z Grupp26

Architecture specification 1.1

Claudio

Marcus Parkkinen, Fredrik Åhs, Aki Käkelä

12

Table of contents

1. Introduction

1.1. Purpose of this document

1.2. Prerequisites

1.3. Document scope

1.4. Terminology

2. Architecture overview

2.1 Architecture block diagram

2.2 View module

2.3 Controller module

2.4 Model module

3. UML diagrams

3.1 View module UML

3.2 Controller module UML

3.3 Model module UML

3.4 Complete application UML

1 Introduction

1.1 Purpose of this document

This design document aims to provide an introduction and support for new developers in understanding the purpose and role of the core components in the application. It outlines the module structure of the application as well as intramodular and intermodular communication flow, and briefly discusses the benefits and reasons behind the chosen software architecture.

1.2 Prerequisites

As the design concepts discussed in this document are heavily rooted in existing software patterns such as MVC and Observer, it is advised that readers possess at least basic knowledge of their function and general structure to fully understand its content.

1.3 Document scope

The scope of this document is limited to classes that participate in the communication flow between the different modules of the application core. It does not cover the function and role of resource, utility and adapter classes. Furthermore, it does not aim to describe or discuss implementation details of any classes in the application.

1.4 Terminology

Book	-	A collection of audio tracks that collectively compose an audio book.
Bookshelf	-	A collection of all books created by the user.
Seekbar	-	A graphical element that displays progress in a audio track and provides the user control to seek within either a book or a single track in a book.
CRUD	-	Create, Read, Update and Delete.

