in that trade being successful or unsuccessful.

13. Comment, Actor: WebPlayer The initiating actor adds a globally visible snippet of text to an 'event' (a trade occurred, a derivative was offered, et c.) within the system.

14. Create Derivative, Actor: WebPlayer

15. Bid on Derivative, Actor: WebPlayer This is a part of the public auction system. The initiating actor places a particular cash value bid on a derivative currently in public auction.

17. Execute Derivative, Actor: WebPlayer On derivatives that allow it, this causes the early evaluation of the derivative's terms.

18. Close an Offer, Actor: WebPlayer Close an offer at auction, confirming the sale to the highest bidder.

19. Accept or Decline a pending offer, Actor: WebPlayer When a direct offer of a derivative is made to a *portfolio* controlled by the initiating actor, the actor must accept or reject this offer (or leave it outstanding, cluttering the interface to some extent).

20. Open Buy Order, Actor: WebPlayer Create an order for the purchase of a particular amount of a particular *security* at a particular dollar value per share (a limit order).

21. Open Sell Order, Actor: WebPlayer Create an order for the sale of a particular amount of a particular *asset* (a held *security*) at a particular dollar value per share (another limit order).

22. Cancel a Buy or Sell Order, Actor: WebPlayer When a Buy or Sell order is placed via the Buy or Sell use case, the orders are kept alive indefinitely unless canceled. A user who wishes to remove orders which they no longer want active will cancel the order.

10.2.2 Fully Dressed Use Cases

10.2.2.1 UC-1: Buy

Related Requirements: REQ-1, REQ-2, REQ-6, REQ-7, REQ-8, REQ-9

Initiating Actor: Any of: WebPlayer, TwitterPlayer, MobilePlayer

Actor's Goal: To purchase a security from the market, to add it to his portfolio, and see his updated portfolio.

Participating Actors: Database, Stock Information Provider.

Preconditions: The user should have logged in.

Postconditions: The user needs to be able to see his purchased security in his portfolio and track the progress of the security in his portfolio until he "SELLS" it.

Flow of Events for Successful Buy:

1. → The *Player, WebPlayer, or TwitterPlayer* determines a *Security* and how much of it to "BUY". This is sent to the *System*
2. → *System* signals the *Stock Information Provider* for the price of the security.
3. ← *Stock Information Provider* sends the price of the *Security* to the *System.*
4. → *System* requests the amount of cash the *Player* has from the *Database*.
5. ← *Database* returns the amount of cash for the *Player* to the *System*.
6. → *System* checks that there is enough money for complete the transaction and sends the complete transaction for a *Player*, *Security*, and the quantity to the *Database*.
7. ← *Database* signals the *System* the transaction is complete.
8. ← *System* signals to the *Player* that the Buy operation was completed successfully.

Flow of Events for Unsuccessful Buy:

1. → The *Player, WebPlayer, or TwitterPlayer* determines a *Security* and how much of it to "BUY". This is sent to the *System*
2. → *System* signals the *Stock Information Provider* for the price of the security.
3. ← *Stock Information Provider* sends the price of the *Security* to the *System.*
4. → *System* requests the amount of cash the *Player* has from the *Database*.
5. ← *Database* returns the amount of cash for the *Player* to the *System*.
6. ← *System* checks that there is enough money for complete the transaction. There is not enough money. *System* signals to the *Player* "Transaction Not Completed: Insufficient Funds."

10.2.2.2 UC-2: Sell

Related Requirements: REQ-1, REQ-2, REQ-6, REQ-7, REQ-8, REQ-9

Initiating Actor: Any of: WebPlayer, TwitterPlayer, MobilePlayer

Actor's Goal: To purchase a security from the market, to add it to his portfolio, and see the updated portfolio

Participating Actors: Database, Stock Information Provider

Preconditions:

- User is authenticated (logged in).
- Contain in his portfolio at least the quantity of securities his is requesting to sell.

Postconditions:

- The user's portfolio will reflect the quantity of securities sold.

Flow of Events for Successful Sell:

1. → The *Player* determines a *Security* and how much of it to "SELL". They send this information to the *System*.
2. → *System* requests the price of the security from the *Stock Information Provider*
3. ← *Stock Information Provider* sends the price of the *Security* to the *System*.
4. → *System* requests the amount of the *Security* the *Player* owns from the *Database*.
5. ← *Database* returns the amount of the *Security* the *Player* has to the *System*.
6. → *System* checks that there are enough *Securities* to complete the transaction. *System* signals the *Database* to complete the transaction for a *Player*, *Security*, and the quantity.
7. ← *Database* returns an indicator of transaction completion to the *System*.
8. ← *System* signals the transaction successfully completed to the *Player*.

Flow of Events for Unsuccessful Sell:

1. → The *Player* determines a *Security* and how much of it to "SELL". They send this information to the *System*.
2. → *System* requests the price of the security from the *Stock Information Provider*
3. ← *Stock Information Provider* sends the price of the *Security* to the *System*.
4. → *System* requests the amount of the *Security* the *Player* owns from the *Database*.
5. ← *Database* returns the amount of the *Security* the *Player* has to the *System*.
6. ← *System* checks that there is enough *Securities* to complete the transaction. There is not. *System* signals that the transaction was not successfully completed due to insufficient funds to the *Player*.

10.2.2.3 UC-3: View Portfolio

Related Requirements: REQ-1, REQ-2, REQ-6, REQ-10, REQ-11, REQ-14

Initiating Actor: Any of: Web Player, Mobile Player, Twitter Player.

Actor's Goal: To view information regarding their portfolio. This information includes the currently owned securities, minimal statistics regarding those securities (as they relate to the current and past value of the portfolio), current available capital (and similar minimal information regarding its change), and the overall value of the portfolio (also with some statistical information regarding changes over time). The actor desires this information to make decisions regarding what their next interaction with the system should be. They use this info to decide to sell stock they have or buy an increased number of shares of stock they have).

Participating Actors: *Stock Information Provider*, *Database*

Preconditions:

- User is authenticated.

Postconditions: Information is displayed to the user, but no internal actions are taken. Nothing about the users portfolio will be modified by this action.

Flow of Events for Main Success Scenario:

1. → *Player* requests a view of their *portfolio*.
2. → *System* requests the information about the user's portfolio for this particular league from the *Database*.
3. ← *Database* returns the information regarding the portfolio.
4. → *System* forms a query regarding all the currently held securities within the portfolio and dispatches it to the *Stock Information Provider*.
5. ← *Stock Information Provider* returns the requested data.
6. ← *System* forms a view of the portfolio information and returns it to the *Player*

10.2.2.4 UC-4: View League Statistics

Related Requirements: REQ-1, REQ-6, REQ-9

Initiating Actor: WebPlayer

Actor's Goal: To view the performance of his or her portfolio relative to other league members. For a teacher, this may also be used to verify that his or her students are actively participating in the game.

Participating Actors: Database

Preconditions: The league that is being viewed exists and the league is either public or the user is a member.

Postconditions: None; this is a stateless action.

Flow of Events for Main Success Scenario:

1. → *Player* requests to view league performance.
2. ← *System* signals the *Database* for authentication and the league's leaderboard.
3. ← *Database* authenticates the user's ability to view the statistics and returns the league's leaderboard.
4. ← *System* returns a leaderboard of all league members.

Flow of Events for league does not exist:

1. → *Player* requests the league statistics page.
2. ← *System* signals the *Database* for authentication and the league's leaderboard.
3. ← *Database* signals the *System* that the league does not exist.
4. ← *System* returns "page not found" error.

10.2.2.5 UC-5: Invite to League

Related Requirements: REQ-1, REQ-14, REQ-20

Initiating Actor: Administrator

Actor's Goal: To modify settings for the coordinator's league. This includes modifying the league's name, nickname, starting funds, and security settings.

