

Table of Contents

Implementing the User Interface — Task-based (20 pages)	1
Technical Requirements	2
API Styles	2
CRUD-based APIs	3
Task-based APIs	5
Task-based or CRUD-based?	6
Bootstrapping the UI	7
Implementing the UI	10
Model View Controller	11
Model View Presenter	11
Model View ViewModel	11
MVC versus MVP versus MVVM	12
Design pattern considerations	12
Usability	12
Testability	12
Invoking backend services	12
Summary	12
Questions	12
Further reading	12

Implementing the User Interface — Task-based (20 pages)

To accomplish a difficult task, one must first make it easy.

— Marty Rubin

The essence of DDD is a lot about capturing the business process and user intent a lot more closely. In the previous chapter, we designed a set of APIs without paying a lot of attention to how those APIs would get consumed by its eventual users. In this chapter, we will design the GUI for the LC application using the [JavaFX^{\[1\]}](#) framework. As part of that, we will examine how this approach of designing APIs in isolation can cause an impedance mismatch between the producers and the consumers. We will examine the consequences of this *impedance mismatch* and how task-based UIs can help cope with this mismatch a lot better.

At the end of the chapter, you will learn how to employ DDD principles to help you build robust user experiences that are simple and intuitive. You will also learn why it may be prudent to design your backend interfaces (APIs) from the perspective of the consumer.

