POLITECNICO DI TORINO

e-Lite

JavaFX Application Structure

Tecniche di Programmazione – A.A. 2019/2020

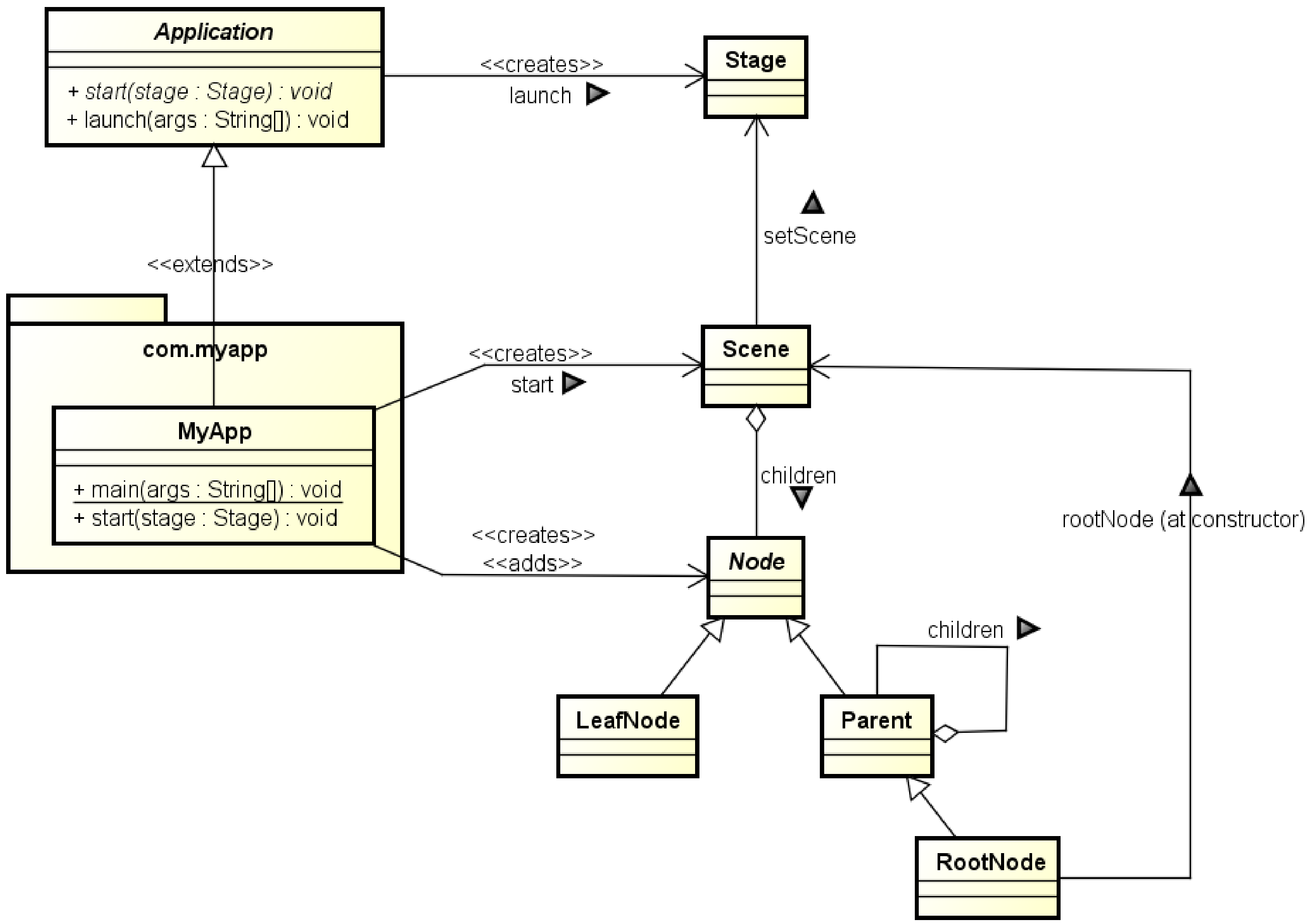
Application structure

Introduction to JavaFX

Empty JavaFX window

```
public class EntryPoint extends Application {
   @Override
   public void start(Stage stage) {
        Parent root = FXMLLoader.load(
            getClass().getResource("/fxml/Scene.fxml"));

        Scene scene = new Scene(root);
        scene.getStylesheets().add("/styles/Styles.css");

        stage.setTitle("JavaFX and Maven");
        stage.setScene(scene);
        stage.show();
   }
   public static void main(String[] args) {
       launch(args);
   }
}
```

