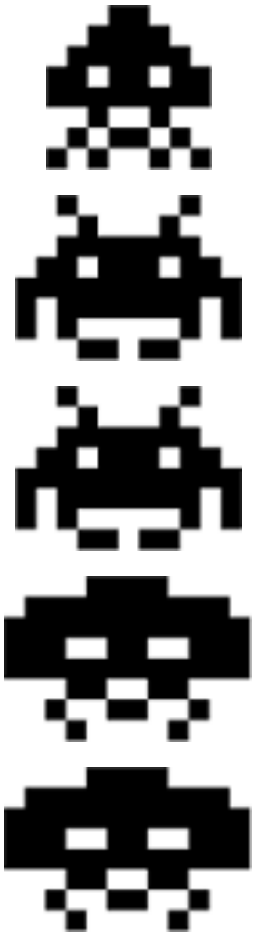
INVADERS FROM SPACE

~

SOFTWARE DESIGN DOCUMENTATION

Software & Documentation by James Armstrong



CONTENTS

INTRODUCTION

This document is an overview of the major systems, architecture, structure, and design patterns used to effectively create the various behaviors of Space Invaders. All system explanations are specific to my object-oriented C# implementation, while it has nearly-identical behavior to the original game the underlying operations are not necessarily similar.

❖ Space Invaders

Space Invaders is an arcade shooting game made by Taito in 1978. The player controls a ship, which can only move left and right, that shoots missiles upwards to defend against a huge formation of alien 'space invaders' that are slowly descending as they move back and forth on the screen. The aliens will increase the speed of their approach as their numbers dwindle, and come in multiple waves. If the player is shot down by an alien, there are a number of extra lives for them, but if the aliens reach the bottom of the screen & land on earth to invade, then it's true game over. The game also featured a 2-player mode where players would take turns fending off aliens, trading on deaths. Below is a link to some gameplay of the original arcade game, for reference.

- http://www.youtube.com/watch?v=VP2T3YlTDG8&NR=1;

❖ Online Demo

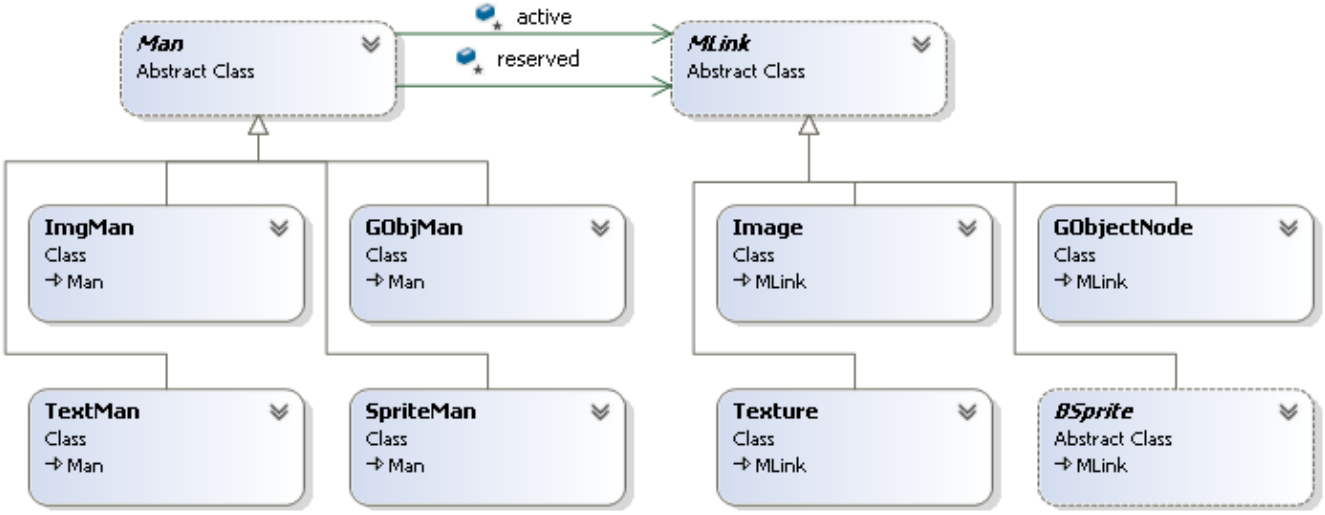
A gameplay demo of the finished version of this implementation is also posted online, with commentary and discussion of features; follow the link below.

- http://youtu.be/kCdTYQdpruI?hd=1

OBJECT MANAGERS

System Classes

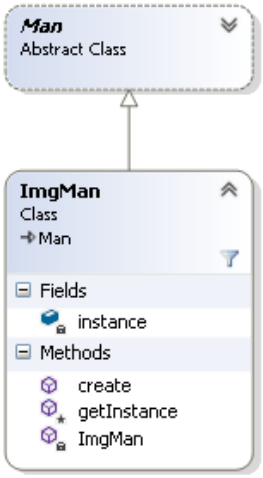
Man (abstract)	TextMan	TimerMan
MLink (abstract)	GobjMan	EventMan
ImgMan	SpriteMan	SBatchMan



Overview of Object Manager classes

Object Managers are used in the application to store various types of objects after they're instantiated, giving a means to search for the objects later and allow the client to use the objects and their data in various places throughout the project. In general, there are managers for nearly every major type of object used by the game, and each is specific to its type of data. *SpriteMan*, our Sprite manager, only deals with Sprite objects and no other object types; the same is true of our *EventMan*, *ImgMan*, and all the rest.

