A Framework for Developing 2D Games

Liam Power	**16148983**
Eric Nolan	**15150305**
Kyran Healy	**16158598**
Daniel Keighley	**16185153**

UNIVERSITY of LIMERICK
OLLSCOIL LUIMNIGH

CS4227 Project
J.J. Collins

Computer Science and Information Systems
University of Limerick

Contents

1 Requirements **5**
1.1 Use Case: Client Developers 5
1.2 Use Case: Customers 5
1.3 Quality Attributes 7
1.3.1 Performance 7
1.3.2 Extensibility 7
1.3.3 Flexibility 8

2 Design **9**
2.1 Entity-Component-System 9
2.1.1 Entities 9
2.1.2 Components 9
2.1.3 Systems 10
2.2 Interceptors 12
2.2.1 Context Objects 12
2.2.2 Concrete Interceptors 13
2.2.3 Dispatchers 14
2.3 Game Saves 14
2.4 User Input 15
2.4.1 Invoker 15
2.4.2 Command Design Pattern 15
2.5 Prototype 17
2.6 Abstract Factory 17
2.7 Builder 18

3 Architecture **20**
3.1 Framework 20

3.2 Game 22

4 Implementation **23**
4.1 Interceptor Dispatcher 23
4.2 CS4125 Code 24
4.3 Space Invaders Game 25
4.3.1 Score System 25
4.4 UI Framework 26

5 Added Value **29**
5.1 Repositories 29
5.2 Reflection-based Interceptors 30

6 Testing **32**
6.1 Unit testing with JUnit4 32
6.1.1 Entity Creation Test 32
6.1.2 Interceptor Test 33
6.1.3 Component Consumer Test 35
6.1.4 Command Test 36

7 Evaluation **37**
7.1 Known Issues 37
7.1.1 Java Generics Type Erasure 37
7.1.2 Save Game Loading 38
7.2 Suitability 39
7.3 Potential Improvements 40
7.3.1 LambdaMetafactory and Reflection Calls 40
7.3.2 EnumMap 40

8 Contribution **41**
8.1 Lines of Code Per Class 41
8.2 Lines of Code Per Contributor 43
8.3 Report 43

Index of Design Patterns

1 Entity-Component-System 9
2 Interceptor 12
3 Repository 14
4 Memento 14
5 Adapter 14
6 Command 15
7 Prototype 17
8 Abstract Factory 17
9 Builder 18

Abstract

We present an implementation of a simple 2D game framework in Java designed to mimic an arcade game machine or games console — end users may load games into the application and client developers may write new games using our exposed APIs. We demonstrate the capabilities of this framework through an example game, a simple *Space Invaders* clone. In our implementation, we show how the latest features of the Java programming language may be used to enhance code expressiveness, safety and flexibility by incorporating concepts from functional programming into traditional design patterns and best practices.

Chapter 1: Requirements

Typically, games consoles support a variety of different games written by many different client developers. It is the responsibility of the console manufacturers and system programmers to build a foundation on which many different kinds of games may be built. Our project attempts to simulate this scenario by exposing a framework which may be used to develop different kinds of 2D games.

Broadly, we envision two distinct use cases for this project: client game developers using our framework to produce and publish their own games, and the customers actually playing these games.

1.1 Use Case: Client Developers

Using our framework, third-party game developers may create and publish their own 2D games. Though not explicitly supported in the delivered project, the intent is that third-party developers will produce and sell software artefacts on our system (similar to the Steam store or PlayStation store).

1.2 Use Case: Customers

Players may log into our system and explore the various games they have purchased. In our project deliverable, we showcase this feature through the example Space Invaders game.

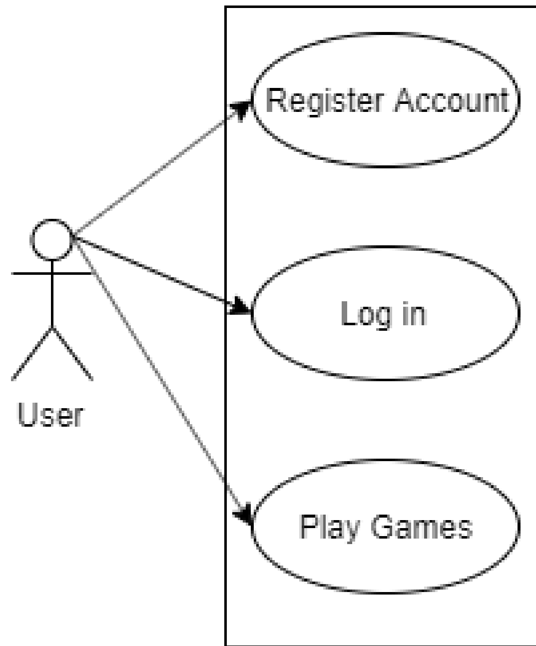

Figure 1.1: Use case diagram for our framework from the perspective of a user

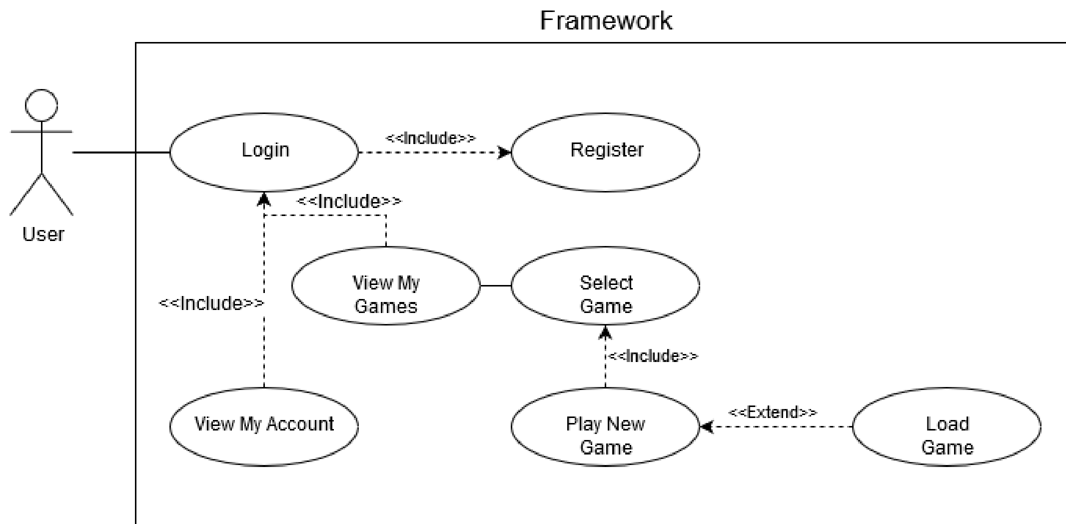

Figure 1.2: Interaction diagram for our framework from the perspective of a user

1.3 Quality Attributes

1.3.1 Performance

It is nearly impossible to produce a quality game system without giving careful consideration to performance. Games must be able to consistently respond to player input quickly enough to sustain the illusion of fluid motion; any discrepancies are quickly noticed (and complained about) by players. To this end, we have used several tactics to ensure that our system delivers acceptable performance.

