Adrenaline

Laneve L., Lopardo A., Mencucci R.

Design Notes

Contents

1 Server Structure

1.1 The `ServerInstance` object

A `ServerInstance` consists of the following components:

- A User Registry, which mantains a list to all the connected users and deals with user authentication;
- A Lobby Registry, which keeps a reference to all the open lobbies in the server and assigns new and reconnected users to a lobby;
- A User Greeter, which waits for new connections from clients (discussed in section 4.1);

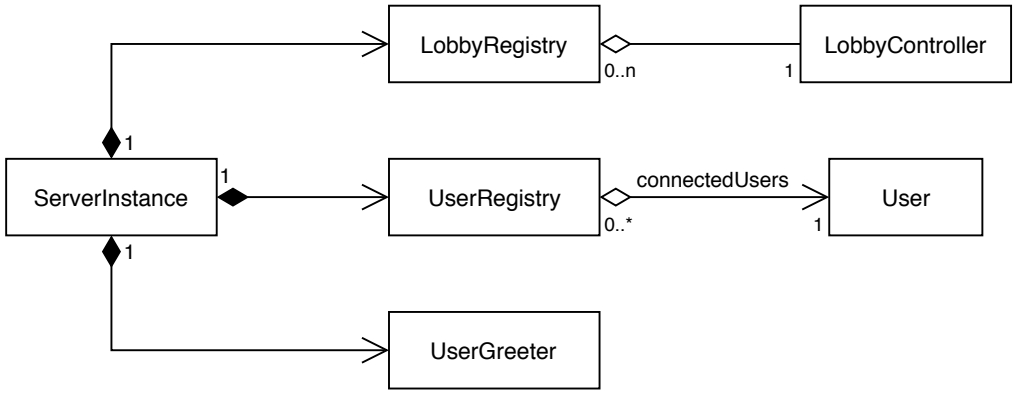

Figure 1.1: Simplified class diagram for the server structure

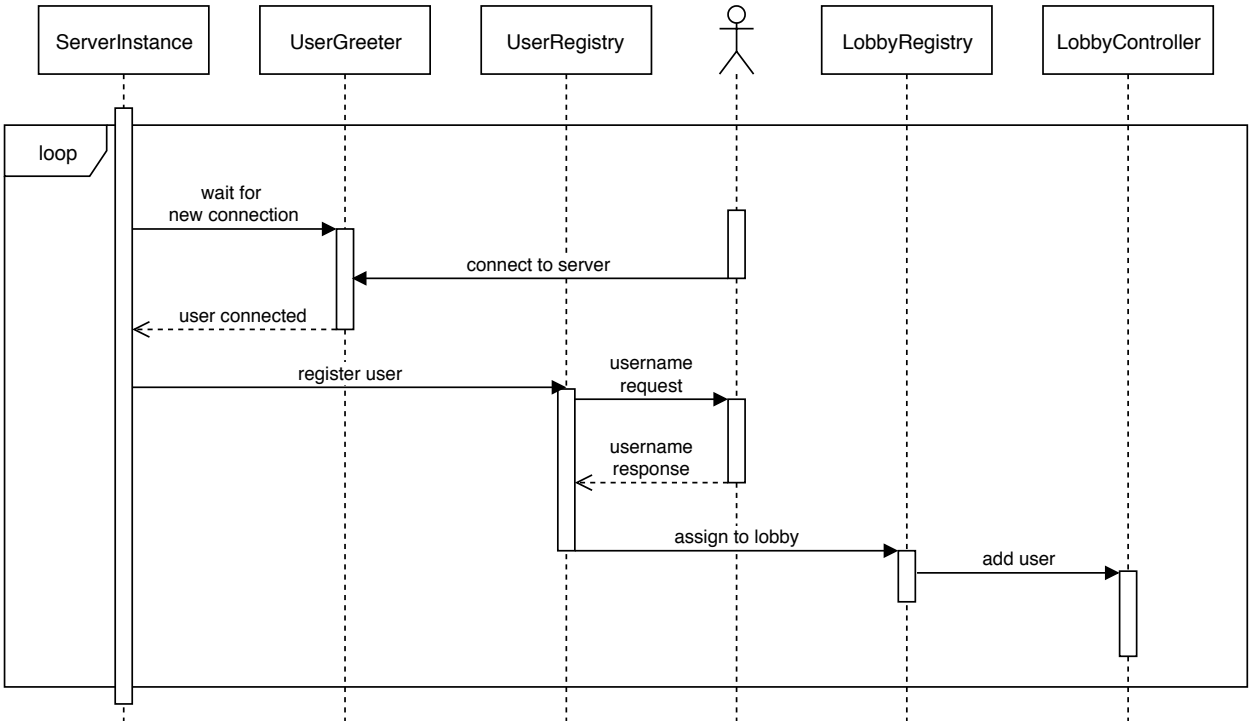

Figure 1.2: Sequence diagram showing the acceptance of an incoming user

1.2 Lobby Controller and Game Controller

While the Lobby Controller manages the lifecycle of a Lobby (waiting for users, starting the game...), the Game Controller manages the life cycle of a Game.

It is worth noting that the Game Controller is the only controller component of the server which is **game-specific**: all the already mentioned components are game-agnostic and the creation of the game controller by the lobby controller is done with the use of a **Factory pattern**.

The lobby controller also consists of a sandbox for the game controller: while the lobby is hosting a game, the lobby controller still listens for user disconnections/reconnections, and will eventually report these events to the enclosed game controller.