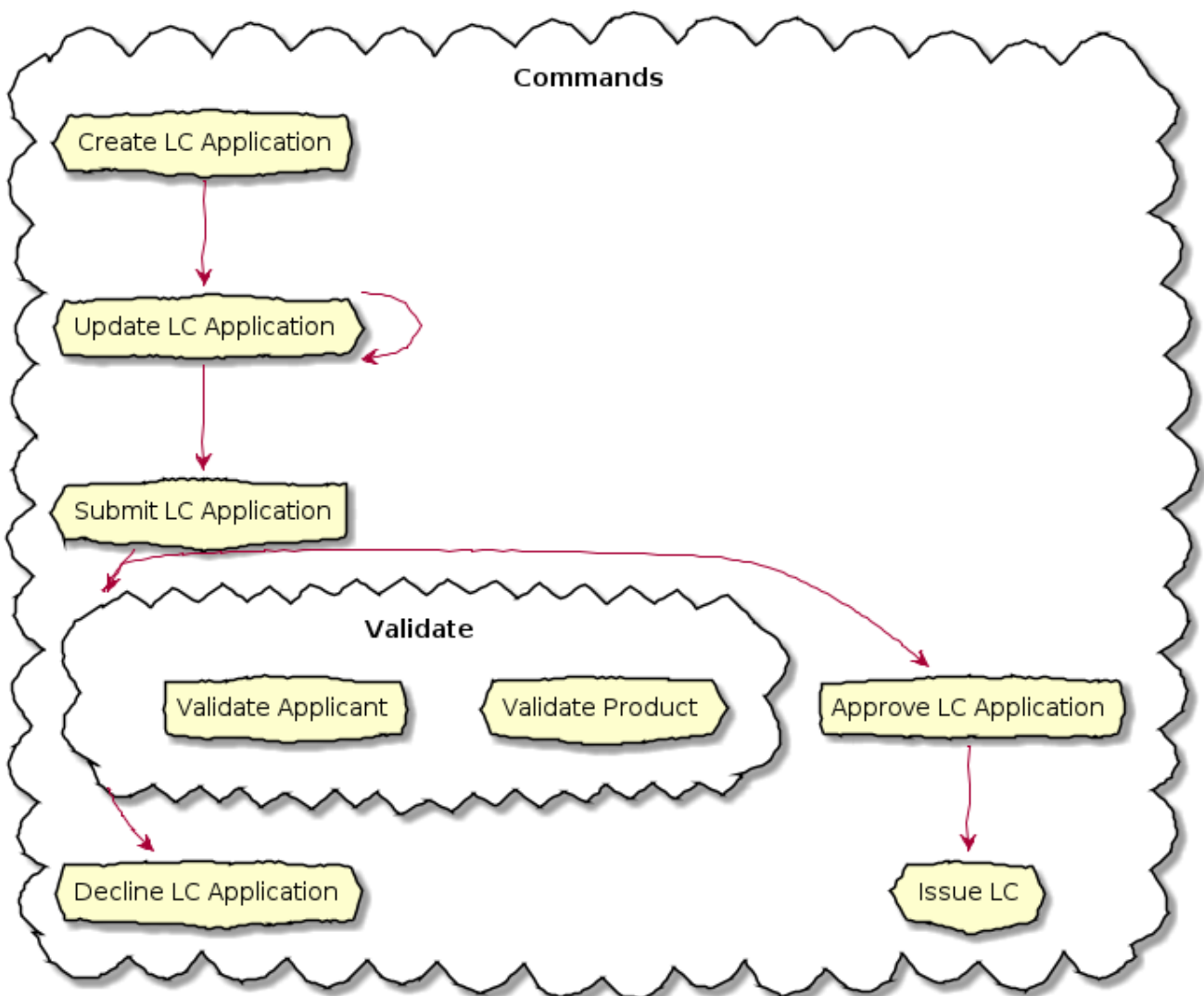
Technical Requirements

- JDK 1.8+ (We have used Java 16 to compile sample sources)
- JavaFX SDK 16 and SceneBuilder
- Maven 3.x
- Spring Boot 2.4.x
- mvvmFX 1.8 (<https://sialcasa.github.io/mvvmFX/>)
- JUnit 5.7.x (Included with spring boot)
- TestFX (for UI testing)
- OpenJFX Monocle (for headless UI testing)
- Project Lombok (To reduce verbosity)

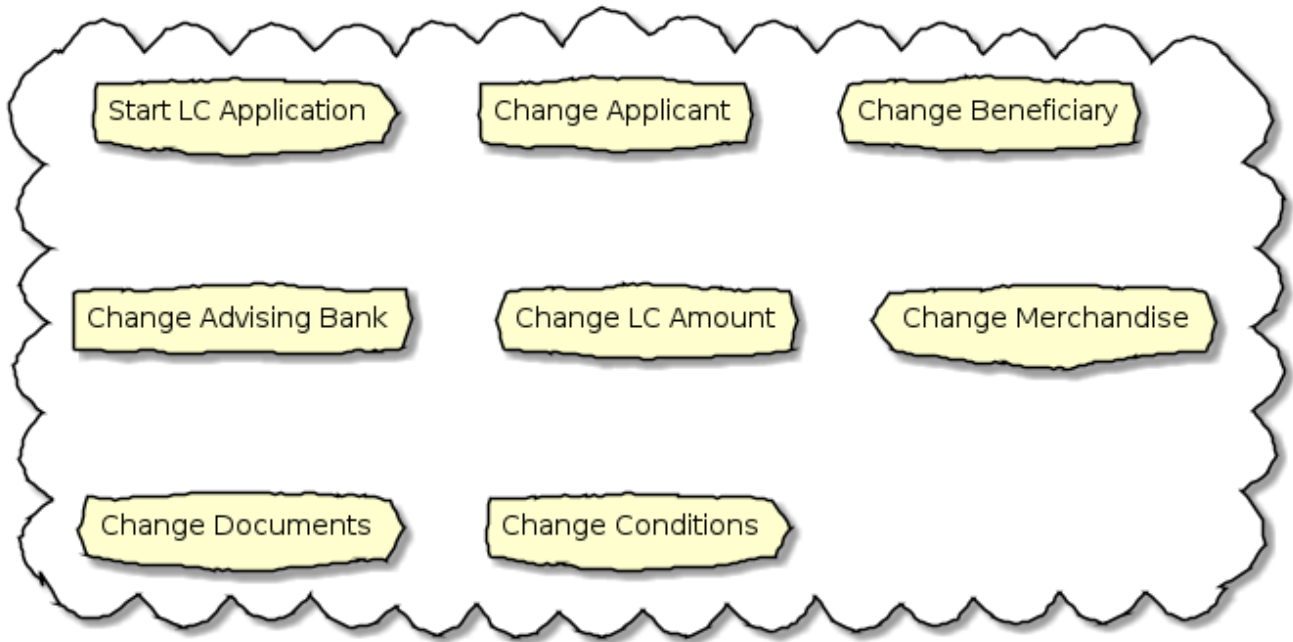
Before we dive deep into building the GUI solution, let's do a quick recap of where we left the APIs.

