# Asteroids

TBD

Software Architecture Document

## 

By: Eric Guzman, Mike Peralta and Alessandro Quezada

# **Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Date** | **Description** | **Author** |
| Elaboration Draft | Oct 15, 2018 | First Draft. | Eric Guzman,  Mike Peralta, Alessandro Quezada |
| Elaboration 2 | Dec 11, 2018 | Added Polymorphism and protected variation GRASP patterns. | ALL |

# **Table of Contents**

[**Asteroids**](#_4ubyul24hmr9) **0**

[**Revision History**](#_767rgzwi8zhr) **1**

[**Table of Contents**](#_tgdmcmkx66iz) **1**

[**1. Introduction**](#_ihhmuu1b138b) **2**

[1.1 Purpose](#_21h3fqn8x3y3) 2

[1.2 Scope](#_t5awe4h83gq2) 2

[1.3 Definitions, Acronyms, and Abbreviations](#_4pwp6euxymv2) 2

[1.4 Overview](#_4f8ntsii282j) 2

[**2. Architectural Representation**](#_d8ir34hr7aap) **3**

[**3. Architectural Decisions**](#_n4ncgxbpy4l8) **3**

[3.1 Low Coupling/High Cohesion](#_obz2y0z9fzr5) 3

[3.2 Creator](#_vcq7rerxqgso) 4

[3.3 Information Expert](#_bhkklltz9sth) 4

[3.4 Controller](#_coubkqgu3cvv) 4

[**4. Logical View**](#_bnogy2d3rvlm) **5**

[4.1 Package Diagram](#_iuzqmu4pul7p) 6

[4.2 Class Diagrams](#_6hy7s0sbfnww) 7

[**5. Polymorphism GRASP**](#_uqkblcolaof4) **10**

[**6. Protected Variations GRASP**](#_3tkctkjwhg6v) **13**

[**7. Domain Model View**](#_15awqhj381o9) **14**

[**8. Design Model View**](#_fhmgaho1riqy) **14**

# **1. Introduction**

This is an overview of the *Software Architecture Document* for Asteroids.

## 1.1 Purpose

This document outlines the architectural decisions made to create the Asteroids video game. Asteroids aims to provide a fun and competitive experience for people of all ages. It is a throwback to the classic arcade Asteroids, but with a modern twist.

This document will explain the architectural decisions made in building Asteroids, and show the architecture in different views.

## 1.2 Scope

The scope of this Software Architecture Document is to outline the architecture of Asteroids built by Team #19.

## 1.3 Definitions, Acronyms, and Abbreviations

For the definitions, see the *Glossary* artifact.

## 1.4 Overview

This Software Architecture Document consists of 6 sections:

* Section 1 is an introduction to the software architecture document and describes how the layout of the document.
* Section 2 is the architectural representation, which describes how the architecture of Asteroids will be shown in different views.
* Section 3 is the architectural decisions. This section explains the decisions made which have affected the architecture.
* Section 4 is the Logical View. This section contains the package diagram and the class diagram.
* Section 5 is the Domain Model, which is included in a separate artifact.
* Section 6 is the Design Model, which is also included in a separate artifact.

# **2. Architectural Representation**

This document details the architecture of Asteroids in 4 different views. The first view is architectural decisions. This view will describe the options the team considered when building the architecture and explain the selection made. It will also identify parts of the architecture such as the controller, creator, and information expert. The second view is the Logical View. This view contains the package diagram and the class diagram.

The last two views are the Domain Model View and the Design Model View. The Domain Model View is a diagram that shows the system’s concepts, and the relationships between those concepts. The Design Model View contains the software sequence diagrams and the software class diagrams.

# **3. Architectural Decisions**

The following subsections describe the decisions that were made according to some of the GRASP patterns. They explain why the decisions were made and reference them to the design model diagrams, which can be found in the *Design Model* artifacts. For more specific information about these GRASP pattern implementations, see the *Implementation Model* artifact.

## 3.1 Low Coupling/High Cohesion

We attempted to have low coupling by avoiding functions that had to go through multiple classes to be carried out. For example, we found that having a Server class containing the ServerDaemon and the ServerTextUI classes had high coupling. When an admin would ask questions to the server class form the ServerTextUI, the Server class would then have to ask the ServerDaemon class and then return the answer received to the ServerTextUI. We decided to make the ServerTextUI a controller which contains a ServerDaemon instance, eliminating the coupling between the Server and ServerDaemon classes.

