

CA314 – Object Oriented Analysis & Design - 11

File name: CA314_Section_11_DES_Ver01

Author: L Tuohey

No. of pages: 37

Table of Contents

Explanatory Note to SDD for <i>Encounter</i> Video Game	4
0. Preliminary notes on context, etc.....	4
I. Role-Playing Game Architecture Framework	5
1. Introduction.....	5
1.1 Purpose.....	5
1.2 Scope.....	5
1.3 Definitions, acronyms and abbreviations.....	5
2. References.....	5
3. Decomposition description	5
3.1 Module decomposition.....	5
3.1.1 <i>RolePlayingGame</i> package	6
3.1.2 <i>Characters</i> package	7
3.1.3 <i>GameEnvironment</i> package	7
3.1.4 <i>Artifacts</i> package [Not implemented -- for future releases].....	7
3.2 Concurrent process decomposition	7
4.0 Dependency description.....	7
5.0 Interface description.....	8
II. Architecture of <i>Encounter</i> Role-Playing Game	8
1. Introduction.....	8
1.1 Purpose.....	8
1.2 Scope.....	8
1.3 Definitions, acronyms and abbreviations.....	8
2. References.....	8
3. Decomposition description	9
3.1 Module decomposition (object model)	9
3.1.1 <i>EncounterGame</i> package	10
3.1.2 <i>EncounterCharacters</i> package.....	11
3.1.3 <i>EncounterEnvironment</i> package	11
3.2 Concurrent process decomposition	11
3.3 Data decomposition	11
3.4 State model decomposition	11
3.5 Use case model decomposition.....	12
4.0 Dependency description.....	13
4.1 Inter-module dependencies (object model).....	13
4.2 Inter-process dependencies	14
4.3 Data dependencies	14
4.4 State dependencies	15
4.5 Layer dependencies.....	15
5. Interface Description.....	16
5.1 Module interfaces.....	16
5.1.1 Interface to the <i>EncounterGame</i> package	16
5.1.2 Interface to the <i>EncounterCharacters</i> package.....	16
5.1.3 Interface to <i>EncounterEnvironment</i> package.....	17
5.2 Process interface	17
5.2.1 Player character movement process.....	17
5.2.2 Foreign character movement process.....	17
Preliminary notes on Detailed Design (sections 6 for I and II))	18
I. Detailed design of Role-Playing Game Framework <i>continued</i>	18
6. Detailed design of Role-Playing Game Framework	18

6.1 Module detailed design	18
6.1.1 Role-playing game package	18
6.2.1 The Characters package	20
6.2.1.1 GameCharacter class.....	20
6.2.1 The GameEnvironment package.....	21
6.2.1 The Artifacts package	21
II. Detailed design of <i>Encounter</i> -- <i>continued</i>	22
6.Detailed design for <i>Encounter</i>	22
6.1 Module detailed design for <i>Encounter</i>	22
6.1.1 The <i>EncounterGame</i> package.....	22
6.1.1.1 The <i>EncounterGameDisplays</i> sub-package of the <i>EncounterGame</i> package	24
6.1.1.2 Sequence diagrams for event handling	25
6.1.1.2.1 Player dismisses report window event.....	25
6.1.1.2.2 Player dismisses report window event.....	25
6.1.1.2.3 Player moves to adjacent area event.....	26
6.1.1.2.4 Sequence diagrams for remaining events.....	27
6.1.1.CM The <i>CharacterMovement</i> class	27
6.1.1.EG The <i>EncounterGame</i> class	28
6.1.1.EN The <i>Engagement</i> class	28
6.1.1.ZZ The <i>Engaging, Waiting, Preparing, and Reporting</i> classes.....	29
6.1.2 The <i>EncounterCharacters</i> package.....	29
6.1.2.EC The <i>EncounterCharacter</i> class	30
6.1.2.ES The <i>EncounterCast</i> class.....	34
6.1.2.FC The <i>ForeignCharacter</i> class	34
6.1.2.PC The <i>PlayerCharacter</i> class.....	34
6.1.3 The <i>EncounterEnvironment</i> package.....	34
6.1.3.AR Area class	35
6.1.3.CO EncounterAreaConnection class.....	36
6.1.3.EE EncounterEnvironment class.....	36
6.1.3.CH ConnectionHyperlink class.....	37
6.1.3.TH ThumbnailMap class.....	37
6.2 Data detailed design.....	37

Explanatory Note to SDD for *Encounter* Video Game

This case study is taken from **Reference 3** “Software Design, from Programming to Architecture”, Braude (Wiley).

The case study is presented in two documents:

Software Requirements Specification (**SRS**)

Software Design Document (**SDD**) [*present document*]

Both the SRS and SDD may be thought of as being made up of two parts. In the case of the SDD we have

Part 1 (sections 1 to 5): **Architectural Aspects**

Part 2 (section 6): **Detailed Designs**

There is a further complication in that within both Parts 1 and 2, two designs are being presented, namely

I. Role-Playing Game Framework

II. Encounter Role-Playing Game

Within I and II, sections are numbered 1, 2 etc. To be absolutely unambiguous these should be “I.1, I.2, etc” or “II.1, II.2, etc”; however, it will be clear in practice which is being referred to.

Note: The section numbering of Reference 1 is retained, that is, a prefix “11” has not been prepended to the section numbers.

0. Preliminary notes on context, etc

We have two designs to describe. The first is that of the Role-Playing Video game framework; the second of the *Encounter* role-playing game.

The SDD for both designs are split into two parts. The first parts, SDD sections one through five, shown below, consist of the architectural aspects of the design. The second part, SDD section six, appearing at the end of chapter six, consist of the detailed designs.

The dependence of *Encounter* on the framework is specified in the *Encounter* case study.

I. Role-Playing Game Architecture Framework

1. Introduction

1.1 Purpose

This document describes the packages and classes of a framework for role-playing video games.

1.2 Scope

This framework covers essentials of role-playing game classes. Its main intention is to provide an example of a framework for educational purposes. It is not intended as a framework for commercial games since its size is kept small to facilitate learning.

1.3 Definitions, acronyms and abbreviations

Framework: a collection of interrelated classes used, via inheritance or aggregation, to produce families of applications

RPG: Role-playing game - a video game in which characters interact in a manner which depends on their characteristics and their environment

2. References

Braude, E. J., Software Engineering: an Object-oriented perspective

UML: The Unified Modeling Language User Guide by G. Booch, J. Rumbaugh and I. Jacobson, Addison-Wesley Pub Co; ISBN: 0-201-57168-4

IEEE standard 1016-1987 (reaffirmed 1993) guidelines for generating a Software Design Document

3. Decomposition description

[*Note to the student:* specifies how the framework classes for role-playing video games are to be grouped. This reflects the top-level decomposition: the detailed decomposition into methods, for example, is left for the detailed design.]

3.1 Module decomposition

[*Note to the student:* this section shows the decomposition, then explains each part in a subsection.]

The framework consists of the *RolePlayingGame*, *Characters*, *Artifacts*, and *Layout* packages. These are decomposed into the classes shown in [figure 5.48](#).

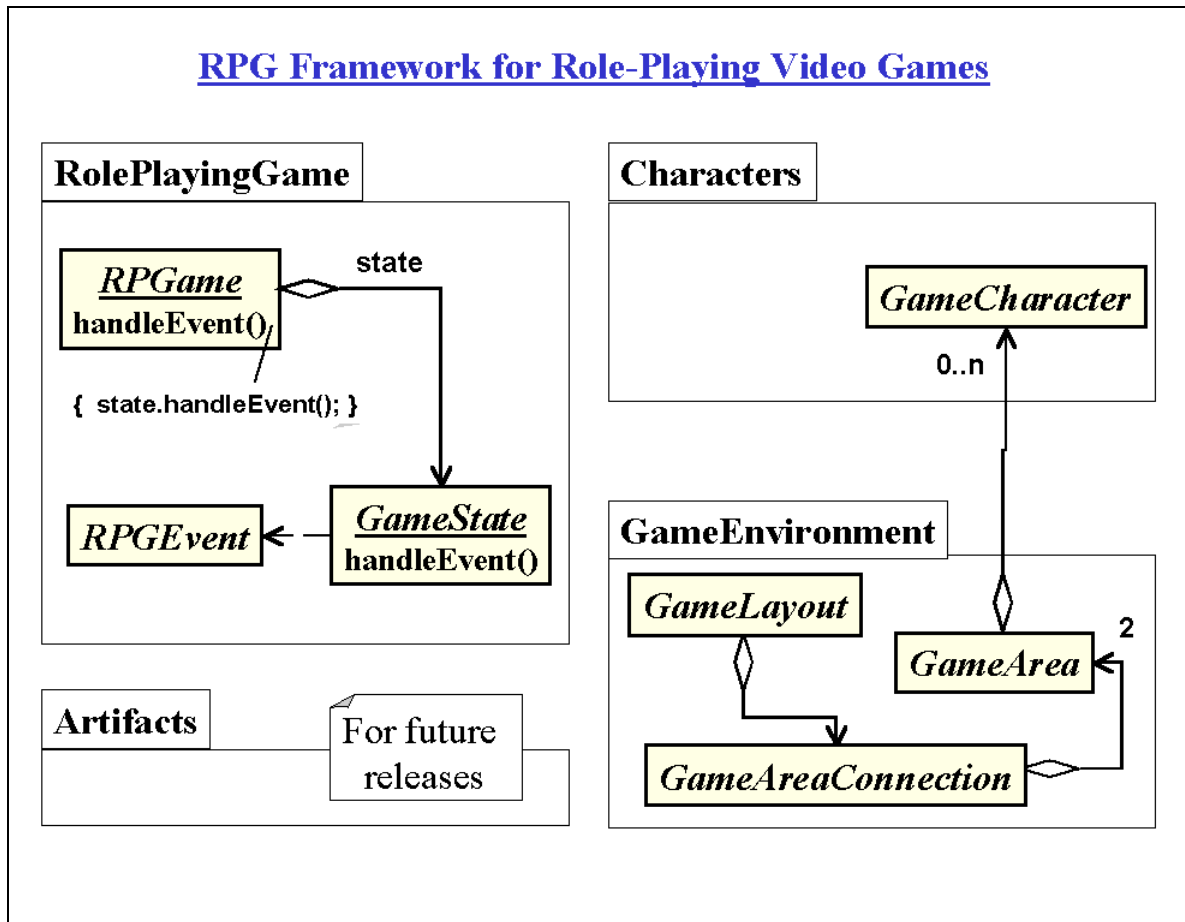


Figure 5.48 RPG Framework for Role-Playing Video Games

The classes in these packages are explained below. Unless otherwise stated, all classes in these packages are public. As indicated by the (UML) italics notation, all of the framework classes are abstract.