Participating Actors: Database

Preconditions:

- League that is being modified exists
- Initiating actor is a coordinator of the league that he or she is modifying

Postconditions:

- League name is still unique
- League nickname is still unique
- Starting funds is positive

Flow of Events for Main Success Scenario:

1. → *Coordinator* requests to view league settings page.
2. ← *System* signals the *Database* for authentication and the league's settings page.
3. ← *Database* authenticates the user's ability to modify the league settings and returns the league settings page.
4. ← *System* returns a league setting page populated with the current settings.
5. → *Coordinator* submits updated league settings.
6. ← *System* Validate new league settings
7. ← *System* sends updated settings to the *database.*
8. ← *Database* signals the *System* that the settings have been updated.
9. ← *System* signals the *Coordinator* "Settings have been updated."

Flow of Events for league does not exist:

1. → *Player* requests the league settings page.
2. ← *System* signals the *Database* for authentication and the league's settings page.
3. ← *Database* signals the *System* that the league does not exist.
4. ← *System* returns "page not found" error.

Flow of Events for user is not a coordinator of the league:

1. → *Player* requests the league settings page.
2. ← *System* signals the *Database* for authentication and the league's settings page.
3. ← *Database* signals the *System* that the league is invite-only and the *Player* is not a member.
4. ← *System* returns "access denied" error.

10.2.3 Use Case Traceability Matrix

The following is the relationship between the use-cases defined above and the requirements discussed in the statement of requirements:

- **UC-1:** REQ-1, REQ-2, REQ-6, REQ-7, REQ-8, REQ-9
- **UC-2:** REQ-1, REQ-2, REQ-6, REQ-7, REQ-8, REQ-9
- **UC-3:** REQ-1, REQ-20
- **UC-4:** REQ-1, REQ-2, REQ-6, REQ-10, REQ-11, REQ-14
- **UC-5:** REQ-1, REQ-6, REQ-9
- **UC-6:** REQ-1, REQ-14, REQ-20
- **UC-7:** REQ-3, REQ-6, REQ-7, REQ-8, REQ-9
- **UC-8:** REQ-3, REQ-6, REQ-7, REQ-8, REQ-9
- **UC-9:** REQ-3, REQ-6, REQ-10, REQ-11, REQ-14
- **UC-10:** REQ-3, REQ-20
- **UC-11:** REQ-1, REQ-13, REQ-17
- **UC-12:** REQ-1, REQ-13, REQ-17
- **UC-13:** REQ-1, REQ-13, REQ-17
- **UC-14:** REQ-1, REQ-13, REQ-17
- **UC-15:** REQ-1, REQ-13, REQ-17
- **UC-16:** REQ-1, REQ-13
- **UC-17:** REQ-1, REQ-13
- **UC-18:** REQ-1, REQ-4, REQ-10, REQ-11, REQ-17
- **UC-19:** REQ-1, REQ-4, REQ-5, REQ-10, REQ-11
- **UC-20:** REQ-1, REQ-2, REQ-15, REQ-20
- **UC-21:** REQ-3, REQ-15, REQ-20
- **UC-22:** REQ-1, REQ-18, REQ-19
- **UC-23:** REQ-1, REQ-2, REQ-18, REQ-19

11 Nonfunctional Requirements

11.1 Usability

The website should be easy to navigate irrespective of the type of user. It should have an appealing user interface which is pleasant to the eyes. A through consideration should be given for its aesthetic design in order to make it easily navigable and to have a good readability. The key focus should be on making the user interface as interactive as possible.

11.2 Performance

In order to have a great performance, the website should be as lightweight as possible by keeping minimum hardware demands. For it to be efficient, any task initiated by the user should be completed in a timely manner. The web server should be able to serve multiple requests and when a large number of users are logged in.

11.3 Reliability

In case of Internet failure, the user's portfolios should be brought back to a consistent state when user logs in the system again after the failed internet connection. The system should keep a backup of user's data in case of server failure. A proper care should be taken to handle a situation where a particular stock source is not available (i.e. Yahoo).

11.4 Security

The system should be secure enough such that user's privacy is maintained. The system should have a login process irrespective of the application i.e via Website, Mobile or Twitter interface.

11.5 Supportability/Extensibility

It should be feasible to extend any server components and include improved versions of modules which can be installed only by administrators. For future purposes of handling the load, it should be easier to include more number of servers to achieve load balancing. The system should be platform independent so that it is easy to move to newer technologies or the next versions of web server.

11.6 Maintainability

The system should be easy to maintain for the administrator. The administrator should be provided with an interface to interact with the entire system to make changes and to recover from any failure manually as well. The interface should give the administrator enough capability to perform future maintenance.

11.7 Testability

The system should be flexible enough to allow creating test databases and fake players so that feature test does not need to manipulate the actual database. This would ensure that it has great testability which can be used to build a more robust

11.8 Consistency

It should be ensured that the application is consistent throughout irrespective of what interface the player is using i.e whether website, mobile application or Twitter interface. Functionality might be limited on these different interfaces but it should not difficult for the user to shift from one application to another to access the system. Buzz words used should be same throughout and on all the interfaces to avoid confusion.

11.9 Documentation

The website should have enough material in the form of tutorial which can help the user to understand the rules and policies of the Stock fantasy league game and how it works.

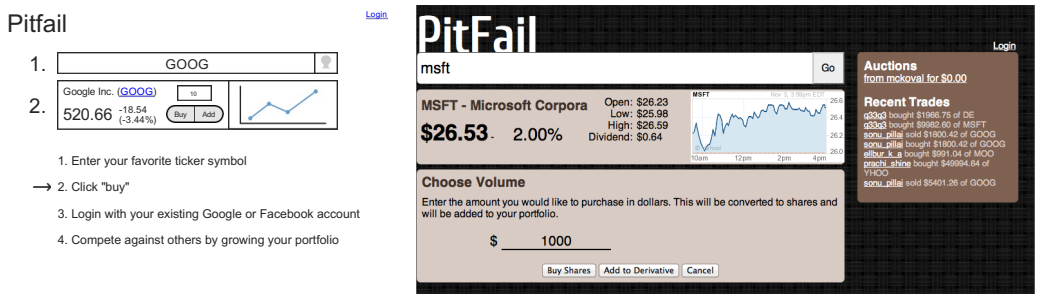
12 User Interface

12.1 User Interface Design and Implementation

PitFail's overall user interface closely resembles the interface depicted in its mockups: most of the changes were merely cosmetic. Most of the functional changes are because the current implementation of PitFail is missing features that were included in the mockup: e.g. companies, leagues, and social interaction. These changes are grouped into general categories, described in detail, and justified in the following sections.

12.1.1 Welcome Page for New User

PitFail was originally described as having a "guided registration" process where the user registers as part of purchasing his or her first stock. While the user can still explore the stock purchasing interface before logging in, the current implementation of PitFail does not support this "zero effort" registration because of a technical limitation. As such, guided messages no longer are displayed next to each step in the purchasing pipeline:

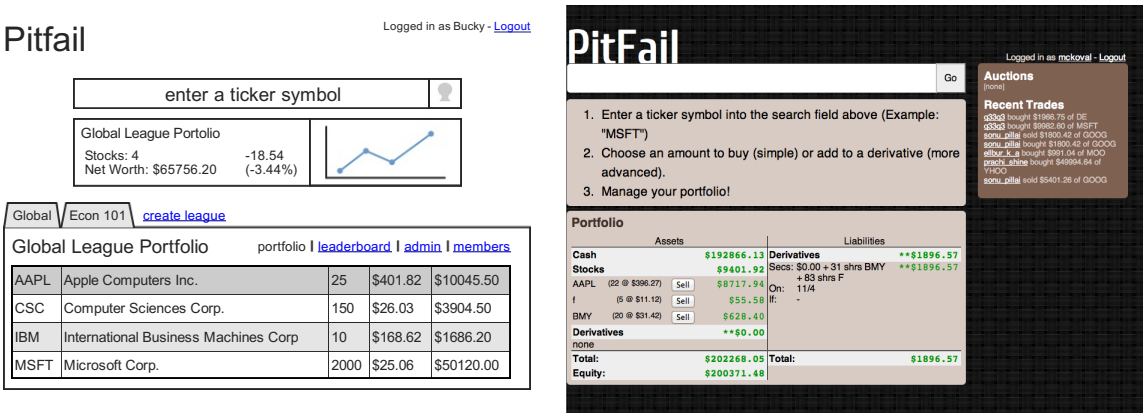


Note that the list of steps is not visible and the current step is not indicated with an arrow. Some form of guided registration will be implemented in the next version of PitFail. Thankfully, this doesn't change user effort: the user simply must login *before* selecting a stock instead of *after* selecting a stock.

