Creating the Amlin Meta-Model in ESL

Purpose:

The Amlin project is the first attempt to use the ideology of ESL when constructing the model. As such, implementing ESL libraries will fall to the Amlin developers. As a consequence it is important to separate ESL's general implementation with that which is specific to the Amlin project.

The document proposes how to split up the implementation between abstract ESL classes and Amlin specific instantiations of said abstract classes. To do this, we will explain how each abstract component in ESL relates to the Amlin project. We will go through each ESL component, explaining 1.) how the Amlin project will employ the general features of ESL, and 2.) how the Amlin project will extend those features.

ESL medium-level overview:

We'll begin with a medium-level overview of the ESL class structure. Figure 1 is a UML description of ESL. It is a medium-level UML description, and as such does not contain most of the class variables and methods of each class.

BalanceSheets are at the core of ESL. They hold a set of *Items* which can be either assets or liabilities. *Items* can either be *Goods* or *Contracts*. *Contracts* are both FiniteStateMachines as well as *Items.* It is presumed that all *Agents* in ESL hold *BalanceSheets.*

At times *BalanceSheets* will require manipulation because of a contractual requirements. When contractual obligations require some *Item* to move from one *BalanceSheet* to another, this manipulation is handled via the *ContractHandlers* through the medium of the *Obligation* vernacular. The *Contract* generates an *Obligation* in order to request the manipulation of two *BalanceSheets.* The *Contract* then sends the *Obligation* to the *ContractHandler*. The *ContractHandler* then attempts to manipulate the *BalanceSheets.* The *ContractHandler* then informs the *Contract* whether the *BalanceSheet* transfer was successful. The *Contract* will then handle the repercussions of a success or unsuccessful *Obligation* request.

The last missing piece of ESL is the 'World'. How do *Agents* form and trade *Contracts?* They do this in the *Market.* The *Market* forms the rules/constraints for *Agent* interaction. These rules might include mechanisms for how bid and ask orders are matched, or when and how agents interact.