❖ Singleton Pattern

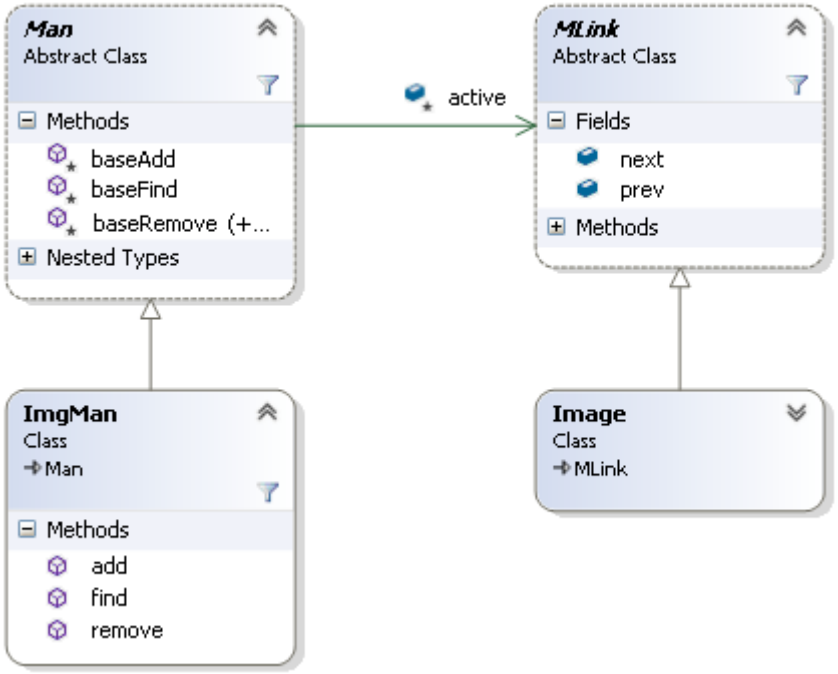


Most of the Object Managers used in the game are Singletons, objects with a single static instance of themselves that is only accessible by a static accessor method. Singletons also generally have private constructors that can only be called by the accessor method (in all of our managers, this is [manager].*getInstance()*) to prevent additional instances from being made. This helps immensely when trying to repeatedly access data that you'll only ever need to instantiate once but will need to use in numerous places, such as the head of a list of objects, since we can call the static accessors from anywhere. This is exactly what we're doing with our managers; our *ImgMan* class –for example— as a singleton, gives us a static access point to all of the images in the entire project. Another benefit to the singleton pattern is that it not only requires the

object be something that you'll only need one of, it also ensures you'll only use one. This sounds minor, but inherently saves memory and helps with debugging and assures that redundant data won't be repeated unintentionally.

In this singleton implementation, our *.getInstance()* accessor method is also private, and is called by the various public static methods we have to access and manipulate the manager's data. This way, the manager's data can only be publicly accessed for specific uses, in some cases these uses particular to the manager.

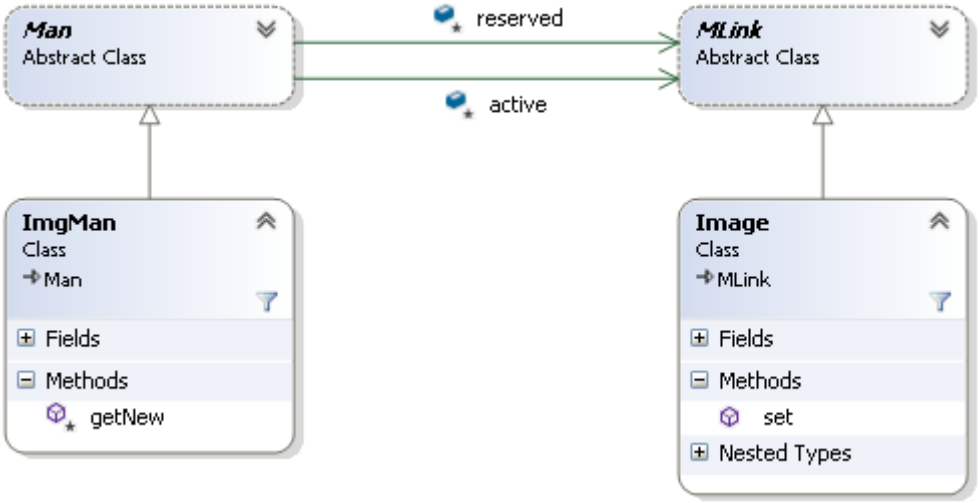
❖ Abstracted List Control



Many of the Singleton managers used by the game—such as the *ImgMan* example above—are used to store lists of objects in use by the game. These managers all need to share the same basic list manipulation functionality: *add()*, *remove()*, *find()*, and objects with *next* and/or *previous* links to their neighbors in the list. This is where our *Man* and *MLink* classes come into play; *Man* is our abstract list manager and *MLink* is our abstract list node, *Man* handles lists of *MLink* objects. All of our managers that hold lists of objects are derived from *Man*, and all objects that need to be held in lists by a manager are derived from *MLink*. This layer of abstraction lets our concrete managers leave the trouble of *add(), find()* and *remove()* to the abstract manager.

❖ Resource Pooling

Dynamic memory allocation and release is the bane of game programming, wasting precious time on every frame & generating lag. One of the goals of this application is to dynamically allocate little to nothing during runtime, and instead pre-emptively allocate all of the space that you'll need at the beginning of the game. The way this application approaches the problem is by allocating 'blank' objects—initializing all of their data to null with the manager's *getNew()*—and, as objects are needed, rather than allocate space for new objects instead fetch a blank and overwrite its data with a *set()*