3.1.1 *RolePlayingGame* package

This package is designed as a state-transition machine. The concept is that a role-playing game is always in one of several states. This package makes it possible to describe the possible states of the game, and the actions that can take place in response to events. It implements the *State* design pattern. The state of the game is encapsulated (represented) by the particular *GameState* object aggregated by the (single) *RPGGame* object. This aggregated object is named *state*. In other words, *state* is an attribute of *RPGGame* of type *GameState*.

The function *handleEvent()* of *RPGGame* is called to handle each event occurring on the monitor (mouse clicks etc.). It executes by calling the *handleEvent()* function of *state*. The applicable version of *handleEvent()* depends on the particular subclass of *GameState* that *state* belongs to.

3.1.2 Characters package

[*Note to the student:* It may seem strange to have a package containing just one class, but everything in software design has a tendency to grow. Even if the class does not grow, this does not disqualify its usefulness. For another example of a package with just one class, see *java.applet*, whose only class is *Applet* (but which also contains a few interfaces).]

This package contains the *GameCharacter* class, which describes the characters of the game.

3.1.3 GameEnvironment package

This package describes the physical environment of the game. Each connection object aggregates the pair of *GameArea* objects which it connects. This architecture allows for multiple connections between two areas.

Each *GameArea* object aggregates the game characters that it contains (if any), and can detect encounters among characters.

3.1.4 Artifacts package [Not implemented -- for future releases]

This package is intended to store elements that are to be located in areas, such as trees or tables, and entities possessed by characters, such as shields and briefcases.

3.2 Concurrent process decomposition

The framework does not involve concurrent processes.

4.0 Dependency description

[*Note to the student:* this section describes all the ways in which the modules depend on each other.]

The only dependency among the framework modules is the aggregation by *GameArea* of *GameCharacter*.

5.0 Interface description

All classes in these packages are public, and thus the interfaces consist of all of the methods in their classes.

II. Architecture of Encounter Role-Playing Game

1. Introduction

1.1 Purpose

This document describes the design of the *Encounter* role-playing game.

1.2 Scope

This design is for the prototype version of *Encounter*, which is a demonstration of architecture, detailed design, and documentation techniques. The architecture is intended as the basis for interesting versions in the future. This description excludes the framework classes, whose design is provided elsewhere.

1.3 Definitions, acronyms and abbreviations

none

2. References

"Role-Playing Game Architecture Framework".

Braude, E. J., *Software Engineering: an Object-oriented perspective*

UML: The Unified Modeling Language User Guide by G. Booch, J. Rumbaugh and I. Jacobson, Addison-Wesley Pub Co; ISBN: 0-201-57168-4

IEEE standard 1016-1987 (reaffirmed 1993) guidelines for generating a Software Design Document

3. Decomposition description

The *Encounter* application is provided using three models: use case, class (object) model, and state. In addition, the relationship between the domain packages of *Encounter* and the framework will be shown.

[*Note to the student:* The IEEE standard is extended using sections 3.4 and 3.5 in order to describe these models. Recall that the other possible model is data flow, which we have not considered useful in this case. In the particular case of this video game, we chose to use the state description as part of the requirement as well as the design.]

3.1 Module decomposition (object model)

[*Note to the student:* this section should not duplicate the "detailed design" section described in the next chapter. We do not go into detail here regarding the contents of the packages.]

The package architecture for *Encounter* is shown in [figure 5.49](#).

Architecture and Modularization of *Encounter* Role-playing Game

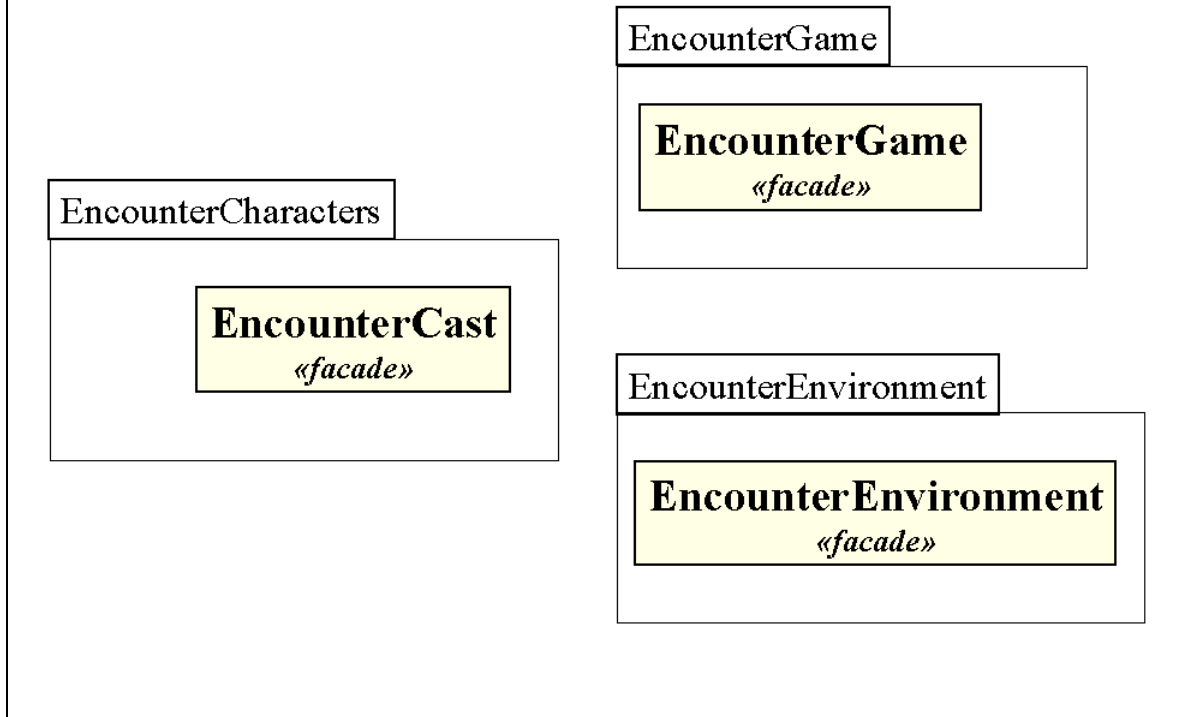


Figure 5.49 Architecture and Modularization of *Encounter* Role-Playing Game

The three packages are *EncounterGame*, *EncounterCharacters* and *EncounterEnvironment*. These have façade classes *EncounterGame*, *EncounterCast*, and *EncounterEnvironment* respectively. The façade class of each package has exactly one instantiation, and is an interface through which all dealings with the package take place. The remaining classes are not accessible from outside the package.

3.1.1 *EncounterGame* package

The *EncounterGame* package consists of the classes controlling the progress of the game as a whole. The package is designed to react to user actions (events).

3.1.2 *EncounterCharacters* package

The *EncounterCharacters* package encompasses the characters involved in the game. These include character(s) under the control of the player, together with the foreign characters.

3.1.3 *EncounterEnvironment* package

The *EncounterEnvironment* package describes the physical layout of *Encounter*, including the areas and the connections between them. It does not include moveable items, if any.

3.2 Concurrent process decomposition

There are two concurrent processes in *Encounter*. The first is the main action of the game, in which the player manually moves the main character from area to adjacent area. The second consists of the movement of the foreign character from area to adjacent areas.

3.3 Data decomposition

[*Note to the student:* describes the structure of the data within the application.]

The data structures flowing among the packages are defined by the *Area*, *EncounterCharacter*, and *EncounterAreaConnection* classes.

3.4 State model decomposition

Encounter consists of the states shown in [figure 5.50](#).

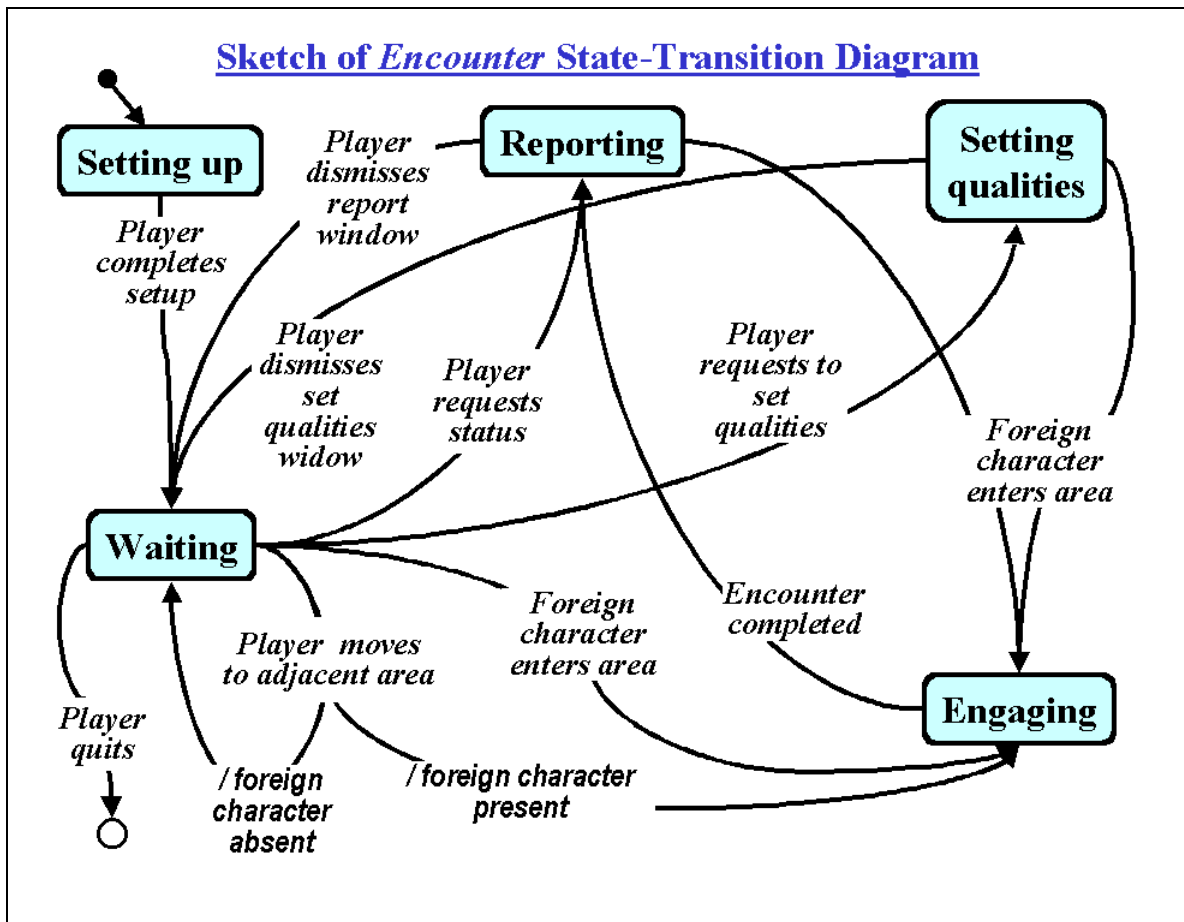


Figure 5.50 *Encounter* State-Transition Diagram

[*Note to the student:* this state diagram was provided in the SRS, where it was used to describe the requirements for *Encounter*. The remaining states mentioned in the requirements will be implemented in subsequent releases.]

