**Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Date** | **Description** | **Author** |
| Elaboration Draft | 10/31/2018 | First Draft. | All |
| Elaboration 2 | 12/14//2018 | Added the poly things like long rock and also incorporated new classes and packages  **New Packages**   * TechnicalServices::Social   **New Classes**   * Domain::Game::GameHistoryInterface * Domain::Game::GameHistory * Domain::Game::GAME\_HISTORY\_ENTRY\_INTERFACE * Domain::Game::GAME\_HISTORY\_ENTRY * TechnicalServices::Social::SocialMedia * TechnicalServices::Social::FRIEND\_ON\_SOCIAL\_MEDIA * TechnicalServices::Social::FRIEND\_ON\_SOCIAL\_MEDIA\_INTERFACE | All |

Source Code to Design Cross Reference

# Source / Header Implementation of classes

* FactoryInterface → Asteroids/Factory/FactoryInterface.h
* Factory → Asteroids/Factory/Factory.h / cpp
* GameInterface → Asteroids/Game/GameInterface.h
* Game → Asteroids/Game/Game.h / cpp
* GameFactoryInterface → Asteroids/Game/GameFactoryInterface.h
* GameFactory → Asteroids/Game/GameFactory.h / cpp
* Randomer → Asteroids/Game/Randomer.h / cpp
* ActorInterface → Asteroids/Game/Actors/ActorInterface.h
* Actor → Asteroids/Game/Actors/Actor.h / cpp
* Alien → [not implemented in artifacts nor code yet / just an empty placeholder file]
* Bullet → Asteroids/Game/Actors/Bullet.h / cpp
* GunShip → Asteroids/Game/Actors/GunShip.h / cpp
* Rect → Asteroids/Game/Actors/Rect.h / cpp
* Rock → Asteroids/Game/Actors/Rock.h / cpp
* Vector → Asteroids/Game/Actors/Vector.h / cpp
* ClientServerCommunicatorInterface → Asteroids/Net/ClientServerCommunicatorInterface.h
* ClientServerCommunicator → Asteroids/Net/ClientServerCommunicator.h / cpp
* NetFactoryInterface → Asteroids/Net/NetFactoryInterface.h
* NetFactory → Asteroids/Net/NetFactory.h / cpp
* AuthenticationInterface → Asteroids/Net/Authentication/AuthenticationInterface.h
* Authenticator → Asteroids/Net/Authentication/Authenticator.h / cpp
* AUTH\_ATTEMPT → Asteroids/Net/Messages/AUTH\_ATTEMPT.h / cpp
* AUTH\_STATE → Asteroids/Net/Messages/AUTH\_STATE.h / cpp
* HACCS\_REPORT\_INTERFACE → Asteroids/Server/HACCS\_REPORT\_INTERFACE.h / cpp
* HACCS\_REPORT → Asteroids/Server/HACCS\_REPORT.h / cpp
* HACCS\_REPORT\_ACTIVITY → Asteroids/Server/HACCS\_REPORT\_ACTIVITY.h / cpp
* ServerDaemonInterface → Asteroids/Server/ServerDaemonInterface.h
* ServerDaemon → Asteroids/Server/ServerDaemon.h / cpp
* ServerFactoryInterface → Asteroids/Server/ServerFactoryInterface.h
* ServerFactory → Asteroids/Server/ServerFactory.h / cpp