Firstly, we provide an implementation of the Entity-Component-System architectural pattern. This allows client developers to process their game data in aggregated chunks which should result in much better cache utilisation compared to a traditional deep inheritance hierarchy.

We also offer Object Pools in our framework to help minimise Java heap allocations. This is demonstrated in the Space Invaders example game, where object pools are used to store component data for projectiles; without pools these projectiles could potentially create huge amounts of garbage objects and cause immersion-breaking garbage collection pauses.

1.3.2 Extensibility

Extensibility is of huge importance to our project — client developers must be able to use our framework to construct a range of different game applications.

We support framework-level extensibility primarily through the use of the Interceptor pattern, as this allows client developers to fluidly hook into framework events simply by annotating their interceptor methods with the `@Intercepts` annotation. The interceptor pattern allows us to develop framework subsystems completely independent from one another — for example our authentication code communicates with the rest of the system only by receiving `LoginAttemptEvent` instances and emitting further events for the rest of the framework to process.

We have also designed the Entity-Component-System to be as extensible as possible by making heavy use of Java generic classes and functional interfaces. As discussed in section 1, we allow client developers to specify

their own component data packages and relationships between them. This allows our ECS implementation to be adapted into a wide range of use cases without the need for changes to the framework code. For example, to add health-bars (a common sight in 2D games) to our Space Invaders example game, we would simply create the *Combiner* `drawHealthBar` which accepts a `SpriteComponent` and a `HealthComponent`. We can then draw a rectangle on the screen using the input health value to compute its width, and the input sprite component to compute its screen coordinates.

1.3.3 Flexibility

Flexibility is also a key attribute of our project. Our framework allows us to allow clients to be able to develop their own applications/games independently. Our entity component systems allows clients to create their own components that can have totally different properties and attributes to our space invaders components. The ECS is entirely generic and allows for clients to supply any classes to it.

Our framework is also independent of the Game package and any classes in that package. This means that a client can create their own game using our framework that can play totally differently to our Space invaders example game Our game view class is the only communication between the GameApplication that is being used and the Framework. This allows clients to create virtually any 2D game with our framework. The only requirement is it extends from the `GameApplication` class.

CHAPTER 2: DESIGN

2.1 ENTITY-COMPONENT-SYSTEM

2.1.1 Entities

In our implementation, entities exist only as opaque keys in a database, represented by the `EntityId` class. This allows the framework to change how entity ids are generated and implemented without affecting client code. In our implementation, we use a simple incrementing `AtomicInteger`, though this could easily be changed to a UUID or some other form of unique identifier.

2.1.2 Components

Components are small packages of related data containing little or no behaviour. In our example Space Invaders game, we use components to represent several different kinds of game data including velocity, sprites and health values. We use the tag interface `Component` to communicate the intent of a component class and to ensure that classes which are not explicitly marked as components are not accidentally added into the entity database by client developers.

In the name of brevity, we deliberately ignore some commonly accepted Java best practices in component classes, in particular the use of public mutable fields. It is likely that using accessor methods to hide these fields would simply bloat the codebase and possibly cause performance problems in a sensitive part of the system.

2.1.3 Systems

In traditional entity-component-system architectures, "entities" are constructed from discrete data chunks, the "components" and then processed in groups by the systems. This design ensures clean separation of concerns between game systems as each system is only operating on one kind of data. In our implementation, we augment this pattern with concepts from functional programming, primarily the idea of first-class functions and the `map` idiom. We use Java 8 Functional Interfaces and lambda expressions to emulate the first-class functions found in purely functional languages. We begin by stripping the API of "System" classes down to two function types, *Processors* and *Combiners*, bearing the following signatures in Java:

```
// Component consumer -> processes component data
// C is any class that implements Component
void process(C component);

// Component combiner -> combines two kinds of data
// A and B are any classes that implement Component
void combine(A inout, B in);
```

We define a *Consumer* to be a function which accepts a type of component data and performs some transformation on it. Consumers can be used to implement "end-of-the-line" functionality; for example a `RenderConsumer` class might accept sprite data components and display them to the screen.

```
private void drawSprite(SpriteComponent s) {
  graphics.drawImage(s.getImg(), s.x, s.y);
}
```

Listing 1: Example of a consumer function: rendering sprites in Space Invaders

A *Combiner* is a function which, like the consumer, accepts and transforms component data. Unlike the consumer however, combiners operate on two component types; an `inout` parameter and an `in` parameter. In a combiner, the `inout` parameter is permitted to be changed by the combiner function while the `in` parameter is not. Combiners can be used where the processing of one component is dependant upon some other linked component. In our example game, combiners are used to implement animation by adding the velocity of every velocity component to every sprite component. We do not need to introduce combiners with more than two

parameters as combiners with this functionality may be composed out of simpler two-parameter combiners.

```
private void drawHealthBar(SpriteComponent s, HealthComponent h) {
  graphics.drawRect(s.x - 10, s.y + 10, h.currentHealth, 5);
}
```

Listing 2: Example of a combiner: Drawing health-bars in Space Invaders

To actually execute these functions, clients call either `applyConsumer` or `applyCombiner`, much like executing queries on a database. For example, in our Space Invaders renderer, we use a consumer to render sprites to the screen with the following call:

```
entityDB.applyConsumer(this::renderSprite, SpriteComponent.class)
```

It should be noted that while this design takes ideas from functional programming it is not, strictly speaking, functional. This is because these consumers and combiners affect the program's state by modifying their parameters directly; in a purely functional implementation, we would compute new component instances from input data and have a class further up the call chain operate on these new objects.

2.2 INTERCEPTORS

The *Interceptor* architectural pattern forms the spine of our framework, organising and processing messages between the various independent subsystems. In our implementation, we make use of Java Annotations to define our Interceptors as instance methods which accept a context object as a parameter. This design allows arbitrary classes to register themselves as interceptors without the need for maintaining interceptor interfaces on the framework side

To allow client developers to mark methods as interceptors, we provide the `@Intercepts` source annotation which accepts one parameter: a particular context object this interceptor will respond to. Client developers must then register their classes with the framework. During registration, the framework will scan registered classes for methods marked with the Intercepts annotation. Marked methods are then placed into an index structure which maps context object types to responder objects. When the framework fires an event, the dispatcher will look up the particular event type in this index and then execute all of its responder methods.

2.2.1 Context Objects

Our context objects are simple POJO (Plain Old Java Object) classes with little to no behaviour; they are simple data packages passed between the framework and client applications. Context Objects are also required to be immutable — this ensures that malicious or erroneous interceptors may not corrupt the message before it is sent to other interceptors. To propagate state changes to the framework, client developers must emit their own events.

As our context objects are required to be immutable, we deliberately eschew some established Java best practices to enhance code brevity and readability. We do not, generally, write accessor methods for context object fields; most of our context object classes are simply collections of fields marked `public final`.

```java
public final class PopupMessageEvent {
  public final String msg;

  public PopupMessageEvent(String msg) {
    this.msg = msg;
  }
}
```

Listing 3: Entirety of one of our context objects: a message signaling to show the user a pop-up message.