3.5 Use case model decomposition

[*Note to the student:* this section is added to the IEEE specification, which does not address the *use case* concept. It has been added at the end of this section so as not to disturb the standard order.]

Encounter consists of three use cases: *Initialize* and *Engage Foreign Character*, as shown in [figure 5.51](#).

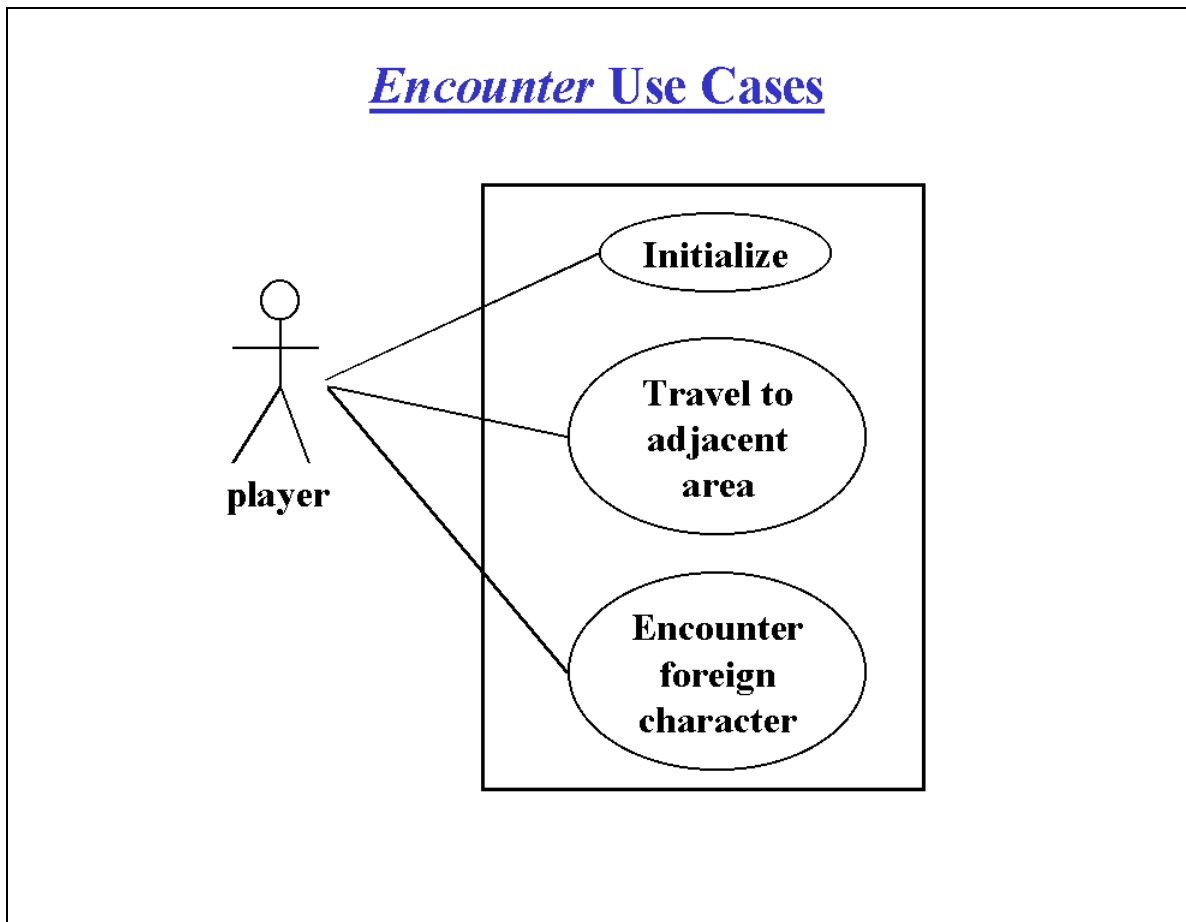


Figure 5.51 *Encounter* use cases

These use cases are explained in detail in the SRS, sections 2.2, and are detailed later in this document.

[*Note to the student:* details are given in the "detailed design" section.]

4.0 Dependency description

This section describes the dependencies for the various decompositions described in section 3.

[*Note to the student:* There are no significant dependencies among the use cases.]

4.1 Inter-module dependencies (object model)

The dependencies among package interfaces are shown in [figure 5.52](#).

Architecture / Modularization

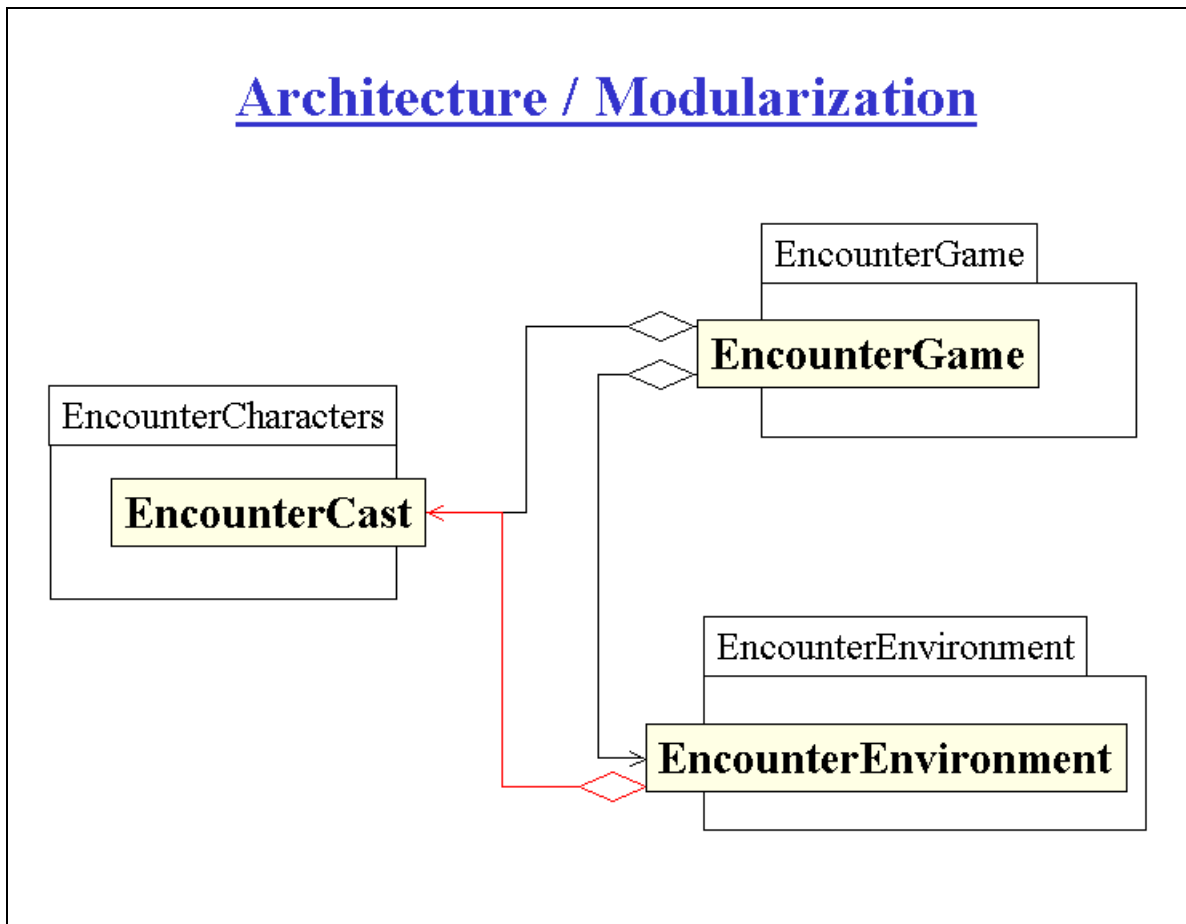


Figure 5.52 Architecture of *Encounter*

The *EncounterGame* package depends on all of the other *Encounter* packages.

The *EncounterEnvironment* package is designed to depend on the *EncounterCharacters* package. This is because the game's character interaction takes place only in the context of the environment. In particular, *Area* objects are responsible for determining the presence of the player's character together with the foreign character.

Dependencies among non-interface classes are explained later in this document. [Note: such dependencies are detailed design specifications.]

4.2 Inter-process dependencies

When an engagement takes place, the process of moving the main character about, and the process controlling the movement of the foreign character, interact.

4.3 Data dependencies

The data structures flowing among the packages are defined by the classes, whose mutual dependencies are described in section six of this document.

4.4 State dependencies

Each state is related to the states into which the game can transition from it.

4.5 Layer dependencies

The *Encounter* application depends on the Role-playing game framework as shown in [figure 5.53](#).