When the game is over, the game controller will notify the enclosing lobby controller, which then will dispose all the resources and remove itself from the lobby registry.

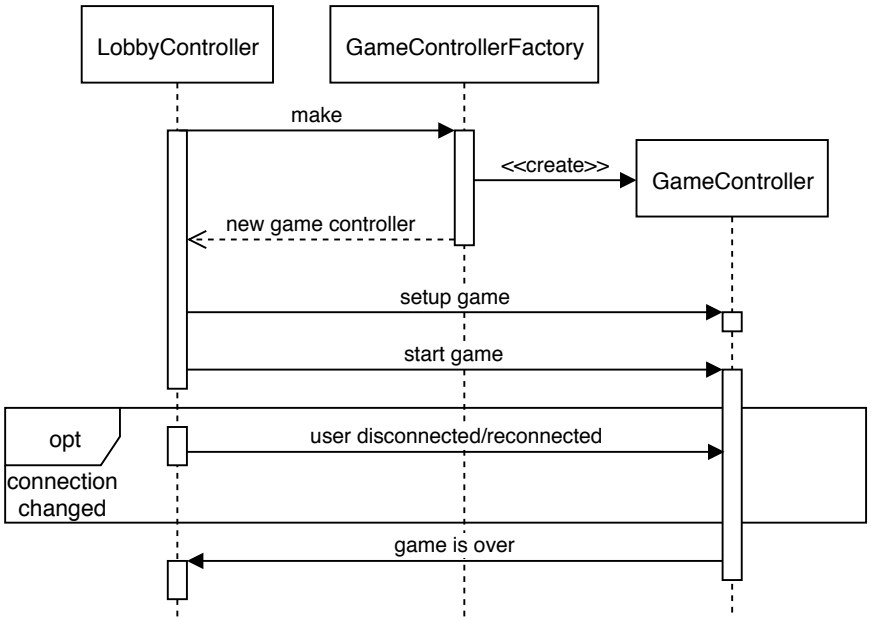

Figure 1.3: Interaction between lobby and game controller

2 Weapon Cards and Power Up Cards

2.1 The `Effect` interface

The base weapon and power up decks of Adrenaline include a variety of cards with different effects, and a language that can conveniently express all the cards touches the Turing-completeness.

So we decided to create an functional interface `Effect`, and every weapon and power up will implement this interface. We used **Java Reflection** to allow us to specify the name of the Effect class associated to a particular card. For example, in the `weapons.json` file used by the server, we have the following data for the *Hellion* weapon card:

```
{
  "id": 11,

  "pickupPrice": {
    "red": 0,
    "blue": 1,
    "yellow": 0,
    "any": 0
  },

  "reloadPrice": {
    "red": 0,
    "blue": 2,
    "yellow": 0,
    "any": 0
  },

  "effectClass": "it.polimi.deib.newdem.adrenaline.controller.effects
                  .HellionEffect"
}
```

This, along with the class added to the server's Java classpath, is sufficient to fully define the card. This allows to add new cards by simply **add a new implementation of effect** and reference it in the JSON file as shown above, fully following the Open-Closed principle.

