Inf2-SEPP 2024-25

Tutorial 4 (Week 6)

Notes on Answers

1 The Observer pattern

As with all design patterns, a key point to appreciate is the problem that this pattern solves. In brief, it is useful in the situation when a *subject* object needs to notify one or more *observer* objects of particular events, e.g. changes in its state. In such a situation, the simplest way to enable such notifications to be sent to the observer objects is to have directed associations from the subject's class to each of the observers' classes. However, this introduces strong coupling, which is often undesirable as explained in class. One scenario in which weak coupling is desirable is when we wish to design the subject class with no knowledge of the particular observer classes it may eventually send notifications to. This is the case with classes such as `JButton` which represent GUI components that generate events on user actions. The Observer pattern explains how to achieve weak coupling by introducing a minimally specified Observer interface/abstract class that all Observer classes realise/inherit from.

See Figure 1 for a class diagram for the `JButton` example. See Figure 2 for a sequence diagram, just showing the constructor and operation calls relevant to the Observer pattern.

Answers to questions raised on the tutorial sheet:

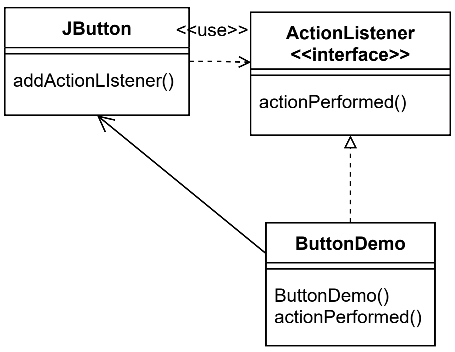

Figure 1: Button demo class diagram

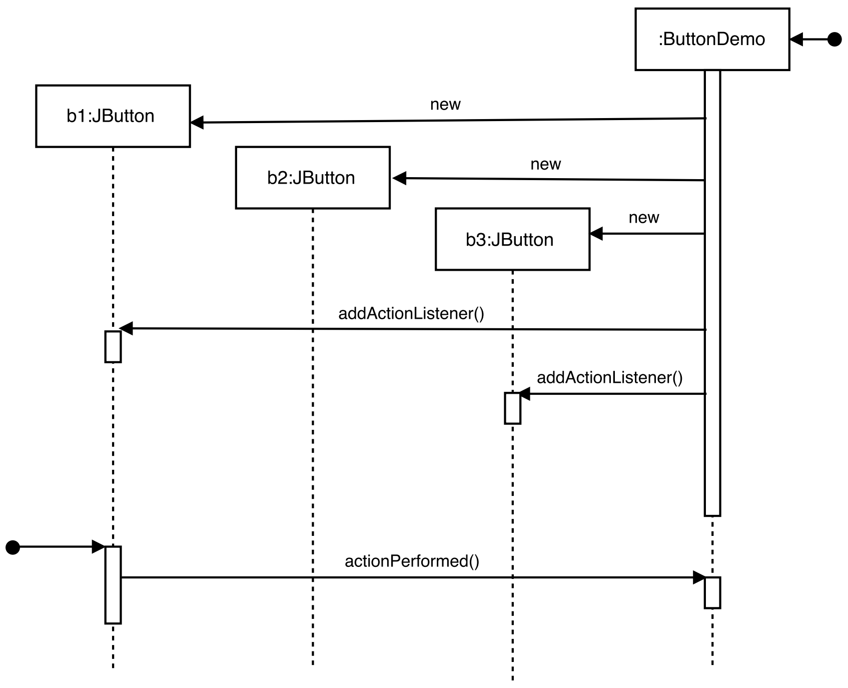

Figure 2: Button demo sequence diagram

1. There is a directed association from the JButton object to the ActionListener interface.

2. In the JButton demo, the actionPerformed() method of the ActionListener interface carries an ActionEvent object. Also the b1, b2 and b3 fields provide a way for a ButtonDemo object to directly interrogate the states of JButton objects.

3. Yes. The association from the ButtonDemo class to the JButton class. More generally, there might be method in some third class that is responsible for registering action listeners such as ButtonDemo with the JButton objects.

4. Yes. The obvious one is that in the lecture and associated reading the Observer is an abstract class, whereas in the implementation it is an interface. Conceptually, even using an interface for the Observer concept is somewhat heavy weight, as at heart what is registered with the Subject is a function (often called a *call-back* function because of the way in which it is used).

2 Pattern identification 1

1. Having operations for the mathematical calculations and their undo operations in the classes `A` and `B` themselves means that the classes would be responsible for how the operations are performed, reducing cohesion. Moreover, it leads to a lot of duplicated functionality between the classes, and possibly making mistakes when duplicating functionality. Having them instead in a separate class `Calculator` reduces this duplication,

and increases cohesion. However, it also increases coupling because of the additional associations with the `Calculator` class. Updating the operations in the `Calculator` class (e.g. to receive an additional argument for the number of decimals of the result), or replacing the `Calculator` class with several classes (e.g. for trigonometry calculators, logic calculators etc.) would require updates to classes `A` and `B` as well (reduced maintainability).

