--DRAFT--
Breve descripción de los módulos de la arquitectura del Engine. (j3dEngine?)

Arquitectura

Tenemos 3 módulos principales:

- *CoreObjects*
- *Processors*
- *GameActionAPI*

Además tenemos un cuarto módulo, *ConfigurationEditor* que sería una aplicación separada, la cual sirve para crear y configurar un *World* [1].

Los primeros 3 y las librerías thirdparty formarían el SDK. El configuration editor sería el tool o herramienta.

En el proyecto no implementaremos física, audio, networking, AI. Sin embargo el engine debe permitir extensión hacia esas funcionalidades.

Las librerías thirdparty que utilizaremos son:

- Xith3D (y dependencias) para scenegraph, rendering y 3D model loading.
- jogl o lwjgl para acceso a dll de OpenGL
- JInput/HIAL para manejo de Input devices

El diagrama de módulos muestra cada módulo y las dependencias entre estos.

Funciones de cada Módulo

CoreObjects

El *GameObject* es el objeto principal del módulo. Un tipo especial de *GameObject* es el *World*. El *World* contiene todos los *GameObject* del nivel cargado en runtime. El World se carga a memoria a través del *GameObjectManager*, el cual provee de servicio de lookup para encontrar los *GameObject* por su nombre.

Un *GameObject* tiene atributos y estado. El estado de un *GameObject* en un instante dado debe ser uno de los posibles estados válidos para ese *GameObject*. Un estado tiene atributos.
Otro tipo de *GameObject* son los *DynamicGameObject*. Los estados de los *DynamicGameObject* tienen comportamiento (*Behavior*). Al cambiar de estado, se ejecuta un comportamiento de desactivación para el primer estado y se ejecuta uno de

[1] *World*: Decidimos llamar World a lo que también se conoce en otros Engine como Level o Scenario.

activación para el nuevo estado. El comportamiento del *DynamicGameObject* será el que tenga el estado actual de ese *DynamicGameObject*.

Hay otros *GameObject* especiales, las *Cámaras* y los *Player*.
Los *GameObject* pueden tener una representación espacial, o sea, una *Geometry*. Esta *Geometry* es una referencia al nodo del Scenegraph que representa el modelo 3D. Cualquier modificación que impacte sobre la geometría del *GameObject*, se realizará a través de la referencia del *Geometry*.

Otro componente importante de un DynamicGameObject es el MovementController. Éste provee la funcionalidad para transformar la posición geométrica del objecto.
Si queremos iniciar un movimiento con una velocidad en una cierta dirección sobre un *DynamicGameObject*, lo haremos a través de su *MovementController*.

Processors

Este módulo contiene la lógica de ejecución del Engine.
El objeto principal es el *Processor*. El *Processor* contiene la lógica de ejecución de algún dominio de la ejecución del Engine.

Hay un *Processor* para cada funcionalidad o dominio: User Interface, Rendering, Game Logic, Physics, Audio, y cualquier *Processor* necesario. Cada *Processor* utiliza las librerías necesarias para llevar a cabo la ejecución correspondiente.
En este sentido el Engine es **extensible**. Si deseo agregar una funcionalidad nueva, por ejemplo que provea una librería de física, sea PhysicsLib. Debo agregar un *Processor* que adapte esa funcionalidad a los objetos del Engine, supongamos que se llama PhysicsProcessor. Ahora el Engine dispondría de un "Physics Engine" el cual estaría compuesto por el PhysicsProcessor más el PhysicsLib.

Cada *Processor* ejecuta una lógica propia del dominio sobre el que es responsable. Además, debe consumir los eventos externos que son colocados durante la ejecución del Engine en la cola de eventos. Cada *Processor* tiene una cola de eventos, y los eventos son específicos del dominio del *Processor*.
El *GameEventMulticaster* se encarga de hacer el dispatch de los eventos a los *Processor* que corresponda.

Por ejemplo, el *GameLogicProcessor* podría tener un pseudo-código como el siguiente:

```
class GameLogicProcessor : Processor{

	execute(){
		for each event in eventQueue{
			processEvent(event);
		}

		objects = GameObjectManager.getAllDynamicObjects();
		for each object in objects{
			object.update();
		}
		//...
	}
}
```

Aunque no nos enfocamos ni testeamos performance, la arquitectura del Engine debe permitir ser mejorada y optimizada. Queremos hacer uso de las capacidades multithreading del sistema operativo y de los nuevos microprocesadores multicore.
El *ProcessorManager* se encarga de cargar la configuración de los *Processor* y disponer los threads de ejecución. El Engine no fuerza un tipo de loop, sino que permite configurar los loops y darles prioridades.