2.2 Effect Manager and Context

Effect programming is enclosed by a sort of sandbox that provides methods to interact with the game and the active user. This sandbox is defined by the `EffectManager` class.

Both effects and the effect manager are agnostic to the game. They need a **context** to interact with, which is defined by the `EffectContext` class. Please refer to `EffectContext` JavaDoc to the various information and features it provides.

3 `Turn` and humans

3.1 Design process and motivation

In a game of Adrenaline, a `Turn` encapsulates all the actions and interactions that a human can have with the game. It allows `Game` and other Model elements to disregard all the complexity that comes with interacting with humans and better focus on their designed role. As stated in the Adrenaline rules manual, a player during their turn may take a certain number of some actions shown in their boards. These gameplay elements are represented in the codebase by the `Action` interface. `Turn` handles directly only the execution of the appropriate number of `Actions`, delegating further finer handling to other components.

It quickly became apparent that Adrenaline's actions have a modular nature: each action can be broken down into a few smaller, simpler components, which can then be combined together to build all the actions shown in the action board with ease. One such component is the one representing movement within a given distance, appearing in many different actions followed by various other components and even on its own. In the codebase, we called these components `Atoms`, assuming they were the simplest element an action could have. We later found our assumption to be incompatible with later design decisions.

Once we have established that an `Action` can be broken down into `Atom`s, we need to know how to put the `Atom`s back together to represent an `Action` usable by `Turn`. Luckily, Adrenaline makes this easy on us: the components of a gameplay action never change during the execution of said action and are always executed in a fixed order. This allows us to represent them with a data structure reminiscent of Finite-State Automata, which can be dynamically created and destroyed as needed but will not change their structure during their execution. Let's assume for now that all atoms will be executed correctly: this allows us to control trivially the finite state machine, moving from one state to the next until we conclude in the accept state. We will talk later about handling some non-idealities.

Our initial implementation was based on the components mentioned above: actions and atoms, represented by their homonymous classes. Each `Atom` had an implementation detailing how it should have handled user interaction, with the assumption that all user inputs *and gameplay action* be correct.

This initial implementation did work as intended, allowing us to start running games and test other parts of the project, but it proved to be *de facto* unusable by humans. Forcing a human to play in such a way that an illegal action is never attempted was not a reasonable expectation of a player and we should not have taken it for granted, in spite of it being allowed by our tutors. Furthermore, there exists a timeout which can interrupt the turn at any time. While designing `Model`, we assumed that at any given time the game will be in a legal state, even after an interruption by a timer. Ensuring such a constrain proved to be unnecessarily difficult with our monolithic implantation of `Atom`, which had to handle too many different cases all on its own.

While playing the game, we were inspired by our human decision process in choosing what to do in a turn: we would play the game out in our head, see if we liked the end result and then decide to go through with it or not. Similarly, if we realize we're stuck in a situation with an impossible action or an undesired outcome, we would go back to the last decision we made that lead us to such dead end and change it. We found simulating and reverting elements of a turn similar to Push Down Automata, and got inspiration to proceed.

Now that we have established our desire to support rollback of at least part of a game, we need to know when to do so. It then came to our attention that a player may decide, as we said before, to undo any one choice they made, but our monolithic `Atom` can allow more than one choice. One good example of this is a grab atom executed while the active player is on a spawn point, asking for a new weapon to pick up, to pay its cost and – potentially – to discard a card from the player's inventory. Furthermore, a human may try to attempt and undo a choice as many times as they desire, making an implementation based on FSA for `Atom` incapable of expressing this problem. This meant that our initial implementation of `Atom` could not easily be adapted to fit our new design choice and a complete rework was needed.

Let's now focus on how the previous example of a grab atom should work, disregarding the outer complexity of `Turn` and `Action`. We need to ask the user what weapon they wish to pick up, how to pay for it and if necessary which weapon to discard. All of these interactions need to be individually reversible in our new design. This leads us to the concept of `Interaction`s, which encapsulate one singular step in processing game logic and may require up to one choice from a human player.