Typical Class Diagram

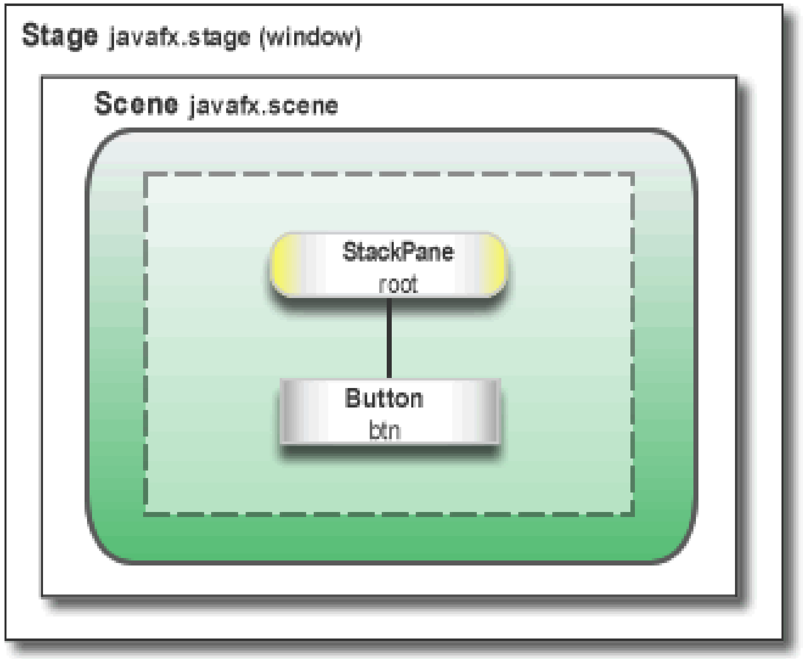


General rules

- A JavaFX application extends `javafx.application.Application`
- The main() method should call `Application.launch()`
- The start() method is the main entry point for all JavaFX applications
  - Called with a Stage connected to the Operating System's window
- The content of the scene is represented as a hierarchical scene graph of nodes
  - Stage is the top-level JavaFX container
  - Scene is the container for all content

Minimal example

```
public class HelloWorld extends Application {
    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        primaryStage.setTitle("Hello World!");

        StackPane root = new StackPane();

        Button btn = new Button();
        btn.setText("Say 'Hello World'");

        root.getChildren().add(btn);

        primaryStage.setScene(new Scene(root, 300, 250));
        primaryStage.show();
    }
}
```



Stage vs. Scene

javafx.stage.Stage

- The JavaFX Stage class is the top level JavaFX container.
- The primary Stage is constructed by the platform.
- Additional Stage objects may be constructed by the application.
- A stage can optionally have an owner Window.

javafx.scene.Scene

- The container for all content in a scene graph
- The application must specify the root Node for the scene graph
- Root may be Group (clips), Region, Control (resizes)
- If no initial size is specified, it will automatically compute it

Nodes

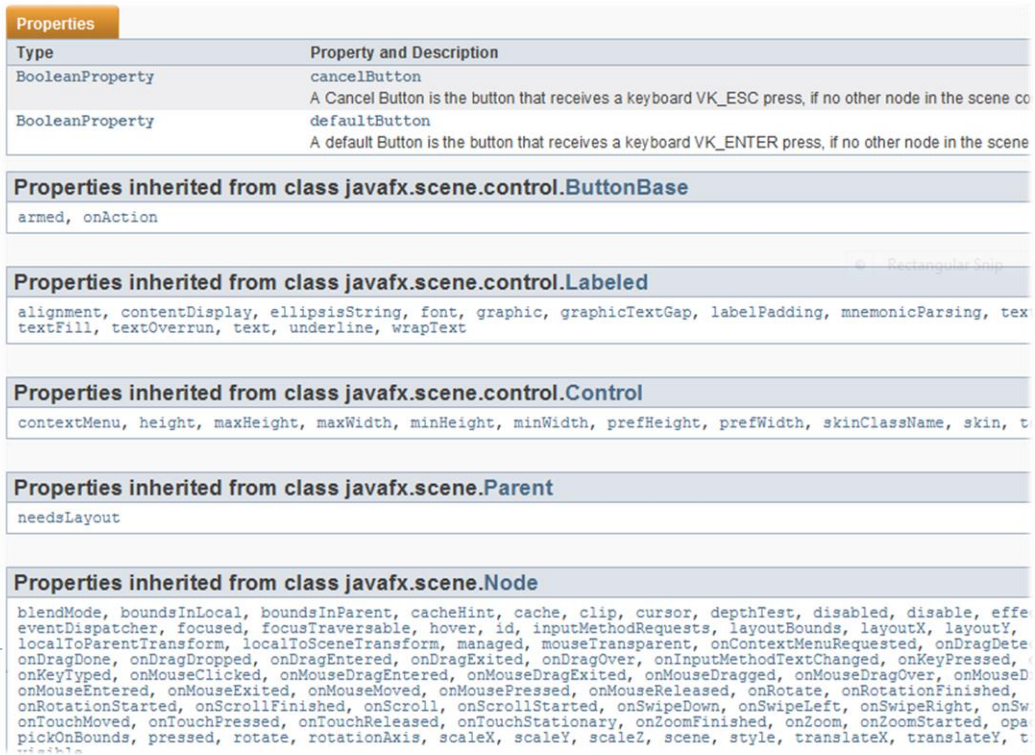
- The Scene is populated with a tree of Nodes
  - Layout components
  - UI Controls
  - Charts
  - Shapes
- Nodes have Properties
  - Visual (size, position, z-order, color, ...)
  - Contents (text, value, data sets, ...)
  - Programming (event handlers, controller)
- Nodes generate Events
  - UI events
- Nodes can be styled with CSS

