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

//TODO

Responsibility	**Michal Koval**	**Cody Schafer**	**Owen Healy**	**Brian Goodacre**	**Roma Mehta**	**Sonu Iqbal**	**Avanti Kulkarni**
Customer Reqs. (6)							100%
Glossary of Terms (4)	%	10%	10%	10%	10%	10%	10%
Functional Reqs.							
? Stakeholders (2)		100%					
? Actors (2)		100%					
? Goals (4)	%	50%					
? Casual UC (5)		100%					
? Dressed UC (11)	%	20%		40%			
? UC Diagram (4)		100%					
? UC Tracability	%						
Seq. Diagrams (9)						100%	
Nonfunc. Reqs. (6)						100%	
Domain Analysis							
? Concepts (12)			100%				
? Associations (4)			100%				
? Attributes (3)			100%				
Contracts (6)					100%		
User Interface (8)	%						
Plan of Work (3)				100%			
References (1)	14%	14%	14%	14%	14%	15%	14%

2 General Information

2.1 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.

2.2 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.

2.2.1 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.

2.2.2 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).

2.2.3 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] [Option2]. So for example:

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

2.2.4 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).

2.2.5 Applicative Functors

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

2.2.6 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.

2.2.7 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.

3 Glossary

4 Architecture

4.1 Overall Architecture

PitFail follows roughly an MVC [MVC] architecture. David Pollak 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.

4.1.1 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/`.

4.1.1.1 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.

4.1.2 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 ??????
- Facebook view in ??????

4.1.3 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

4.1.4 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 function programming features.
- Scalatest for unit testing.
- Java Servlet API for servlets.
- GSON for JSON serialization.
- Scribe for OAuth protocol.

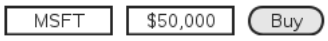
4.2 The Website

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).

4.2.1 Overall Website Architecture

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

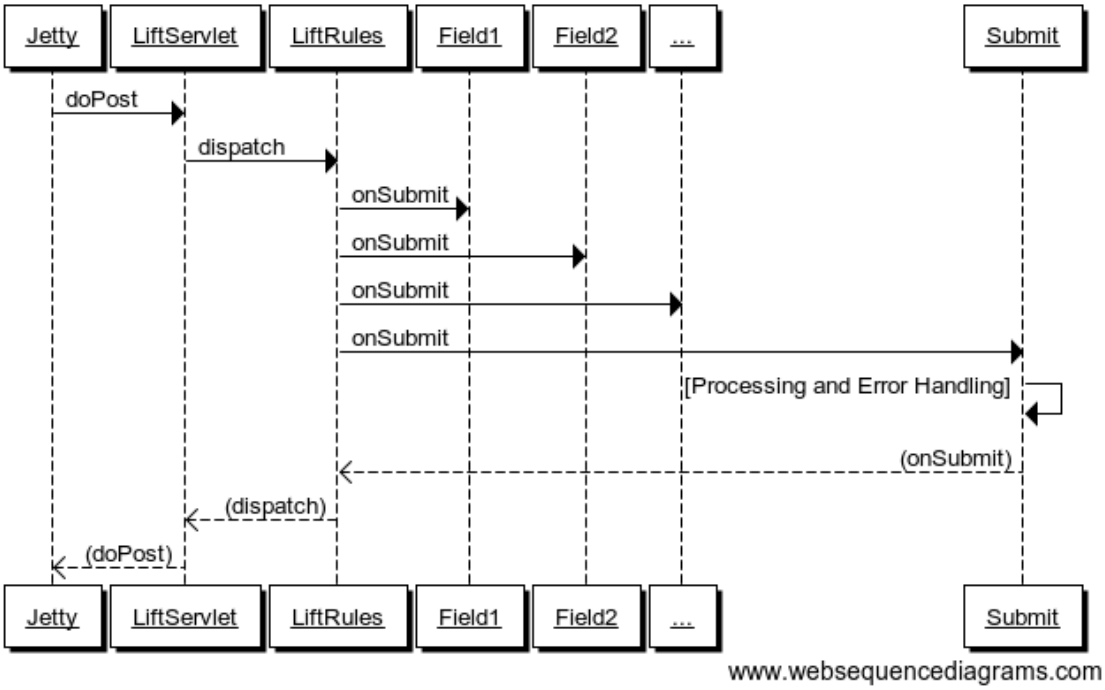
4.2.1.1 Performing actions (Buy/Sell/...) via the Web frontend Suppose the user has filled out a form like this one:



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.



4.2.1.2 When Lift gets an HTTP POST 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.

4.2.1.3 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:

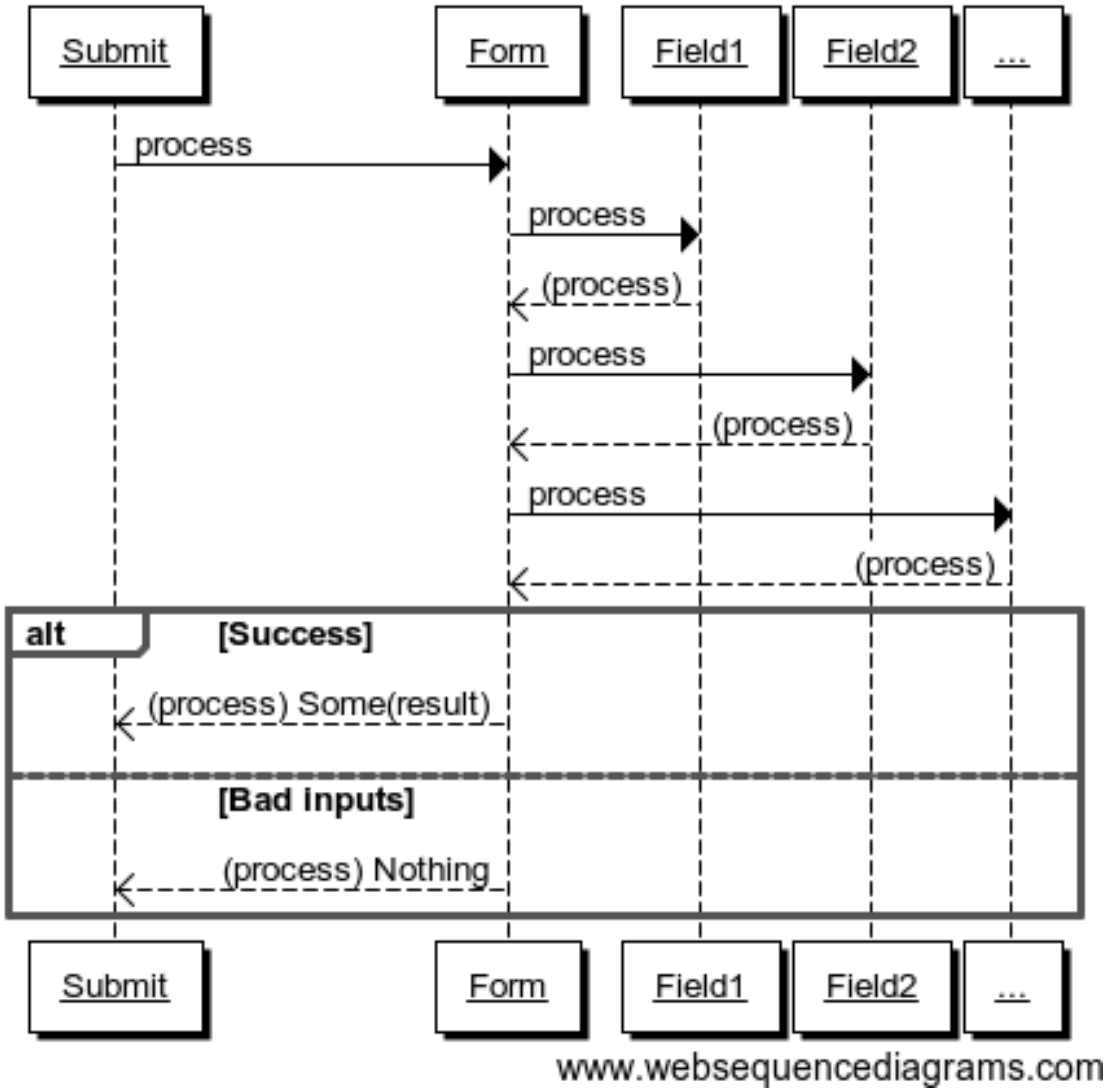
1. Convert the user's loosely structured input into a strongly-typed internal representation (example website/view/ModelFields.scala ref_717).

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:



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:

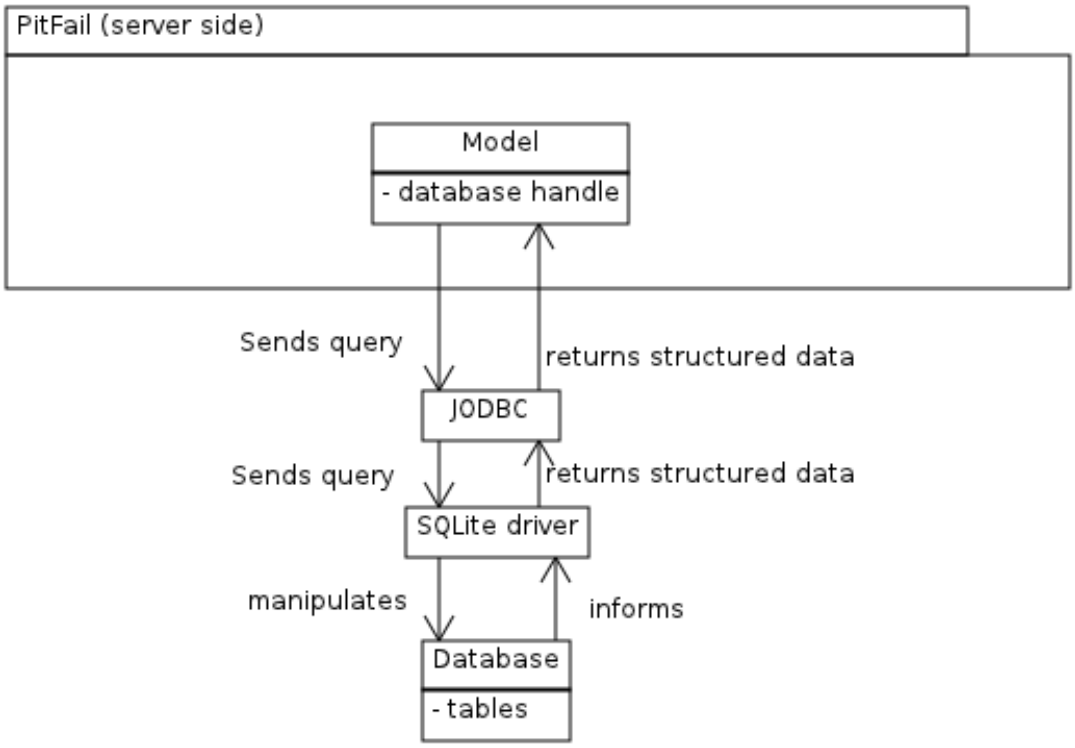
1. The Model (`Portfolio`, `StockAsset`, ...) do not have to duplicate the checks made in the view. For example, the model never needs to check that a string is formatted correctly like a number.

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 Domain Model

5.1 How the Domain Model Has Changed

The original domain model diagram looked like this:



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.

5.2 Users, Portfolios, and Leagues

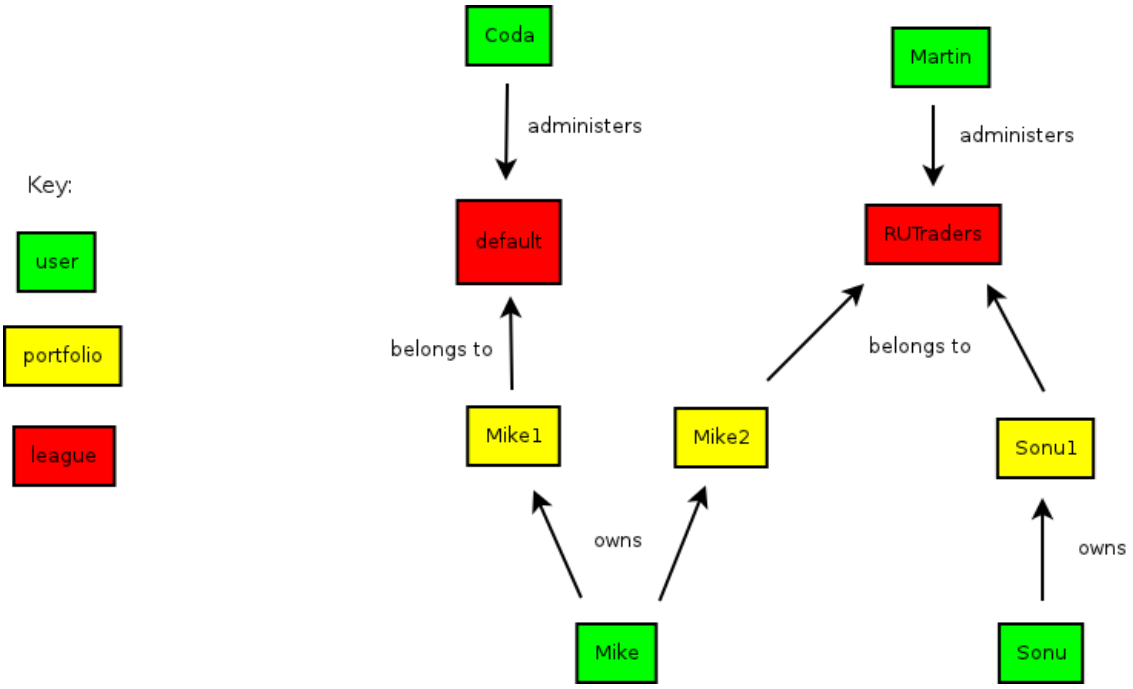
5.2.1 Basic Definitions

- "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:



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:

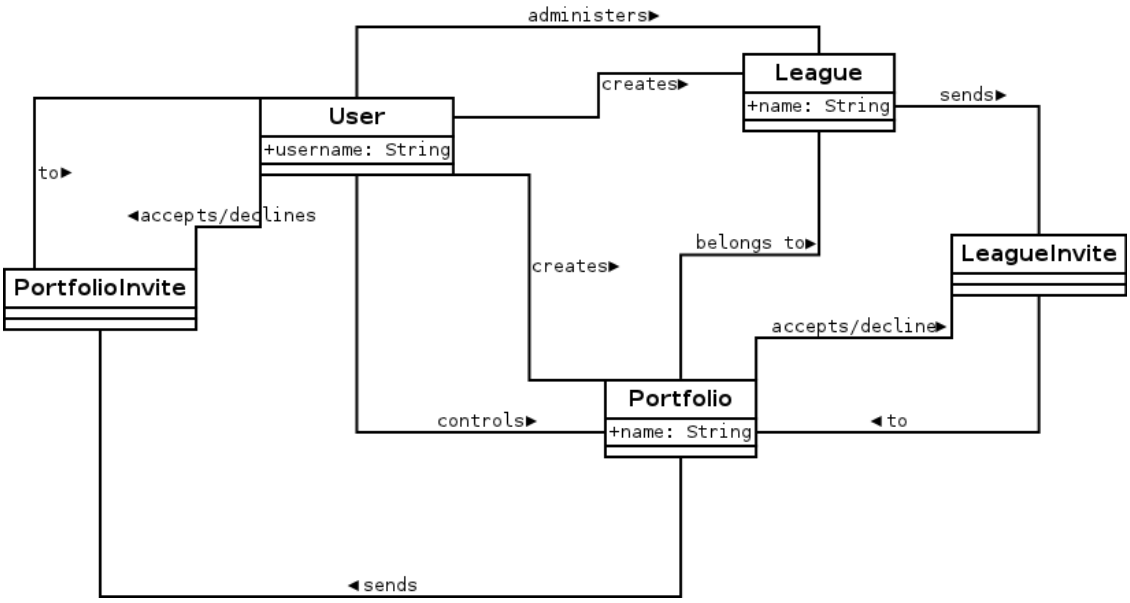
- "User" -- This provides a way for an actual human user to log into the site, to have an experience that is tied to them.
- "Portfolio" -- These actually do the trading. A Portfolio is the one actually credited with owning assets and being responsible for the payment of liabilities, *not* the user.
- "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".

5.2.2 The User-Portfolio-League domain model

The basic concepts and relationships for the idle system are:

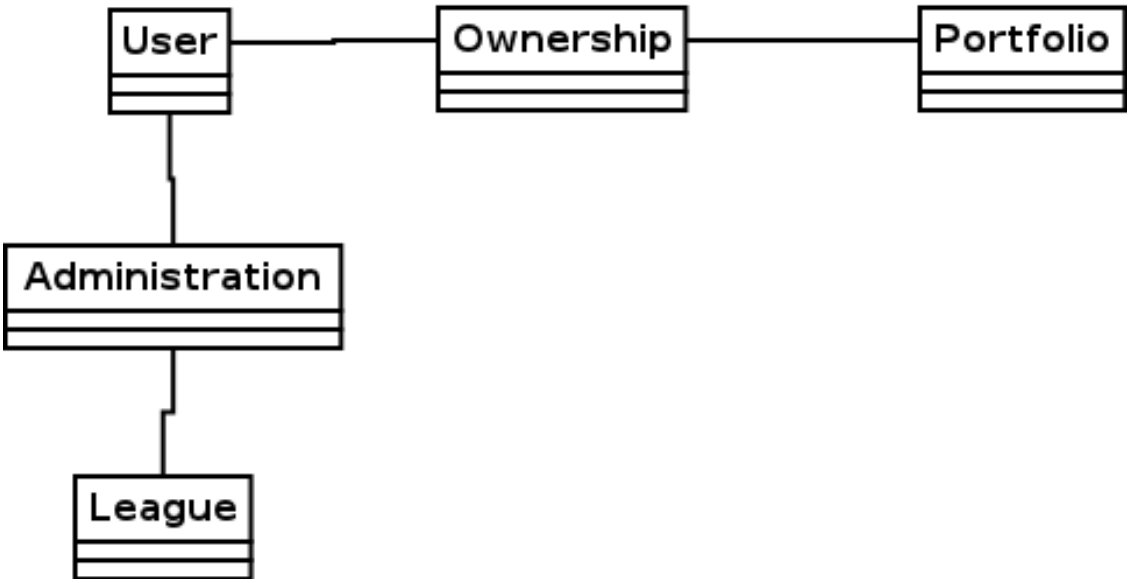
Adding some of the creation/joining operations, this becomes:



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):

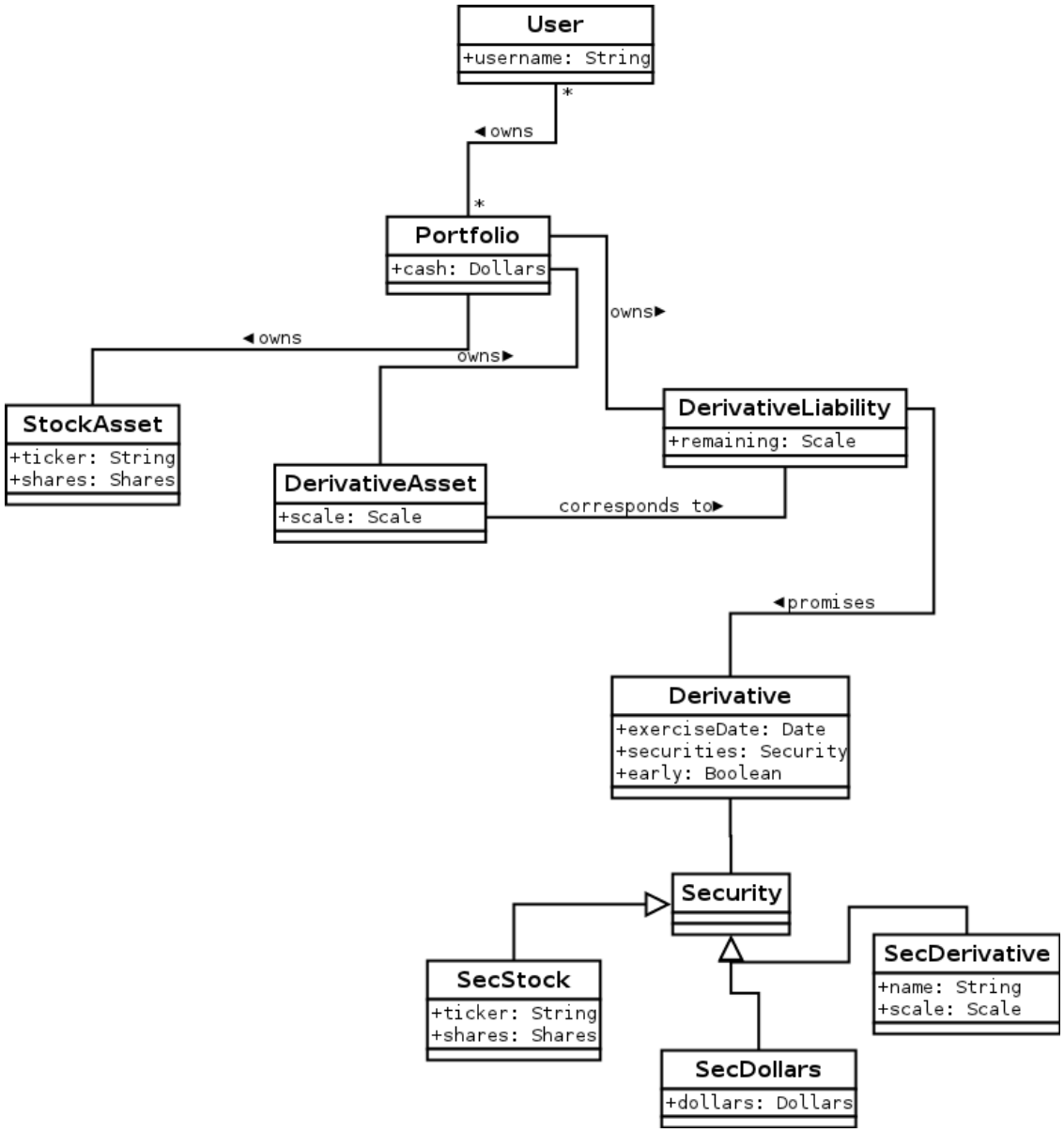


But this is a detail of the implementation and not part of the domain model; no meaningful attributes are stored with Ownership and Administration.

5.3 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:



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

5.3.1 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.