2. The Command pattern can help encapsulate the different types of mathematical operations and their undo operations into separate classes, which can each be set up to work with certain calculator class(es). Objects of `A` and `B` would not need to know how to perform the operations or which calculator class(es) to use, they could just have access to a collection of Command objects which they could ask to do or undo operations. At runtime, the correct `do` and `undo` operations from the concrete commands would be triggered. This increases cohesion, understandability and maintainability of classes `A` and `B` compared to the first solution. It also reduces coupling between these classes and the calculator class(es) and increases maintenance compared to the second solution.

3. A class diagram is provided in Figure 3. Here, we assumed that the mathematical operations can take any number of operands, and in the case of addition and substraction they would all be added/substracted, while in the case of sine only the first one would be used. Classes `A` and `B` will each contain attributes for the current command (as normal in the pattern) , but also for the history of performed commands (collection). The private methods would be used to add each newly performed command (i.e. after the `do` operation is used) to this collection, and remove commands once they are undone. The `undo` operation takes a number of commands that it would undo by navigating backwords in the collection before removing the undone commands from the collection.

4. The Java implementation should closely follow the class diagram, additionally include the attributes mentioned above, constructors, and implement the methods in classes.

3 Pattern identification 2

1. To keep all the logged data in one place that can later be queried for the purpose of producing reports, we should have a class for this purpose, let's call it `Log`. Classes involved in bank operations would need to be associated to the `Log` class such that their objects could request `Log` objects to record the data. But why have several `Log` objects (first problem)? This would be highly inefficient both in terms of space and time as a lot of memory would be used to store all those objects and obtaining reports would involve iterating over a large number of objects. Disregarding the requirement to only focus on object-riented design, one may claim that the different `Log` objects saving records to a database would solve this matter. This is actually not necessarly the case, as the different objects may end up competing over which one writes to the database and the database entries may end up being incomplete.

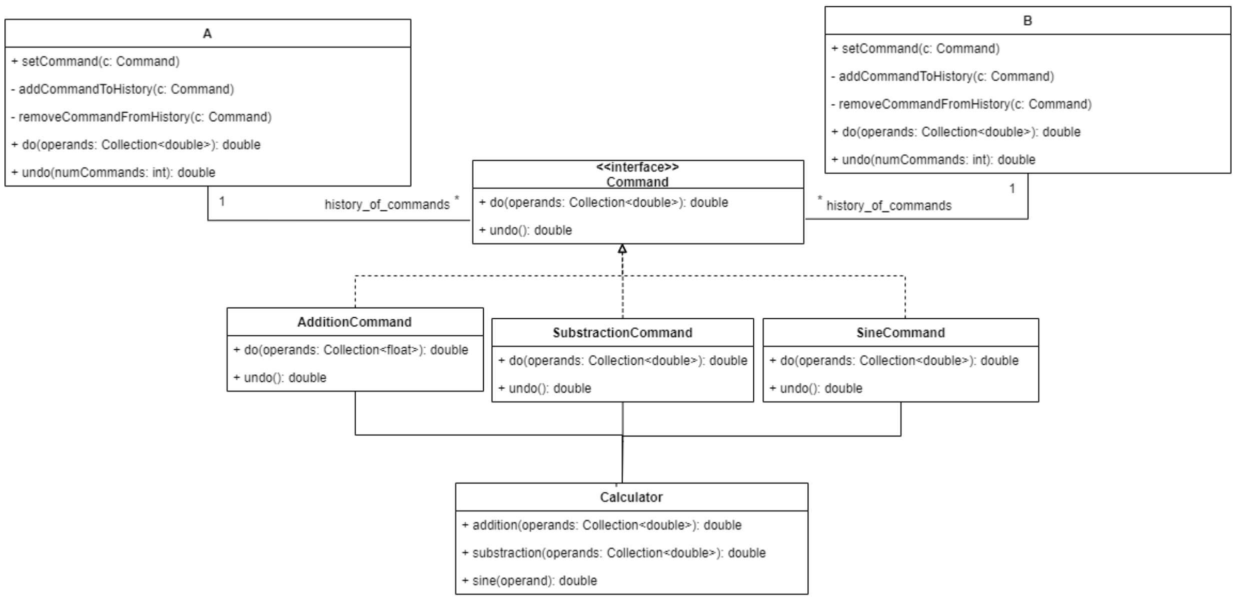
A
+ setCommand(c: Command)
- addCommandToHistory(c: Command)
- removeCommandFromHistory(c: Command)
+ do(operands: Collection<double>): double
+ undo(numCommands: int): double
B
+ setCommand(c: Command)
- addCommandToHistory(c: Command)
- removeCommandFromHistory(c: Command)
+ do(operands: Collection<double>): double
+ undo(numCommands: int): double
1
history_of_commands *
<<interface>>
Command
+ do(operands: Collection<double>): double
+ undo(): double
* history_of_commands
1
AdditionCommand
+ do(operands: Collection<float>): double
+ undo(): double
SubstractionCommand
+ do(operands: Collection<double>): double
+ undo(): double
SineCommand
+ do(operands: Collection<double>): double
+ undo(): double
Calculator
+ addition(operands: Collection<double>): double
+ substraction(operands: Collection<double>): double
+ sine(operand): double