12.1.2 Portfolio Management

Perhaps the largest change from the original mockups to the current implementation is the user's portfolio. This was planned to be displayed as a single large table containing the all of the user's assets: a combination of cash, stocks, and derivatives. This design made it difficult to visually differentiate between types of assets and to locate an asset of interest.

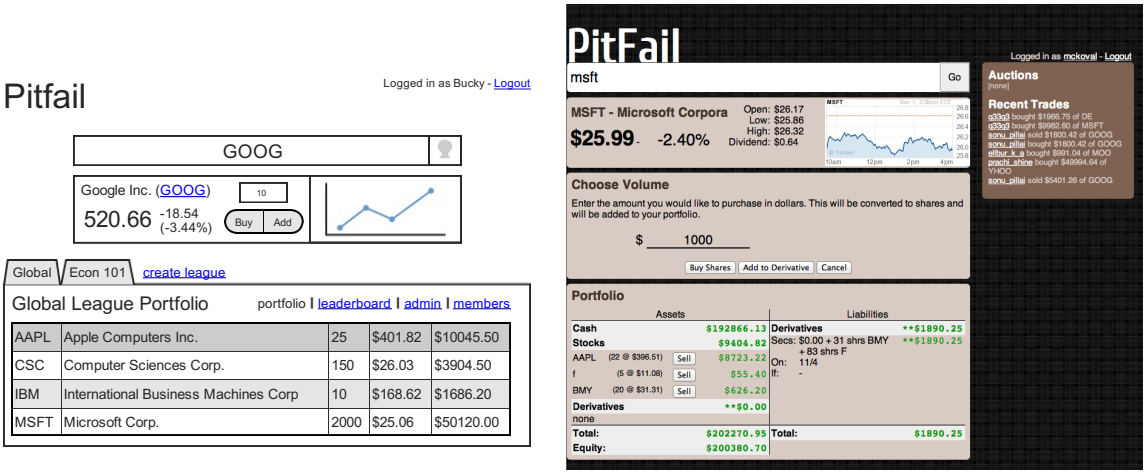
Instead, the portfolio displayed as a "T"-chart, splitting assets and liabilities into two separate columns. The assets column is further subdivided by the type of asset: cash, stocks, and derivatives. These subdivisions allow the user to quickly locate an asset of interest, for example, when selling a stock. Each column is summarized with a "total" row that estimates the current value of his or her portfolio by approximating the value of derivatives as if they were immediately executed. While none of these changes dramatically alter user effort relative to the mockup, reformatting the portfolio as a "T"-chart and adding this additional information makes it much easier for a user to view his or her current assets at a glance:



Besides the changes to the table of assets, there are clearly several features missing from the implementation: (1) historic portfolio performance, (2) multiple portfolios, and (3) league navigation. These missing interface elements will be restored after companies, leagues, and logging of historic prices are implemented in the next iteration of PitFail.

12.1.3 Buying Stocks

Purchasing stocks is one of the fundamental activities on PitFail. The interface for buying stocks is very similar to the interface shown in the original mockups: when the user enters a valid ticker symbol in the large search bar, a small stock quote expands below the search bar. This quote includes a few statistics about the stock's daily performance and a graph of the stock's performance over time.

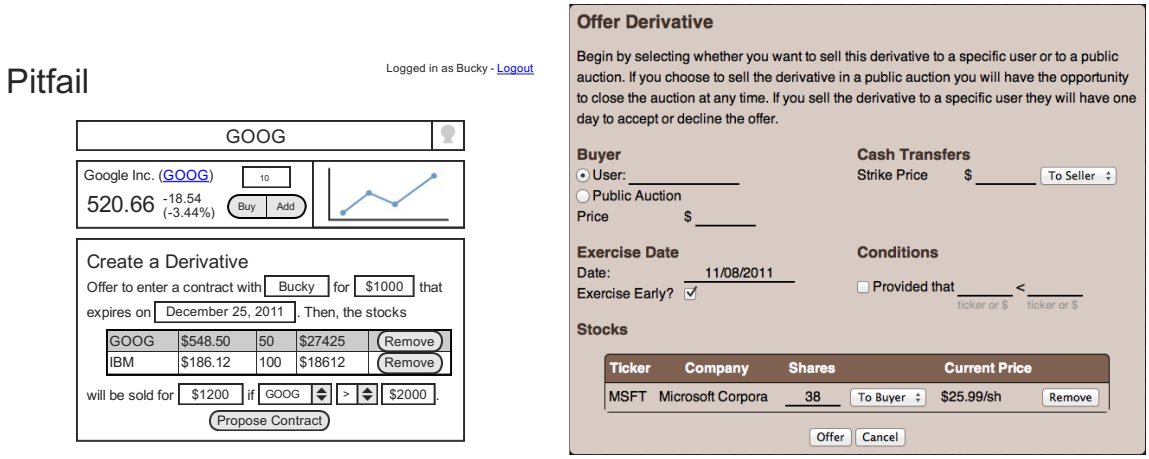


Unlike the original mockup, the options for interacting with the stock are not embedded in the stock quote. Instead, they are displayed in a dedicated section of the webpage. This extra space is used to display a short description of stock trading and helps guide new users through the process: something that will be even more important once options are supported. While the original mockups allowed the user to enter an amount in either shares or dollars, this was found to be confusing and was removed in the current version of the user interface.

Neither of these changes do not considerably effect user effort.

12.1.4 Trading Derivatives

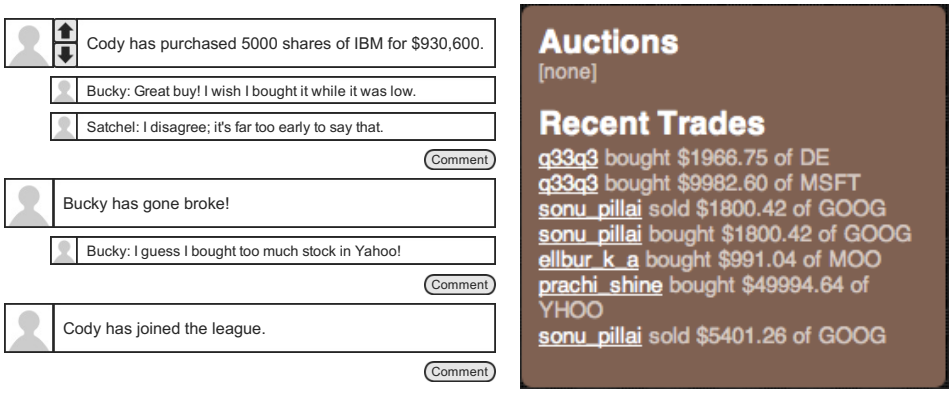
If the user clicks the "add to derivative" button instead of the "buy stock" button, he or she is presented with the derivative offering page. In the original mockups this was shown as a prose-like description of a derivative with a number of blanks. Originally intended to guide the user through the derivative creation process, this was found to be unfeasible with the number of derivative configuration options supported in PitFail. As such, this was redesigned to resemble a traditional form: a prose description followed by a table of input fields.



Once the derivative has been created it can either be offered to a specific user or to a public auction. If a buyer is specified, that user is prompted to accept or decline the offer using a special form in his or her portfolio. If the derivative is offered to a public auction, a link to the auction page is added to the sidebar and other users have an opportunity to bid. These features were not included in the mockups, so see the User Effort Estimation section below for a detailed usability analysis.