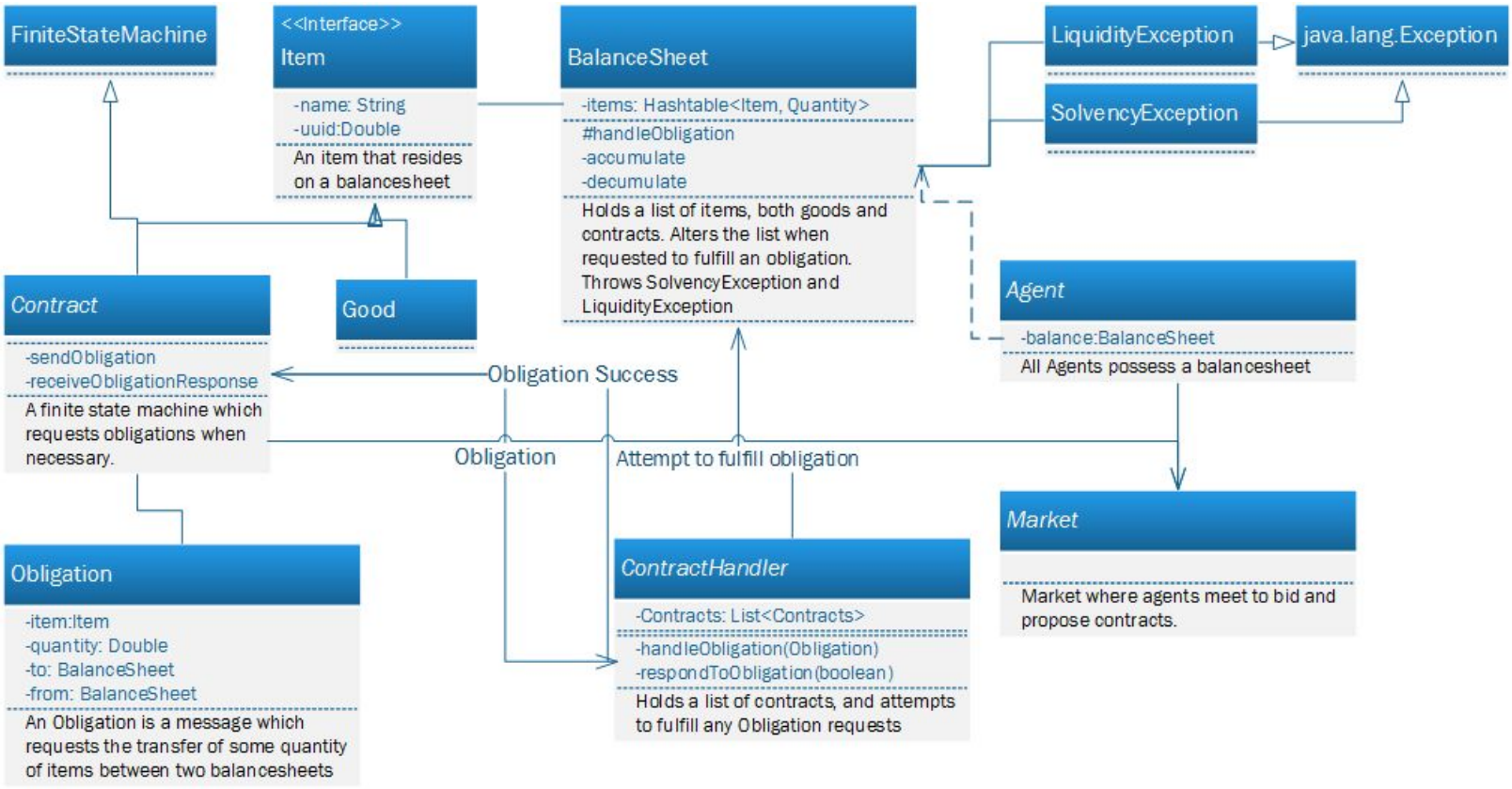


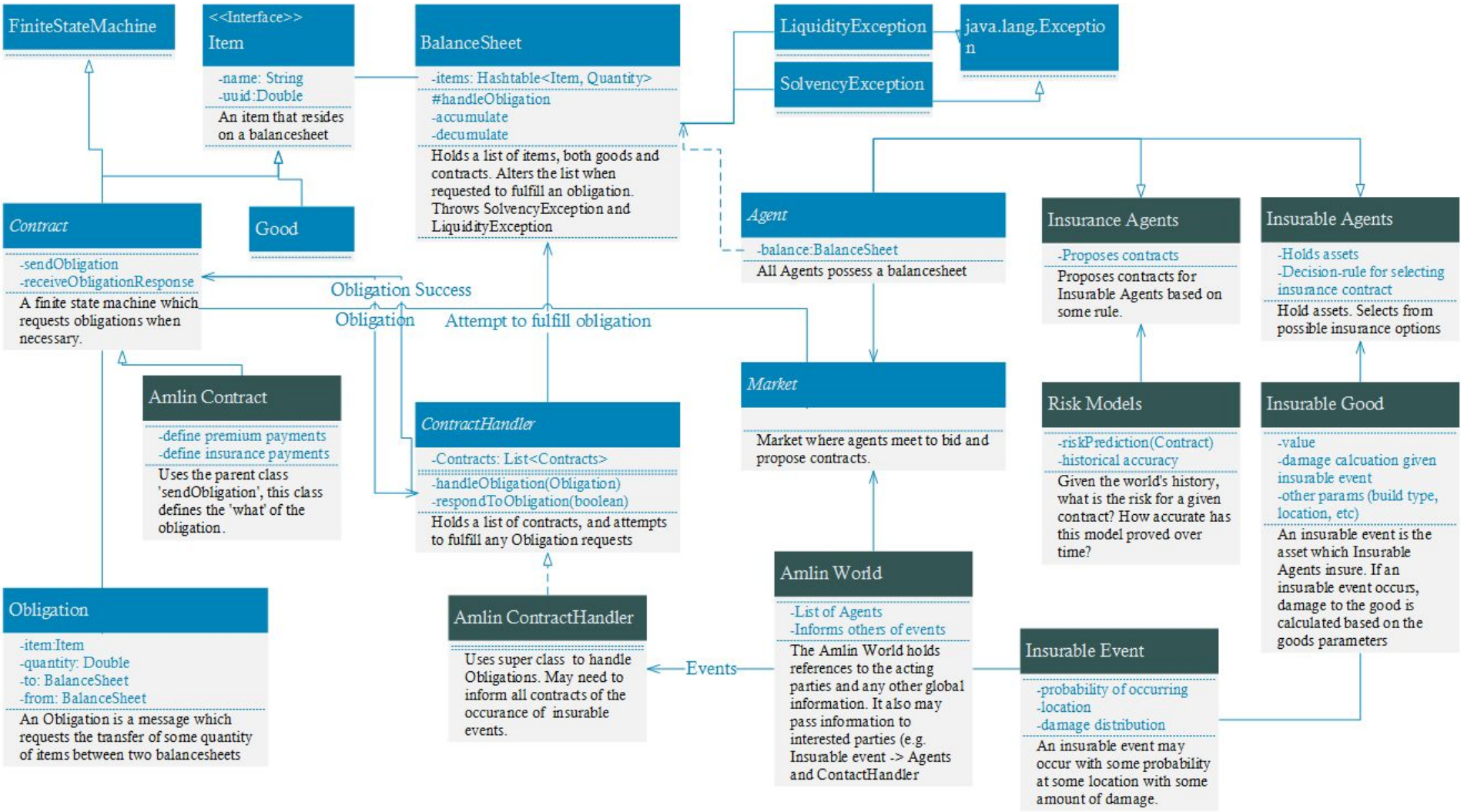
Figure 1: Medium-level UML diagram of ESL

