SWEN30006 Report - Project 1 Automail

Part 1: From Problem Domain to Domain Model

To ensure that we had a solid understanding of the domain, we constructed a partial domain model (see Figure 1) to formally capture the business requirements of the latest version of Automail.

Part 2: From Domain Model to Design Model

Our partial design diagram (see Figure 2) reflects the actual implementation of the charging system. The sections below give more detailed accounts of the modifications and their justifications of design.

2.1. Service fee lookup

Rather than having our system communicate directly with the WifiModem class from *WifiModem.jar* (which we have no control over), we introduce an interface *IServiceFeeApi,* and a canonical implementation of that interface, *WifiModemAdapter*. *WifiModemAdapter* follows the indirection pattern - it is an intermediate object which reduces coupling between the `automail` package and WifiModem.jar. The interface *IServiceFeeApi* is an example of polymorphism - in tests we pass in a *MockServiceFeeApi* instance (instead of a *WifiModemAdapter)* which does not connect to the WifiModem.

2.2. Charge calculation

We created a new class, *ChargeCalculator* for the sole purpose of calculating charges for mail items, using service fees returned from an *IServiceFeeApi* instance. Separating this functionality into its own software object will help maintain high cohesiveness, and is a form of pure fabrication.

We also created a nested class, *ServiceFeeLookupResult,* within *ChargeCalculator*, to cache lookup results in case of future lookup failures (if the lookup failed and no cached lookup results available we will not charge the tenant service fee), and to ensure that we only ever perform one service fee lookup per mail item delivery (and thus only charge the tenant one lookup fee per delivery, as mentioned in the spec).

We extracted the *calcCharge()* method to an interface, *IChargeCalculator,* decoupling the rest of Automail from the charge formula given in the spec, and can swap in a new *IChargeCalculator* implementation (using polymorphism), if required. For example, if the company wished to factor item weight into the activity cost (as mentioned in the spec).

We store all statistics in *ChargeStats*, this is another example of how we maintain a high cohesion and low coupling. It is also a form of pure fabrication as it is not present in the Domain Model. It also allows for the addition of an overridden toString for formatting the log output. We also applied the creator pattern and information expert pattern when deciding *ChargeCalculator* as the creator of *ChargeStats*.

2.3 MailPool refactor and priority scheduling

We increased the cohesion of *automail.MailPool* by refactoring it to remove the list of waiting robots. The MailPool should only worry about storing/ordering incoming mail and loading it onto robots. It shouldn't be concerned with keeping track of waiting robots.

We converted the *ArrayList* of mail items to a *TreeSet*, which is a more efficient priority queue than repeatedly re-sorting a list, and cleaner too. After this refactoring, dispatching priority mail first was easy, we modified MailPool. We had overridden Item's *compareTo()* method to order priority items first.

2.4. Extracting interfaces

Using the protected variation pattern, we extracted several new interfaces: *IChargeCalculator*, *IClock*, *IMailPayload*, *IRobotStateObserver*, *IMailPoolObserver* - to allow for alternate implementations in the future, and to support unit testing of individual components (such as unit-testing MailPool with a mock implementation of *IChargeCalculator*).

In particular, *simulation.SimulationOutput* implements *automail.IMailPayload*, *automail.IRobotStateObserver*, and *automail.IMailPoolObserver*. It prints out specific log lines for key events (mail delivery, robot state change, and mail pool updates respectively), to match the expected output. We chose to keep *SimulationOutput* separate from *Simulation* so that *SimulationOutput* was solely concerned with output, increasing cohesion.

We also decreased coupling by removing the *Robot's* dependency on *MailPool* (we modified *Robot.operate()* to return undelivered MailItems instead). The dependency on the global *Clock.Time()* also disappeared by extracting out *IRobotStateObserver.*

2.5. Avoiding statics, remove cyclic dependency between `automail` and `simulation`

The *Building* and *Clock* classes were full of static state, making it difficult to use them in unit tests. We refactored *Building* and *Clock* to get rid of static state - instead we construct a single instance of each class inside *Simulation*. While this appeared to increase coupling between classes (e.g.: introduced extra constructor parameters), the dependency on *Building* and *Clock* was still previously there, just hidden deep inside certain methods instead (e.g.: inside *Robot.operate()*).