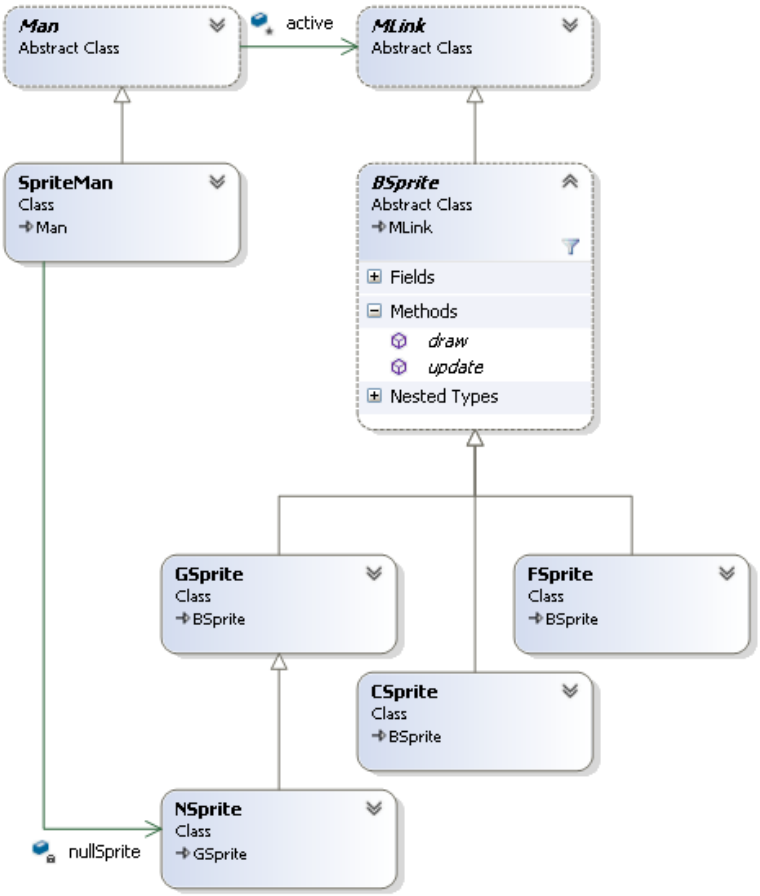


method. The managers handle this behavior by holding two distinct object lists: an *active* list and a *reserve* list. At launch the managers are initialized and they each load a preset number of 'blanks' into the *reserve* list, the preset number is determined through testing but is ideally exactly the maximum number of those objects that will ever be needed at one time. In this system, when a manager calls *add()*, it's actually calling the base class *Man*'s *add()* which takes an object off of the *reserve* list and puts it on the *active* list, then the concrete manager sets the 'new' objects data. This system completely avoids dynamic memory allocation during the operations of the program; when an object would be 'destroyed', it's simply taken off of the *active* list and added to the *reserve* list. This is ideal for something like a game, that could have a variable number of allocations during runtime and could run for a variable—but usually, relatively long—period of time.

SPRITES

System Classes

SpriteMan	GSprite	FSprite
BSprite (abstract)	CSprite	NSprite



Overview of Sprite classes

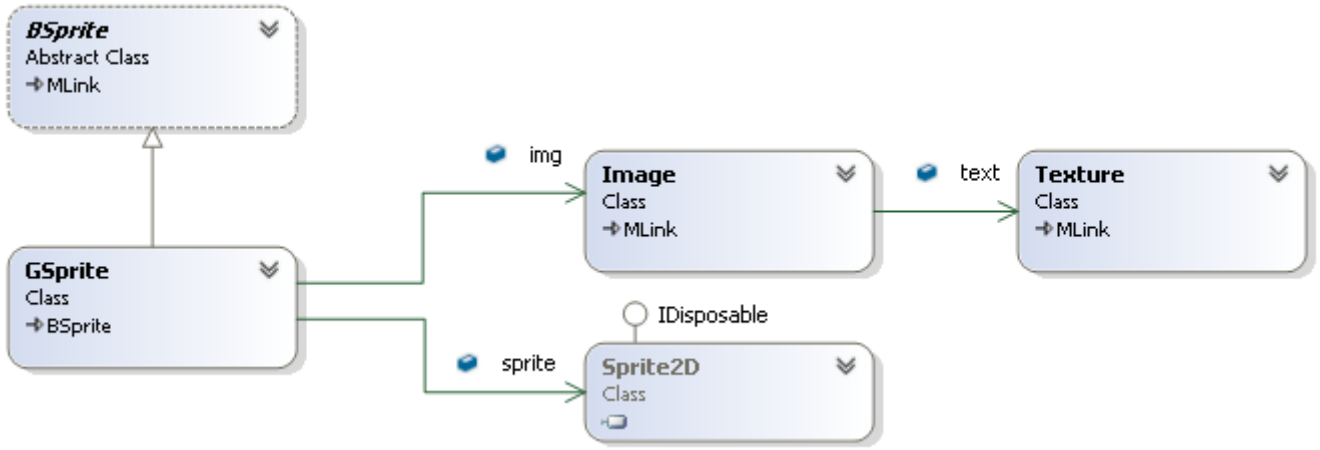
The Sprite System is used in this game—and most games with 2D graphics—to hold and draw images that visually represent our game objects to the player. An object manager, like those discussed earlier, holds lists of *BSprites*, our abstract sprite base class that all Sprite types are derived from. Every *GameObject* has a *BSprite* that it pushes its data onto to ensure the drawn sprites accurately show the object's movements.

❖ Minimal Engine Coupling

Azul, the Game Engine that was used with this project, already has sprite objects as well as methods to update, manipulate, and draw those sprites. However, in this project the coupling to the engine has been made as small as possible; we have our own sprite classes to handle and manipulate, the only functionality borrowed from the Azul engines sprites is the *draw()* routine. Each of our *GSprites*—game sprites—holds an Azul sprite that it, when it needs to draw, pushes its data and image onto and draws

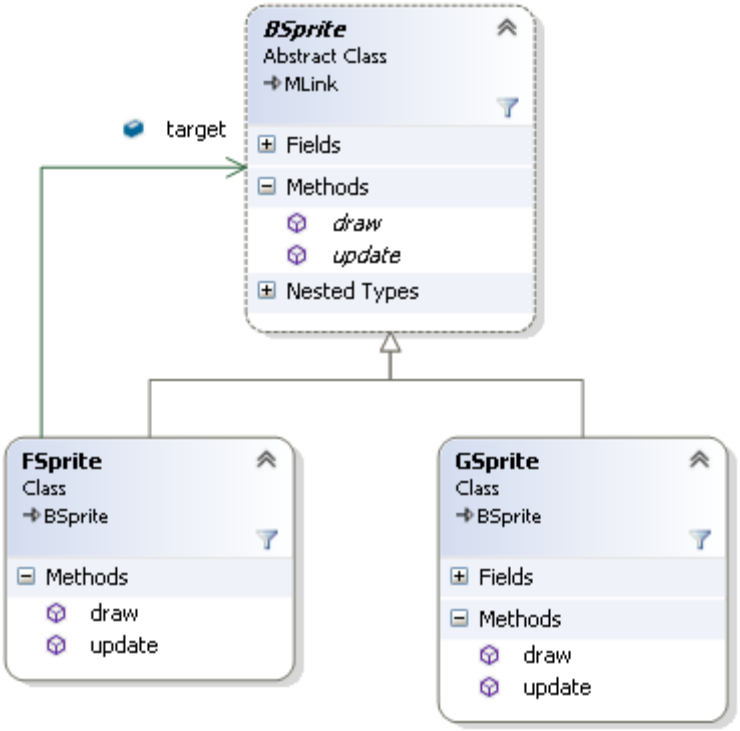
using *Azul.Sprite2D.Draw()*. This way, we minimize the system's dependency on the particulars of the engine and also minimize the risk or problems that could arise from swapping to a new build of the engine.