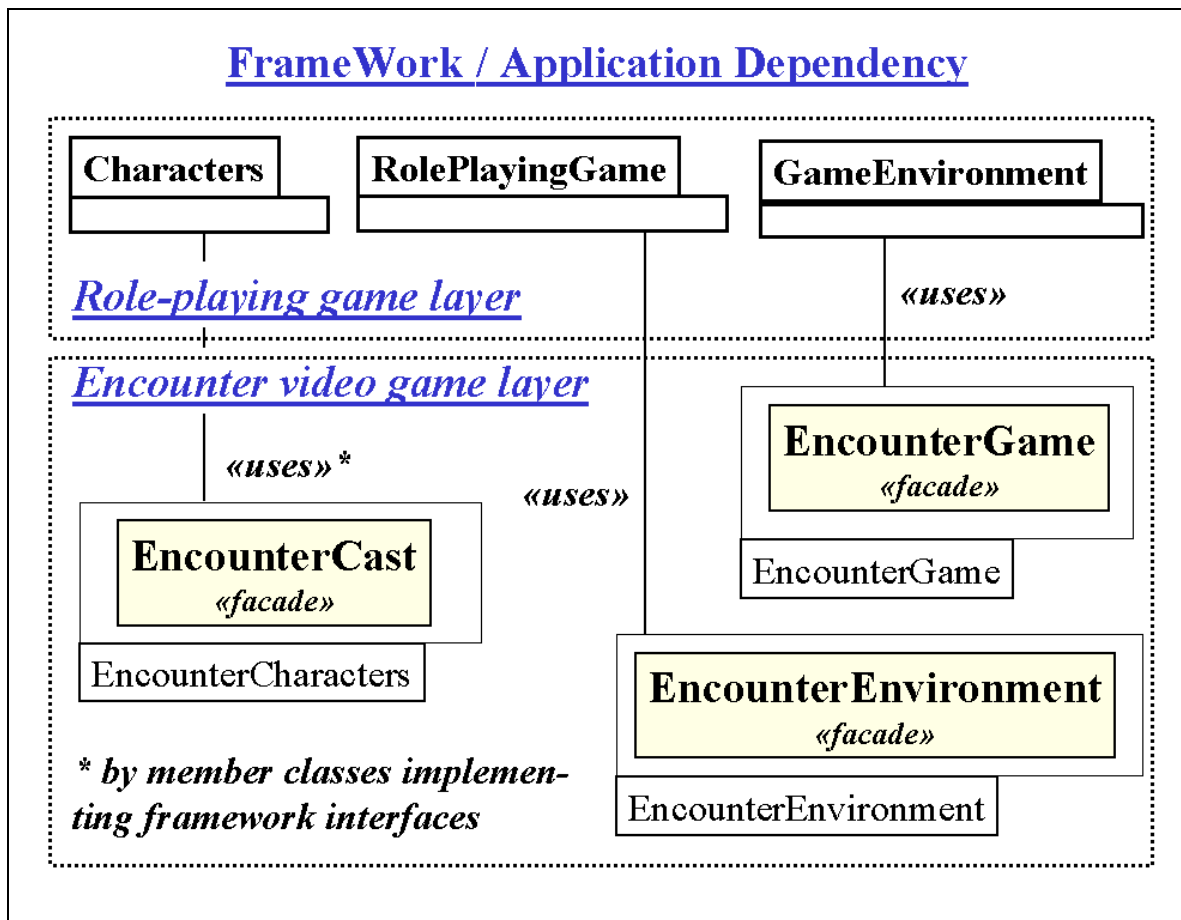


Figure 5.53 Framework / Application Dependency

Each application package uses exactly one framework package.

[Query: It would seem that the above figure is in error in that the **«uses»** of "EncounterGame" and "EncounterEnvironment" ought to be swapped??? – see Fig. 6.52 below]

5. Interface Description

This section describes the interfaces for the object model. Note that several of the classes described are defined in the design description of the Role-playing Game Framework.

5.1 Module interfaces

[*Note to the student:* Describes the interaction among the packages.]

5.1.1 Interface to the *EncounterGame* package

The interface to the *EncounterGame* package is provided by the *theEncounterGame* object of the *EncounterGame* façade class. It consists of the following.

1. `EncounterGame getTheEncounterGame()` // gets the only instance
 2. `GameState getState()` // current state of the `EncounterGame` instance
 3. `void setState(GameState)` // -- of the `EncounterGame` instance
- // Any event affecting the single `EncounterGame` instance:
4. `void handleEvent(AWTEvent)`

5.1.2 Interface to the *EncounterCharacters* package

The interface to the *EncounterCharacters* package is provided by the *theEncounterCast* object of the *EncounterCast* façade class. It consists of the following.

1. `EncounterCast getTheEncounterCast()` // gets the single instance
 2. `GameCharacter getPlayerCharacter()` // i.e., the unique character
 3. `GameCharacter getTheForeignCharacter()` // the unique character
- // Exchange quality values specific to the game area

5.1.3 Interface to *EncounterEnvironment* package

The interface to the *EncounterEnvironment* package is provided by the *theEncounterEnvironment* object of the *EncounterEnvironment* façade class. It consists of the following.

1. `EncounterEnvironment getTheEncounterEnvironment() // gets the façade object`

5.2 Process interface

[*Note the student:* we stated in section 3.2 that there are two processes involved in *Encounter*. There is a significant design decision to be made in regard to the interface to the foreign character movement process, and here is the place where the resulting interface is described. One option would be to have the foreign character be an individual thread, part of the *EncounterCharacters* package. This has advantages, but would require this character either to know the environment -- a disadvantage in terms of changing and expanding the game -- or to be able to find out about the environment dynamically, which would be an elegant design, but too ambitious for the scope of this case study. The architecture opts for another alternative, which is stated here.]

5.2.1 Player character movement process

The interfaces to the process which moves the player's character about the game, consists of the graphical user interfaces specified in the SRS. The process reacts to events described in section 3.4, and these are handled by the *EncounterGame* package, in accordance with its specification, described later in this document.

5.2.2 Foreign character movement process

The process of moving the foreign character is a separate process associated with, and controlled by the *CharacterMovement* object. This process is controlled by the methods inherited from *java.lang.Thread*.

Preliminary notes on Detailed Design (sections 6 for I and II)

We have to describe two detailed designs. The first is the detailed design of the Role-Playing video game framework, and the second that of *Encounter* application. By separating them, we are making it easier to re-use and maintain the framework.

I. Detailed design of Role-Playing Game Framework *continued*

[*Note to the student*: this is a continuation of the SDD (architecture)]

6. Detailed design of Role-Playing Game Framework

The overall architecture of the packages described in this section can be found in Part 1.

6.1 Module detailed design

[*Note to the student*: these sections give *all* of the non-trivial required details on each of the modules described in section 3.1 in this SDD (i.e., for the game framework).]

6.1.1 Role-playing game package

All mouse events are listened for by objects of the class *RPGMouseListener*, which inherits from *MouseListener*, as shown in [figure 6.50](#).

Detailed Design of *RolePlayingGame* Package

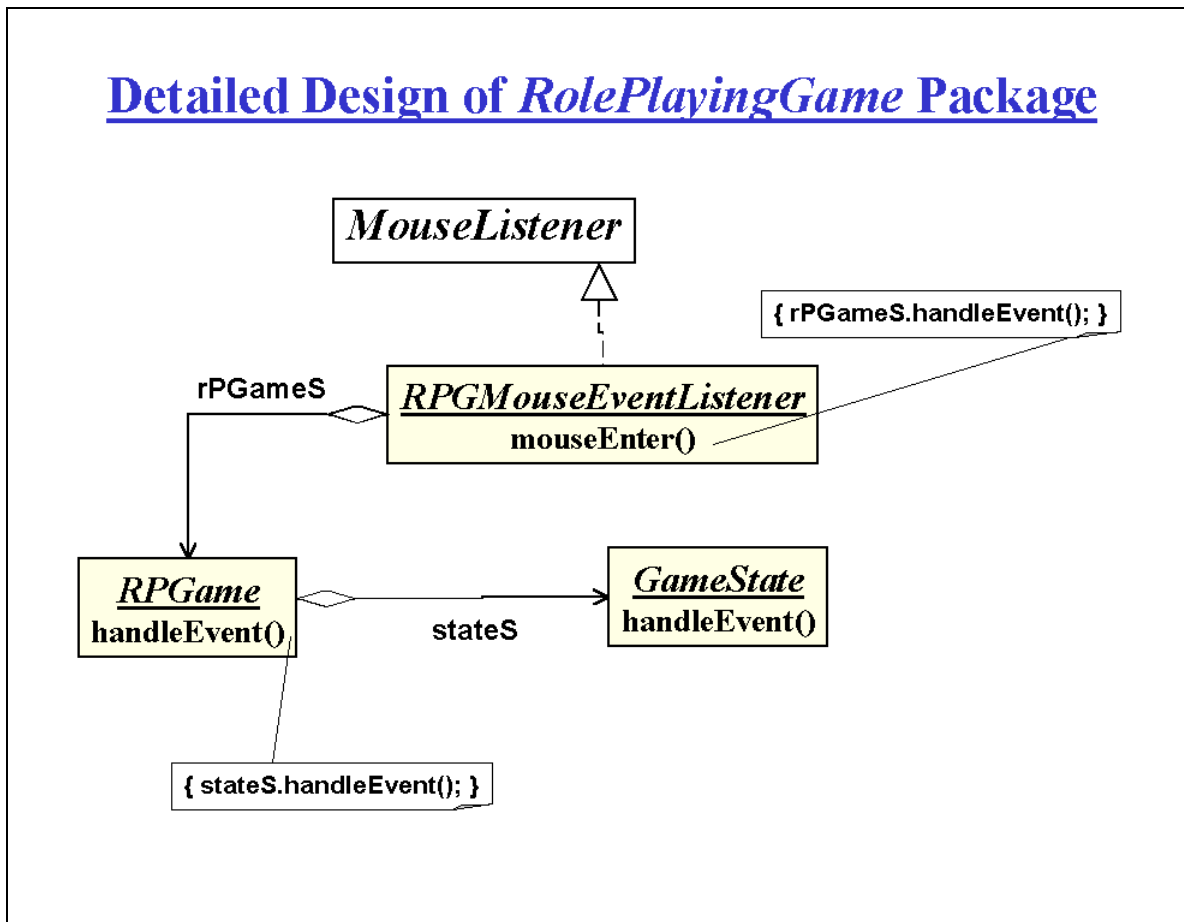


Figure 6.50 Detailed Design of *RPGGame* (Role-Playing Game)Package

Each object which is sensitive to mouse events asks an *RPGGame* object to handle the event. *RPGGame* passes control to the *handleEvent()* method of its aggregated *GameState* object. The sequence diagram for this is shown in [figure 6.51](#).