API Styles

If you recall from chapter 5, we created the following commands:



If you observe carefully, there seem to be commands at two levels of granularity. The "Create LC Application" and "Update LC application" are coarse grained, whereas the others are a lot more focused in terms of their intent. One possible decomposition of the coarse grained commands can be as depicted here:



In addition to just being more fine-grained than the commands in the previous iteration, the revised commands seem to better capture the user's intent. This may feel like a minor change in semantics, but can have a huge impact on the way our solution is used by its ultimate end-users. The question then is whether we should *always* prefer fine-grained APIs over coarse grained ones. The answer can be a lot more nuanced. When designing APIs and experiences, we see two main styles being employed:

- CRUD-based
- Task-based

Let's look at each of these in a bit more detail:

CRUD-based APIs

CRUD is an acronym used to refer to the four basic operations that can be performed on database applications: Create, Read, Update, and Delete. Many programming languages and protocols have their own equivalent of CRUD, often with slight variations in naming and intent. For example, SQL — a popular language for interacting with databases — calls the four functions Insert, Select, Update, and Delete. Similarly, the HTTP protocol has **POST**, **GET**, **PUT** and **DELETE** as verbs to represent these CRUD operations. This approach has got extended to our design of APIs as well. This has resulted in the proliferation of both CRUD-based APIs and user experiences. Take a look at the **CreateLCApplicationCommand** from Chapter 5:

```
import lombok.Data;

@Data
public class CreateLCApplicationCommand {

    private LCAApplicationId id;

    private ClientId clientId;
    private Party applicant;
    private Party beneficiary;
    private AdvisingBank advisingBank;
    private LocalDate issueDate;
    private MonetaryAmount amount;
    private String merchandiseDescription;
}
```

Along similar lines, it would not be uncommon to create a corresponding `UpdateLCApplicationCommand` as depicted here:

```
import lombok.Data;

@Data
public class UpdateLCApplicationCommand {

    @TargetAggregateIdentifier
    private LCAApplicationId id;

    private ClientId clientId;
    private Party applicant;
    private Party beneficiary;
    private AdvisingBank advisingBank;
    private LocalDate issueDate;
    private MonetaryAmount amount;
    private String merchandiseDescription;
}
```

While this is very common and also very easy to grasp, it is not without problems. Here are some questions that taking this approach raises:

1. Are we allowed to change everything listed in the `update` command?
2. Assuming that everything can change, do they all change at the same time?
3. How do we know what exactly changed? Should we be doing a diff?
4. What if all the attributes mentioned above are not included in the `update` command?
5. What if we need to add attributes in future?
6. Is the business intent of what the user wanted to accomplish captured?

In a simple system, the answer to these questions may not matter that much. However, as system complexity increases, will this approach remain resilient to change? We feel that it merits taking a look at another approach called task-based APIs to be able to answer these questions.

