# Chapter 8

# Design

'A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools.'

—*Douglas Adams*

This chapter is the about the contents, structure and behavior of the Sheep framework. It will demonstrate how the various components have practical value, academic value or both.

## 8.1 Goals

The overall goal of the framework is to allow the students to save time, but the framework itself may also be instrumental in teaching the students software architecture. Consider these opposite (yet equally feasible) approaches for framework design:

- Create a 'library' rather than a framework, where the flow of control is dictated by the client code, i.e. create a flat API as free from 'architecture' as possible. The rationale behind this approach is that the library should be unintrusive. The internal architecture should not 'color off' onto the architecture of the games, as to allow the students to come up with their own solutions.
- Create an extensible framework where the structure and behavior of as many components as possible are based on known design patterns. The rationale behind this approach is quite the opposite of the former one; by creating an 'intrusive' framework which requires that things are done according to good practices, the students might actually be 'forced' to learn design patterns. Examples will follow throughout this chapter.

The Sheep framework will be an 'intrusive' framework, because it is believed that the students do not need to *implement* a design pattern in its entirety to *use* and *understand* the pattern. As an example, object iteration in the spatial partitioner could be done using the *Visitor* pattern. If the student wants to draw the items in the spatial partitioner on screen, there would be no other way of doing so other than creating a *vistor* class. By implementing a visitor class, it will become obvious to the student how the entire Visitor pattern works, even though he or she did not create the part of the code that actually iterated the objects.

The idea is that the Sheep framework becomes something more than just a productivity booster; it also becomes a way of learning some principles in software architecture directly.

In summary, the two overall goals for all major components in the Sheep framework are:

G1 Simplify a common task in game development, so the students can spend more time on structure and less time on technical matters.

G2 Use known design patterns to interact with client code, as to teach students these design patterns.

Not all components will achieve both of these goals; some may be of no direct academic value to the students, and may simply exist as a convenience. Some components may be of great academic value, but may not be practical in a certain game genre or specific game design.

## 8.2 Oveview

The Sheep framework is organized in the `sheep` package, with several subpackages. Alphabetically, they the packages are as follows:

- `sheep.audio` provides components for loading and playback of sound.
- `sheep.collision` contains collision detection and spatial partitioning components.
- `sheep.game` assist in structuring the game logic (the model) of the game.
- `sheep.graphics` contains components for loading images and fonts.
- `sheep.gui` holds the graphical user interface system.
- `sheep.input` contains the input devices, and the interfaces needed to subscribe to events.
- `sheep.math` contains some math classes which aren't directly related to collision detection.
- `sheep.util` is meant to contain miscellaneous components, but for now it only contains a singleton which keeps track of time between frames.

We will explore key concepts and design choices from some of these packages, and omit those which are less interesting. All packages and classes are also briefly described in appendix C.

## 8.3 sheep.game

This package provides components which help organize the game model. We're going to look closer these two concepts:

- *Worlds and Layers*. Organizes the game world in layers.
- *States*. Keeps track of the high-level states of the game.

### 8.3.1 Worlds and Layers

Organizing graphics in *layers* is often a feasible approach in 2D games. A layered world manager would be applicable in most of the student projects of 2008.

- `Sprite` represents something drawable which can be drawn on screen with a certain position, orientation and scale. It can also have a geometric shape attached for collision detection.

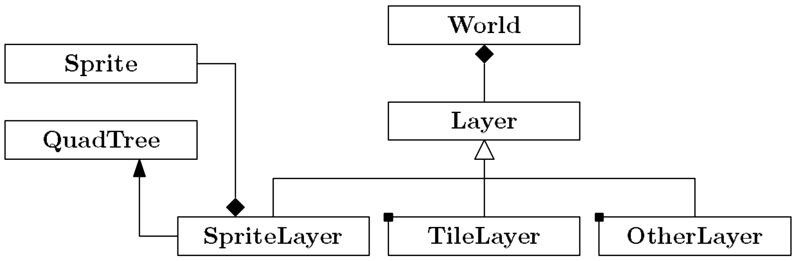

Figure 8.1: Classes in object management system.

- `Layer` is an abstract class which represents one visible layer as seen by the player. Layers are specially crafted to perform a certain task, for instance `SpriteLayer` which draws sprites with arbitrary positions, and `TileLayer` which draws tiled graphics useful for role-playing games and platformers. Client classes may extend the `Layer` class to fulfill a special need.
- `QuadTree` is part of the spatial partitioning system, which will be explained in section 8.4.1.