Interfaces were created anywhere a class had to be used by another class that was in another package. Coupling to an interface is better than coupling to the class itself. A full list of interfaces can be seen in the *Implementation Model*.

To achieve high cohesion, we followed the principle that an object’s duties should be closely related as possible. This can be seen in Figure 1.2 in the *Design Model View*. Functions within the Store package are broken down into many classes, such as the Store class whose only responsibility is querying/modifying things related directly to the store. The PaymentHelper class communicates with the server to handle payments.

## 3.2 Creator

We decided to have a “Factory” object in each of the main domain packages (Game, Net, Server, Store), and for the UI package. Each of these factories would contain all the information needed to create objects within their packages. At first, these were the only “Factory” objects. We found this method had high coupling, because the information was spread out.

The solution we came up with was to also make a package called “Factory”, which will contain the information needed to create the other package Factory objects. This main Factory object is simply named, “Factory” while the other package factories include the name of their package, such as “GameFactory”.

This method can be seen in all of the class diagrams in the *Design Model View*. Each package class diagram contains the main Factory object which then creates the package Factory object.

## 3.3 Information Expert

We assigned responsibilities to objects based on their knowledge of the task assigned to them. For example, the main factory class is the only one with the knowledge to create the other package factories. Another example is the game object. The game object is responsible for tasks related to the game itself, such as updating actors, keeping, manipulating, and returning the game state information. It can carry out these tasks, because it has the information it needs.

For more specific information on the information experts in Asteroid, see the Gasp Implementation - Information Expert section of the *Implementation Model* artifact.

## 3.4 Controller

The two main controllers that were chosen are in the UI layer. These are the ClientTextUI package and the ServerTextUI package. This follows the facade pattern, because we felt that the entire program depends entirely on user input. The ClientTextUI package takes in the user input and then drives the programs operations by initiating a game, browsing the store, or accessing any of the other features of the game.

Initially, we wanted a “Game” class to be the controller, but we felt that would not be appropriate, because Asteroids has other features that are not related to a game, such as the store and the admin interface. Because Asteroids is user driven, we found that everything would go through the ClientTextUI, so we decided to make that class the controller, and the Game class would handle just the gameplay features.

You can see this in the *Design Model View* in figures 1.1, 1.2, and 1.5. In these diagrams, ClientTextUI is present, and connects to the game, store, and UI packages, which are the main features of Asteroids.