We also moved *Building*, *IMailDelivery* (renamed *IMailPayload*) into the `automail` package, to get rid of a mutual dependency between the `automail` and `simulation` packages. Now `automail` can be used without the `simulation` package.

Finally, we decoupled the Simulation from *System.out* and the "*automail.properties"* file. The Simulation constructor now takes an *InputStream* to read properties from, and a *PrintStream* to write log lines to. This allows us to perform simulation tests which read/write to/from a String (see *simulation.SimulationTest*).

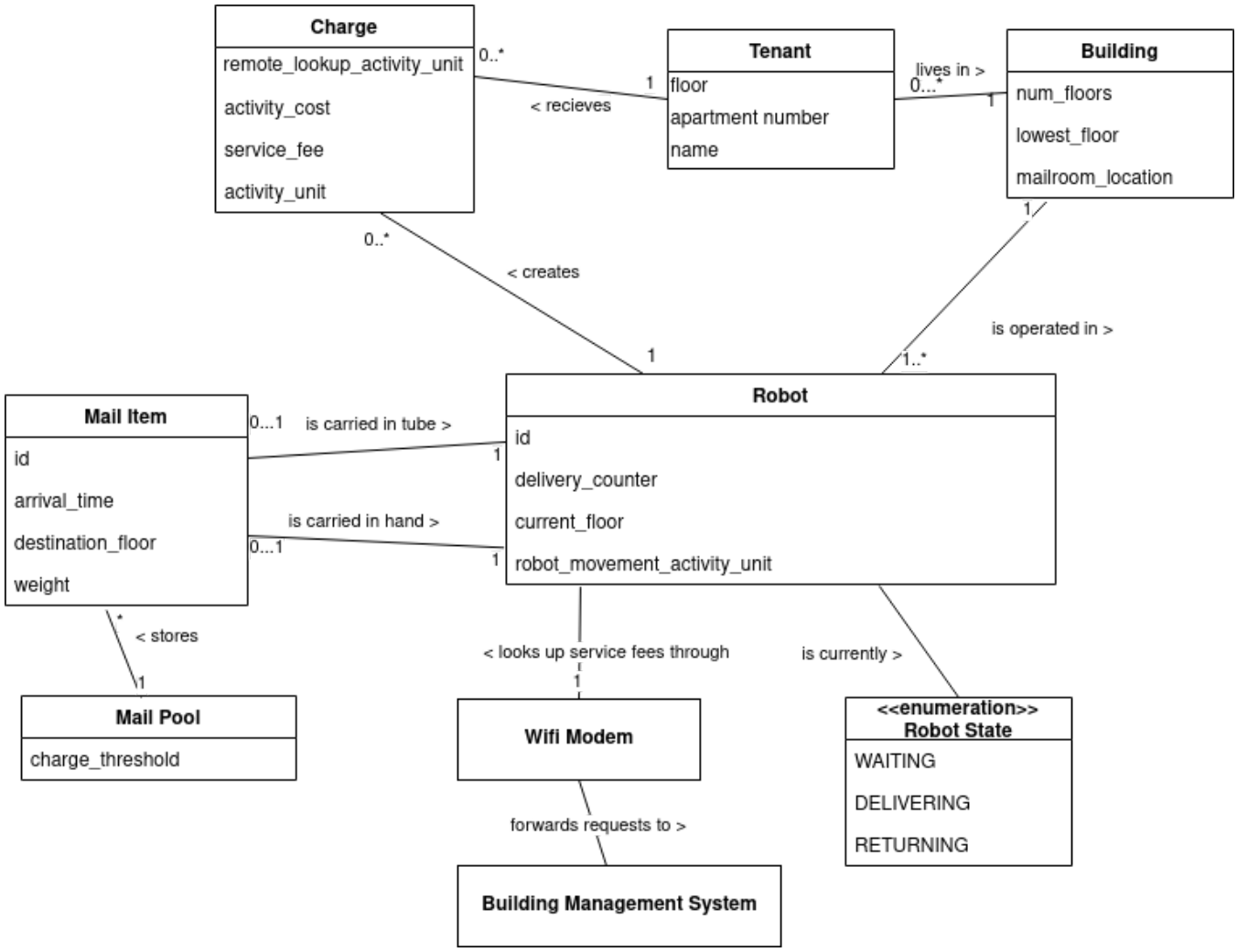

Figure 1: Partial domain diagram describing the Automail system based on business requirements in the project specifications.