2 Architecture overview

2.1 Application block diagram

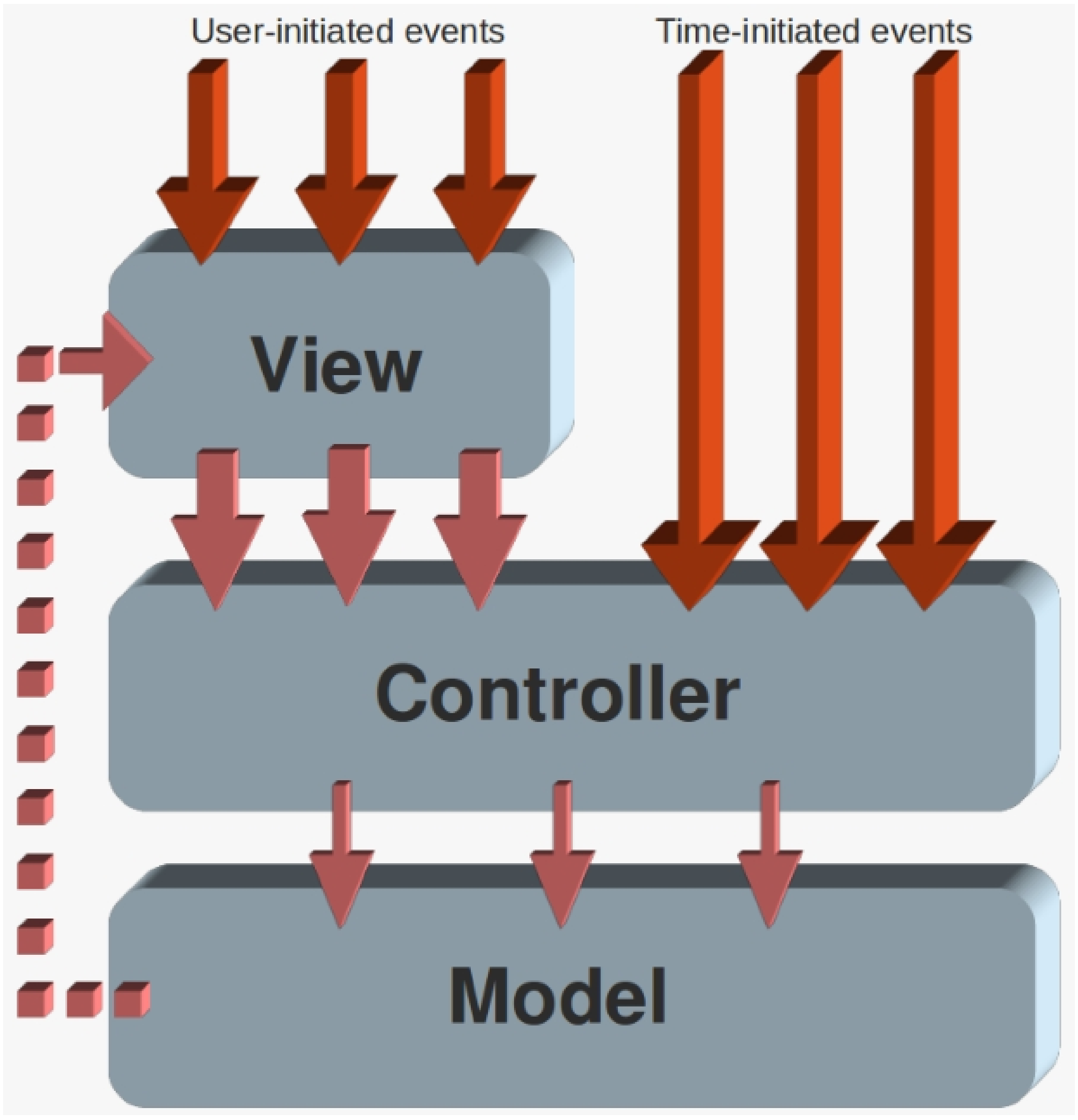

Figure 1 - Architecture block diagram. Darker arrows denote root sources for events that enter the application while brighter hue arrows depict information flow between application components.

As illustrated in figure 1 above, the structure of the application is heavily rooted in the MVC-architecture, where the two main sources that trigger change within the application are user-initiated events and time-initiated events. User-initiated events constitute all forms of interaction with user controls contained in the view module such as buttons, seek bars and lists, while time-initiated events are triggered by threads contained in different parts of the controller module. As depicted in Figure 1, the intermodular information