A method of automatically and formally declaring objects immutable would be an excellent addition to this design, and to the Java programming language. While we investigated some methods of achieving this goal, such as writing custom Java compiler plugins, these were determined to be too time consuming for this project.

2.2.2 Concrete Interceptors

In our design, concrete interceptors are actually not class instances but rather instances of the Java reflection `Method` class — in essence, function pointers. Interceptor methods are marked as such by the `@Intercepts` Java annotation, with a parameter specifying which event types they are promising to handle. To begin receiving events, a class with interceptor methods defined must register itself with a `Dispatcher` class instance. Later, when an event is emitted, the dispatcher will look up which interceptor method instances to call depending on the type of event emitted.

In the example below, we mark the method `onKeyDown` as an interceptor for the `KeyInputEvent` event class and then register the containing class instance with a dispatcher. When the user hits a key on their keyboard, the framework will automatically invoke the interceptor method.

```java
// (in class constructor, given a Dispatcher instance)
dispatcher.register(this);
// ...

// Interceptor
@Intercepts(KeyInputEvent.class) // Specifies event type
public void onKeyDown(KeyInputEvent key) {
  // Client processing
}
```

Listing 4: Example code to create an interceptor for keyboard events

2.2.3 Dispatchers

In the design of our framework, we employ only one *Dispatcher* instance. Because we use Runtime Type Information (RTTI) to determine which interceptor to call at runtime, our dispatchers simply other classes throughout client code and the framework who emit events.

2.3 Game Saves

We utilise a combination of the *Adapter*, *Memento* and *Repository* patterns to support extensibile and reliable data storage and retrieval. Repositories allow us to abstract away the implementation details of data storage backends, allowing our system to support transparently swapping out an SQL storage backend for a JSON one, for example. In our implementation, we provide one concrete repository, the `JsonRepository`.

To allow players to save and load their games, we utilise the Memento design pattern. Since the intention is that most developers will make use of the EntityDatabase class to store their games' state, we allow client developers to take 'Snapshots' of the database's internal state. These Snapshot instances are entirely opaque to clients, who may only pass them back to a database instance when they wish to restore it to the Snapshot's state. This is accomplished by having `Snapshot` be an internal class of `EntityDatabase` with all fields, constructors and methods marked private. This ensures that Snapshots are not accidentally corrupted by client code.

Since our repository code was originally written for the CS4125 project, our repositories are constrained by the type system to work only with

classes that extend the `Model` abstract class. To showcase the adapter pattern, we treat the repository code as a third-party library and create repositories of snapshots via a generic `ModelAdapter`. By supplying what type to adapt to a Model as a generic parameter, we can create a single adapter class which allows any other class to be converted into a `Model` for use with repositories.

2.4 USER INPUT

2.4.1 Invoker

The role of an Invoker in the Command Design Pattern is the execute a command that is assigned to it. In our program it executes the command based off which key is pressed and acts essentially like a middle man for the Command design and Application. When a key is pressed the Invoker is called to see if it is linked to a action. If it is the Invoker then executes the command.

```
private void initGameControlScheme() {
  gameControls.put(KeyEvent.VK_A, new CommandMoveLeft(this.movableSprite));
  // ...
}
```

Listing 5: Passing in value to Invoker

2.4.2 Command Design Pattern

We use the Command Design Pattern for player movement and actions like saving game. As stated above the user presses a key which executes a command via Invoker. Each command is a class that implements the following interfaced name ActionCommand.

```
interface ActionCommand {
  void execute();
}
```

Listing 6: Command Interface

The interface consists of only one method which is called execute(). This is so the classes are not forced to implement redundant methods. This al-

lows for polymorphism which makes executing child classes much easier. There are numerous benefits to adopting this Design pattern for instance the ability to allow numerous keys to trigger the same command and it also promotes separation of concerns.

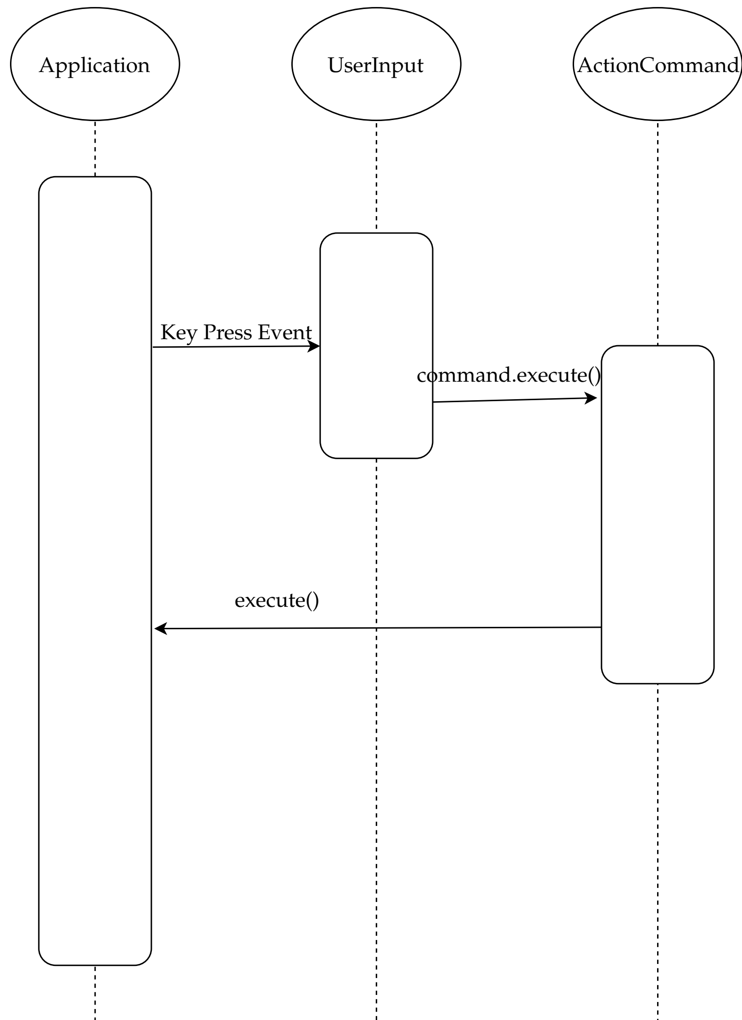

Figure 2.1: Communication between invoker and command

2.5 PROTOTYPE

The Prototype design pattern delegates the process of being cloned to the object being cloned. In our program we have an interface called Prototype which is inherited by components.Components are used to build player and non-player entities. The interface contains two methods reproduce() and equals(). reproduce() reproduces the component it is called on and returns the copy. equals() compares two components of the same instance to see if they are identical. We utilize the Prototype design when making projectiles.

```
public interface Prototype<C extends Component> {
  C reproduce();

  default boolean equals(C c) {
    return false;
  }
}
```

Listing 7: Prototype Interface

2.6 ABSTRACT FACTORY

An Abstract factory encapsulates a group of individual factories without specifying its concrete classes. In this project we used an abstract factory method to create different alien types with different attributes.

