**Software Engineering in the Java based Multi-Agent System**

**Design patterns**

**State:** Allows an object to change its behaviour depending on its internal state by effectively changing its class.

The state design pattern has been used in many areas of my code. One example, the *SmartMeterAgentBody* class *brainHandler* attribute. The *brainHandler* attribute can be in one of two states – *NormalBrainHandler* or *IPBrainHandler*. This depends on the *Actuator* given to the agent at instantiation. One state allows the agent to handle *IPCommunicationActions* the other allows any non-IP related action e.g. *RecordAction*.

**Observer/Observable:** The observer design pattern is used for event parsing between objects without explicit method calls. In java the *update* and *notifyObervers* methods are used to do this.

A custom version of the java implementation is used in GAWL for event parsing between agent parts, agents and environments, and environments and their associated physics. For of these concepts there is one or more classes in the system that represent them e.g. *NationalGridUniverse, SmartMeterAgentBody, HouseEnvironmentPhysics* etc.

**Factory:** The factory design pattern provides an abstraction from object creation. No knowledge of the creation of the object is required, only the method in the factory for the desired object.

The *HouseModelFactory* class is one example of the factory design pattern in the system. It abstracts from house model creation, only an error term is required for creation.

**Singleton:** The singleton pattern is used to ensure that only one instance of a class can be created. This instance can be accessed in a static way, the class is responsible for the creation of this single instance.

The HouseModelFactory class is one example of the singleton design pattern. Factories are often singletons as there should only ever be one instance.

**Check style**

During the Java development process the google coding standards/check style was used to keep the formatting of code consistent and readable and also to make it easier to follow proper java coding standards.

**Test Strategies and TDD**

TDD is a software development strategy in which the requirements of the system are broken up into essentially atomic tests. The tests are written and then code is written so that tests succeed. All tests are re-run each time the system is tested. The system is built up in this way so that (hopefully) by the end everything will work. In java JUnit is used for TDD, JUnit and a TDD strategy was used when testing some important utility classes such as *TestArgumentUltilities*, *TestTimeDateTracker*, and *TestMathUtilities* all of which can be found in the *test* package.

TDD was not a development strategy that would provide any advantage in the rest of the project. It becomes difficult to create unit tests for classes that interact heavily with one another and in a multi-threaded system it was nearly impossible. Instead, I used an integration based test strategy, testing the interacting between Observer/Observable pairs in a chain using simple print statements. For example to test event parsing between all parts of an agent – place a meaningful print statement in each *update* method and evaluate their order/correctness manually.

**UML**

**Revision Control**