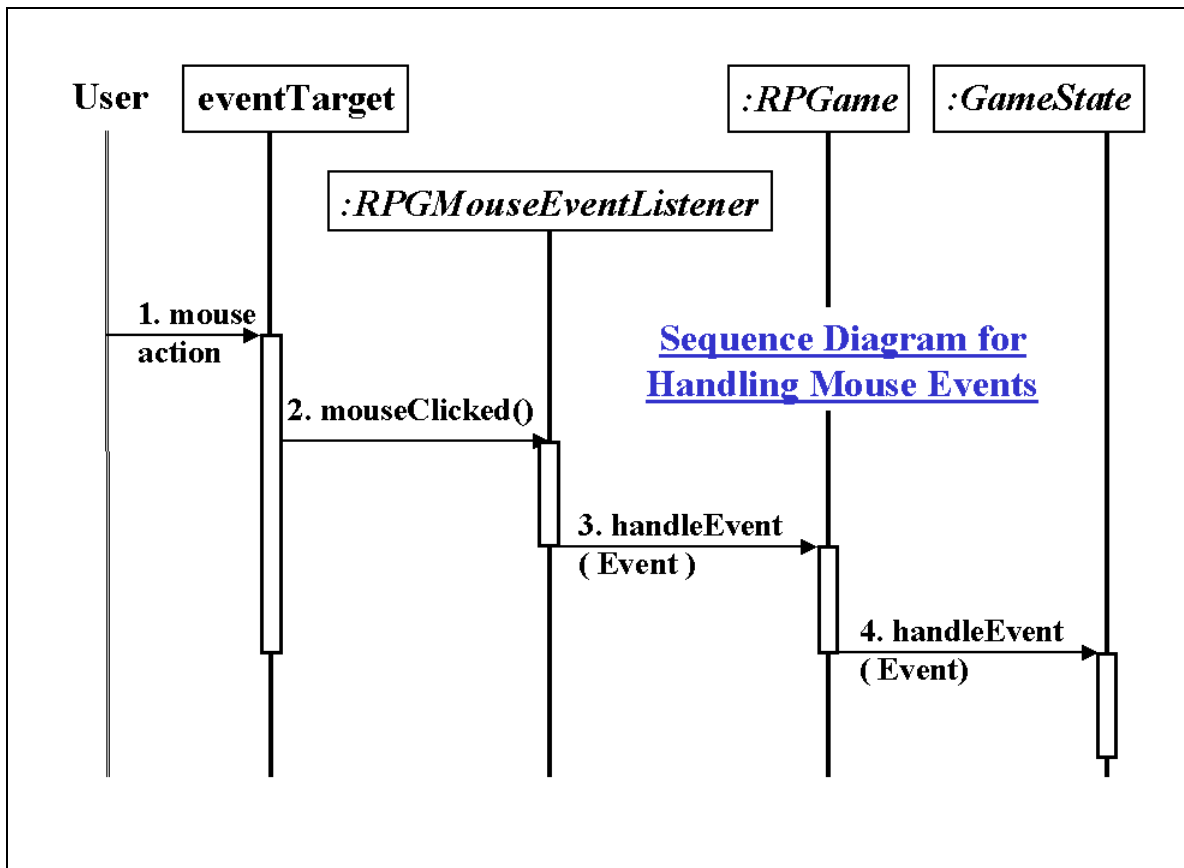


Figure 6.51 Sequence Diagram for Handling Mouse Events

For the current release, the methods are either trivial, or are shown in the figure.

[*Note to the student:* pseudocode for selected methods within selected classes may be included here. In addition, detailed use cases can be included. Since the methods and their details are still one or two lines in this case, it is sufficient to elaborate on the (barely) nontrivial methods with the notation shown in the figure.]

6.2.1 The Characters package

This section elaborates on section 3.1.2 of this SDD.

There is one class in the *Character* package: *GameCharacter*.

6.2.1.1 GameCharacter class

Methods of GameCharacter

setName().

Preconditions: none; Postconditions: none; Invariants: none

Its pseudocode is as follows.

IF aName parameter OR maxNumCharsInName() make no sense

Set name to default value and show this in system
window

ELSE

IF parameter string too long

truncate at maxNumCharsInName()

ELSE assign the parameter name

6.2.1 The GameEnvironment package

This package is described completely by figure 5.48 in section 3.1 of this SDD.

6.2.1 The Artifacts package

[Not applicable in this iteration.]

[End of Detailed Design for the Framework packages]

II. Detailed design of *Encounter* -- continued

6. Detailed design for *Encounter*

The overall architecture showing the relationships among the packages, and the domain classes, is described in this section is shown in [figure 6.52](#). (??confirm)

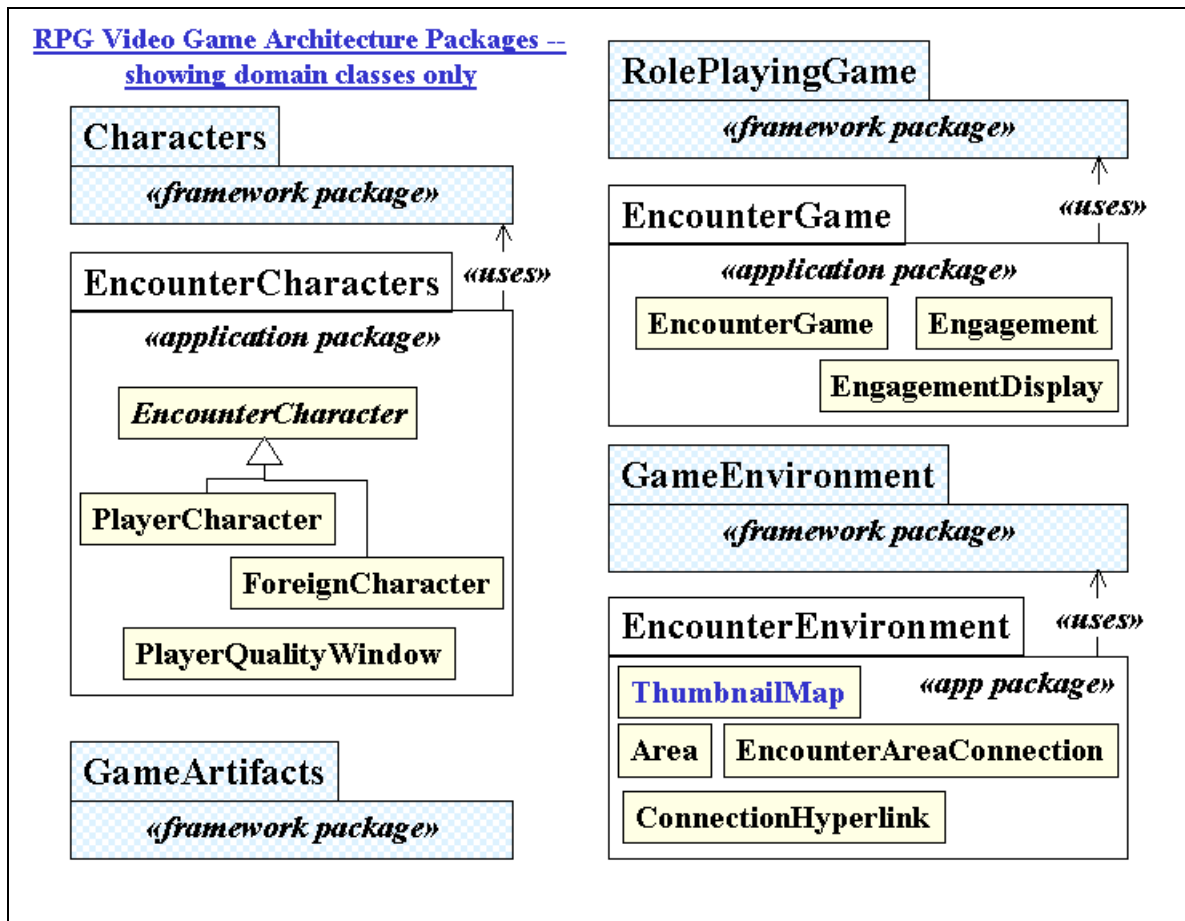


Figure 6.52 Video Game Architecture Packages, Showing Domain Classes

6.1 Module detailed design for *Encounter*

[Note to the student: these sections give all of the required details on each of the modules described in section 3.1 in this SDD (i.e., for *Encounter*).]

6.1.1 The *EncounterGame* package

[Note to the student: This section gives all of the required details on section 3.1.1 in this SDD. It describes completely the classes of the *EncounterGame* package, and all of their nontrivial behavior. Most of this is described by the state transition diagram.

In the interests of keeping the class model free of clutter, we do not show all object references.]

The state diagram for *Encounter* is shown elsewhere (check??). To realize these states and transitions, the *EncounterGame* package object model is designed as in [figure 6.53](#).

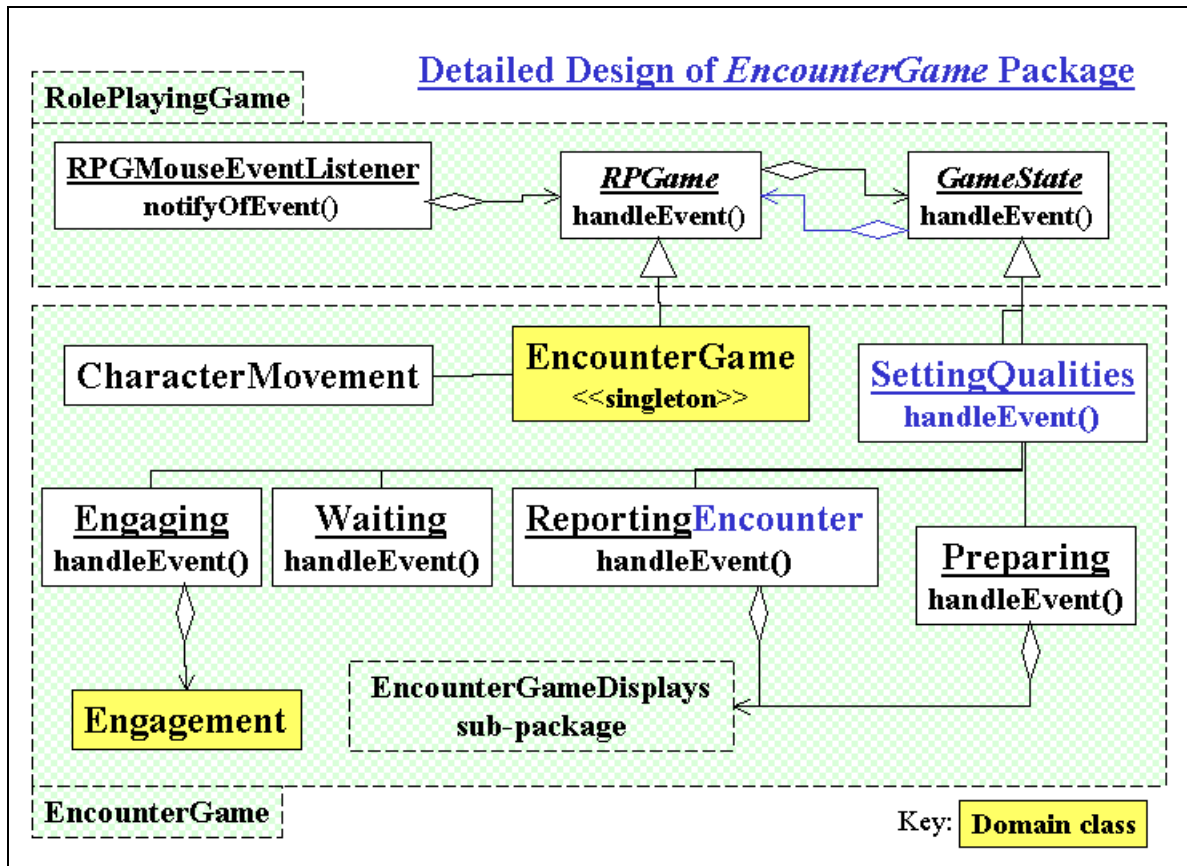


Figure 6.53 Detailed Design of *EncounterGame* Package

There is exactly one instance of the *EncounterGame* class. The states of this object reflect the states and sub-states shown in the above state-transition diagram (?). The *Engaging* state aggregates an *Engagement* object that encapsulates the engagement of the game characters. The *Engagement* class aggregates a display called *EngagementDisplay*. The latter is mouse-sensitive, registering with an *RPGMouseListener* object. When the *Encounter* game is executed, this listener object references the *EncounterGame* object (an *RPGGame* object). This enables *EncounterGame* to handle all events according to its state using the *State* design pattern. The *EncounterGame* package has the responsibility of directing the

movement of the foreign character over time. This is performed by methods of the class *CharacterMovement*, which is a thread class.