12.1.5 Social Features

PitFail's original mockups included a real-time newsfeed at the bottom of every page. This news feed was a log of trading history and served as a hub for social interaction between users. A limited implementation of this newsfeed is included in the current version of PitFail. Unlike the mockup, the newsfeed is included in every page's sidebar instead of the footer. This is similar to the real-time feed that was recently added to Facebook and will be familiar to the majority of PitFail's users.



Besides the different location, much of the functionality displayed in the mockups has not yet been

implemented. Notably, this includes: (1) user-specific newsfeeds, (2) voting, (3) commenting, (4) messages for derivative trades, and (5) messages for a users going broke. These features will be implemented in the next version of PitFail and do not effect user effort.

12.2 Effort Estimation using Use Case Points

Several of the most common usage scenarios for the PitFail website are evaluated below. In particular, note that common scenarios (e.g. buying a stock) are much easier to perform than rare scenarios (e.g. creating a new league):

Usage Scenario	**Clicks**	**Keystrokes**
purchase a stock	3	7
create a derivative	4	27
act on a pending derivative offer*	1	1
bid on a derivative auction*	4	5
close a derivative auction*	1	1
sell a stock	3	2
create a new league	n/a	n/a
modify an existing league	n/a	n/a
invite a user to a league	n/a	n/a

Features that are not currently implemented are shown as empty rows and actions that have been added since the original mockups are marked with asterisks. Both these new usage scenarios and existing usage scenarios that were modified are analyzed in detail below. This includes buying and selling stocks because of the lack of league support in the current version of PitFail.

12.3 Purchase a Stock

Assume the user wishes to purchase 10 shares of Google stock. The user must:

- **Navigation:** total of one click, as follows
  1. Click on “login”.
- **Data Entry:** total of two clicks and seven keystrokes, as follows
  1. Click on the “enter a ticker symbol” text field.
  2. Press the keys “G”, “O”, “O”, and “G”.
  3. Press “enter” to load the quote.
  4. Press the keys “1” and “0” to specify 10 shares.
  5. Click the “buy” button to confirm the purchase.

Note that the user could press “enter” instead of clicking the “buy” button.

12.4 Creating a Derivative

Assume the user wishes to offer a call option to Bucky that includes 10 shares of Google stock and expires on December 25, 2011. This option costs $1000 to begin active and one can buy the shares for $10,000 if and only if the market rate for Google stock is greater than $1000 per share. The user must:

- **Navigation:** total of one click, as follows

    1. Click on “login”.

- **Data Entry:** total of 3 clicks and 27 keystrokes, as follows

    1. Click on the “enter a ticker symbol” text field.
    2. Press the keys “G”, “O”, “O”, and “G”.
    3. Press the “enter” key to load the quote.
    4. Press the keys “1” and “0” to specify 10 shares.
    5. Click the “add” button to begin creating a derivative.
    6. Press the “B”, “u”, “c”, “k”, and “y” keys to enter the recipient’s name.
    7. Press “tab” to move to the “premium” field.
    8. Press the keys “1”, “0”, “0”, and “0” to enter $1000.
    9. Press “tab” to move to the “expiration date” field.
    10. Press the “1”, “2”, “/”, “2”, and “5” keys to select December 25th of the current year.
    11. Press “tab” to move to the “strike price” field.
    12. Press the “1”, “0”, “0”, “0”, and “0” keys to enter $10000.
    13. Click on the “Propose Contract” button to complete the transaction.

12.5 Sell a Stock

Assume the user wishes to sell 10 shares of Google stock from his or her Global League. The user must:

- **Navigation:** total of one clicks, as follows

    1. Click on “login”.

- **Data Entry:** total of two clicks and two keystrokes, as follows

    1. Click on the text input in the row corresponding to Google.
    2. Press the keys “1” and “0” to specify 10 shares.
    3. Click the “sell” button to confirm the purchase.

    Note that the user could press “enter” instead of clicking the “sell” button.

12.6 Act on Derivative Offer

Assume the user wishes to accept a derivative that was directly offered to him or her:

- **Navigation:** total of one click, as follows

    1. Click on “login”.

- **Data Entry:** total of one click, as follows

    1. Click on the “accept” button next to the correct derivative.

12.7 Bid on Derivative

Assume the user wishes to bid $50,000 on a derivative that is being sold in a public auction:

- **Navigation:** total of two clicks, as follows
  1. Click on “login”.
  2. Click on the correct derivative link in the sidebar.
- **Data Entry:** total of two clicks and five keystrokes, as follows
  1. Click on the “your bid” field.
  2. Press the keys “5”, “0”, “0”, 0“, and ”0“.
  3. Click the ”Cast Bid“ button.

12.8 Close Derivative Auction

Assume the user wishes to close an auction that he or she posted:

- **Navigation:** total of one click, as follows
  1. Click on ”login“.
- **Data Entry:** total of one click, as follows
  1. Click on the ”close“ button next to the correct auction.

13 Class Diagram and Interface Specification

13.1 Comments on the UML

13.1.1 stockdata

It might be surprising that the classes `FailoverStockDB`, `YahooYQLStockDB`, `YahooCSVStockDB`, and `CachingStockDB` do not have any associations with each other. The reason is that, for benefits of testability, the `stockdata` classes are design with dependency injection. This allows each one to be tested individually without requiring any of the others.

When the classes are actually used, they are built into a pipeline, for example (Figure 86):

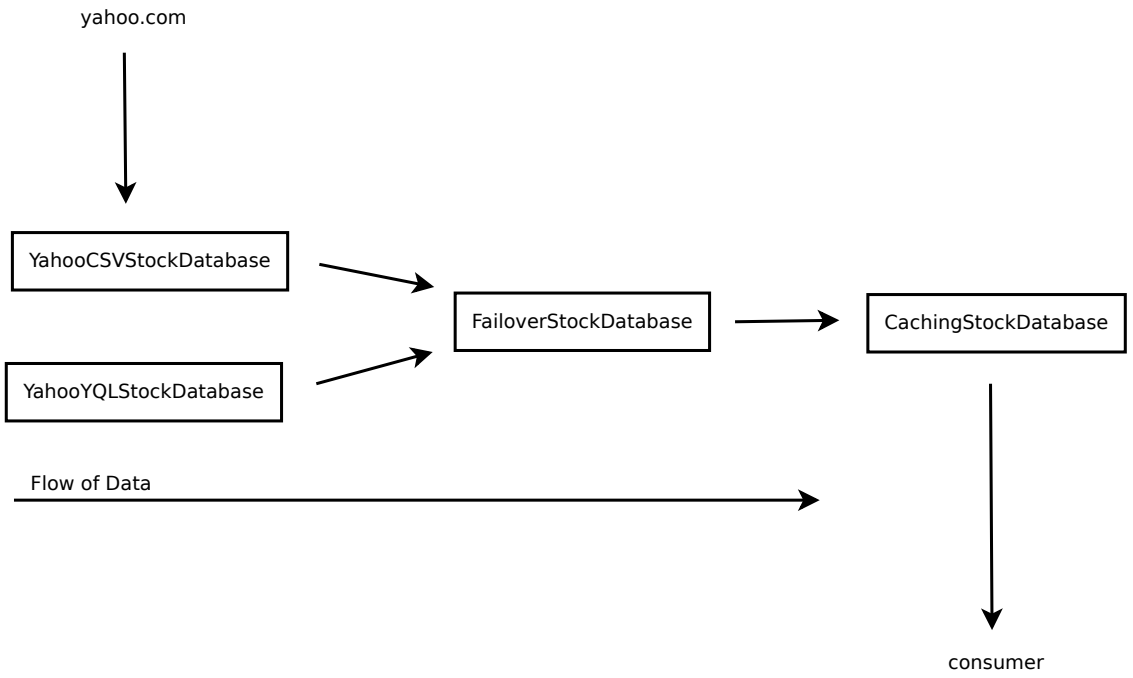


Figure 86: The pipeline architecture of the stock querying classes.

13.1.2 model.schema

These closely resemble the concepts in the Domain Model (See *Domain Model*). The biggest omission is that some of the Domain Concepts do not appear in actual code:

1. The ”execution“ of a trade was represented as a concept, but does not appear in the code as a class.
2. Likewise the ”cancellation“ of a trade.
3. A DividendEvent appeared as a concept but is merely procedural in the code.