From client perspective, one more more layers are created, either by using one of the preexisting classes or creating custom ones, then those layers are added to a `World` object. Each frame, only the `World` object is updated and drawn; the `World` object propagates the update and draw calls into the child layers. It is then up to the specific `Layer` implementation to decide what to do.

#### Practical Value

Organizing the game graphics in layers[1] is an obvious move in many 2D games, because the ordering of the graphics naturally occur in 'layers' inside our heads. The idea is that the `World` class becomes an aid in creating the 'view' part of the Model-View-Controller (appendix B.7). Client code needs only insert `Sprite` objects into the `World`, and they will be updated and drawn appropriately.

Being able to 'forget' about drawing objects on screen should save the students some time and enable them to focus on more relevant matters.

#### Academic Value

Although organizing the graphics in layers is common in 2D games and in that respect can be regarded as a pattern, there are no *widely accepted* design patterns at work here.

### 8.3.2 Game States

When designing the internals of a game, you typically want some main controller object similar to the `Game` singleton in Microsoft XNA. This controller contains methods for loading content, updating its internal state, drawing itself, and responding to input events.

A game typically contains more than one state, for instance the player might be presented with a loading screen or a title screen just after the game has launched. When the player chooses to start the game, the internal state changes again; this time to the main game content. In the middle of a game, the player might wish to pause the game, which is another example of a game state. In each scenario, the way things are drawn on screen and how the game should respond to user input is different.

[1]Layered graphics. Not to be confused with 'Layers', the architectural pattern.

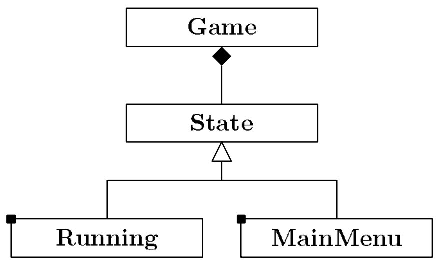

Figure 8.2: The classes in the game state system.

The Sheep framework will contain a main `Game` singleton consisting of any number of states in the form of `State` objects, as seen in figure 8.2. The framework will also provide one default state class, called the `RootState`, which cannot be popped from the *state stack* (which will be explained shortly). Client classes (shown with dashed lines in 8.2) will implement the actual game logic, drawing code and input management code.

Let us extend the previously mentioned usage scenario a little.

1. The player starts the game and is presented with a loading screen or title screen.
2. The player is presented with the main menu and chooses to start the game.
3. The player starts the game, and is presented with the main game content.
4. The player pauses the game, and is presented with the paused game menu.
5. The player resumes the game, and returns back the main game content.
6. The player pauses the game (again), and is presented with the paused game menu.
7. The player chooses to quit the game, and returns the the main menu.

Looking at the steps in this scenario, we discover that they follow the last in first out (LIFO) principle.

1. (+1) Push loading/title state.
2. (+1) Push menu state.
3. (+1) Push running game state.
4. (+1) Push paused game state.
5. (-1) Pop paused game state.
6. (-1) Push paused game state.
7. (-2) Pop paused game state and running game state.

At any time, the top state on the stack will be the delegate of the messages from the `Game` singleton. In some cases, the current state may need to access the state directly beneath it. For instance, when a game is paused, one may wish to draw a menu over the running game state.

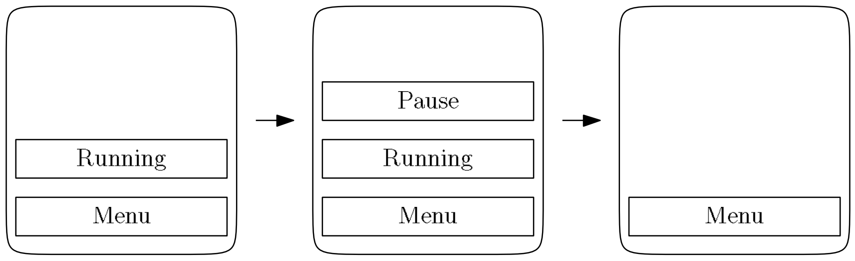

Figure 8.3: The state stack undergoing the changes of steps 5, 6 and 7 in the scenario.

#### Practical Value

Implementing a state stack is not very difficult by itself, and most students will be able to do this relatively quickly. However, one must also identify which interfaces to inherit from in the Android API to receive input events, and how to handle and process those events. This may be time consuming and distract students from more relevant issues. Having a complete state system in place is beneficial because it allows relevant input events to be presented more clearly and quickly to the students.