Events

- FX Event (javafx.event.Event):
  - Event Source => a Node
  - Event Target
  - Event Type
- Usually generated after some user action
- Event types
  - `ActionEvent, TreeModificationEvent, InputEvent, ListView.EditEvent, MediaErrorEvent, TableColumn.CellEditEvent,TreeItem.TreeModificationEvent, TreeView.EditEvent, WebEvent, WindowEvent, WorkerStateEvent`
- You can define **event handlers** in your application

Properties

- Extension of the Java Beans convention
  - May be used also outside JavaFX
- Encapsulate properties of an object
  - Different types (string, number, object, collection, ...)
  - Set/Get
  - Observe changes
  - Supports lazy evaluation
- Each Node has a large set of Properties



Properties

Type	Property and Description
BooleanProperty	cancelButton A Cancel Button is the button that receives a keyboard VK_ESC press, if no other node in the scene co
BooleanProperty	defaultButton A default Button is the button that receives a keyboard VK_ENTER press, if no other node in the scene

Properties inherited from class javafx.scene.control.ButtonBase

```
armed, onAction
```

Properties inherited from class javafx.scene.control.Labeled

```
alignment, contentDisplay, ellipsisString, font, graphic, graphicTextGap, labelPadding, mnemonicParsing, tex
textFill, textOverrun, text, underline, wrapText
```

Properties inherited from class javafx.scene.control.Control

```
contextMenu, height, maxHeight, maxWidth, minHeight, minWidth, prefHeight, prefWidth, skinClassName, skin, t
```

Properties inherited from class javafx.scene.Parent

```
needsLayout
```

Properties inherited from class javafx.scene.Node

```
blendMode, boundsInLocal, boundsInParent, cacheHint, cache, clip, cursor, depthTest, disabled, disable, effe
eventDispatcher, focused, focusTraversable, hover, id, inputMethodRequests, layoutBounds, layoutX, layoutY,
localToParentTransform, localToSceneTransform, managed, mouseTransparent, onContextMenuRequested, onDragDete
onDragDone, onDragDropped, onDragEntered, onDragExited, onDragOver, onInputMethodTextChanged, onKeyPressed,
onKeyTyped, onMouseClicked, onMouseDragEntered, onMouseDragExited, onMouseDragged, onMouseDragOver, onMouseD
onMouseEntered, onMouseExited, onMouseMoved, onMousePressed, onMouseReleased, onRotate, onRotationFinished,
onRotationStarted, onScrollFinished, onScroll, onScrollStarted, onSwipeDown, onSwipeLeft, onSwipeRight, onSw
onTouchMoved, onTouchPressed, onTouchReleased, onTouchStationary, onZoomFinished, onZoom, onZoomStarted, opa
pickOnBounds, pressed, rotate, rotationAxis, scaleX, scaleY, scaleZ, scene, style, translateX, translateY, t
```

Bindings

- Automatically connect («bind») one Property to another Property
  - Whenever the source property changes, the bound one is automatically updated
  - Multiple bindings are supported
  - Lazy evaluation is supported
  - Bindings may also involve computations (arithmetic operators, if-then-else, string concatenation, ...) that are automatically evaluated
- May be used to automate UI
- May be used to connect the Model with the View

Model-View-Controller

JavaFX programming

Application complexity and MVC

- Interactive, graphical applications exhibit complex interaction patterns
- Flow of control is in the hand of the user
- Actions are mainly asynchronous

- How to organize the program?
- Where to store data?
- How to decouple application logic from interface details?
- How to keep in sync the inner data with the visibile interface?