Also not that some of the arrows in the UML diagram are encapsulated in association classes (See *Domain Model*).

13.1.3 texttrading

The texttrading code should also be viewed as a pipeline (Figure 87):

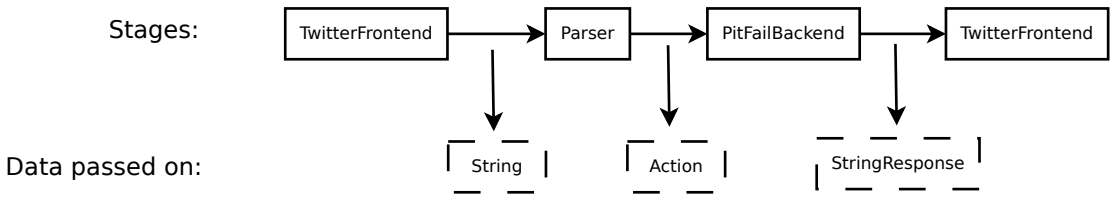


Figure 87: The pipeline architecture of texttrading.

13.1.4 website.control

This is the stateful part of the website: storing the current user, and the portfolio they are currently working with.

13.1.5 website.view

Not all the classes are shown here; however, all of them follow the same structure as the ones shown. A website view class has:

1. Some HTML in the form of Scala inline XML.
2. Some calls into the model to retrieve data.
3. Places in the HTML where data is inserted.
4. Some callbacks for user-input events (submitting a form).
5. Some calls into the model to perform the requested action.

The view is intended to be as thin a layer as possible on top of the real logic in the model.

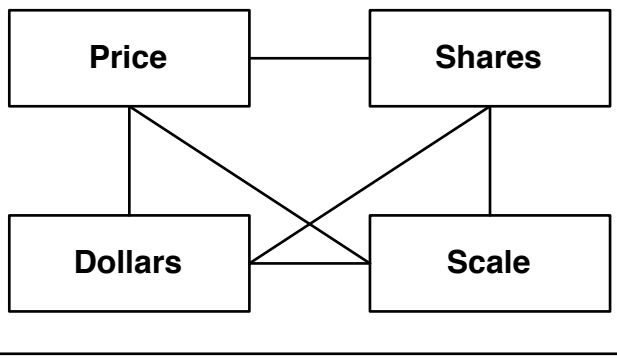
13.1.6 Android view

The Android is another View in the MVC architecture; the classes shown:

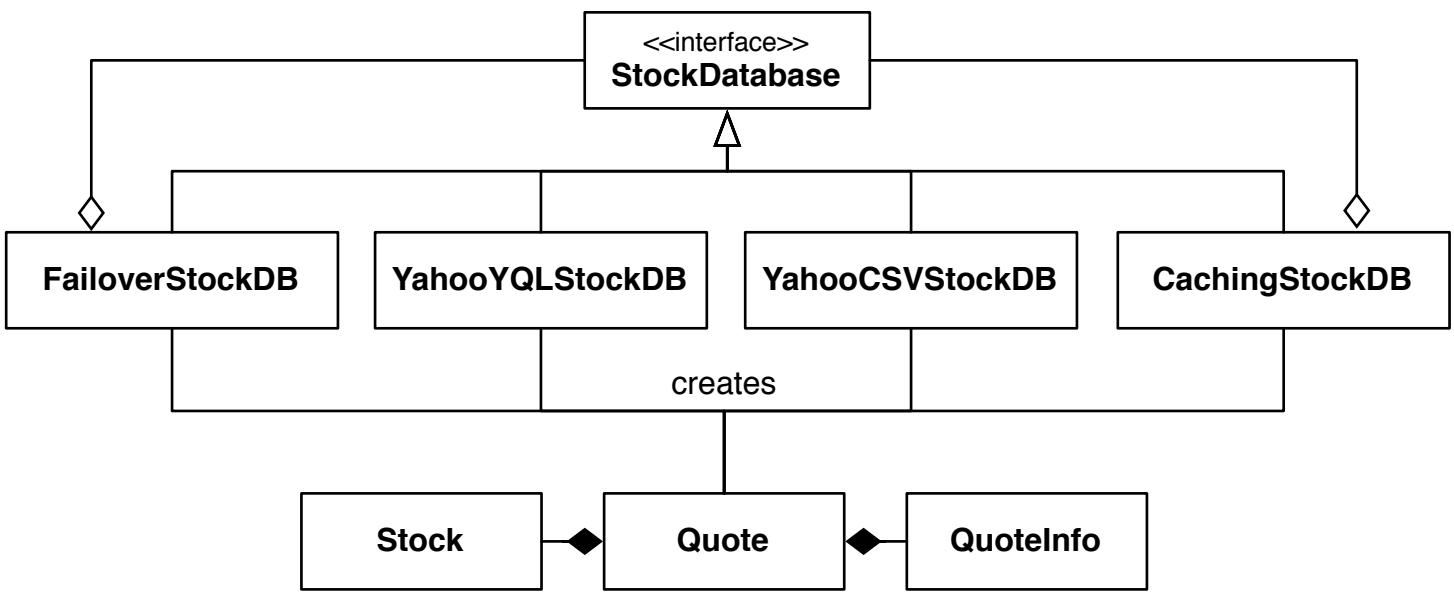
2. Make calls into the model (via servlets) to retrieve data.
1. Take user input (via the Android UI).
3. Make calls into the model to perform the user's actions.

The only real difference between the design of the android and website frontends is that the Android UI has no choice but to run on a separate machine, and so it must retrieve data and perform actions over HTTP.