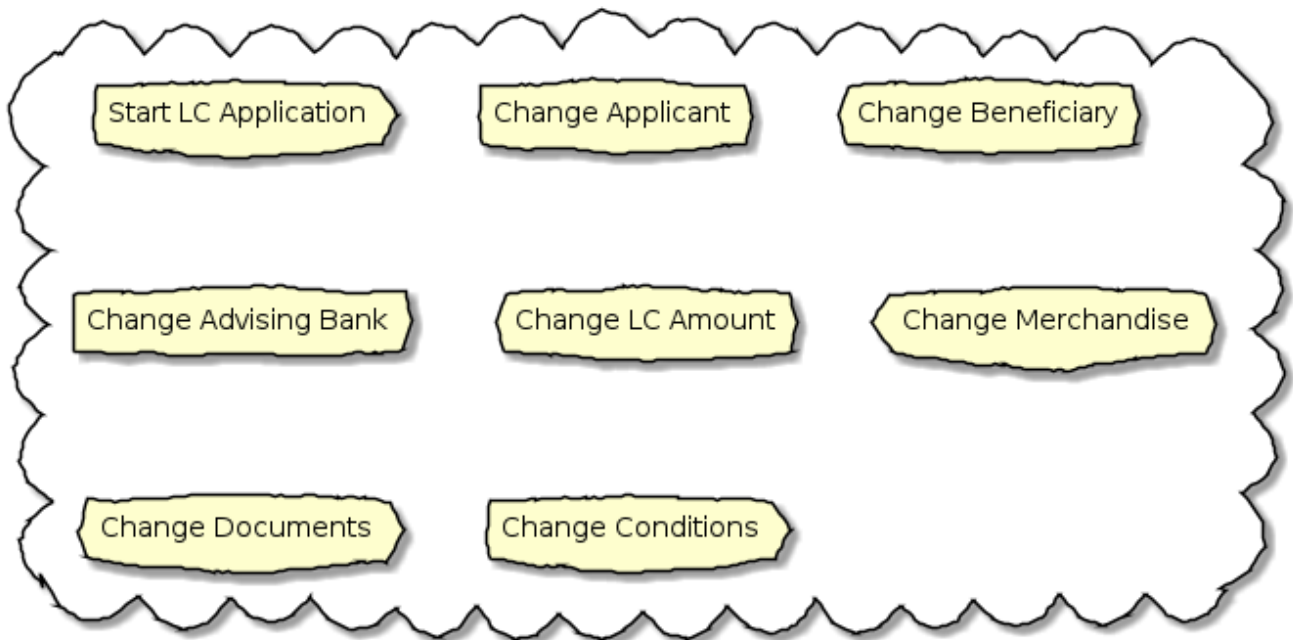
Task-based APIs

In a typical organization, individuals perform tasks relevant to their specialization. The bigger the organization, the higher the degree of specialization. This approach of segregating tasks according to one's specialization makes sense, because it mitigates the possibility of stepping on each others' shoes, especially when getting complex pieces of work done. For example, in the LC application process, there is a need to establish the value/legality of the product while also determining the credit worthiness of the applicant. It makes sense that each of these tasks are usually performed by individuals in unrelated departments. It also follows that these tasks can be performed independently from the other.

In terms of a business process, if we have a single `CreateLCApplicationCommand` that precedes these operations, individuals in both departments firstly have to wait for the entire application to be filled out before either can commence their work. Secondly, if either piece of information is updated through a single `UpdateLCApplicationCommand`, it is unclear what changed. This can result in a spurious notification being sent to at least one department because of this lack of clarity in the process.

Since most work happens in the form of specific tasks, it can work to our advantage if our processes and by extension, our APIs mirror these behaviors.

Keeping this in mind, let's re-examine our revised APIs for the LC application process:



While it may have appeared previously that we have simply converted our coarse-grained APIs to become more fine-grained, this in reality is a better representation of the tasks that the user intended to perform. So, in essence, task-based APIs are the decomposition of work in a manner that aligns more closely to the users' intents. With our new APIs, product validation can commence as soon as `ChangeMerchandise` happens. Also, it is unambiguously clear what the user did and what

needs to happen in reaction to the user's action. It then begs the question on whether we should employ task-based APIs all the time? Let's look at the implications in more detail.

Task-based or CRUD-based?

CRUD-based APIs seem to operate at the level of the aggregate. In our example, we have the LC aggregate. In the simplest case, this essentially translates to four operations aligned with each of the CRUD verbs. However, as we are seeing, even in our simplified version, the LC is becoming a fairly complex concept. Having to work with just four operations at the level of the LC is cognitively complex. With more requirements, this complexity will only continue to increase. For example, consider a situation where the business expresses a need to capture a lot more information about the **merchandise**, where today, this is simply captured in the form of free-form text.

```
public class Merchandise {
    private MerchandiseId id;
    private Set<Item> items;
    private Packaging packaging;
    private boolean hazardous;
}

class Item {
    private ProductId productId;
    private int quantity;
    // ...
}

class Packaging {
    // ...
}
```

In our current design, the implications of this change are far reaching for both the provider and the consumer(s).