#### Academic Value

The *State* pattern is a well known pattern which allows an object partially change its class at run-time (see appendix B.3). This is exactly what the `Game` class does when new states are pushed onto its state stack. There is no need to derive from the `Game` class in order to override certain member functions; specific game behavior should be implemented via subclasses of the `State` class. A `State` is not unlike a 'view' in the Model-View-Controller; each state represents a different view of the game.

## 8.4 sheep.collision

This package provides functionality for detecting interactions between objects in the game world, and generates collision events which may be subscribed by observers. It also contains components which divide space with various techniques to save processing time. Let us take closer look on how these features are implemented and designed; *collision detection*, and *spatial partitioning*.

### 8.4.1 Spatial Partitioning

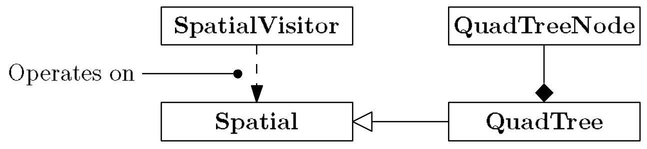

Figure 8.4: Classes of spatial partitioning system.

A quadtree (see section 5.3.2) will be used in the Sheep framework, due to the high 'always works' factor. The fixed grid approach can be faster in some cases, but it significantly enough to justify the creation of both spatial partitioning techniques.

- The `QuadTree` class will be the main controller class the client code interacts with when it inserts/removes objects into/from the tree. This class is also responsible for overlap queries used to optimize collision detection and world culling. The `QuadTree` is composed of `QuadTreeNode` objects which contain the actual references to the objects in the tree.
- The `Spatial` interface is a tentative name for an object that is insertable into the tree.
- The `SpatialVisitor` interface is an interface visitor objects.

There is a good reason for creating separate interfaces for objects that are insertable into the class; loose coupling between components. The most popular architectural pattern by far in 2008 was the Model-View-Controller (appendix B.7), as we discovered in section 4. It is important that the framework does not violate the principles of this pattern. Consider; instead of creating a separate interface, we could have required that general *Sprite* objects (with the visual representation embedded) be inserted into the tree. But this would merge the 'view' with the 'model', and would hinder the use of this pattern.

Using a separate interface combined with the Visitor pattern (appendix B.1), we achieve a more loosely coupled solution.

#### Practical Value

The main purpose of the `QuadTree` is scalability and performance. In any game with either a substantial number of objects, or a substantially large game world, you would want to 'divide and conquer', so to speak.

The two main usage areas are collision detection and culling of invisible objects during drawing of the scene, as explained in section 5.3. The price to pay for this performance improvement is only a slight increase in memory usage. A fair price to pay, especially considering that Android devices so have 'more' memory capacity than they have computational capacity [14].

#### Academic Value

Client code will need to use the Visitor pattern (appendix B.1) to iterate the objects in the tree. Consider this scenario: a student wants to draw the objects that overlap a certain rectangular area of the game world (the current area visible by the 'camera'). A member function in the `QuadTree` class accepts one rectangle, and one `SpatialVisitor`. The `QuadTree` then passes the rectangle down to the `QuadTreeNodes`, which determine which objects intersects with the area and notifies the visitor object. The visitor may then choose what to do with the `Spatial` objects (for instance draw them).

### 8.4.2 Collision Detection

Sheep will revolve heavily around collision detection, as implementing even simple algorithms can take lots of precious time, and is hard to get working correctly. In an effort to create an extensible system, multiple approaches were considered, which will be presented in the following section.

#### Architectural Issues

Designing an efficient, extensible collision detection class hierarchy which follows good object-oriented design principles proved harder than it first might seem. Here, some bad attempts will be presented before the final solution is revealed.

Simply adding a abstract methods to a `Shape` class and deriving specific objects from this might be the first thing entering the mind of an object-oriented programmer, but this approach does not provide extensibility *nor* is it able to follow good object-oriented principles (figure 8.5. Each specific class (`Rectangle`, `Circle`) needs to know details about every other specific class in the hierarchy. If a rectangle is to test for collision with a circle, it needs to know more than that the circle is a `Shape`; it needs to know its radius, i.e. data specific to that particular subclass. This could be solved by adding each specific shape in the abstract class (figure 8.6, but it does not solve the extensibility problem. If you wanted to add a custom shape (for instance convex polygon), you would have to modify existing classes, which is unacceptable.

