Université catholique de Louvain
Ecole Polytechnique de Louvain

LINFO2252
Software Maintenance and Evolution

Implementing a smart learning system

Lab Session 2

Professor: Kim Mens
(kim.mens@uclouvain.be)

Teaching assistants: Julien Liénard and Pierre Martou
({julien.lienard,pierre.martou}@uclouvain.be)

October 1, 2024

1 Approach of the lab sessions

In the previous lab session, we explored how to design a software system using feature modelling. Now we will explore how we can actually program such an application using the Java programming language (use your preferred IDE, not necessarily Eclipse).

In this new lab session, we suggest that you first implement a straightforward object-oriented version of your *smart learning system*. The focus is on revisiting some of the *object-oriented programming concepts* of Java while respecting *good programming practices* when developing your Java program. Given the theme of this course, your object-oriented implementation should be as *maintainable* as possible, its components should be as *reusable* as possible, and it should be easy to *evolve* and adapt the system in the future. Keep these quality criteria in mind when you design and implement your system. Furthermore, of particular importance will be that your implemented prototype should be *dynamically adaptive*; i.e., you should be able to add or remove features at runtime. To get the idea and avoid having to modify everything, we suggest you read the entirety of this lab session before starting to implement.

2 Implementing your system with the Model-View-Controller architectural pattern

To get you started on your object-oriented implementation of a *smart learning system*, we provide some guidance.

Initially, we do not ask you to implement a full user interface (UI), as that will be part of a future lab session. For now, it suffices if your application prototype is limited to commands given at the console and printing textual results on the terminal while running the application, or printing the successive states of your system in logs (more about that in the View component).

Also note that for this prototype you can "mock up" components that are out of scope of this course: no need for a database to register actual accounts or user's advancement, no need for an actual server if you include social features, and the same is true for all other difficult features that would require external tools. The lessons (of the learning system) themselves do not really need to be instructive either (you can come up with something funny), and can be short. What is of importance is the underlying mechanisms that are used.

In particular, the software architecture of your implementation should

follow the Model-View-Controller (or MVC) architectural pattern. Its goal is to uncouple application and user interaction. In this section, we describe a typical [1] MVC architectural pattern, then precise each component for our use case separately. Fig. 1 illustrates the way in which the MVC architectural pattern will be structured, though many variations exist and yours might slightly differ.

The "Model" component represents the core functionality and data of the application. It registers dependent Views, and notifies dependent components about data changes.

The "View" component represents what is shown on the screen and how the user can interact with it. It implements updates at runtime as requested by the Model component, and might transmit some interactions to the Controller component.

The "Controller" component receives user interactions or developer inputs as events, and translates these events to service requests to the Model and View components. In the case of your 'smart learning system' the controller is also responsible for handling all runtime feature activation/deactivation requests.

The Model component must maintain *logs* at all times. The logs are a description of the current system *state* in the form of a textual description of your application's functionalities and UI and are changed at runtime. Upon feature (de)activation, your Model should modify the system *state* represented in the logs accordingly, and those modifications will be reflected in the View.

The advised implementation route is as follows: start by implementing an infinite loop that takes inputs from the command line (basic Controller), then implement one or two features with some basic logic, and whose (de)activations affects the current system state (basic Model and logs). The View component will be added in later lab sessions in JavaSwing.

3 Controller

We advise to start by implementing a core module to your application, that will read inputs from the command line in order to understand and execute

[1] The Model-View-Controller architectural pattern was first conceptualized in the late 1970s and used in the Smalltalk 80 programming language to structure the graphical user interface of the Smalltalk programming environment in a way that separated internal representations of data from the ways they were presented to and interacted with by users. Over time, various adaptations of MVC have emerged to suit different programming environments and architectures.