All alien types extend from the `AlienFactory` class. The factories have a concrete method `buildAlien` which creates an entity ID and the components that the Alien requires. In this project an Alien requires a health, velocity, wobble (oscillation) and Sprite component.

```java
abstract class AlienFactory {
  final ResourceCatalog gameResources;

  AlienFactory(final ResourceCatalog gameResources) {
    this.gameResources = gameResources;
  }

  abstract void buildAlien(EntityDatabase eDB, int aX, int aY);
}
```

Listing 8: API of our abstract alien factory

We use Abstract Factories to support the creation of aliens with more health ("tank" aliens) faster aliens along with our regular default enemies. Creation logic for different kinds of aliens is entirely encapsulated within the factories.

```java
private void buildAliens() {
  for (int alienX = 0; alienX < 5; ++alienX) {
    for (int alienY = 0; alienY < 6; ++alienY) {
      if (alienY == 0 || alienY == 5) {
        tankAlienFactory.buildAlien(this.entityDB, alienX, alienY);
      } else if (alienY == 1 || alienY == 4) {
        fastAlienFactory.buildAlien(this.entityDB, alienX, alienY);
      } else {
        defaultAlienFactory.buildAlien(this.entityDB, alienX, alienY);
      }
    }
  }

  this.alienCount = 30;
}
```

Listing 9: Constructing aliens with factories in Space Invaders

2.7 Builder

Over the course of the project, we were required to create several different GUI screens using the JavaFX Swing API's GridBagConstraints. Manually creating these screens, represented in our code as `JPanel` objects, is time-consuming and repetitive. To reduce code duplication and speed up

development, we abstract the process of creating `JPanel` screens into a Builder class, `JPanelFormBuiler`.

```
// Screen initialisation
++constraints.gridy
constraints.gridx = 0;
constraints.weightx = 0.5;
form.add(new JLabel("Username"), constraints);
constraints.gridx = 1;
constraints.weightx = 1;
constraints.gridx = 1;
constraints.weightx = 1;
form.add(new JTextField, constraints);
// Many more repeated component additions...
```

Listing 10: Laborious manual construction of GridBagConstraints GUI screens

```
JPanelFormBuidler builder = new JPanelFormBuilder();
builder.add(new JTextField(), "Username");
builder.add(new JPasswordField(), "Password");
builder.add(new JButton("Login"));
JPanel resultForm = builder.getForm();
```

Listing 11: The same form refactored to use the Builder pattern

CHAPTER 3: ARCHITECTURE

At the highest level, we employ a split architecture, utilising the Model-View-Controller architectural pattern for areas of the framework that are not performance-sensitive, such as the UI and user account systems; for the main game code which must be made to run as fast as possible, we employ a variation on the traditional Entity-Component-System.

The main project source tree is split into two packages, `framework` and `game`. The `framework` package contains all of the code necessary to write a simple 2D game, including an Entity-Component-System, texture library and a Java Swing renderer. The `game` package contains code for a simple *Space Invaders* game which showcases the various components made available in our framework. The `framework` package is entirely independent from the `game` package.

3.1 FRAMEWORK

Our framework source code is entirely contained inside the `framework` package. In principle, this package is completely decoupled from the `game` package. The framework controls most aspects of the application's execution, and is responsible for starting up the program. On startup, a list of classes implementing `GameApplication` is loaded from some source; in our proof-of-concept implementation we simply instantiate `SpaceInvadersGame` directly however it would not be difficult to refactor this design so that games were loaded from JAR files, for example.

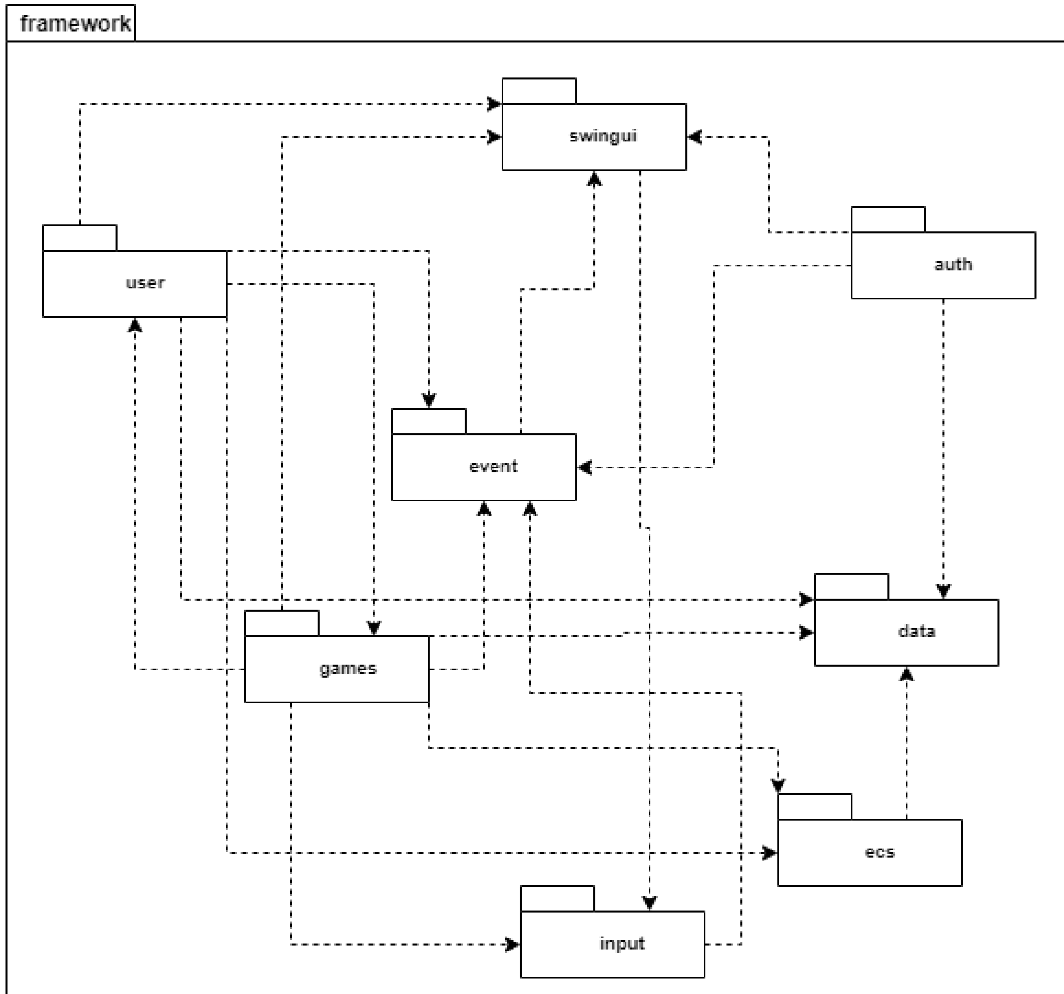

Figure 3.1: Framework package diagram

3.2 Game

The game top-level package contains all of the code for our Space Invaders example game.