5.3.2 How Derivative Assets/Liabilities work

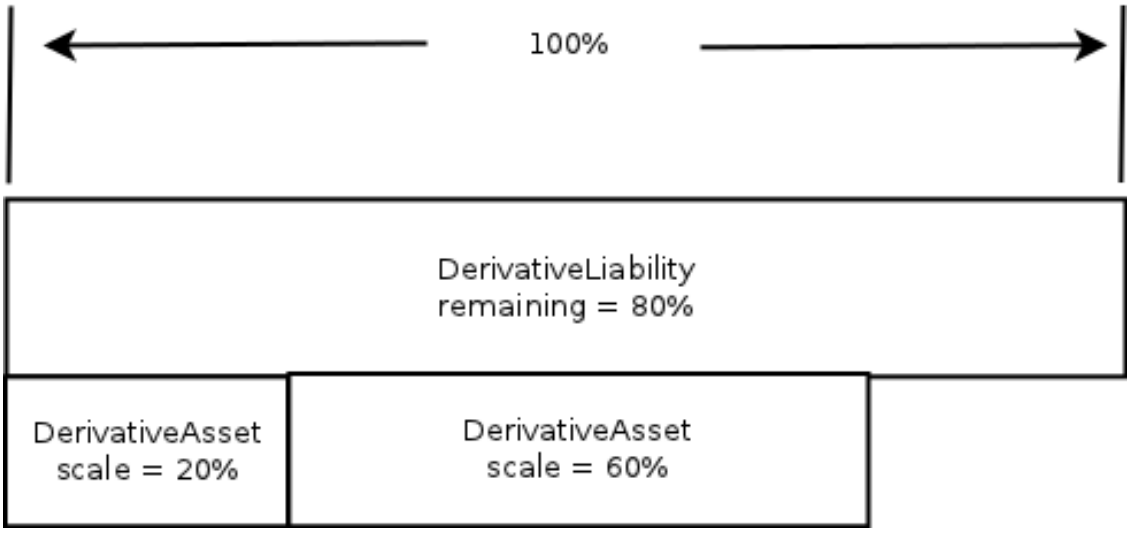
A derivative, in PitFail, is a promise to exchange a list of assets on or before a specified date. There are 3 parts to this contract:

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:



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`.

5.4 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:

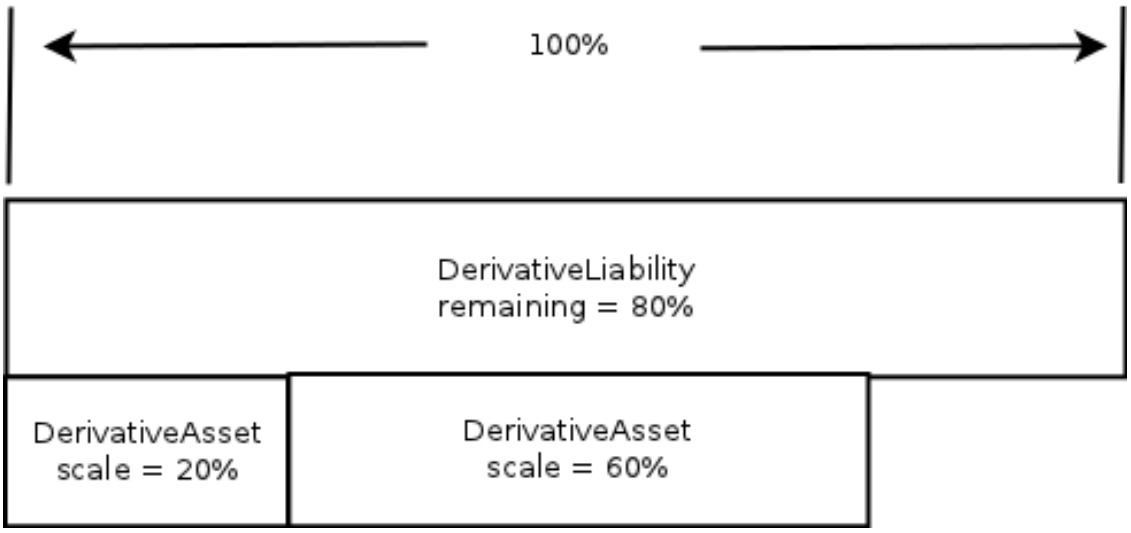
1. A "stock" security, `SecStock`, which holds a ticker symbol and a number of shares.
2. A "dollars" security, `SecDollar`, which holds a dollar amount.
3. A "derivative" security, `SecDerivative`, which holds a named liability and a scale (see the section on Scaling Derivatives). (At the moment there 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.

5.4.1 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":



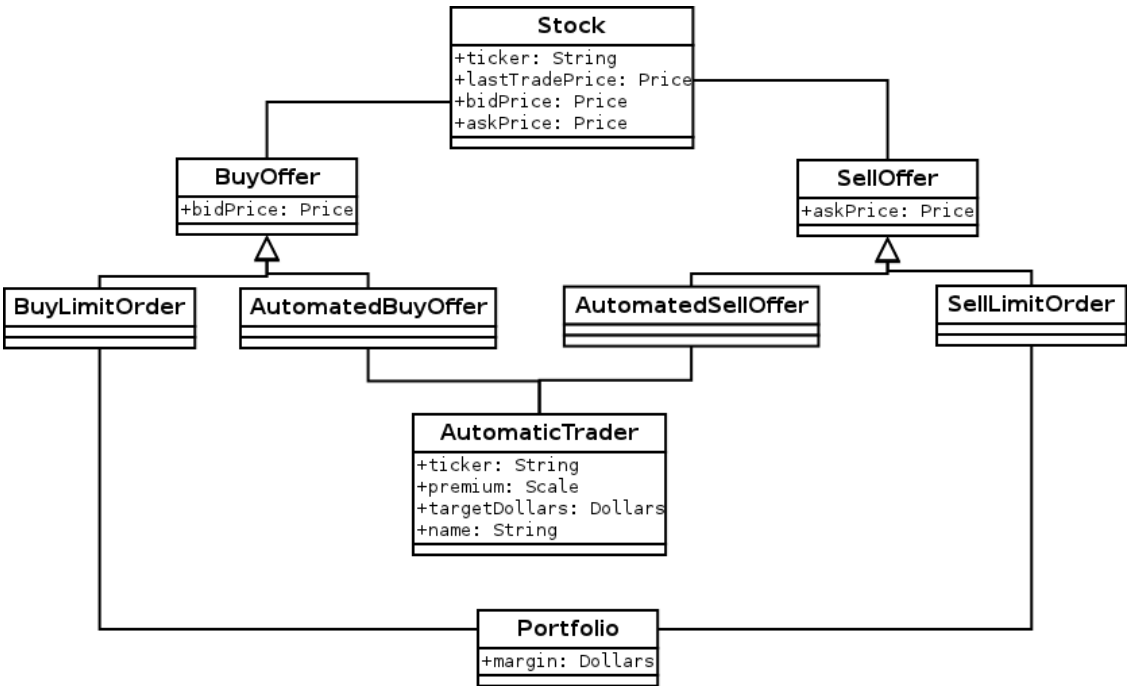
Scaling is done by scaling each security promised:

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.

5.5 Trading Stocks

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



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. Autamated (synthetic) trading orders.

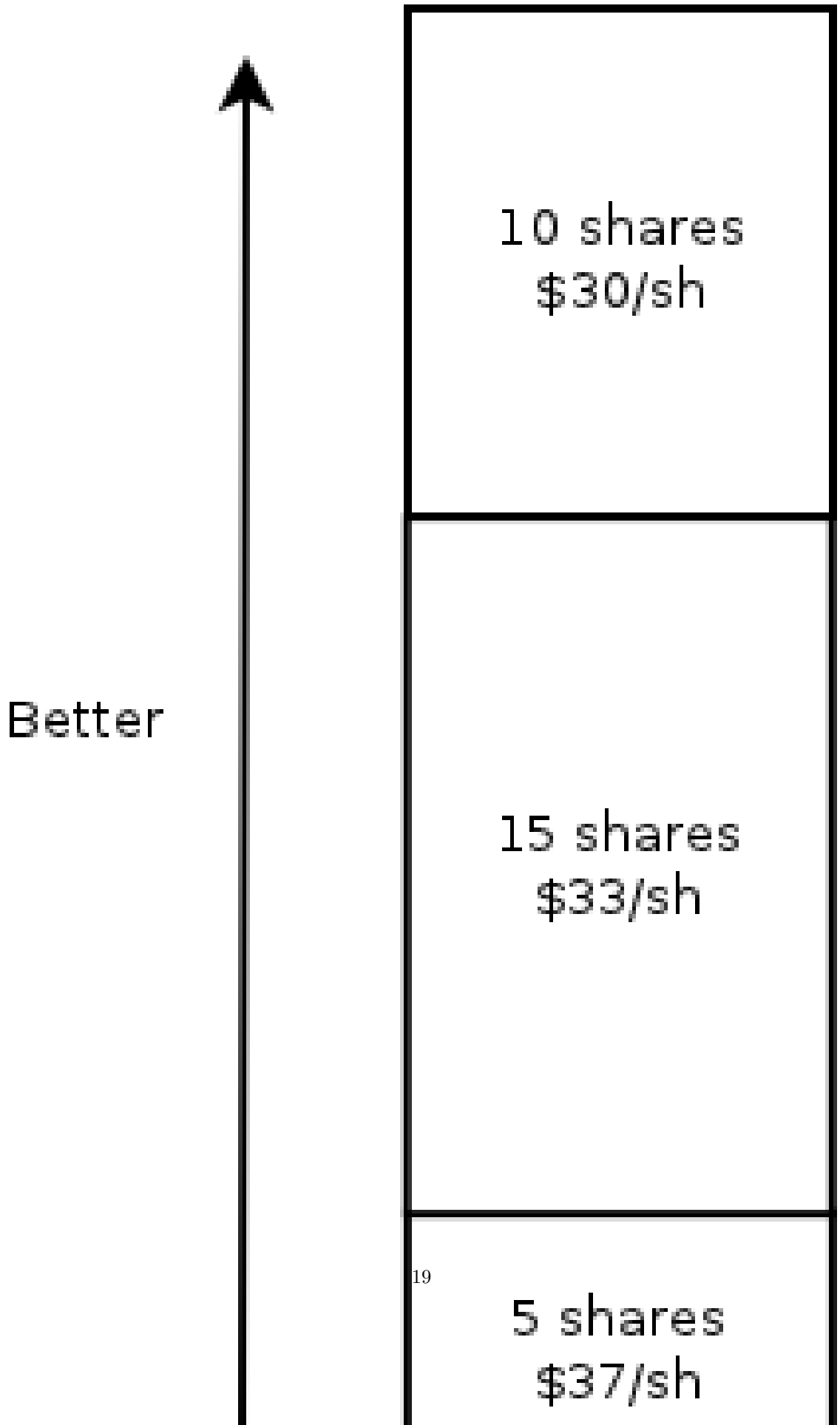
Market orders do not exist when the system is idle because market orders are executed at the offering price as soon as they are created. PitFail does not provide explicit support for stop orders, but it would be easy for a user to create one using the javascript automated trading API (and, when a Stop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

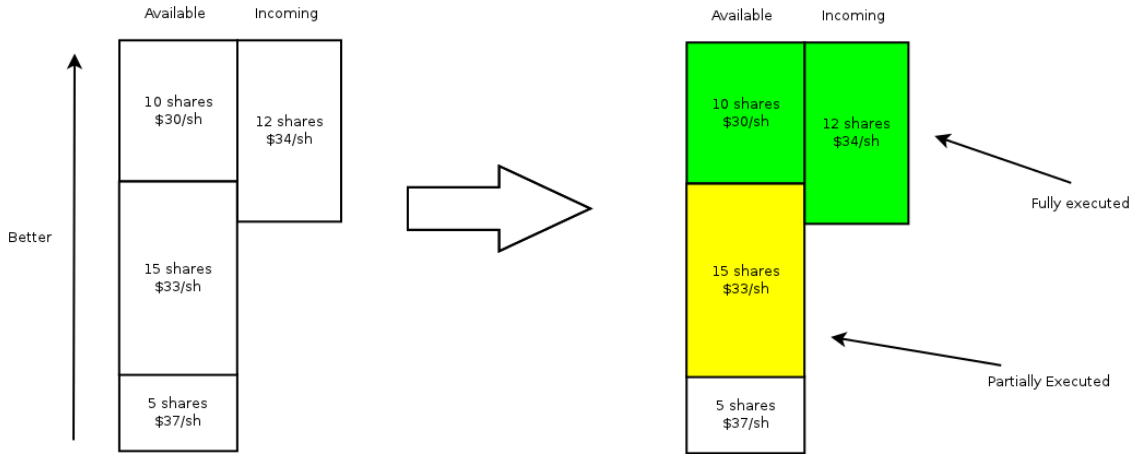
An invariant of the system is that when the order system is Idle, there are no orders that can be matched with one another.

5.5.1 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):


Better
10 shares
$30/sh
15 shares
$33/sh
5 shares
$37/sh

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:



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.

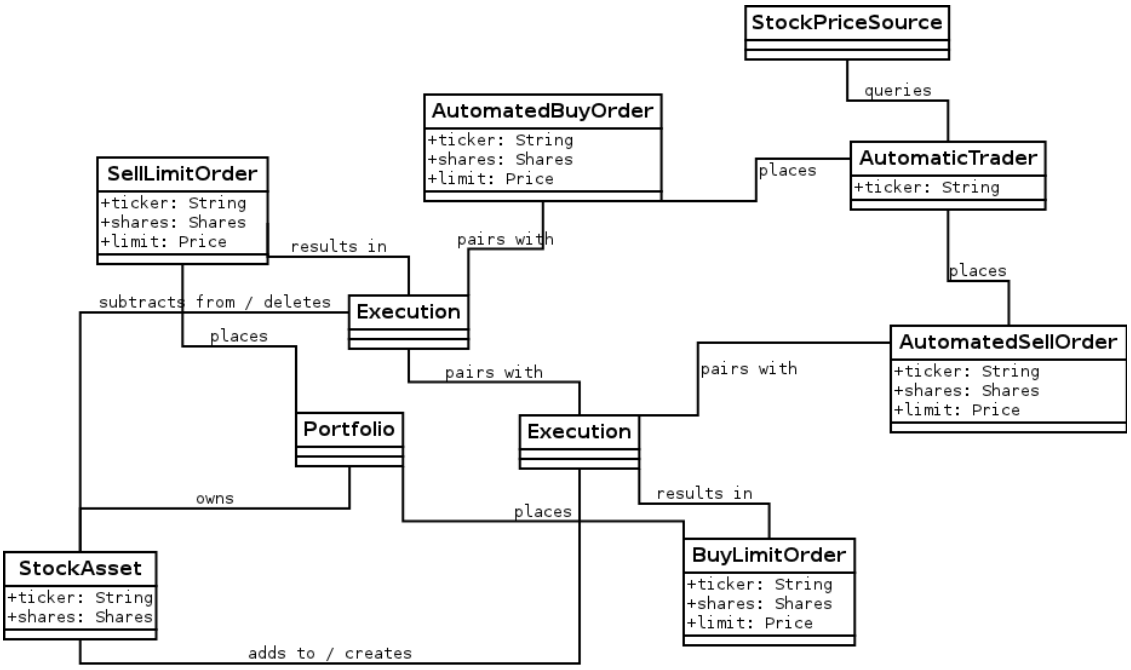
5.5.2 Margin

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.

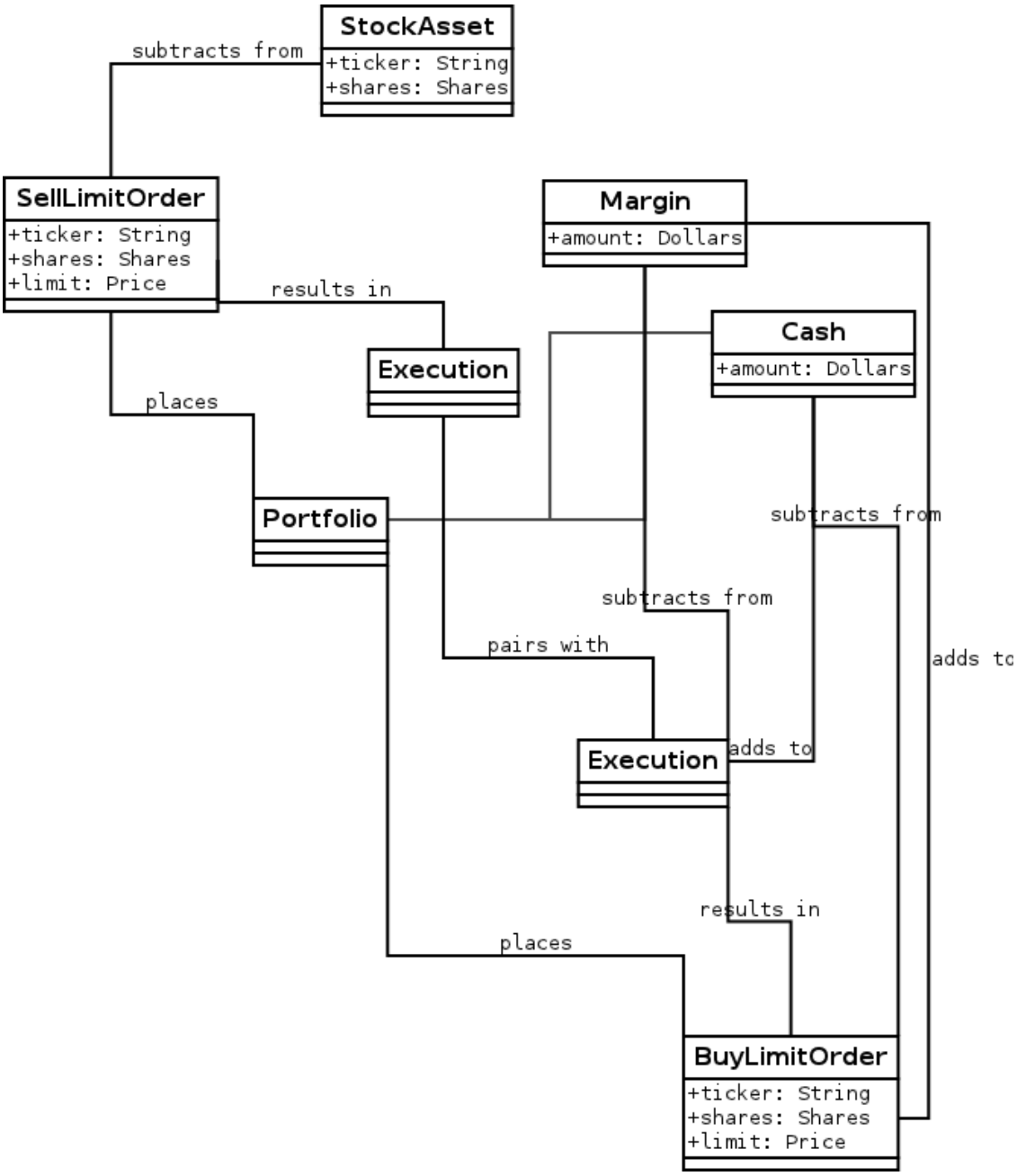
5.5.3 Domain model for trading

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.



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.

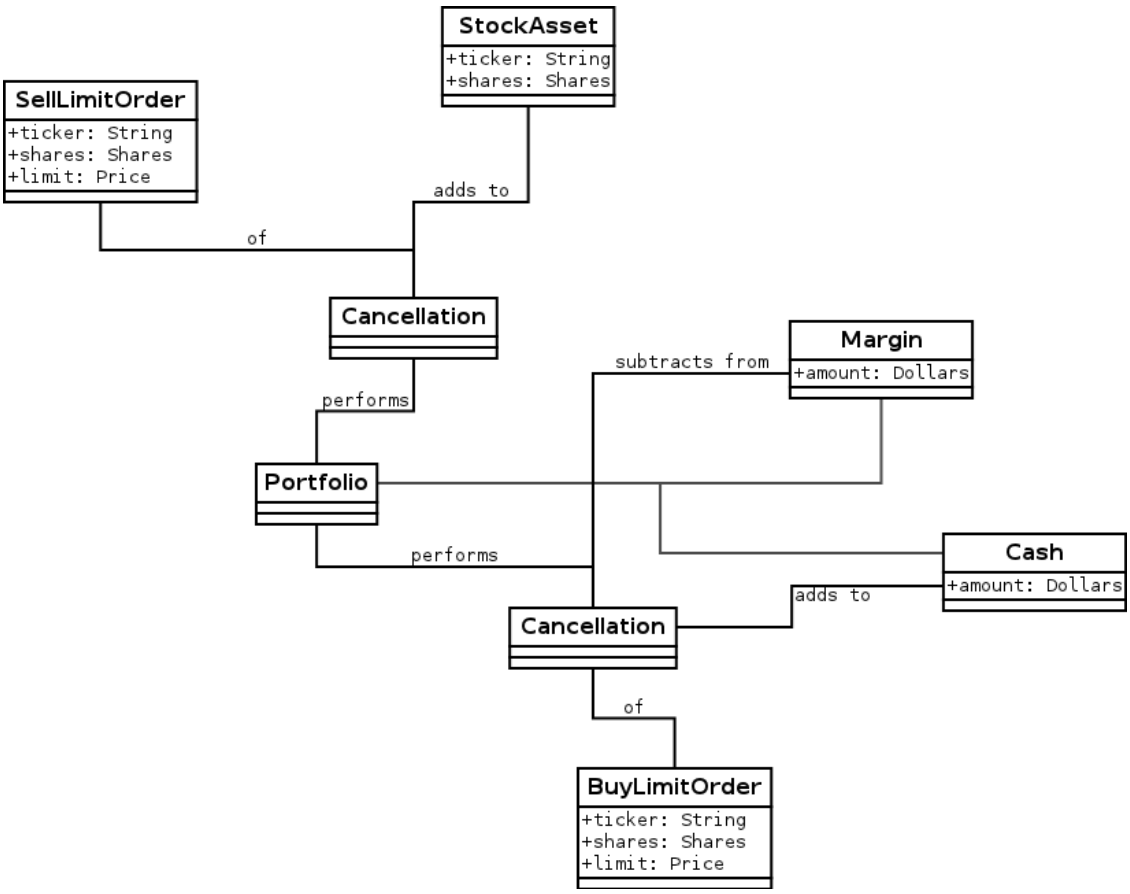
Because there is too much to fit on one diagram, here is the part of the domain model that deals with cash and margin:



(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:



