Tutorial for Web Engineering

Solution Assignment 5

Koblenz, Winter Term 2015/16

Names of group members:

Name	Matriculation No	E mail address
Mutjaba Rafiq	215 202 923	mrafiq@uni-koblenz.de
Aemal Sayer	215 202 503	aemal@uni-koblenz.de
Maximilian Strauch	211 201 869	maxstrauch@uni-koblenz.de
Daniel Vivas Estevao	211 200 044	dvivas@uni-koblenz.de

Task 1:

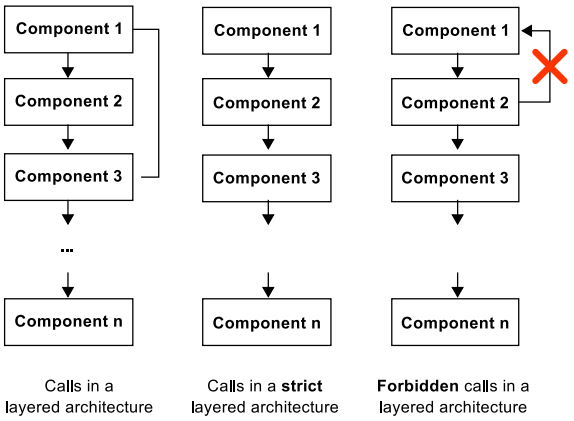

Fig. 1: Visualisation of layered architecture. Adapted and translated from: https://upload.wikimedia.org/wikipedia/commons/e/e1/SchichtenarchitekturAufrufstrukturen.svg.

a)

Figure 1 shows different examples of layered architectures. Most remarkable is the fact that in a layered architecture only upper layers can communicate with lower layers. For strict layering a layer is only allowed to communicate with its lower layer.

By design the MVC pattern[1] was developed for GUI elements e.g. a list to manage their model separated from the controller and the view. Most MVC components communicate to each other and

[1] See https://en.wikipedia.org/wiki/Model-view-controller.

they are separated for flexibility (e.g. change view for differen Look and Feel). The MVC pattern relies on bidirectional connections as explained in task 2 or shown in figure 2 to interchange data and notification messages.

Applying these both facts to the question leads to a "no" as response since in MVC components communicate bidirectional with each other and at least the model-view relation violates this rule since the model informs the view about changes and the view requests data from the model which are sent by the model to the view.

Furthermore a layered architecture is a hierarchy if one thinks of the OSI or internet model where upper layers get more rich concerning the features, logic and data density (the physical layer is pretty stupid, writing 1's and 0's to the channel in an NRZI scheme with bit stuffing whereas the appliciation layer contains a lot of logic). This hierarchy is not available in the MVC pattern. It's clear that the model is simpler as the controller as the view but it is not trivial to define whether the view lies on the controller or vice versa since they are kind of equivalent.

b)

The following architectural styles have been presented in the lecture:

- **Layered** – as explained in a) with the advantages: few parts in the architecture, refinable (zooming) and enforces *separation of concerns*.
- **Data flow** – like UML activiy diagrams, actions are performed in a batch-sequentail way with each step performing independent data transformation in a synchronous manner (each step must terminate before the next step can be performed).
- **Data centered** – activities and independend components use the service of a common repository.
- **MVC** – as explained in task 2, GUI pattern to separate view from its data (model) and controller (model edit etc.).

Social Web Applications:

The main purpose of a social web application is the usage through (social) people. Therefore the way of how data are processed (data flow architecture) and stored (data centered architecture) become less important. Also the model is quite unimportant. More important is the presentation layer so that humans can interchange their social artifacts. Therefore a **layered** architecture is well-suited for this application.

The upper most layer encapsulates the presentation layer and manage the way how the social web application presents itself in terms of Look and Feel and type service to the user. Beneath that the application layer follows containing the business logic, determining what the system actually does. It enforces the business logic rules and establishes these processes. Beneath that the data layer is located which then contains the data of the web application. All layers together form the application when stack in the described order. From the front, for the user, there is only one presentation layer providing only the data for the user and nothing more. And that is all the user is interested in.

Transactional Web Applications:

A transactional web application relies on the scheme – just like transactions in databases – of performing operations in a sequence and not concurrent. This matches the definition of a **data**

flow architectury very good. In this kind of architecture, the single operations are executed in a sequence where each operation must end before the next operation can start. So each operation is here a transaction of the transactional web application. So by example no data conflicts can arise if bank account withdraws are performed sequentially rather than applying them asynchronously at the same time (then the result depends on the random execution sequence which might not be desireable for a bank account owner).

Collaborative Web Applications:

A **data centered** architecture would be of advantage for this kind of application since the different users work collaborative and concurrent (*assuming this from the title*; e.g. a GoogleDoc like application for multiple users) on artifacts over the web. This requires for all users to get the current state of the artifact. Since it is goning to be concurrently edited the artifact must be "stored" central allowing the "independend components use the service of a common repository" (definition from the top) which is the storage location of the artifact.