* ServerUpdater → Asteroids/Server/ServerUpdater.h / cpp
* PaymentHelperInterface → Asteroids/Store/PaymentHelperInterface.h
* PaymentHelper → Asteroids/Store/PaymentHelper.h / cpp
* PAYMENT\_RESULTS\_DETAILS\_INTERFACE → Asteroids/Store/PAYMENT\_RESULTS\_DETAILS\_INTERFACE.h
* PAYMENT\_RESULTS\_DETAILS → Asteroids/Store/PAYMENT\_RESULTS\_DETAILS.h / cpp
* POWERUP\_ADDED\_INFO → Asteroids/Store/POWERUP\_ADDED\_INFO.h / cpp
* POWERUP\_ADDED\_INFO\_INTERFACE → Asteroids/Store/POWERUP\_ADDED\_INFO\_INTERFACE.h
* StoreInterface → Asteroids/Store/StoreInterface.h
* Store → Asteroids/Store/Store.h / cpp
* StoreFactoryInterface → Asteroids/Store/StoreFactoryInterface.h
* StoreFactory → Asteroids/Store/StoreFactory.h / cpp
* STORE\_ITEM\_INTERFACE → Asteroids/Store/STORE\_ITEM\_INTERFACE.h
* STORE\_ITEM → Asteroids/Store/STORE\_ITEM.h / cpp
* TechnicalServicesFactoryInterface → Asteroids/TechnicalServices/TechnicalServicesFactoryInterface.h
* TechnicalServicesFactory → Asteroids/TechnicalServices/TechnicalServicesFactory.h / cpp
* FinancialServicesInterface → Asteroids/TechnicalServices/FinancialServices/FinancialServicesInterface.h
* FinancialServices → Asteroids/TechnicalServices/FinancialServices/FinancialServices.h / cpp
* GameHistoryInterface → Asteroids/Game/GameHistoryInterface.h
* GameHistory→ Asteroids/Game/GameHistory.cpp
* GAME\_HISTORY\_ENTRY\_INTERFACE→ Asteroids/Game/AME\_HISTORY\_ENTRY\_INTERFACE.h
* GAME\_HISTORY\_ENTRY→ Asteroids/Game/GAME\_HISTORY\_ENTRY.cpp
* SocialMedia → Asteroids/TechnicalServices/SocialMedia.h
* FRIEND\_ON\_SOCIAL\_MEDIA → Asteroids/TechnicalServices/FRIEND\_ON\_SOCIAL\_MEDIA.cpp
* FRIEND\_ON\_SOCIAL\_MEDIA\_INTERFACE → Asteroids/TechnicalServices/FRIEND\_ON\_SOCIAL\_MEDIA\_INTERFACE.h
* DatabaserInterface → Asteroids/TechnicalServices/Persistence/DatabaserInterface.h
* Databaser → Asteroids/TechnicalServices/Persistence/Databaser.h / cpp
* UIFactoryInterface → Asteroids/UI/UIFactoryInterface.h
* UIFactory → Asteroids/UI/UIFactory.h / cpp
* ClientTextUIInterface → Asteroids/UI/ClientTextUI/ClientTextUIInterface.h
* ClientTextUI → Asteroids/UI/ClientTextUI/ClientTextUI.h / cpp
* DisplayGrid → Asteroids/UI/ClientTextUI/DisplayGrid.h / cpp
* Input → Asteroids/UI/ClientTextUI/Input.h / cpp
* ServerTextUIInterface → Asteroids/UI/ServerTextUI/ServerTextUIInterface.h
* ServerTextUI → Asteroids/UI/ServerTextUI/ServerTextUI.h / cpp
* Reference to main(), in the server executable →Asteroids/main-server.cpp
* Reference to main(), in the client executable → Asteroids/main-console-client.cpp