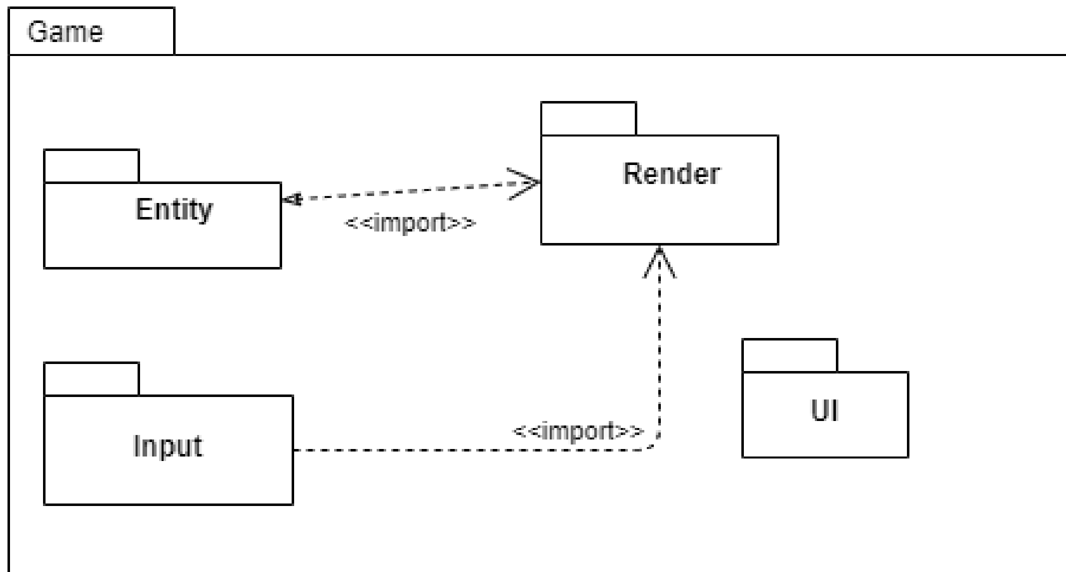

Figure 3.2: Game package diagram

CHAPTER 4: IMPLEMENTATION

This section details some interesting aspects of the project's implementation code, both in the framework package and in the example game.

4.1 INTERCEPTOR DISPATCHER

We use a central dispatcher for all of our interceptors. This is made possible by inserting all interceptor methods into an index-like structure where the class names of the context objects are the keys and the interceptor methods are the values. In our `Dispatcher` class, this index is represented by a `ConcurrentHashMap` from `Class<?>` keys to a sorted list of `Responder` instances (object-method tuples).

When registering a new responder class with the dispatcher, the class is examined for any methods marked with the `@Intercepts` annotation. If any methods with this annotation are found, they are placed in the dispatchers index with a key corresponding to the annotation's `value` field.

```
public void register(Object responderObject) {
  Class<?> c = responderObject.getClass();
  while (c != Object.class) {
    for (Method m : c.getDeclaredMethods()) {
      if (m.isAnnotationPresent(Intercepts.class)) {
        Class<?> eventType = m.getAnnotation(Intercepts.class).value();
        m.setAccessible(true);

        Priority.Level priority = m.isAnnotationPresent(Priority.class) ?
                                  m.getAnnotation(Priority.class).value() :
                                  Priority.Level.NORMAL;
        Responder responder = new Responder(m, responderObject, priority);
        List<Responder> responders;

        if (! this.responderIndex.containsKey(eventType)) {
          responders = new ArrayList<>();
          this.responderIndex.put(eventType, responders);
        } else {
          responders = this.responderIndex.get(eventType);
        }

        responders.add(responder);
        Collections.sort(responders);
      }
    }

    c = c.getSuperclass();
  }
}
```

Listing 12: Dispatcher's register method

This data organisation optimises for event dispatching — while registrations are more expensive due to the overhead of sorting, dispatching is made very efficient because we can query for all the interceptors that respond to a particular event type in constant time.

4.2 CS4125 Code

We have ported the authentication, GUI, dispatcher and repository patterns from our previous CS4125 project [4]. We have applied several refactorings and changes to these software modules to fit them into our game

framework project:

1. Our dispatcher has been refactored to support priority-sorted lists of interceptors
2. Our `JsonRepository` implementation has been extended to support custom deserialisers
3. Our GUI code has been refactored to support building forms with our `JPanelFormBuilder`

4.3 Space Invaders Game

Along with the framework, we have written a small Space Invaders game to showcase the capabilities of our framework. In our Space Invaders game, we make heavy use of the framework's ECS API. We define several different kinds of entity in our game, including the player, aliens and projectiles. These entities are defined by their data, the components they hold. We attach `HealthComponents` to aliens and the player, `VelocityComponents` to aliens and projectiles and `DamageComponents` to projectiles. These components are then processed by systems, such as the render system.

4.3.1 Score System

Aliens contain a `HealthComponent` and a `ScoreComponent`, among other components. A `HealthComponent` contains information about an entitys current and maximum health and a `ScoreComponent` contains an integer value for the amount of points the entity is worth.

Within the `CollisionSystem` a collision involves subtracting the damageValue of a colliding entity with a `DamageComponent` from the currentHealth of a colliding entity with a `HealthComponent`. Once a collision results in the death of an entity, ie its currentHealth in its `HealthComponent` is equal to or less than zero, a method is triggered to dispatch a new `ScoreEvent` using a `ComponentCombiner` to associate the specific entitys health and score value. The health and score are accessed from the given components mapped to its EntityId taken from the component table.

The score event holds the score of the entity and is intercepted by the updateScore method in the `RenderSystem`. This method simply adds the en-

titys score to the scoreboards currentScore. The scoreboard is updated and drawn on screen in a separate method within the RenderSystem.

4.4 UI Framework

The UI is made using the Java Swing library and follows Model-View-Controller architecture. We have a MainWindow that contains elements that will always be visible such as the back button and the title of the application. Within the window we have a JFrame that displays the current view, which will always initially be LoginView. Views do not define behaviour, they only contain the contents and layout of the JFrame as well as any listeners that are added to buttons. As shown below in the LoginView:

```
public LoginView(Dispatcher d) {
  super(d, "Login");

  usernameLabel = new JLabel("Username");
  passwordLabel = new JLabel("Password");
  usernameInput = new JTextField();
  passwordInput = new JPasswordField();
  loginButton = new JButton("Login");
  registerButton = new JButton("Register");

  usernameLabel.setBounds(80, 70, 200, 30);
  passwordLabel.setBounds(80, 110, 200, 30);
  usernameInput.setBounds(300, 70, 200, 30);
  passwordInput.setBounds(300, 110, 200, 30);
  loginButton.setBounds(150, 160, 100, 30);
  registerButton.setBounds(250, 160, 100, 30);


  this.addComponent(usernameLabel);
  this.addComponent(passwordLabel);
  this.addComponent(usernameInput);
  this.addComponent(passwordInput);
  this.addComponent(loginButton);
  this.addComponent(registerButton);

  loginButton.addActionListener(e -> getDispatcher()
    .dispatch(new LoginAttemptEvent(getUsername(), getPassword())));

  this.registerButton.addActionListener(e -> getDispatcher()
    .dispatch(new Redirect(new RegisterView(getDispatcher()))));
  }
}
```

The registerButton will dispatch a Redirect to the RegisterView. A Redirect simply changes the view the MainWindow JFrame is displaying to the supplied view.