Furthermore the MVC architecture can be embedded into the data centered architecture to separate the model which is the central repository from the view at the client (or user or "independend component") and the controller which communicates with the repository or view of the MVC for changes initiated by the client.

c)

High cohesion: cohesion measures how good functionality between modules in a system are related. Saying that a system has *high cohesion* one means that the functionality in the system is strongly related but every module is responsible for its own part of this.

Low coupling: coupling measures the degree of connection between different software modules. A system consisting of modules with *low coupling* is rather good since the modules are coupled only minimal allowing high flexibility.

Having software with *high cohesion* and *low coupling* is desirable and important since this software is simply maintainable and adjustable. Considering the MVC pattern as a system both of these two measures are fulfilled:

- MVC has a *high cohesion* since every component (model, view and controller) has its distinct functionality which it implements. Bringing these parts together it forms a functioning "system" without redundancy of functionality.
- MVC has a *low coupling* since it defines only the minimum set of data which needs to be exchanged between each component to work properly.

A *low coupling* can be enforced by strictly using interfaces between modules of a system. This also goes nicely together with UML component diagrams and allow a simple concurrent team based development of modules since the persons, where each is responsible for a module, only need to interchange their interfaces and then they can develop independent from each other. What is done inside one module is a total black box to the other modules forming this flexiblity.

Task 2:

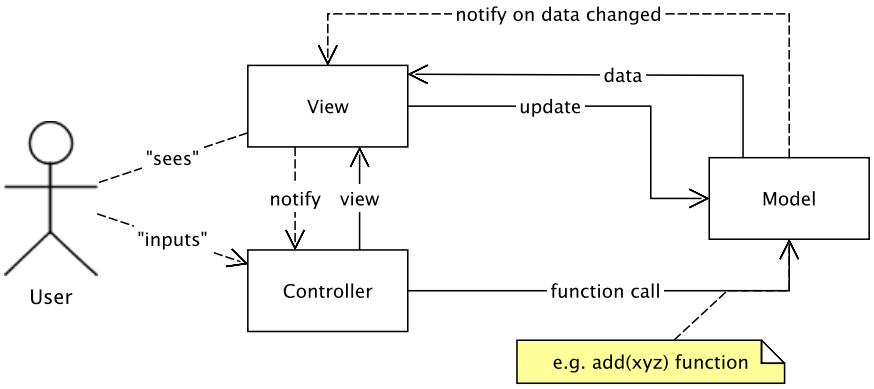

Fig. 2: Schematic view of the *Model View Controller (MVC)* pattern with a stakeholder (e.g. "GUI user"). Adapted from: `WE09webarchitecture.pdf`, slide 62.

Figure 2 shows the basic structure of the *Model View Controller (MVC)* where the rule of *separation of concerns* is applied and the **view** which is responsible only for displaying the data is separated from the **model** which only represents the data in a logic way and from the **controller** which controls the user input and changement of data (therefore it controls the view and data). Besides other advantages a simple use case of this design pattern is the usage of different views for the same data. Given a list of websites and their visitor frequency the MVC pattern could be used with two different views where the first view displays the visitor frequency per website as a table and the second view displays the visitor frequency in a histogram. Then a program could simply switch between these views using the same controller and model. Therefore the implementation effort and change effort is minimized and the resulting piece of software is much more flexible.

a)

The most prominent design pattern used in the MVC is the *observer pattern*. The relation between the model and the view uses this pattern whereas the model is a *observable object* and the view is an *observer object. See b) for explanation of this.*

Furthermore, the second occurance of this pattern is between view and controller where the view is the subject object and the controller is the observer object. If the view detects e.g. a mouse pointer click it will inform the controller and the controller can then handle the event and e.g. update the model (if the user changed something in the view).

Beyond that the *strategy* pattern where the controller can be implemented in a way that its behaviour can be interchanged (e.g. for handling desktop based mouse and keyboard input and an other controller for touch gestures input) can be found in the MVC pattern.

b)

Observer pattern:

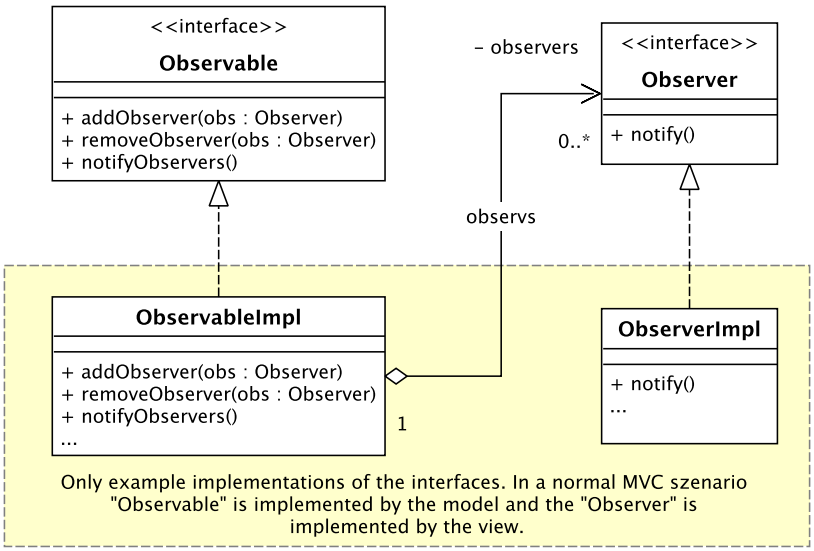

Fig. 3: UML 2 class diagram showing a schematic view of the *observer* pattern.