3.2 Control flow of a `Turn`

`Interaction`s are the new way in which an atom resolves itself. When the `Atom` begins execution, a new `Interaction` is created from that `Atom`'s entry point and pushed on an interaction stack. At this point, the control flow passes to the `Interaction`. If no user interaction is required, the `Interaction` executes its duty quickly and pushes one or zero new `Interaction`s on the stack. Once the current `Interaction` terminates successfully, the calling `Atom` takes back control and looks at the top of the interaction stack. If it has changed since the last execution, it will execute whatever is now on top, otherwise the atom's execution will be declared complete and the atom's execution will conclude, returning the control flow to the calling

`Action`. At this point, the `Action` will process like an FSA any other atoms left and terminate after the last one.

`Interaction`s come in many different versions: some do not require user interaction and simply check some game elements to determine how to proceed in the resolution of an atom (at the beginning of a grab interaction is the player on a spawn point or on a drop tile), some require player input. Of the latter, some allow a user to select none of the provided options, while others do not. Some interaction need to propagate a value to their children to ensure successful application and reversion of the desired outcome. All of these details are highly dependant on the individual interaction and are what classes implementing the `Interaction` interface must define appropriately.

Whenever a user is given a choice, they may answer they wish to undo the previous choice they made. In the codebase, this is represented by an `UndoException`. An `Interaction` may receive an `UndoException` directly from the call to a player's choice or may interpret an otherwise legal answer as an undo depending on the logic associates to that particular gameplay element. Once an interaction receives an `UndoException` it will ensure that the model is in a legal state. Then, it will propagate the exception to the calling `Atom`, which will pop from the interactions stack the running `Interaction` which threw the exception and will keep popping interactions off the stack up until the last `Interaction` which required user input. This last `Interaction` which will be reverted ad then executed again, asking the user for their choice as if it were the first time and populating further choice in the interaction stack normally. If an `Atom` attempting to revert its interactions stack finds it to be empty it may further propagate the `UndoException` to the calling `Action`. The `Action` will then attempt to run the previous `Atom`, after reverting it with the same logic as the one that threw the exception so that the user will see first the choice they made last in the previous `Atom`.

3.3 Interruptions and additional actions

Targeting scopes and tagback grenades have been challenging, as they are played during an action, breaking the hypothesis we made before. We are happy with our solution to this problem, which not only works for the cards currently in the game, but may allow us to easily add a new interrupting game element.

Targeting scopes are implemented without interruption. Once a player has completed shooting with a weapon, a new `AdditionalDamageInteracion` is pushed on the interactions stack. This `Interaction`'s job is to check if the active player may use a scope and, if so, to allow them to do so. This does not break the previously described control flow and has not been exceedingly difficult to add in code.

Tagback grenades were challenging. They require that we take control away from the active player, give it to some other player and then resume the current player's turn. We achieved this by recording all players eligible for a revenge during the resolution of any `Effect` and, as part of the resolution of said effect, a trigger temporarily pushed all damaged player on an opportune stack in `TurnDataSource`, making them each one at a time the target of the following `Action` to use the revenge card.

As hinted at in the previous paragraph, it's possible to run `Action`s beyond what's written on a player's boards. This is the case with the aforementioned targeting scope and tagback grenades, but also with other specific cases, such as powerups at end of turn or reload at end of turn.

4 Network Connections

4.1 The User Greeter and Incoming User Modules

The user greeter is an object that waits for incoming connections to the server and returns them to the server instance.

Different and multiple `IncomingUserModule` objects can be added. These modules represent a source of users, each implementing and managing a different way for users to connect with the server.