State classes need to reference other packages in order to implement *handleEvent()*, and this is done through the façade objects *EncounterCast* and *EncounterEnvironment*.

6.1.1.1 The *EncounterGameDisplays* sub-package of the *EncounterGame* package

Displays corresponding to some of the states are handled by a separate sub-package, *EncounterGameDisplays*, which is shown in [figure 6.54](#).

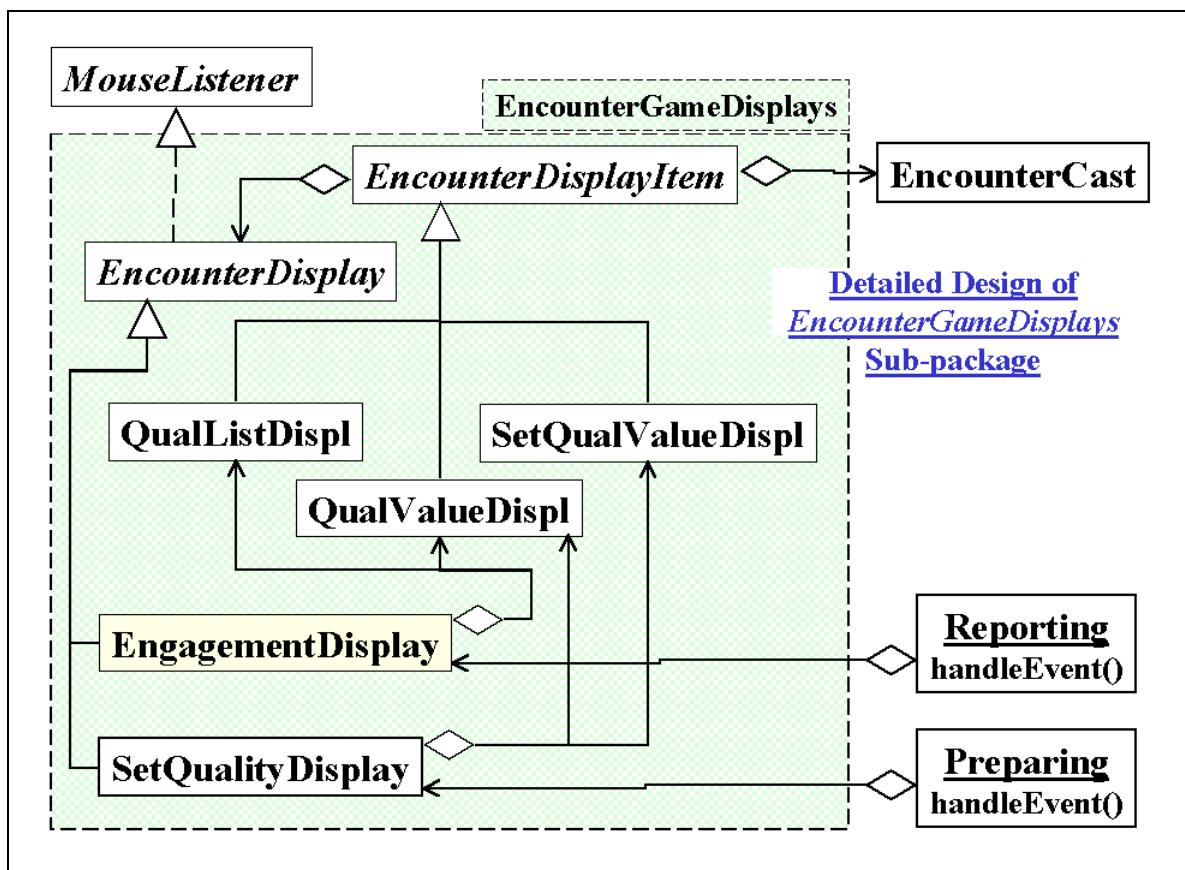


Figure 6.54 Detailed Design of *EncounterGameDisplays* Sub-Package

QualListDispl is a list box consisting of the qualities of *Encounter* characters.

QualValueDispl is a read-only text box for displaying the value of a quality.

SetQualValueDispl is an editable text box for setting the value of a quality.

EncounterDisplayItem abstracts the properties and methods of displayable Encounter items such as these.

EngagementDisplay is designed to display the current value of any selected quality.

SetQualityDisplay is designed to enable the player to set the value of any quality.

EncounterDisplay abstracts the properties of these displays, and is a mediator base class.

This document does not provide further details of the design of these classes.

6.1.1.2 Sequence diagrams for event handling

6.1.1.2.1 Player dismisses report window event

[Figure 6.55](#) shows the sequence involved in dismissing the engagement display. (This dismissal event is shown on the state transition diagram.)

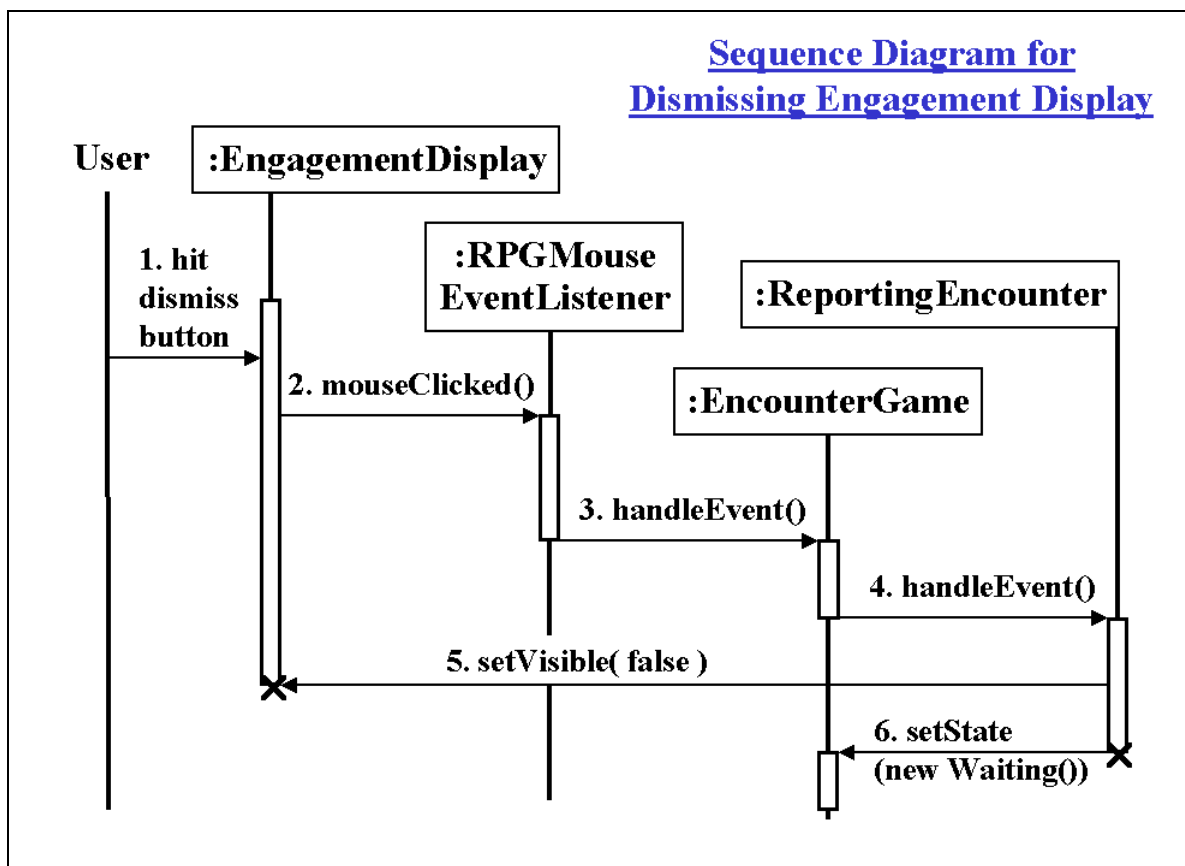


Figure 6.55 Sequence Diagram for *Dismiss Engagement Display*

6.1.1.2.2 Player dismisses report window event

[Figure 6.56](#) shows the sequence involved in dismissing the engagement display. (This dismissal event is shown on the state transition diagram.)