Figure 3 shows an UML diagram which explains the structure of the observer pattern. The actual pattern itself is represented by the classes `Observable` and `Observer`. The shown `*Impl` classes are only for demonstration of function. As explained in a) the `Observable` is in the MVC represented by the `Model` class and the `Observer` is the `View` class.

So the model implements `Observable` and the required methods for adding and removing `Observer` objects and notifying them. The view implements `Observer` which basically "watches" the "behaviour" of the class implementing the `Observable`. At an initialization stage the view calls `addObserver(this)` of the model to "subscribe" to the model. When the model is changed (e.g. by the controler when the user inputs a new value and the controller inserts it into the model; see figure 2, "notify on data change") it calls `notifyObservers()` which has basically the code of listing 1 calling the `notify()` method of the previously subscribed observer class, therefore the view. So the view gets the information that the model has changed and can now in the `notify()` method call the model to update and the model will provide the data (see figure 2).

Listing 1: Java source code of `Observable#notifyObservers()`.

```
for (Observer observer : observers) {
   if (observer != null) {
      observer.notify();
   }
}
```

Strategy pattern:

Figure 4 shows a UML representation of the strategy pattern. This pattern is very simple: an abstract strategy (`AbstractStrategy`) encapsulates some operation like `getLastArrowKeyPress()` which returns the last pressed arrow key on the keyboard (one value out of up, down, left and

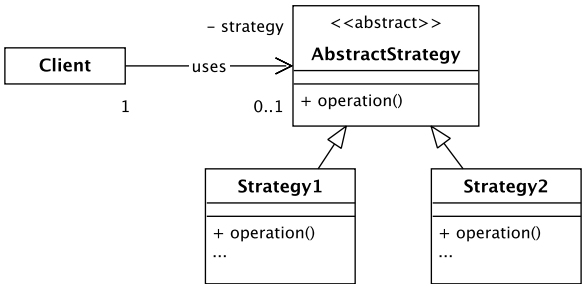

Fig. 4: UML 2 class diagram showing a schematic view of the *strategy* pattern.

right). Since not every device (like a smartphone) has a keyboard there are multiple strategies: `Strategy1` obtains the last arrow key from the last pressed key on the keyboard and `Strategy2` contains some magic code to recognize gestures as arrow key presses (wipe left for left key etc.) and provides the data through the `operation()` method. A client – in this case our application – uses then one strategy but both are interchangeable so that the client does not need to be aware of which is used actually. He only gets the data. An other part of the program then sets the strategy class used by the program (e.g. through a *factory pattern*). So the client depends only on the abstraction not on the concrete implementation which gives flexibility and swappability to the program.

Note: the `AbstractStrategy` could also be implemented using an interface.

c)

Here are some advantages of design patterns in software development verbalised as concise statements with explanation:

- **Patterns simplify transfer of knowledge!** – when a certain part of a program is recognized as a "pattern" the author extracts the core (as presented in b)) out of this program and creates a platform or technology independent representation. This allows for a concise representation and simple explanation of the behaviour of the pattern whithout dealing with platform specific problems. Every user of this pattern can than translate this abstract representation into a concrete representation which fits his/her needs. So the process of notating a pattern can be used as a serialization of specific experiences, knowledge and expertise for others.
- **Don't reinvent the wheel! Reuse it!** – creating a pattern from a certain program structure makes this behaviour or code scheme first of all code independent and abstracts from platform specific details. This enables an accurate presentation of the pattern which then can be used by others in their programs. Thus, other authors do not need to reinvent special structures. They can choose from a "zoo" of collected patterns and speedup development (*reuseablilty*). If patterns are provided through platform specific libraries another benefit is that its well tested (or should be) instead of a custom implementation so possible bugs are prevented and debugging time is reduced.
- **One name for many different things which mean the same.** – Using a design pattern, a behaviour or certain structure is concisely intended. This makes communication in a developing team easier because everything has one specific name. Without using patterns, everybody

would have their own representation of the structure with slightly different names (possibly) and talking about things would not be so concise.

- **Save the time and have a break!** – most problems in the software development can be reduced to patterns. So using it reduces the work time since the structure already exists. There is only the need for adapting it to the current problem.

Task 3:

MVC is a methodology or design pattern while JSP is only used in views of an MVC architecture. In MVC models are any "set" of classes (Java Beans) which are responsible for dealing with data. A view is a Java Server Pages (JSP) containing Java, HTML, CSS and Javascript code together, although there are several better ways of dealing with views in order to maintain separation of concerns, for example using a template engine. A controller is responsible for managing all the request and response traffic between client and server, in Java, Servlets are used as controllers.

For example for JavaEE applications the Spring framework2 exists which makes intensive use of the MVC pattern. But JSP is only part of JavaEE.