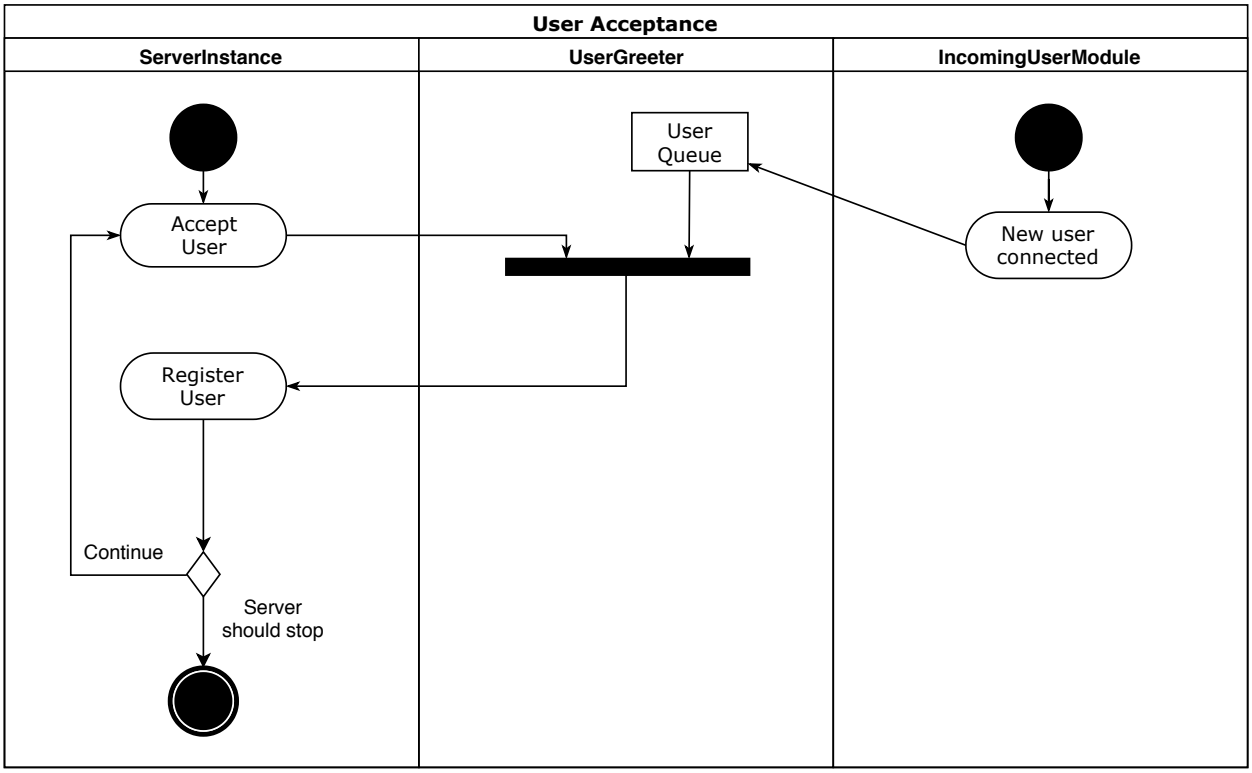

Figure 4.1: Activity diagram of the acceptance of users

In the current implementation, the User Greeter itself starts a thread for each module that continuously keeps on calling `IncomingUserModule#newUser()` and pushes the returned user into the queue.

4.2 The `UserConnection` interface

A user connection consists of an abstract connection that takes advantage of the **Publisher-Subscriber pattern**. The messages are expressed by the `UserEvent` interface and its subclasses. New types of messages can be added by simply subclassing this interface, no modification of the code of connections are required, fully following the Open-Closed principle.

Objects that want to subscribe to an event `T` must implement the functional interface `UserEventSubscriber<T>` and use `UserConnection#subscribeEvent(Class<T>, UserEventSubscriber<T>)` to start receiving events from that connection.

The interface `UserConnection` is implemented by the abstract class `UserConnectionBase`, which implements the common features of publishing/subscribing of events and behaviours that are agnostic to the network technology, such as Heartbeats (which will be discussed in Section 4.5).

4.3 Socket Connections

Socket user connections are implemented through the use of Java Sockets and the `ObjectInputStream` and `ObjectOutputStream`.

The `ServerSocket` object is held by the `SocketUserModule`, which implements the `IncomingUserModule` interface.

4.4 RMI Connections

The RMI architecture that underlies the RMI user connections consists of two types of objects:

- The `RMIServerGreeter` object, which is the only object bound to the server's RMI registry and that is called by connecting clients;
- The `RMIEndpoint` object, which defines a method that allows the other side to read events (through an event queue);

When a client connects to a server, passes its end point to the server greeter through `ServerGreeter#connect(RMIEndpoint)`, which will generate and return the end point of the server. At this point, both server and client will create the user connection with the couple of end points remote-local they have, and start to read/write events. A scheme of the event exchange is illustrated by the following diagram:

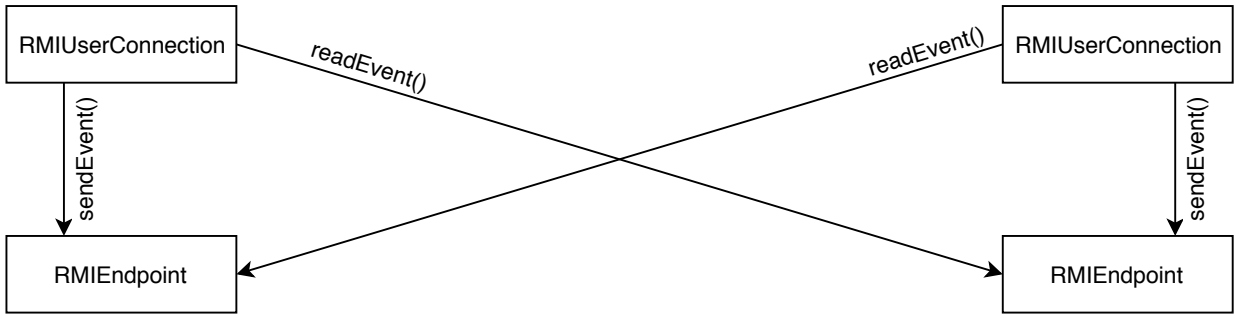

Figure 4.2: Architecture of an RMI connection

4.5 Heartbeats

Since idle TCP connections hinders the responsiveness of the system against network failures, we needed a way to check this failure at application level. This responsiveness is achieved through the use of **heartbeats**.

An heartbeat is a special event (defined by the `HeartbeatEvent` class) that must be sent by all the user connections periodically. If a user connection does not receive an heartbeat from its counterpart within a 5 seconds time period, then it will conclude that the connection has failed and will automatically close the connection.

5 From the Server to the Client

5.1 From the model to the user interface

When the state of the application is modified, an event is generated and sent to the views passing through multiple passes:

- Model objects hold a `Model Listener` object, and not a view. This improves decoupling between the model and the views;
- **Virtual Views** implement the Listener objects and transforms model data into plain, model-independent data which is sent to the client;
- the **Client Instance** handles the network connections and forwards the event to the **UI Views**, which will display the change to the user appropriately;

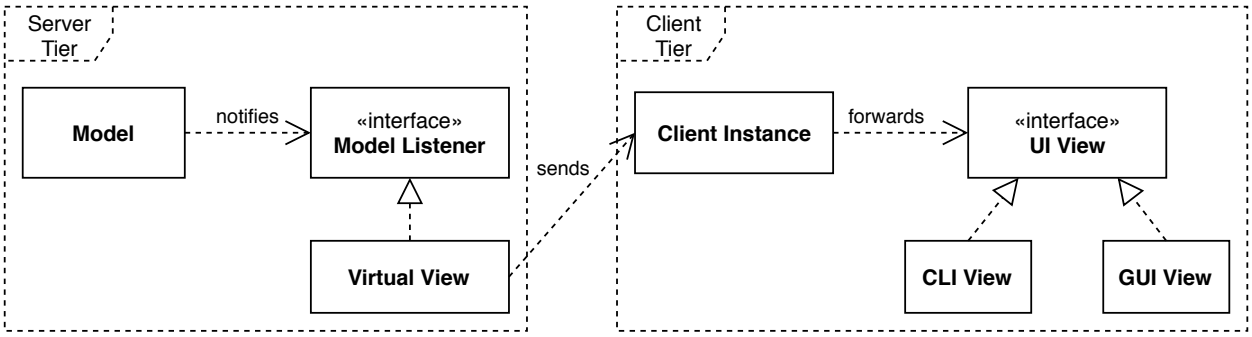

Figure 5.1: General scheme of the views

5.2 The Virtual View

The virtual view is the only view held by the server. These views uses of the connection objects described in chapter 4 to send/receive events that are translated from/will translate to events in the server.

These objects also filters the data that each client can and cannot see (e.g. power ups owned by other players).

5.3 The ClientInstance object and the ViewMaker abstraction layer

The `ClientInstance` class defines the control flow of the client, which consists of the creation and management of the connection to the server, and the forwarding of the various events between the server and the user interface.

It is worth noting that the client instance is **independent** from the user interface: the interactions is made by the `ViewMaker` interface, whose implementations will create views that are UI-specific.

This perfectly separates the client components into UI-specific and UI-agnostic components. Who starts the client will just need to create a new `ClientInstance` with the `ViewMaker` implementation of the chosen UI. New types of user interfaces can be created without modifying the UI-agnostics components of the client.