- Image & Texture Management



The drawing of each sprite is determined by its *Image*, which is determined by the images *Texture*. In this case the *Texture* is the entire image from a source file (in this case, a .tga file), whereas an *Image* is a 'cutout' from a texture, determined by rectangular coordinates. Like *BSprites*, *Images* and *Textures* are held in singleton object managers, *ImgMan* and *TextMan*.

- ❖ Flyweights



Some of our sprites are repeated many, many times with only minor variations. This is most noticeable true of the Alien sprites; the game begins with 55 aliens, 11 with one sprite in common, 22 with another

sprite in common, and 22 more with a third sprite in common. Each of the Aliens needs the full functionality of a *Game Object*, will need to be collidable, and will need to have a distinct unique location. However, between all 55 aliens there are only 3 sprites used, and Aliens' sprites that share a sprite type differ in only a few minor ways: their x and y coordinates on the screen, and their IDs so they can be differentiated from each other (a Name and an Index). Under these circumstances the Flyweight Pattern becomes very useful by allowing us to isolate the data that is unique to each alien (x, y, name, index) and let them share the rest of the data. This is accomplished by instantiating the data the objects have in common once and letting the flyweights all hold references to that single source of data, while the flyweights themselves only store the data unique to them. This can save huge amounts of memory depending on the number of times an object is repeated, and the amount of data that the objects have in common; every flyweight used indicates that much more memory saved.

In the case of our sprite system flyweights, we treat each flyweight sprite, or *FSprite*, in the same way as a *GSprite*, or any other type or sprite, as they all inherit from the common *BSprite* base class. Because of that our *FSprite* objects are also functioning as Proxy Objects, each *FSprite* is a proxy to the *GSprite* holding the complete sprite data which all of the flyweights are referring to. Often Proxy Objects are used control access to an object, but in our case we're using them to allow us to handle flyweight and non-flyweight sprites agnostically. Our Flyweight Proxies have all of the methods of a *GSprite*, but all of their methods delegate to the *GSprite* they refer to, after pushing their local data onto the *GSprite*. In this way, our *FSprite* looks like a *GSprite*, acts like a *GSprite*, but is actually a memory-saving flyweight proxy object.

❖ Collision Sprites

Data-driven, image-dependent construction. Each *Game Object* holds a reference to a *BSprite*, and also holds a *Collision Object*—which will be discussed more later on—that holds a reference to a *Collision Sprite*, as well as a reference back to the *Game Object* holding it. The *Collision Sprite*, or *CSprite*, has size and location information pushed onto it by the *Collision Object* that holds it, which in turn gets its information from its *GameObject* holder's *BSprite*. In this way objects' *Collision Sprites* are automatically sized, positioned, and updated to match that object's drawn image. *Collision Sprites*, however, draw a bounding rectangle to the screen rather than an image like other sprites.