Characteristic	CRUD-based	Task-based	Commentary
Usability	☹	☺	Task-based interfaces tend to provide more fine-grained controls that capture user intent a lot more explicitly, making them naturally more usable — especially in cases where the domain is complex.
Reusability	⊗	☺	Task-based interfaces enable more complex features to be composed using simpler ones providing more flexibility to the consumers.
Scalability	⊗	☺	Task-based interfaces have an advantage because they can provide the ability to independently scale specific features. However, if the fine-grained task-based APIs are used almost all the time in unison, it may be required to re-examine whether the users' intents are accurately captured.

Characteristic	CRUD-based	Task-based	Commentary
Security	⊗	☑	For task-based interfaces, security is enhanced from the producer's perspective by enabling application of the <i>principle of least privilege</i> ^[2] .
Latency	☑	⊗	Arguably, coarse-grained CRUD interfaces can enable consumers to achieve a lot more in less interactions, thereby providing low latency.
Management Overhead	☑	⊗	For the provider, fine-grained interfaces require a lot more work managing a larger number of interfaces.
Complexity	?	?	Complexity of the system as a whole is proportional to the number of features that need to be implemented. Irrespective of API style, this usually remains constant. However, spreading complexity relatively uniformly across multiple simpler interfaces can enable managing complexity a lot more effectively.

As we can see, the decision between CRUD-based and task-based interfaces is nuanced. We are not suggesting that you should choose one over the other. That will depend on your specific requirements and context. In our experience, task-based interfaces treat user intents as first class citizens and perpetrate the spirit of DDD's ubiquitous language very elegantly. In a lot of scenarios, providing both styles of APIs may work well for consumers, although it may add a certain amount of overhead to the interface provider.

This is a chapter on evolving the user interface, however, we have spent a lot of time discussing the backend APIs. However, the same principles apply when designing graphical user interfaces as well. Let's revert back to creating the user interface for the LC application.

Bootstrapping the UI

We will simply be building on top of the LC application we created in Chapter 5: Implementing Domain Logic. For detailed instructions, refer to the section on Bootstrapping the Application. In addition, we will need to add the following dependencies to the `dependencies` section of the Maven `pom.xml` file in the root directory of the project:

```
<dependencies>
  <!--...-->
  <dependency>
    <groupId>org.openjfx</groupId>
    <artifactId>javafx-controls</artifactId>
    <version>${javafx.version}</version>
  </dependency>
  <dependency>
    <groupId>org.openjfx</groupId>
    <artifactId>javafx-graphics</artifactId>
    <version>${javafx.version}</version>
  </dependency>
  <dependency>
    <groupId>org.openjfx</groupId>
    <artifactId>javafx-fxml</artifactId>
    <version>${javafx.version}</version>
  </dependency>
  <dependency>
    <groupId>de.saxsys</groupId>
    <artifactId>mvvmfx</artifactId>
    <version>${mvvmfx.version}</version>
  </dependency>
  <dependency>
    <groupId>de.saxsys</groupId>
    <artifactId>mvvmfx-spring-boot</artifactId>
    <version>${mvvmfx.version}</version>
  </dependency>
  <!--...-->
</dependencies>
```

To run UI tests, you will need to add the following dependencies:


```

<dependencies>
  <!--...-->
  <dependency>
    <groupId>org.testfx</groupId>
    <artifactId>testfx-junit5</artifactId>
    <scope>test</scope>
    <version>${testfx-junit5.version}</version>
  </dependency>
  <dependency>
    <groupId>org.testfx</groupId>
    <artifactId>openjfx-monocle</artifactId>
    <version>${openjfx-monocle.version}</version>
  </dependency>
  <dependency>
    <groupId>de.saxsys</groupId>
    <artifactId>mvmfx-testing-utils</artifactId>
    <version>${mvmfx.version}</version>
    <scope>test</scope>
  </dependency>
  <!--...-->
</dependencies>

```

To be able to run the application from the command line, you will need to add the **javafx-maven-plugin** to the **plugins** section of your **pom.xml**, per the following:

```

<plugin>
  <groupId>org.openjfx</groupId>
  <artifactId>javafx-maven-plugin</artifactId>
  <version>${javafx-maven-plugin.version}</version>
  <configuration>
    <mainClass>com.premonition.lc.issuance.App</mainClass>
  </configuration>
</plugin>