Amlin UML Overview:

In order to complete the Amlin model we will scaffold from the ESL paradigm. Figure 2 shows how the Amlin model will extend from the abstract classes of ESL. Again, this is a medium-level UML document, so many details are excluded. The ESL classes remain the same compared to Figure 1, but several additional Amlin specific classes have been added.

AmlinContracts will employ the *Contract Obligation* vernacular, but will also define premium and insurance payments. *AmlinContractHandlers* will use the *ContractHandler's* functionality for handling *Obligations*, but may also send *Contracts* information about events in the world (e.g. when a catastrophic event occurs).

Two agents will extend *Agent. InsurableAgent* owns *InsurableAssets* and seek out insurance for said asset(s). *InsuranceAgents* represent insurance companies. They use *RiskModels* to create *AmlinContracts* which are then proposed to InsurableAgents in the *Market.* With some probability *InsurableEvents* 1.) occur, 2.) are located, and 3.) cause a certain amount of damage. The Amlin *World* holds references to these agents, passing information back and forth about the world.


FiniteStateMachine
<<Interface>>
Item
-name: String
-uuid:Double
An item that resides on a balancesheet
BalanceSheet
-items: Hashtable<Item, Quantity>
#handleObligation
-accumulate
-decumulate
Holds a list of items, both goods and contracts. Alters the list when requested to fulfill an obligation. Throws SolvencyException and LiquidityException
LiquidityException
java.lang.Exception
SolvencyException
Contract
-sendObligation
-receiveObligationResponse
A finite state machine which requests obligations when necessary.
Good
Agent
-balance:BalanceSheet
All Agents possess a balancesheet
Insurance Agents
-Proposes contracts
Proposes contracts for Insurable Agents based on some rule.
Insurable Agents
-Holds assets
-Decision-rule for selecting insurance contract
Hold assets. Selects from possible insurance options
Obligation Success
Obligation
Attempt to fulfill obligation
Amlin Contract
-define premium payments
-define insurance payments
Uses the parent class 'sendObligation', this class defines the 'what' of the obligation.
ContractHandler
-Contracts: List<Contracts>
-handleObligation(Obligation)
-respondToObligation(boolean)
Holds a list of contracts, and attempts to fulfill any Obligation requests
Market
Market where agents meet to bid and propose contracts.
Risk Models
-riskPrediction(Contract)
-historical accuracy
Given the world's history, what is the risk for a given contract? How accurate has this model proved over time?
Insurable Good
-value
-damage calcuation given insurable event
-other params (build type, location, etc)
An insurable event is the asset which Insurable Agents insure. If an insurable event occurs, damage to the good is calculated based on the goods parameters
Obligation
-item:Item
-quantity: Double
-to: BalanceSheet
-from: BalanceSheet
An Obligation is a message which requests the transfer of some quantity of items between two balancesheets
Amlin ContractHandler
Uses super class to handle Obligations. May need to inform all contracts of the occurance of insurable events.
Events
Amlin World
-List of Agents
-Informs others of events
The Amlin World holds references to the acting parties and any other global information. It also may pass information to interested parties (e.g. Insurable event -> Agents and ContactHandler
Insurable Event
-probability of occurring
-location
-damage distribution
An insurable event may occur with some probability at some location with some amount of damage.

The Details:
The rest of the document describes in more detail how the ESL code should be separated from the specific Amlin functionality.

Balancesheets:

ESL Balancesheets:

The *BalanceSheets* class needs to be re-written in Java 8. *BalanceSheets* accept objects of type *Item*. *Items* can be *Goods* or *Contracts*. *Balancesheets* contain several methods for manipulating the balancesheet, but these methods are private. *BalanceSheets* are interacted with through the *Obligation* vernacular. A more detailed description of the *Balancesheet* class can be found here.

Amlin BalanceSheets:

No additional adjustments should be required to meet the needs of the Amlin model, since the balancesheet implementation is agnostic on the data it holds. Having said that, for Amlin purposes, balancesheets will likely just deal with:

- Insurable assets (e.g. houses in Anders' model)
- Money
- Insurance contracts
- Reinsurance contracts (which may be identical in implementation to Insurance contracts)

Obligation:

ESL Obligations:

Obligations allow a contract to request that an *Item* is transferred between two *BalanceSheets.* The request has four parameters:

- to: Agent - The agent who is receiving the transfer
- from: Agent - The agent who is passing the Item.
- item: Item - The item passing between balancesheets
- quantity: Double - The quantity of the Item passing between balancesheets.

Contracts will create *Obligations* when they are required to manipulate two *Balancesheets.*

Amlin Obligations:

The ESL obligation should be sufficient for the Amlin project. As such, no additional functionality is required.

Contracts:

Abstract Contracts:

Contracts are finite-state machines. As such, there is not a lot that is implemented in the abstract class. Rather, it is a pattern of development which is enforced (i.e. the finite-state machine construct).

Abstract Contracts - requesting payments:

Having said that, the abstract *Contract* will force the use of the *Obligation* vernacular to request payments. When the *Contract* needs to transfer money between the insurer and insuree, the request will run through the *ContractHandler* in the form of an *Obligation*. The *ContractHandler* will then inform the *Contract* whether the payment has been successfully conducted. The *Contract* will react accordingly (e.g. go into default if the payment was unsuccessful)

Amlin Contracts:

An insurance contract is composed of several parts:

- Premium payments - these occur at least once and then potentially at regular intervals. The party who is insured pays the insurer.
- Insurance payment - If an outside event occurs which is covered by the contract, then payment may be initiated. Payments have the following parameters
    - Damage Threshold - This is the amount of damage incurred before payment is required.
    - Damage Limit - This is the maximum amount of damage covered by the contract.
    - Percentage - This is the percent of the damage within the limit which the insurer agrees to pay.
- Link to reInsurance - A contract's damage limit and / or percentage can be updated by reinsuring the initial contract. In this case damage which was initially within the scope of the contract is sent to another insurer who has reinsured the original contract.