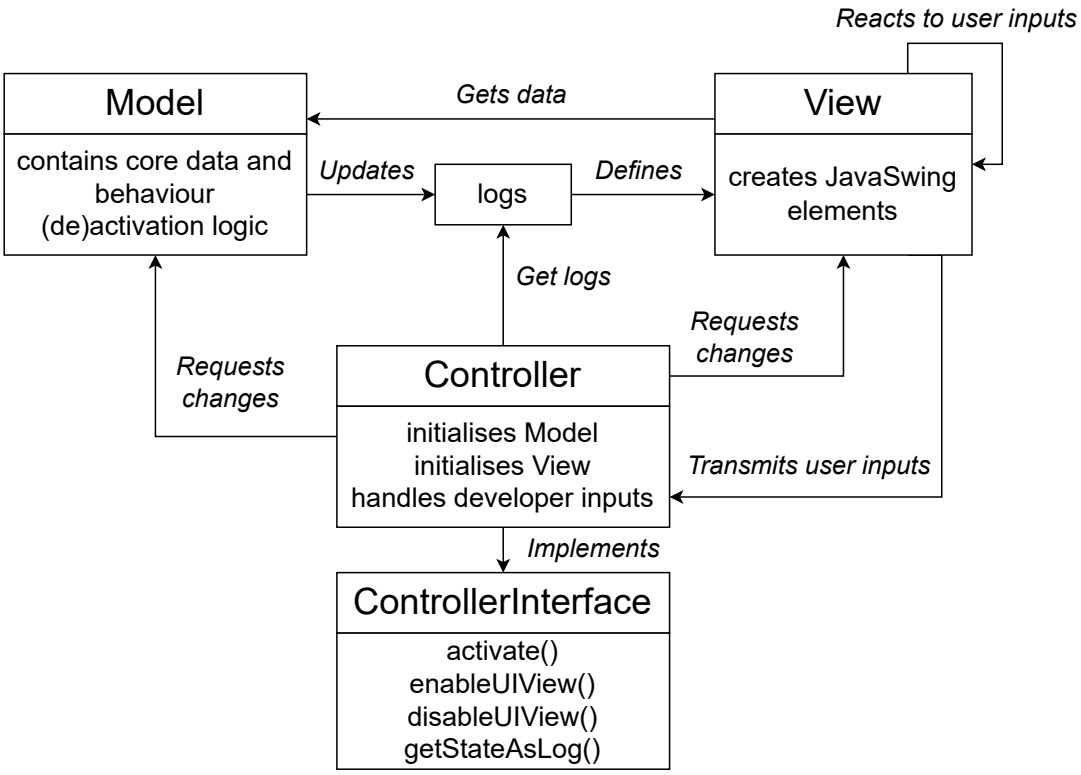

Figure 1: Typical structure of a MVC pattern

commands. Do not hesitate to use object-oriented concepts or patterns listed at the end of this lab session to implement a maintainable core architecture, that can evolve easily with new features.

If you don't see how to do so at first, start with an infinite loop that reads input and recognises commands among a hard-coded list. Later, this core module could receive inputs either from the command line (always useful for debugging and simulating your application without UI) or from the View component. You should be able to activate/deactivate any number of features at once. It is not expected that you check whether the feature configuration is correct according to your feature model, but appreciated. Incorrect configurations can lead to crashes or unstable behaviour and you should not spend time correcting those. Transitions are *atomic*: when you (de)activate a set of features, all changes must happen before anything else can interfere with the system.

We advise you that your command line reader can transmit feature (de)activations to the core module in some way, e.g. when the input is in the following form:

```
activate feature_1, ..., feature_n;
deactivate feature_n+1, ..., feature_n+m;
```

Your main Controller class should implement the provided interface and its listed methods (*ControllerInterface.java* on Moodle). They will be used in a later lab session in a generic way, so they should respect the interface's specifications (no change are allowed, but you can discuss the details with the assistants if you need). These methods will allow any developer to plug in to your implementation and handle it externally, i.e. without any knowledge of your actual implementation.

For the same reason, you are only allowed to (de)activate the features listed in your feature model in your Controller. You might want to change the feature model if you realise that some features are not precise enough or that some are missing, and you can improve it until the end of the semester. However, be careful to keep to the concepts of the course and not worsen the quality of your feature model (e.g. avoid adding features that represent technical details invisible to the user).

4 Model

In the Model component, you will implement the "back-end" of your application: features and what (de)activating non-abstract features at runtime does to your application and its data through logs. Each feature will be linked to two parts. First, a set of classes implements its logic (there is *not* necessarily one class per feature, since feature can represent a set of functionalities which are split in multiple classes). As a reminder, you can fake the elements that are out of scope of this course, such as databases, servers, and other external tools. The course's content can be short and funny. Please do not hesitate to ask the course assistants in case of doubt.

The second part is the (de)activation protocol associated to the feature, responsible for making appropriate changes and update the logs (textual representation of your application). The logs can follow your own code and practice, our only criterion is that it should be textual. As you can see in the provided Controller interface, you can organise your logs in a list of String. The order should not matter in this list (important for later lab sessions).