```

To run the application from the command line, use:

```
mvn javafx:run
```



If you are using a JDK greater than version 1.8, the JavaFX libraries may not be bundled with the JDK itself. When running the application from your IDE, you will likely need to add the following:

```

--module-path=<path-to-javafx-sdk>/lib/ \
--add-modules=javafx.controls,javafx.graphics,javafx.fxml,javafx.media

```

We are making use of the mvvmfx framework to assemble the UI. To make this work with spring boot, the application launcher looks as depicted here:

```
@SpringBootApplication
public class App extends MvvmfxSpringApplication { ❶

    private static final InputStream icon = Objects.requireNonNull(App.class
        .getResourceAsStream("/lc-icon.png"));

    public static void main(String[] args) {
        Application.launch(args);
    }

    @Override
    public void startMvvmfx(Stage stage) {
        stage.setTitle("LC Issuance");

        final Parent parent = FluentViewLoader.fxmlView(MainView.class)
            .providedScopes(new UserScope("admin"))
            .load().getView();

        final Image icon = new Image(App.icon);
        stage.getIcons().add(icon);
        final Scene scene = new Scene(parent);
        stage.addEventHandler(KeyEvent.KEY_PRESSED, event -> {
            if (KeyCode.ESCAPE == event.getCode()) {
                stage.close();
            }
        });
        stage.setScene(scene);
        stage.show();
    }
}
```

❶ Note that we are required to extend from the mvvmfx framework class `MvvmfxSpringApplication`.



Please refer to the ch06 directory of the accompanying source code repository for the complete example.

Implementing the UI

When working with user interfaces, it is fairly customary to use one of these presentation patterns:

- Model-View-Controller (MVC)
- Model-View-Presenter (MVP)
- Model-View-ViewModel (MVVM)

Each of these patterns enable us to produce code that is loosely coupled, testable and maintainable.

Let's briefly examine each of these in more detail here:

Model View Controller

This is arguably the oldest, most popular when implementing user interfaces, given that it has been in existence since the early 1970s. The pattern breaks the app into three components:

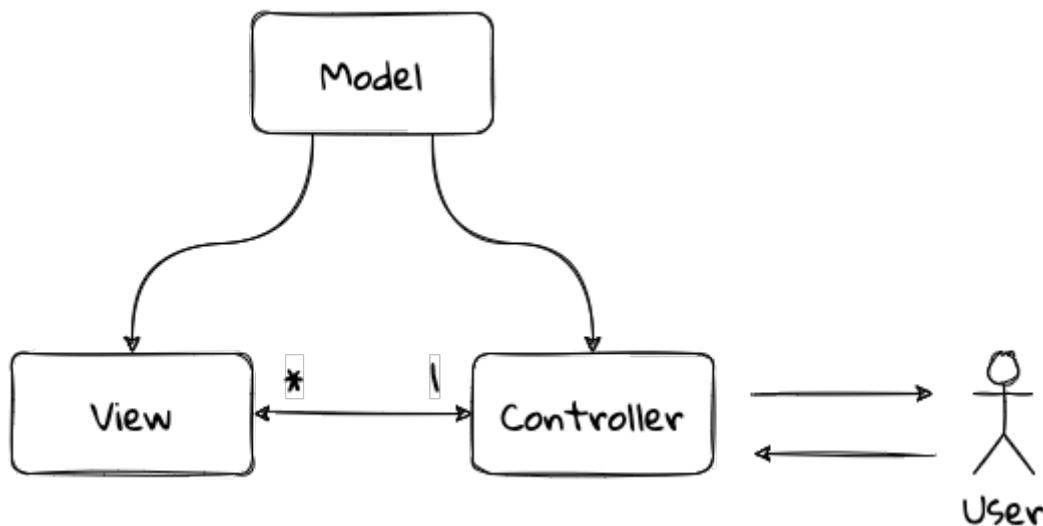


Figure 1. MVC design pattern

- Model: responsible to house the business logic and managing the state of the application.
- View: responsible for presenting data to the user and handling user interactions.
- Controller: responsible to act as a glue between the model and the view. It is also responsible for handling incoming requests, data management, networking and validation.