Un ejemplo de archivo de configuración podría ser:

```
<executionConfig>

      <loop name=”main loop” priority=”10”>
            <processor name=”Renderer”>
                  edu.ua.processors.Renderer
            </processor>
      </loop>

      <loop name=”main loop” priority=”5”>
            <processor name=”UserInterface”>
                  edu.ua.processors.UIProcessor
            </processor>
            <processor name=”GameLogic”>
                  edu.ua.processors.GameLogicProcessor
            </processor>
      </loop>

</executionConfig>
```

En este caso se armarían dos loops de procesamiento. O sea, habría dos threads en ejecución paralela. El primer thread sólo ejecutaría al Renderer y tiene la mayor prioridad. El segundo thread ejecuta dos *Processor* diferentes en secuencia, y posee una menor prioridad al primer thread. En este caso lo configuramos de esta manera porque asumimos que hay una dependencia entre la actualización del estado de los input devices y la ejecución de la lógica del juego. También asumimos que el rendering puede ejecutarse en paralelo a estos dos, ya que no hay una dependencia entre las ejecuciones.

GameActionAPI

Este módulo provee la funcionalidad para ejecutar acciones que afecten al *World*. Las acciones u operaciones están representadas y encapsuladas en el *GameAction*. Hay distintos tipos de *GameAction* y pueden agregarse todos los tipos que sean necesarios.

Por ejemplo, el *GeometryAction* es un subtipo de *GameAction* que encapsula operaciones de modificación de la geometría de los *GameObject*.
Podría existir un *GeometryAction* que sirva para cambiar de posición a un objeto:

```
class MoveObjectAction : GeometryAction {

	//constructor
	MoveObjectAction(targetObject, newPosition){
		//...
	}

	execute(){

		moveObjectTo(targetObject, newPosition);

		//...
	}
}
```

Los *GameAction* utilizar la funcionalidad de los demás módulos para implementar las operaciones.
El *GameActionAPI* provee de un facade como punto de acceso centralizado a la funcionalidad del API, éste es el *GameActionOperations*. Tiene métodos simples para ejecutar las distintas operaciones. Estos métodos delegan la ejecución de las operaciones en los *GameAction*.

Cuando se actualiza un *DynamicGameObject* se ejecuta su *Behavior*. La implementación del *Behavior* invoca *GameActions* para llevar a cabo las operaciones necesarias. Pero puede utilizar los métodos del *GameActionOperations* para simplificar el código de la implementación.
Supongamos que el *Behavior* de un *DynamicGameObject* fuese mover al objecto a una posición determinada que varía al azar, se implementaría de la siguiente forma:

```
class RandomMoveBehavior : Behavior {

	execute(){
		//...
		action = new MovementAction(this, randomPosition);
		action.execute();
		//...
	}
}
```