Amlin events:

A *Contract* will need to be made aware when an *InsurableEvent* has occurred. We should consider the form this input should take. Should contracts be passed all events in the world and internally test whether an insurance payment is required? This would potentially involve catastrophic insurance being informed of auto accidents. However, the alternative forces other objects to possess information about the *Contracts*, eroding encapsulation.

Contract-Handler:

The *ContractHandler* is in charge of handling the interaction between *BalanceSheets* and *Contracts*. It processes the *Obligation* payment request from a *Contract* and attempts to fulfill the request by manipulating the necessary *BalanceSheets*.

Abstract ContractHandler:

At the highest level, the abstract Contract-Handler will only fulfill Obligations. This functionality should remain constant across almost all economic models, so it can be concretely instantiated in the abstract class, allowing subclasses of *ContractHandler* to reuse the functionality.

Amlin ContractHandler:

For the Amlin project we will likely need to extend the abstract *ContractHandler*. Perhaps this will include passing information to the *Contracts* regarding the *InsurableEvents* which are occurring in the world? Again, some more thought needs to be put into how Contracts are informed about the world whilst maintaining as much encapsulation as possible.

The Agents:

Abstract Agents:

The abstract *Agents* class has a *BalanceSheet*. All agents possess *BalanceSheets*.

Further, given that we're using the Mason scheduler, we need to link *Agents* to the interface of Agents in Mason. To do this we just need to implement the Steppable Interface. **However, no abstract class should directly implement any Mason Interface.** We should use wrapper/anonymous classes to call any function in the *Agents*. This way we decouple scheduling from the implementation of any of the classes.

Last, it is possible we'll want to include requirements for how agents interact with a market. This is a question for David. Is there a messaging vernacular for how agents communicate with markets? Is this general enough that we can require Agents to implement their side of the communication?

Amlin Agents

Agents in Amlin will consist of:

- *InsurableAgents:* Asset holders who desire insurance. Behavioral functionality is needed for the agent to decide between multiple insurance offers.
- *InsuranceAgents* (i.e. Insurance companies). Functionality will need to be written in order to generate insurance *Contracts*. The premium of contracts may be based on their use of
    - *RiskModels*
    - Solvency II regulations

    - their current portfolio
    - loyalty to historic customers
    - etc…
- Reinsurance companies...though potentially the implementation will be much the same as insurance companies.

Market

Abstract Market:
I've haven't been concentrating on understanding the market piece. David, is there an abstract market class? A matching function which can be used for Bid and Ask orders?

Amlin Market:

In the Amlin Market, we will want insurance companies, asset holders, and (potentially) reinsurance companies. The flowchart of the market interaction will be:

1. An asset holder requests their asset to be insured to a certain level.
2. Insurance companies offer bids. In the simplest instantiation the insurance companies will offer bids to cover the asset holder's entire request...though perhaps at some point some portion of the request may be bid on.
3. Asset holders select the contract they desire.

It should be noted that a matching algorithm may not be required, especially if each asset is unique compared to other agent's assets. If, however, like in Anders' model, every asset holder owns a cloned house, then some contract matching criteria may be valuable.

The World - Additional Players

The rest of the implementation is independent of the abstract economic model platform. These players include:

- *InsurableAssets*: The first question is: What type of assets are we modelling? The second questions is, what are the attributes of the asset? Potential characteristics of interest include:
    - Value
    - Geographical location
    - Assorted attributes which may affect risk to particular events
        - Construction Type
        - Correlation to other assets

  The InsurableAsset should have a function for how to calculate damage to itself given some *InsurableEvent.*

- *RiskModels*: Risk models predict the probability distribution for an *InsurableEvent* occurring. It can be used in tandem with an *InsurableAsset* to predict the likelihood of having to payout for the destruction of the asset.

- *InsurableEvent* - An event which harms *InsurableAssets.* With some probability *InsurableEvents* 1.) occur, 2.) are located, and 3.) cause a certain amount of damage. *RiskModels* are tied to *InsurableEvents* in that *RiskModels* are attempting to predict the underlying probability distributions of the events.