P2 - Exercise hour

Pooja Rani

2021-05-21

Account Database

Parser

- Read input, parse accounts

Create Accounts

- A account class to store all the attributes of the account
- Use factory, prototype, or builder pattern to create the account object

Patterns

Design patterns

- Chain of Responsibility
- Factory
- Visitor
- Singleton
- Builder
- Null Object
- Prototype
- Iterator

Chain of Responsibility

- Search by client type
- Search needs to be chained
- First search in business clients

```
   // Chain objects
public class BussinessClient{
    public boolean match(String query) {
        if (super.match(query)) {
            return true;
        }
        return this.SalariedClient.match(query);
    }}
```

```
   // Chain objects
public class SalariedClient{
    public boolean match(String query) {
        if (super.match(query)) {
            return true;
        }
        return this.studentClient.match(query);
    }}
```

Factory pattern

- Use interface or inheritance to create an object

```
public interface Account{
        ..
}
public class CurrentAccount implements Account {
        ..
}

public class SavingAccount implements Account {
        ..
}
```

Factory pattern

```
public class FactoryDemo{
        String name =  getInputValue();
        Account account = null;
        if(name.equals("current"){
                account = new CurrentAccount();
        }
        if(name.equals("saving"){
                account = new SavingAccount();
        }
        assert account != null;
        ..
}
```

Factory pattern
- Open/Closed Principle.
- You can introduce new variants of accounts without breaking existing client code.

Visitor pattern

- Use the pattern when a behavior makes sense only in some classes of a class hierarchy, but not in others.
- To visit different clients and accounts

```
- ClientVisitor
- AccountVisitor
```

Visitor pattern

To visit different types of clients such as Business, Student, Salaried.

```
public interface ClientVisitor {
    void visitBusinessClient(BusinessClient businessClient);
    ....
}
public class BusinessClient {
    @Override
    public void accept(ClientVisitor clientVisitor) {
        clientVisitor.visitBusinessClient(this);
    }
}
```

Visitor pattern

We have different types of accounts such as current, saving, premium.

```
public interface Account{
        void accept(AccountVisitor accountVisitor);
}

public class CurrentAccount {
    @Override
    public void accept(AccountVisitor accountVisitor) {
        accountVisitor.visit(this);
    }

public class SavingAccount {
    }
}
```

Visitor pattern

To visit different types of accounts such as current, premium.

```
public interface AccountVisitor {
    void visit(CurrentAccount currentAccount);
    void visit(PremiumAccount premiumAccount);
}

public interface CurrentAccountVisitor implements
AccountVisitor {
    void visit(CurrentAccount currentAccount){ .. }
    void visit(PremiumAccount premiumAccount) { .. }
    ....
}
```

Visitor pattern

Visit accounts

```
public class VisitorDemo{
    Account[] accounts =
            {new CurrentAccount(), new PremiumAccount()}
    CurrentAccountVisitor ca = new CurrentAccountVisitor();
    for (Account account: accounts){
         account.accept(ca);
    }
}
```

Singleton pattern

- ensure that a class has only one instance

```
public abstract class Account {
    protected Account() {}

    public static Account instance() {
        if (instance == null) {
            instance = defaultInstance();
        }
        return instance;
    }
}
```

Builder pattern

- Use Builder pattern to create complex objects

```
public class PlaintextParser {
    ..
  Account.AccountBuilder accountBuilder = new
                    Account.AccountBuilder(id, client);
 ..
}

public static class AccountBuilder {
    ...
    public AccountBuilder(){
        ....
    }
    public Account build(){
              return new Account(this);
    }
}
```

Builder pattern

- Instantiate the account with the data provided by AccountBuilder
- AccountBuilder is a helper class to create account instance
- It can validate each account attribute separately
- Single Responsibility Principle.

Nullable fields

- A few fields are marked as optional
- You can use @Nullable annotation

```
public Account(String client, @Nullable Date date, .  ,
          @Nullable Integer boxOffice)
      {
         //account does not exists
      assert (AccountDB.find(client).isEmpty());

      // set all the attributes of the account
}
```

Null Object Pattern

- Handle null cases for the objects
- Null object has no side effects as it does nothing
- Used as stub in testing, when certain features such as database is not available for testing

```
public class NullRenderer implements Renderer {
    @Override
      public void render(Account account) { /* do nothing */ }
}
```

Other patterns

- Prototype
- Iterator pattern

Smalltalk

- Smalltalk is a dynamic typed language
- Style matches to the natural language, English
- GToolkit provides a live programming environment
- Supports live debugging
- Inspect objects with custom representations

Basic blocks

```
2 raisedTo: 30                          "1073741824- "
15 / 25                                  "(3/5)- Fraction"
'Hello Smalltalk'                   "'Hello Smalltalk' -ByteString"
anArray := #(1 2)
```

How do you write Loops?

Java

```
for(int i = 1; i < 10 ; i++)
            System.out.print(i);
```

GT

```
(1 to: 9) do: [:x | Transcript show: x printString]
```

Detect first odd number from the array?

Java

```
int[] array = {21, 23, 53, 66, 87};
        Integer result = null;
        for (int i = 0; i < array.length ; i++) {
            if (array[i] \% 2 == 1) {
                  result = array[i];
                  break;
} }
        if (result == null)
            throw new Exception("Not found");
```

GT

```
#(21 23 53 66 87) detect: [ :x | x odd]
```

Note: Note that arrays are 1-based-
that is, the first valid index is 1, rather than 0.

Exercise 11

- Turtle game similar to exercise 3
- Move turtle using 4 commands
- Commands are already created
- Understand ‘TurtleModel‘ and ‘BoardModel‘ class and document the classes

Document the classes

- Document all the details like purpose of the classes, what they do, instance variables, APIs warnings, observations etc. that you think is important to understand and extend these classes
- Smalltalk use Class comments as a primary source to document all such details
- Write all the details in comments
- Document ‘TurtleModel‘ and ‘BoardModel‘ class and document the classes

NOTE

- Deadline 28th May, 2021