Also shown is that the loginButton will dispatch a LoginAttemptEvent along with the login details provided by the user.

```java
public final class LoginAttemptEvent {

  public final String username;
  public final String password;

  public LoginAttemptEvent(String username, String password) {
    this.username = username;
    this.password = password;
  }
}
```

This is a simple event that holds the submitted username and password, while the attemptLogin method inside of AuthController intercepts any LoginAttemptEvent. It then checks if such a user exists and if the credentials are correct. If so, it sets the currently logged in user to that user and dispatches a Redirect to the DashboardView. Otherwise, it creates an error message in a popup window using the dispatchErrorMessage method from the Controller superclass which dispatches a PopupMessageEvent which is intercepted within MainWindow to display the given error within a generic popup error window.

CHAPTER 5: ADDED VALUE

In this section we detail some noteworthy features of our project which enhance our chosen quality attributes. A common theme amongst all of these enhancements is the use of the Java static type system to enforce domain contracts at compile time, and the use of first-class functions to enhance code conciseness and readability.

5.1 REPOSITORIES

We did not select the repository design pattern as our chosen research pattern, nor is it covered in CS4227. Nevertheless, the repository pattern is so useful and powerful that we felt the need to include it.

In our project, *Repositories* are abstractions over a data source which may be queried and mutated, similar to modern relational databases. We expose the following API for repositories:

```
public interface Repository<M extends Model> {
  Optional<M> findFirst(Predicate<M> query);
  Stream<M> findAll(Predicate<M> query);

  void insert(M m) throws IOException;
  void delete(M m) throws IOException;
  void update(Consumer<M> transformer) throws IOException;

  void save() throws IOException;

  List<M> all();
}
```

The underlying data source is completely abstracted away; a client may be working with an SQL database, a JSON file or even a network byte stream. This is extremely useful for supporting both extensible and flexibility: we may extend our program simply by writing new concrete repositories against the `Repository` interface, such as `JsonRepository`. We achieve great flexibility by using Java generics to support many different kinds of data objects without requiring clients to explicitly write new repository classes for each kind of object they want to store, for example the same `JsonRepository` class may be used to store `UserAccount` classes or database snapshots.

We also use bounded generics to support more reliable code — our repositories only accept data which extends from the `Model` abstract class. This ensures that data objects in repositories will always have a unique ID, a guarantee that may be exploited to write more efficient searching and storage code. As an added bonus, this approach means that the Java compiler will automatically enforce our domain constraints without the need for us to write runtime validation code.

Finally, we make use of several modern Java language features in our Repository interface, such as `Predicate` and `Optional`. Predicates in particular allow us to use a powerful standard language feature to express the concept of data queries without the need to write any custom code. Clients may simply query for data with lambda expressions or function references, such as this example from our login handler:

```
this.userRepository.findFirst(u -> u.validateCredentials(username, password)
```

5.2 Reflection-based Interceptors

In a traditional interceptor design, each type of interceptor requires its own interface, with clients needing to write new classes to implement these interfaces. In our project, we achieve a greater degree of flexibility and extensibility by utilising reflection and runtime type information to build a dynamic index of interceptors.

The Java language allows programmers to obtain handles to methods dynamically at runtime via the `Method` class, these may then be invoked with dynamically typed parameters. Combining this feature with Java *anno-*

tations, markers in a program's source code which carry metadata about the token they are attached to, allows clients to clearly and concisely mark interceptor methods in their code by adding the `Intercepts` annotation to a method, such as this example from our framework GUI code, which intercepts and handles screen redirect requests:

```
@Intercepts(Redirect.class)
public void setContent(Redirect next) {
  viewHistory.push(this.currentView);
  setView(next.view);
  currentView.getContent().requestFocus();
  currentView.getContent().addKeyListener(new KeyboardInputHandler(dispatcher));
}
```

On the framework side, we create a `Map` of event types to ordered lists of interceptor object →interceptor method tuples. An ordered list is necessary to support interceptor priorities, which clients indicate via the optional `@Priority` annotation. When the framework wishes to broadcast an event to registered interceptors, we can efficiently look up the list of interested interceptors using the event type as a Map key. Once the list of interceptors is obtained, they are executed in priority order, with `NORMAL` being the default priority.

While this approach to the interceptor pattern results in more concise code and less initial effort from both clients and framework developers, it is not perfect. Because of the dynamic nature of reflection method calls, valuable compile-time type checks are lost; clients must take care not to mismatch their interceptor method signatures (for example, by forgetting to add the context object as a parameter) or the system will throw a reflection exception.

CHAPTER 6: TESTING

6.1 UNIT TESTING WITH JUNIT4

Unit tests are important as they provide automated testing for a project. Automated tests are invaluable in system testing as they save a large amount of development time. An automated test can also be ran numerous times and on a schedule. They are also useful as they can see if new code can cause the program to fail unexpectedly

6.1.1 Entity Creation Test

This unit test asserts that the entity component system is able to add and remove components form the database. The test creates a health component and an Entity database. It then adds the health component to the database and asserts it exists. It then removes it and asserts that it was removed

```java
public class EntityCreationTest {
  private final EntityDatabase eDB;
  private final EntityId testEntity;
  private final HealthComponent testHealth;

  public EntityCreationTest() {
    eDB = new EntityDatabase();
    testEntity = new EntityId();
    testHealth = new HealthComponent(100);
  }

  @Test
  public void entityCreator() {
    eDB.addComponentToEntity(testEntity, testHealth);
    assertTrue(eDB.getEntitiesForComponent(HealthComponent.class)
              .containsKey(testEntity));

    eDB.removeComponentFromEntity(testEntity, HealthComponent.class);
    assertFalse(eDB.getEntitiesForComponent(HealthComponent.class)
               .containsKey(testEntity));
  }
}
```