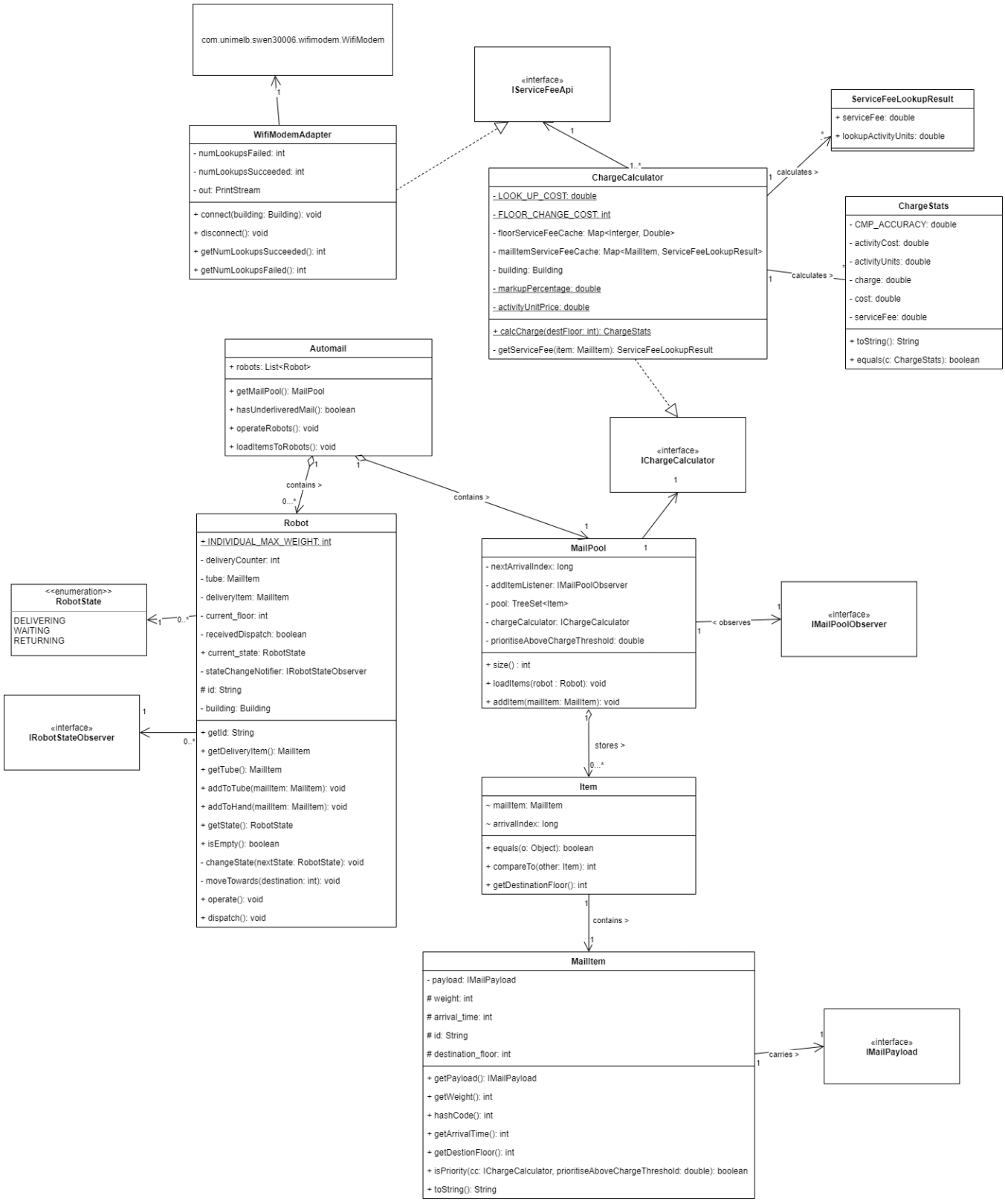

Figure 2: a Design Class Diagram of the Automail system classes and elements. Note that it is a partial diagram that focuses on the implementation of the charge capability.