flow follows a linear pattern, transitioning between the view, controller and model modules in a circular sequence. The structural benefits of this design include enhanced control over the application's behavior as UI-elements lack the ability to directly mutate any objects present in the model, as well as simplified debugging through better cohesion of classes.

The dotted arrow spanning between the model and view modules marks a relationship defined by the observer-pattern to further minimize the coupling between the Model and View modules.

2.2 View module

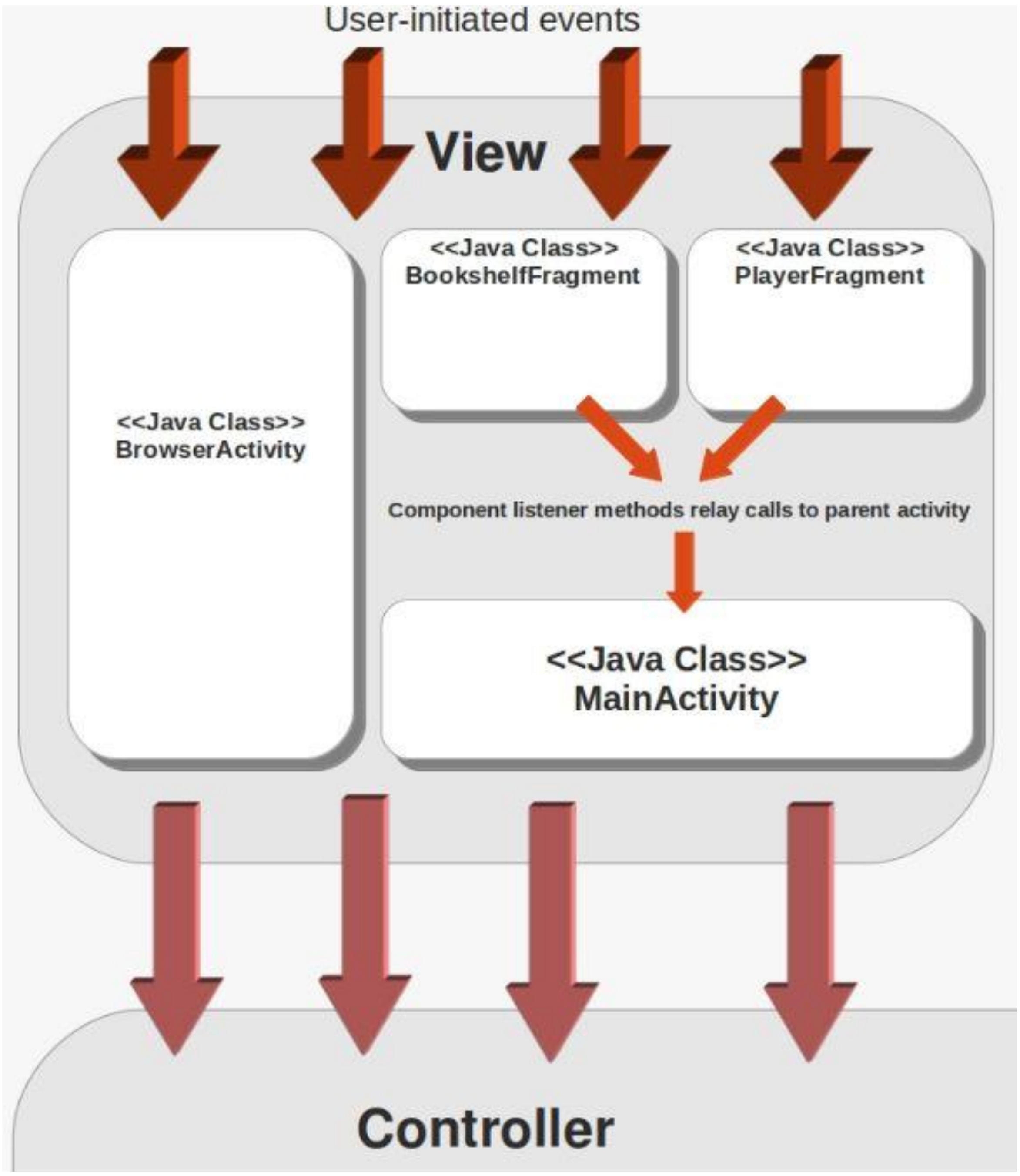
User-initiated events
View
<<Java Class>> BookshelfFragment
<<Java Class>> PlayerFragment
<<Java Class>> BrowserActivity
Component listener methods relay calls to parent activity
<<Java Class>> MainActivity
Controller