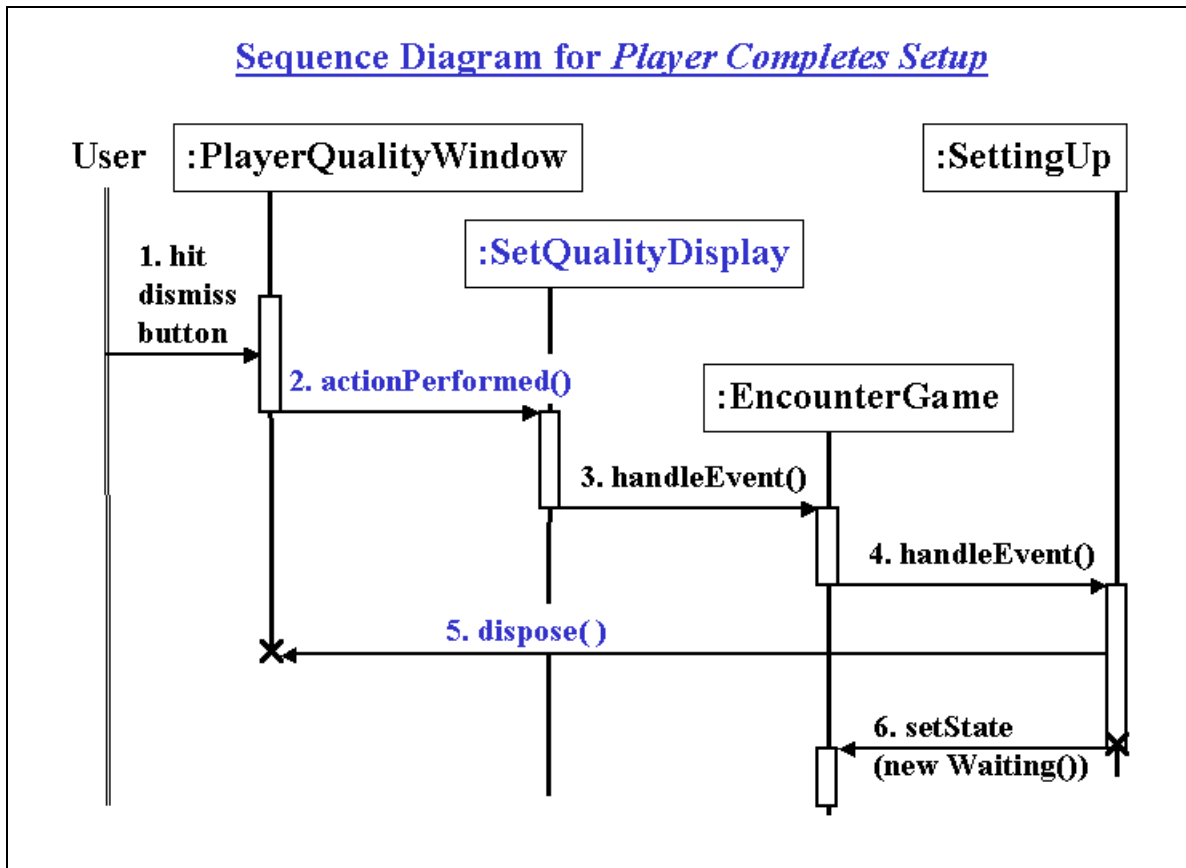


Figure 6.56 Sequence Diagram for *Player Completes Setup*

6.1.1.2.3 Player moves to adjacent area event

[Figure 6.57](#) shows the sequence when the player moves to an adjacent area by clicking on a connection hyperlink. (This dismissal event is shown on the state transition diagram.)

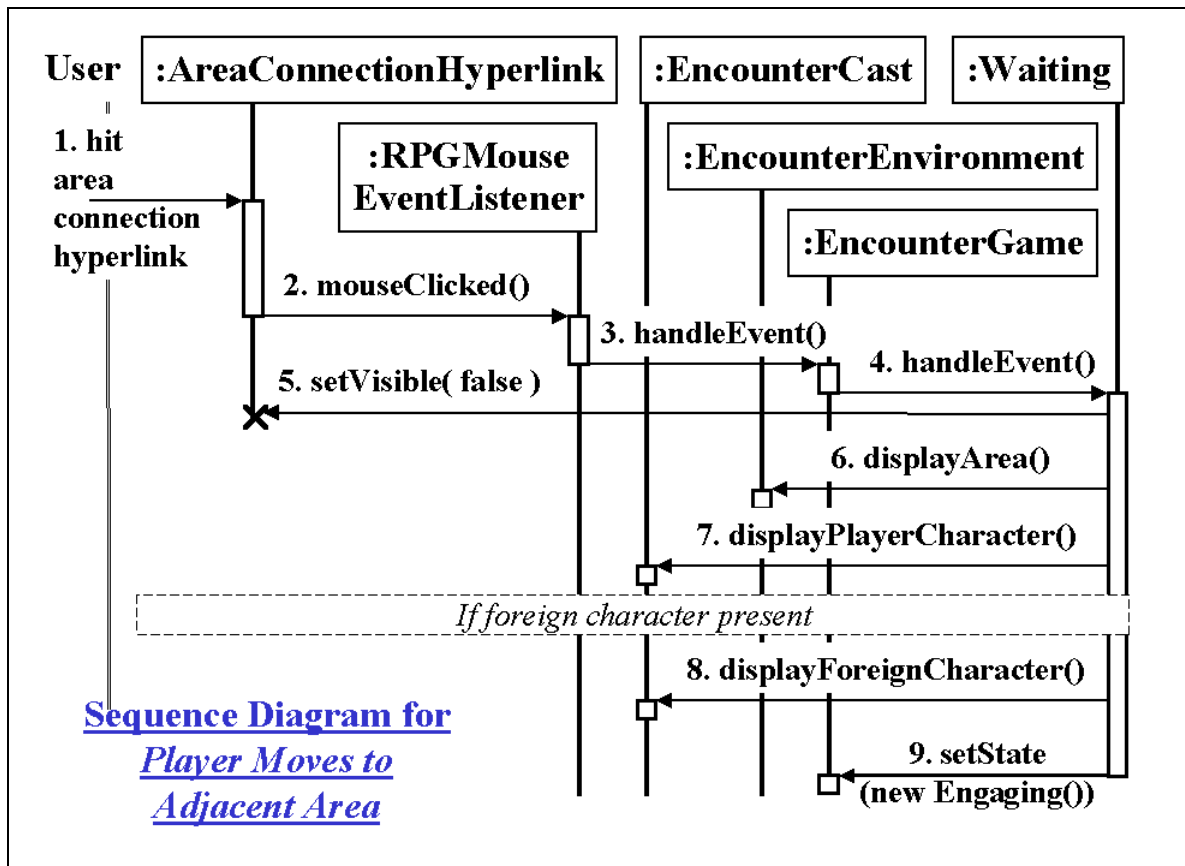


Figure 6.57 Sequence Diagram for *Player Moves to Adjacent Area*

6.1.1.2.4 Sequence diagrams for remaining events

The remaining events are handled very similarly to those described in sections 6.1.1.1 through 6.1.1.3.

6.1.1.CM The *CharacterMovement* class

[*Note to the student:* as in the SRS, we are using an alphabetical "numbering" scheme to make it easier to find and insert classes. There is also a benefit in being able to trace the requirements to the SRS.]

This class controls the movement of the foreign character.

Inheritance

This class inherits from *java.lang.Thread*.

Methods of CharacterMovement

```
public static EncounterGame run();
```

Starts the foreign character in the dungeon. Moves the foreign character from area to area via area connections, changing areas to a randomly selected neighbor, at random times averaging every two seconds.

6.1.1.EG The *EncounterGame* class

This is the façade singleton for the *EncounterGame* package.

Inheritance

This class inherits from *RPGame* in the framework.

Attributes of *EncounterGame*

EncounterGame encounterGameS: This is the singleton *EncounterGame* object

Constructors of *EncounterGame*

```
private EncounterGame():
```

Preconditions: none.

Postconditions: creates an *EncounterGame* instance.

Methods of *EncounterGame*

```
public static EncounterGame getTheEncounterGame();
```

Preconditions: none.

Postconditions: encounterGameS not null

Returns: encounterGameS

6.1.1.EN The *Engagement* class

This class encapsulates engagements between the player's character and the foreign character, and corresponds to requirement 3.2.EN.

Methods

```
public void engage(); // corresponds to requirement 3.2.EN.3.1
```

Preconditions: the player's character and the foreign character are in the same area.

Postconditions: the values of the characters' qualities are as required by SRS requirement 3.2.EN.3.1; the player's character and the foreign character are in random, but different areas.

6.1.1.ZZ The *Engaging, Waiting, Preparing, and Reporting* classes

[The "ZZ" numbering is used to collect classes that are not specified individually.]

Inheritance

These classes inherit from *GameState* in the framework package.

Each of these classes implements its *handleEvent()* method in accordance with the sequence diagrams of section 6.1.1.1.

6.1.2 The *EncounterCharacters* package

[This section elaborates on section 3.1.2 in this SDD.]

The design of the *EncounterCharacters* package is shown in [figure 6.58](#). It is implemented by the *Façade* design pattern, with *EncounterCast* as the façade object.

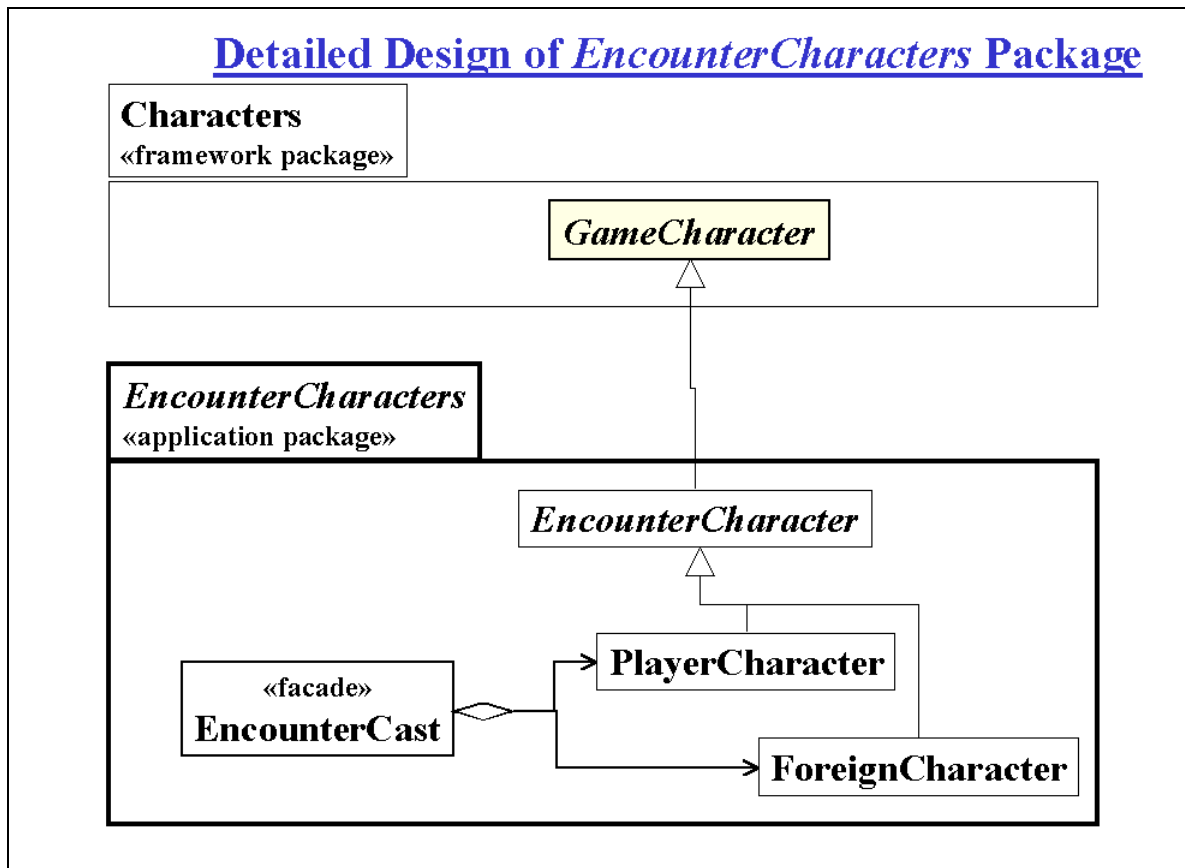


Figure 6.58 *EncounterCharacters* Package

6.1.2.EC The *EncounterCharacter* class

This class encapsulates the requirements for *Encounter* characters that are not covered by *RPGGameCharacters*. It satisfies requirements SRS 3.2.EC.