Design Patterns

O'REILLY®

10th Anniversary Updated for Java 8

Head First Design Patterns

A Brain-Friendly Guide

Avoid those embarrassing coupling mistakes

Learn why everything your friends know about Factory pattern is probably wrong

Discover the secrets of the Patterns Guru

Load the patterns that matter straight into your brain

Find out how Starbuzz Coffee doubled their stock price with the Decorator pattern

See why Jim's love life improved when he cut down his inheritance

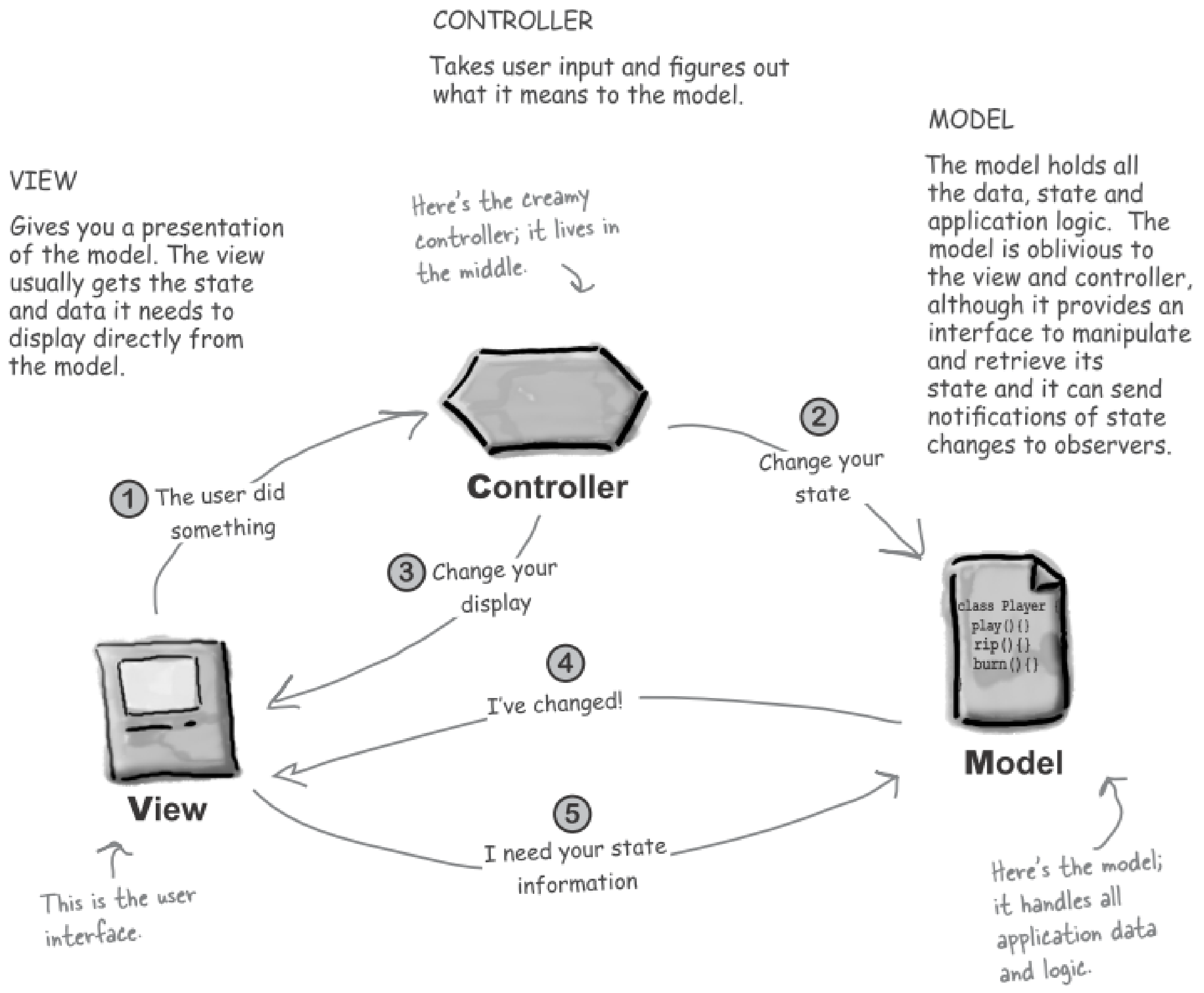
Eric Freeman & Elisabeth Robson

with Kathy Sierra & Bert Bates

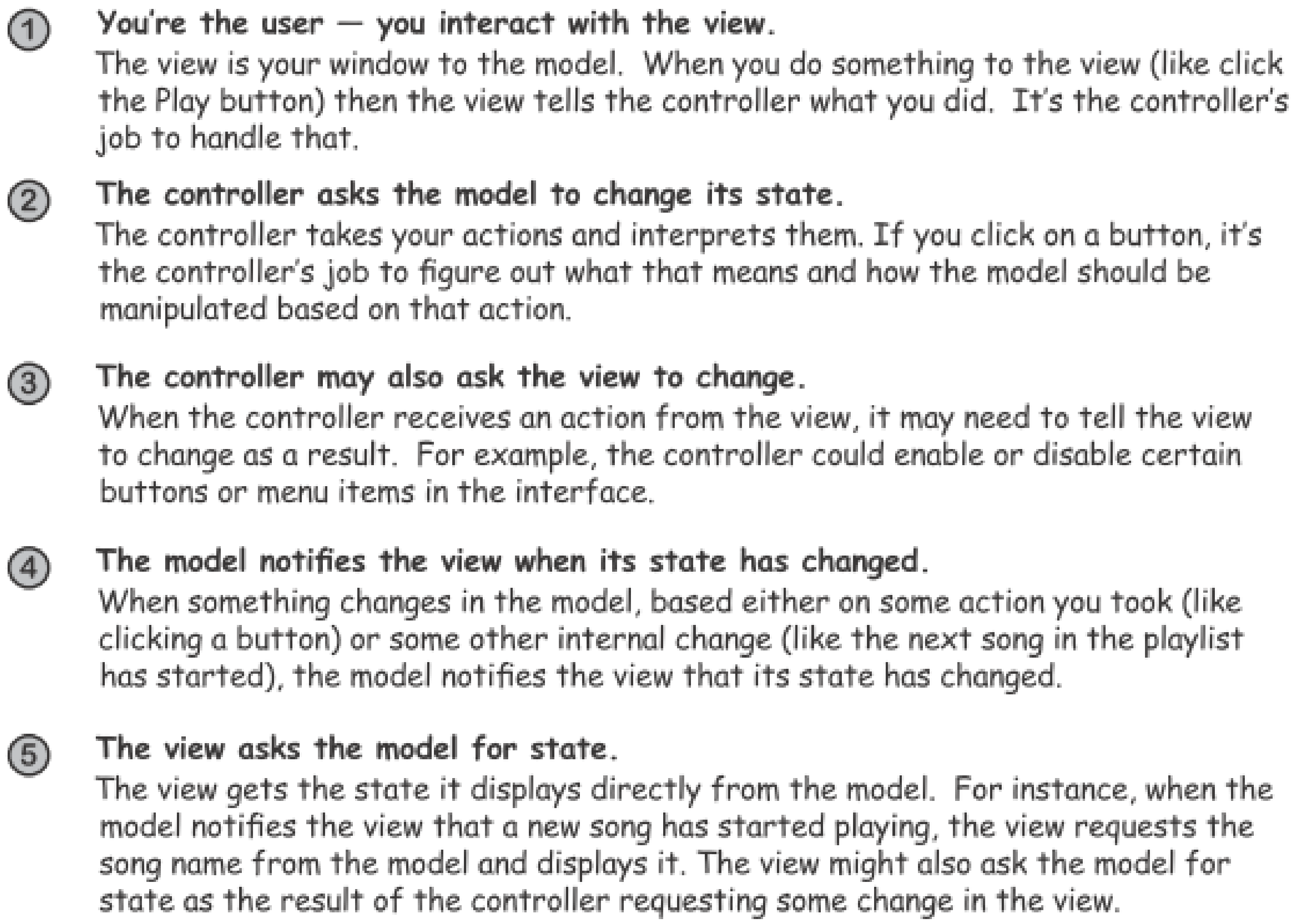
Design Patterns

- How to build systems with good OO design qualities
  - Reusable, extensible, maintainable
- Patterns: Proven solutions to recurrent problems
  - Design problems
  - Programming problems
- Adopt and combine the OO constructs
  - Interface, inheritance, abstract classes, information hiding, polymorphism, objects, statics, …
- Help dealing with *changes* in software
  - Some part of a system is free to vary, independently from the rest

MVC pattern defined


CONTROLLER
Takes user input and figures out what it means to the model.
MODEL
The model holds all the data, state and application logic. The model is oblivious to the view and controller, although it provides an interface to manipulate and retrieve its state and it can send notifications of state changes to observers.
VIEW
Gives you a presentation of the model. The view usually gets the state and data it needs to display directly from the model.
Here's the creamy controller; it lives in the middle.
Controller
1 The user did something
2 Change your state
3 Change your display
4 I've changed!
5 I need your state information
class Player
play() {}
rip() {}
burn() {}
Model
View
This is the user interface.
Here's the model; it handles all application data and logic.

Normal life-cycle of interaction



1. **You're the user — you interact with the view.**
The view is your window to the model. When you do something to the view (like click the Play button) then the view tells the controller what you did. It's the controller's job to handle that.

2. **The controller asks the model to change its state.**
The controller takes your actions and interprets them. If you click on a button, it's the controller's job to figure out what that means and how the model should be manipulated based on that action.

3. **The controller may also ask the view to change.**
When the controller receives an action from the view, it may need to tell the view to change as a result. For example, the controller could enable or disable certain buttons or menu items in the interface.

4. **The model notifies the view when its state has changed.**
When something changes in the model, based either on some action you took (like clicking a button) or some other internal change (like the next song in the playlist has started), the model notifies the view that its state has changed.