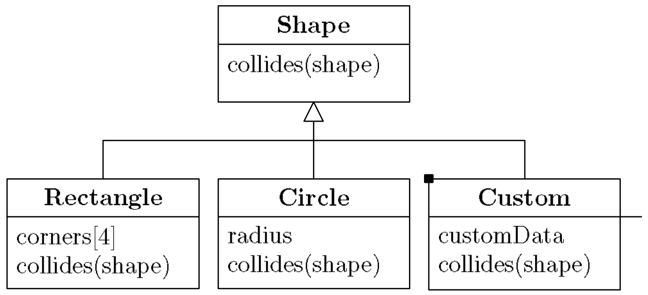

Figure 8.5: A naive attempt at a collision detection hierarchy.

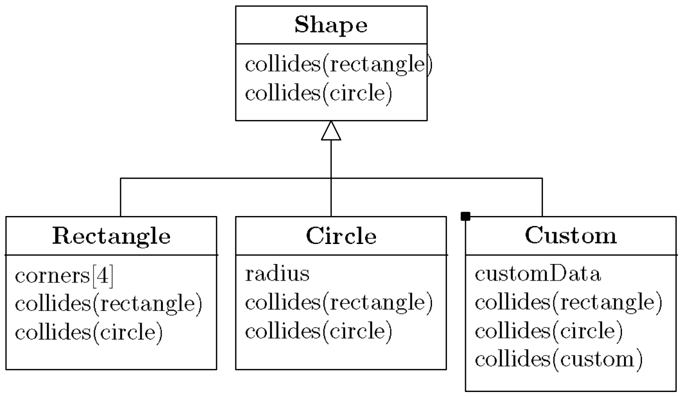

Figure 8.6: Extending the naive collision detection hierarchy.

The problem is rooted in the lack of the *multiple dispatch* paradigm in certain object oriented languages, such as Java and C++ [40]. Each class requires data specific to every other class in the hierarchy; a special algorithm is needed for each pair of specific classes. This makes it hard to properly model this kind of problem using single dispatch programing languages. One proposed solution is the *double dispatch* 'pattern' [41], which is not unlike the visitor pattern. Using double dispatch, we get a looser coupling between some classes, although it does not solve all of our problems. In figure 8.7, the concept is illustrated. When a `Shape` collides with an `Entity`, the method `collide()` is called on the `Shape` object with the `Entity` object as the parameter. The `collide()` method in `Shape` must be overridden in each subclass. The purpose of the overridden method is to call the `collide()` method in the passed `Entity` object with the `this` pointer as a parameter. As we can see from the figure 8.7, this will cause the correct method to be dispatched in the `Entity` object, regardless of which derived class that object might point to. One can therefore add as many subclasses of `Entity` class as needed without modifying the existing classes, but if you want to add more shapes, the `Entity` class itself must be modified. This fact makes this technique infeasible in our case.

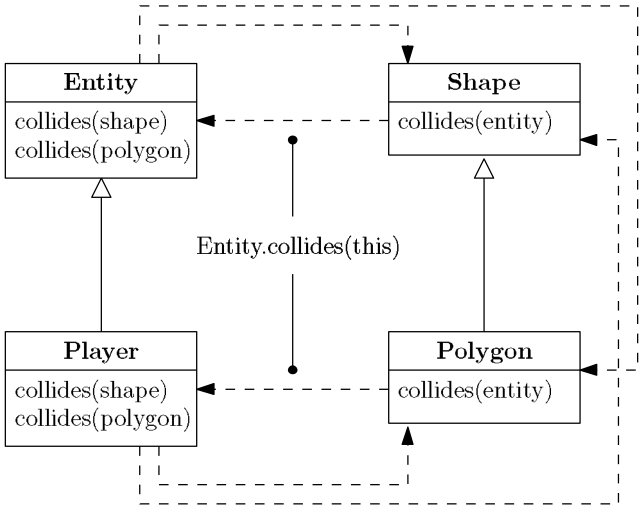

Figure 8.7: The double dispatch pattern [41].

**Final Solution**

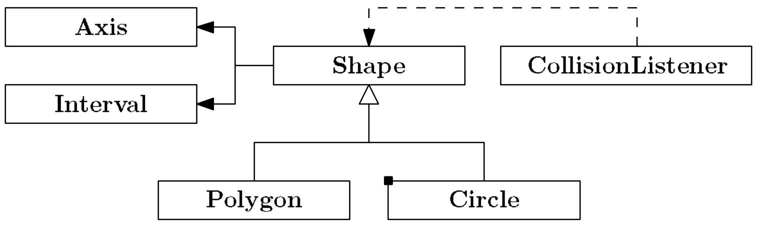

Figure 8.8: The flexible separating axis solution.