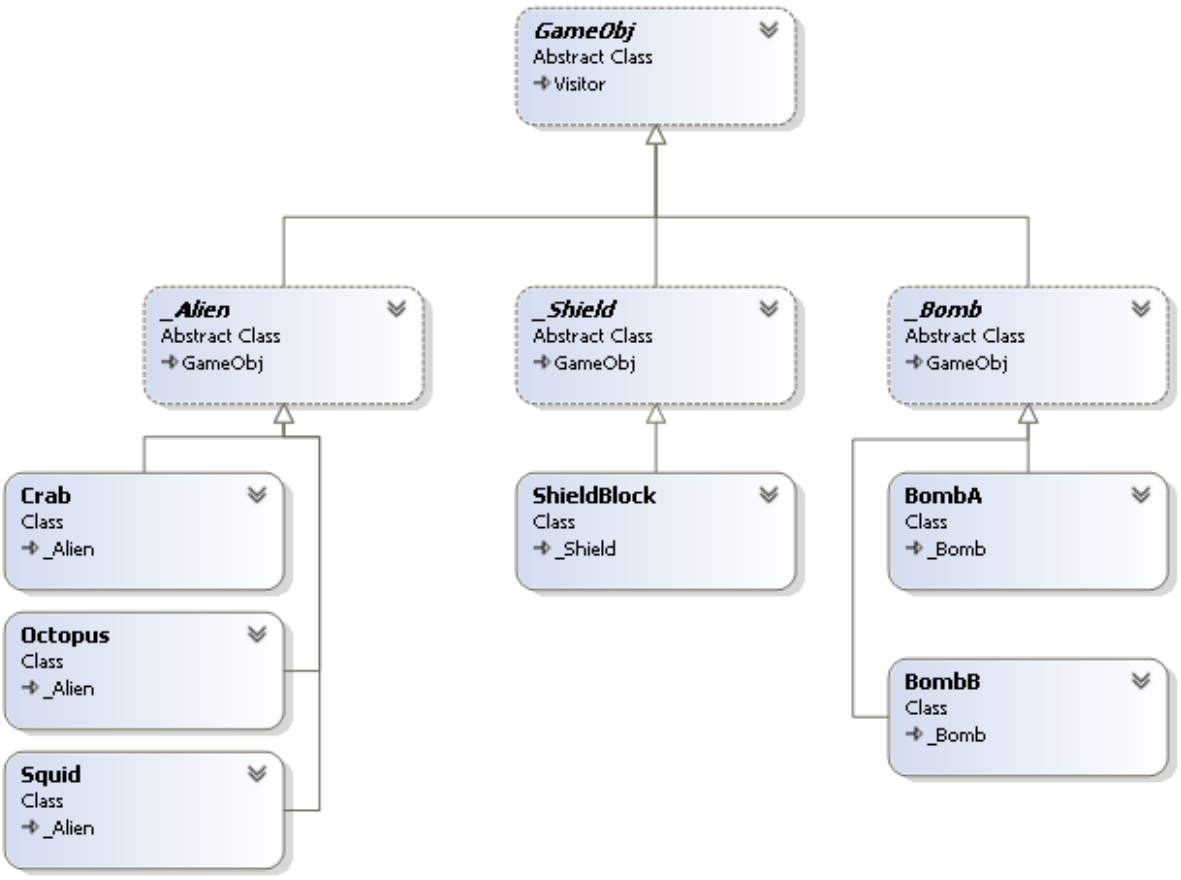
❖ Null Objects

Some *Game Objects* will never need to be drawn, but will still need all of the other functionalities of a *Game Object*, such as objects needed for object hierarchy and collision testing purposes. As *Game Objects*, however, having a *BSprite* is a part of the deal that can't be avoided. Since this issue is an exception more than a rule, finding a way to use the same *Game* Object system without significantly altering it was ideal. Null Objects were the ideal solution to the problem; similar to the Proxy Object technique used for flyweights, the benefit is to be able to handle objects without a draw in the same way as objects that draw normally. Null Objects, in this case Null Sprites or *NSprites*, are like proxies in that they have all of the same methods as the object they are serving as a Null or Proxy of. However, unlike a Proxy Object, whose methods delegate, a Null Object's methods simply do nothing. In this specific case, they have a *draw()* method that can be called and they can be attached as a *BSprite* to an object, but when drawn they do nothing.

GAME OBJECTS

System Classes

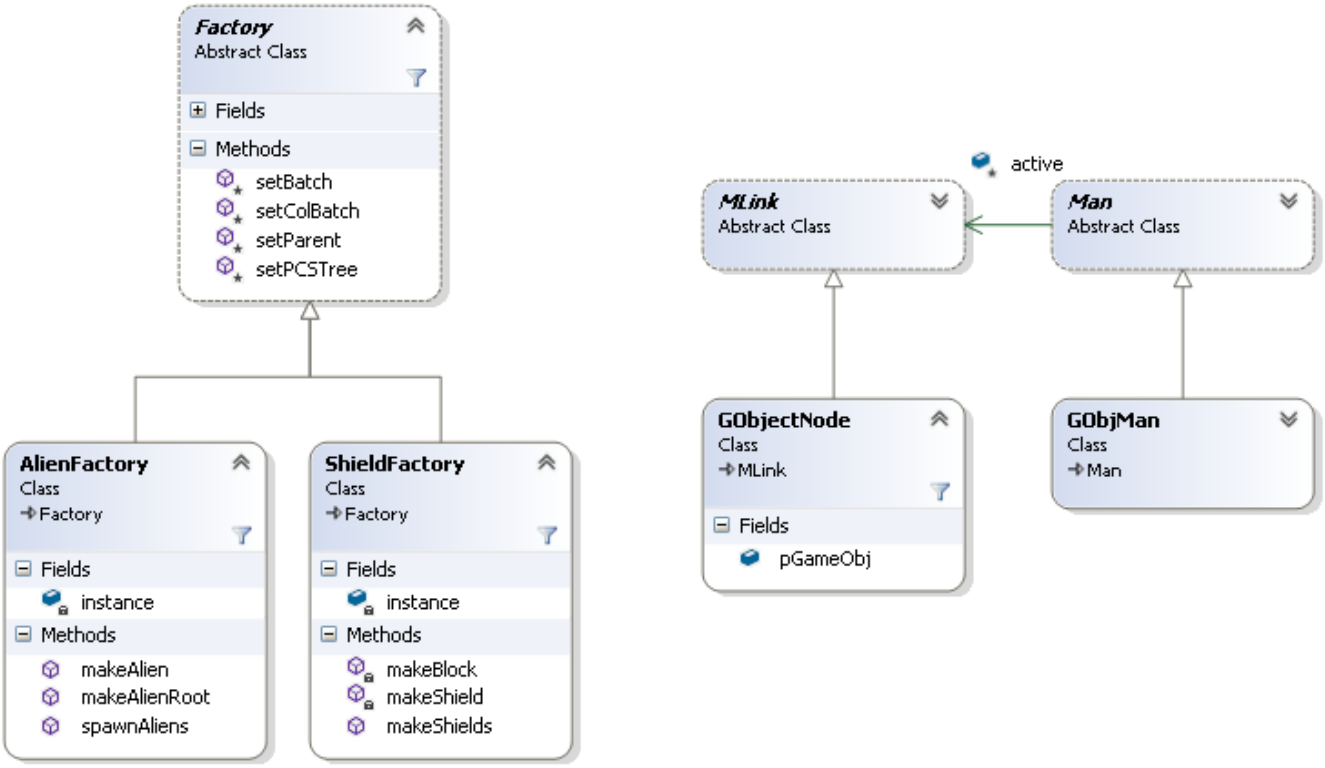
GObjMan	_Alien (abstract)	_UFO (abstract)
GameObj (abstract)	_Missile (abstract)	_Shield (abstract)
GObjNode	_Bomb (abstract)	_Wall (abstract)
Factory (abstract)	_Player (abstract)	_Explosion (abstract)



Overview of Game Object structure

Game Objects are the meat of the game's operation, creating them, handling them, updating them, colliding them, destroying them, and more. All game objects share a certain core of behaviors, being manageable in a data structure, having sprite and collision properties, being identifiable by a name and index, and maintaining on-screen coordinates. However, one class, *GameObj*, the common base class of all game objects, can't reasonably account for how diverse the many types of objects are without becoming much too bloated, so the can is kicked to the next level. Every major type of *Game Object* has another level of abstraction that I call their type class; Aliens derive from the *_Alien* type class, Shields and shield components derive from *_Shield*, and so on. This allows for all of our objects to still be treated as *Game Objects* while allowing a huge amount of diversity and preventing any one class from knowing too much on its own.

❖ Abstract Factory Pattern



All *Game Objects* are fairly complex and many are unique from one another, Aliens will use flyweight sprites whereas UFOs will not (because there will only ever be one UFO at a time), the Player only needs to be spawned in one location whereas Shields have many components that need to be spawned in many locations in a specific pattern. The responsibility for these unique construction methods falls to the factory. The abstract *Factory* base class handles the simple but important jobs: building the object structure and adding newly created objects as needed, accessing and inserting new objects' sprites on the appropriate *Sprite Batches*—which will be discussed later on—and ultimately passing the created object structures off to the Game Object Manager, or *GObjMan*. Then, each major object type has a unique concrete factory to handle the specifics: the size and alignment of the Alien formation, randomly selecting the movement direction of a newly created UFO, etc. This lets us leave all of the details of, say, constructing our Shield system that consists of multiple shields with many columns and blocks each, entirely to our *ShieldFactory* object's *makeShields()* method which will pass off the built components to the appropriate managers as it builds.