# Grasp Implementation - Creator

We chose to look ahead and assume that many of the higher level objects in each package would eventually need much information to create, and thus we assigned a dedicated “Factory” object for each major package (Game, Net, Server, Store, UI), with the intent that each “Package Factory” object would eventually contain all needed information to create these objects. We then made a dedicated package called “Factory”, which itself will eventually contain all needed information to create the *other* package Factory objects. This way, code instantiating objects inside Asteroids need only worry about instantiating the main Factory object. This had the added benefit of lowering coupling somewhat more.

The main factory class files are:

* Asteroids/Factory/FactoryInterface.h
* Asteroids/Factory/Factory.h
* Asteroids/Factory/Factory.cpp

The other factory classes follow a similar name pattern:

* Game Package
  + Asteroids/Game/GameFactoryInterface.h
  + Asteroids/Game/GameFactory.h
  + Asteroids/Game/GameFactory.cpp
* Net Package
  + Asteroids/Net/NetFactoryInterface.h
  + Asteroids/Net/NetFactory.h
  + Asteroids/Net/NetFactory.cpp
* Server Package
  + Asteroids/Server/ServerFactoryInterface.h
  + Asteroids/Server/ServerFactory.h
  + Asteroids/Server/ServerFactory.cpp
* Store Package
  + Asteroids/Store/StoreFactoryInterface.h
  + Asteroids/Store/StoreFactory.h
  + Asteroids/Store/StoreFactory.cpp
* UI Package
  + Asteroids/UI/UIFactoryInterface.h
  + Asteroids/UI/UIFactory.h
  + Asteroids/UI/UIFactory.cpp
* Social Package
  + Asteroids/TechServices/SocialMedia.h
  + Asteroids/TechServices/SocialMedia.cpp

Additionally, there were certain objects that we reasonably guessed were only ever going to be created+used by one other class, and so we delegated responsibility for their creation to their direct user:

* Game (Asteroids/Game/Game.cpp) creates:
  + GunShip (Asteroids/Game/Actors/GunShip.h/cpp
  + Rock (Asteroids/Game/Actors/Rock.h/cpp
* The GunShip is the only one so far that shoots bullets, but Alien will also eventually fire bullets, so we use their shared superclass to create them:
  + Bullet (Asteroids/Game/Actors/Bullet.h/cpp)
* The Actor superclass also creates:
  + Asteroids/Game/Actors/Rect.h/cpp
  + Asteroids/Game/Actors/Vector.h/cpp

We allowed the GameFactory to create the Randomer object (Asteroids/Game/Randomer.h/cpp) because we felt it might be useful to other classes later.

# Grasp Implementation - Information Expert

We attempted to skillfully assign responsibilities to objects based on their knowledge of the task assigned to them:

* **Factory**: The main factory class is the only one with the (future) knowledge to create the other package factories, so we gave it that responsibility. This crosses over with the Creator pattern
* **Game:** The Game object is responsible for carrying out responsibilities related to the overall game: updating actors, keeping, manipulating, and returning game state information. It can do this because it has the information needed: The actual Actor instances, the count of points, the current level, etc. You can see this in *Asteroids/Game/Game.cpp::update()*
* **Game Actor**: The Actor class has the information needed to represent a game board piece (vector, rect, and other custom properties), so it was given the job of maintaining the general state of those things (ie applying physics, handling navigation input, etc). You can see this in *Asteroids/Game/Actors/Actor.cpp::Actor::update()*
* **Game GunShip**: The GunShip is responsible for “responding to user input”, because it is the only class given knowledge (via function implementation) of how exactly to manipulate its actor/physics properties in response to user input. You can see this in *Asteroids/Game/Actors/GunShip.cpp::use\_input()*
* **Net ClientServerCommunicator:** We placed the responsibility of communicating with the Asteroids server (from any end user client application) squarely with the ClientServerCommunicator.cpp/h file/class. It will eventually contain all the information needed to reach and communicate with the server. At the moment it only returns dummy data when trying to authorize a credit card, which you can see at *Asteroids/Net/ClientServerCommunicator.cpp::authorizePayment()*
* **Net/Authentication Authenticator**: We similarly placed responsibility for authenticating with the server, with the Authenticator. We do not initially consider this to be exactly the same thing as “knowing how to communicate with the server”, because it is a more specific job. We may however, eventually merge the Authenticator into the ClientServerCommunicator, or perhaps have the Authenticator ask the ClientServerCommunicator to perform lower level socket/communication functions on its behalf, once/if we get to the point where the Authenticator needs to perform a real query instead of returning dummy data. You can see this in *Asteroids/Net/Authentication/Authenticator.cpp::attemptLogin()* and *Asteroids/Net/Authentication/Authenticator.cpp::sendAuthToServer()*
* **ServerDaemon:** We made the ServerDaemon class responsible for maintaining the state of the Server, because that’s the main job of a daemon. When it receives network requests, and requests from the ServerTextUI controller, it will change and hold state information directly within itself (or at least maintain it via direct manipulation of a later-implemented data store object). It will maintain things like the currently online users, uptime, permissions, reports, etc. You can see an example of it gathering report data at *Asteroids/Server/ServerDaemon.cpp::get\_next\_haccs\_report()* and an example of report forgiveness in *forgive\_haccs\_report()*, or wrath at *ban\_user()*
* **ServerUpdater:** The ServerUpdater class will presumably eventually contain knowledge needed to update the server executable. For example, we might give it information on where to find updated files / configs, where to place them temporarily, and how to actually change to the new executable. The magic of this class all lies within *Asteroids/Server/ServerUpdater.cpp::update()*
* **Store**: The store object is given the responsibility of querying/returning store items, because it will be the only one that directly knows about those items. It can later be given the responsibility of querying store properties (such as latest advertisements, current prices, store logic, etc) because it makes sense that the Store class would know about this. You can see this in *Asteroids/Store/Store.cpp::get\_available\_store\_items()*
* **Store PaymentHelper:** This class will be given everything needed to perform the job of “paying for things”. Right now that means powerup items from the Store. It will know about available payment methods and their implementation status, thus can answer queries about those things. It will also perform the job of actually making payments and thus will need to know exactly how to do that. It will eventually rely on the Net/ServerClientCommunicator class when it’s ready to return real data (currently returns dummy data). You can see it attempting to make a payment at *Asteroids/Store/PaymentHelper.cpp::pay\_for\_store\_item\_with\_credit\_card()*. It also generates a list of payment methods in *get\_available\_payment\_methods()*