# **4. Logical View**

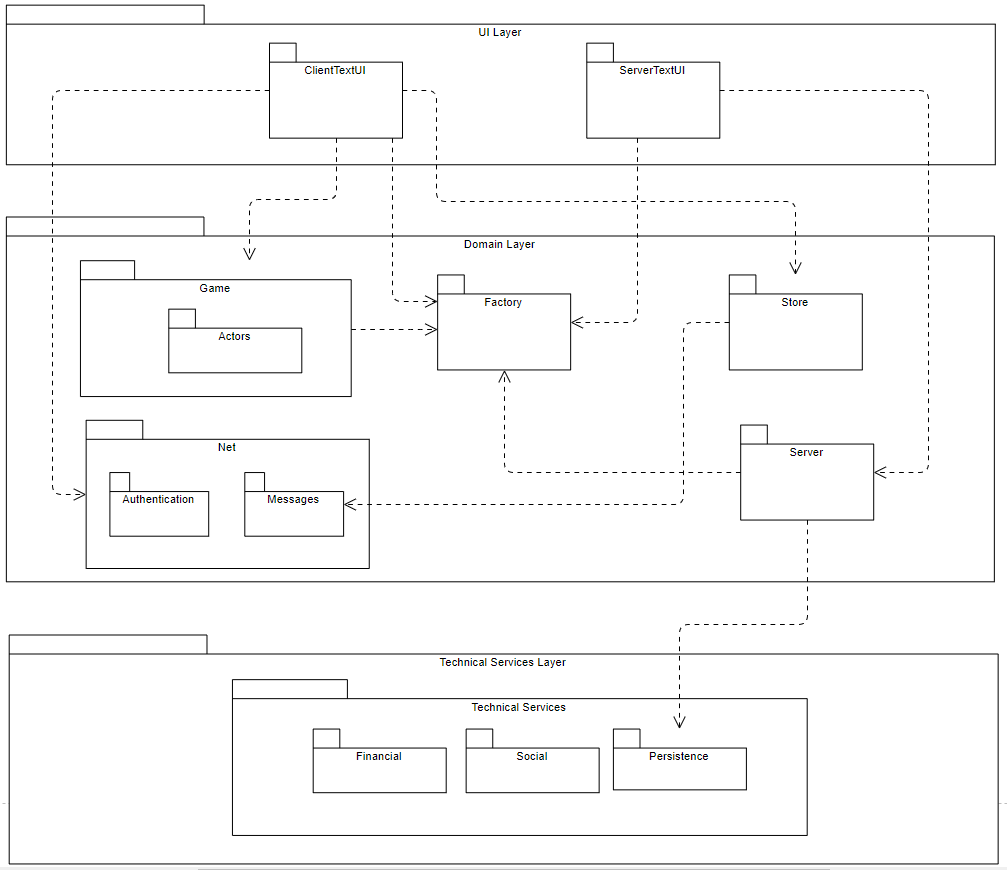
For the logical view we are using the layer architecture that is composed of classes, packages, or subsystems that has cohesive responsibility for a major aspect of the system. The layers are organized such that “higher” layers (such as the UI layer) call upon services of “lower” layers, but not vice versa. The layers are ordered as:

* User Interface Layer
* Domain Layer
* Technical Service Layer

Sections 4.1 and 4.2 are on the following pages.

## 4.1 Package Diagram

Figure 4.1.1 Package Diagram



## 4.2 Class Diagrams

Figure 4.2.1 Factory Package Interfaces

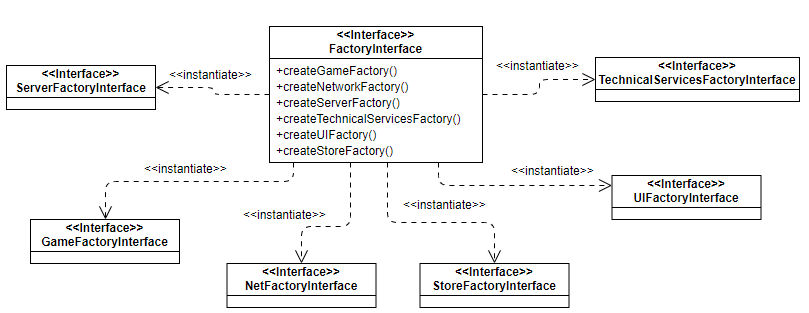


Figure 4.2.2 Game Package Interfaces

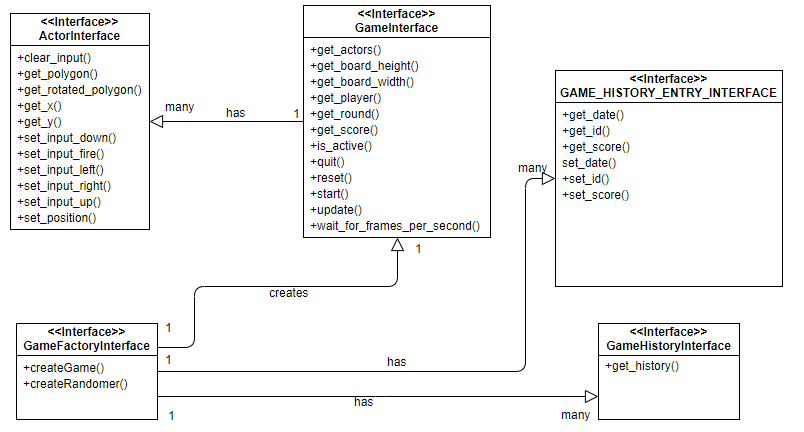
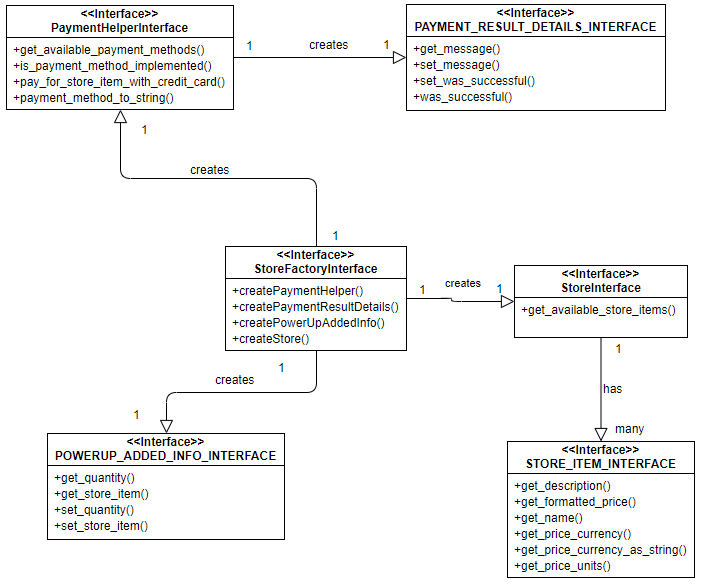