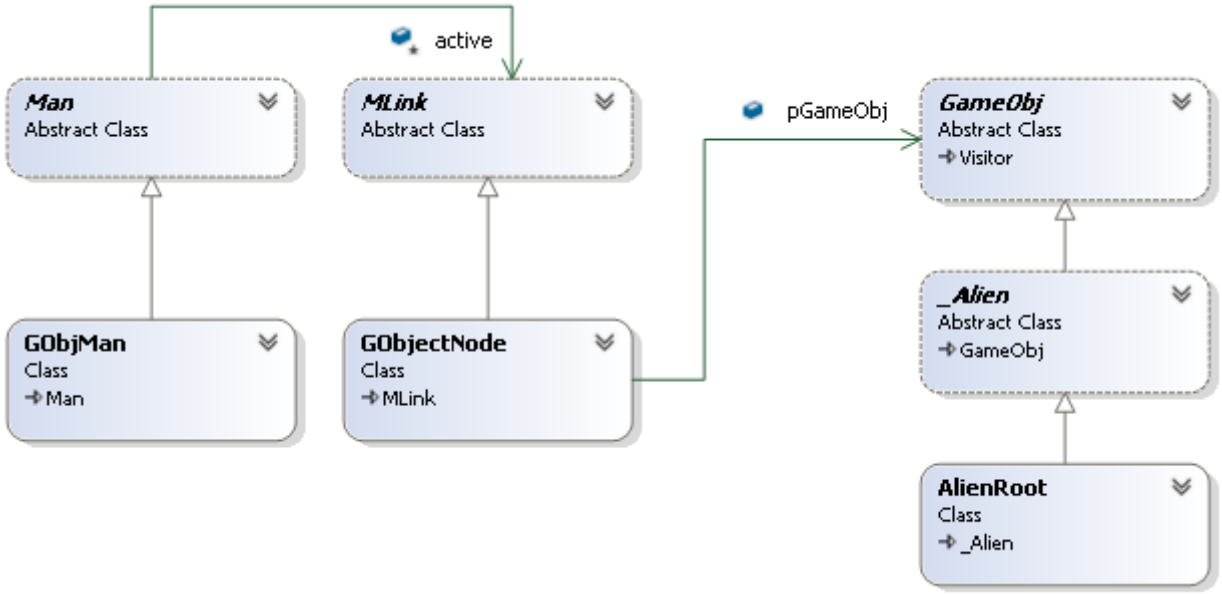
- Specialized Construction

While the factories control much of the intelligence of constructing objects and systems of objects, the details of what makes a Crab Alien different from an *AlienRoot* is outside of its scope of responsibility; our *AlienFactory* knows how to connect a *Crab* to an *AlienColumn* to an *AlienRoot* but knows little more about the objects' construction. This intelligence is built directly into the objects' constructors, for example *Crab* aliens and *Squid* aliens both know which sprites they should be built with. Our abstract type layer also helps with this specialization; all *_Alien* objects will use Flyweight Sprites, all *_Wall* objects will use Null Sprites, and *_Alien* and *_Wall* objects inherently know to construct themselves in this way.

❖ Container Objects

Our object management uses an active-reserved list system to pool objects, as discussed earlier. However, in the case of storing *Game Objects*, the method of making 'blanks' for the reserve list at startup doesn't quite work. Because the objects are so specialized, a blank alien on the reserve and a blank UFO on the reserve cannot be used interchangeably, but the object manager needs to be able to treat all objects it manages equally. The solution to this problem was in Container Objects, *GObjNode*s that hold references to *GameObj*s passed to the manager by a factory. This way many object containers can be instantiated and added to the reserve list at launch without worrying about what types of *Game Objects* they'll be used to hold, their only responsibility will be to receive and hold a reference to a factory-built object.

❖ Storage & Access



We've discussed the process of requesting objects from our factories, which then pass off their newly-built objects and structures to our *GObjMan* that holds them in *GObjNode* containers. But, how does our manager store these objects to make them more easily searchable and accessible? Holding hundreds of objects in a linked list that will need to be searched many times every frame isn't a practical solution. On the other hand, a linked list of nodes is the basis of our abstract manager interface. Fortunately, this can be resolved without any major changes to the manager's structure.

○ Object Trees

Rather than having a 1:1 relationship between all of our *GameObjs* and *GObjNodes*, each of our object nodes hold the root node of an object tree; our *GameObjMan* only holds the root of each major object type. The responsibility of constructing the trees is left to their respective factories that then attach them by the root to our object manager. Our *GameObjMan* then, to find Alien Crab 0, only needs to step through its object nodes to find the Alien tree then search that tree, rather than search through all Game Objects. Another nice feature of this structure is that the trees' root objects are only hierarchal objects—they're needed for structure and collision but aren't drawn objects—so they'll never need to be removed. Even if their trees are empty the roots will remain

so once the object nodes are filled they won't need to change, adding or removing objects can then be done through the trees.