Listing 13: Entity creation test source

6.1.2 Interceptor Test

This test asserts the Dispatcher can correctly responds to a dispatch request. The test creates a testEvent and sets 3 different responders. One high,one medium and one low priority request. The request is then sent to the dispatcher. The test then asserts that the dispatch calls the the events in order of priority (High,Medium,Low).

```java
public class DispatcherTest {
  private static class TestEvent {}

  private boolean responded = false;
  private Dispatcher d = new Dispatcher();

  private final ArrayList<Level> calledOrder = new ArrayList<>();

  public DispatcherTest() {
    this.d.register(this);
  }

  @Intercepts(TestEvent.class)
  public void normalResponder(TestEvent tE) {
    this.calledOrder.add(Level.NORMAL);
    responded = true;
  }

  @Test
  public void testDispatcherMethodCalled() {
    this.d.dispatch(new TestEvent());
    assertTrue(responded);
  }

  @Intercepts(TestEvent.class)
  @Priority(Level.HIGH)
  public void highResponder(final TestEvent e) {
    this.calledOrder.add(Level.HIGH);
  }

  @Intercepts(TestEvent.class)
  @Priority(Level.LOW)
  public void lowResponder(final TestEvent e) {
    this.calledOrder.add(Level.LOW);
  }

  @Test
  public void testDispatcherMethodCalledInOrder() {
    this.d.dispatch(new TestEvent());

    final List<Priority.Level> expected = Arrays.asList(
      Level.HIGH,
      Level.NORMAL,
      Level.LOW
    );

    assertEquals(expected, this.calledOrder);
  }
}
```


Listing 14: Dispatcher Test Source

6.1.3 Component Consumer Test

This test ensures that an entity consumer correctly maps over a particular component type. The before list creates an entity ID and a health component. The entity is then added to the beforelist. Then the process method is called and the health component is mapped to the afterlist . The test then asserts that the before list and after list are equal.

```
public class ComponentConsumerTest {
  private final EntityDatabase eDB;
  private final EntityId eID;
  private final HealthComponent health;
  private final ComponentConsumer<HealthComponent> eP;
  private List<EntityId> beforeList, afterList;

  public ComponentConsumerTest() {
    this.eDB = new EntityDatabase();
    this.eID = new EntityId();
    this.health = new HealthComponent(100);
    this.eP = this::process;
    this.beforeList = new ArrayList<>();
    this.afterList = new ArrayList<>();
  }

  private void createBeforeList() {
    for (int i = 0; i < 5; i++) {
        EntityId eID = new EntityId();
        this.eDB.addComponentToEntity(eID, health);
        beforeList.add(eID);
    }
  }

  private void process(HealthComponent health, EntityId eID) {
    afterList.add(eID);
  }

  @Test
  public void entityProcessTest() {
    createBeforeList();
    eDB.applyConsumer(eP, HealthComponent.class);
    assertTrue(afterList.containsAll(beforeList););
  }
}
```

6.1.4 Command Test

This test is done to test that the command pattern is implemented correctly. The test requires we create a command test class much like in the main project which implements the ActionCommand Interface. The execute of this Command is to change the boolean value of commandPressed from true to false or false to true. The test also requires an interceptor and a dispatcher to process the CommandEvent. With all this the main test simply dispatches the test event and asserts that when the execute is called the commandPressed value is changed from false to true.

```
public class CommandPatternTest {
  private static class CommandEvent { }

  private Dispatcher d = new Dispatcher();
  private boolean commandPressed = false;

  public CommandPatternTest() {
    this.d.register(this);
  }

  @Intercepts(CommandEvent.class)
  public void commandResponder(CommandEvent cE) {
    CommandTest test = new CommandTest();
    test.execute();
  }

  @Test
  public void commandPatternTest() {
    this.d.dispatch(new CommandEvent());
    assertTrue(commandPressed);
  }

  class CommandTest implements ActionCommand {
    @Override
    public void execute() {
      commandPressed = !commandPressed;
    }
  }
}
```

Listing 15: Command test source code

CHAPTER 7: EVALUATION

7.1 KNOWN ISSUES

7.1.1 Java Generics Type Erasure

Regrettably, there is no clean way in the Java programming language to determine the type of a generic type variable at runtime. To work around this issue, we require clients to specify the generic type variable both at build-time, as a generic type, and at run-time, as a parameter of type `Class`. An example of this issue can be found in our `JsonRepository` constructor. The third-party JSON library must know the type of object to serialise at runtime, but we are serialising a generic ArrayList of 'M'; the type of M is erased by the Java language at runtime. Clients must pass a `Class<M>` so that we may determine what concrete object to serialise.

```
// JsonRepository<M> -- Class<M> is the type of M.
public JsonRepository(Class<M> modelType, String fileName) {
  // File loading code elided
  Type modelListType = TypeToken
                     .getParameterized(ArrayList.class, modelType)
                     .getType();

  // Load JSON
  ArrayList<M> data = this.gson.fromJson(json, modelListType);
}

// In client code
Repository<UserAccount> users = new JsonRepository(UserAccount.class, "users.json");
```

Listing 16: Java generic type erasure means that clients must pass the concrete class of a generic type variable when constructing repositories

7.1.2 Save Game Loading

In our ECS implementation, components are stored in tables keyed by their class name. To allow users to save and load their games, we allow the ECS state database to be restored from a 'snapshot' memento object, these snapshots are stored in JSON files. Because the actual component objects stored in the database are abstract `Component` references, there is no way for the deserialisation code to generically determine what component to instantiate.

```
Map<Class<? extends Component>, Map<EntityId, Component>> index;
```

Listing 17: Internal format of the ECS Database — a map from component classes to component tables

We solve this problem by writing a custom JSON deserialiser to deserialise `ModelAdapter<EntityDatabase>`. As shown in the listing below, this custom deserialiser determines which concrete component class to instantiate based on the key class name.

```java
public ModelAdapter<Snapshot> deserialize(JsonElement jsonElement, Type type,
                             JsonDeserializationContext context)
   EntityDatabase d = new EntityDatabase();
   JsonElement table = jsonElement.getAsJsonObject()
                       .getAsJsonObject("wrapped")
                       .getAsJsonObject("savedState")
                       .getAsJsonObject("componentsToEntities");

  Map<String, Map<EntityId, Component>> index = new HashMap<>();

  table.getAsJsonObject().entrySet().forEach(entry -> {
    Map<EntityId, Component> componentTable;
    Class<?> key = Class.forName(entry.getKey());
    assert(componentClass.isAssignableFrom(Component.class));

    Type mapType = TypeToken.getParameterized(HashMap.class, EntityId.class, key)
                            .getType();
    componentTable = context.deserialize(entry.getValue(), mapType);
    index.put(entry.getKey(), componentTable);
  });

  d.componentsToEntities = index;
  return ModelAdapter.of(d.saveState());
}
```

Listing 18: Custom GSON deserialiser used for our save game loading

While this approach works, it causes a host of architectural problems. This deserialiser is currently placed inside the `EntityDatabase` class, coupling the ECS code to the IO/persistence layer (worse, to a concrete persistence layer!). It also has several security implications, as our only security check is an `assert` that the input class implements the `Component` interface. To properly implement this feature, a robust way of storing component classes should be written, perhaps as a custom serialiser.

7.2 Suitability

In general, we believe that this project fulfils its functional and non-functional requirements well. We have tuned our implementation details carefully using data gathered from the JVisualVM profiler to achieve the best possible performance. Our use of generic code and Java functional interfaces

allows both framework maintainers and client developers to easily extend our implementation to new use cases, while being flexible enough to adapt to a wide range of potential domains.