For example, behind a feature User, there could be a singleton class managing the list of users in the application and another class that represents one user (its name, password, status, ...). When it is activated, the logs will include the new functionalities:

1. add a button to create an account or connect: "buttonMedium connectOrCreate mainWindow-TopLeft" in the logs
2. add a button to disconnect: "buttonSmall disconnect mainWindow-TopRight" in the logs
3. add a window where the profile's details are shown: "window profile" in the logs
4. add a button to access the profile's details: "buttonSmall accessProfile mainWindow-BottomLeft" in the logs

In this example, we used some keywords. The first word is the nature of the functionality, the second which behaviour is associated to it, and the third where it is placed if needed. When the View will be implemented, it will receive the same modifications and create the corresponding elements.

You can create and elaborate your own keywords as you implement more of your UI, later. For now, start with what you think will be useful, focus on describing shortly the elements of your application and avoid technical details (e.g. the example uses "connectOrCreate" and not how it is done).

Due to inevitable feature interactions, the protocol of a particular feature (de)activation can depend on the presence/absence of other features. During the (de)activation, you can check the current system state through the logs or some other way (internal memory with the status of all features, knowledge of which feature are (de)activated alongside this one, etc).

Important note: your changes must occur at runtime and modify the current logs. You should avoid recreating the whole logs at each feature (de)activation.

5 View

The View should follow the separation of concern principle. It should contain only the code to instantiate and manipulate UI elements, but the data represented as well as the more complex logic should all be part of the Model component. It should use the logs to create the current views, as well as access the Model to get the represented data and behaviour according to the user's interactions. You will use JavaSwing, but you should not start implementing it for now.

For example, the content of the course itself (questions, expected answers, what happens when an user answers correctly, etc) are part of the Model that The View will access accordingly. For example, the View wants to show the

"Question 2" from the "Easy" module of the "English" course in the form of an open question. Open question can refer to a specific layout of the window, and the View component will find each particular content (question, answers, etc) in the Model component.

Even if it will be implemented later, this mindset can help you better define your Model and logs.

6 Software patterns

Remember that the most important things to keep in mind here are maintainability and evolution, as you will implement all the features you modelled in the previous lab session. Do not forget to use and apply all the good practices that you have already seen or will see later in the course when extending your system. In particular, use good object-oriented style, avoid any bad smells and try to respect good Java coding conventions.

We suggest that you implement your application incrementally. That is, work on a small set of features which works from the Controller to the Model (and then to the View later), and then extend your application with more features. And think about keeping your code modular and reusable by refactoring it incrementally as you evolve it. Adding a new feature and fixing the behaviour of an existing one should be made as easy as possible.

All of the following suggestions are neither exhaustive nor mandatory. It is not expected to implement any of those, especially at the start, but it is expected to end with a modular and maintainable architecture at the end of the semester. Courses describing software patterns will follow in the next weeks (cf. lecture 6 on Design Patterns), but you are also encouraged to analyse your shortcomings and find suitable solutions by yourselves. The following website can help you understand the goal of different patterns and how to implement them : `https://refactoring.guru/design-patterns`. You can certainly find there solutions to your design problems.

In the Model componet, some of the advised software patterns to implement the features' behaviour include the Singleton, Composite, Factory, Decorator, Strategy or Template patterns. While listing all feature (de)activations methods in a single class can seem easy at first (and you can start by doing that), we strongly advise against doing that in the long term. The (de)activation mechanism used to update the logs can be part of a software pattern, such as the Interpreter, Command, or Observer pattern.

How to transmit requests such as the feature (de)activations between the Controller and the Model can also get complex. Software patterns of interest

include the Iterator, Chain of Responsibility, Observer, Command patterns and using Java reflection.

The Observer can also be used to facilitate the transmission of messages between the View and the Controller when the View is implemented.

More generally and for any component, we list here some concepts that might help you design your *smart* learning system (core module, features, ...). This list is not exhaustive and is not an accurate summary of the theory courses.

- Classes, objects and message sending
- Encapsulation and information hiding
- Polymorphism and late or dynamic binding
- Abstraction, inheritance and class hierarchies
- Java interfaces

And even though this concept is not something that was explicitly tackled during the theory course, you may also consider using the more advanced feature of reflection. You can find some more information on how to use reflection in Java here:
`https://www.oracle.com/technical-resources/articles/java/javareflection.html` or online.

7 Mission 1b

The next deadline is a report on November 1st (date to be confirmed). Part of this report will be an updated feature model and explanation on how you implemented your Model-View-Controller architecture, as well as the logs. You will receive more details in the following lab sessions.