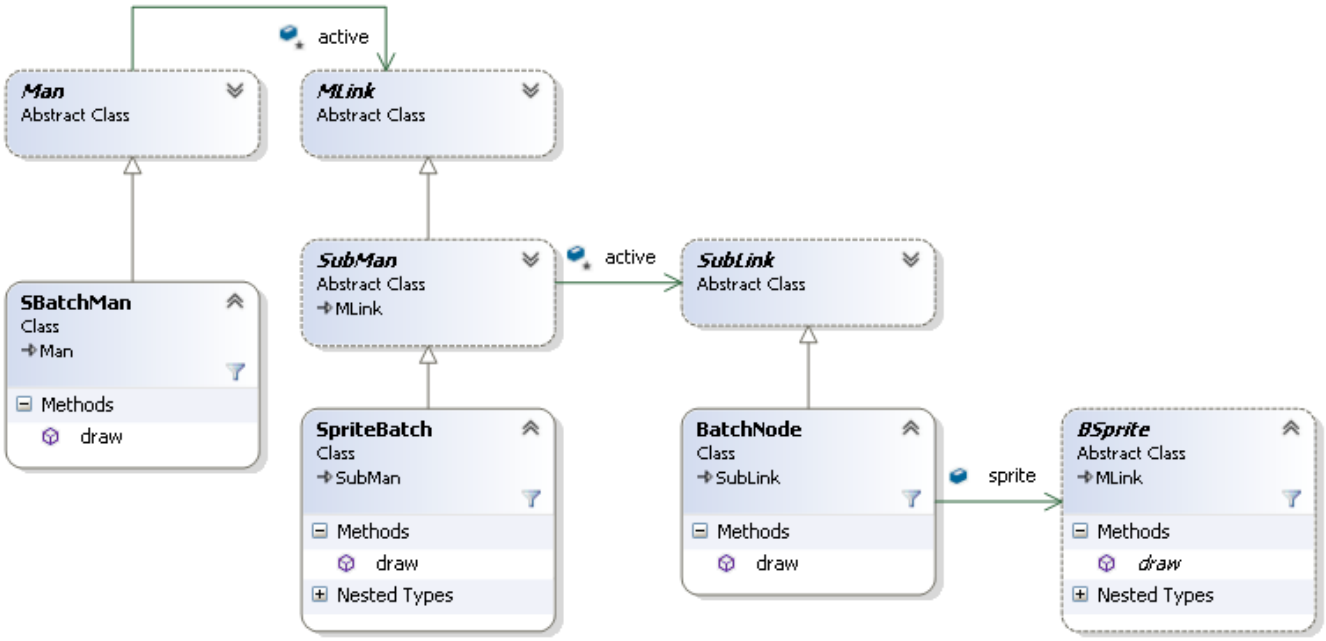
- Temporary Storage

Most objects will be generated by a factory at the beginning of a game and attached to their object trees where they will remain until they are destroyed, and they will only need to be re-added at the end of a round. Some objects however won't be needed at the start of the game, and also won't be needed for the entire game once they are added, and may need to be added and removed a variable number of times in a single round. These include *_Missile*, *_Bomb*, *_UFO*, and *_Explosion* objects; we don't want these objects to be created dynamically every time they're needed, but we also shouldn't leave them on their active object trees to be updated and collided when they're not in use. We could flag these objects as disabled or enabled, but instead we use a separate structure to store the unused objects we'll temporarily need later, *TempObjMan*. Our factories can produce all of the projectiles, explosions, and the like up-front and put them in *TempObjNodes* on our manager and as they are needed they are removed from our temporary manager and added to their respective object trees, and visa-versa when they are no longer needed.

BATCHES & HIERARCHY

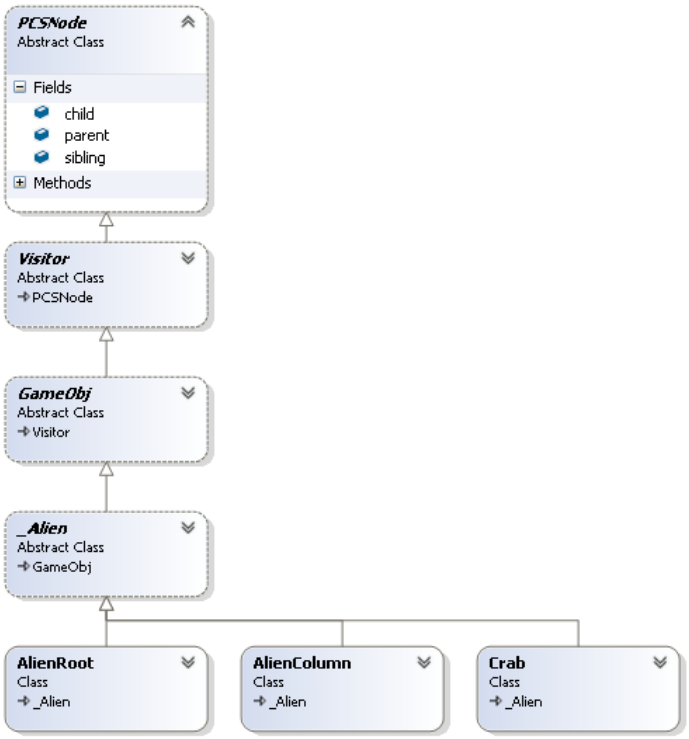
System Classes

SBatchMan	SubMan (abstract)	PCSNode (abstract)
SpriteBatch	SubLink (abstract)	AlienGrid
BatchNode	PCSTree	



Overview of Sprite Batch Structure

Many general operations need to be carried out every frame, such as collision checks, object position updates, and draw operations. All of these operations need to be carried out on nearly every game object in a particular organized sequence for them to work correctly. While Sprite Batches and Object Hierarchies have different uses, they both offer similar benefits: a data structure that holds many objects in an iterable sequence, so that operations can easily be carried out on each object in order.



Overview of Game Object hierarchy

❖ Object Control by Group

The goal of batches and object trees is to be able to operate on a group of objects as a whole, treating them all equally. These structures leave the iterative structure to their abstract interface classes, *PCSNode* and *SubLink*, so each object will know where to find the next object in a sequence accurately as long as the batches and hierarchies are built correctly.

❖ Game Object Hierarchy

Our game objects are built in a multi-level hierarchy, beginning with a root held by our *GameObjMan*. This structure lets any object have knowledge of and access to all of the objects below it in the hierarchy. Certain operations such as movement, updating, and collision take advantage of this by operating on all objects below them when they are called. This makes some things, like moving the entire Alien Grid in-step, trivial rather than tedious. For update operations on the other hand this is actually of critical importance and not just a convenience. Higher objects in the hierarchy derive their data from the level below them, *AlienGrid* position is informed by *AlienColumn* positions and so on, so the hierarchy needs to be updated from the bottom-up, or the higher levels of the tree will be updated incorrectly based on old data. Without updating from the bottom-up, while the bottom level would be accurate, the second-from-the-bottom level would be behind by one frame, and the third would be 3 frames behind, and so on.

○ PCS Tree Composite

To achieve hierarchal movements, updates and other operations, a data structure that holds objects in distinct levels (parents and children) and is also traversable is needed. A Composite object could be

used, an object that could hold any number of children, and any action that could be performed on a child could be performed on the composite. When that operation is called on the composite object it proceeds to call the action on all of its children, some or all of which could also be composite objects. In this implementation a PCS tree is used instead of a composite object, however. PCS stands for Parent, Child, Sibling; while a composite has a variable number of links out to a variable number of children, every node in a PCS tree has only exactly 3 outward links regardless of the number of objects at any level of the structure. Another benefit of a PCS tree is that no separate composite objects are needed; rather than create a composite object and add a child to it, PCS nodes can simply be added to each other. To navigate all of a PCS node's children is as simple as going to its child and then stepping through that child's siblings until a sibling-less node (the last sibling) is found.