Model View Presenter

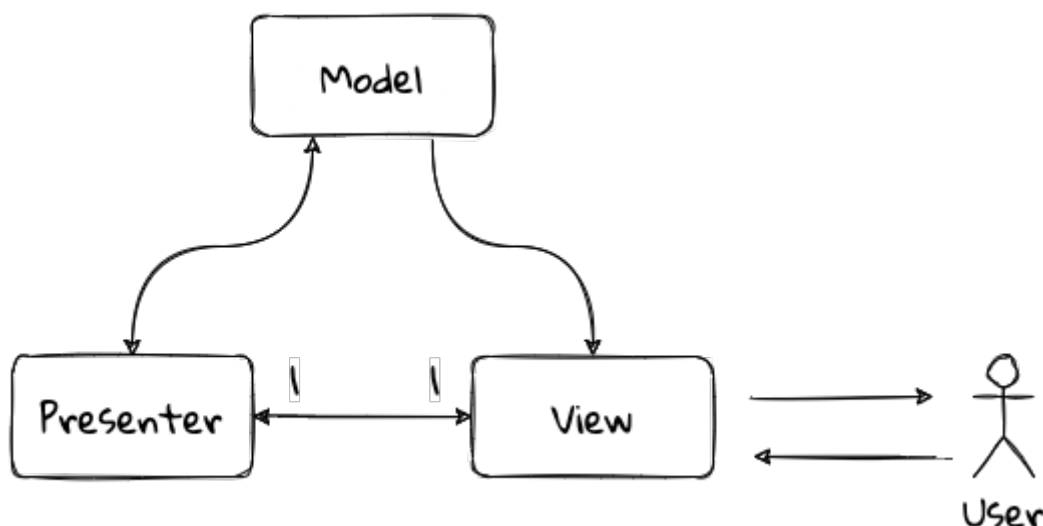


Figure 2. MVP design pattern

Model View ViewModel

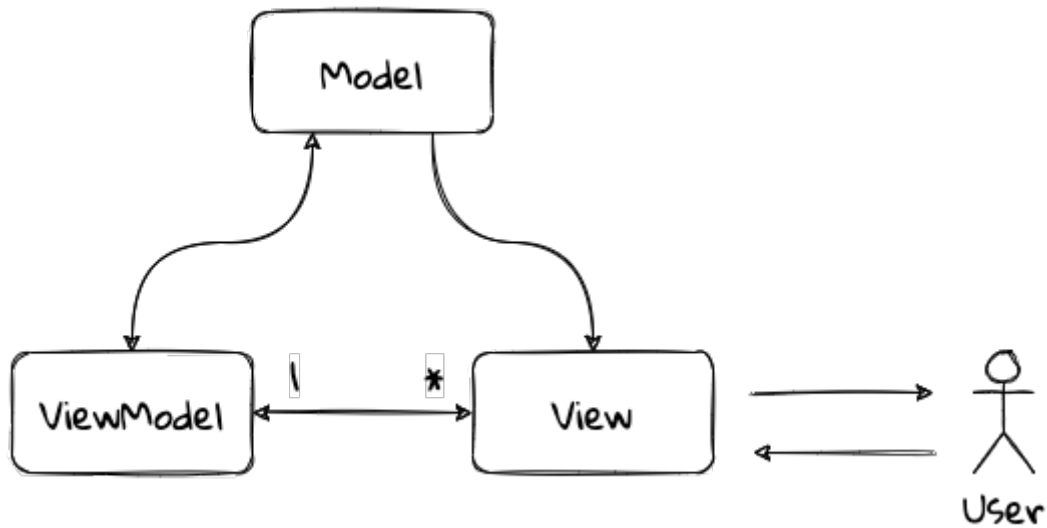


Figure 3. MVVM design pattern

MVC versus MVP versus MVVM

Design pattern considerations

Usability

Testability

Invoking backend services

Summary

Questions

Further reading

Title	Author	Location
TODO	TODO	TODO

[1] <https://openjfx.com/>

[2] https://en.wikipedia.org/wiki/Principle_of_least_privilege