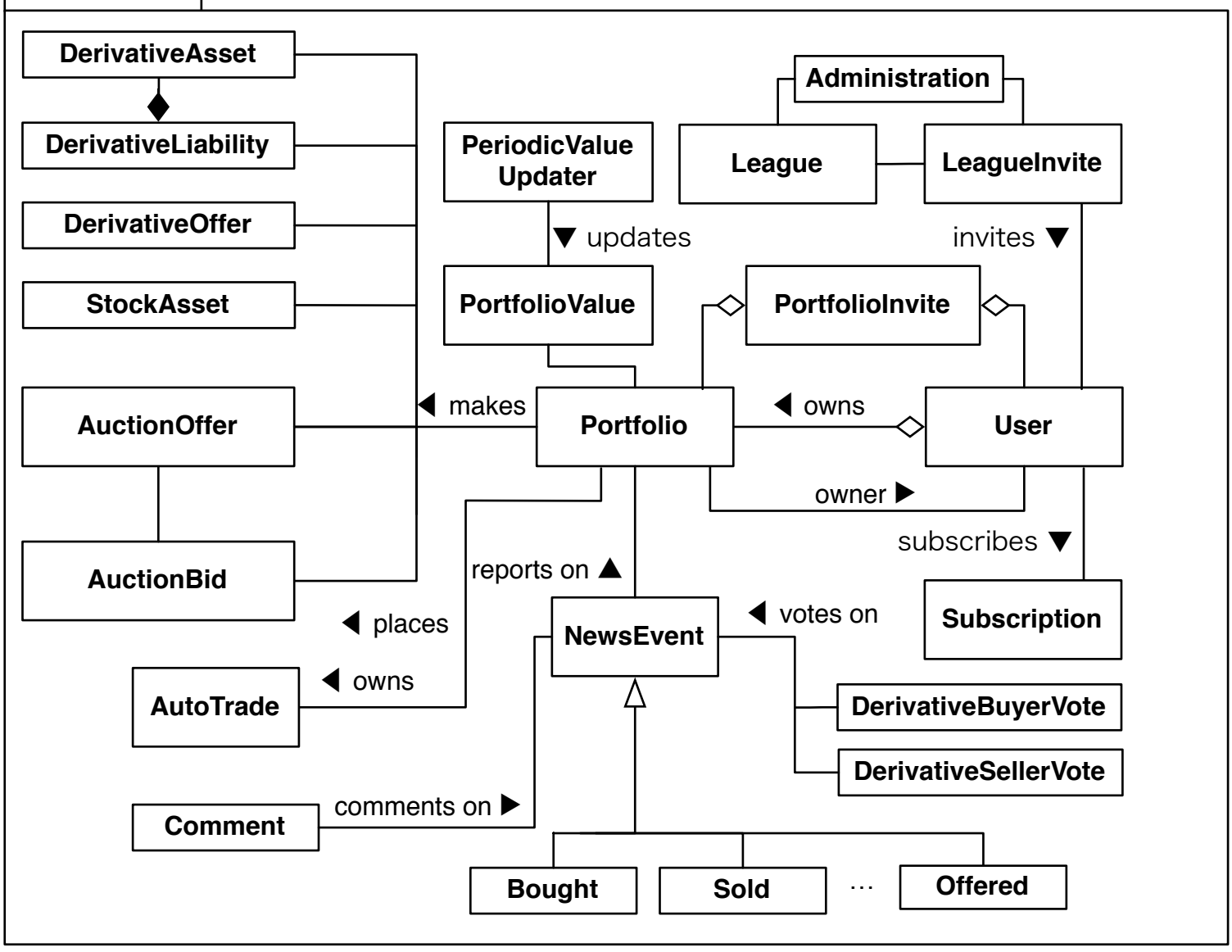
model



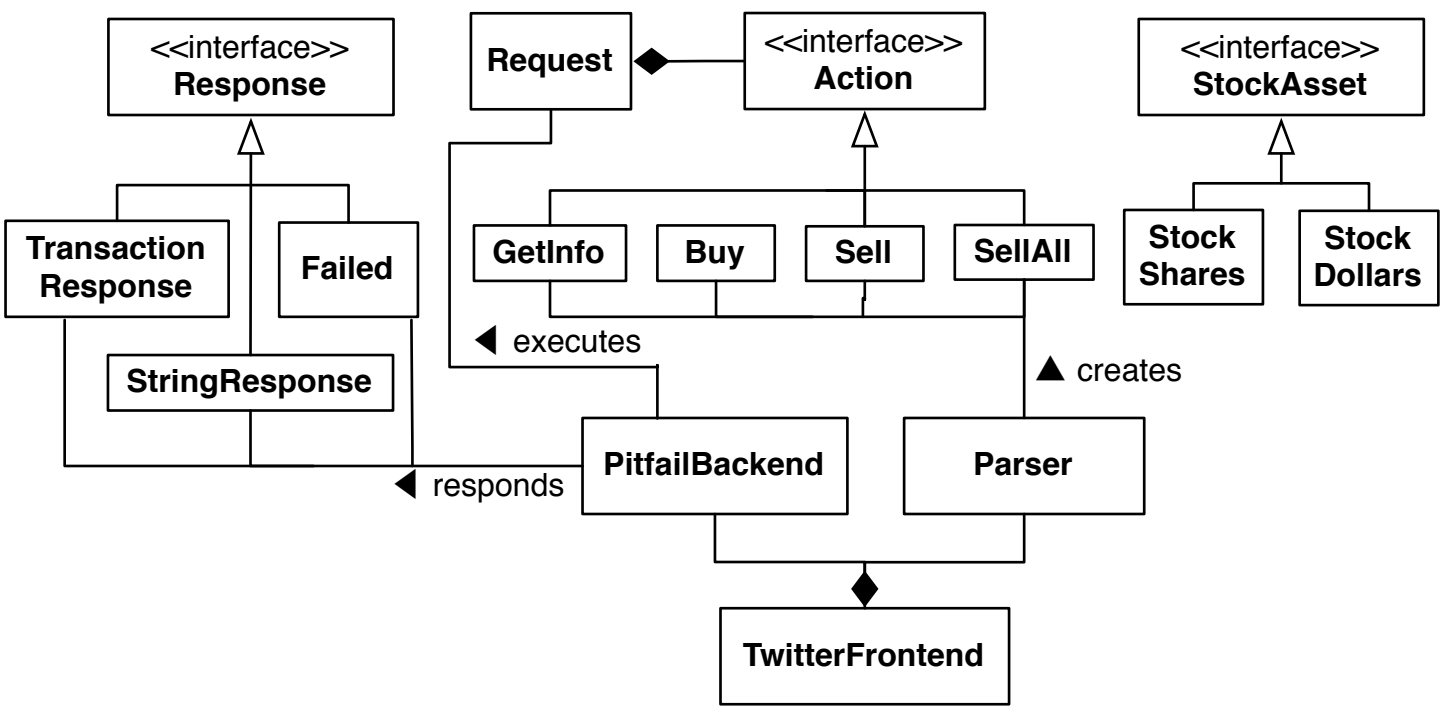
stockdata



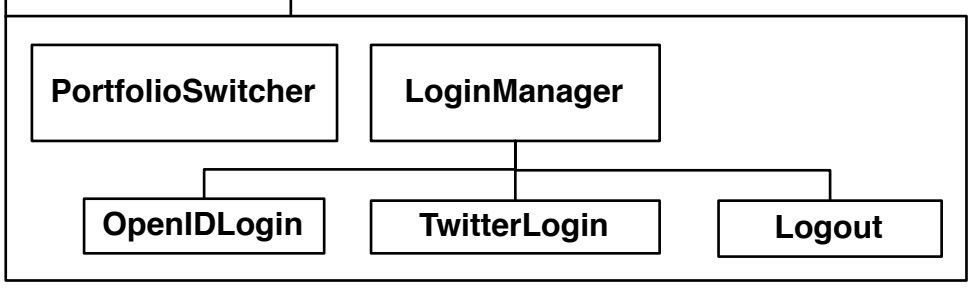
model.schema



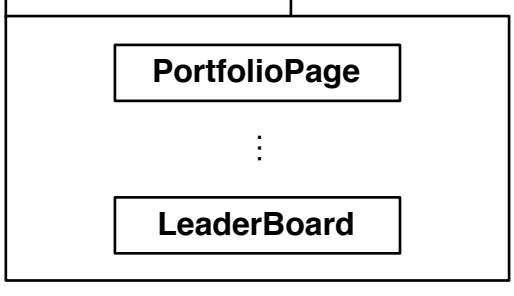
texttrading



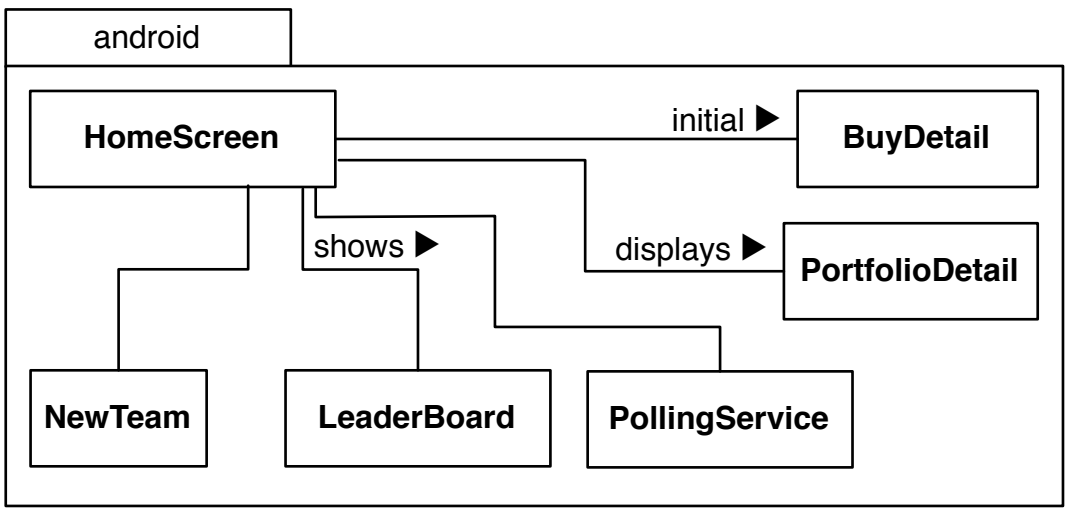
website.control



website.view



android



Shares (model package)

