WEEK 3

ONDERWERPEN

Design patterns:

- singleton
- command
- adapter & facade
- template method
- iterator
- composite
- state and proxy

OPGAVE 1: COMMAND PATTERN

In this exercise you will implement a remote control based on the Command Pattern. On Blackboard you'll find some Java file that you have to complete. As you see, there already exists a Television device.

a) Add a device Stereo (*receiver*) with methods to turn the Stereo on or off, and to increase or decrease the volume.
b) Add a *concrete commands* to use all the methods in Television and Stereo.
c) Add an undo() to the *command interface* and implement this for each of the above commands.
d) Add an undo-button in the *invoker*.
e) In the invoker, add a java.util.Stack<E> to keep track of the command history.
f) Complete the *client*, demonstrate the working of all the commands and undo commands (using System.out()).

OPGAVE 2: CREATING THREADS

a) Explain how Java implements multi-threading (interface Runnable and class Thread) using the command pattern, i.e. explain what classes have the role of invoker, command interface, concrete command and receiver.
b) Draw a sequence diagram to demonstrate how this works, i.e. show client, invoker, command and receiver.

OPGAVE 3: TEMPLATE METHOD PATTERN

a) Give an example of the template method pattern in the Java API.
b) Implement a small application using the template method pattern.

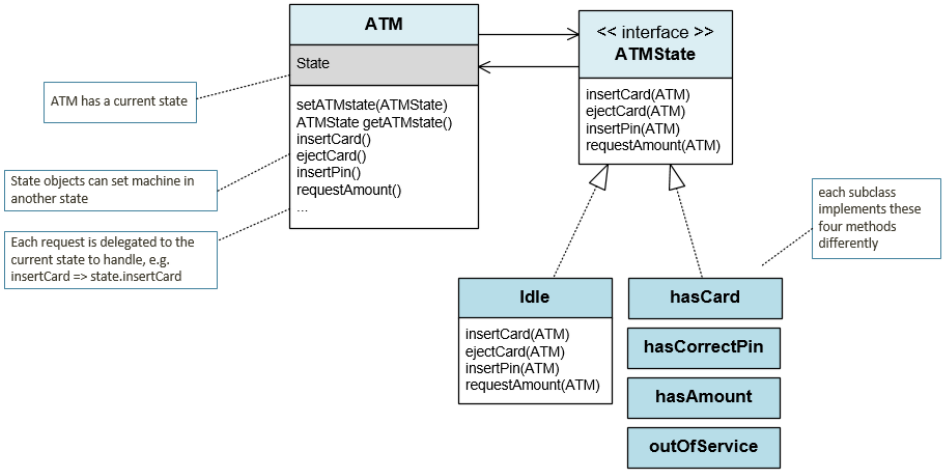
OPGAVE 4: ITERATOR PATTERN

Dit was ooit een toetsopgave, vandaar dat deze opgave in het Nederlands is. Zie voor het iterator pattern ook Liang hoofdstuk 20. Op Blackboard zijn de bestanden MyStack.java en Main.java te vinden die verder moeten worden afgemaakt.

a) De klasse MyStack implementeert een stack (datastructuur) op basis van een ArrayList. MyStack moet ook het interface Iterable<String> implementeren. Maak in MyStack.java de methoden push(), pop() en isEmpty(). De methoden push() plaatst een element op de stapel, de methode pop haalt het bovenste element van de stapel en isEmpty() geeft met een boolean aan of er nog een element op de stapel is.
b) Maak een klasse StackIterator die het interface java.util.Iterator implementeert. De klasse heeft twee methoden: hasNext() en next(). (De methode remove() hoef je niet te implementeren, dit is een 'optional operation').
c) Maak de klasse Main; een deel is al gegeven in de file Main.java. De functionaliteit van de methode main() is als volgt. Eerst worden drie elementen op de stapel geplaatst (is al gegeven). Haal deze drie elementen van de stapel en druk ze af; gebruik hierbij stack.iterator(). Plaats nogmaals de drie elementen op de stapel, en doe weer hetzelfde, maar nu met een for-statement.

OPGAVE 5: STATE DIAGRAM ATM

In the lecture week 3-2 the State Diagram was discussed, using the simulation of an ATM as an example. The class diagram for this assignment is shown below:



a) Draw the *state* diagram for the ATM.
b) Implement this state machine using the state pattern (part of the solution is available on Blackboard)
c) Assign numbers to events; then allow the user to enter events and show the evolving states using println().

OPGAVE 6: PROXY PATTERN

Add a (protection) proxy to the ATM application *to protect* the ('real') ATM. See the class diagram in the sheets.

OPGAVE 7: STATE VS STRATEGY

What are differences and similarities between state and strategy pattern?

OPGAVE 8: CHOOSE A PATTERN

a) Choose a design pattern and explain the workings of this pattern using a class diagram.
b) Give an example of an application of this pattern. The application should contain at least 6 classes.
c) Implement this example. At least the basic functionality should demonstrate some user interaction.
d) Explain the workings of this application using a class diagram.

OPGAVE 9: MATCH EACH PATTERN WITH ITS DESCRIPTION

NR	PATTERN
1	strategy
2	observer
3	decorator
4	factory
5	singleton
6	command
7	adapter
8	facade
9	template method
10	iterator
11	composite
12	state
13	proxy

LETTER	DESCRIPTION
A	simplifies the interface of a set of classes
B	allows objects to be notified when a specific object changes
C	clients treat collections of objects and individual objects in the same way
D	encapsulate varying behavior for the same object based on its internal state and uses delegation to switch between behavior
E	subclasses decide how to implement steps in an algorithm
F	wraps an object to provide new behavior
G	encapsulates all information needed to perform an action or trigger an event at a later time
H	defines a family of algorithms, encapsulate each one, and makes them interchangeable
I	provides a way to traverse a collection of objects without exposing its implementation
J	wraps an object to control access to it
K	wraps an object and provides a different interface
L	encapsulates object creation by providing an interface for creating an object without revealing the object's actual class
M	ensures one and only object is created