The final solution solves the problem by using a more flexible algorithm for the job. As it turns out, it also implies a well-known design patterns for the students to learn.

Basing our class hierarchy on the method of separation axis (see section 5.1.2), we get a system that follows good object oriented design principles and excellent extensibility.

The `Shape` class contains the actual algorithm itself, but allows some parts of it to be overridden by derived classes. The flow is as follows:

- The `Shape` class generates the potential separation axes. The specifics of this method are implemented in a derived class.
- The `Shape` class loops through the potential separating axes and calls a method which projects the shape into the axis. This method is implemented in the derived classes, and it returns an `Interval`.
- Using the pairs of `Interval` objects (one pair for each axis), the `Shape` can determine whether the shapes are colliding or not, and the minimum displacement vector needed to move the shapes out of the collision state.

This approach allows for excellent modifiability, although it does require some understanding of the algorithm. Subclasses of the `Shape` only need to implement two methods: one that 'submits' the potential separating axes for that shape, and one that projects the shape into an axis.

#### Practical Value

The practical value of a collision detection system is substantial. Collision detection were used in most of the student projects in 2008, and getting the details of such systems to work right can be incredibly time consuming.

There are definitely some interesting observations to be made with respect to the limitations of single dispatch programming languages, but even when providing the students with a full collision detection system, they lose very little academically compared to what they gain in productivity.

#### Academic Value

Two patterns will be visible from the perspective of the student; the Template pattern (see appendix B.4), and the Observer pattern (see appendix B.2).

The Template pattern is evident in the `Shape` class, where the overall algorithm is fixed, but some subparts are modifiable by derived classes. However, extending the `Shape` class is not required under normal circumstances, and this option is mostly provided as an performance optimization option.

The Observer pattern will be used for custom collision responses. As an example, perhaps a player should lose health when it is hit by another object. A `LoseHealthListener` could then be attached to listen for collision events occurring to the player object.

## 8.5 sheep.gui

### 8.5.1 Hierarchical Message Passing

The Sheep framework provides a graphical user interface system, though it does not provide many prebuilt `Widget` classes. The system can be used to create complex windowed menus, or just simple buttons.

The system is built around the `Widget` superclass, which represents any entity, whether it is possible to interact with it or not. For instance, both the clickable `TextButton` class and an invisible `Container` classes are subclasses of `Widget`.

`Container` may hold references to other `Widgets`, and therefore also other `Containers`. This forms a hierarchy of widgets where events are passed in a specific manner, as we will explore more deeply in section 9.1.7.

Classes implementing `WidgetListener` may subscribe to `WidgetActions` issued by specific widgets. The actual event the `WidgetAction` represents is chosen by the specific widget issuing the event. A `WigetAction` can be issued when a button is clicked, for instance, or when a text area is scrolled.

#### Practical Value

Implementing graphical user interface systems is interesting, and perhaps a good exercise for the students, but implementing proper ones may take time. Unless the game in question require many different menus and buttons, it may distract from the main problem. In the previously analyzed student projects, most use if widgets were very simple. A few buttons were present to allow the user to start and quit the game. The `TextButton` class can be used to fill the demand for quick and simple menus. At least one group

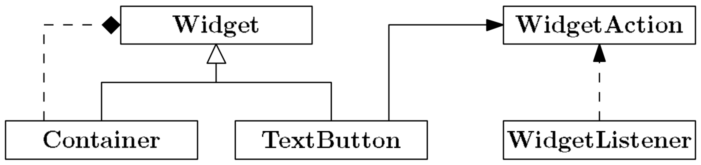

Figure 8.9: Classes in the graphical user interface.

created a role playing game, where graphical user interface elements played an important role. In this case, the system is an integral part of the project itself, and the Sheep framework should not attempt to provide prebuilt widgets. Providing a complete set of widgets which suit all projects is near impossible. Therefore, the graphical user interface subsystem focuses on the system and its extensibility.

### Academic Value

The Observer pattern (appendix B.2) is used here to listen for events generated by the widgets. The Chain of command (appendix B.6) is used to control how input events are passed through the widget hierarchy.

For many patterns, it is clear how an entire pattern works just by implementing parts of the pattern. For instance, even a novice programmer can imagine how an observable object works, even if he or she just implements an interface to observe that object. It should be noted that in the case of this hierarchical chain of command pattern, it may be more unclear how everything ties together. As a programmer, you must make sure that your custom widget responds according to rules which may not be apparent (and less enforced) by the class structure itself. We must therefore rely on external documentation, code comments, or examples in thise case.