Patrick Coyle & Blake Lawrence

COMP 3721 – Object Oriented Methods and Design

<u>Design Document</u>

To begin, we made all our instance variables private, opting for getters and setters, to ensure that the program doesn't introduce coupling, and so that the code remains cleaner. We then decided that Lollipop and MusicSlippers should be their own interfaces, since they each provide different functionality. We dropped the "Fish" from FishLollipop since the flavours imply whether the lollipop is fish flavored or not, and allows us to open up the same interface for many different kinds of lollipops. For musical slippers, we figured they could be their own interface. We considered whether they should perhaps inherit something from normal slippers, but ultimately decided that this could easily be achieved through adding a slipper interface and employing multiple inheritance, and since we have no information on the slippers themselves at this point, left it as a possible area to expand. Finally, we thought that lollipops and musicslippers shared a commonality: they are both products at the store. We thus deemed it necessary to also create a Product Abstract Class. The reason we left it as an abstract class, is because product should be the highest abstraction you can get at a store, and thus every product you can come up with will have the getCost method and cost variables, for example. To cut back on code, abstract class was the best choice. Thus, every product we create must extend this class. Finally, for the transactional aspect of the software, we decided we would have a "cart" object, not unlike the carts you see during e-commerce. In this cart, products can be added to the list, and the total cost can be returned. The customer would contain this cart object, and then request a checkout by creating a "clerk" object, which essentially decides whether the transaction is legal (Does the customer have enough money, is it shipping to a legal country, etc.). Now that this was figured out, we decided we must next decide what kind of design patterns could be applied to this code. The design pattern that jumped out to me first was the Factory Class design. We could easily create a factory for slippers as well as lollipops. Further, we could create an abstract factory to contain these factories. And finally, each factory class itself could be a singleton, since we do not need to create more than one. The Abstract factory made sense, since we are indeed creating a “Suite” of different “products”. In addition, this allows us to hide from the rest of the program the way the products are implemented, therefore enforcing privacy. Then, we make the abstract factory contain the factory methods for each product, in this case, lollipops and musical slippers. These factories are useful, since we can easily extend them by adding more of the same type of product, without modifying too much code, and allows programs to utilize the pattern without needing to know what the concrete class is. Finally, singleton was just a very logical choice for the factory methods: We do not ever need more than one factory method, so to save memory and possible confusion; it helps to make these singletons.