Figure 2 - Event information flow chart for the view module of the application.

The application contains three classes that provide elements that the user can interact with. As depicted in Figure 2 above, these are *BrowserActivity*, *BookshelfFragment* and *PlayerFragment*. Depending on which of these component classes contain the UI-elements the user interacts with, events can either be relayed through a fragment container (a FragmentActivity; *MainActivity*), as in the case with the fragment classes, or directly translated into a method call in the controller module as with the *BrowserActivity* class. This way, each user-initiated interaction with a graphical component is indirectly mapped to a method call in a controller class, as depicted in Figure 3 below.

MainActivity also extends the PropertyChangeListener interface, and is thus the class that receives updates from the model module of the application. The information contained in the updates is translated into method calls in the fragment classes.

2.3 Controller module

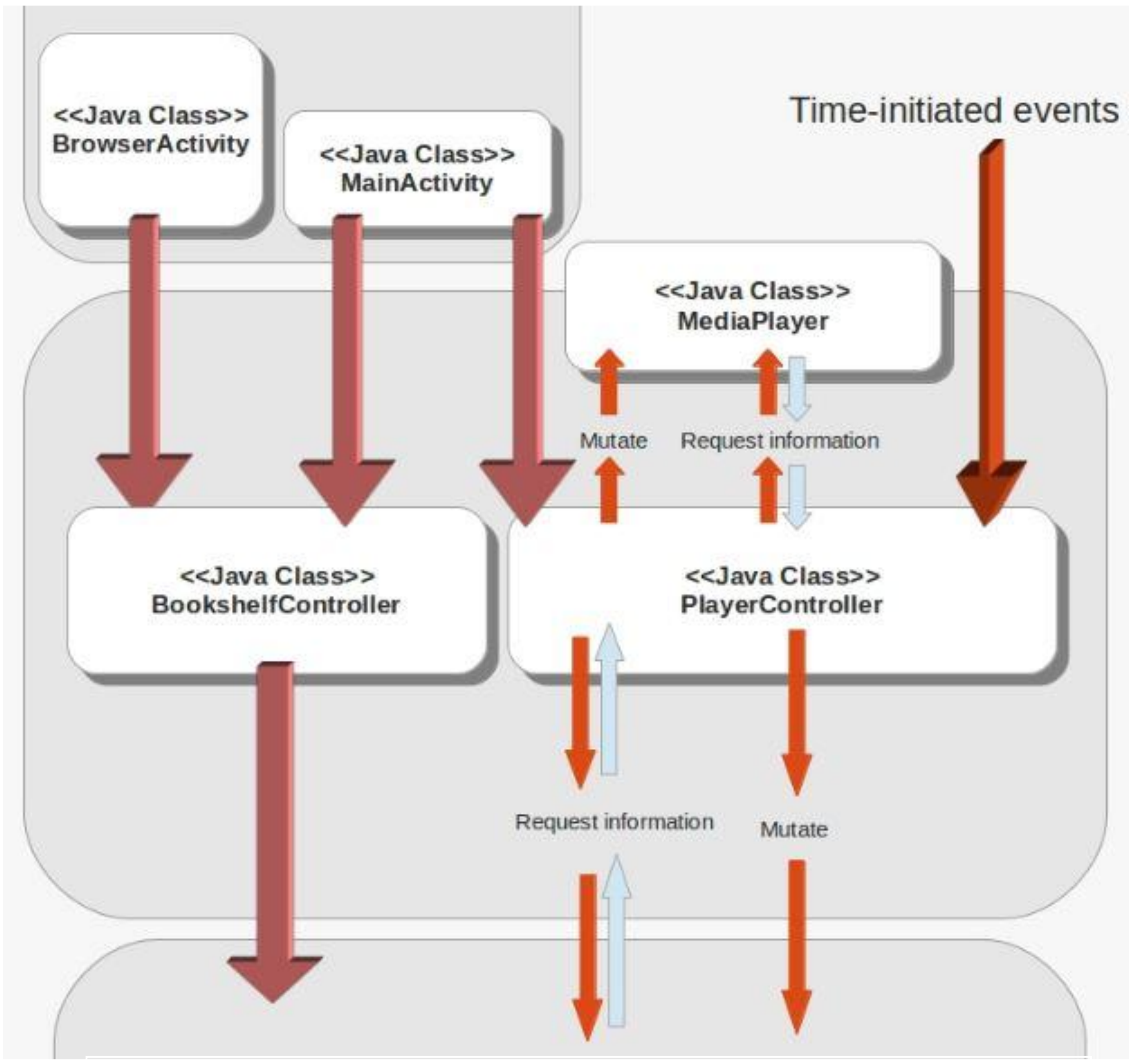

Figure 3 - Information flow chart for the controller module of the application.

The main purpose of classes contained in the controller module is to mutate the model based on the information from incoming events and in certain time intervals. This responsibility is divided between the *BookshelfController* and *PlayerController* classes:

- The *PlayerController* class provides argument-based mutation of the model by utilizing accessor methods present in model classes, as illustrated in Figure 3. Changes to the state of the application can thus be conditionally calculated based on its current values. This is necessary when manipulating the MediaPlayer object wrapped in the class as playback-related information such as paths to audio tracks are stored within the application's model. The information flow is thus bidirectional between the Bookshelf and PlayerController classes, as shown in Figure 3 and Figure 4.
- The main purpose of BookshelfController is to simply translate user-initiated events (View module events) to mutator method calls in the Bookshelf class, as seen in Figure 3. Such calls cover basic CRUD-functionality for the bookshelf (adding, viewing, updating or removing books). These operations are executed regardless of the current state in the model, why the information flow between BookshelfController and Bookshelf may be regarded as unidirectional as opposed to the case with PlayerController and Bookshelf.

2.4 Model module

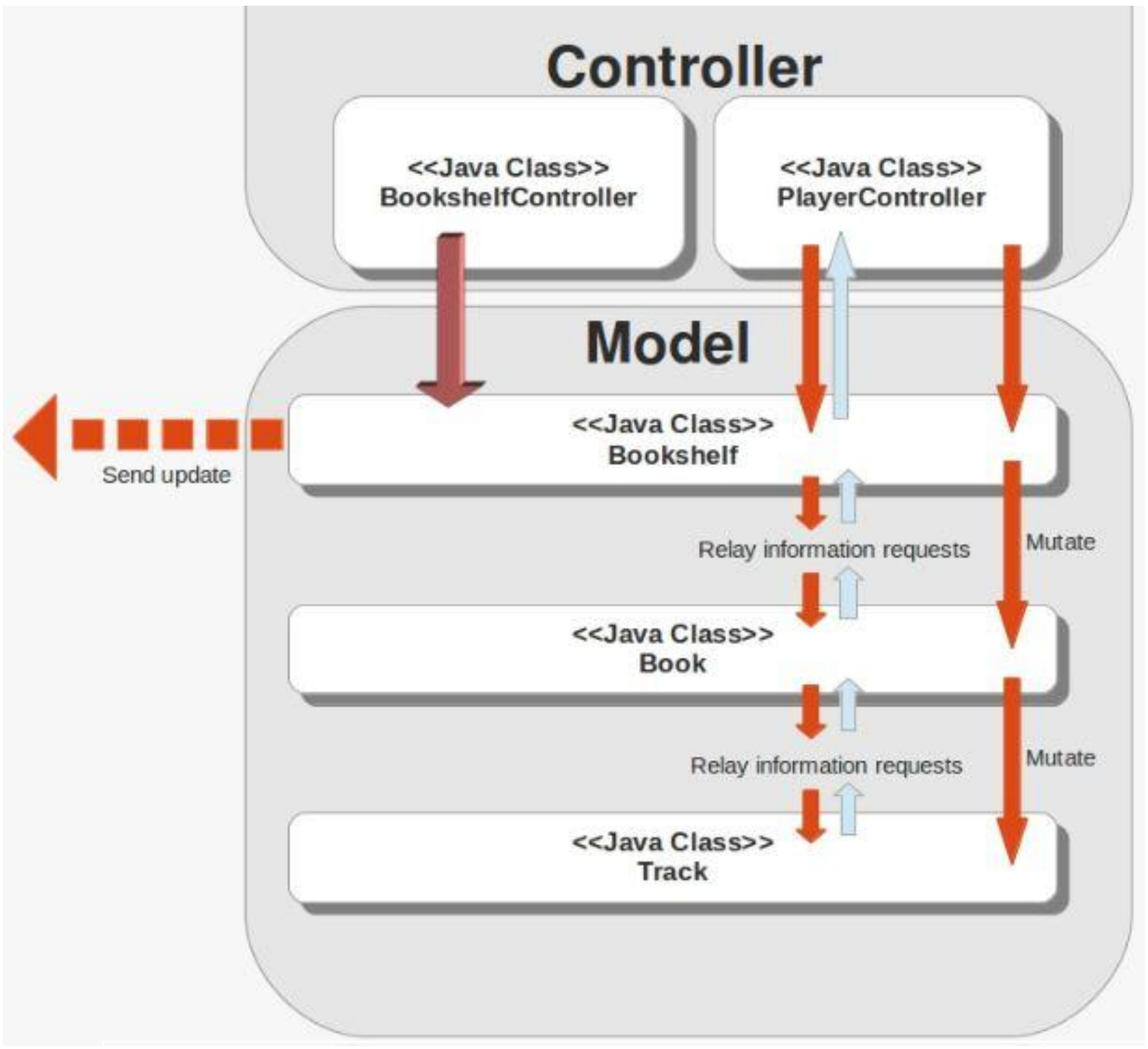

Figure 4 - Information flow chart for core classes in the model module of the application.

In accordance with fundamental principles of the MVC-architecture, the application's model module is strictly composed of classes that collectively maintain the current state of the application. Information contained in the classes that compose the model, such as elapsed time, durations and track lists, is maintained and updated by the governing controller classes through user-initiated events in the view module.

Whenever a controller class receives an event that triggers any form of change in the application, the event is translated into a mutator method call in the *Bookshelf* class. Besides containing all logic related to the current state of the user's bookshelf, this class also acts as an interface for the whole module by containing all mutation methods present in any of the classes (see the interfaces in part 3.3 below). All mutational queries sent from controller classes are thus received in the Bookshelf class, and then relayed to relevant classes further down the object tree when needed, as illustrated in Figure 4. The main purpose of this design is to create a single choke point class for all mutational queries, as this simplifies the relationship between the model and view modules by reducing the amount of classes involved in the observer design pattern communication to two classes (Bookshelf as sender, MainActivity as receiver).

3.0 UML-diagrams

3.1 View module UML

See appendix 1.

3.2 Controller module UML

See appendix 2.

3.3 Model module UML

See appendix 3.

3.4 Complete application UML

See appendix 4. Note: due to issues with the UML-generator tool, an updated version could not be created. The provided version is therefore slightly outdated, but still serves the purpose of providing insight in the relationship between the modules of the application.