Figure 3: Calculator class diagram

It is much better to keep one single `Log` object. To make sure only one such object is ever used, the objects of the classes involved in bank operations would need to pass it on between themselves (dependency injection). Moreover, the object would need to be initialised at the very start of the program. This is all very error prone, i.e. new `Log` objects can mistakenly be created, the `Log` object can be mistakenly overwritten.

2. The Singleton pattern is particularly useful in this context, because it would ensure that the `Log` class only ever has one instance (i.e. object), which provides global access to any other class in the system (in view of the bank's potential extension to recording many types of operations) such that they all record bank operations efficiently in one place, and which is protected from being overwritten such that the records are kept complete and correct.

3. A class diagram which incorporates the Singleton design pattern, but would also facilitate obtaining reports from the logged data, is provided in Figure 4. Note: only the constructor for the Singleton pattern is included (to make the fact that it must be private visible), and only the `getInstance()` operation which is important for Singleton is included, otherwise constructor, getter and setter operations being omitted. Also, one example method for getting reports printed out on withdrawals having been performed within a given time interval is provided.

   The underlined attribute and operation in the `Log` class mean that they are static. This is essential for the Singleton pattern: the static `getInstance()` method is what provides global access to the object of this class: it would be used as `Log.getInstance()`, and static methods can only work with static variables and so the `instance` attribute must also be static.

   We could have included an association between the `Log` class and itself, instead of representing the `instance` attribute, but this would have missed showing that the

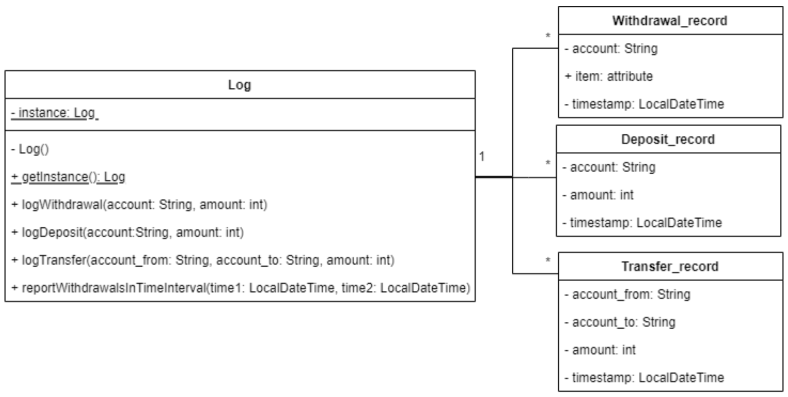
Log
- instance: Log
- Log()
+ getInstance(): Log
+ logWithdrawal(account: String, amount: int)
+ logDeposit(account:String, amount: int)
+ logTransfer(account_from: String, account_to: String, amount: int)
+ reportWithdrawalsInTimeInterval(time1: LocalDateTime, time2: LocalDateTime)
1
*
Withdrawal_record
- account: String
+ item: attribute
- timestamp: LocalDateTime
*
Deposit_record
- account: String
- amount: int
- timestamp: LocalDateTime
*
Transfer_record
- account_from: String
- account_to: String
- amount: int
- timestamp: LocalDateTime

Figure 4: The logging functionality for the bank system

attribute must be static which is important for Singleton. As usual, our decision to include each type of notation must be based on its usefulness for clarifying the diagram.

4. Apart from closely matching the class diagram from the previous task, the Java code should include the following:

   - Attributes holding collections of objects of each of the classes referring to the three types of deposit records.
   - An implementation for the Singleton pattern like in the notes next to the class diagrams on slides 12-13 of Lecture 12 Part 1. Eager initialisation uses up space in memory earlier, when the class loads, and is thus less efficient. Lazy initialisation is more efficient but can only work in single threaded systems (in multithreaded ones several instances may get created by the different threads).
   - The log methods in the `Log` class creating new records and adding them to the appropriate collection attributes.
   - A constructor, getters and setters for each of the classes referring to the three types of deposit records.
   - Implementations for any report methods demonstrated.

Cristina Adriana Alexandru. 21 Feb 2025.