○ Iterator

One benefit lost by eschewing the Composite Object structure is the need to iterate. Composite also iterates, but it does so through its structure as each call on a composite object call the same method on all of its children achieving a natural depth-first iteration. However, we can simulate this this by simply making our methods iterate in a depth-first pattern. This could be separated and encapsulated in an iterator, but as there are only two major operations that will need depth-first iteration (*move()* & *update()*) this implementation simply calls those methods recursively on its children and siblings:

```
depthFirst()
{
        this.child.depthFirst();
        this.sibling.depthFirst();
        // do work on this node here
}
```

By adding a few recursive calls as shown, and by using the PCS tree structure, *move()* and *update()* are easily modified to operate in a depth-first pattern through a hierarchy of objects.

❖ Batch Drawing

Every frame, every object with a sprite will need to be drawn, and batches let us control the sequence of these draws. By controlling the draw sequences we enable some amount of depth in our images, we can present images in a foreground-middleground-background relationship. For example, explosions will always need to be drawn above the objects that they are exploding so they will need to be drawn after them, but things like the score should be completely untouchable by other objects, meaning they will need to be drawn in the absolute foreground even after the explosions. Our Batches are held in a linked list structure, the first batch to be drawn effectively serving as the background, and layering on top of that as we step through each batch. Our *Sprite Batch Manager*, or *SBatchMan*, handles this so the client only needs to call *SBatchMan.draw()* to iterate through all batches, which in turn iterate through all sprites, and draw all objects in the correct sequence.

○ Batch Control

Collision Sprites present another challenge; *CSprites* exist and are draw-able only for debugging purposes, and though useful, should not be drawn most of the time. Rather than customizing our collision sprites to be turned on or off, this functionality is pushed up to the batches instead. All *SBatches*

can be enabled or disabled, allowing or preventing them from drawing the sprites they hold. This feature makes turning off collision sprites very simple, as long as they are all placed into the same batch, and could also easily be used for other features and saves the time of iterating through a batches sprites to disable or remove them.

TIME EVENTS

System Classes

TimerMan	Event (abstract)	PriorityNode
EventMan	EventNode	
Timer	PriorityQueue	

[system uml]

Overview of Time Event system

Various events in the game, some repetitive and others not, will need to operate on timers rather than by any other inputs. Some, like the Alien Animations & movement, will be repeating and also changing speed over the course of the game. Others, like Bombs being dropped, will need to be repeated at random intervals. A separate Time-Event handling system can encapsulate this complexity, so we don't need to add this responsibility to our Alien and Bomb objects.

❖ Command Pattern

[pattern uml]

Our timer system isolates the time-based changes from the objects that the changes are acting on, and it does this with Commands. Our *TimerMan* holds *Timer* objects that will wait for some amount of time before executing an assigned command held in an *Event* object. In the case of animations, our *AnimationEvent* objects will swap the images of their assigned sprites when they execute. For repeating Time Events, the *Timer* object can simply put itself back onto the timer after it's executed. The benefit of this system is that our *BSprite* objects don't need any knowledge or connection to our *AnimationEvent* objects, and our *Event* objects need no knowledge of our *Timer* objects, these systems can operate and be modified or expanded independently of each other.

❖ Priority Queue

The timer uses a linked-list structure, like our other managers, but keeps all items in a sorted order unlike other object managers. It does this by inserting each new *Timer* in trigger time order, creating a priority queue. This saves a lot of list traversal time; when the *TimerMan* updates and checks for timers that are ready to execute it needs to look no further than the first *Timer* that will not trigger, because all *Timers* below it on the list will have greater or equal trigger times. This prevents unnecessary progression through the timer list, but also allows multiple time events to be triggered at once and each will execute in time-sorted order.

❖ Event Handling

Events are abstract and have various derivatives that are specialized to different cases.

○ Animation

AnimationEvent objects hold a list of *Image* objects and a *BSprite* target, and on execute will swap the target's image with the next on their list. Their targeted sprite will continue to draw and behave as normal, but will now be using a different image.

Movement

This specialized event is only used for timed Alien movements; *MoveEvent* objects require a *GameObj* target and x/y movement offsets. *MoveEvent* objects will always be added to the timer with the same timing as Alien Animations, to remain synchronized.

Sound

SoundEvent objects load an audio clip and, when executed, play their sound effect. These, like *MoveEvent* objects, are used for aliens and will always be added to the timer in-sync with the alien animations. Unlike other time events, however, sound events are also used to react in a non-repeating way immediately to other events by Observers, which will be discussed later.

Spawning

These events, *UFOEvent* and *PlayerSpawnEvent* call factory methods to create new objects when they execute. More specifically, they call factory methods that fetch objects from the *TempObjMan*, set their values, and add them to appropriate sprite batches and object hierarchies. This is used for UFO spawning at random intervals as well as player respawing after a set wait time after death.

Stop Events

StopEvents are a specialized solution to the problem of stopping explosions. When an object is destroyed an *_Explosion* object is used rather than an animation (though some explosion objects do have animations) because objects, when exploding, have unique properties such as becoming non-collidable. *StopEvent* objects require a *GameObj* target; when an Explosion is set, a *StopEvent* targeting it is added to the timer, to remove its target when it executes.

COLLISION

System Classes

ColMan	Subject	Visitor (abstract)
ColPair	Observer (abstract)	ColObject

[system uml]

Overview of Collision system classes

Quick Text Description

- ❖ Collision Objects

…

- ❖ Intersections

…

- ❖ Tiered Collisions

…

  - o Unions

…

  - o Visitors

…

- ❖ Collision Pairs

…

  - o Subjects & Observers

…

CONTROLS & SCENES

System Classes

InputManager State (abstract)	PlayerState_[various] ColumnState_[various]

[system uml]

Overview of Input Control & Scene Control classes

Quick Text Description

- Player Ship Controls

…

  - States & Strategies

…

- Swapping Scenes

…

LOOKING BACK

Quick Text Description

- Multiplayer

…

- Shield Erosion

…

- Removing Switches

…