5. **The view asks the model for state.**
The view gets the state it displays directly from the model. For instance, when the model notifies the view that a new song has started playing, the view requests the song name from the model and displays it. The view might also ask the model for state as the result of the controller requesting some change in the view.

Mapping concepts to JavaFX

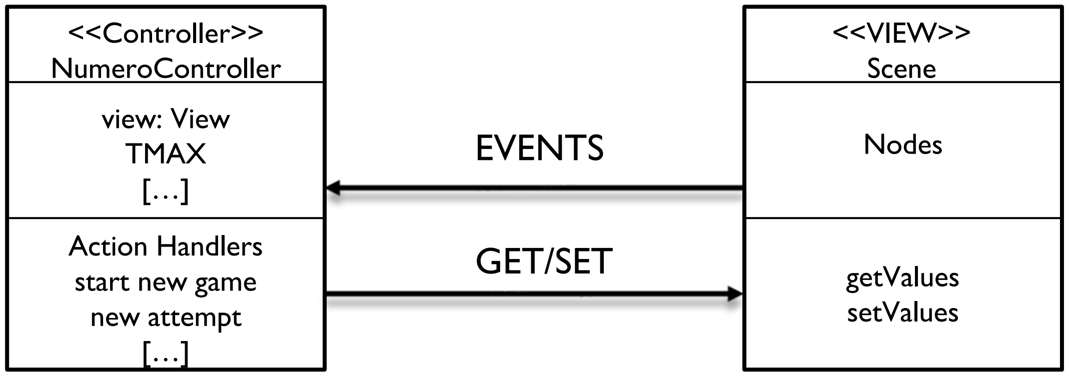
- View: presenting the UI
  - FXML
  - The Nodes in the Scene Graph
- Controller: reacting to user actions
  - Set of event handlers
- Model: handling the data
  - Class(es) including data
  - Persistent data in Data Bases

Exercise

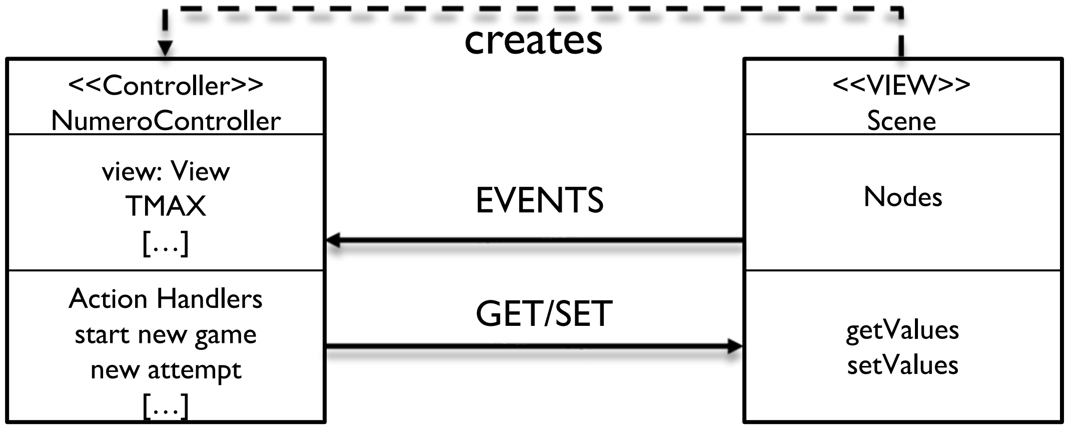
- Update the «IndovinaNumero» by using the MVC pattern

- Where do you declare the data class?
- Which class should have access to which?
- Who creates what objects?