Class invariant: The values of `qualValueI` are non-negative (see "attributes" below for the definitions of `qualValueI`).

Inheritance

This class inherits from *GameCharacter* in the framework package.

Attributes of *EncounterCharacter*

These satisfy requirement SRS 3.2.EC.1

```
private static final String[] qualityTypeS
```

This represents the qualities that *Encounter* characters possess. These are concentration, stamina, intelligence, patience, and strength.

private String imageFileI

Contains the gif image of the character

private float[] qualValueI

This is an array containing the values of the qualities.

private float[] oldValueI

This is an array containing the previous quality values.

Constructor of *EncounterCharacter*

These satisfy requirements 3.2.EC.3 of the SRS

Null constructor

Postcondition: the qualities are all equal fractions of 100.

protected EncounterCharacter(String nameP)

Postconditions:

(1) the qualities are all equal fractions of 100.

(2) the character's name is *NameP*

Methods of *EncounterCharacter*

public void showCharacter()

Displays the character, facing left or right as specified.

Known defect: facing right vs. left is performed algorithmically: should be different images.

public float getQualityValue(String qualityP)

public synchronized void adjustQuality(String qualityP, float qualityValueP)

This method satisfies requirement 3.2.EC.3.2.

Invariants: none

Preconditions:

qualityP is in *qualityTypesS[]*

AND *qualityValueP* ≥ 0

AND *qualityValueP* \leq the sum of the quality values

Postconditions:

qualityP has the value *qualityValueP*

AND

the remaining quality values are in the same proportion
as

prior to invocation, except that values less than 0.5 are
zero.

The following is the pseudocode for the method *adjustQuality()*.

Set the stated quality to the desired amount

IF the caller adjusts the only non-zero quality value,

*divide the adjustment amount equally among all
 other qualities.*

*ELSE change the remaining qualities, retaining their
mutual proportion,*

*Set each quality whose value is now less than 0.5 to
zero*

public float getQualityValue(String qualityP)

Preconditions: *qualityP* is a valid quality string

Returns: the value of *qualityI*

```
public float getOldValue( String qualityP )
```

Preconditions: *qualityP* is a valid quality string

Returns: the value of *oldQualityI*

```
public float getTolerance( )
```

Returns: the value below which quality values cannot go

```
protected int maxNumCharsInName()
```

Returns: the maximum number of characters in the names of
Encounter characters.

```
public float sumOfQualities()
```

Returns: the sum of the values of the qualities

This method satisfies requirement 3.2.EC.3.2

```
public void showCharacter( Component componentP, Graphics drawP, Point posP, int  
heightPixP, boolean faceLeftP )
```

Displays the character in *componentP*, with center at *posP*, with height
heightPixP, facing left if *faceLeftP* true

This method satisfies requirements 3.2.PC.1 and 3.2.PQ.1

```
private void setQuality( String qualityP, float valueP )
```

Preconditions: *qualityP* is a valid quality string

Sets the quality indicated by the parameter to *valueP* if the latter is \geq
0.5, otherwise sets *valueP* to zero.

This method satisfies requirement 3.2.EC.2 (lower limit on nonzero
quality values)

6.1.2.ES The *EncounterCast* class

The method specifications for this singleton, interface class are given in section 5 of this document.

6.1.2.FC The *ForeignCharacter* class

This class is analogous to *PlayerCharacter*, described next, and is designed to satisfy the SRS 3.2.FC

6.1.2.PC The *PlayerCharacter* class

This class is designed to satisfy the requirements 3.2.PC.

Inheritance

This class inherits from *EncounterCharacter*.

Attributes:

```
private static final PlayerCharacter playerCharacterS;
```

This is the singleton object representing the player's character.

Methods:

```
public static final PlayerCharacter getPlayerCharacter();
```

This method returns playerCharacterS.

6.1.3 The *EncounterEnvironment* package

The classes of this package describe the environment in which the game takes place.

It is shown in [figure 6.59](#).

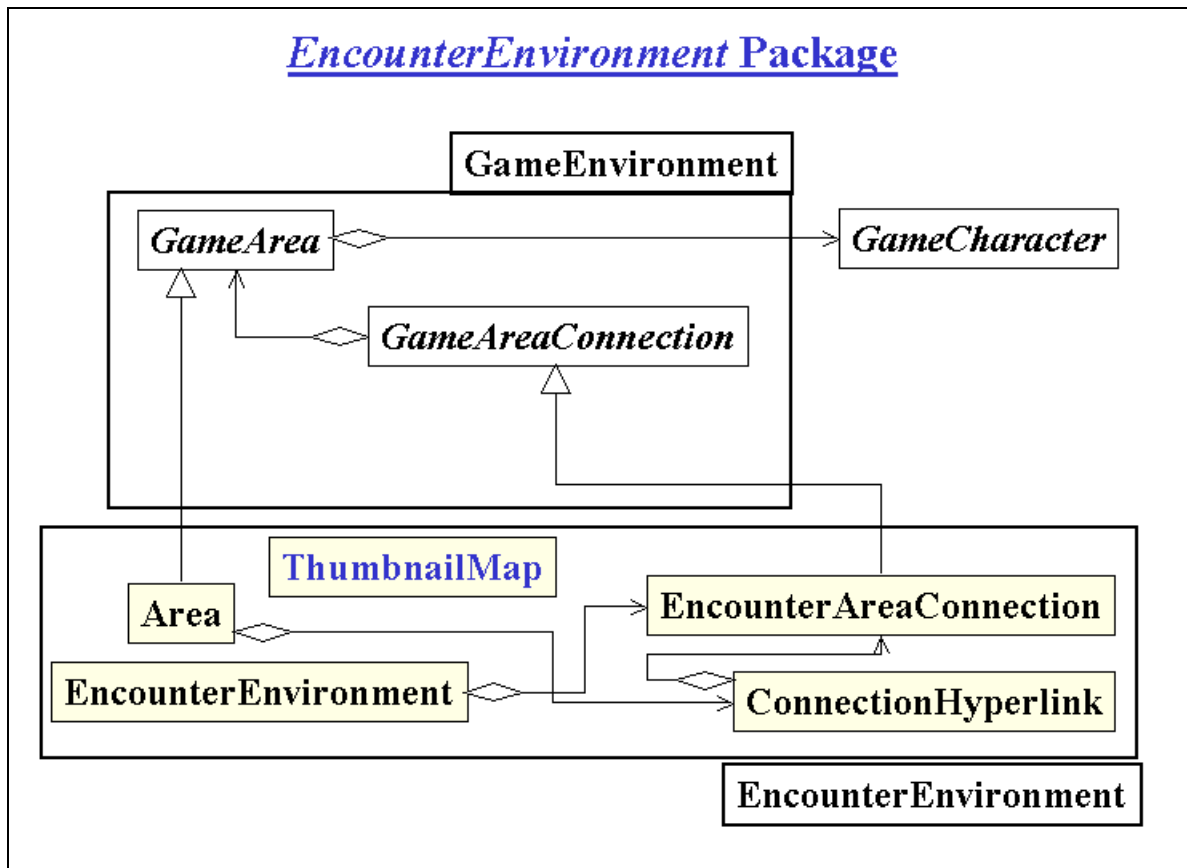


Figure 6.59 *EncounterEnvironment* Package

6.1.3.AR Area class

This class encapsulates the places in which the characters exist, and corresponds to requirement 3.2.AR.

Inheritance

This class inherits from *GameArea*.

Attributes:

private String nameI; // corresponding to requirement 3.2.AR.1.1

private Image imageI; // corresponding to requirement 3.2.AR.1.2

private String[] qualitiesI; // corresponding to requirement 3.2.AR.1.3

private Vector connectionHyperlinksI;

Methods:

public void display() shows the area object's image on the monitor.

public static Area getArea(String areaNameP) returns the area corresponding to areaNameP according to requirement 3.2.AR.2.2

public static AreaConnection getAreaConnection(String areaConnectionNameP) returns the areaConnection object corresponding to areaConnectionNameP according to requirement 3.2.AR.2.2

6.1.3.CO EncounterAreaConnection class

This class encapsulates the ways to get from areas to adjacent areas. It inherits from *AreaConnection*, and corresponds to requirement 3.2.CO.

Inheritance

This class inherits from *GameAreaConnection* in the framework package.

Attributes:

private Area firstAreaI; // corresponding to requirement 3.2.CO.1.1

private Area secondAreaI; // corresponding to requirement 3.2.CO.1.1

Methods: these are accessors for the attributes above.

6.1.3.EE EncounterEnvironment class

This is the façade class for the EncounterEnvironment package.

Attributes:

private EncounterEnvironment encounterEnvironmentS; // the singleton *Façade* object

// [Area name][Area connection name]["North" | "South" | "East" | "West"]:

private String[3] layoutS;

Methods:

public static EncounterEnvironment getEncounterEnvironment ()

Returns: encounterEnvironmentS

```
public static String[3] getLayout()
```

Returns: layoutS

The remaining methods are specified by section 5 of this document.

6.1.3.CH ConnectionHyperlink class

This class encapsulates the ways to get from areas to adjacent areas. It corresponds to requirement 3.2.CH. This class implements the *MouseListener* interface.

Attributes:

```
private Connection connectionI; // the corresponding connection
```

Methods:

The only meaningful method is `mouseClick()`.

6.1.3.TH ThumbnailMap class

This class provides an image of the areas in the immediate vicinity of the player character.

Currently, the entire game environment is hard coded.

Methods:

drawMap() draws the thumbnail picture in the *Component* parameter provided, scaling itself to the size of the component.

6.2 Data detailed design

There are no data structures besides those mentioned as part of the classes in section 6.1.