- Attributes
  - +shares: BigDecimal
  - +«operator» -: Dollars
  - +«operator» ###: String
- Methods
  - +«constructor»(BigDecimal)
  - +«constructor»(String>
  - /compare(Shares): Int
  - +«operator» +(Shares): Shares
  - +«operator» -(Shares): Shares
  - +«operator» *(Price): Dollars
  - +«operator» *(Scale): Shares

Price (model package)

- Attributes
  - +price: BigDecimal
  - +«operator» $: String
- Methods
  - +«constructor»(BigDecimal)
  - +«constructor»(String)
  - /compare(Price): Int
  - +«operator» +(Price): Price
  - +«operator» -(Price): Price
  - +«operator» *(Shares): Dollars
  - +«operator» *(Scale): Price

Dollars (model package)

- Attributes
  - +dollars: BigDecimal
  - +«operator» -: Dollars
  - +«operator» $: String
- Methods
  - +«constructor»(BigDecimal)
  - +«constructor»(String)
  - +/compare(Dollars): Int
  - +«operator» +(Dollars): Dollars
  - +«operator» -(Dollars): Dollars
  - +«operator» *(Scale): Dollars
  - +«operator» -/-(Price): Shares
  - +«operator» /-/(Price): Shares

Scale (model package)

- Attributes
  - +price: BigDecimal
  - +«operator» -: Scale
  - +«operator» %: String
- Methods
  - +«constructor»(BigDecimal)
  - +«constructor»(String)
  - /compare(Scale): Int
  - +«operator» +(Scale): Scale
  - +«operator» -(Scale): Scale
  - +«operator» *(Price): Price
  - +«operator» *(Shares): Shares
  - +«operator» *(Scale): Scale

Stock (model.schema package)

- Attributes
  - +symbol: String
  - +toString: String

Quote (model.schema package)

- Attributes
  - +stock: Stock
  - +exchange: String
  - +price: Price
  - +updateTime: ateTime
  - +info: QuoteInfo
  - +toString: String
- Methods
  - /equals(Quote): Boolean

QuoteInfo (model.schema package)

- Attributes
  - +percentChange: Option[BigDecimal]
  - +openPrice: Option[BigDecimal]
  - +lowPrice: Option[BigDecimal]
  - +highPrice: Option[BigDecimal]
  - +dividendShare: Option[BigDecimal]
- Methods
  - /equals(Quote): Boolean

StockDatabase (stockdata package)

- Methods
  - +getQuote(Stock): Quote
  - +getQuotes(Seq[Stock]): Seq[Quote]

YahooYQLStockDatabase (stockdata package)

- Attributes
  - -queryService: HttpQueryService
- Methods
  - +«constructor»(HttpQueryService)
  - +getQuote(Stock): Quote
  - +getQuotes(Seq[Stock]): Seq[Quote]

YahooCSVStockDatabase (stockdata package)

- Attributes
  - -queryService: HttpQueryService
- Methods
  - +«constructor»(HttpQueryService)
  - +getQuote(Stock): Quote
  - +getQuotes(Seq[Stock]): Seq[Quote]

CachingStockDatabase (stockdata package)

- Attributes
  - -database: StockDatabase
  - -cache: Map[Stock, Quote]
- Methods
  - +«constructor»(StockDatabase)
  - +getQuote(Stock): Quote
  - +getQuotes(Seq[Stock]): Seq[Quote]

FailoverStockDatabase (stockdata package)

- Attributes
  - databases: Seq[StockDatabase]
- Methods
  - +«constructor»(Seq[StockDatabase])
  - +getQuote(Stock): Quote
  - +getQuotes(Seq[Stock]): Seq[Quote]

HomeScreen (android package)

- Attributes

    - -activity: Activity
    - -spinner: Spinner
    - -indexOfUserNameForSpinner: Int
    - -search: Button
    - -sell: Button
    - -team: Button
    - -leaderboard: Button
    - -cashText: TextView
    - -ticker: AutoCompleteTextView
    - -companies: String[*] {unique}
- Methods
    - +onCreate(Bundle): Unit
    - +onClick(View): Unit
    - +afterTextChanged(Editable): Unit
    - +beforeTextChanged(CharSequence, Int, Int, Int): Unit
    - +onTextChanged(CharSequence, Int, Int, Int): Unit

BuyDetail (android package)

- Attributes
    - -tickerName: TextView
    - -buy: Button
    - -activity: Activity
    - -tickerString: String
    - -valueString: ArrayList<String>
- Methods
    - +onCreate(Bundle): Unit
    - +ImageOperations(String, String): Drawable
    - +fetch(String): Object

PortfolioDetail (android package)

- Attributes
    - -cashText: TextView
    - -portfolioHeader: TextView
    - -selectedPortfolio: String
- Methods
    - onCreate(Bundle): Unit

NewTeam (android package)

- Attributes
    - -portfolio: EditText
    - -invite: EditText
    - -inviteButton: Button
    - -activity: Activity
- Methods

- +onCreate(Bundle): Unit
- +onClick(View): Unit

LeaderBoard (android package)

- Methods
  - onCreate(Bundle): Unit

PollingService (android package)

- Attributes
  - TAG: String { readonly }
  - timer: Timer
  - update_id: Int
- Methods
  - onBind(Intent): IBinder
  - onCreate(): Unit
  - onDestroy(): Unit

14 History of Work & Current Status of Implemenation

14.1 Comparison to Planned Milestones

The planned milestones from Report 2 differed from reality in that they were overly aggressive and did not take into account that quickness that Pitfail team members could implement certain functions. When creating the planned deadlines in Report 2, team members assumed working two to four hours a day on Pitfail. What happened is that other responsibilities in other classes resulted in stretches of inactivity in Pitfail, thus throwing up the planned deadlines. As the Demo 2 day approached, great amounts of time during the day and night were put into Pitfail in a way that Microsoft Project could not correctly capture a "typical Pitfail working day." The result is a History of Work heavily concentrated around Demo days. If Pitfail were a company, Report 2's Plan of Work would have been a great guiding factor in agile development. Instead, Report 3's History of Work better explains how milestones were achieved.

The History of Work shows the milestones that were not accomplished as tasks that are corssed out. The various non-accomplished were not accomplished either because their predecessors were not accomplished, the milestones were minor goals if time permitted and time ran out, or the milestones were no longer deemed necassary:

1. The support for comlplex actions (orders, derivatives) was not implemented because the need for free-form Twitter input seemed unnecessary. The structured Twitter input was easily understandable, but without an upgrade to an unstructured Twitter input recongizer, advanced actions would not be easily understood in the structured Twitter system. Hence, advanced support for Twitter was not implemented.

2. Challeges was not implemented because the teams and leagues were delievered very close to Demo 2. Implemented challenges would have been a trade-off between itself and debugging and debugging was deemed more important.

3. Implemented OpenID for Facebook and Google was deemed not necassary since Twitter offered a similar service that was already implemented.

14.2 Key Accomplishments

The following are the key accomplishments of the Pitfail project that were implemented split across the platforms they were implemented on and the different use cases that were implemented:

- Multiple Interface
  - Website
  - Android
  - Twitter
  - Facebook
  - Email
- Use Cases
  - Stocks - Buy/Sell
  - Option for Orders
  - Derivatives
  - Auctions
  - Portfolio Graphs
  - Auto Trades
  - Comments
  - Voting
  - Teams - cooperative
  - Leagues - competitive
  - Leaderboard

15 Conclusions and Future Work

15.1 What goals of PitFail are still unmet?

What are PitFail's goals?

Definite interfaces for interactions between subsystems have yet to be clearly defined. This lack of a standard interface bettween 'things' leads to duplication of effort within the codebase. In particular, the model interface is extremely adhoc. Their exists one abstraction of the model within the texttrading/ code (refered to there as "backend"). The website/view/ does database lookups nearly directly. Java servlets are implimented via a set of compatability methods within the model/ codebase. A unified interface model interface has the posibility to provide a clear and simple manner for adding additional components to the codebase, where each of these compenents interacts with the model in some way.

