Design Patterns

In this recitation, you will model several problem domains and discover design patterns that solve common design problems in those domains. This recitation has two parts, one part for the decorator pattern and one part for the adapter pattern. For each part we will describe a problem domain and present a possible solution to the problem. We will then present a different problem domain for which you will construct a possible solution, producing an object model. We will then "discover" the design pattern together by examining the similarities between your solution and our solution, even though our solutions are for different problem domains. By the end of the recitation, you should be able to generalize the key properties and identify the quality attributes achieved by the decorator and adapter patterns.

Discovering the decorator pattern

1. Recall the coffee shop from Homework 2:

   > Each beverage has a recipe and cost associated with it. The cost changes depending on the type of beverage (i.e coffee or tea) and the recipe changes if the beverage contains milk. Beverages can contain any arbitrary number of additional ingredients such as ginger, milk, whipped cream, etc. Additional ingredients add to the cost of the beverage.

   In Homework 2 you implemented one possible solution. See the object model for that solution in `coffee_shop.pdf`.

2. Create an object model for the following problem domain:

   > Consider a logging system for printing a given message. There should be a basic logger class, but different clients have different, additional requirements. Clients have requested features such as (1) log messages including a header with a timestamp, (2) log messages with a footer with additional debugging information, and (3) log messages in a different color. Some clients want various combinations of the above features, such as including a timestamp header with color printing or including both the footer and header. Your design should be able to easily accommodate any request.

3. Consider the similarities between the coffee shop and the logger problems. In what ways is your logger object model similar to the coffee shop object model?

Discovering the adapter pattern

1. Consider the following problem:

   > You have an existing List implementation with `get`, `add`, and `remove` methods (based on the list position). You need, however, a Set class. Because you are constrained in time (and cannot use standard libraries?!?) you wish to reuse your List class for your Set implementation.

   If efficiency is not required then the solution is straightforward: your Set can simply store data in a List and (inefficiently) search the list to support the standard Set operations. The UML class diagram for this solution is similarly simple; see `list_to_set.pdf` for an example.

2. Create an object model for the following problem domain:

   > Consider an email client which can import a user's emails from any type of account (e.g. Gmail, Yahoo mail, Hotmail). Each of these existing email services provides a class for retrieving the user's emails, with methods like `getAllGmails()` and `hotmailGetAll()`. The existing email services, however, return email objects that are different from each other; they

all fundamentally contain the same data (subject, sender, body, timestamp, etc.) but have different field names and access methods. For example, Gmail objects may have the fields `subjectText`, `senderAddress`, `bodyText`, and `timeStamp`, while Hotmail objects contain the fields `subject`, `sender`, `body`, and `time`. The email client needs a single, common interface for its internal email representation and, ideally, a single interface for getting emails from the email services.

3. Consider the similarities between the List/Set and the email client problems. In what ways is your email client object model similar to the List/Set object model?