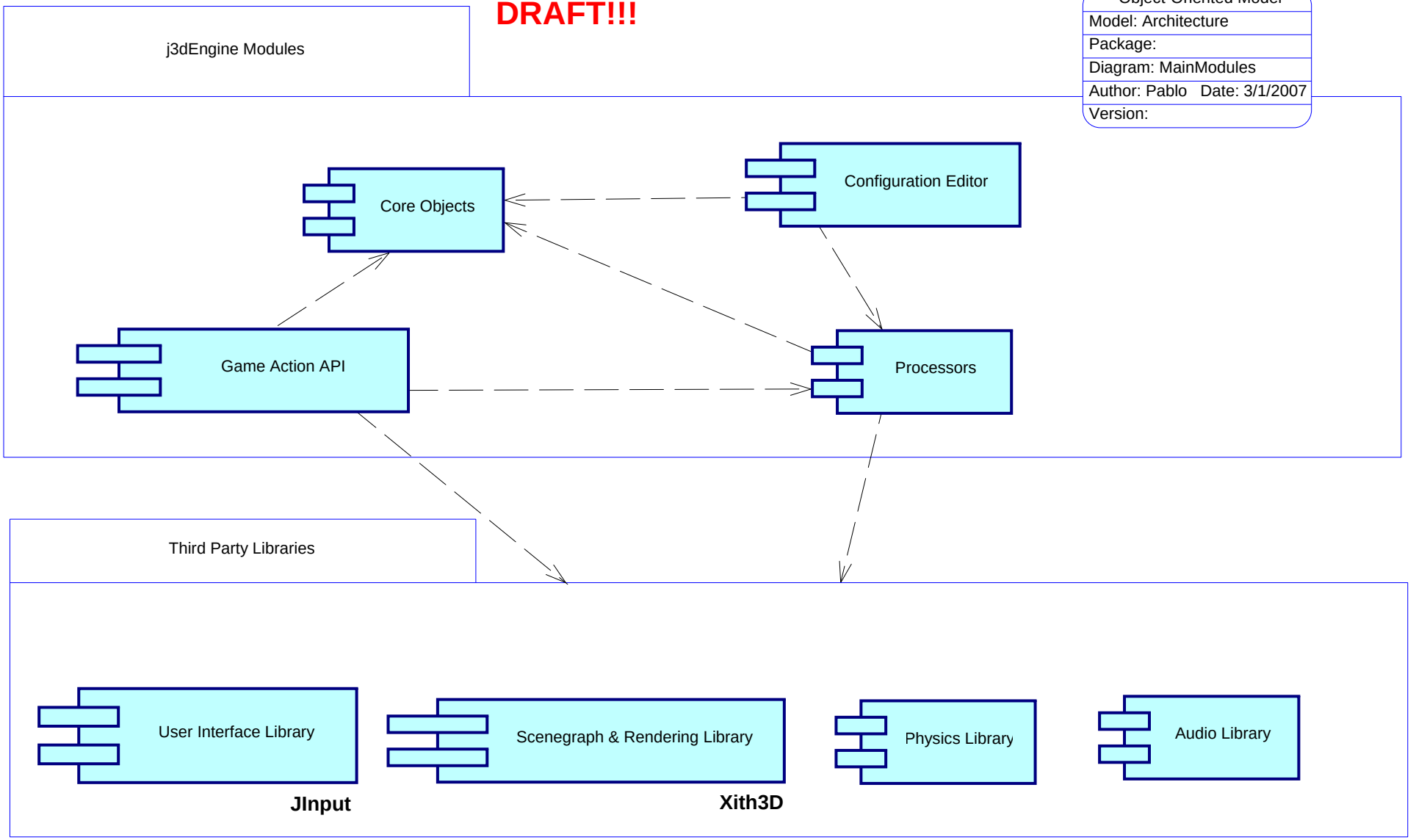
En cambio si utiliza el *GameActionOperations*:

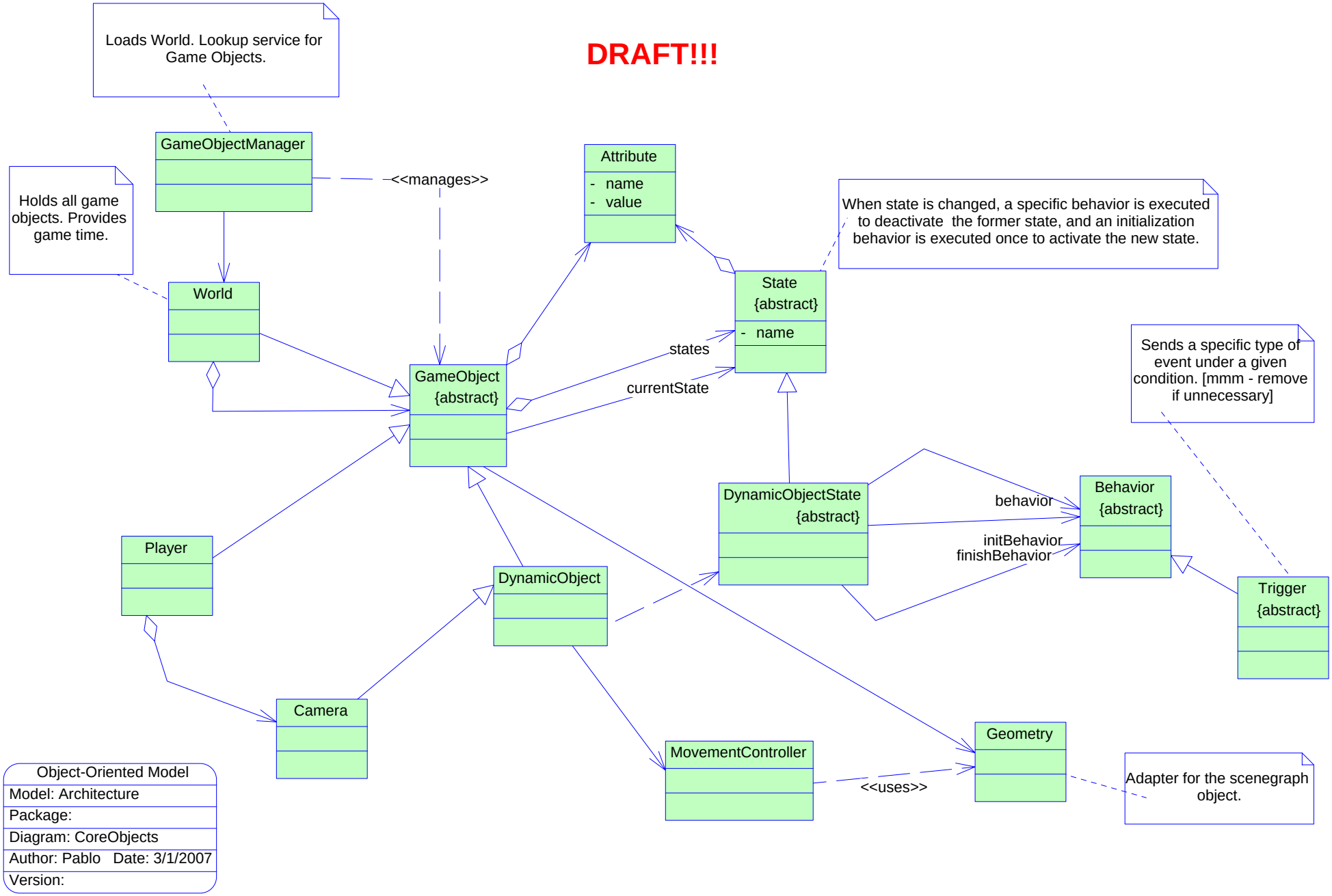
```
import GameActionOperations.*; //importa las operaciones del facade

class RandomMoveBehavior : Behavior {

      execute(){
            //...
            move(this, randomPosition);
            //...
      }
}
```

Si quisiéramos implementar un intérprete de scripts, los scripts serían más fáciles de programar al utilizar este facade.