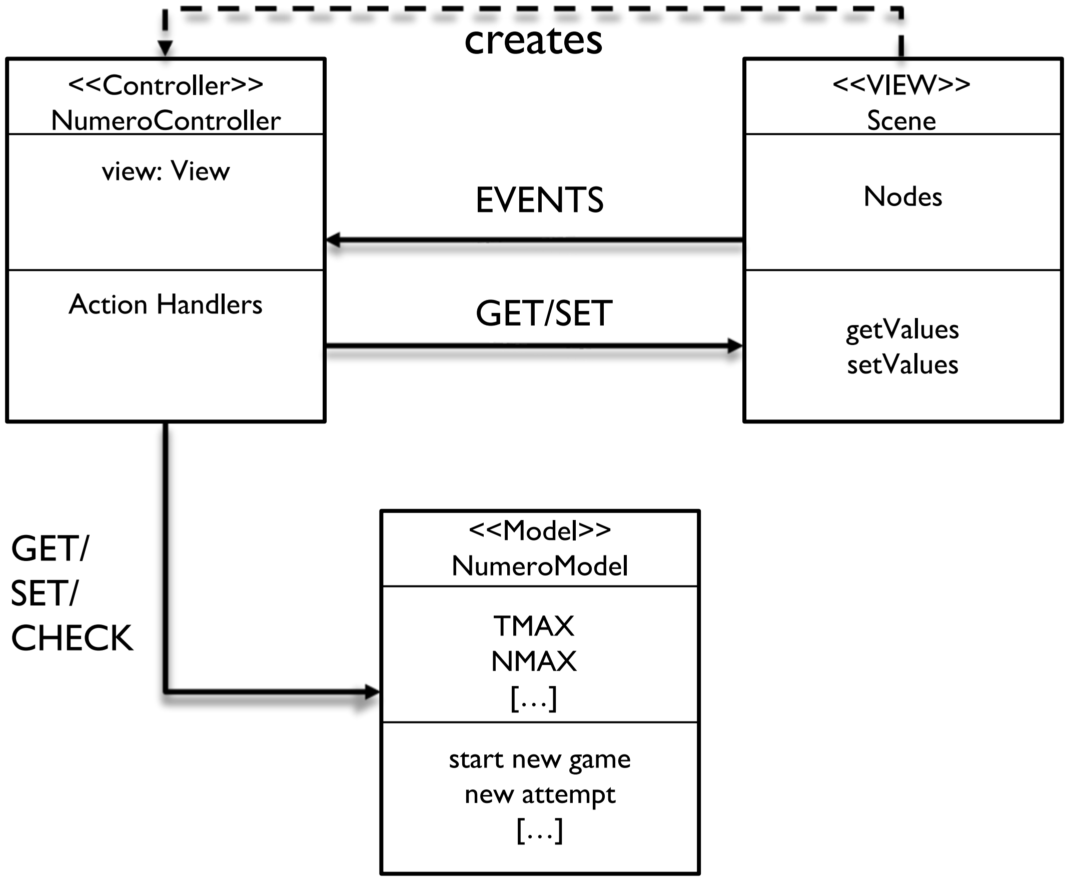
A possible solution



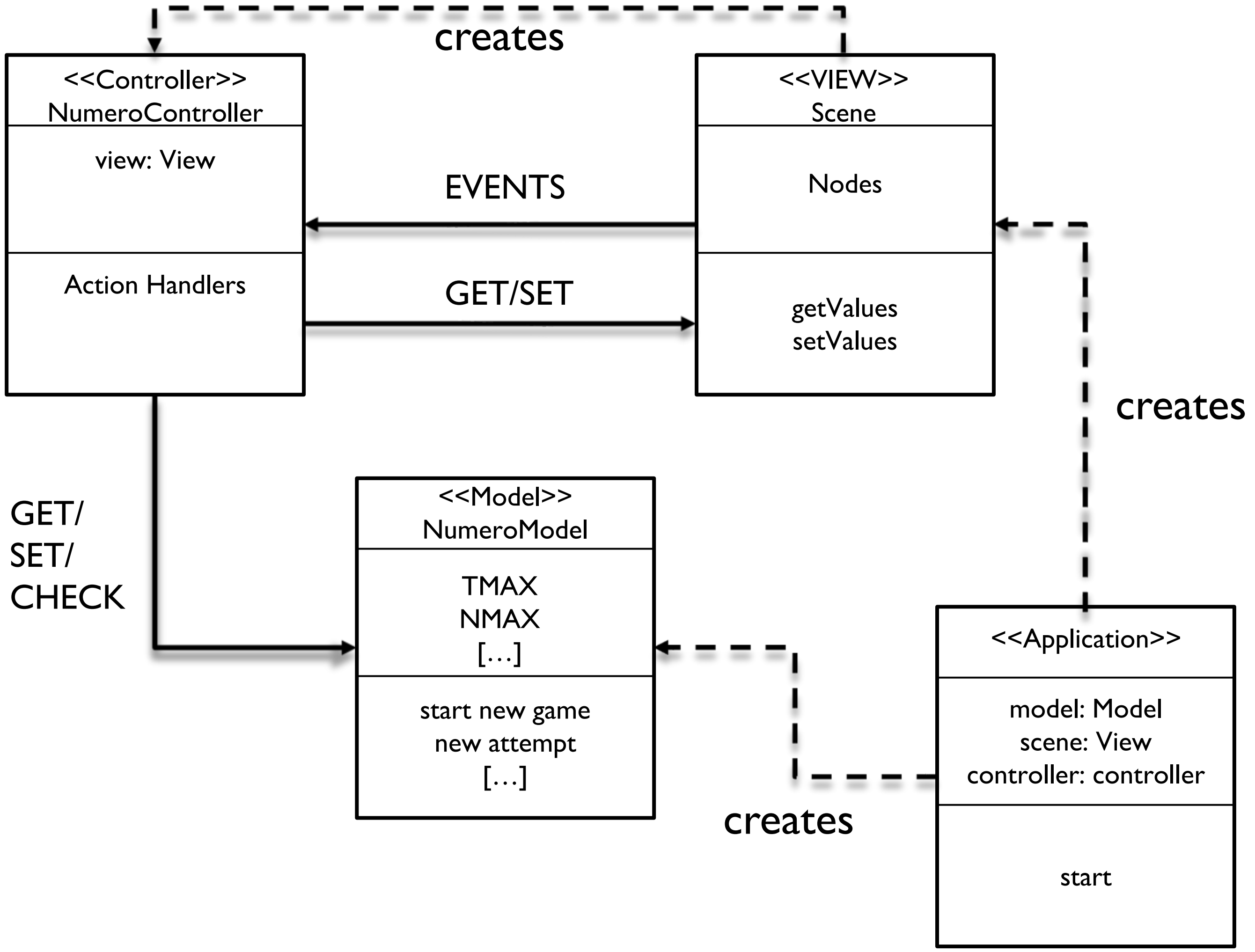
A possible solution