7.3 Potential Improvements

7.3.1 LambdaMetafactory and Reflection Calls

In Java, reflection method calls are very slow[3]. An alternative to reflection method calls is the newly-introduced `LambdaMetafactory`, a class primarily intended for compiler authors which can dynamically generate lambda expressions. Migrating our dispatcher to use the invokable `CallSite` instances generated by the `LambdaMetafactory` would significantly speed up our dispatcher.

7.3.2 EnumMap

`EnumMap` is a special case of the more general Java `Map` classes where all of the keys are entries in an `enum`. `EnumMaps` can be faster than regular maps because they are implemented internally as simple arrays. Refactoring our ECS index to us an `EnumMap` instead of a regular Java map keyed by component class name could significantly speed up our game processing code. This would, however, require client developers to create and maintain an enumeration of component types.

Chapter 8: Contribution

8.1 Lines of Code Per Class

```
  64 ./src/invaders/framework/auth/AuthController.java
  12 ./src/invaders/framework/auth/LoginAttemptEvent.java
  54 ./src/invaders/framework/auth/LoginView.java
  15 ./src/invaders/framework/auth/RegisterEvent.java
  61 ./src/invaders/framework/auth/RegisterView.java
  27 ./src/invaders/framework/Controller.java
 119 ./src/invaders/framework/data/JsonRepository.java
  15 ./src/invaders/framework/data/Model.java
  33 ./src/invaders/framework/data/ModelAdapter.java
  36 ./src/invaders/framework/data/Repository.java
   6 ./src/invaders/framework/ecs/Component.java
   6 ./src/invaders/framework/ecs/ComponentCombiner.java
  13 ./src/invaders/framework/ecs/ComponentConsumer.java
 196 ./src/invaders/framework/ecs/EntityDatabase.java
  31 ./src/invaders/framework/ecs/EntityId.java
  10 ./src/invaders/framework/ecs/Prototype.java
  71 ./src/invaders/framework/event/Dispatcher.java
  14 ./src/invaders/framework/event/Intercepts.java
  19 ./src/invaders/framework/event/Priority.java
  37 ./src/invaders/framework/event/Responder.java
  85 ./src/invaders/framework/Framework.java
  13 ./src/invaders/framework/games/FrameTickEvent.java
  45 ./src/invaders/framework/games/GameApplication.java
  59 ./src/invaders/framework/games/GameCanvas.java
   9 ./src/invaders/framework/games/GameStartEvent.java
  21 ./src/invaders/framework/games/GameView.java
  30 ./src/invaders/framework/games/ResourceCatalog.java
  31 ./src/invaders/framework/input/KeyboardInputHandler.java
  10 ./src/invaders/framework/input/KeyInputEvent.java
  41 ./src/invaders/framework/swingui/JPanelFormBuilder.java
```

```
101 ./src/invaders/framework/swingui/MainWindow.java
 14 ./src/invaders/framework/swingui/PopupMessageEvent.java
 15 ./src/invaders/framework/swingui/Redirect.java
 40 ./src/invaders/framework/swingui/SwingView.java
 32 ./src/invaders/framework/user/AccountView.java
 39 ./src/invaders/framework/user/DashboardView.java
 51 ./src/invaders/framework/user/GamesListView.java
 73 ./src/invaders/framework/user/SaveGameView.java
 93 ./src/invaders/framework/user/UserAccount.java
 21 ./src/invaders/framework/user/UserController.java
 22 ./src/invaders/game/entity/AlienFactory.java
 48 ./src/invaders/game/entity/CollisionComponent.java
 78 ./src/invaders/game/entity/CollisionSystem.java
 33 ./src/invaders/game/entity/DamageComponent.java
 46 ./src/invaders/game/entity/DefaultAlienFactory.java
 44 ./src/invaders/game/entity/FastAlienFactory.java
 36 ./src/invaders/game/entity/HealthComponent.java
 32 ./src/invaders/game/entity/HealthSystem.java
  7 ./src/invaders/game/entity/PlayerComponent.java
 11 ./src/invaders/game/entity/ScoreComponent.java
 24 ./src/invaders/game/entity/ScoreSystem.java
 42 ./src/invaders/game/entity/TankAlienFactory.java
 11 ./src/invaders/game/entity/TeamComponent.java
 25 ./src/invaders/game/GameResources.java
 11 ./src/invaders/game/input/ActionCommand.java
 20 ./src/invaders/game/input/CommandFire.java
 18 ./src/invaders/game/input/CommandMoveLeft.java
 17 ./src/invaders/game/input/CommandMoveRight.java
 23 ./src/invaders/game/input/CommandQuit.java
 22 ./src/invaders/game/input/CommandSaveGame.java
 40 ./src/invaders/game/input/InputSystem.java
 34 ./src/invaders/game/ProjectileFactory.java
 38 ./src/invaders/game/render/AnimationSystem.java
 51 ./src/invaders/game/render/RenderSystem.java
 63 ./src/invaders/game/render/SpriteComponent.java
 36 ./src/invaders/game/render/VelocityComponent.java
 16 ./src/invaders/game/render/WobbleComponent.java
  9 ./src/invaders/game/ScoreEvent.java
183 ./src/invaders/game/SpaceInvadersGame.java
 29 ./src/invaders/game/ui/GameOverEvent.java
 14 ./src/invaders/game/ui/GameQuitEvent.java
 51 ./src/invaders/game/ui/MenuScreens.java
 46 ./test/CommandPatternTest.java
 65 ./test/ComponentConsumerTest.java
 68 ./test/DispatcherTest.java
```

```
  38 ./test/EntityCreationTest.java
3013 total
```

Lines were counted using the shell command `find . -name '*.java' | xargs wc -l` [1]

8.2 Lines of Code Per Contributor

We determined the lines of code inserted and deleted per-author using the following shell command [2]:

```
git log -author="_Your_Name_Here_"
        -pretty=tformat:
          -numstat | awk
            '{ add += $1; subs += $2; loc += $1 - $2 }
            END {
              printf "added lines: %s,
                      removed lines: %s,
                      total lines: %s\n", add, subs, loc }'
```

Results:

	Added	Deleted	Total
Liam	9391	5992	3399
Eric	784	134	650
Daniel	1077	72	1005
Kyran	616	130	486

8.3 Report

	Report Contribution
Liam	25%
Eric	25%
Daniel	25%
Kyran	25%

BIBLIOGRAPHY

[1] Peter Elespuru. *How to count all the lines of code in a directory recursively?* URL: https://stackoverflow.com/a/1358573/3121161.

[2] jkschneider. *Git - Is there a way to view the number of lines committed by Author?* URL: https://stackoverflow.com/questions/2731283/git-is-there-a-way-to-view-the-number-of-lines-committed-by-author/23832962#23832962.

[3] Oracle. *Trail: The Reflection API*. URL: https://docs.oracle.com/javase/tutorial/reflect/index.html.

[4] L. Power et al. *UL E-Store*. URL: https://gitlab.com/liamp/invaders.