The external interfaces exposed by the Java servlets (utilized by the Android and Facebook applications) lack authorization mechanizms and are thus unsuitable as web service APIs.

15.2 Which areas of the system would we focus on to meet PitFail's goals?

While the website holds most of PitFail's complicated features, the "light" frontends are much more convenient, simple, and intuitive. The Twitter frontend has a particularly nice syntax and requires practically no user effort to get started.

If the website were demoted to just displaying the "content heavy" parts (e.g. plots) and the Twitter and mobile frontends became the main mode of interaction, this would allow us to focus on the unique aspects of PitFail (derivative trading, PitFail-style voting).

16 References

[ADTs] Marie Gleichman. "Functional Scala: Algebraic Datatypes – Sum and Product Types“ http://gleichmann.wordpress.com/2011/02/05/functional-scala-algebraic-datatypes-sum-and-product-types/

[American] Investopedia. ”How do you tell whether an option is American or European style¿‘ http://www.investopedia.com/ask/answers/06/americanvseuropean.asp#axzz1gFsL9Mp8

[Anemic] StackOverflow ”Anemic Domain Model: Pros/Cons“ http://stackoverflow.com/questions/258534/anemic-domain-model-pros-cons

[Applicative1] Haskell Wikibook - Applicative Functors. http://en.wikibooks.org/wiki/Haskell/Applicative_Functors

[Ask] Investopedia - Ask price http://www.investopedia.com/terms/a/ask.asp

[Bid] Investopedia - Bid price http://www.investopedia.com/terms/b/bidprice.asp#axzz1gTt8rHSo

[Browse] Mark Harrah’s Browse Plugin https://github.com/harrah/browse

[CAMC] Steven Counsell, Stephen Swift. ”The Interpretation and Utility of Three Cohesion Metrics for Object-Oriented Design“. ACM Trans. Softw. Eng. Methodol. 15, 2 (April 2006), 123-149. DOI=10.1145/1131421.1131422 http://doi.acm.org/10.1145/1131421.1131422

[CC] Challa Bonja and Eyob Kidanmariam. 2006. Metrics for class cohesion and similarity between methods. In Proceedings of the 44th annual Southeast regional conference (ACM-SE 44). ACM, New York, NY, USA, 91-95. DOI=10.1145/1185448.1185469 http://doi.acm.org/10.1145/1185448.1185469

[Controllers] Paul Oldfield. ”Domain Modelling“ http://www.aptprocess.com/whitepapers/DomainModelling.pdf

[CurryHoward] Haskell Wikibook - The Curry-Howard Isomorphism. http://en.wikibooks.org/wiki/Haskell/The_Curry-Howard_isomorphism

[Currying] HaskellWiki - Currying. http://www.haskell.org/haskellwiki/Currying

[Data] Haskell Wikibook - Type Declarations. http://en.wikibooks.org/wiki/Haskell/Type_declarations

[DRY] Ward's Wiki - Don't Repeat Yourself. http://c2.com/cgi/wiki?DontRepeatYourself

[DSL] Ward's Wiki - Domain Specific Language. http://c2.com/cgi/wiki?DomainSpecificLanguage Ed. Eric McLaughlin and Mary O'Brien. Sebastopol: O'Reilly, 2006.

[Erasure] Oracle. "Type Erasure". The Java Tutorials. http://docs.oracle.com/javase/tutorial/java/generics/erasure.html

[H2] H2 Database Engine. http://www.h2database.com/html/main.html

[HList] Mark Harrah. "Type Level Programming in Scala". http://apocalisp.wordpress.com/2010/06/08/type-level-programming-in-scala/

[HTTP] Wikipedia. "Hypertext Transfer Protocol". http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

[Implicits] Martin Odersky. "Poor Man's Type Classes". http://lampwww.epfl.ch/~odersky/talks/wg2.8-boston06.pdf

[Inversion] Martin Fowler. "Inversion of Control". http://martinfowler.com/bliki/InversionOfControl.html

[Iry1] James Iry. "Why Scala's Option and Haskell's Maybe types will save you from null". http://james-iry.blogspot.com/2010/08/why-scalas-and-haskells-types-will-save.html

[Iry2] James Iry. "Getting to the bottom of nothing at all". http://james-iry.blogspot.com/2009/08/getting-to-bottom-of-nothing-at-all.html

[JDBC] Wikipedia. "Java Database Connectivity". http://en.wikipedia.org/wiki/Java_Database_Connectivity

[Jetty1] Jetty Web Server. http://jetty.codehaus.org/jetty/

[Kiselyov] Oleg Kiselyov and Ralf Lämmel and Keean Schupke. "Strongly typed heterogeneous collections". Haskell 2004: Proceedings of the ACM Sigplan workshop on Haskell.

[Lambda] "A Tour Of Scala: Anonymous Function Syntax". http://www.scala-lang.org/node/133

[Lift1] Lift Web Framework. http://liftweb.net/

[Lift2] Lift Forms. http://exploring.liftweb.net/master/index-6.html

[Limit] Investopedia - Limit Order http://www.investopedia.com/terms/l/limitorder.asp

[Loop] StackOverflow. "How to iterate through a heterogeneous recursive value in Haskell". http://stackoverflow.com/questions/5024148/how-to-iterate-through-a-heterogeneous-recursive-value-in-haskell

[LSCC] J Al Dallal, Lionel C. Briand. "A Precise Method-Method Interaction-Based Cohesion Metric for Object-Oriented Classes". ACM Transactions on Software 2010.

[Makers] Wikipedia. "Market Maker". http://en.wikipedia.org/wiki/Market_maker

[Marsic] Marsic, Ivan. *Software Engineering*. Piscataway: Rutgers University, 2011. PDF.

[ML] The Standard ML Basis Library - The Option Structure. http://www.standardml.org/Basis/option.html

[Monads1] Burak Emir. "Monads in Scala". http://lamp.epfl.ch/~emir/bqbase/2005/01/20/monad.html

[MVC] Wikipedia. "MVC". http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

[Option1] Scala Standard Library - Option. http://www.scala-lang.org/api/current/scala/Option.html

[Pollak] David Pollak. "Separating Presentation Logic from scala files". http://markmail.org/message/cco7biz2g3jeilg6

[Scalaz] Scalaz Libarry. http://code.google.com/p/scalaz/

[SCOM] Luis Fernández and Rosalía Peña. "A Sensitive Metric of Class Cohesion". Information Theories and Applications.

[SICP1] Harold Abelson, Gerald Sussman, Julie Sussman. "The Structure and Interpretation of Computer Programs". http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-20.html#%_sec_3.1.1

[SideEffects] Ward's Wiki - Side Effect. http://c2.com/cgi/wiki?SideEffect

[Squeryl1] Squeryl. http://squeryl.org/

[Squeryl2] Squeryl source code, showing where it fails at inner classes. https://github.com/max-l/Squeryl/blob/master/src/main/scala/org/squeryl/internals/PosoMetaData.scala

[Stop] Investopedia - Stop Order. http://www.investopedia.com/terms/s/stoporder.asp#axzz1g4pXxPbD

[Struct] Steven Schmidt. "Scala Goodness: Structural Typing". http://codemonkeyism.com/scala-goodness-structural-typing/

[Traits] "A Tour Of Scala: Traits". http://www.scala-lang.org/node/126

[Typing] Haskell Wiki - Typing. http://www.haskell.org/haskellwiki/Typing

[UML] Miles, Russ and Kim Hamilton. *Learning UML 2.0*.

[Unit] Wikipedia - "Unit Type". http://en.wikipedia.org/wiki/Unit_type

[View] Lift - "View First". http://www.assembla.com/wiki/show/liftweb/View_First

[XML] "A Tour of Scala: XML Processing". http://www.scala-lang.org/node/131

[Android] ”Developers Guide“. http://developer.android.com/guide/index.html