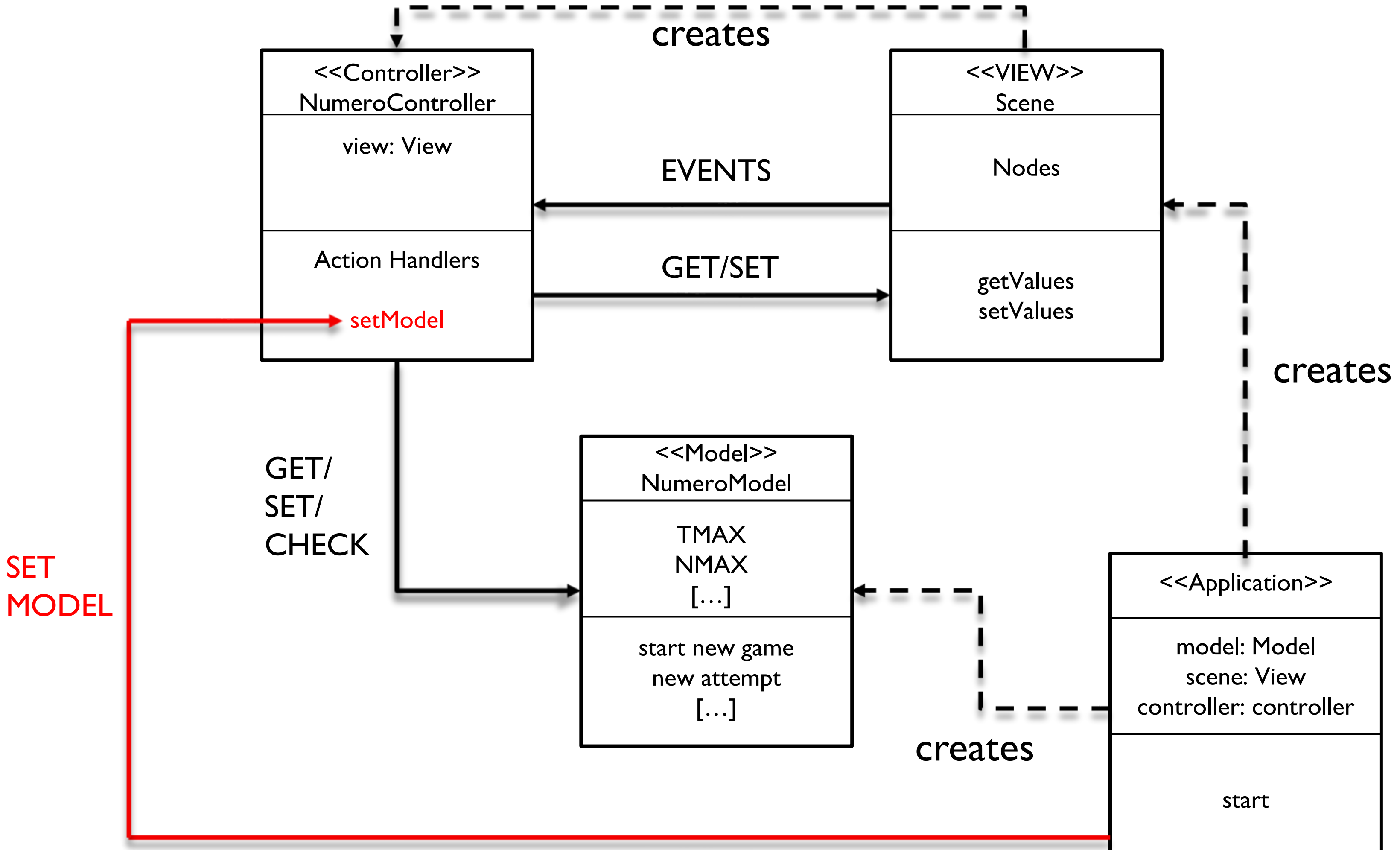
A possible solution



A possible solution



A possible solution



Licenza d'uso