# Grasp Implementation - Low Coupling

We attempted to keep low coupling by avoiding too many “go between” classes, or classes that simply played middleman or wrapper to information in a second class, where we were able to. For example, we initially had a class named Server, which contained both the ServerDaemon and ServerTextUI class. We found that whenever we wanted to ask questions of the Server class, we actually ended up simply having the Server class ask the ServerDaemon class and return the answer, creating too-high coupling between Server and ServerDaemon. Rather this might have just been unneeded coupling, anyway. So we then realized the ServerTextUI class should be a controller, and directly contain a ServerDaemon instance, so we could do away with the normal Server class altogether, reducing coupling. You can see the ServerTextUI creating an instance of the ServerDaemon in *Asteroids/UI/ServerTextUI/ServerTextUI.cpp::init()*, and utilizing it in response to user input in the method *do\_review\_flagged\_user()*

We also attempted to use Interfaces anywhere we thought any class had the potential to be used by any other class that was either in another package, or by another class that we felt didn’t have enough intimate information of the first class to justify direct coupling. Thus even in cases where there was higher coupling, we felt the use of Interface classes did help reduce coupling from the implementation (ie coupling to the interface is better than coupling to the implementation).

You can see loads of examples of our use of interfaces throughout the code:

* Asteroids/Factory/FactoryInterface.h
* Asteroids/Game/GameFactoryInterface.h
* Asteroids/Game/GameInterface.h
* Asteroids/Game/Actors/ActorInterface.h
* Asteroids/Net/NetFactoryInterface.h
* Asteroids/Net/ClientServerCommunicatorInterface.h
* Asteroids/Net/AuthenticationInterface.h
* Asteroids/Server/ServerFactoryInterface.h
* Asteroids/Server/ServerDaemonInterface.h
* Asteroids/Server/ServerUpdaterInterface.h
* Asteroids/Store/StoreFactoryInterface.h
* Asteroids/Store/StoreInterface.h
* Asteroids/Store/PaymentHelperInterface.h
* Asteroids/UI/UIFactoryInterface.h
* Asteroids/UI/ClientTextUI/ClientTextUIInterface.h
* Asteroids/UI/ServerTextUI/ServerTextUIInterface.h
* Asteroids/Factory/GameHistoryInterface.h
* Asteroids/Factory/GAME\_HISTORY\_ENTRY\_INTERFACE.h
* Asteroids/TechServies/Social/FRIEND\_ON\_SOCIAL\_MEDIA\_INTERFACE.h

# Grasp Implementation - Controller

The two main controllers we designed are in the UI package:

* UI/ServerTextUI/ServerTextUI.h/cpp
* UI/ClientTextUI/ClientTextUI.h/cpp

We decided to make these controllers because we felt the whole of the program’s flow depended entirely on user input. A user decides when a game is started on the client, or when a store is to be queried, or when payments are to be authorized, or when users flagged for review are to be processed, or when the server is to be updated and restarted. Because this application is so user-driven, making the user interfaces into controllers seemed very intuitive.