Figure 4.2.3 Store Package Interfaces

Figure 4.2.4 Net Package Interfaces

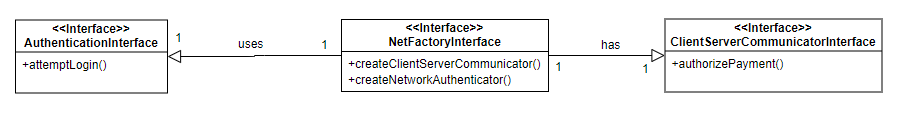


Figure 4.2.5 Server Package Interfaces

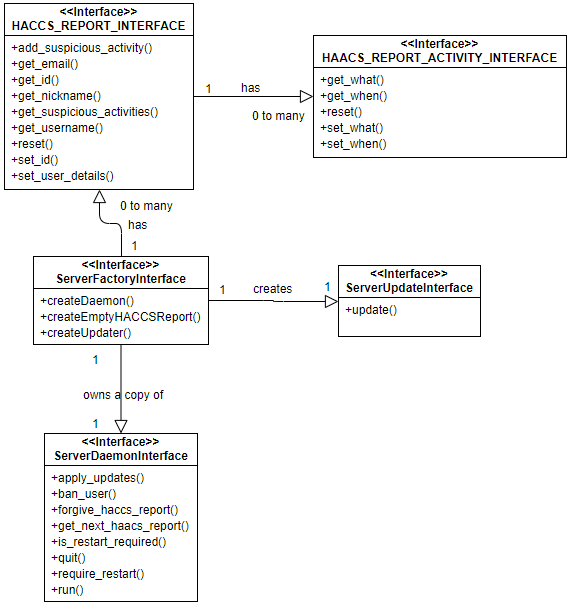
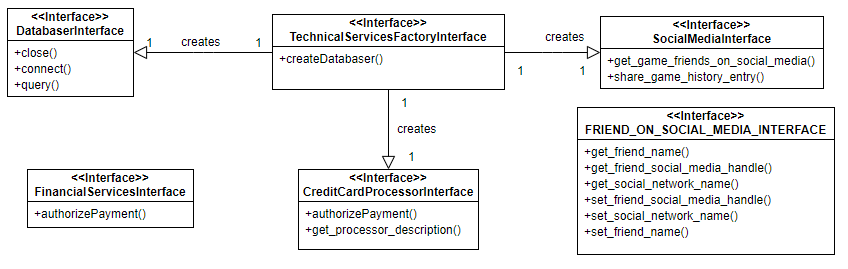
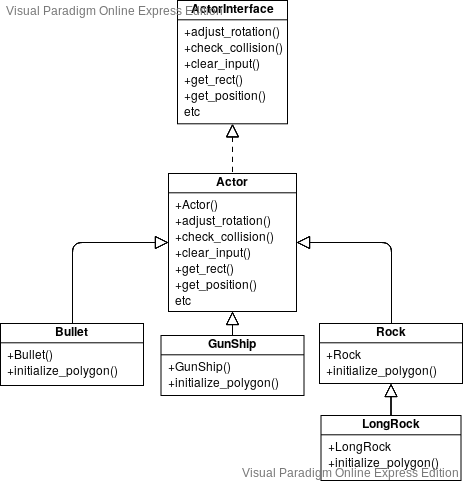


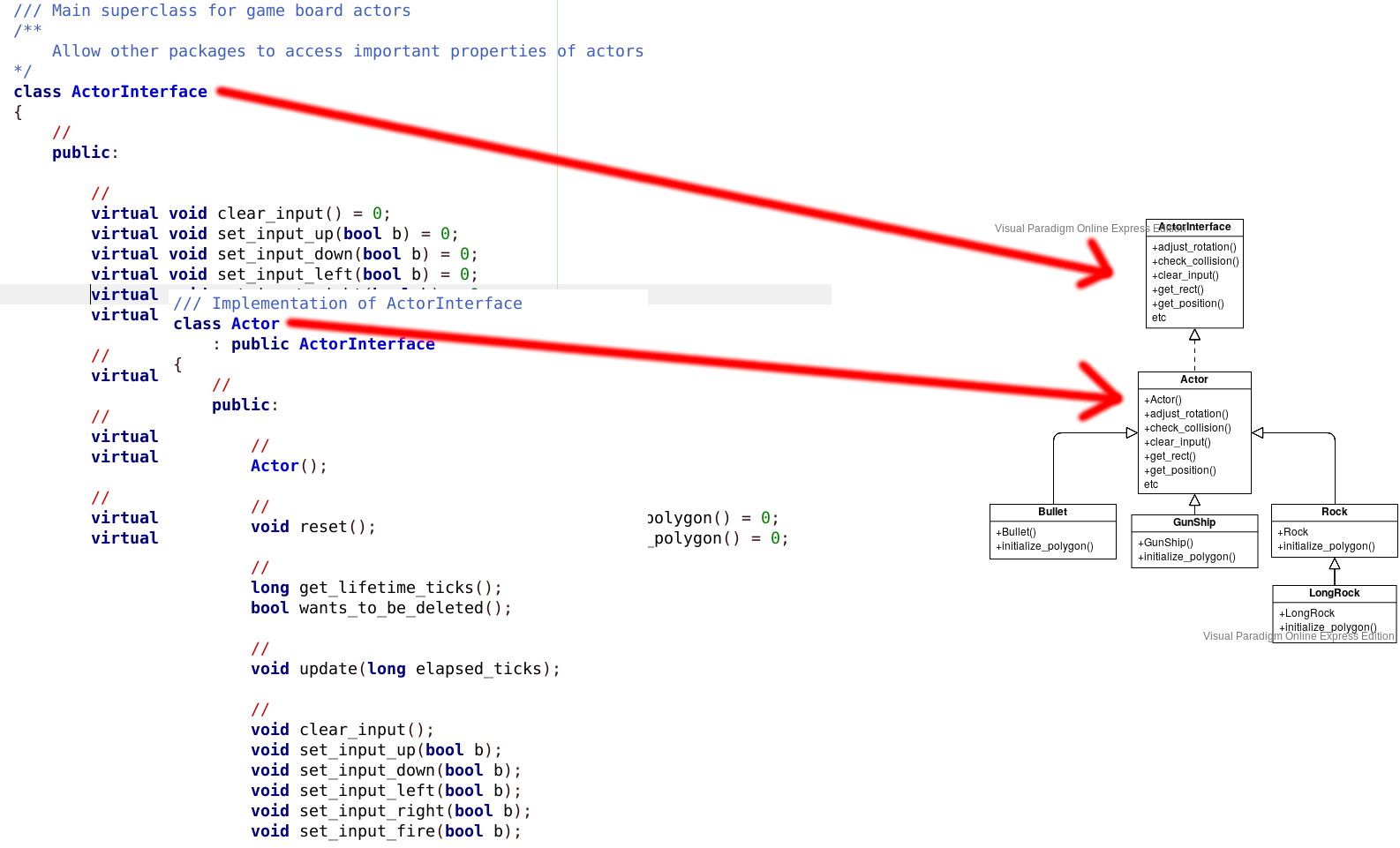
Figure 4.2.6 Technical Services Package Interfaces

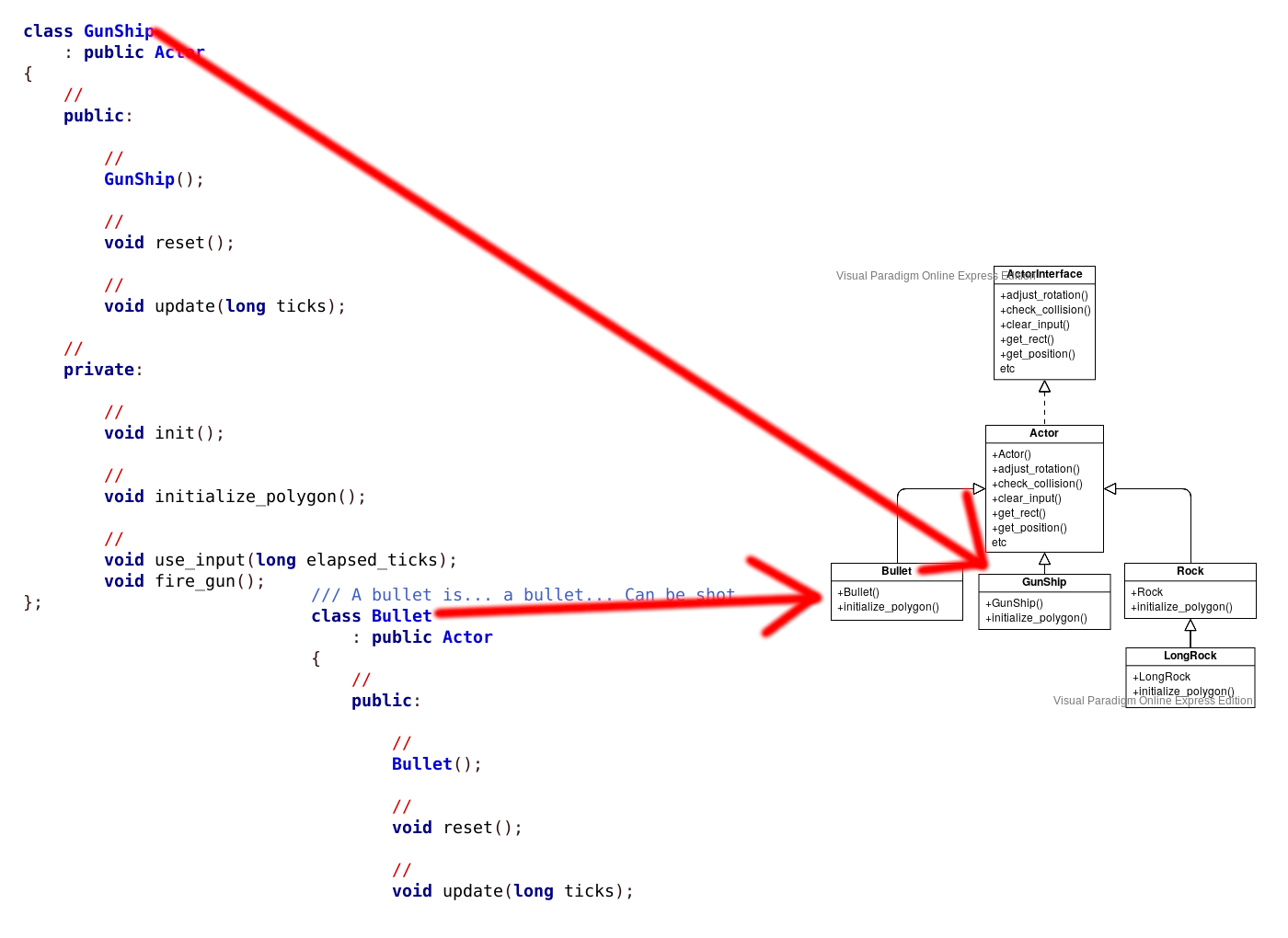
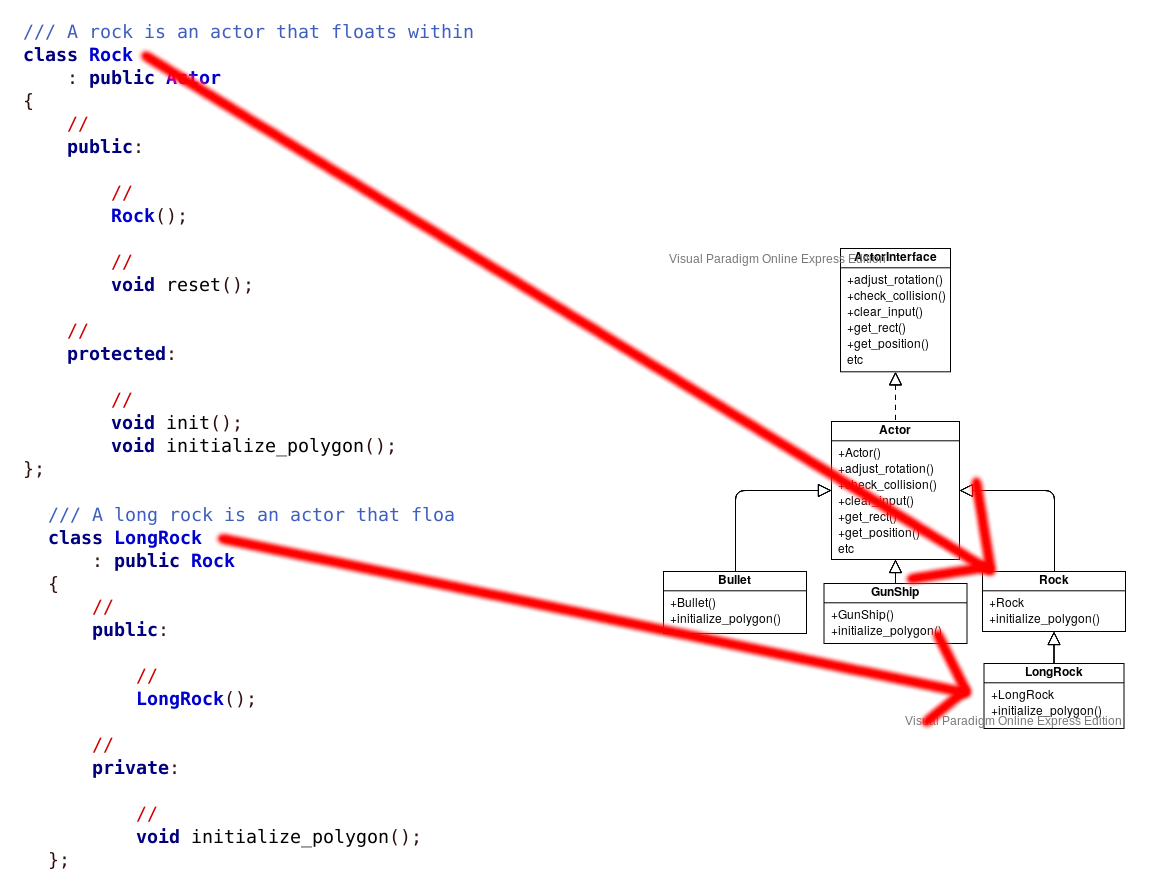


# **5. Polymorphism GRASP**

1. We start with only one type of rock: the “Rock” class, which inherits from Actor
2. We then subclass from Rock to LongRock
3. The game then includes two types of rocks

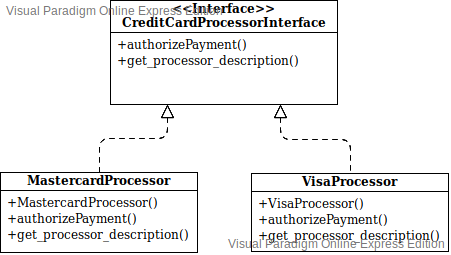


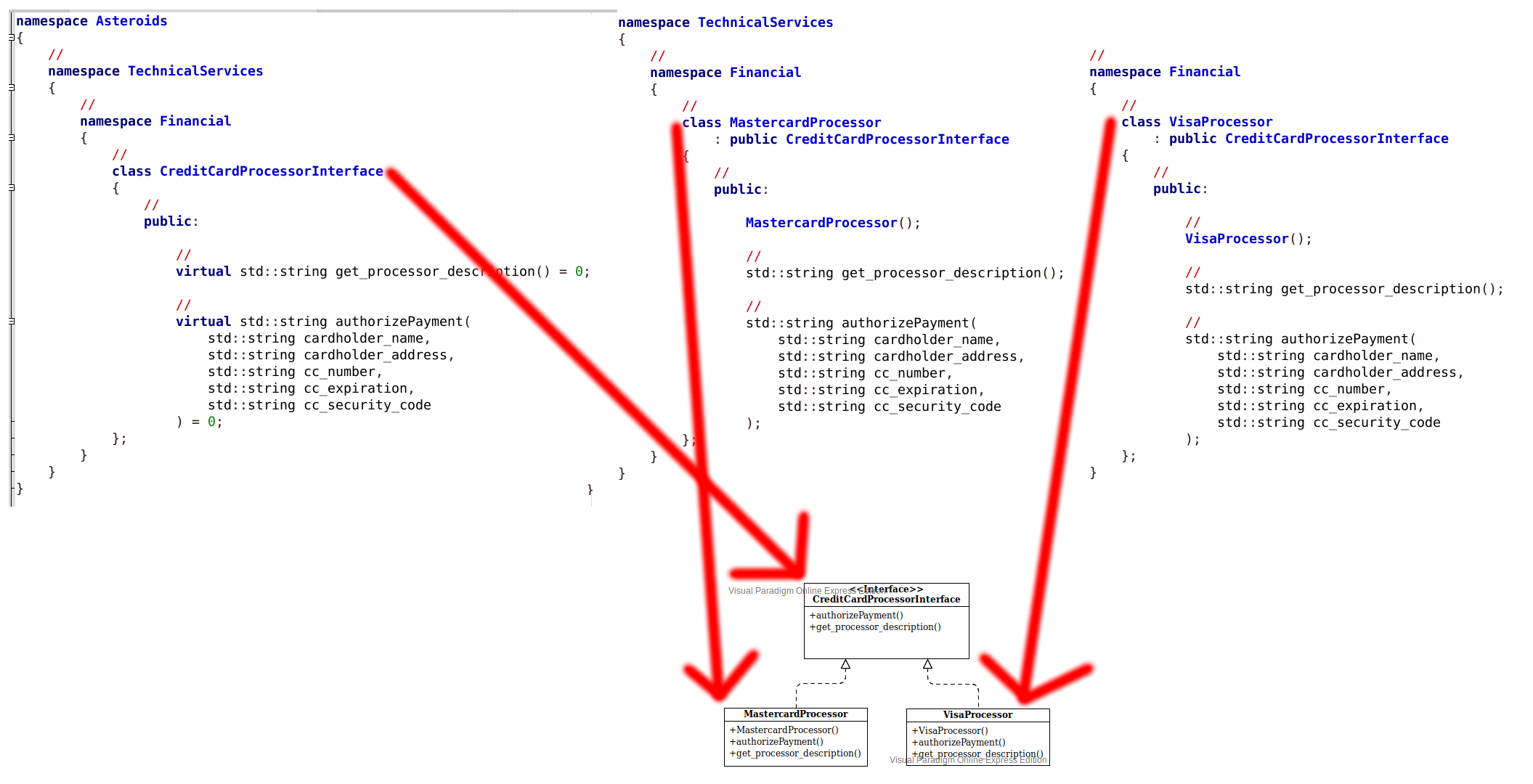




# **6. Protected Variations GRASP**

* Our credit card processing interface is CreditCardProcessorInterface
* It allows us to change which payment processor we use, to reduce risk
* By default, our application’s implementation is MastercardProcessor
* With the environment variable USE\_VISA set, the application instead implements with VisaProcessor





# **7. Domain Model View**

The Domain Model View is a set of diagrams that capture the system’s concepts and their relationships. To see the Domain Model diagrams, see the *Domain Model* artifact.

# **8. Design Model View**

The Design Model View is a set software sequence diagrams and software class diagrams that serves as an abstraction of the source code. To see the Design Model diagrams, see the *Design Model* artifact.