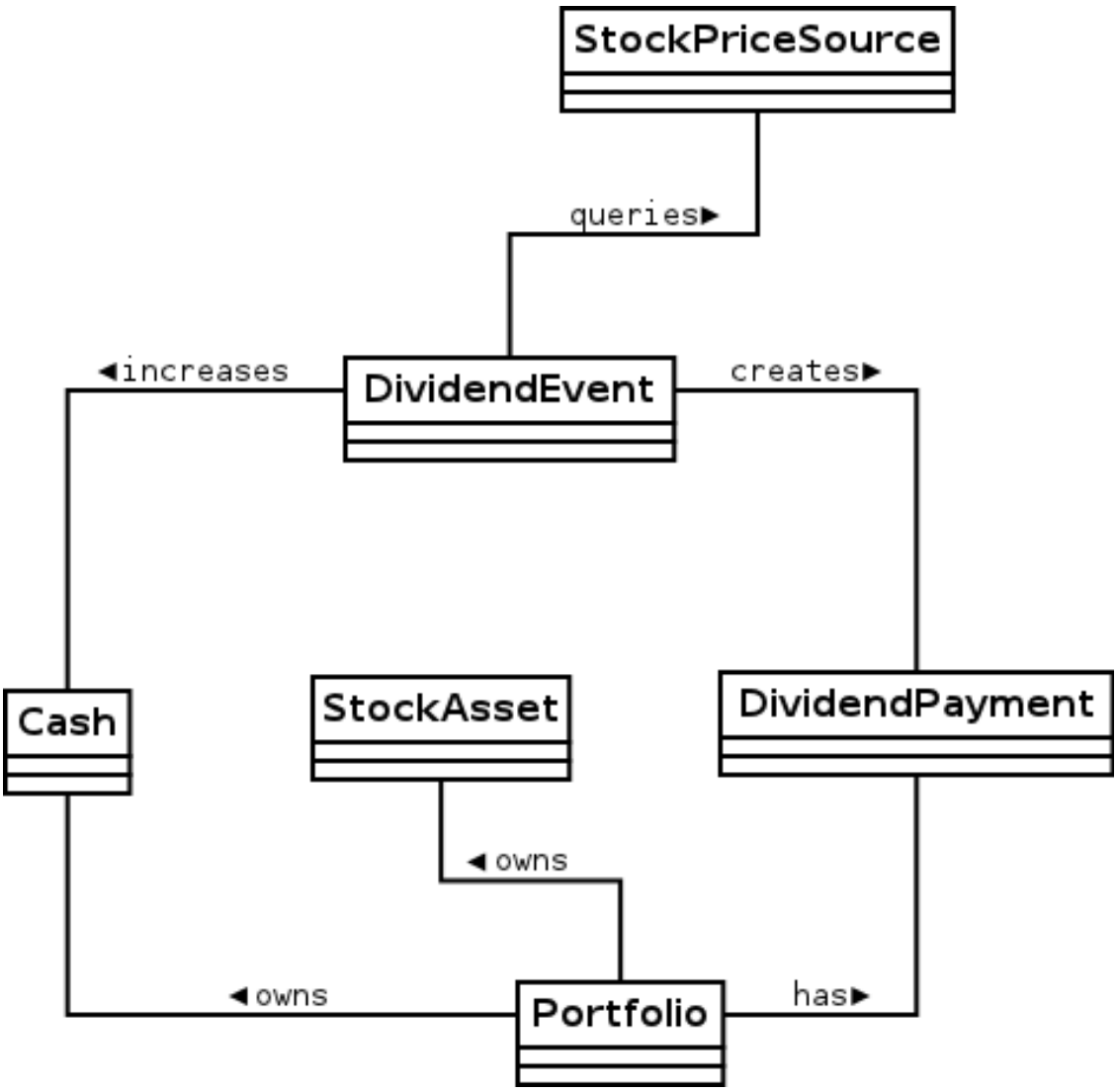
5.6 Dividends

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`:



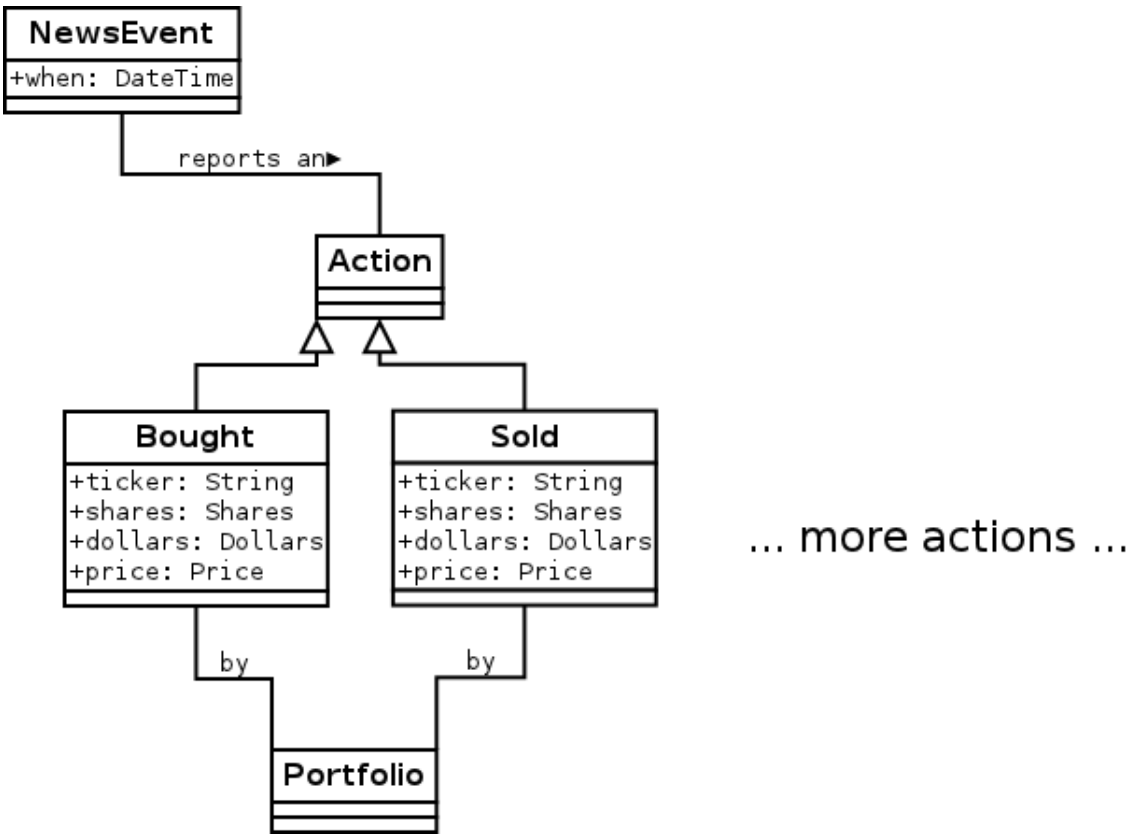
The `DividendPayment` object is created only to allow the user to view the history of their dividend payments.

5.7 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.

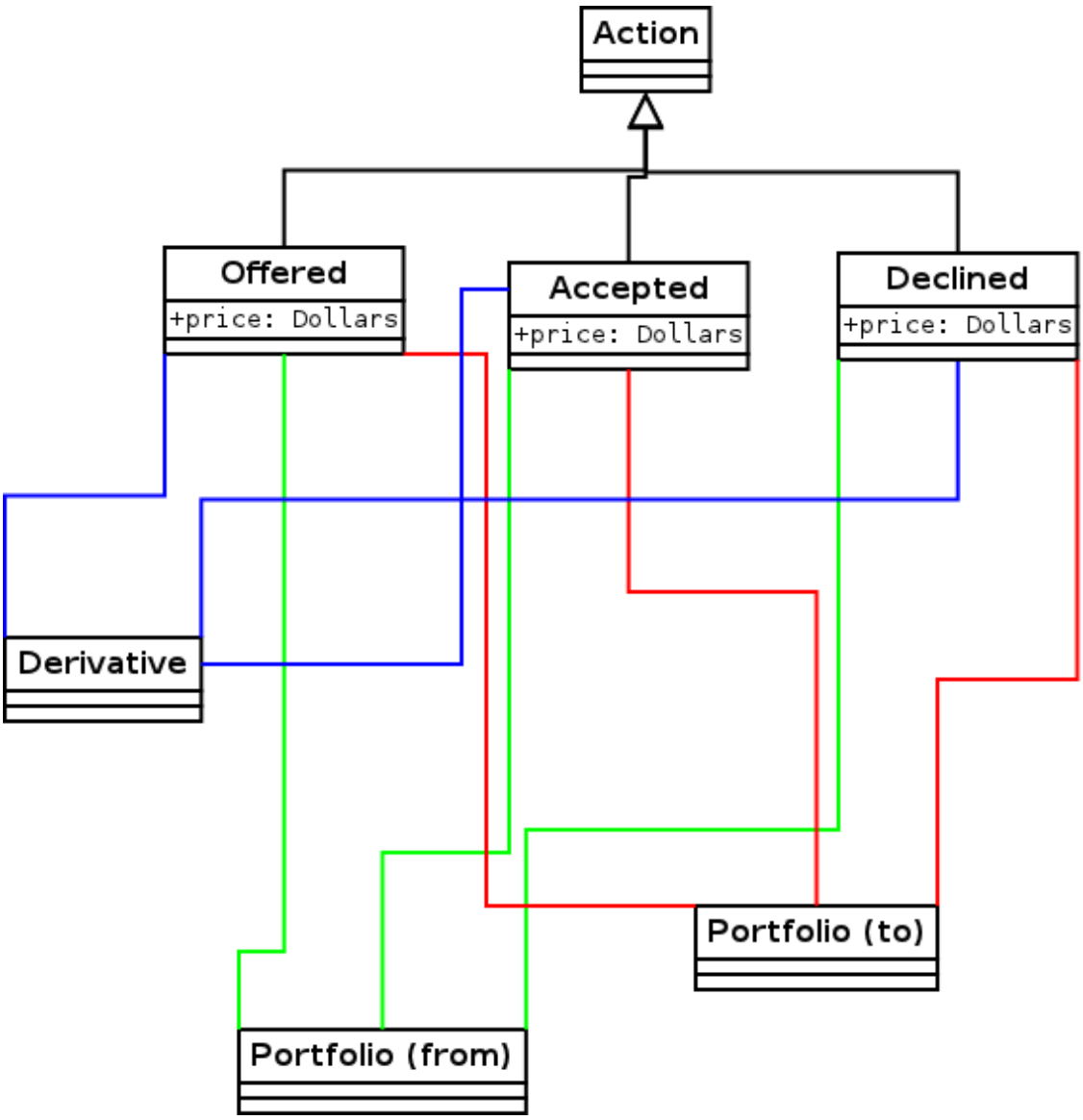
The basic model for News is:



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

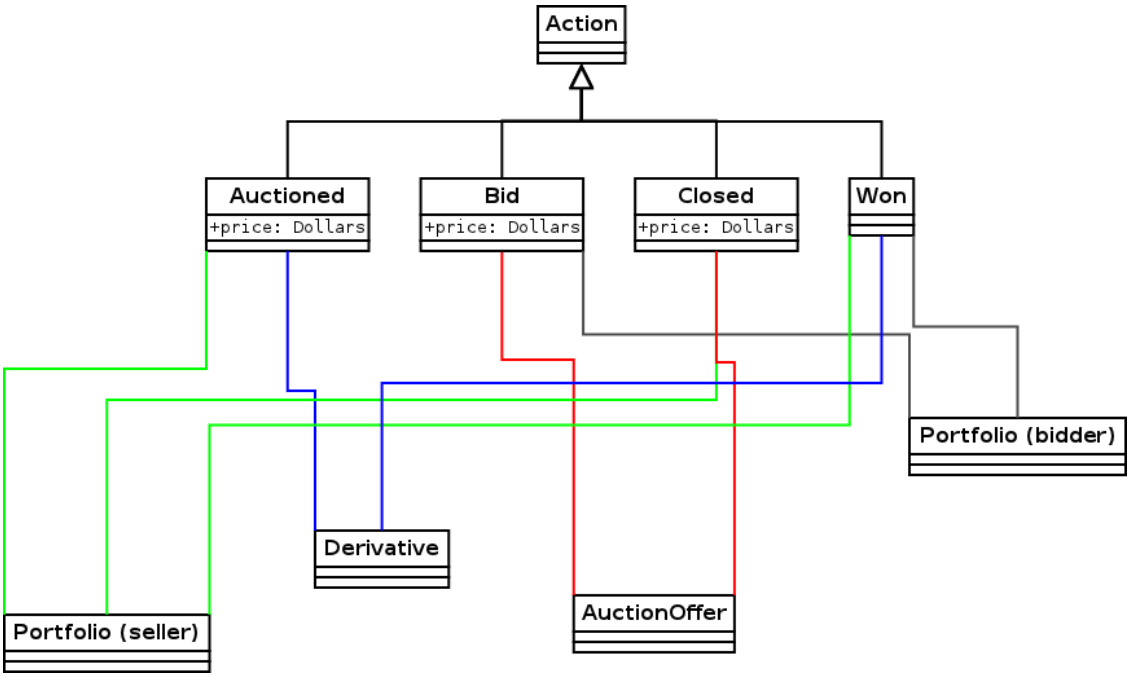
Buying and selling stocks, as shown above, refer to the Portfolio who "did" the action, and the information about what was bought or sold. This only applies to orders that are executed (either immediately or later). Orders that are delayed will generate another kind of an event.

Derivative Trading has the following kinds of events:



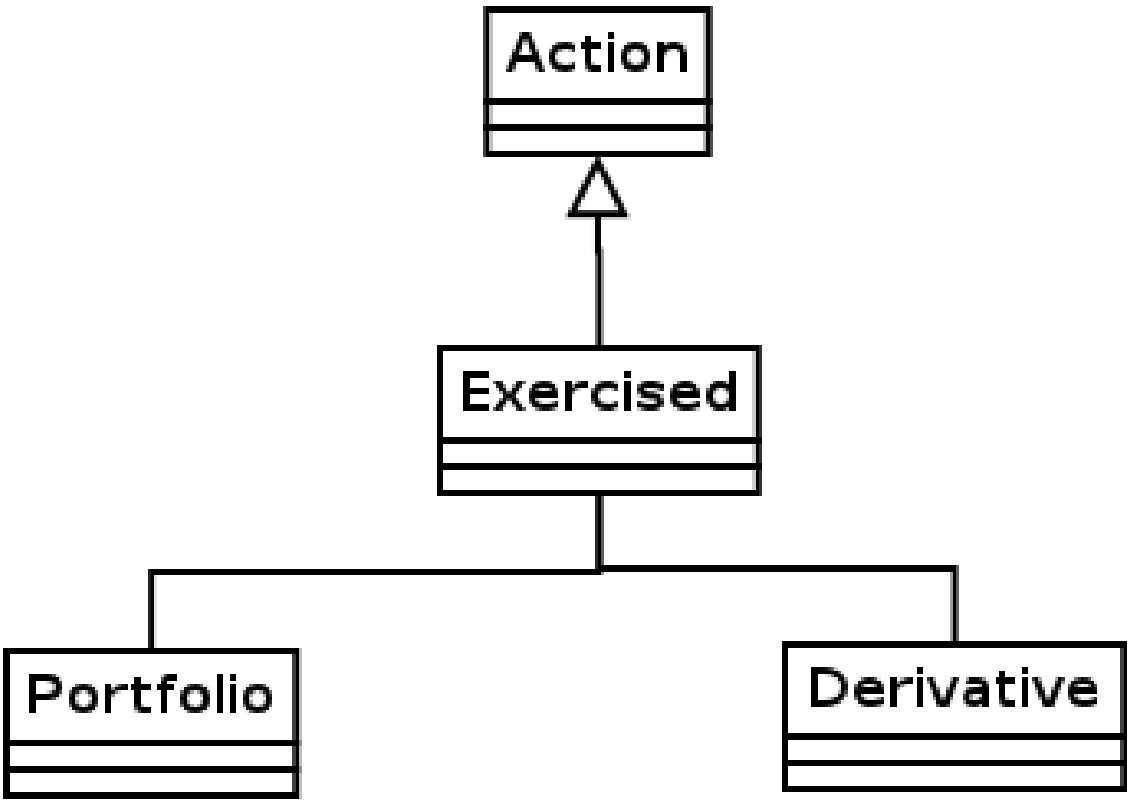
`from` and `to` are shown as separate concepts even though they are instances of the same class, because they play a different role in these events: one is the portfolio making the offer, the other is the portfolio receiving, and possible accepting, the offer.

For Auctions we have:



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

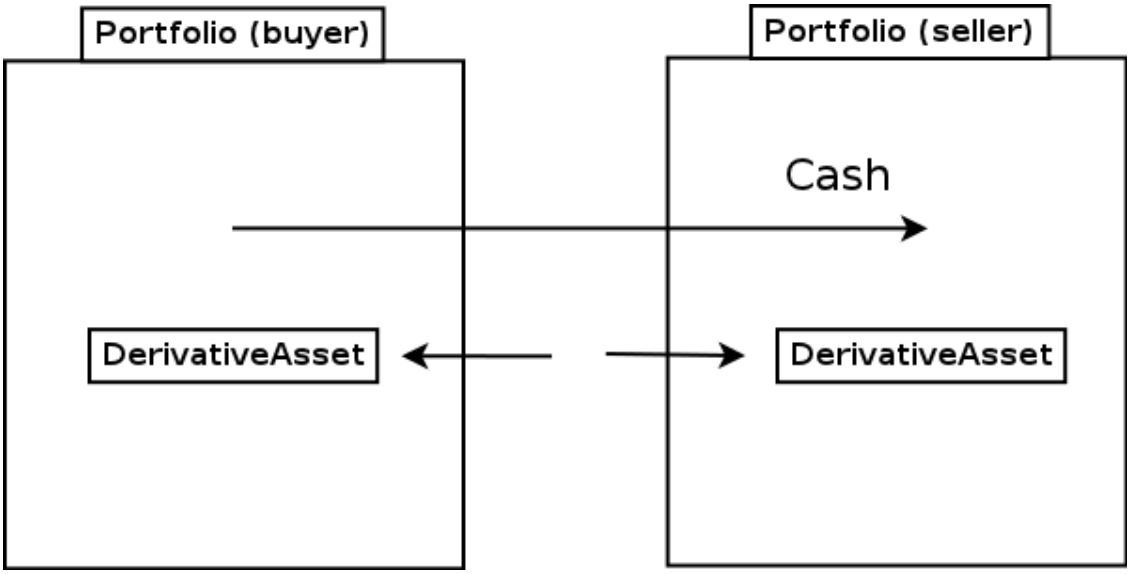
Placing orders that get delayed are described by:



Where the associated portfolio is the one who performed the buy or sell.
There is one more event for exercising derivatives:
Where the associated portfolio is the one who did the exercising.

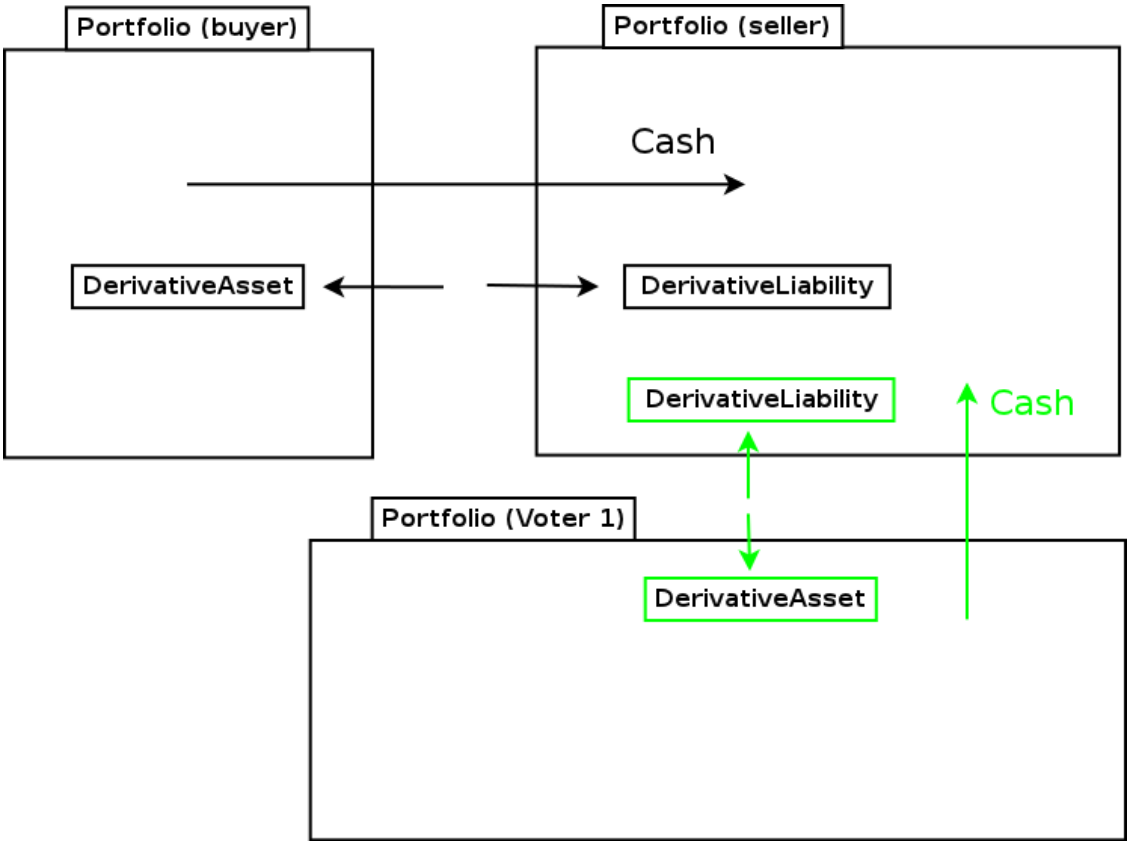
5.8 Voting

When players enter into a contract involving a derivative, the following assets are moved:



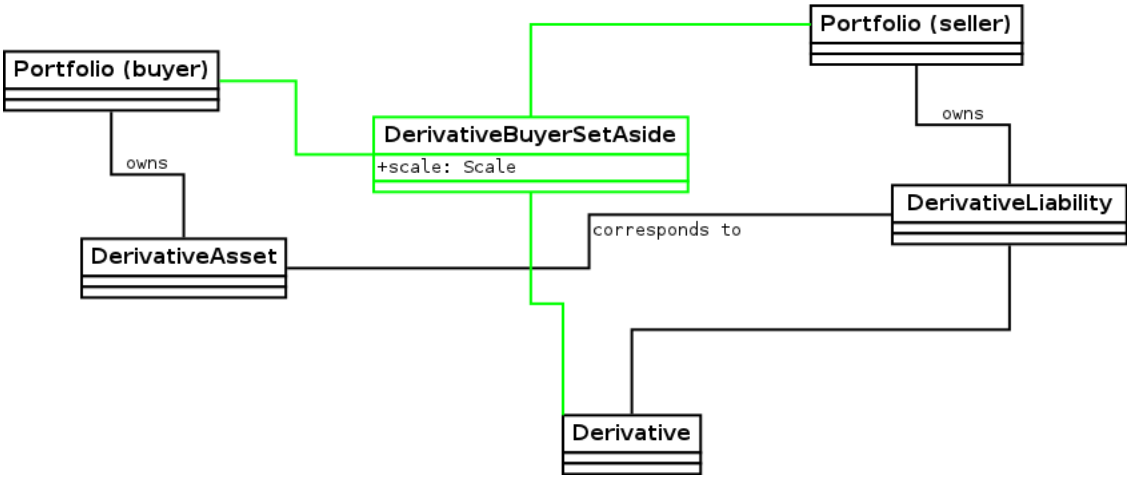
If owning the asset (being in the buyer side of the contract) pays off more than the cash payed, the buyer is happy. If owning the liability (being in the seller side of the contract) is not bad enough to negate the cash received, the seller got a good deal. These are not necessarily mutually exclusive.

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:



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).

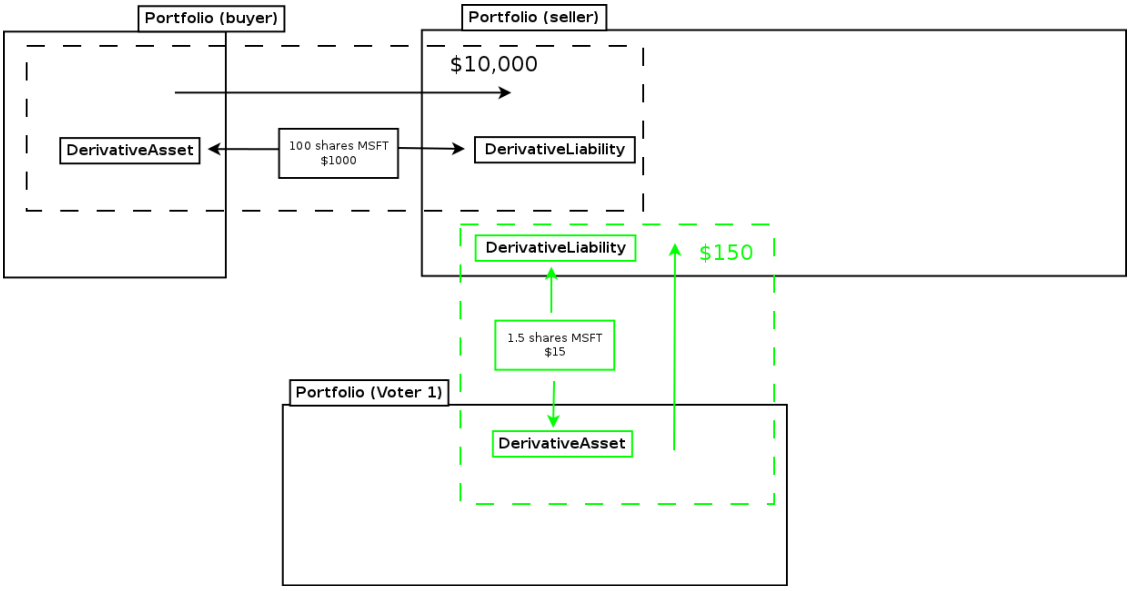
When two portfolios enter a derivative, an object is created called `DerivativeBuyerSetAside` (there is a nearly identical process for sellers, which is discussed next):



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

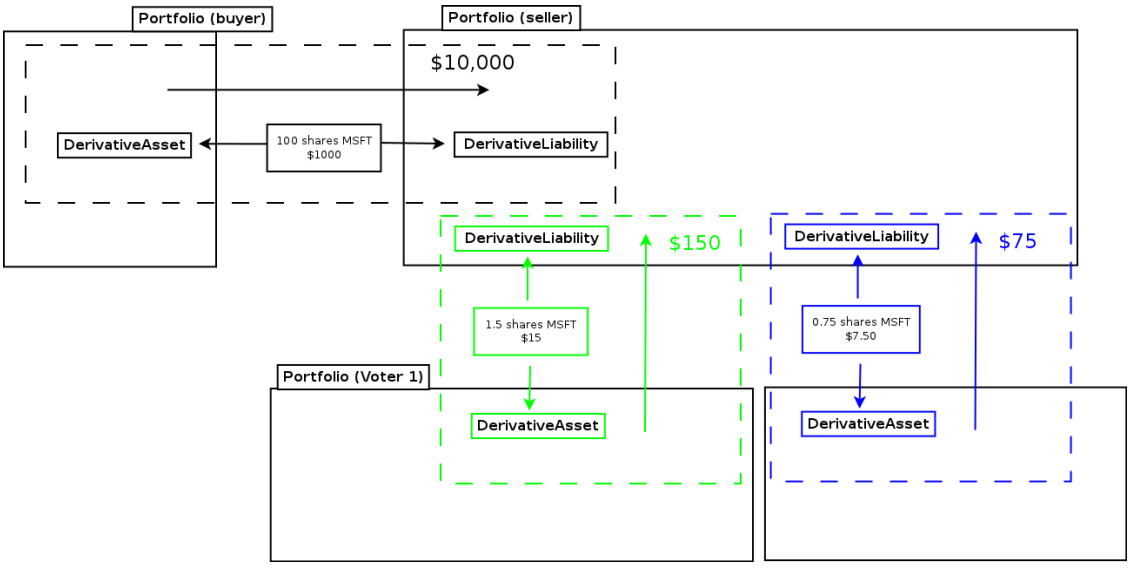
The DerivativeBuyerSetAside holds one attribute, which is the "amount" left to be voted on. For the precise meaning of this scale, see the section on Scaling Derivatives.

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:

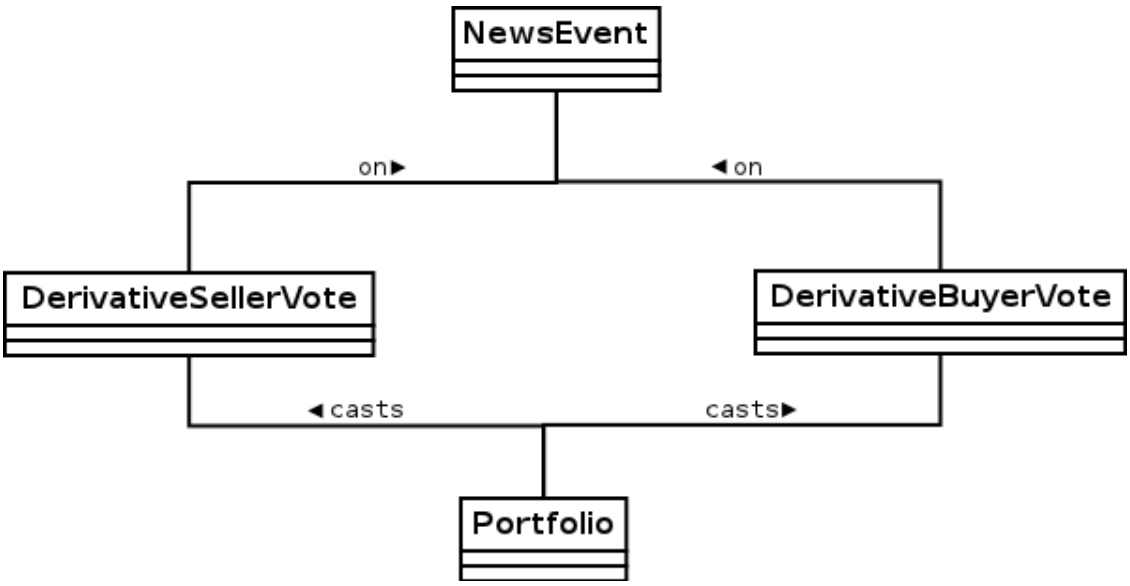


The `scale` remaining is then cut by half to 1.5%.

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



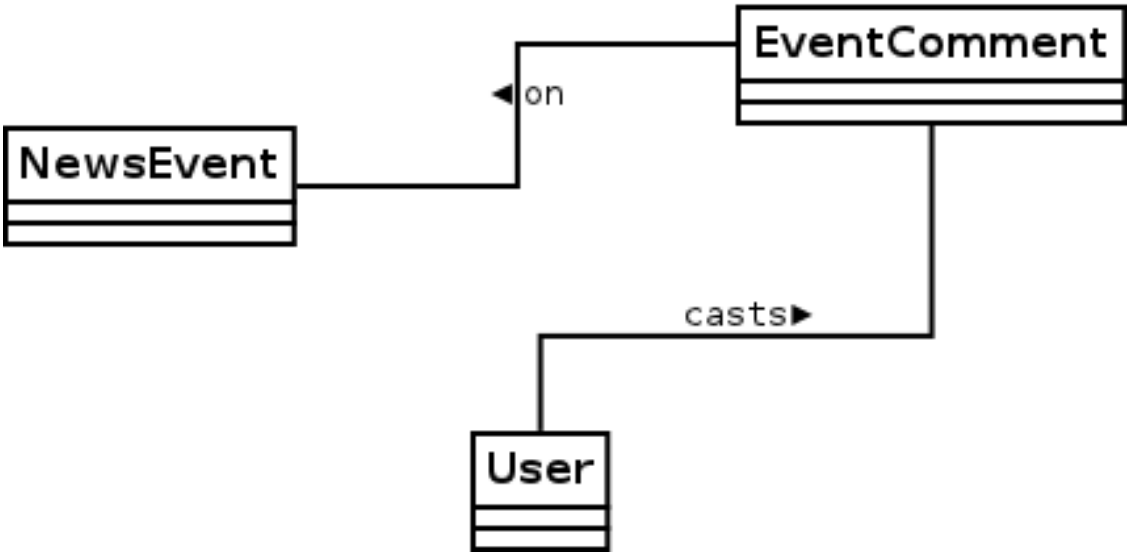
Votes are recorded and associated with the origanal NewsEvent, so that a score of buyer-votes and seller votes can be calculated:



5.9 Comments

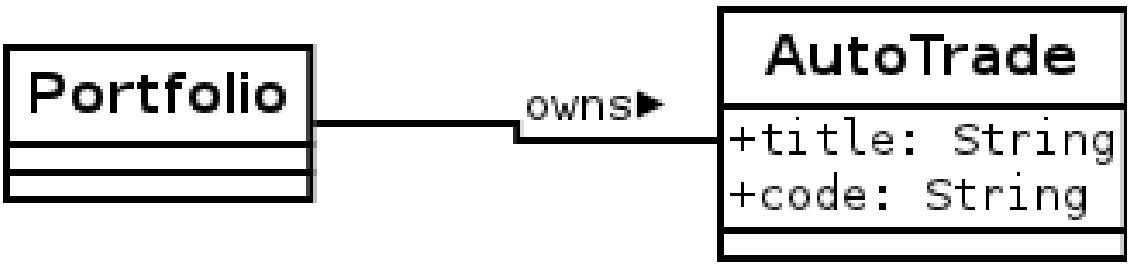
Compared to voting, comments are refreshingly simple.

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

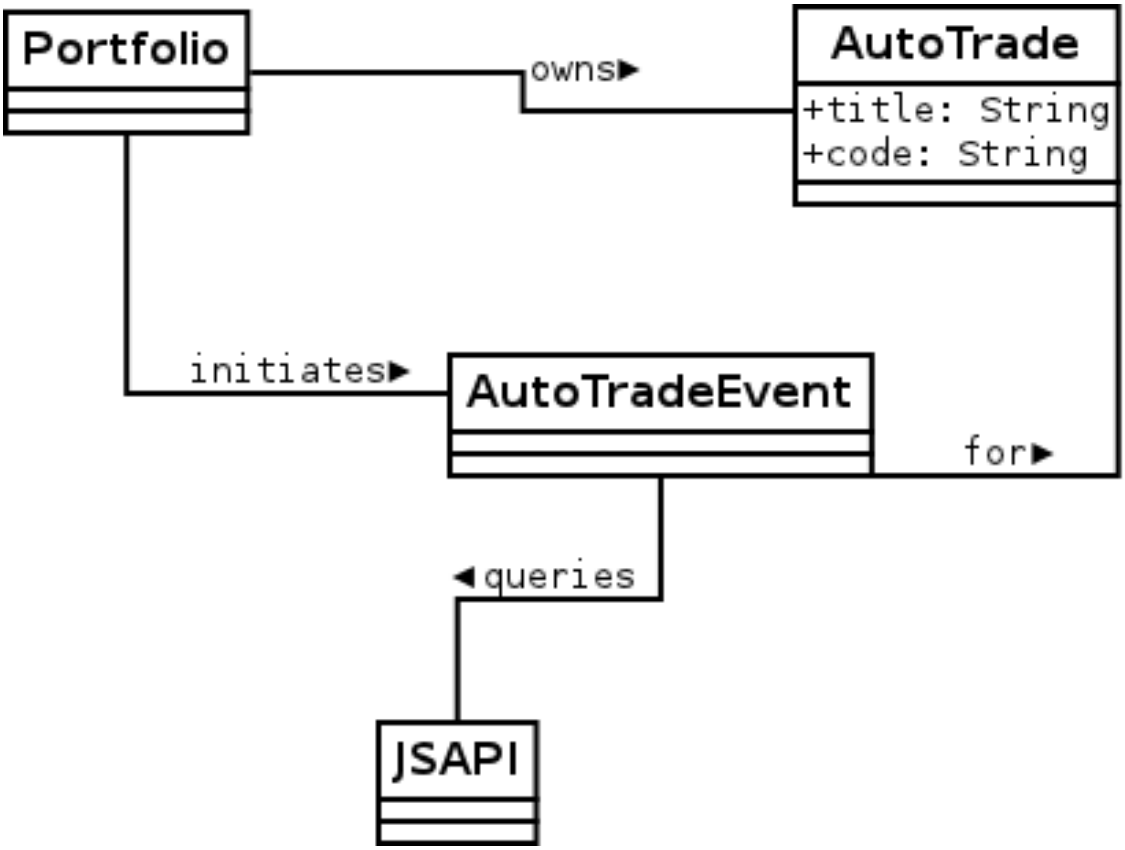


5.10 Auto Trades

While the system is idle, an auto-trade is represented as:



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



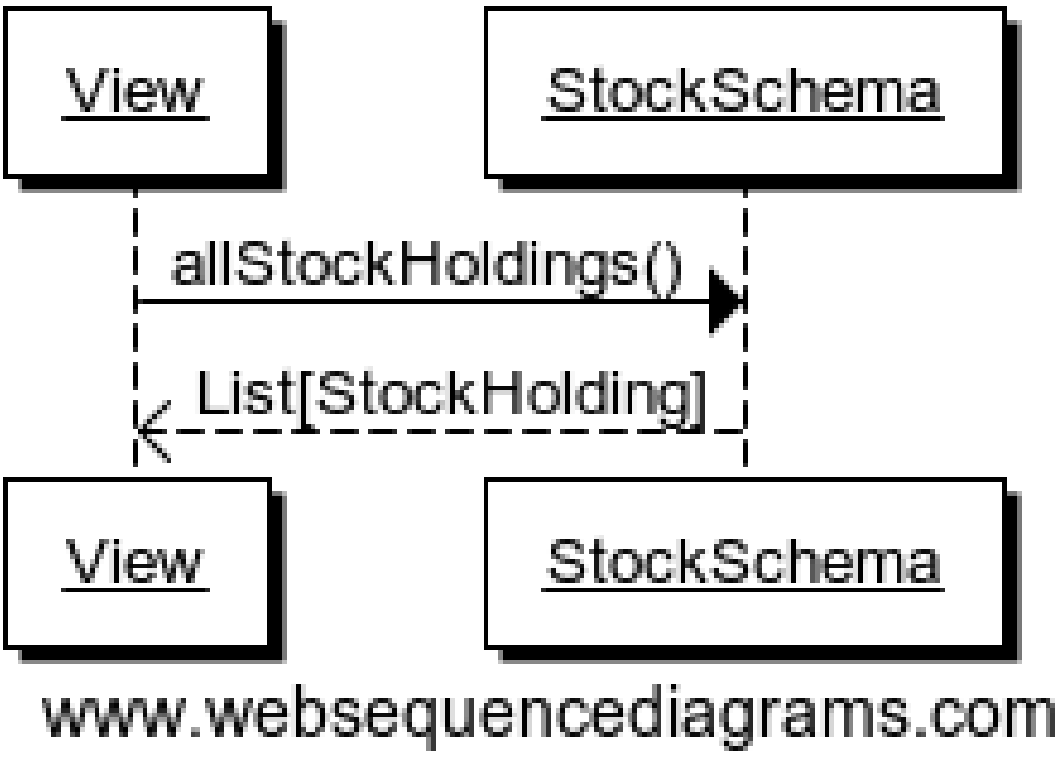
The `JSAPI` is a set of JavaScript functions and corresponding server-side handlers that allow the Auto Trade to actually perform actions.

6 Perturbations and Interactions

6.1 Stocks

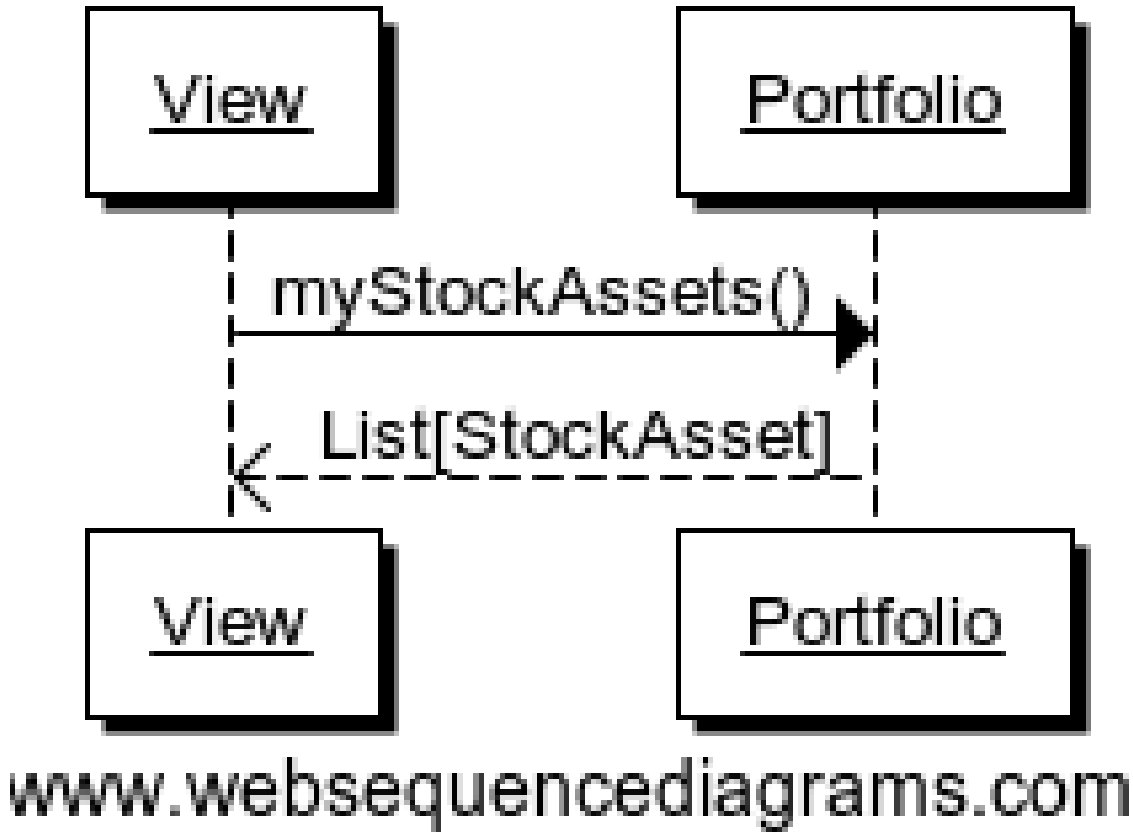
6.1.1 `allStockHoldings`

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



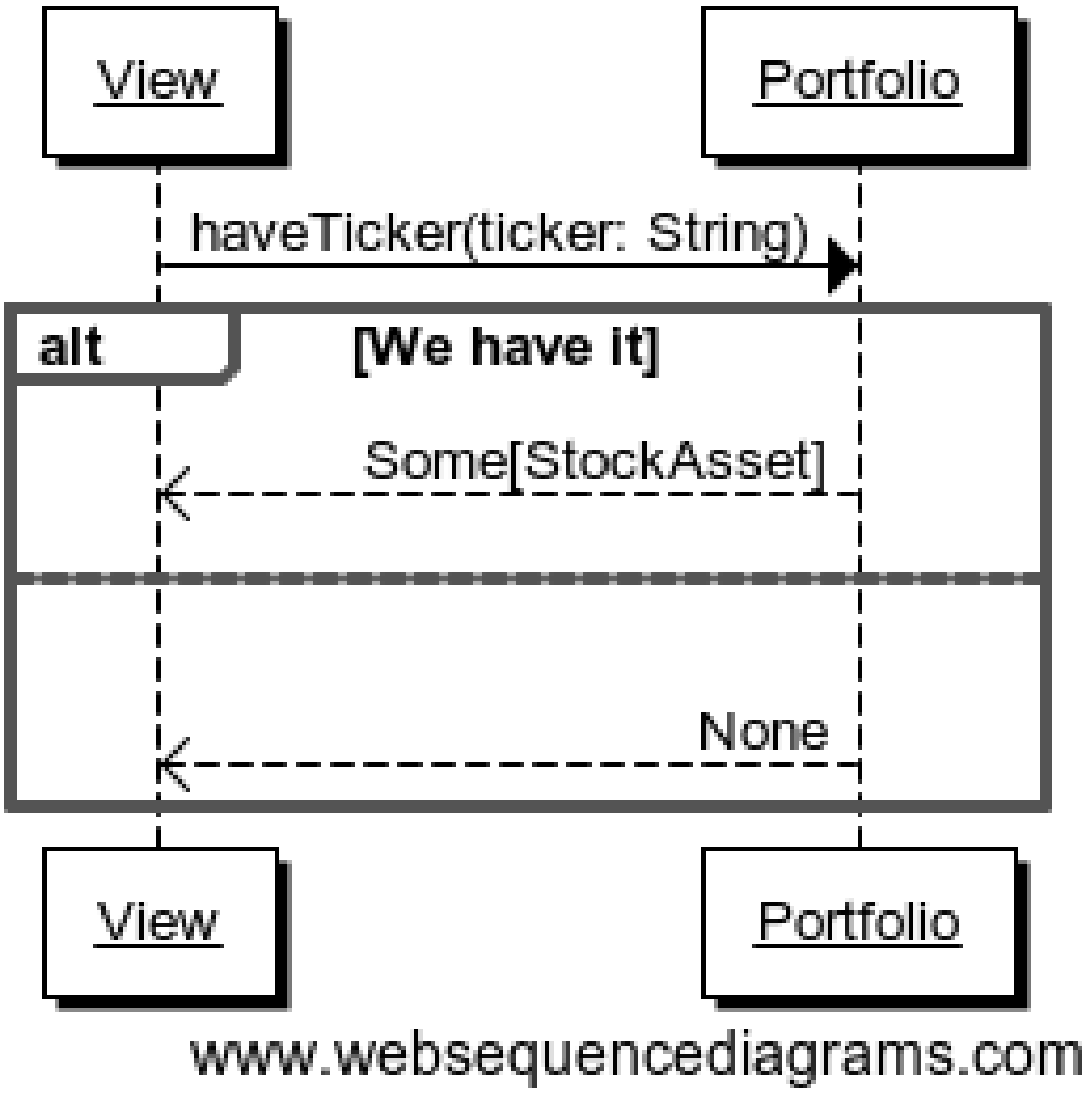
6.1.2 `Portfolio.myStockAssets`

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



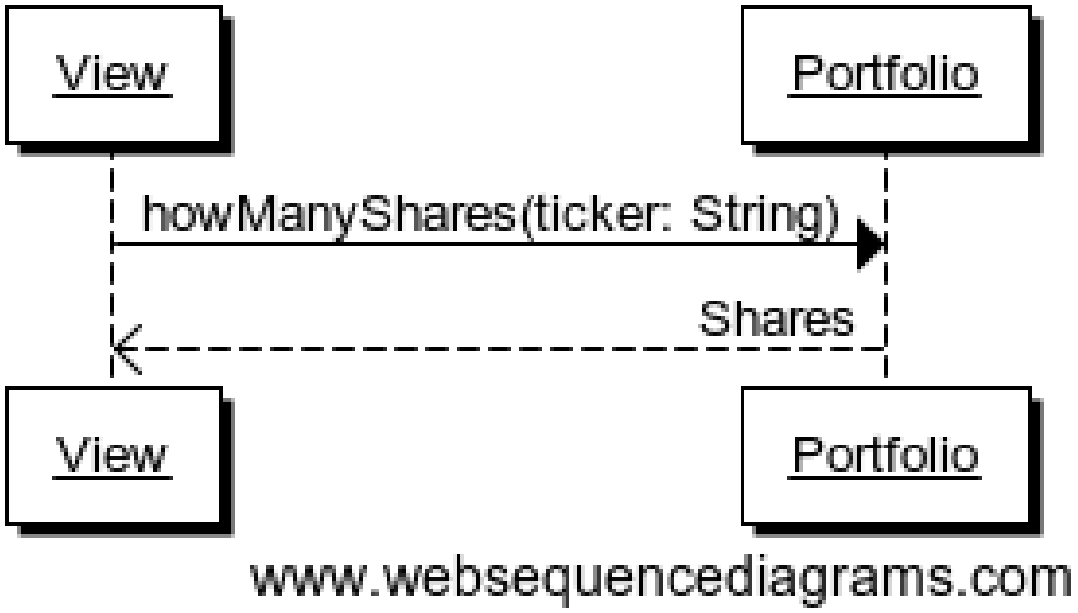
6.1.3 `Portfolio.haveTicker`

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



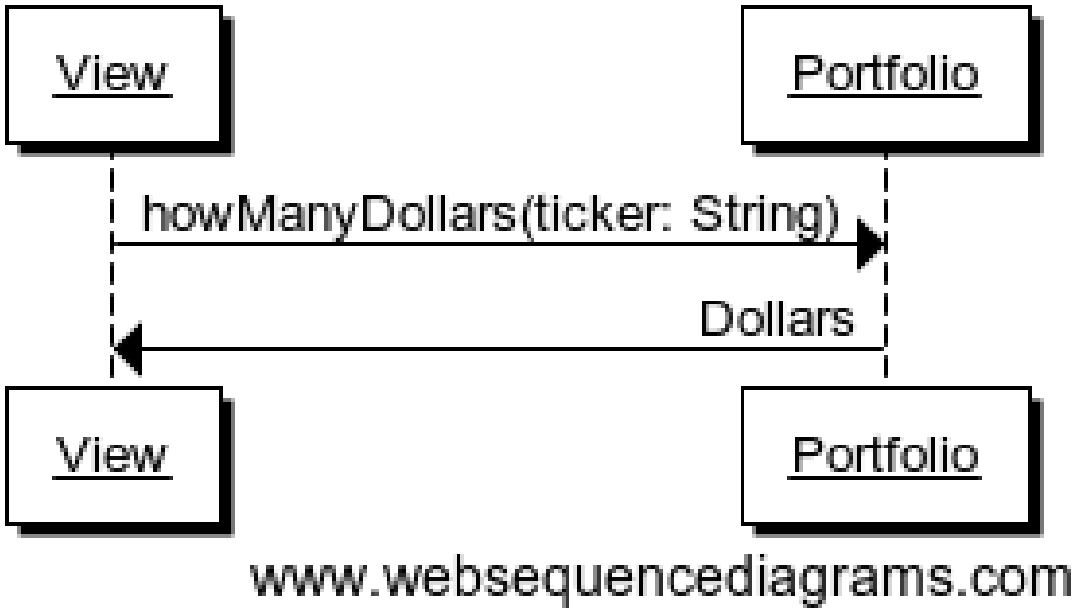
6.1.4 `Portfolio.howManyShares`

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



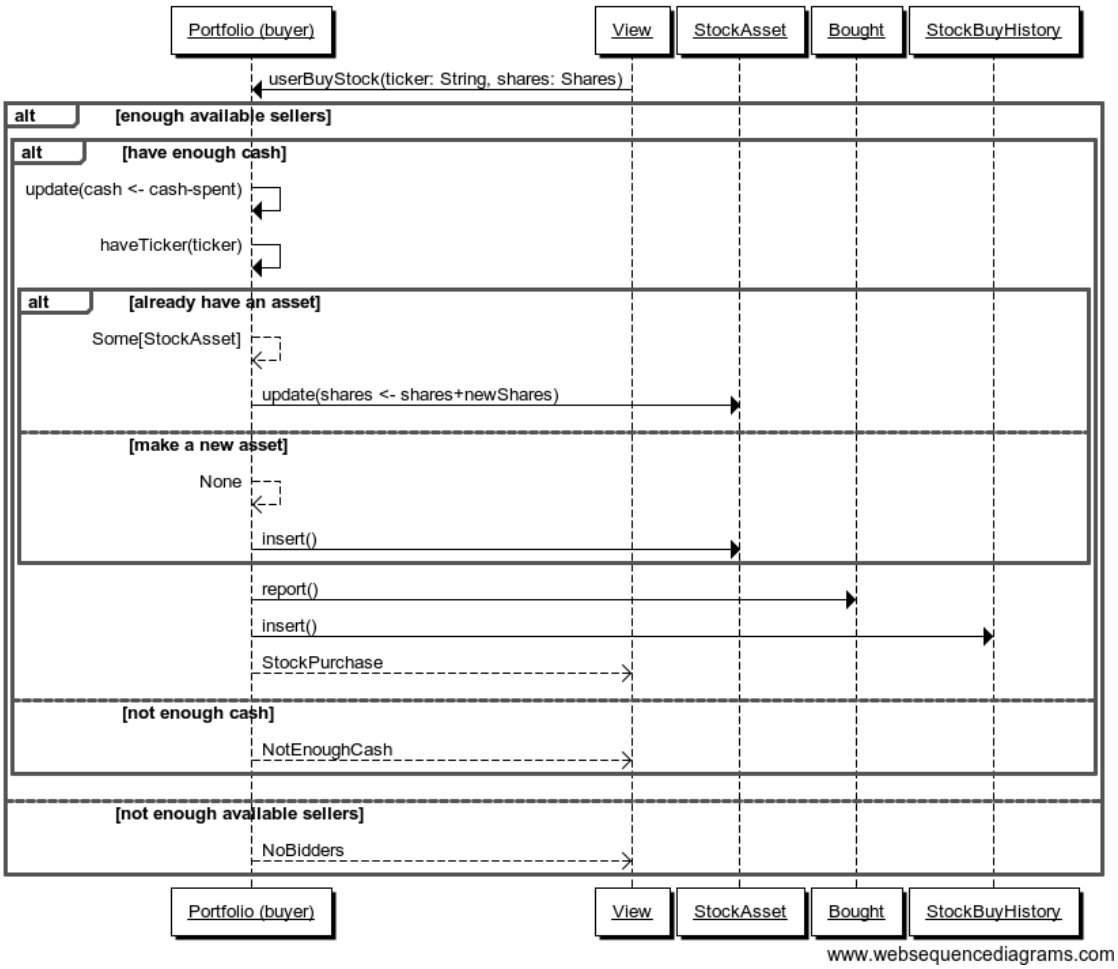
6.1.5 `Portfolio.howManyDollars`

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



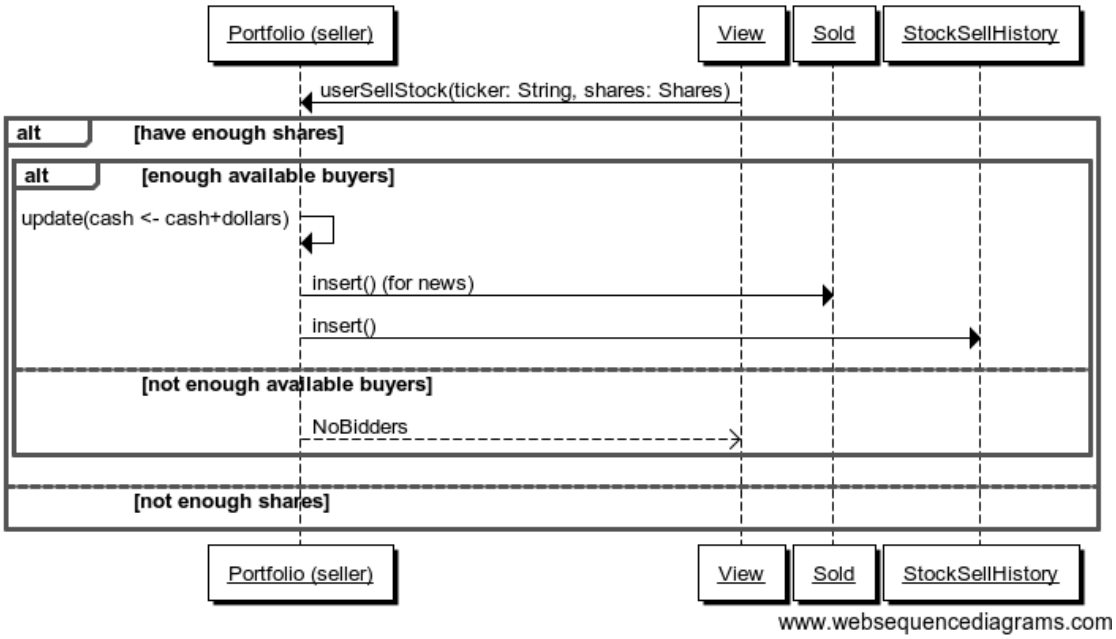
6.1.6 `Portfolio.userBuyStock`

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



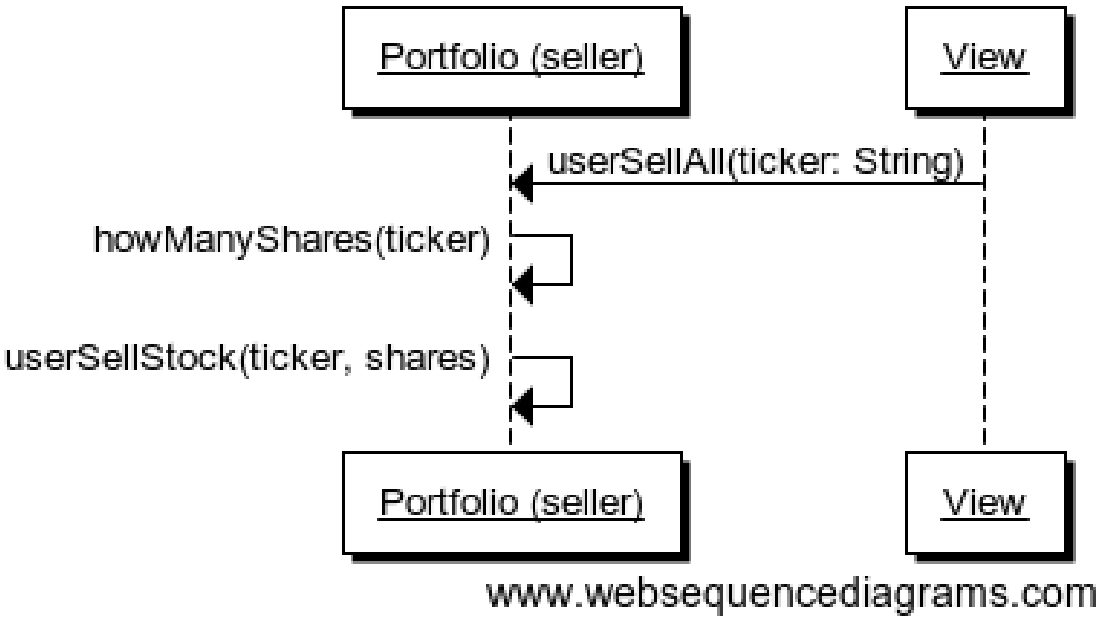
6.1.7 `Portfolio.userSellStock`

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



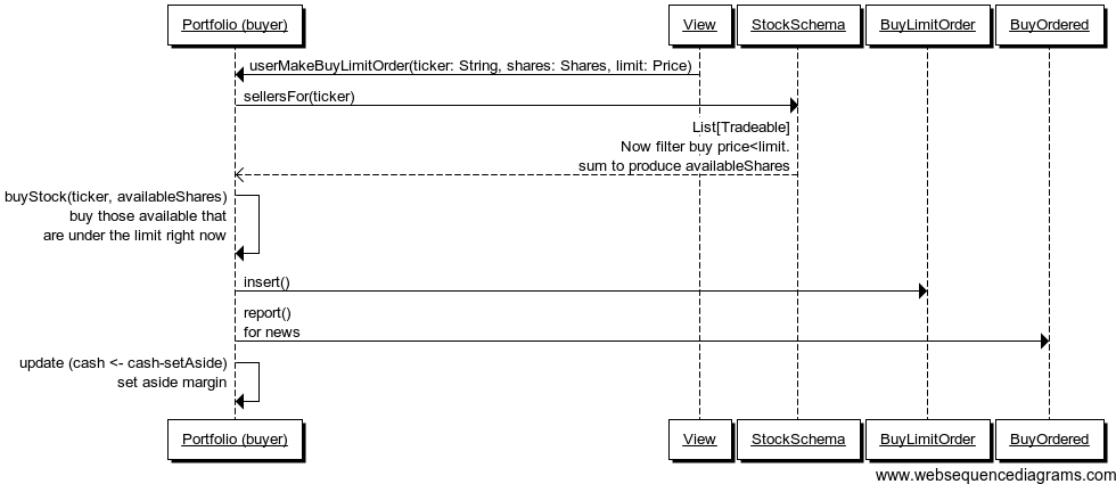
6.1.8 `Portfolio.userSellAll`

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



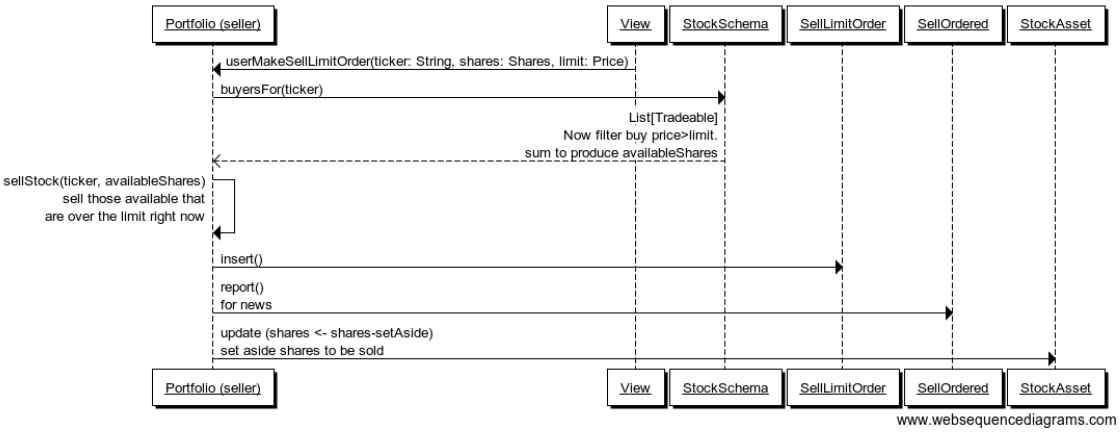
6.1.9 `Portfolio.userMakeBuyLimitOrder`

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).



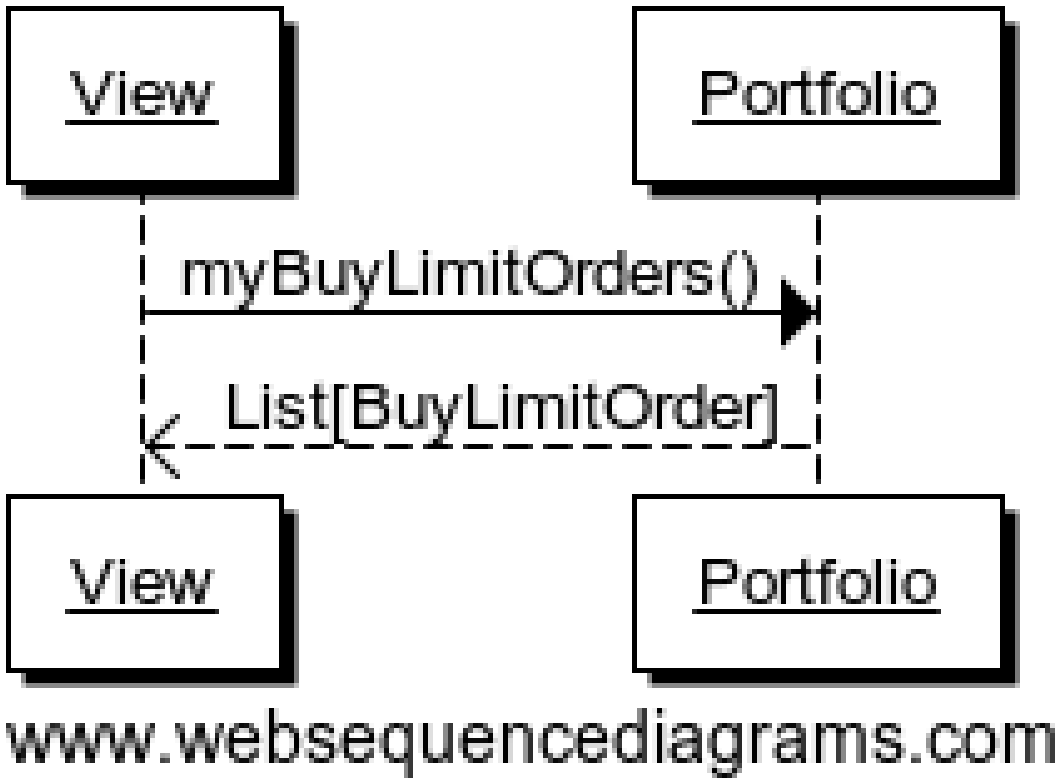
6.1.10 `Portfolio.userMakeSellLimitOrder`

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).



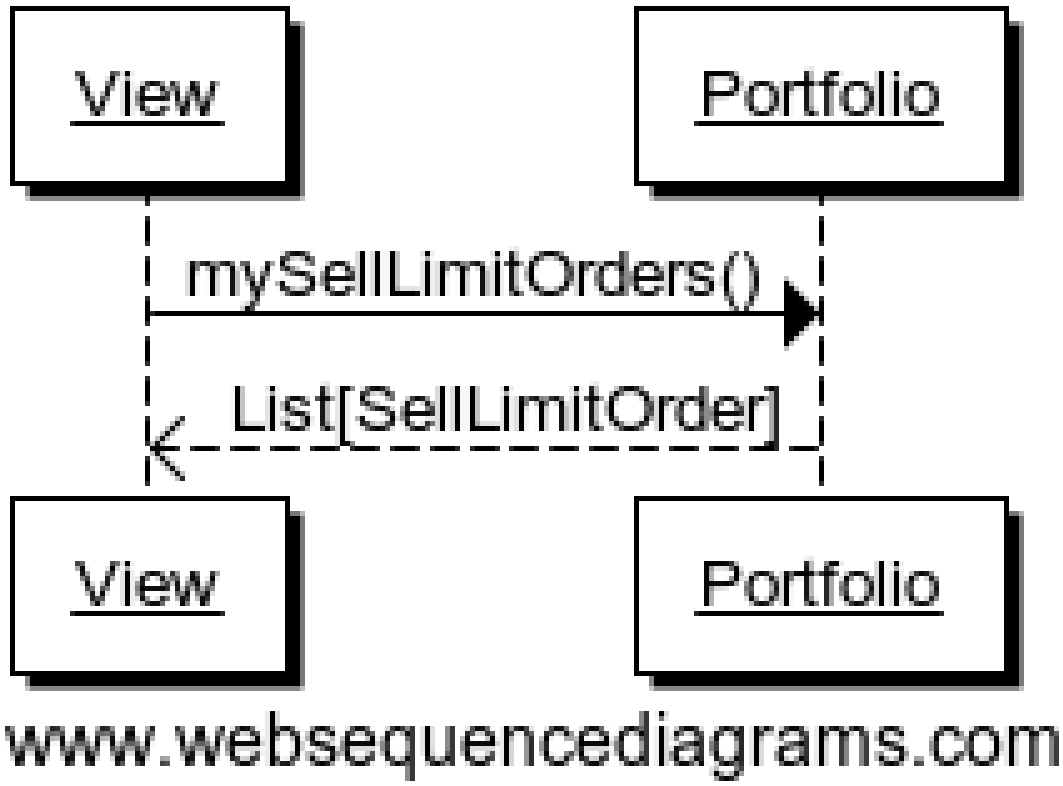
6.1.11 `Portfolio.myBuyLimitOrders`

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



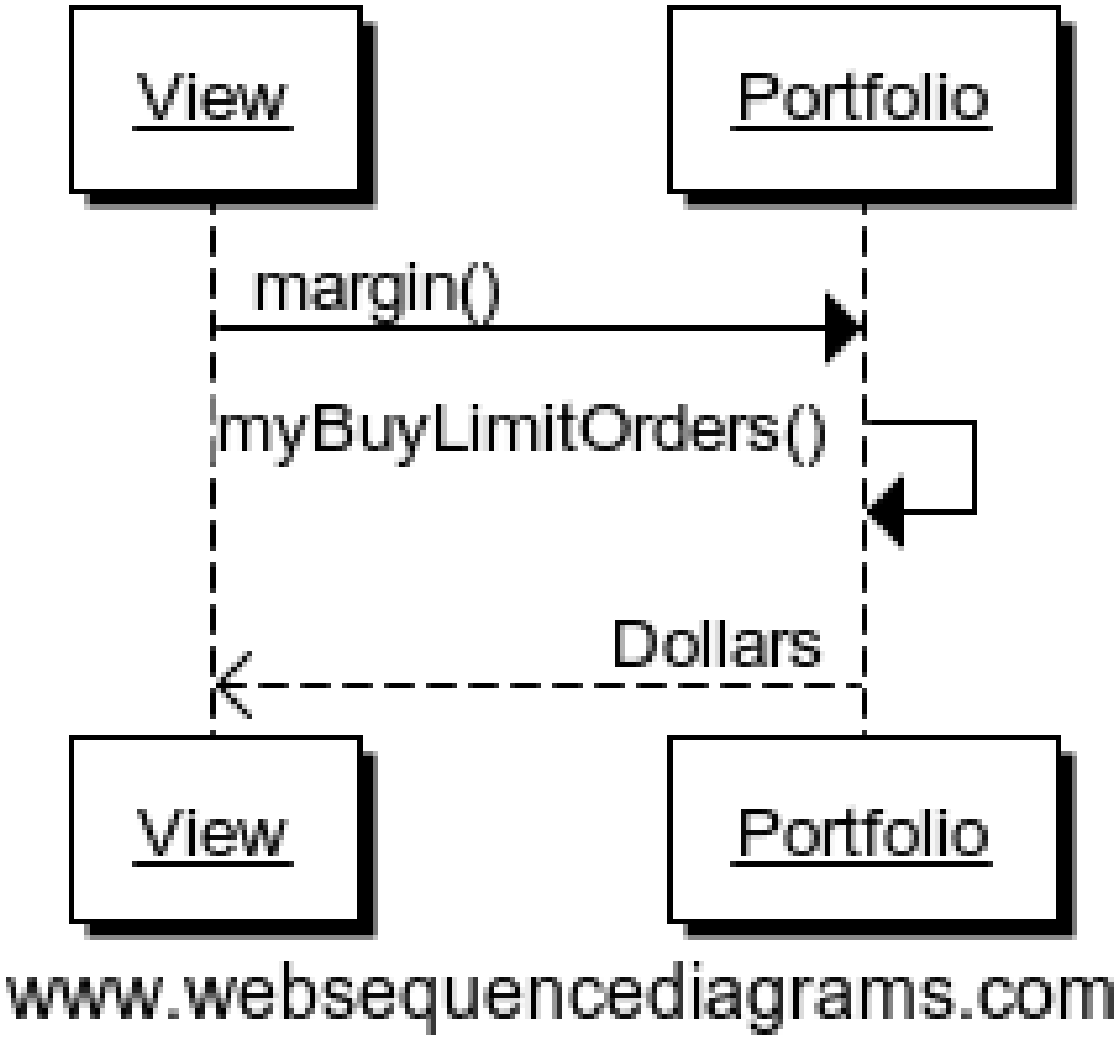
6.1.12 `Portfolio.mySellLimitOrders`

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



6.1.13 `Portfolio.margin`

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



6.2 Derivatives

6.2.1 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).

6.2.1.1 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).

6.2.1.2 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).

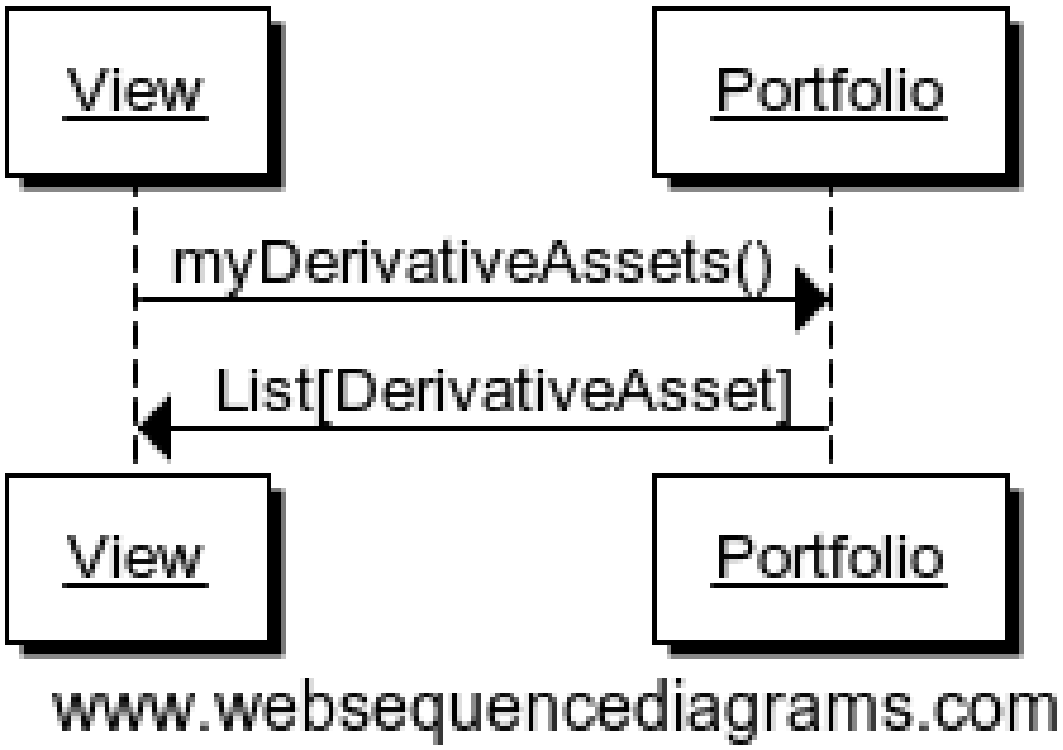
3. If A does not have enough shares *or* enough cash, this should generate a margin call and A's assets should be liquidated, but PitFail does not support this feature.

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).

6.2.1.3 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.

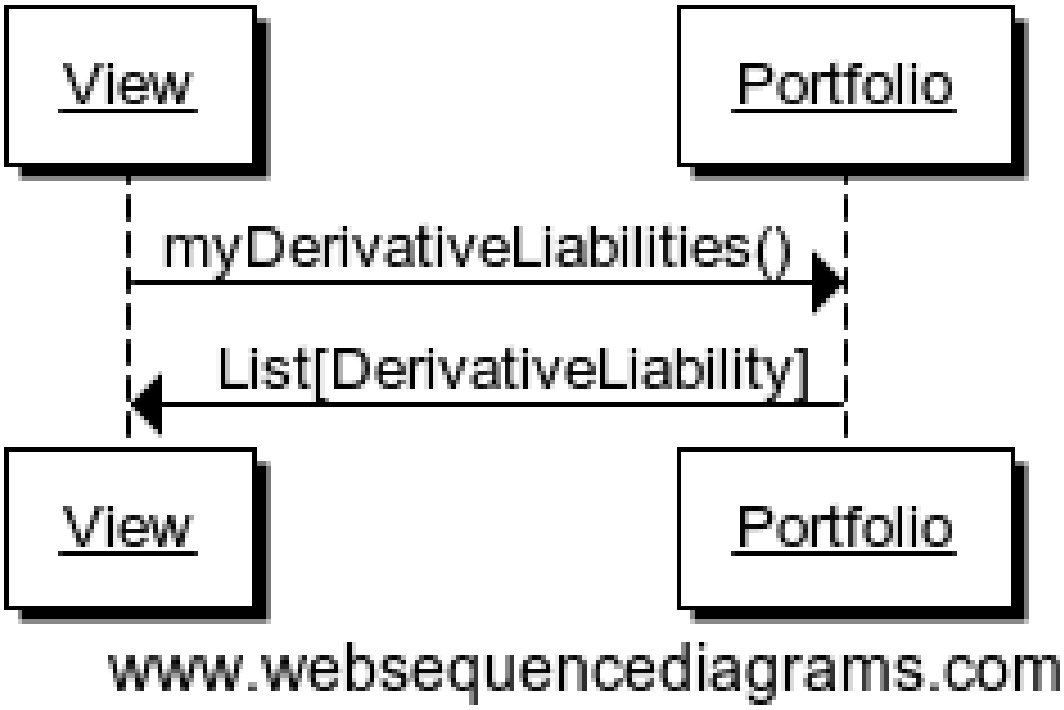
6.2.2 `Portfolio.myDerivativeAssets`

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



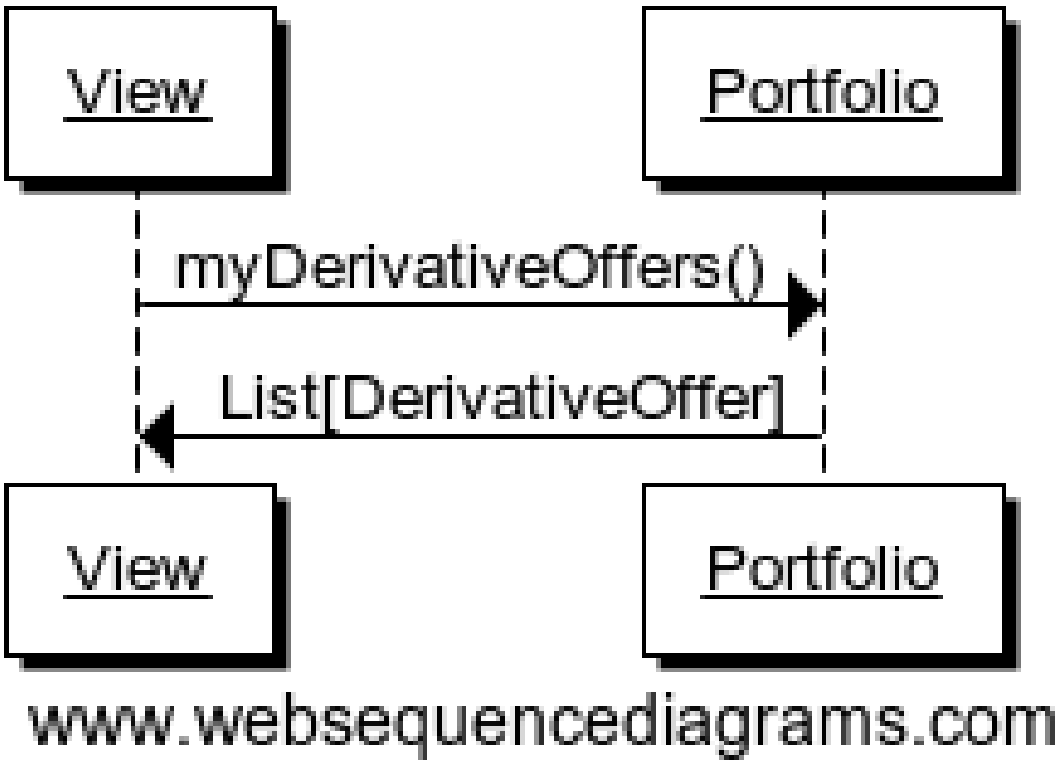
6.2.3 `Portfolio.myDerivativeLiabilities`

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



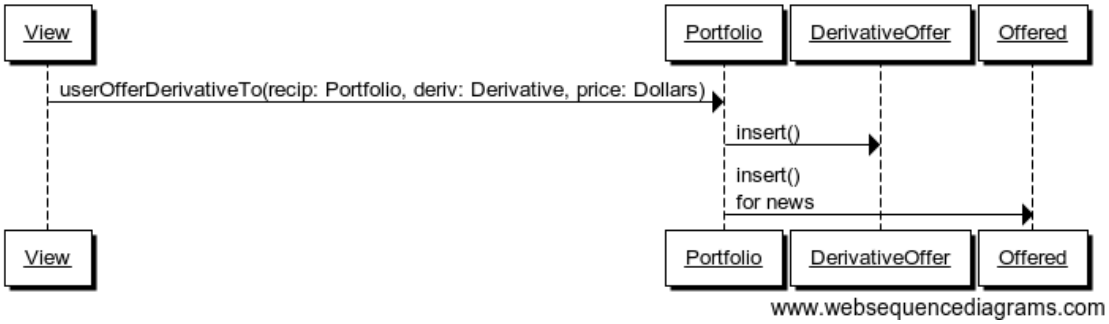
6.2.4 `Portfolio.myDerivativeOffers`

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



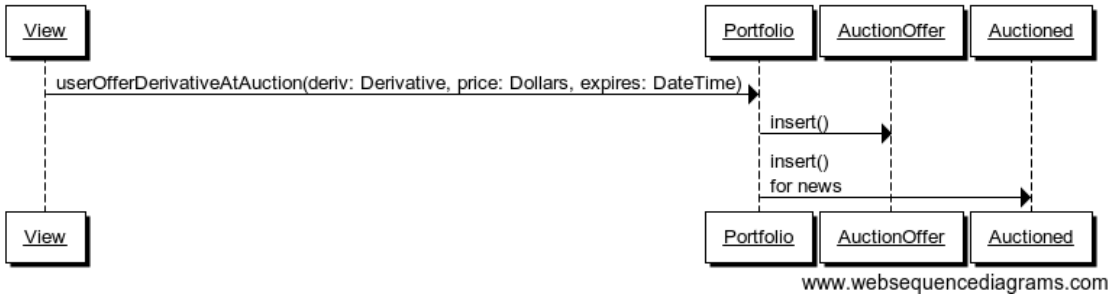
6.2.5 `Portfolio.userOfferDerivativeTo`

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



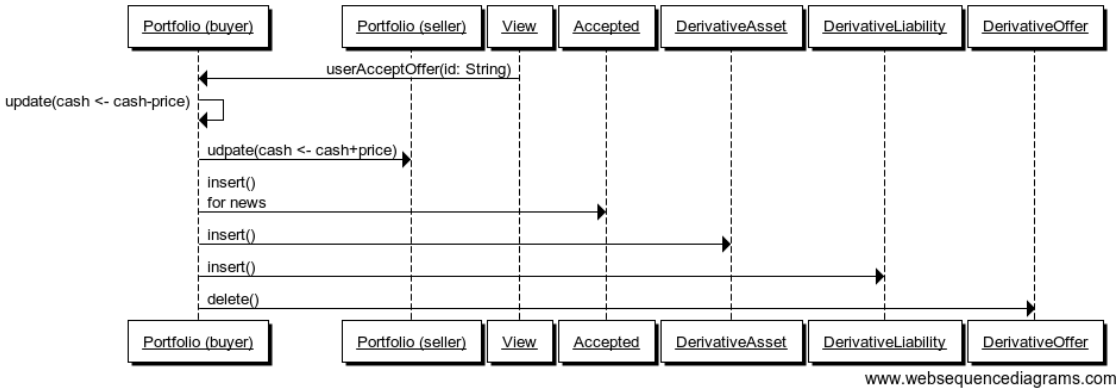
6.2.6 `Portfolio.userOfferDerivativeAtAuction`

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



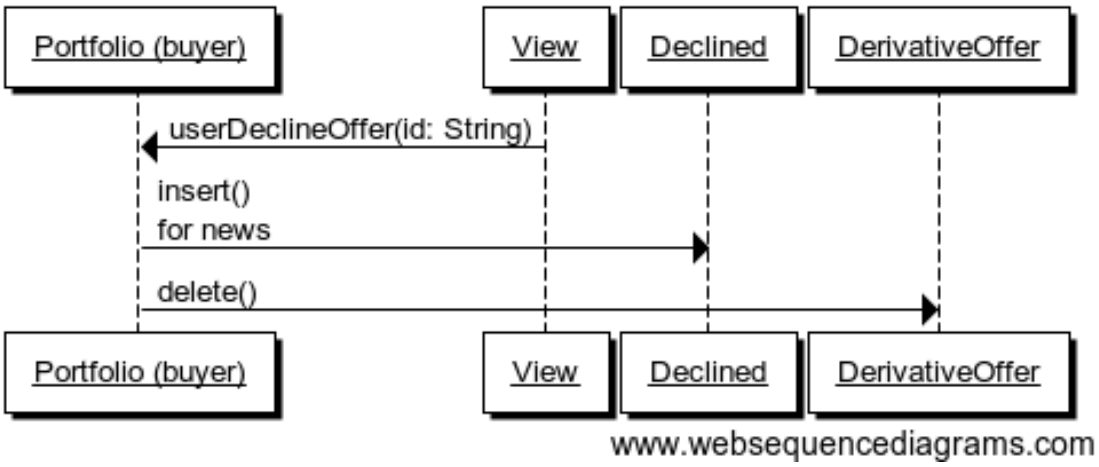
6.2.7 `Portfolio.userAcceptOffer`

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



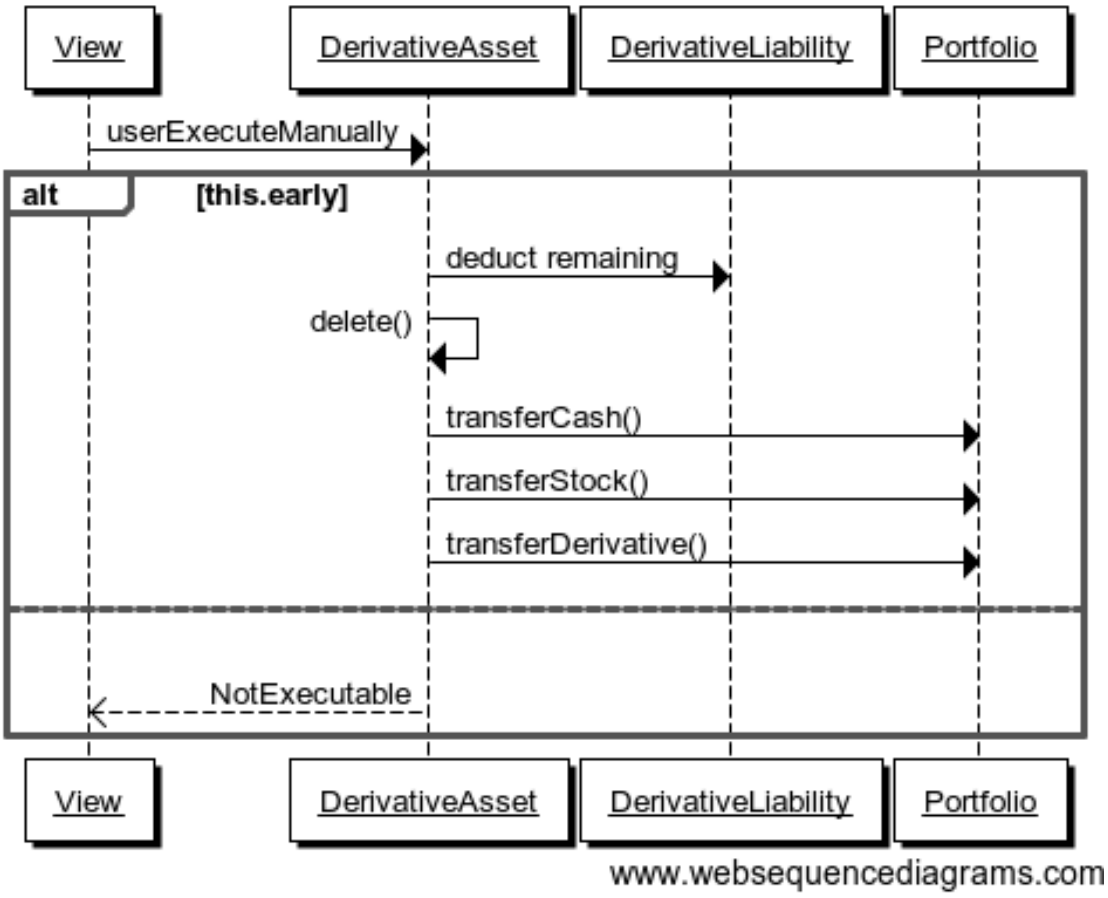
6.2.8 `Portfolio.userDeclineOffer`

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



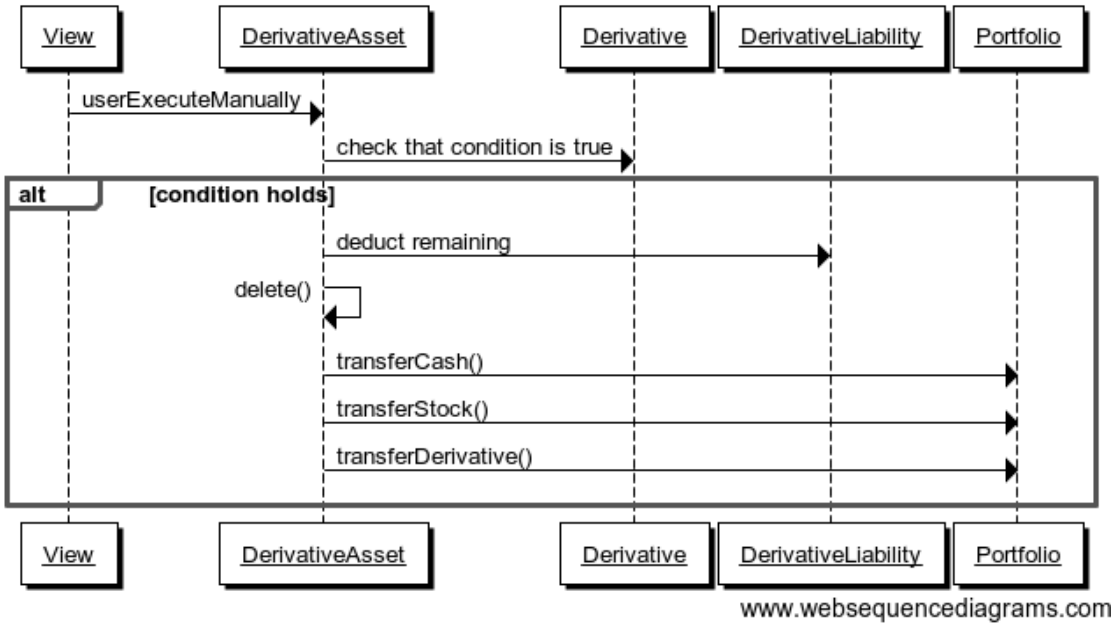
6.2.9 `DerivativeAsset.userExecuteManually`

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



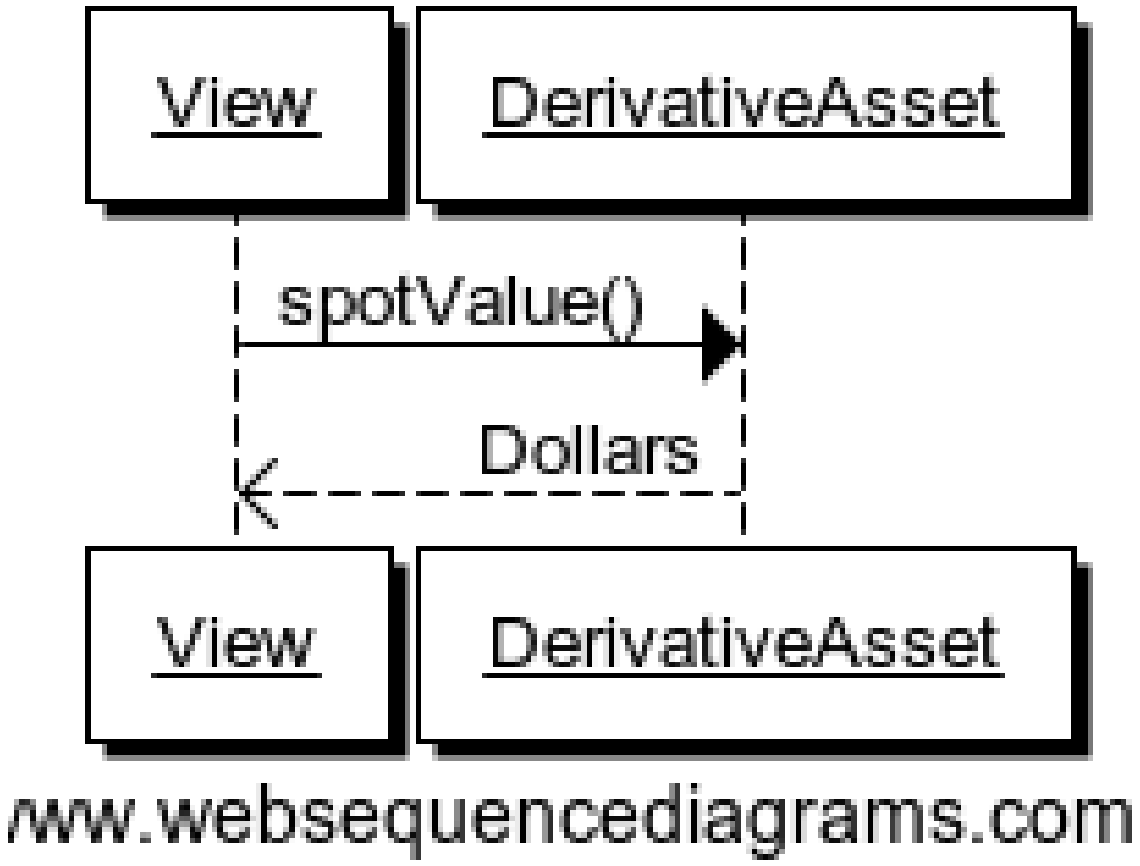
6.2.10 `DerivativeAsset.systemExecuteOnSchedule`

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



6.2.11 `DerivativeAsset.spotValue`

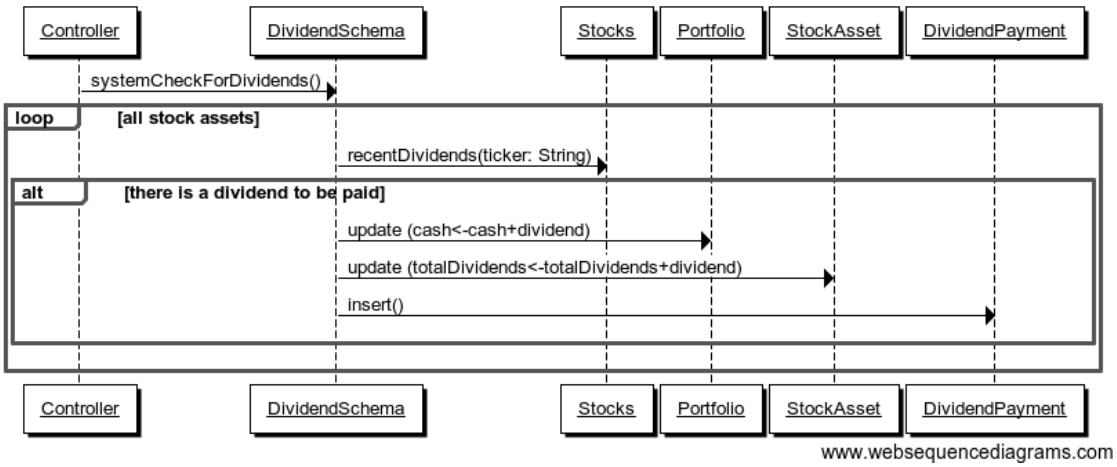
Gets how much a derivative would be worth should it be exercised today (model/derivatives.scala ref_319).



6.3 Dividends

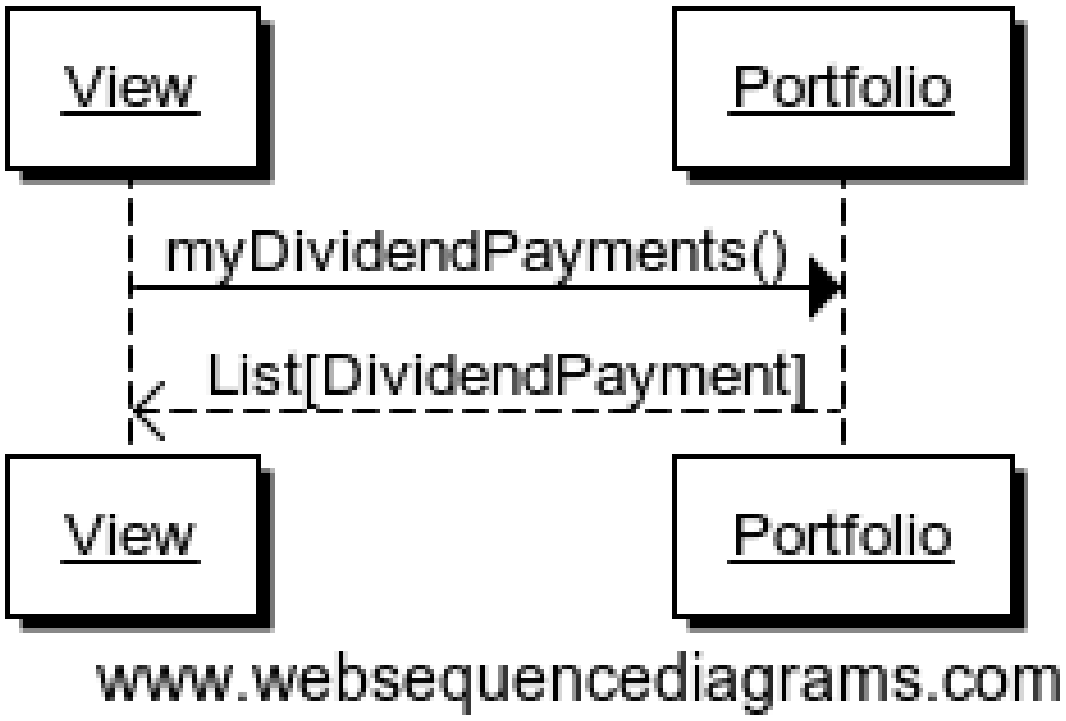
6.3.1 `DividendSchema.systemCheckForDividends`

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



6.3.2 `Portfolio.myDividendPayments`

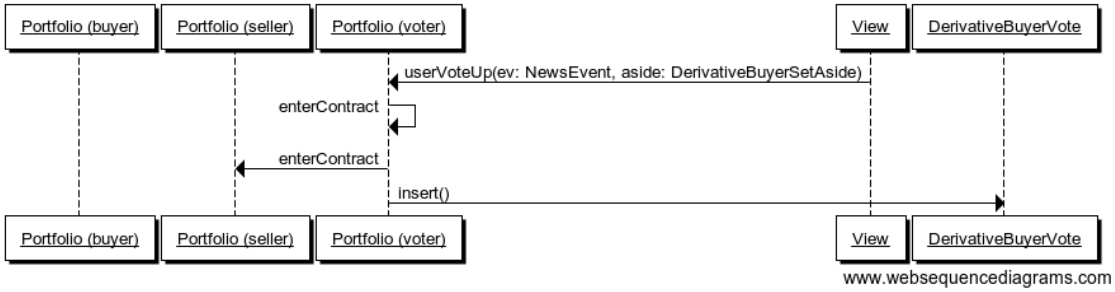
Gets a list of dividend payments that we have received (model/dividends.scala ref_489).



6.4 Voting

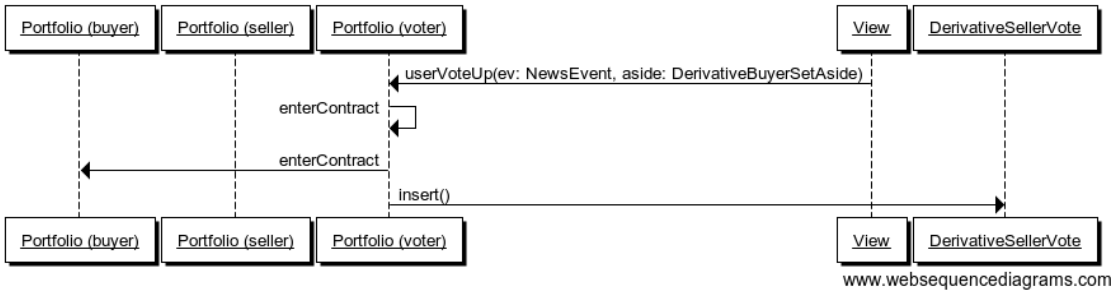
6.4.1 `Portfolio.userVoteUp`

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



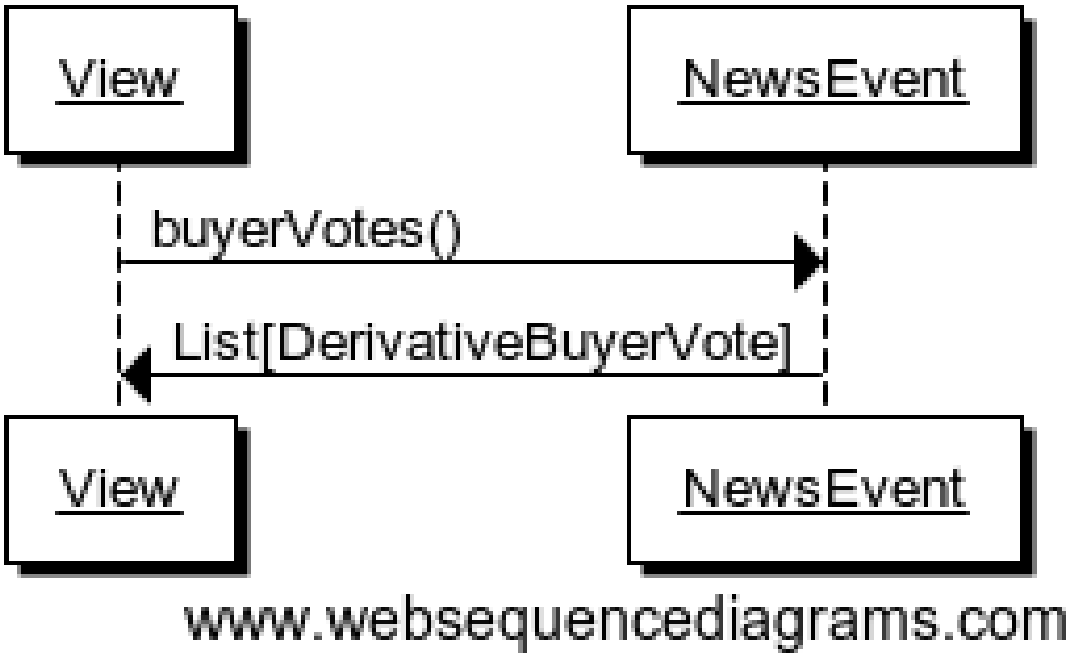
6.4.2 `Portfolio.userVoteDown`

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



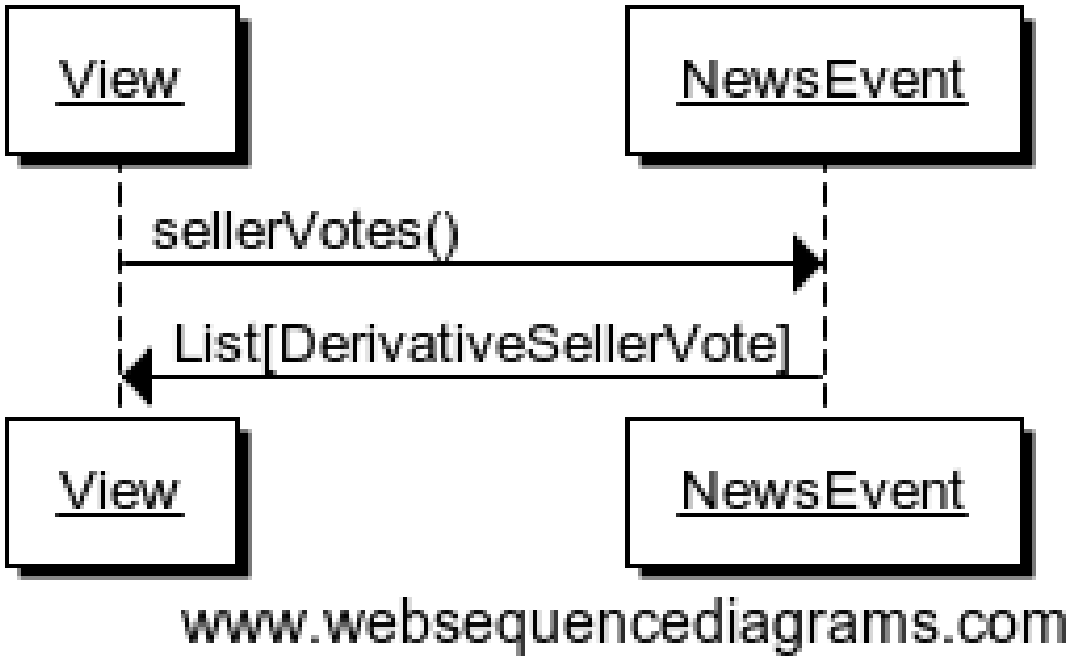
6.4.3 `NewsEvent.buyerVotes`

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



6.4.4 `NewsEvent.sellerVotes`

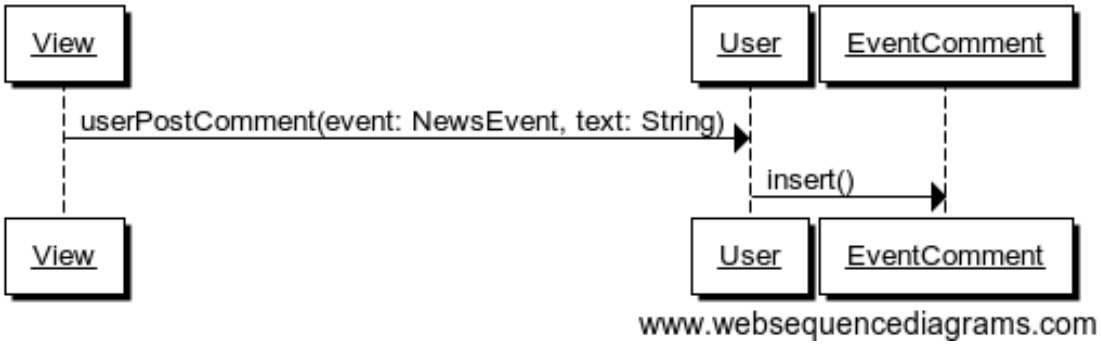
Gets all for-seller votes on this event (model/voting.scala ref_405).



6.5 Comments

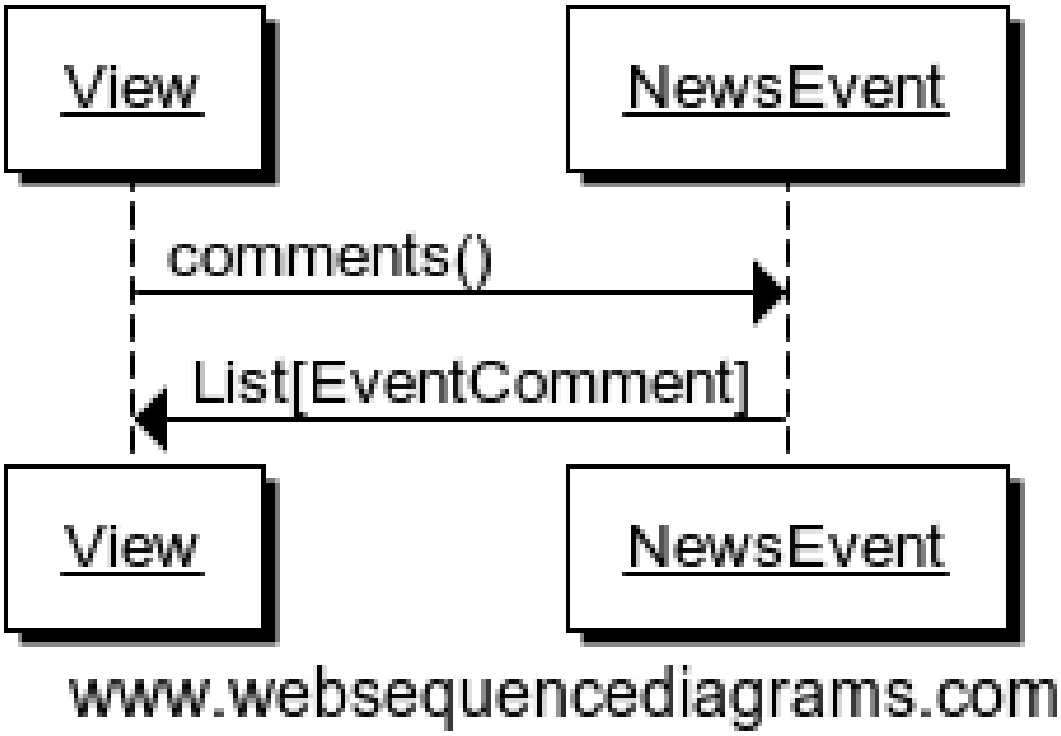
6.5.1 `User.userPostComment`

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



6.5.2 `NewsEvent.comments`

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

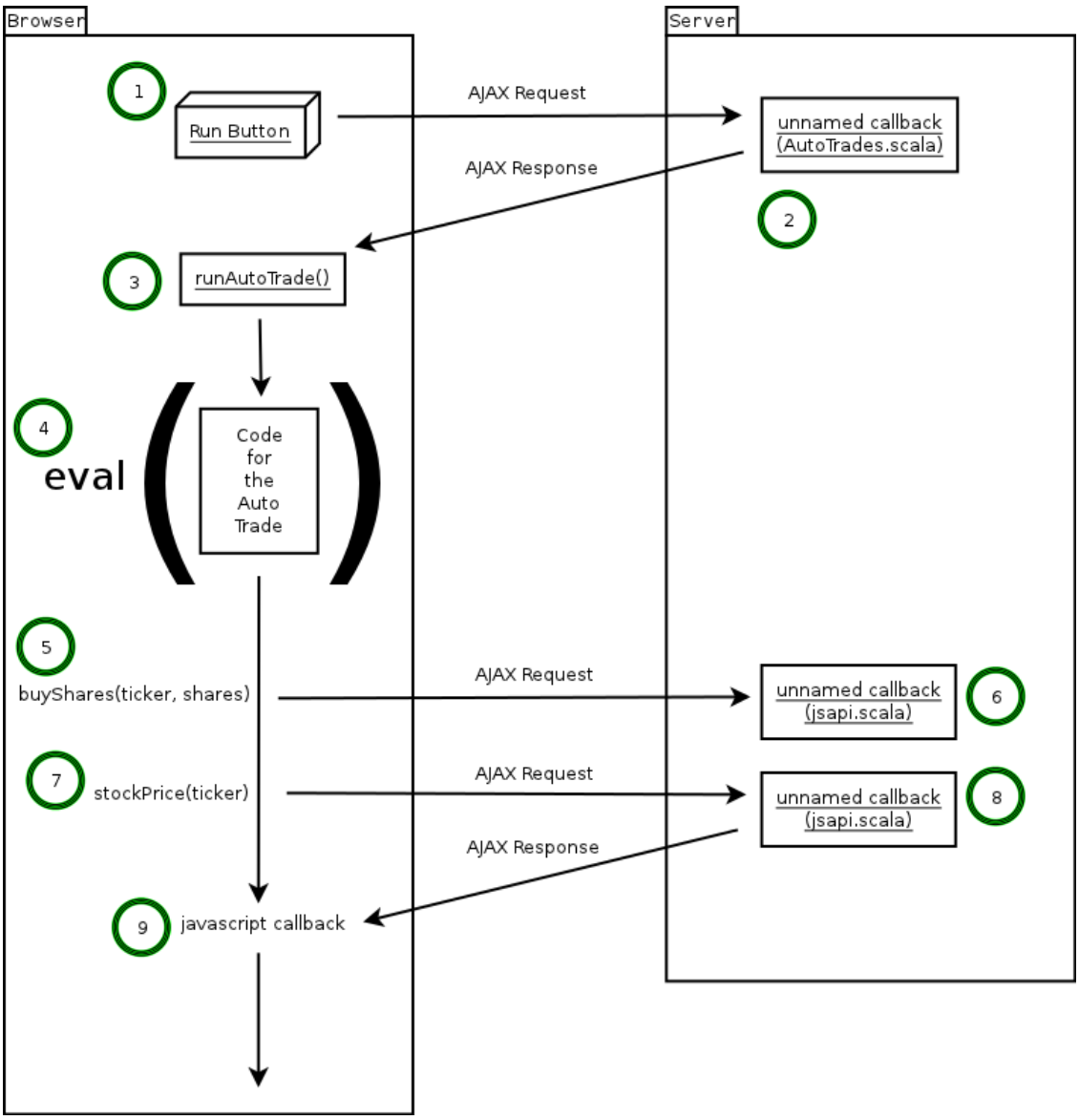


6.6 Auto Trades

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).

6.6.1 Running an Auto Trade

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



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

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

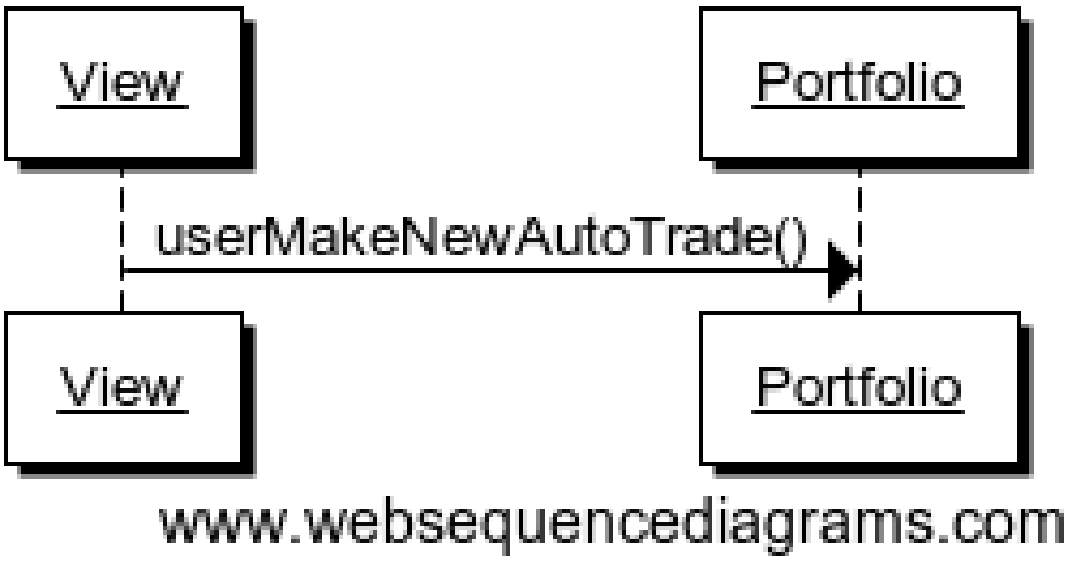
The steps are:

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).

6.6.2 Creating

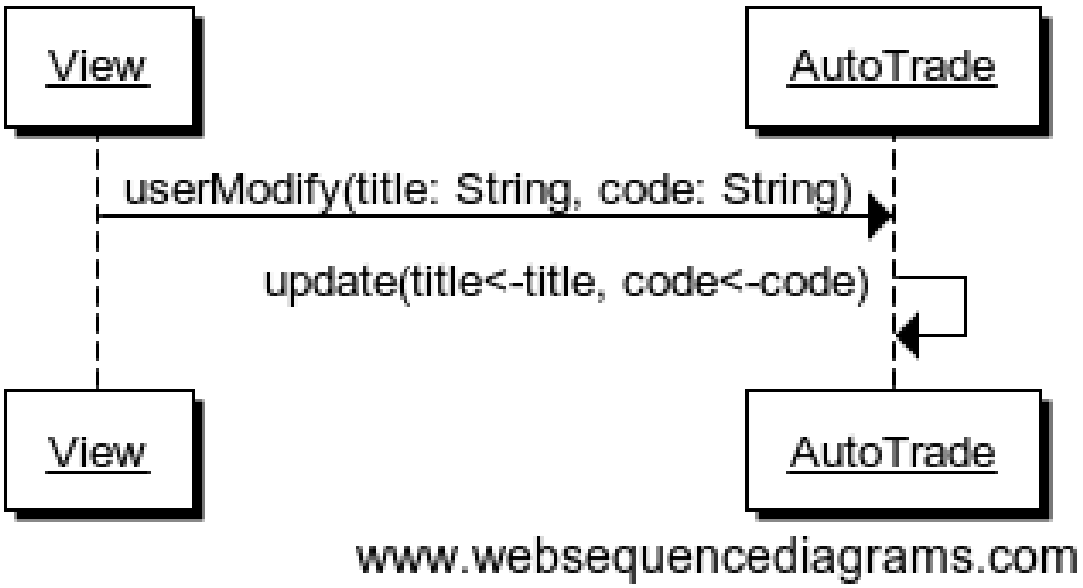
This creates a new (blank) auto trade (model/auto.scala ref_168).



6.6.3 Modifying

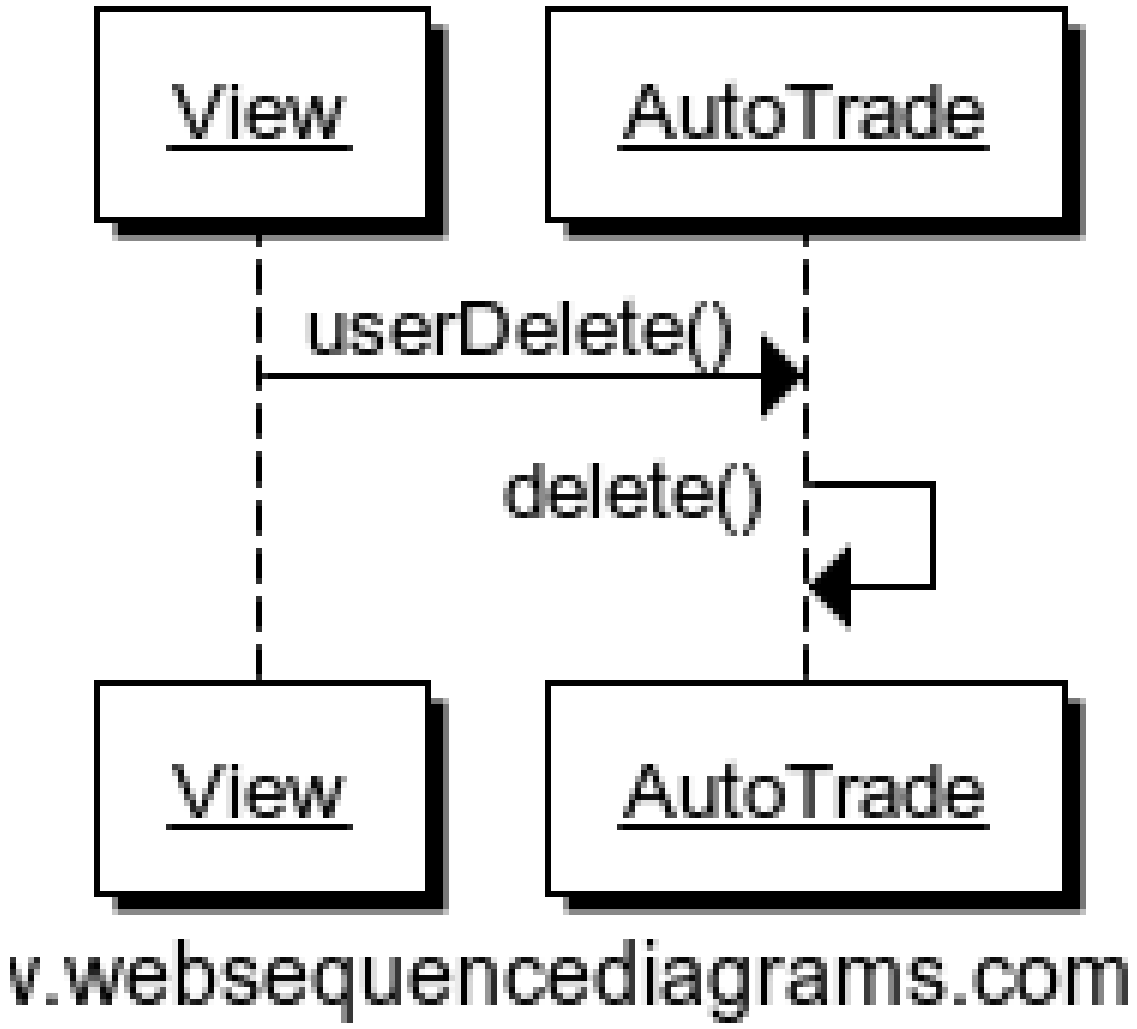
This updates the stored information about an auto-trade (model/auto.scala ref_337).

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



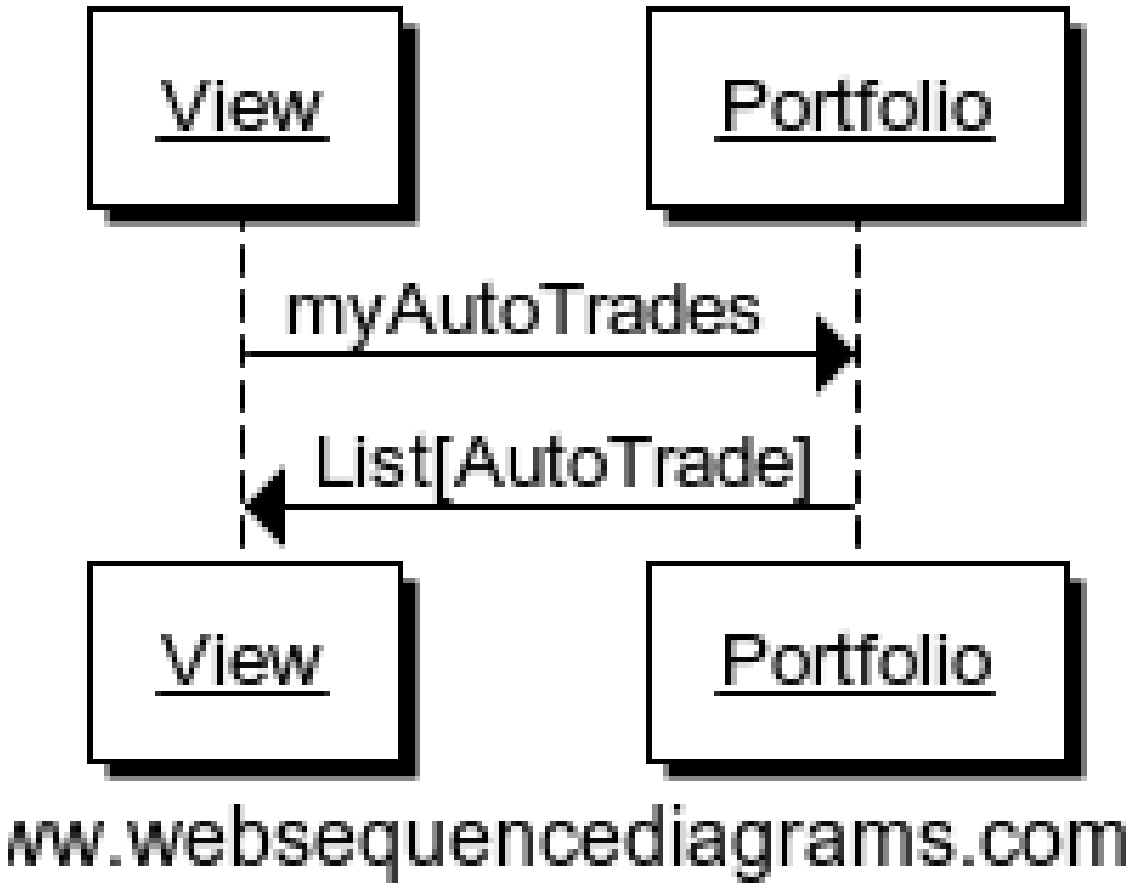
6.6.4 Deleting

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



6.6.5 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).



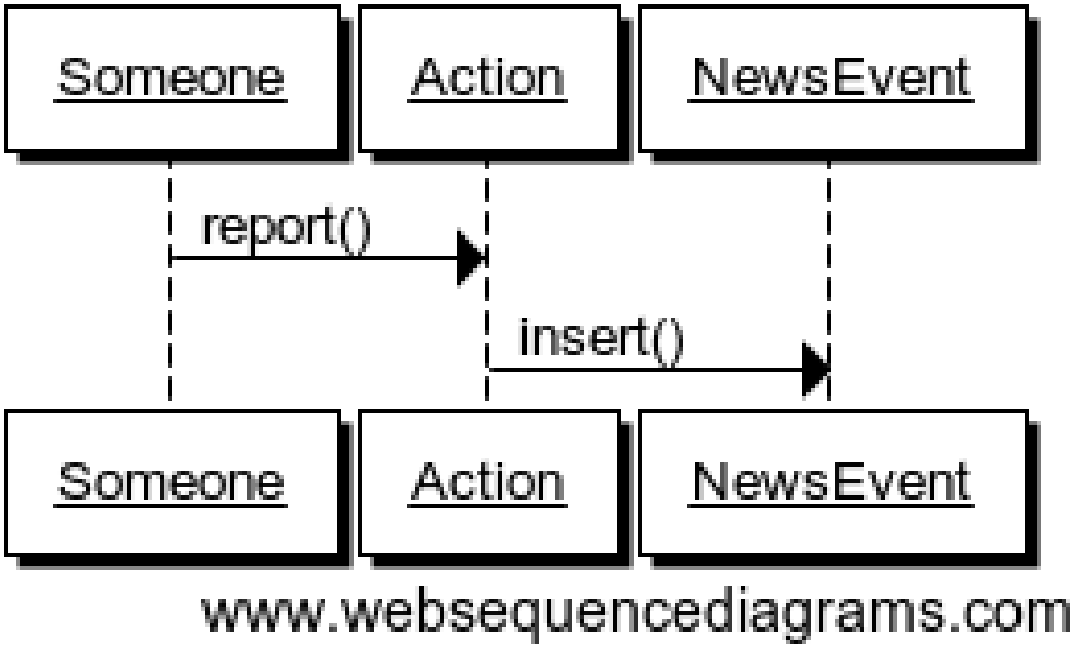
6.7 News

6.7.1 Getting recent news events

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

6.7.2 Reporting an event

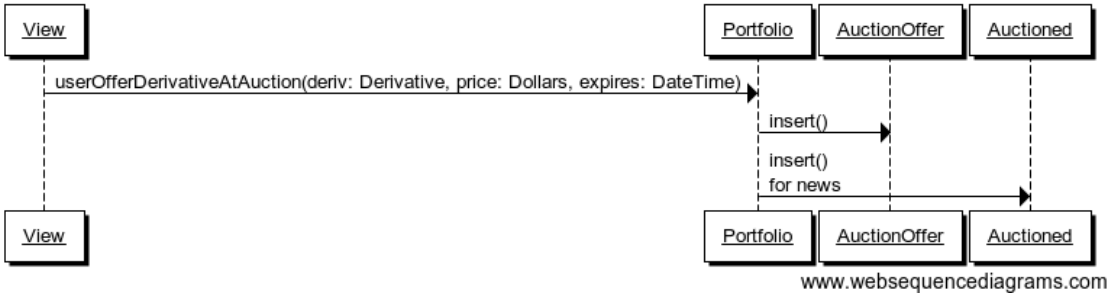
The API into reporting events is the `report()` method in the class `Action`, which takes the action, associates a timestamp with it, and adds it to the list of all events that have occurred (model/news.scala ref_121).



6.8 Auctions

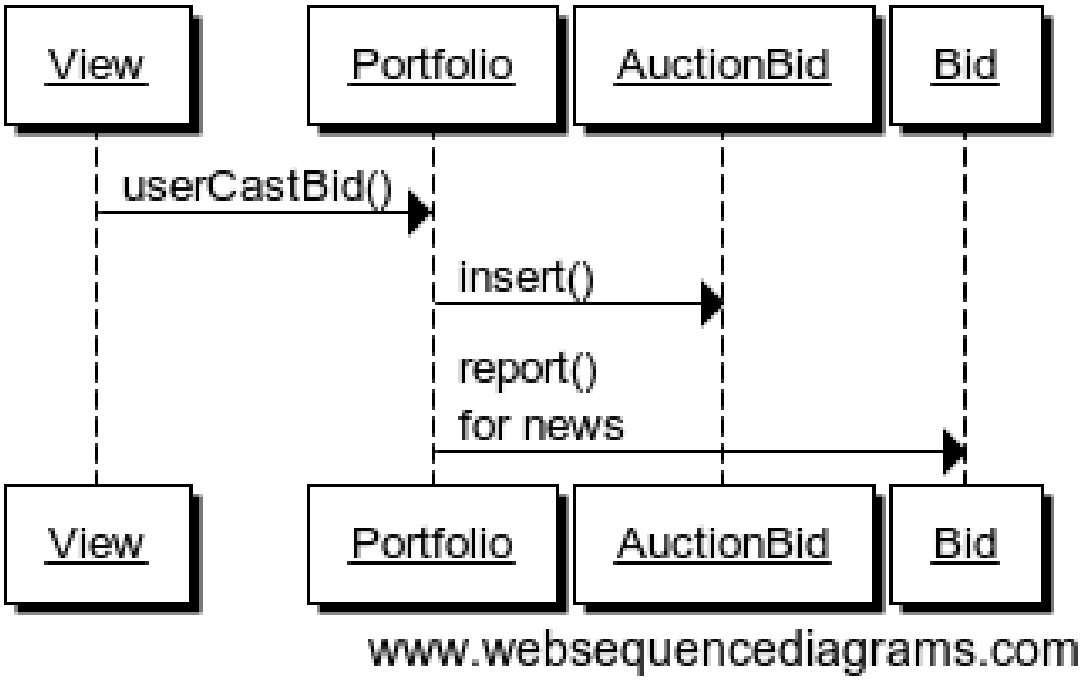
6.8.1 Offering a derivative at auction

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



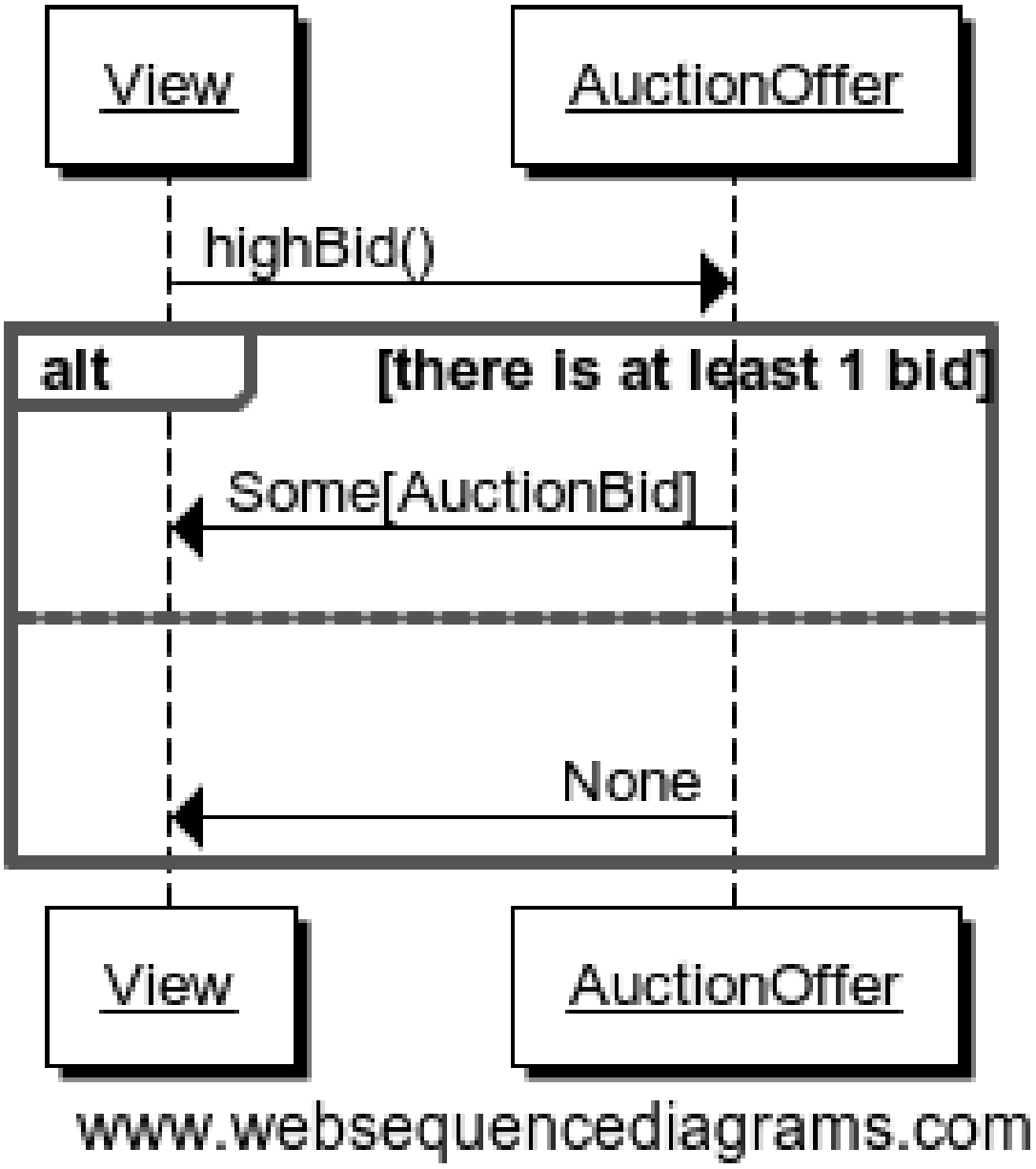
6.8.2 Bidding on an auction

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



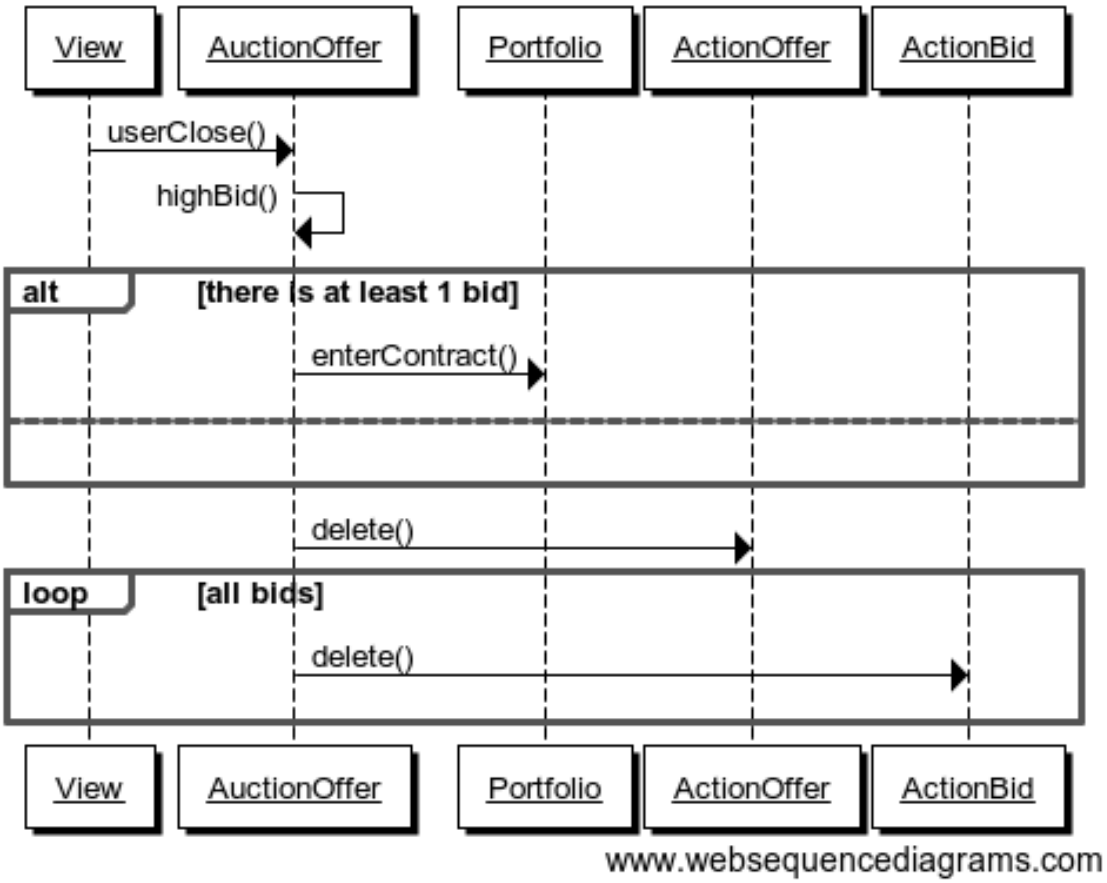
6.8.3 Getting the current high bid

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).



6.8.4 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).



7 Class Diagram and Interface Specification

8 System Architecutre and System Design

8.1 Serializing objects without using reflection

8.1.1 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).

8.1.2 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).

8.1.3 Generic representation of products

We use the same representation of products as Mark Harrah's HLists [HList1], 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
```

8.1.4 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) }
}
```

8.1.5 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).

8.1.6 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).

8.1.7 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.

8.1.8 Putting this all together

Ideally we would like to add the 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).

8.2 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.

8.2.1 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.

8.2.2 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.

8.2.3 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.

8.2.4 Can you assume too little?

Yes, 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.

8.2.5 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).

8.2.6 Improving Lift Forms

8.2.6.1 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/).

8.2.6.2 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).

8.2.6.3 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).

8.2.6.4 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.

9 Algorithms and Data Structures

10 User Interface Design and Implementation

11 Summary of Changes

//TODO

12 Customer Statement of Requirements

..TODO

13 Functional Requirements Specification

..TODO

14 Nonfunctional Requirements

..TODO

15 Effort Estimation using Use Case Points

..TODO

16 Class Diagram and Interface Specification

..TODO

17 Design Patterns

..TODO

18 Object Constraint Language (OCL) Contracts

..TODO

19 System Architecture and System Design

20 Algorithms and Data Structures

..TODO

21 User Interface Design and Implementation

..TODO

22 History of Work & Current Status of Implemenation

22.1 Gantt Chart

include pdf for the *Plan of Work*

22.2 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.

22.3 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

23 Conclusions and Future Work

..TODO

24 References

[ADTs] http://gleichmann.wordpress.com/2011/02/05/functional-scala-algebraic-datatypes-sum-and-product-

[American] http://www.investopedia.com/ask/answers/06/americanvseuropean.asp#axzz1gFsL9Mp8

[Anemic] http://stackoverflow.com/questions/258534/anemic-domain-model-pros-cons

[Applicative1] http://en.wikibooks.org/wiki/Haskell/Applicative_Functors

[Controllers] http://www.aptprocess.com/whitepapers/DomainModelling.pdf

[CurryHoward] http://en.wikibooks.org/wiki/Haskell/The_Curry-Howard_isomorphism

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

[Data] http://en.wikibooks.org/wiki/Haskell/Type_declarations

[DSL] http://c2.com/cgi/wiki?DomainSpecificLanguage Ed. Eric McLaughlin and Mary O'Brien. Sebastopol: O'Reilly, 2006.

[Erasure] http://docs.oracle.com/javase/tutorial/java/generics/erasure.html

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

[HList1] http://apocalisp.wordpress.com/2010/06/08/type-level-programming-in-scala/

[HList] http://apocalisp.wordpress.com/2010/06/08/type-level-programming-in-scala/

[HTTP] http://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

[Implicits] http://lampwww.epfl.ch/~odersky/talks/wg2.8-boston06.pdf

[Inversion] http://martinfowler.com/bliki/InversionOfControl.html

[Iry1] http://james-iry.blogspot.com/2010/08/why-scalas-and-haskells-types-will-save.html

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

[JDBC] http://en.wikipedia.org/wiki/Java_Database_Connectivity

[Jetty1] 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] http://www.scala-lang.org/node/133

[Lift1] http://liftweb.net/

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

[Loop] http://stackoverflow.com/questions/5024148/how-to-iterate-through-a-heterogeneous-recursive-value

[Makers] http://en.wikipedia.org/wiki/Market_maker

[Marsic] Marsic, Ivan. *Software Engineering.* Piscataway: Rutgers University, 2011. PDF. Miles, Russ and Kim Hamilton. *Learning UML 2.0.*

[ML] http://www.standardml.org/Basis/option.html

[Monads1] http://lamp.epfl.ch/~emir/bqbase/2005/01/20/monad.html

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

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

[Option2] http://james-iry.blogspot.com/2010/08/why-scalas-and-haskells-types-will-save.html

[Pollak] http://markmail.org/message/cco7biz2g3jeilg6

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

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

[SICP1] http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-20.html#%_sec_3.1.1

[SideEffects] http://c2.com/cgi/wiki?SideEffect

[Squeryl1] http://squeryl.org/

[Squeryl2] https://github.com/max-l/Squeryl/blob/master/src/main/scala/org/squeryl/internals/PosoMetaData.scala

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

[Traits] http://www.scala-lang.org/node/126

[Typing] http://www.haskell.org/haskellwiki/Typing

[Unit] http://en.wikipedia.org/wiki/Unit_type

[View] http://www.assembla.com/wiki/show/liftweb/View_First

[XML] http://www.scala-lang.org/node/131