The ClientTextUI class then owns the objects it needs to perform tasks. For instance, it owns a Game class instance, and Input class instance, and a DisplayGrid class instance, so it can take user input, send it to the Game, update the game, and then query the game for state information in order to send it to the DisplayGrid class and then receive back a final render of the game, suitable for display to the user. You can see this at *Asteroids/UI/ClientTextUI/ClientTextUI.cpp::play\_game(), handle\_game\_input(), and draw\_game()*

The ServerTextUI is much simpler but follows the same logic. It contains an instance of the ServerDaemon (we currently consider the Server as only deployable with a text based user interface for administration purposes, at the moment). It then takes user input and uses that to determine how to manipulate/query the ServerDaemon class, to help a sitting admin accomplish their goals. You can see this in *Asteroids/UI/ServerTextUI/ServerTextUI.cpp::main\_menu() and do\_review\_flagged\_user()*

# Grasp Implementation - High Cohesion

We attempted to achieve high cohesion by adhering to the principle that an object’s duties should be as closely related as possible. We partially achieved this by creating a great many classes for smaller groups of related tasks (and future anticipated tasks).

For example, we decided the Store class is only responsible for querying/modifying things related directly to the store (prices, items, etc). For actual payments, we used a dedicated PaymentHelper class, which further would (eventually) delegate the responsibility of communicating with the server to the ClientServerCommunicator class.

Even with the ClientTextUI controller, we delegated the job of “translating the game state to a format suitable for text display” out to a dedicated class: DisplayGrid, so the ClientTextUI’s could remain more closely responsible only for things related to “servicing user commands”.

The points where we have the Domain layer side of the project are in the following namespaces:

* Factory (Hold the main **Factory**, that generates more specific factories in other packages)
* Game (The **Game** package holds things related to playing the game)
* Store (The **Store** namespace holds things related to the **Store**)
* Server (The **Server** namespace holds stuff related to the **Server** executable)
* Net (The **Server** namespace holds stuff related to the **Server** executable)

The Factory namespace is responsible for generating all of the other factories that create the rest of the game. The only class in this namespace is the Factory class.

The Game namespace is responsible for all the things that have to do with the game side of the project. It houses all the functionality of the game. In this namespace we also have another namespace called Actor.This namespace is responsible for organizing all the **Actor** classes and helpers away from the **Game** logic classes. The classes that are in the actor namespace are Bullet, Gunship, rock, rect, and vector. The classes that are inside the game namespace are the following: GameFactory, game and randomer.

The Store namespace holds all the classes and objects related to the store side of the project. All the classes in this namespace have interfaces. The classes in this namespace are PAYMENT\_RESULT\_DETAILS, PaymentHelper, POWERUP\_ADDED\_INFO, STORE, STORE\_ITEM, StoreFactory.

The Server has all the server executables. Every class in this namespace has an interface as well. The classes in this namespace are HACCS\_REPORT, HACCS\_REPORT\_ACTIVITY, ServerDaemon, ServerFactory, ServerUpdater.

The Net holds all classes that help a client communicate with server. In this namespace we also have subpackages that are Authentication and Messages. These subpackages are responsible for authentication of the user and sending appropriate messages after authentication. Like the other packages all the classes in this package have interfaces. The classes that are in this package are ClientServerCommunicator and NetFactory.

The points where we have the Technical Service layer sIde of the project are in the following namespace:

* TechnicalServices (Holds all things related to technical services layer)

The technical services package holds all the packages that are responsible for the technical services layer of the project. Inside this package/namespace we also have subpackages which are the Financial package and the Persistence package. The Financial package helps with external finance services. In this package we have one class called FinincialServies and its interface. The Persistence package is responsible for everything related to the database, that is managing and manipulating the database. Inside this package we have a class called Databaser and its interface class.