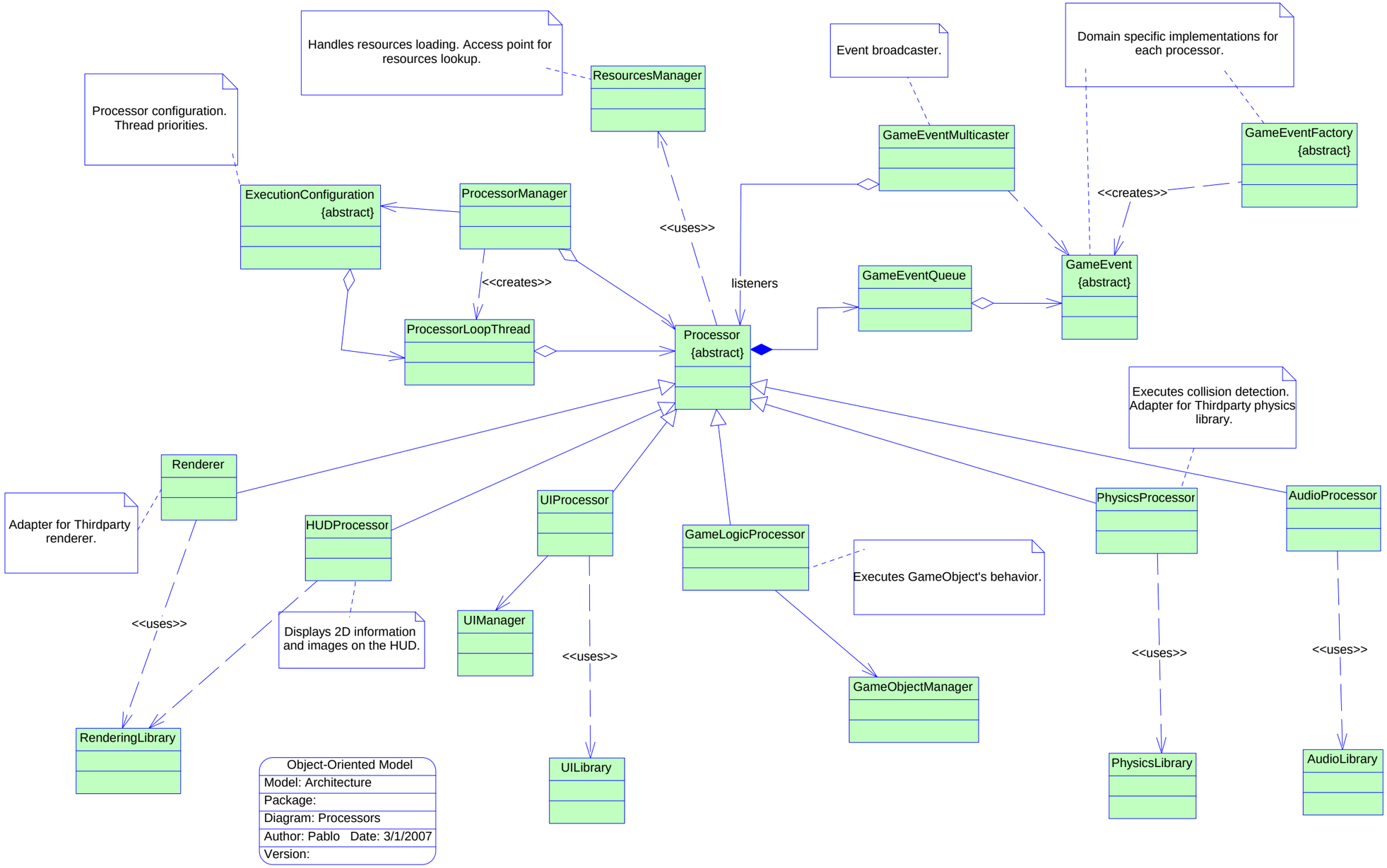
DRAFT!!!

Object-Oriented Model
Model: Architecture
Package:
Diagram: MainModules
Author: Pablo Date: 3/1/2007
Version:




Loads World. Lookup service for Game Objects.
DRAFT!!!
GameObjectManager
<<manages>>
Attribute
- name
- value
Holds all game objects. Provides game time.
When state is changed, a specific behavior is executed to deactivate the former state, and an initialization behavior is executed once to activate the new state.
World
State
{abstract}
- name
states
currentState
GameObject
{abstract}
Sends a specific type of event under a given condition. [mmm - remove if unnecessary]
DynamicObjectState
{abstract}
behavior
initBehavior
finishBehavior
Behavior
{abstract}
Player
DynamicObject
Trigger
{abstract}
Camera
Geometry
MovementController
<<uses>>
Adapter for the scenegraph object.
Object-Oriented Model
Model: Architecture
Package:
Diagram: CoreObjects
Author: Pablo Date: 3/1/2007
Version:

DRAFT!!!


Handles resources loading. Access point for resources lookup.
ResourcesManager
Event broadcaster.
Domain specific implementations for each processor.
Processor configuration. Thread priorities.
GameEventMulticaster
GameEventFactory {abstract}
ExecutionConfiguration {abstract}
ProcessorManager
<<uses>>
<<creates>>
<<creates>>
listeners
GameEventQueue
GameEvent {abstract}
ProcessorLoopThread
Processor {abstract}
Executes collision detection. Adapter for Thirdparty physics library.
Renderer
Adapter for Thirdparty renderer.
HUDProcessor
UIProcessor
PhysicsProcessor
AudioProcessor
GameLogicProcessor
Executes GameObject's behavior.
<<uses>>
Displays 2D information and images on the HUD.
UIManager
<<uses>>
<<uses>>
<<uses>>
RenderingLibrary
GameObjectManager
PhysicsLibrary
AudioLibrary
UILibrary
Object-Oriented Model
Model: Architecture
Package:
Diagram: Processors
Author: Pablo Date: 3/1/2007
Version:

Object-Oriented Model
Model: Architecture
Package:
Diagram: GameActionAPI
Author: Pablo Date: 3/1/2007
Version: