



# UNREAL ENGINE **5**

## GAME PROGRAMMING **DESIGN PATTERNS**

C++, JAVA, AND C#



Sonicworkflow

# Unreal Engine 5 Game Programming Design Patterns in C++, Java, C#, and Blueprints

Sonicworkflow

**Sonicworkflow LLC.**



Copyright © 2021 Sonicworkflow LLC.

All rights reserved

No part of this book may be reproduced, or stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without express written permission of the publisher.

*To Lilly, Dwayne, and Madison*

# Contents

## Introduction

[How to Read this Book](#)

[What is in this Book?](#)

[What is Not in this Book?](#)

[Blueprint Weirdness and Unconventional Methodologies](#)

## Creational Patterns

[Hotel Motel Holiday Inn... Builder Pattern](#)

[Boil and Bubble, Brew and Bottle... Factory Method](#)

[Programming Elitists' Bane... Singleton Pattern](#)

## Structural Patterns

[Adapt and Overcome... Adapter Pattern](#)

[Decorating Overpowered Enemies... Decorator Pattern](#)

[Space Travel Looks Easy... Façade Pattern](#)

## Behavioral Patterns

[The Freaks Come out at night... Observer Pattern](#)

[Make It Rain... State Pattern](#)

[You Sunk My Battleship... Strategy Pattern](#)

## Creational Patterns (Java)

[Hotel Motel Holiday Inn... Builder Pattern \(Java\)](#)

[Boil and Bubble, Brew and Bottle... Factory Method \(Java\)](#)

[Programming Elitists' Bane... Singleton Pattern \(Java\)](#)

## Structural Patterns (Java)

[Adapt and Overcome... Adapter Pattern \(Java\)](#)

[Decorating Overpowered Enemies... Decorator Pattern \(Java\)](#)

[Space Travel Looks Easy... Façade Pattern \(Java\)](#)

## Behavioral Patterns (Java)

[The Freaks Come out at night... Observer Pattern \(Java\)](#)

[Make It Rain... State Pattern \(Java\)](#)

[You Sunk My Battleship... Strategy Pattern \(Java\)](#)

## The Patterns Using C#

[Hotel Motel Holiday Inn... Builder Pattern \(C#\)](#)

[Programming Elitists' Bane... Singleton Pattern \(C#\)](#)

[Adapt and Overcome... Adapter Pattern \(C#\)](#)

[The Freaks Come out at night... Observer Pattern \(C#\)](#)

You Sunk My Battleship... Strategy Pattern (C#)  
Conclusion

# Introduction

Hi, and welcome to ***Unreal Engine 5 Game Programming Design Patterns in C++, Java, C#, and Blueprints***. I wrote this book because I struggled with learning Unreal Engine’s “Blueprints” visual scripting system. I felt like everything I did in Blueprints was sort of a hack. You see, I knew how to program in many traditional programming languages and possessed a master’s in software engineering from an ABET accredited institution. In fact, my skill level was intermediate to advanced Java, C++, C#, and JavaScript. However, these languages are very mature and one can just Google to find the solution to almost any question. This is FAR from the case when it comes to visual scripting!

Visual scripting is not novel. Unreal Engine is not the first tool to implement this kind of programming, nor will it be the last... I see you Unity! There will be a proliferation in learning resources regarding visual scripting as Unreal and Unity become more popular. This book seeks to be the first one discussing design patterns. We will be using Unreal Engine Editor version 5.0 and we will use 4.2x.x for the Java and C# examples. The 4.2x.x blueprints work in UE5 identically.

# How to Read this Book

Blueprint files should be obtained first for maximal understanding from [sonicworkflow.com](http://sonicworkflow.com). It is advisable to start with the Builder Pattern section. This section goes into details other sections will avoid for the sake of redundancy reduction. Therefore, if the reader skips this section they may be lost if they are not hardened blueprint and design pattern experts. One should find the remaining sections much clearer once the Builder Pattern section is read and understood.

## Errors

Submit to support@sonicworkflow.com

# What is in this Book?

This book is a crosswalk from traditional code to visual scripting. This book is about reconstructing traditional and widely used design patterns in a visual scripting environment. The visual scripting environment is Unreal Engine’s “Blueprints.” We will discuss and implement design patterns to solve some of the most common problems in programming. You will not need to read much of the background material on the design patterns if you are an experienced coder. Chances are, you probably have used many of these patterns previously. Do not fret If you are not a coder. If you are familiar with Blueprints, then you can, and should implement design patterns into your workflow when necessary.

C++, Java, and C# “equivalent” code is provided for each design pattern discussed. Additionally, there is an accompanying UML diagram for each design pattern of the Java code implementation. We follow the code in blueprints as close as possible. Some of you may be thinking, “Hey Umar, why would you use Java and C# when Unreal’s main coding language is C++.” I have nothing against C++. I chose to add Java and C# for a few reasons.

1. Java and C# are easier to read. Especially if you are coming from Unity. Unity uses C#. It possesses keywords that immediately communicate Interfaces, and abstract classes. There is no need to explicitly compose pure virtual functions in Java or C#.
2. If one knows C++, then Java and C# are a breeze. If one knows Java and C#, C++ is still difficult to master.
3. I think one would agree that Blueprints are closer in spirit to Java and C# than to C++.

# What is Not in this Book?

This book is not about how to start using blueprints from scratch. This is not a zero to hero blueprints quest. You will struggle if you do not have a decent familiarity with blueprints. Plainly, **this is not a beginner's book**. Comparatively, one would not attempt to "learn" C++ from the "Gang of Four's" design patterns book. Since I brought it up, *Design Patterns, Elements of Reusable Object-Oriented Software* is the granddaddy of design patterns references. It is basically the authority on the topic of design patterns. You would be hard pressed to find a reputable source on the topic not borrowing ideas from that book. The "Gang of Four", aka GoF, refer to the book's four authors.

This book is not a treatise on design patterns. The Gang of Four's book is a must read for further depth and understanding of the topic.

This book is not a level design, artificial intelligence, user interface, animation etc. blueprint tutorial. The example implementations are trivial, like printing a string to the screen/viewport trivial. To that end, focus is set on the design patterns themselves, not bells and whistles in Unreal Engine. This is done with great care and consideration to reduce obsolescence. Unreal Engine changes so frequently that a large portion of written publications are all but obsolete by the time they are published.

Also, this is not a "best practices" book. Please do not use this book as a reference for blueprint best practices. This book tries to follow the "equivalent" code as much as possible. To that end, we may diverge from blueprint best practices to mimic as close as possible what is written in code. Epic has taken great pains to abstract away common programming tasks in blueprints. I encourage you to learn the official API, because often... "there's a node for that".

There are many ways to skin a cat, so to speak. The implementation of each Design pattern may possess alternative and perhaps even better blueprint implementations. This book only provides one example of many.

# Blueprint Weirdness and Unconventional Methodologies

## Blueprint Interfaces

Blueprint scripting in Unreal has some different ways of implementing traditional programming constructs. The most confusing topic, especially coming from a software engineering background is the blueprint interface. The blueprint interface is widely cited as being a “communication” tool. I nearly had a panic attack when I heard Epic’s staff say this in a live training stream. The same “communication” idea is echoed in the documentation. I cannot tell you how many times I yelled at my computer screen while reading documentation on blueprint interfaces. I would yell, “No, interfaces have nothing to do with communication!” All interfaces do is define a guaranteed behavioral contract for objects which implement them. Yet, I’m wrong. One must let go of their traditional understanding of an interface. One must embrace the blueprint interface “communication” deal. Before you think I’m bashing Unreal, let me provide you with a great excerpt on blueprint interfaces from their documentation:

*“A Blueprint Interface is a collection of one or more functions - name only, no implementation - that can be added to other Blueprints. Any Blueprint that has the Interface added is guaranteed to have those functions.... This is essentially like the concept of an interface in general programming”*

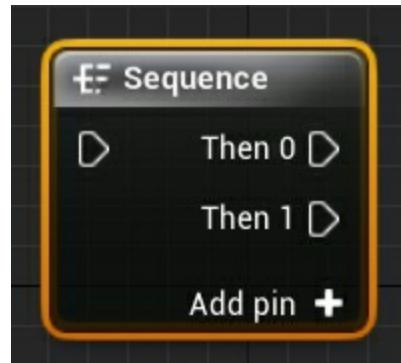
-<https://docs.unrealengine.com/en-us/Engine/Blueprints/UserGuide/Types/Interface>

## Cannot spawn in construction script

The heading is self-explanatory. Why this poses a challenge when trying to construct objects in the “constructor” is not intuitive. The construction script is executed before the game runs in the editor. We cannot “spawn” objects without a world to spawn into. The workaround was using the Add *Child Actor Component* node. There may be a better way, but this works so

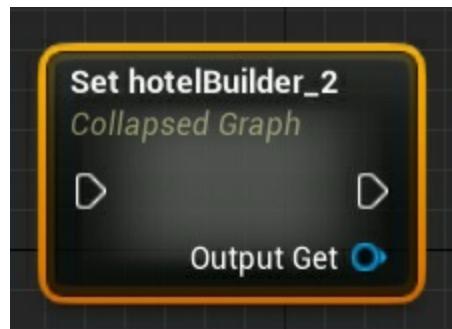
far. We shy away from the extensive use of the construction script in the C++ blueprints but make extensive use of it for the Java and C# implementation.

## Sequence Node Usage



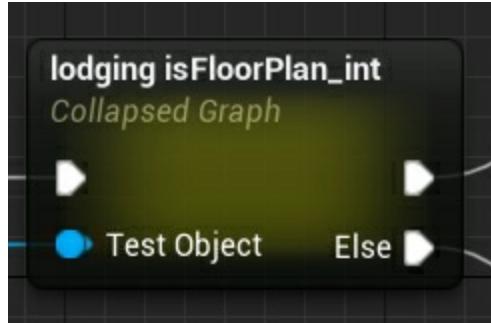
There is heavy use of the sequence node to encourage overall blueprint flow from top to bottom. This is necessary for readability of larger blueprints. Most blueprints are generally composed with a left to right flow. However, there is more space vertically than horizontally in a book.

## Collapse Node



The collapsed graph node is used to separate large blueprints into presentable chunks.

## Yellow Interface check



The collapsed graph with a yellow hue represents an interface check. Specifically, it accepts a test object and checks if it implements the target interface. There may be a validity check on the test object depending on the source of the test object's source. This node is singled out because we will not always go inside this type of collapsed graph for the sake of brevity.

# Creational Patterns

The creational patterns we discuss help to abstract the instantiation process. We use creational patterns when we want to decouple a client from object it may need instantiate. This is especially important. As the number of objects grow, their behavior may begin to diverge. We cannot hardcode every object's behavior into the client. The client would turn into a monolithic mess and the blueprint equivalent would be a nightmare to maintain.

# Hotel Motel Holiday Inn... Builder Pattern

Do you want to encapsulate the creation of complex objects? The Builder pattern might be for you. The Builder pattern seeks to decouple the construction of complex objects from its actual representation (Gamma et.al, 1995). This is done to create flexibility in the construction process because the “products” implementations can be easily swapped in and out.

## The Good

- Decouple construction of complex objects from representation
- Easily swap products to be built/constructed
- Build objects in a step-by-step fashion

## The Not So Good

- Necessary to create concrete builders for each product.
- Some overlap with the Composite design pattern.

## Builder Pattern Implementation

We want to build some lodgings in our game, namely hotels and motels. One can easily replace “hotel” and “motel” with any other thing that needs to be built step by step. However, the interface must be shared for the builder pattern to make sense. For example, an airplane should not logically share the same construction interface as a pizza...unless, NO!

The *Builder* class contains the main function here. In terms of Blueprints, we can think of this class as the one that needs to be physically dragged into the game world. The other “concrete” classes do not need to be physically dragged into the world.

## ***Builder\_Main.h***

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "Builder_Main.generated.h"

UCLASS()
class DESIGN_PATTERNS_API ABuilder_Main : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    ABuilder_Main();

private:
    //The Builder Actor
    UPROPERTY(VisibleAnywhere, Category = "Main")
        class AHotelLodgingBuilder* HotelBuilder;

    //The Engineer Actor
    UPROPERTY(VisibleAnywhere, Category = "Main")
        class AArchitecturalEngineer* Engineer;

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

};
```

## ***Builder\_Main.cpp***

```
#include "Builder_Main.h"
#include "HotelLodgingBuilder.h"
#include "ArchitecturalEngineer.h"
#include "Lodging.h"

// Sets default values
ABuilder_Main::ABuilder_Main()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void ABuilder_Main::BeginPlay()
{
    Super::BeginPlay();

    //Spawn Builder and Engineer
    HotelBuilder = GetWorld()->SpawnActor<AHotelLodgingBuilder>
(AHotelLodgingBuilder::StaticClass());
    Engineer = GetWorld()->SpawnActor<AArchitecturalEngineer>
(AArchitecturalEngineer::StaticClass());

    //Set the Builder for the Engineer and create the buildings
    Engineer->SetLodgingBuilder(HotelBuilder);
    Engineer->ConstructLodging();

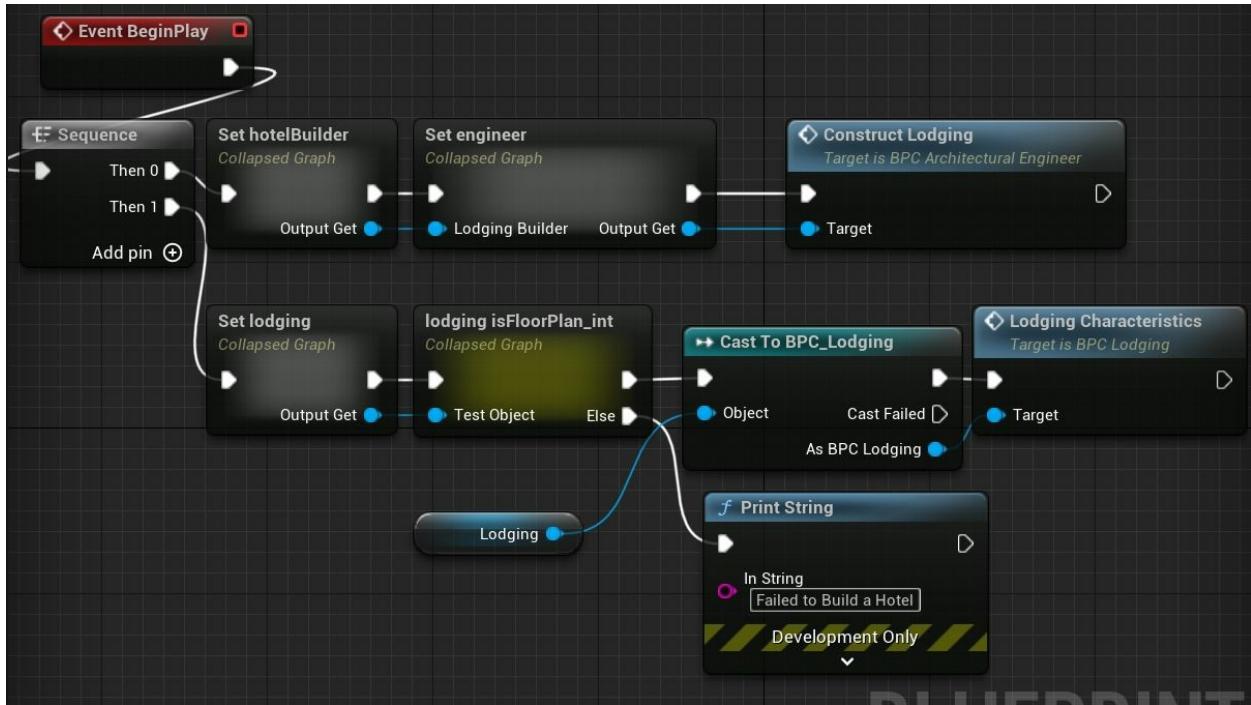
    //Get the Engineer's Lodging and Logs the created buildings
    ALodging* Lodging = Engineer->GetLodging();
    Lodging->LodgingCharacteristics();

}

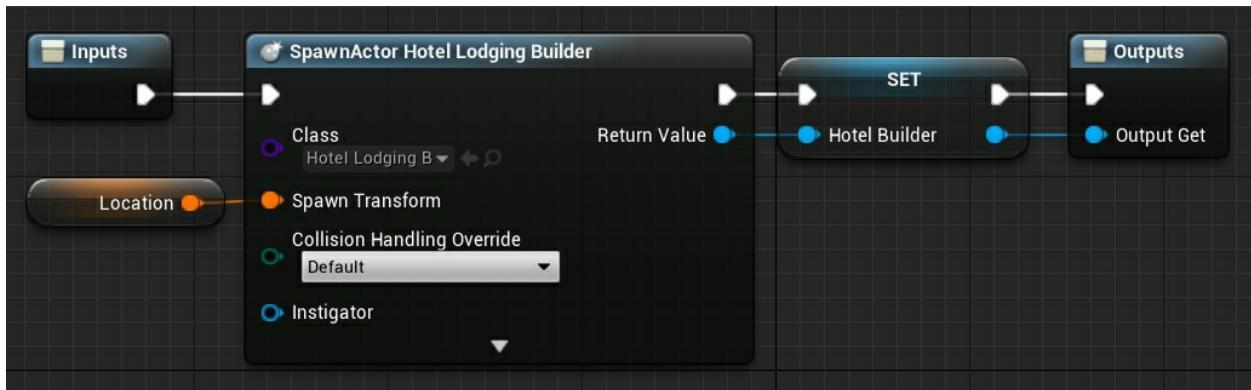
// Called every frame
void ABuilder_Main::Tick(float DeltaTime)
```

```
{  
    Super::Tick(DeltaTime);  
}  
}
```

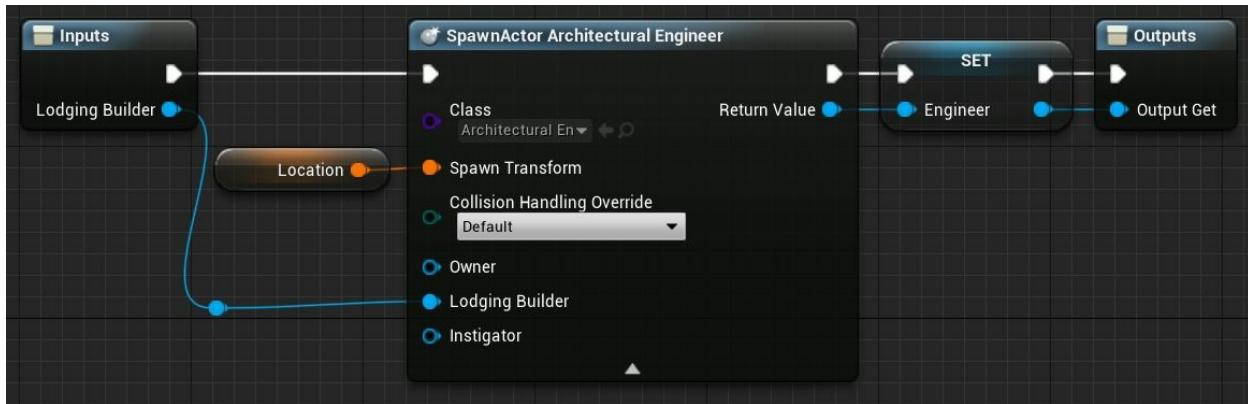
The blueprint equivalent of the *Builder* class:



The *Set hotelBuilder* and *Set engineer* collapsed graph are respectively:



and



The *Set hotelBuilder* collapsed graph represents this line of code:

```
HotelBuilder      =      GetWorld()>SpawnActor<AHotelLodgingBuilder>
(AHotelLodgingBuilder::StaticClass());
```

The *Spawn Actor From Class* node is akin to the *new* keyword in traditional programming. The class to spawn is chosen using the dropdown box under the word “Class”. The Spawn Transform is to one’s own specific scenario. For simplicity, we can just right click on the Spawn Transform and choose “Promote to Variable”. If there is no Spawn Transform then the blueprint will compile with errors. We set (or “initialize”) our *hotelBuilder* variable once the desired actor is spawned.

The same basic idea applies to the *Set engineer* collapsed graph. However, *Architectural Engineer* needs a *LodgingBuilder* to set the Builder for the Engineer as expressed in this line of code:

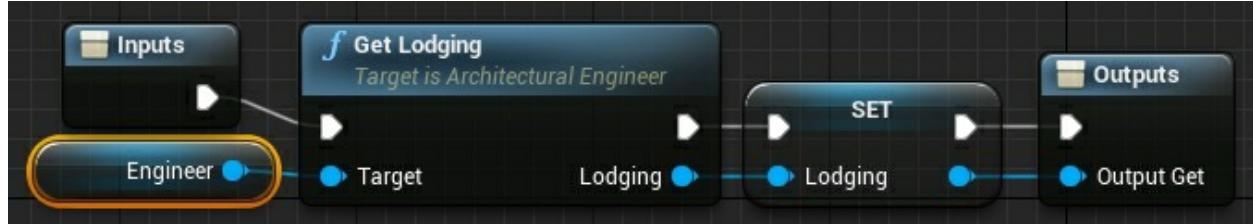
```
Engineer->SetLodgingBuilder(HotelBuilder);
```

Later, we will not be going into such detail on every spawning activity. Trust that there are many, many spawn actor requirements, and discussing each one will become an exercise in tedium.

*The Builder\_Main* blueprint continues with the *Construct Lodging* function call node. This function is called on the *engineer* we spawned previously. The equivalent line of code:

```
Engineer->ConstructLodging();
```

Note the *Sequence* node in *Builder\_Main*. Sequence 0 finishes once the *Construct Lodging* function call executes. Sequence 1 then starts executing. The *Set lodging* collapsed graph:



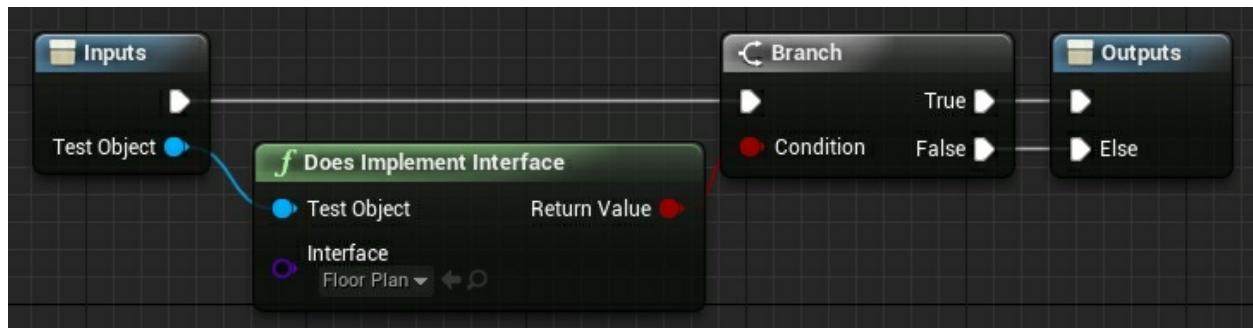
Now, we are equivalently executing this get then set with this line of code:

`ALodging* Lodging = Engineer->GetLodging();`

This last line of non-boilerplate code is:

`Lodging->LodgingCharacteristics();`

This line of code and equivalent blueprint will be discussed in a moment. However, we need to make some validation checks and make sure we built a hotel. We want to make sure the *lodging* actor implements the *FloorPlan* interface. The *lodging isFloorPlan\_int* collapsed graph:



Going forward we will not be showing this interface check collapsed graph. Note, the collapsed graph for this check always possesses a yellow hue. The only item that changes inside the collapsed node is the target interface we are checking the object against. The target interface is always part of the collapsed graph name.

Now we can discuss the interfaces we use in this pattern. First, we create a floor plan interface for all lodgings to implement. Most commercial lodgings in the USA contain a pool, lobby area, rooms, and restaurants. We also create a lodging builder interface for our specific lodging builder classes to implement. The *FloorPlan* interface:

## ***FloorPlan.h***

```
#pragma once

#include "CoreMinimal.h"
#include "UObject/Interface.h"
#include "FloorPlan.generated.h"

// This class does not need to be modified.
UINTERFACE(MinimalAPI)
class UFloorPlan : public UInterface
{
    GENERATED_BODY()
};

class DESIGN_PATTERNS_API IFloorPlan
{
    GENERATED_BODY()

    // Add interface functions to this class. This is the class that will be
    // inherited to implement this interface.

public:

    //The pure virtual functions of the FloorPlan
    virtual void SetSwimmingPool(FString SwimmingPool) = 0;
    virtual void SetLobbyArea(FString LobbyArea) = 0;
    virtual void SetRooms(FString Rooms) = 0;
    virtual void SetRestaurants(FString Restaurants) = 0;
};
```

## ***FloorPlan.cpp***

```
#include "FloorPlan.h"

// Add default functionality here for any IFloorPlan functions that are not
pure virtual.
```

The equivalent blueprint interface is very mundane. The following are the nodes representing the functions in the *FloorPlan* interface.



The saying goes, “Program to an Interface, not an implementation” (Gamma et.al, 1995). In fact, it is a core principle in object-oriented design.

The main benefit of programming to an interface is that it reduces implementation dependencies between subsystems (Gamma et.al, 1995). We see this first hand in our Builder Pattern example. Specifically, The Builder pattern seeks to decouple the construction of complex objects from its actual representation (Gamma et.al, 1995).

The *LodgingBuilder* interface:

### ***LodgingBuilder.h***

```
#pragma once

#include "CoreMinimal.h"
#include "UObject/Interface.h"
#include "LodgingBuilder.generated.h"

// This class does not need to be modified.
UINTERFACE(MinimalAPI)
class ULodgingBuilder : public UInterface
{
    GENERATED_BODY()
};

class DESIGN_PATTERNS_API ILodgingBuilder
{
    GENERATED_BODY()

    // Add interface functions to this class. This is the class that will be
    // inherited to implement this interface.
public:

    //The pure virtual functions of the LodgingBuilder
    virtual void BuildSwimmingPool() = 0;
    virtual void BuildLobbyArea() = 0;
    virtual void BuildRooms() = 0;
    virtual void BuildRestaurants() = 0;
    virtual class ALodging* GetLodging() = 0;
};
```

### ***LodgingBuilder.cpp***

```
#include "LodgingBuilder.h"

// Add default functionality here for any ILodgingBuilder functions that are
```

not pure virtual.

The screenshot of the *LodgingBuilder* interface blueprint follows the same design as the *FloorPlan* interface blueprint and is thus removed for the sake of brevity. Going forward, showing interface blueprint images are minimized unless considered necessary to deepen understanding for the reader.

Next, we create the *Lodging* class which implements the *FloorPlan* interface.

## ***Lodging.h***

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "FloorPlan.h"
#include "Lodging.generated.h"

UCLASS()
class DESIGN_PATTERNS_API ALodging : public AActor, public
IFloorPlan
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    ALodging();

private:
    //The Swimming Pool's name
    FString SwimmingPool;

    //The Lobby Area's name
    FString LobbyArea;

    //The Rooms's name
    FString Rooms;

    //The Restaurants's name
    FString Restaurants;

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
```

```
virtual void Tick(float DeltaTime) override;  
  
//Set the name of the Swimming Pool  
void SetSwimmingPool(FString mySwimmingPool);  
  
//Set the name of the Lobby Area  
void SetLobbyArea(FString myLobbyArea);  
  
//Set the name of the Rooms  
void SetRooms(FString myRooms);  
  
//Set the name of the Restaurants  
void SetRestaurants(FString myRestaurants);  
  
//Logs all the Lodging floors  
void LodgingCharacteristics();  
};
```

## *Lodging.cpp*

```
#include "Lodging.h"

// Sets default values
ALodging::ALodging()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void ALodging::BeginPlay()
{
    Super::BeginPlay();

}

// Called every frame
void ALodging::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

}

void ALodging::SetSwimmingPool(FString mySwimmingPool)
{
    //Set the name of the Swimming Pool with the passed String
    SwimmingPool = mySwimmingPool;
}

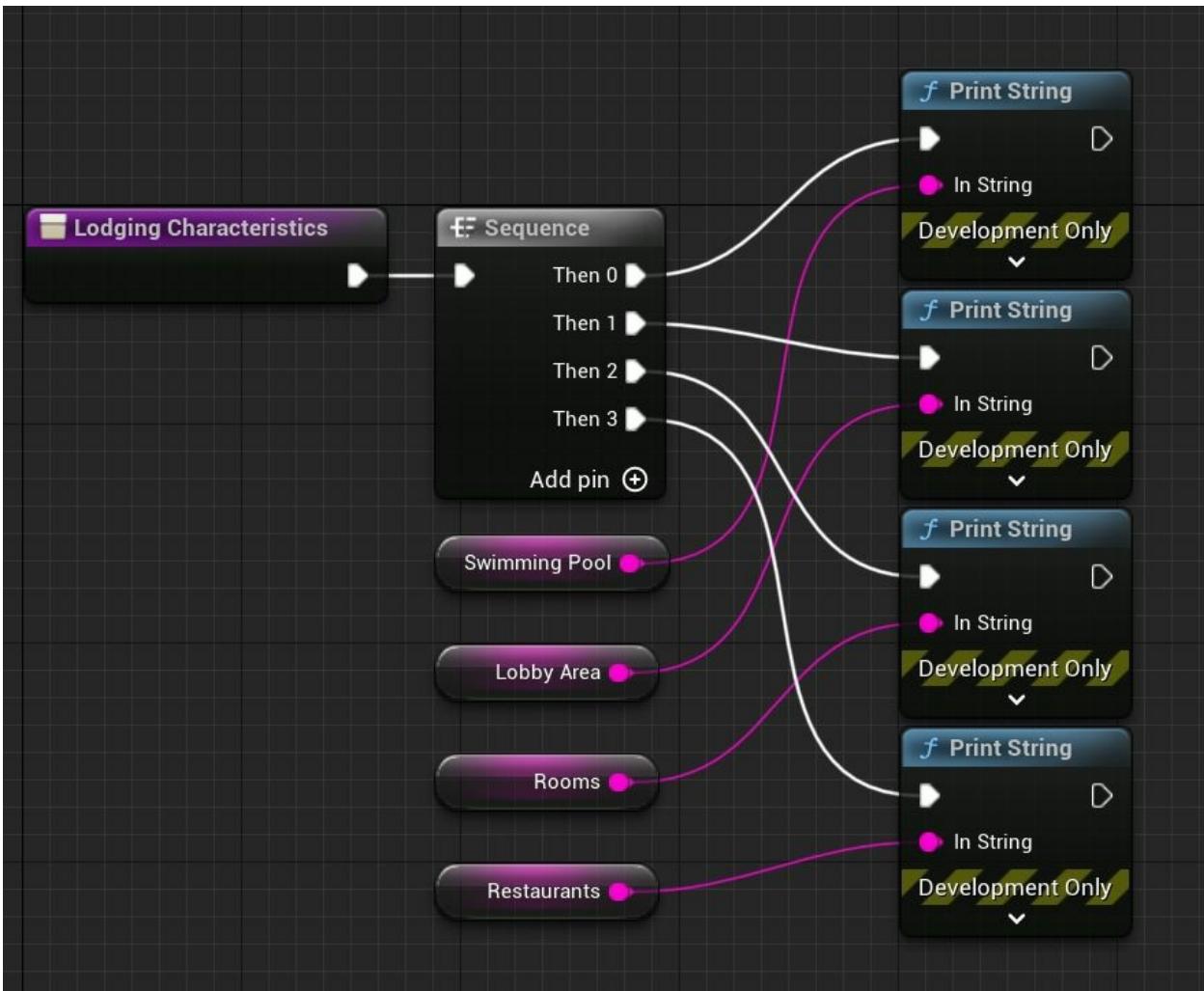
void ALodging::SetLobbyArea(FString myLobbyArea)
{
    //Set the name of the Lobby Area with the passed String
    LobbyArea = myLobbyArea;
}
```

```
void ALodging::SetRooms(FString myRooms)
{
    //Set the name of the Rooms with the passed String
    Rooms = myRooms;
}

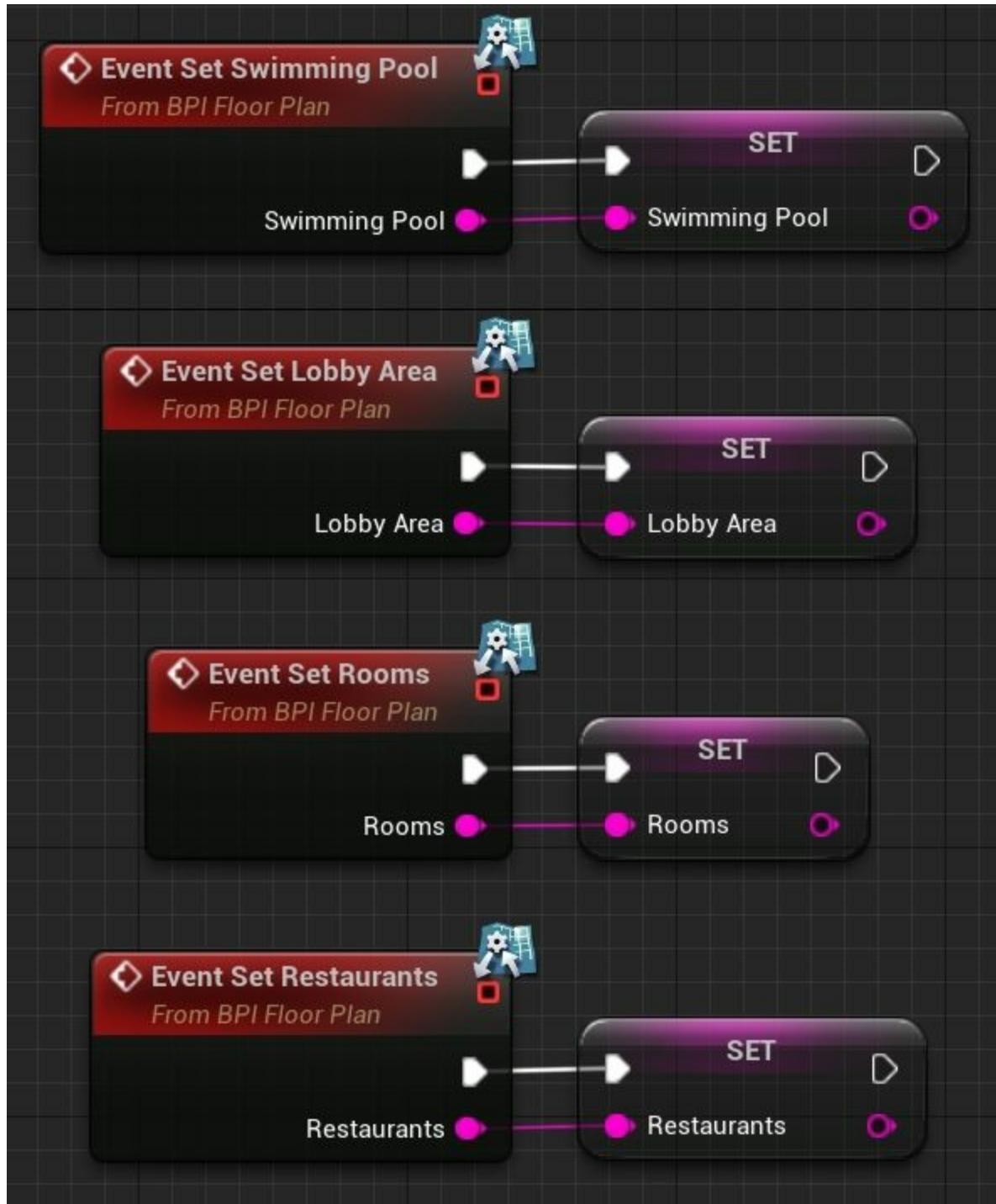
void ALodging::SetRestaurants(FString myRestaurants)
{
    //Set the name of the Restaurants with the passed String
    Restaurants = myRestaurants;
}

void ALodging::LodgingCharacteristics()
{
    //Logs the name of each floors
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
FString::Printf(TEXT("%s"), *SwimmingPool));
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
FString::Printf(TEXT("%s"), *LobbyArea));
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
FString::Printf(TEXT("%s"), *Rooms));
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
FString::Printf(TEXT("%s"), *Restaurants));
}
```

Remember the **LodgingCharacteristics** node in the main blueprint? Here it is:



The Lodging blueprint also contains the events that will be called later by the Architectural Engineer:



Blueprints use *Event* nodes when calling void interface functions. The blueprint version of the *Lodging* class prints the string value passed to the function in addition to setting local variables. Also, we must make sure this blueprint class implements the *FloorPlan* interface by adding it in the class settings Interfaces sections.

The class is the *HotelLodgingBuilder*. It implements the lodging builder interface and provides the custom specs for each type of lodging.

## ***HotelLodgingBuilder.h***

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "LodgingBuilder.h"
#include "HotelLodgingBuilder.generated.h"

UCLASS()
class DESIGN_PATTERNS_API AHotelLodgingBuilder : public AActor,
public ILodgingBuilder
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AHotelLodgingBuilder();

private:
    //The Lodging Actor
    UPROPERTY(VisibleAnywhere, Category = "Hotel Lodging")
        class ALodging* Lodging;

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

    //Create the Swimming Pool
    virtual void BuildSwimmingPool() override;

    //Create the Lobby Area
    virtual void BuildLobbyArea() override;
```

```
//Create the Restaurants
virtual void BuildRestaurants() override;

//Create the Rooms
virtual void BuildRooms() override;

//Returns the Lodging
virtual class ALodging* GetLodging() override;

};
```

## ***HotelLodgingBuilder.cpp***

```
#include "HotelLodgingBuilder.h"
#include "Lodging.h"

// Sets default values
AHotelLodgingBuilder::AHotelLodgingBuilder()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void AHotelLodgingBuilder::BeginPlay()
{
    Super::BeginPlay();

    //Spawn the Lodging Actors
    Lodging      =      GetWorld()->SpawnActor<ALodging>
    (ALodging::StaticClass());

    //Attach it to the Builder (this)
    Lodging->AttachToActor(this,
    FAttachmentTransformRules::KeepRelativeTransform);
}

// Called every frame
void AHotelLodgingBuilder::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);
}

void AHotelLodgingBuilder::BuildSwimmingPool()
{
    if (!Lodging) { UE_LOG(LogTemp, Error, TEXT("BuildSwimmingPool():
    Lodging is NULL, make sure it's initialized.")); return; }
```

```

//Set the Swimming Pool of the Lodging
Lodging->SetSwimmingPool("Indoor Lagoon");
}

void AHotelLodgingBuilder::BuildLobbyArea()
{
    if (!Lodging) { UE_LOG(LogTemp, Error, TEXT("BuildLobbyArea(): Lodging is NULL, make sure it's initialized.")); return; }

    //Set the Lobby Area of the Lodging
    Lodging->SetLobbyArea("Grand Hall");
}

void AHotelLodgingBuilder::BuildRestaurants()
{
    if (!Lodging) { UE_LOG(LogTemp, Error, TEXT("BuildRestaurants(): Lodging is NULL, make sure it's initialized.")); return; }

    //Set the Restaurants of the Lodging
    Lodging->SetRestaurants("5 star Buffet");
}

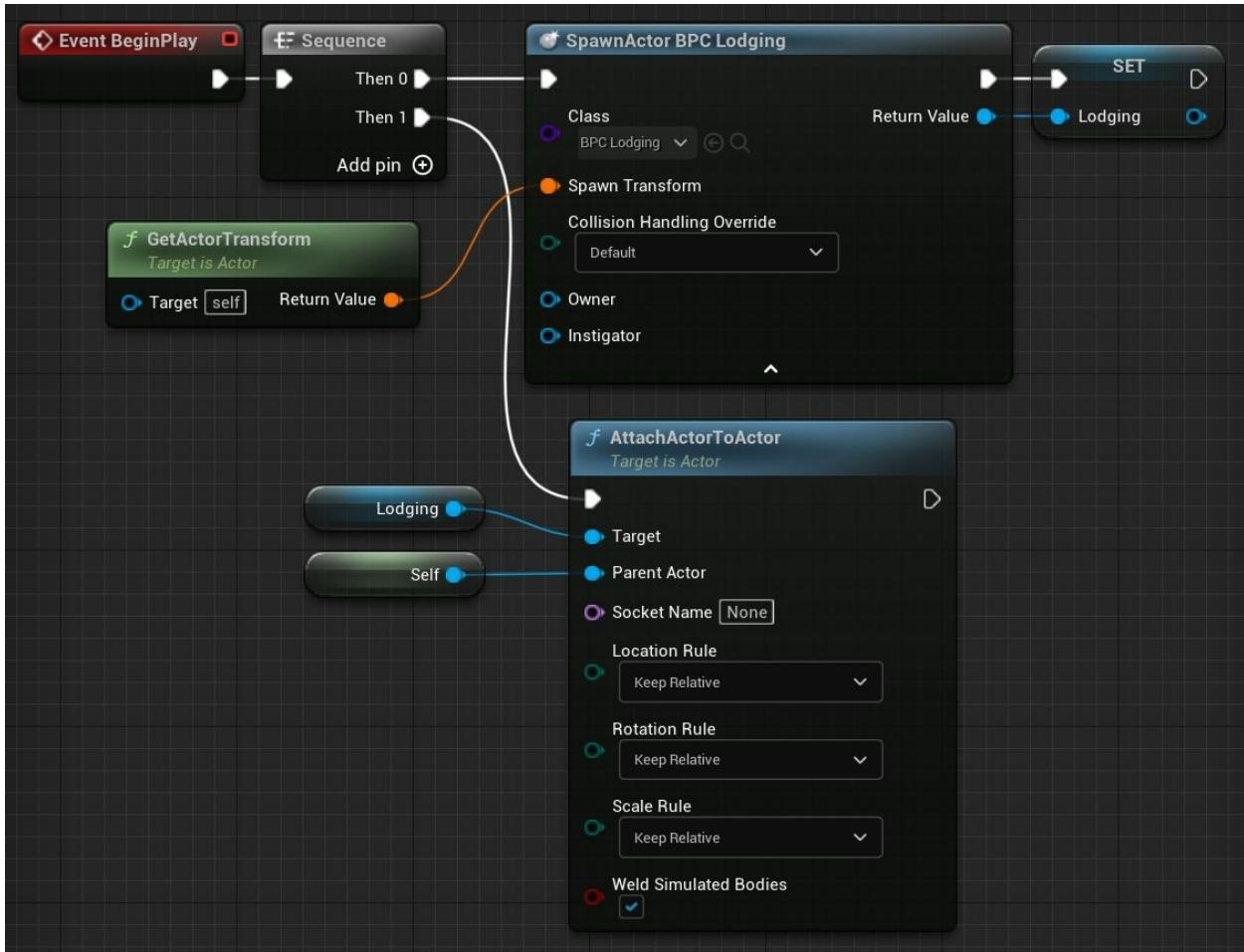
void AHotelLodgingBuilder::BuildRooms()
{
    if (!Lodging) { UE_LOG(LogTemp, Error, TEXT("BuildRooms(): Lodging is NULL, make sure it's initialized.")); return; }

    //Set the Rooms of the Lodging
    Lodging->SetRooms("Luxury Suites");
}

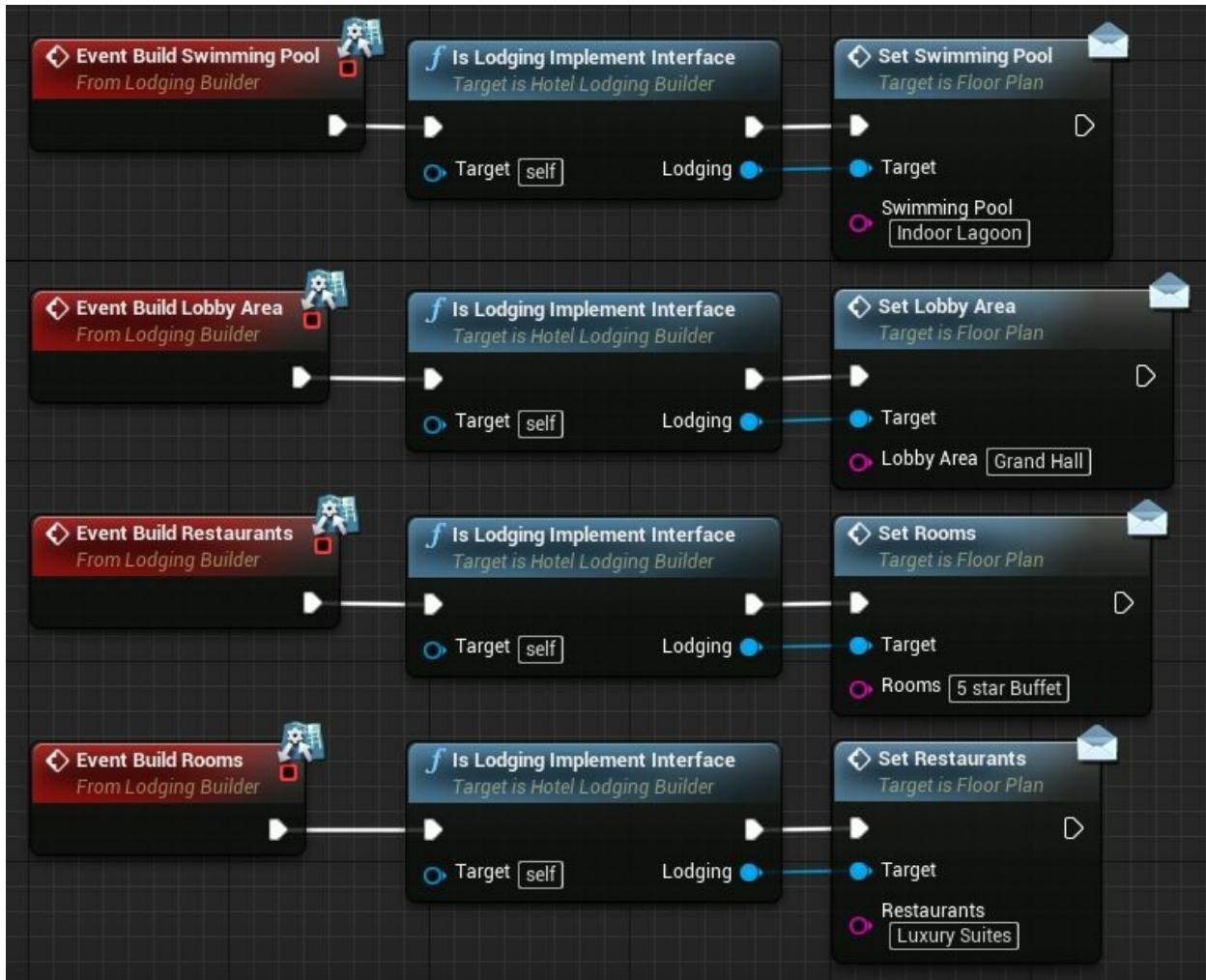
ALodging* AHotelLodgingBuilder::GetLodging()
{
    //Returns the Lodging Actor of the Builder (this)
    return Lodging;
}

```

The *HotelLodgingBuilder* Event Begin Play section of the event graph:

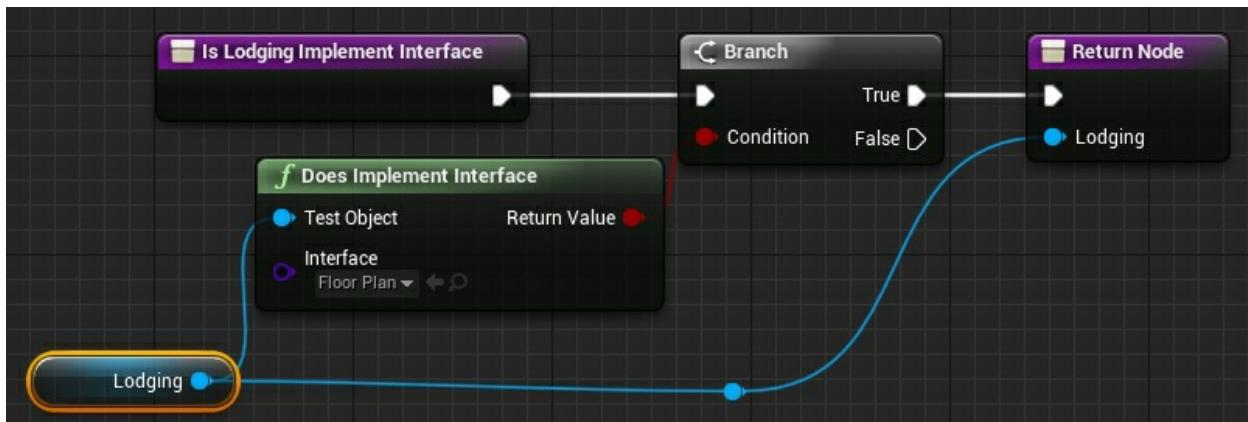


The remaining part of the *HotelLodgingBuilder* event graph:



The *isLodgingImplementInterface* function is a custom interface check function like the yellow hued collapsed graphs.

The *isLodgingImplementInterface* function:



Now we need an architectural engineer to coordinate the step-by-step construction of the lodging.

## ***ArchitecturalEngineer.h***

```
#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "LodgingBuilder.h"
#include "ArchitecturalEngineer.generated.h"

UCLASS()
class DESIGN_PATTERNS_API AArchitecturalEngineer : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AArchitecturalEngineer();

private:
    //The Builder Actor, that must be a LodgingBuilder
    ILodgingBuilder* LodgingBuilder;

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

    //Creates the buildings
    void ConstructLodging();

    //Set the Builder Actor
    void SetLodgingBuilder(AActor* Builder);

    //Get the Lodging of the Builder
    class ALodging* GetLodging();

};

};
```

## ***ArchitecturalEngineer.cpp***

```
#include "ArchitecturalEngineer.h"
#include "Lodging.h"

// Sets default values
AArchitecturalEngineer::AArchitecturalEngineer()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void AArchitecturalEngineer::BeginPlay()
{
    Super::BeginPlay();

}

// Called every frame
void AArchitecturalEngineer::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

}

void AArchitecturalEngineer::SetLodgingBuilder(AActor* Builder)
{
    //Cast the passed Actor and set the Builder
    LodgingBuilder = Cast<ILodgingBuilder>(Builder);

    if (!LodgingBuilder) //Log Error if cast fails
    {
        GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Red,
        TEXT("Invalid Cast! See Output log for more details"));
        UE_LOG(LogTemp, Error, TEXT("SetLodgingBuilder(): The Actor is
```

```
not a LodgingBuilder! Are you sure that the Actor implements that  
interface?"));
```

```
    }  
}
```

```
ALodging* AArchitecturalEngineer::GetLodging()
```

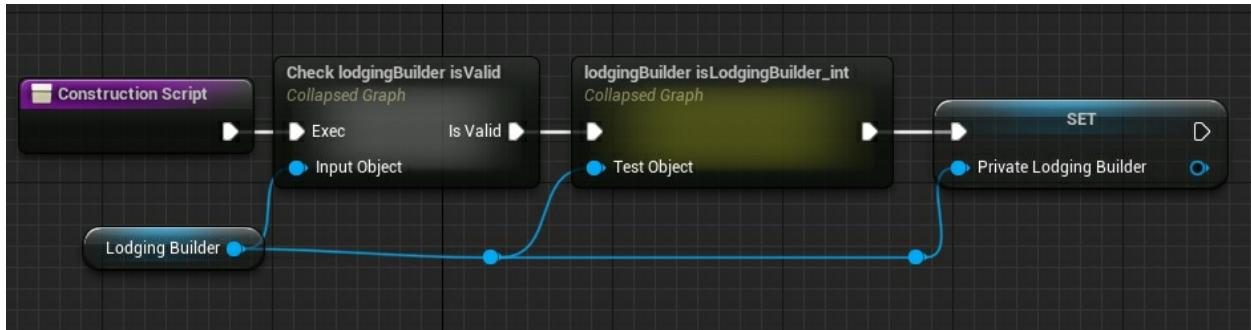
```
{  
    if (LodgingBuilder)  
    {  
        //Returns the Lodging of the Builder  
        return LodgingBuilder->GetLodging();  
    }  
  
    //Log if the Builder is NULL  
    UE_LOG(LogTemp, Error, TEXT("GetLodging(): Return nullptr"));  
    return nullptr;  
}
```

```
void AArchitecturalEngineer::ConstructLodging()
```

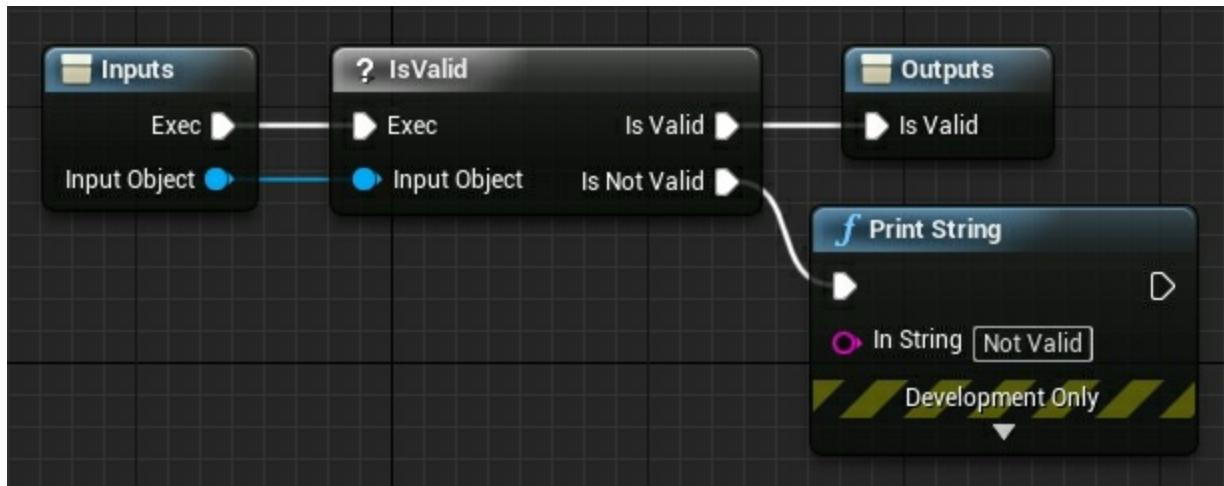
```
{  
    //Log if the Builder is NULL  
    if (!LodgingBuilder) { UE_LOG(LogTemp, Error,  
TEXT("ConstructLodging(): LodgingBuilder is NULL, make sure it's  
initialized.")); return; }
```

```
    //Creates the buildings  
    LodgingBuilder->BuildSwimmingPool();  
    LodgingBuilder->BuildLobbyArea();  
    LodgingBuilder->BuildRooms();  
    LodgingBuilder->BuildRestaurants();  
}
```

The *ArchitecturalEngineer* construction script:



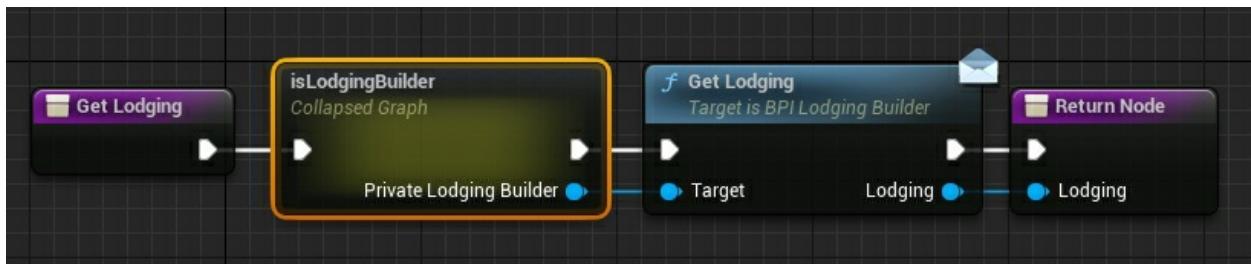
The *Check lodginBuilder isValid* node:



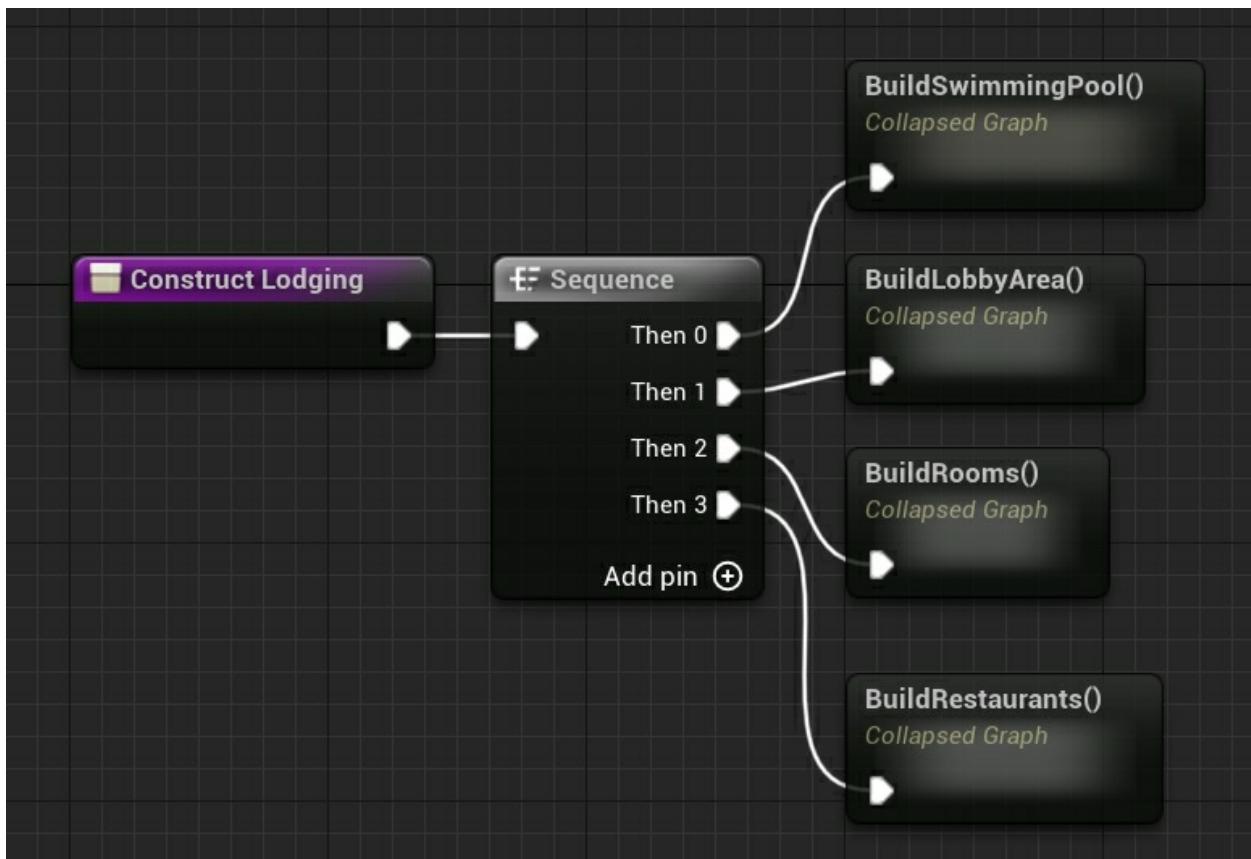
This validity check is executed as a safety mechanism. We want to make sure the object is not null before we start to perform operations on said object. A “Not Valid” string prints to the viewport If the object is indeed null.

The *ArchitecturalEngineer* contains a few important functions, namely *GetLodging* and *ConstructLodging*.

The *GetLodging* function graph:



The *ConstructLodging* function graph:

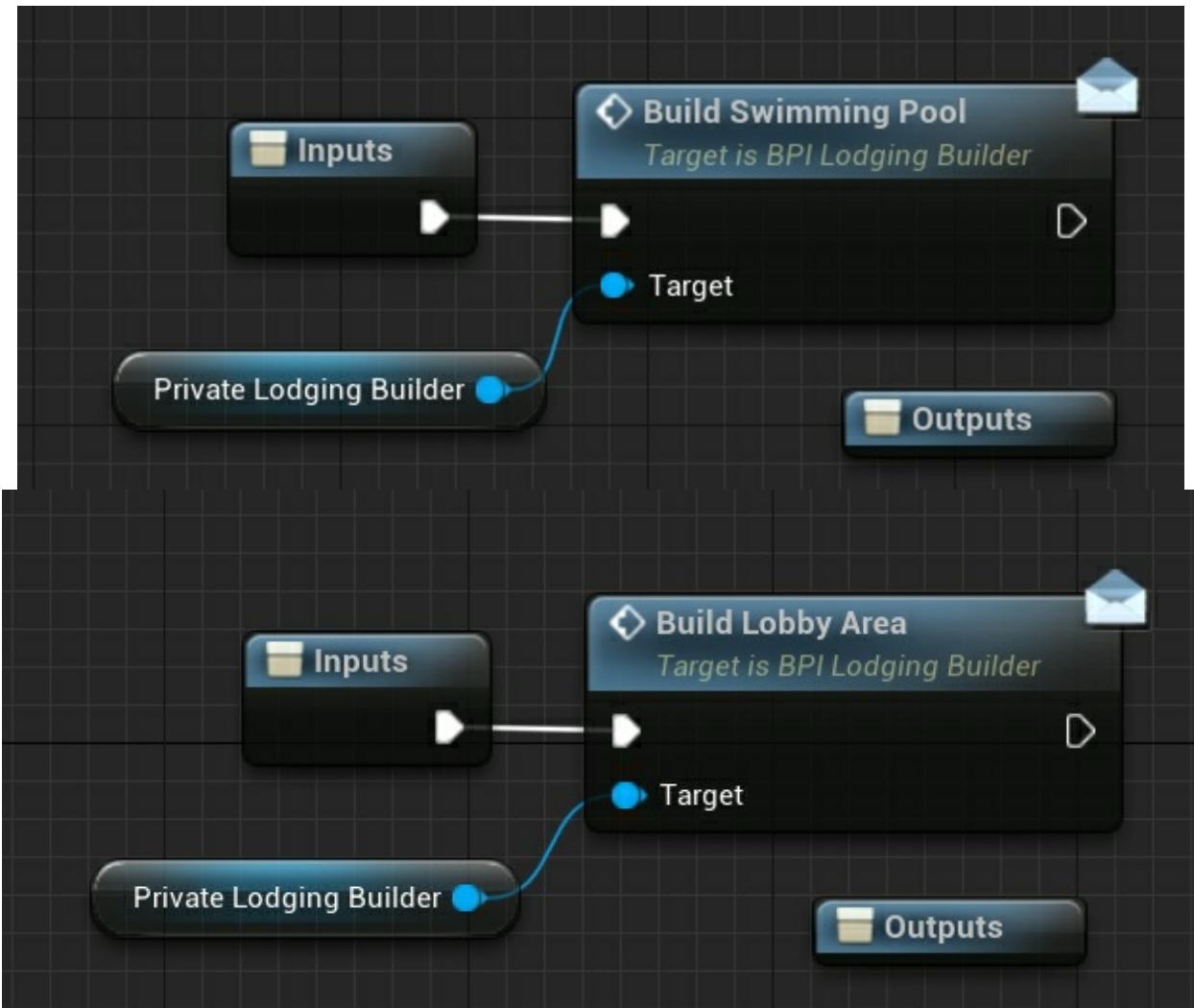


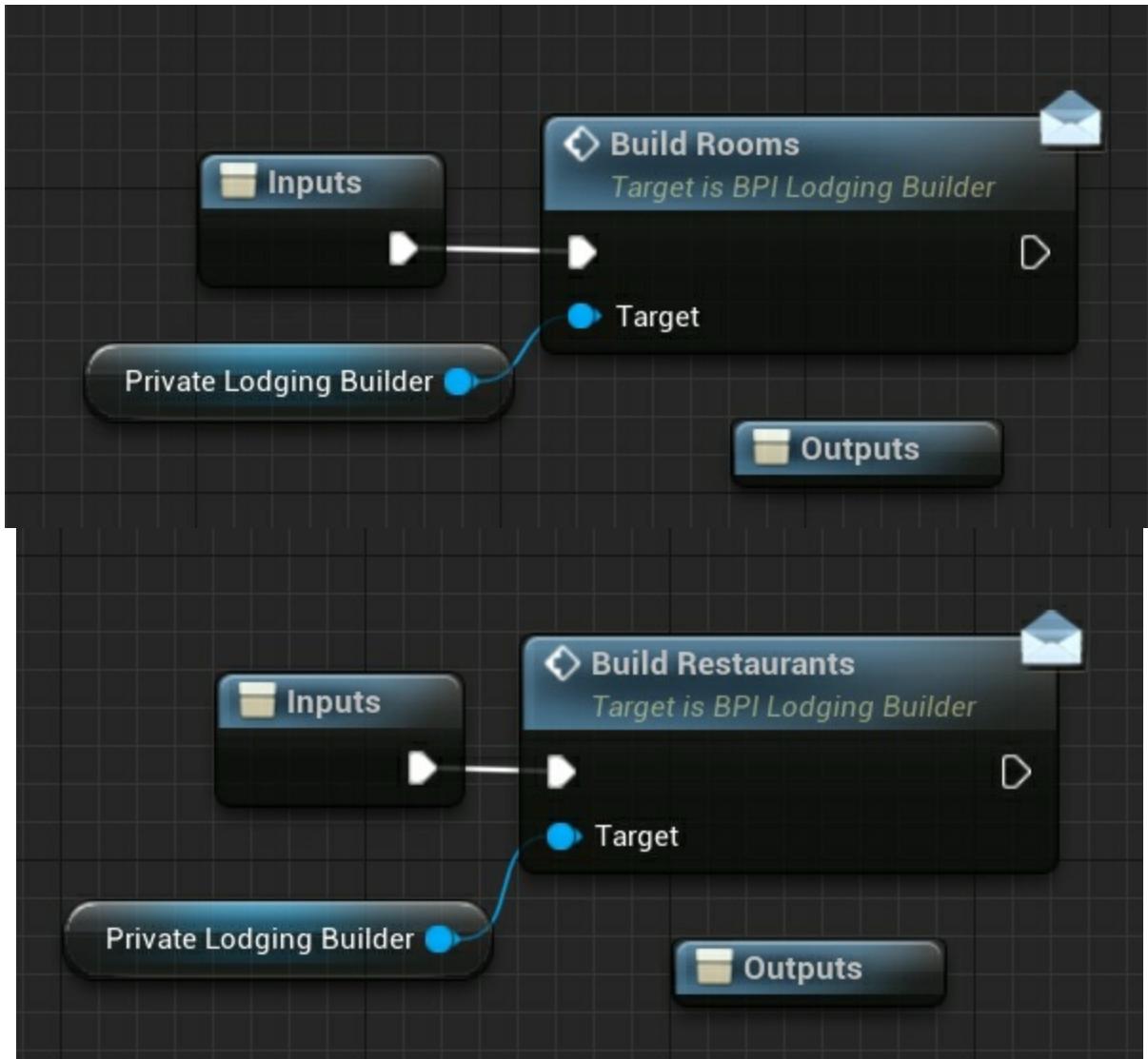
The collapsed graphs represent the body of the *ConstructLodging* function:

```
void AArchitecturalEngineer::ConstructLodging()
{
    //Log if the Builder is NULL
    if (!LodgingBuilder) { UE_LOG(LogTemp, Error,
TEXT("ConstructLodging(): LodgingBuilder is NULL, make sure it's
initialized.")); return; }

    //Creates the buildings
    LodgingBuilder->BuildSwimmingPool();
    LodgingBuilder->BuildLobbyArea();
    LodgingBuilder->BuildRooms();
    LodgingBuilder->BuildRestaurants();
}
```

The *BuildSwimmingPool*, *BuildLobbyArea*, *BuildRooms*, and *BuildRestaurants* graphs are respectively:





This is the final screenshot of the construction of a hotel.



Strings printed to the viewport are not impressive. However, these strings represent “what could be.” In development, one has the 3D or 2D artwork

associated with these strings. Thanks to the Builder Pattern, we have a framework to build different versions of hotels piece by piece.

# Boil and Bubble, Brew and Bottle... Factory Method

GoF says the intent of the Factory Method is to “define an interface for creating an object, but let subclasses decide which class to instantiate” (Gamma et.al, 1995). The Factory Method invokes another important object-oriented principle. That principle is called “Dependency Inversion”. We want subclasses to depend on abstractions and not the other way around. Basically, we want children to depend on their parents.

## The Good

- Encapsulate instantiation
- Concrete products are not coupled to their creator

## The Not So Good

- Can create difficult to follow code because of the level of abstraction.

## Factory Method Implementation

We have creator, concrete creator, and product participants in the Factory Method design pattern. Our example implementation has a *PotionShop* abstract class as a creator. The concrete creators are *InnerRealmPotionShop* and *OuterRealmPotionShop*. These concrete creators inherit functionality from the *PotionShop* abstract class. Our concrete products are:

*InnerRealmHealthPotion*  
*InnerRealmPowerPotion*  
*InnerRealmSkillPotion*  
*OuterRealmHealthPotion*  
*OuterRealmPowerPotion*  
*OuterRealmSkillPotion*

All these products inherit functionality from the *Potion* abstract class.

Using blueprints, we are going to implement the *InnerRealmPotionShop* and its products, namely:

*InnerRealmHealthPotion*

*InnerRealmPowerPotion*

*InnerRealmSkillPotion*

The same ideas and principles apply to the creation of the *OuterRealmPotionShop* and its products.

The *FactoryMethod* class contains the main function. In terms of Blueprints, we can think of this class as the one that needs to be physically dragged into the game world. The other “concrete” classes do not need to be physically dragged into the world.

## *FactoryMethod\_Main.h*

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "FactoryMethod_Main.generated.h"

UCLASS()
class DESIGN_PATTERNS_API AFactoryMethod_Main : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AFactoryMethod_Main();

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

};
```

## ***FactoryMethod\_Main.cpp***

```
#include "FactoryMethod_Main.h"
#include "OuterRealmPotionShop.h"
#include "InnerRealmPotionShop.h"

// Sets default values
AFactoryMethod_Main::AFactoryMethod_Main()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void AFactoryMethod_Main::BeginPlay()
{
    Super::BeginPlay();

    //Create the Shops
    APotionShop* OuterRealmShop = GetWorld()->SpawnActor<AOuterRealmPotionShop>(
        (AOuterRealmPotionShop::StaticClass()));
    APotionShop* InnerRealmShop = GetWorld()->SpawnActor<AIInnerRealmPotionShop>(
        (AIInnerRealmPotionShop::StaticClass()));

    //Create an Outer Health Potion and log its name
    APotion* Potion = OuterRealmShop->OrderPotion("Health");
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
        FString::Printf(TEXT("Potion is %s"), *Potion->GetPotionName()));

    //Create an Inner Health Potion and log its name
    Potion = InnerRealmShop->OrderPotion("Health");
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
        FString::Printf(TEXT("Potion is %s"), *Potion->GetPotionName()));
}
```

```

//Create an Outer Power Potion and log its name
Potion = OuterRealmShop->OrderPotion("Power");
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
FString::Printf(TEXT("Potion is %s"), *Potion->GetPotionName()));

//Create an Inner Health Potion and log its name
Potion = InnerRealmShop->OrderPotion("Power");
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
FString::Printf(TEXT("Potion is %s"), *Potion->GetPotionName()));

//Create an Outer Skill Potion and log its name
Potion = OuterRealmShop->OrderPotion("Skill");
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
FString::Printf(TEXT("Potion is %s"), *Potion->GetPotionName()));

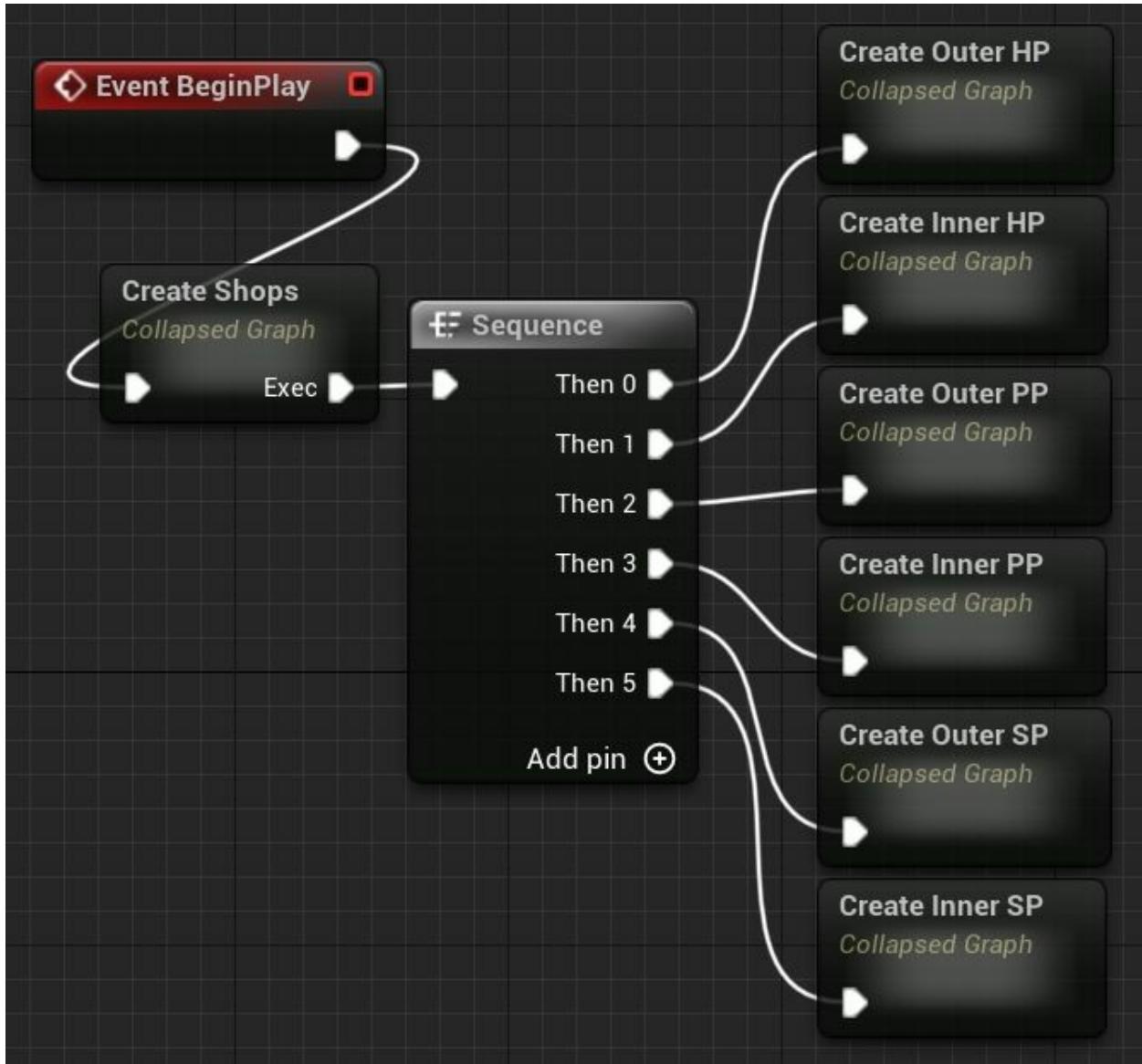
//Create an Inner Health Potion and log its name
Potion = InnerRealmShop->OrderPotion("Skill");
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
FString::Printf(TEXT("Potion is %s"), *Potion->GetPotionName()));
}

// Called every frame
void AFactoryMethod_Main::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

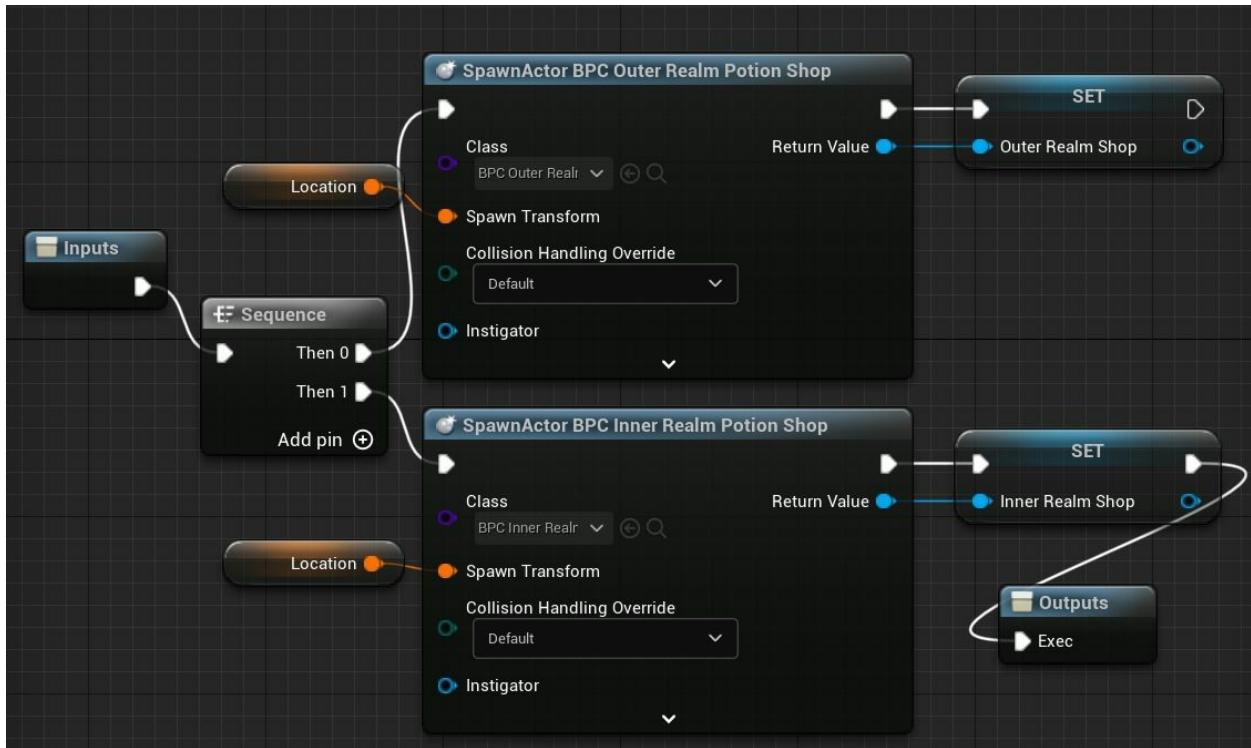
}

```

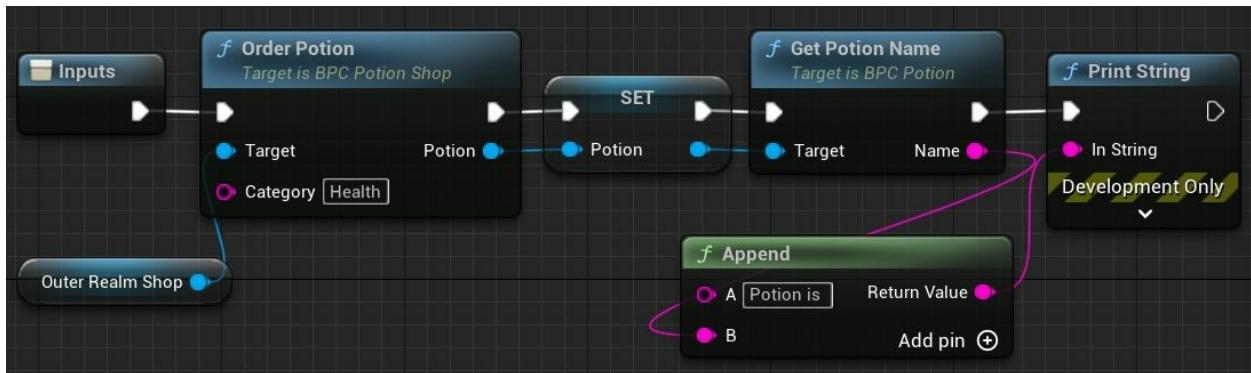
This is the top-level event graph for the *FactoryMethod\_Main* blueprint:



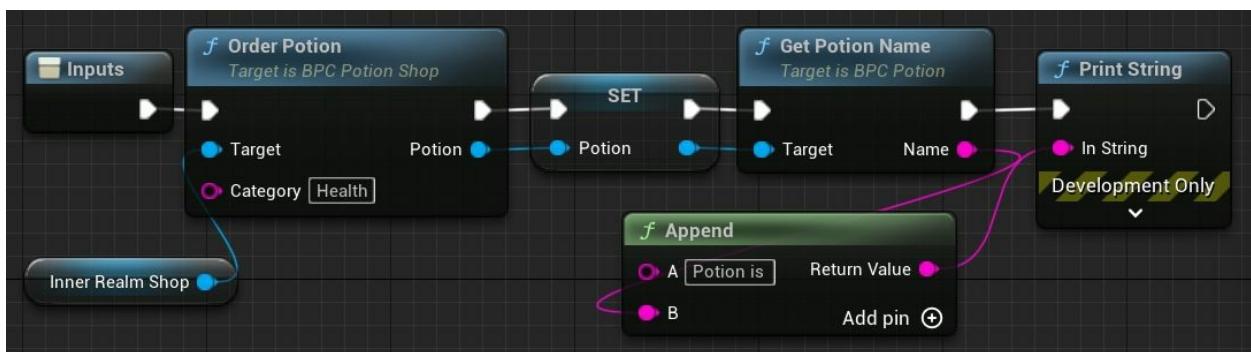
The *Create Shops* collapsed graph:



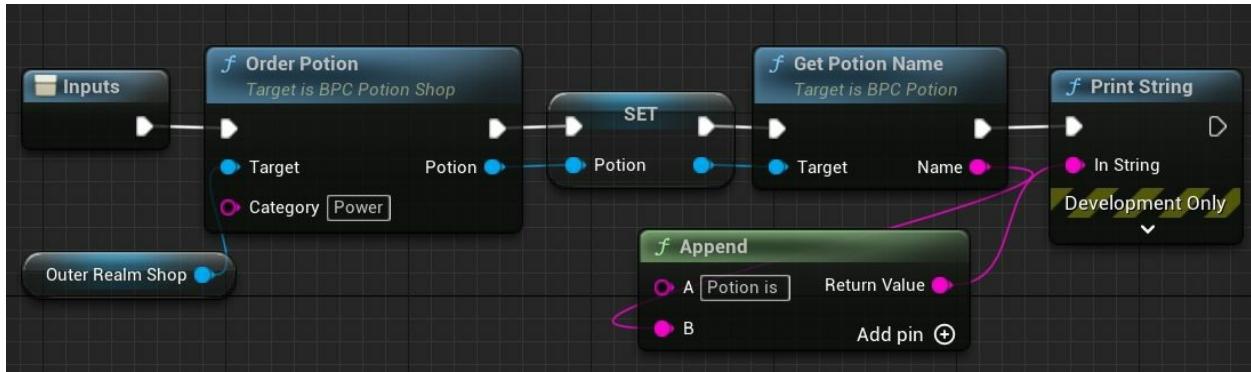
The *Create Outer HP* blueprint:



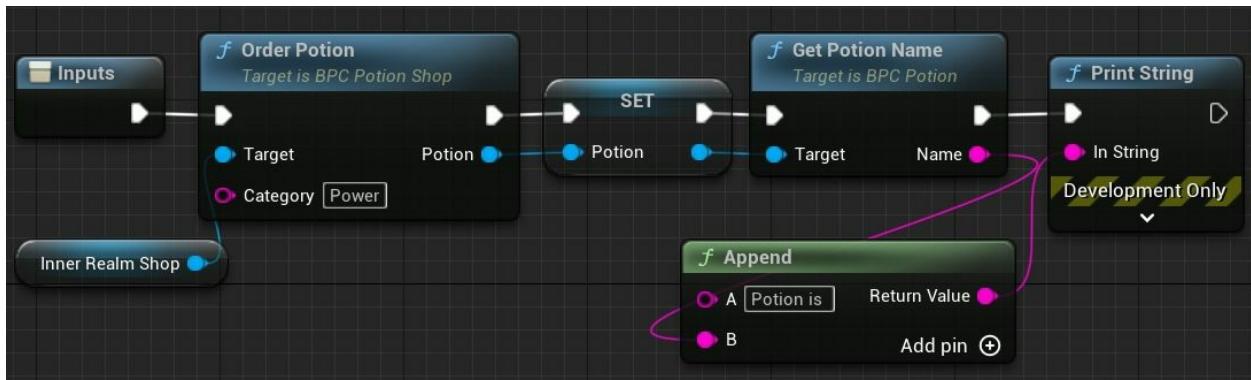
The *Create Inner HP* blueprint:



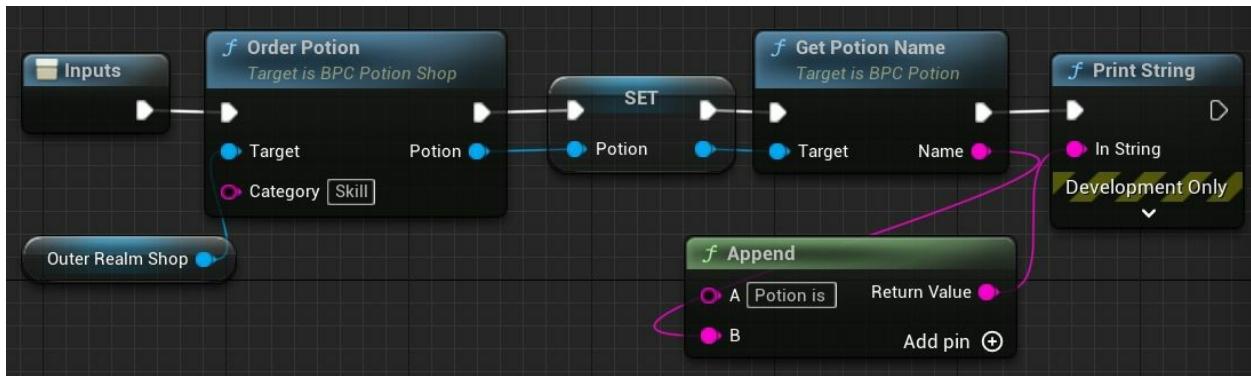
The *Create Outer PP* blueprint:



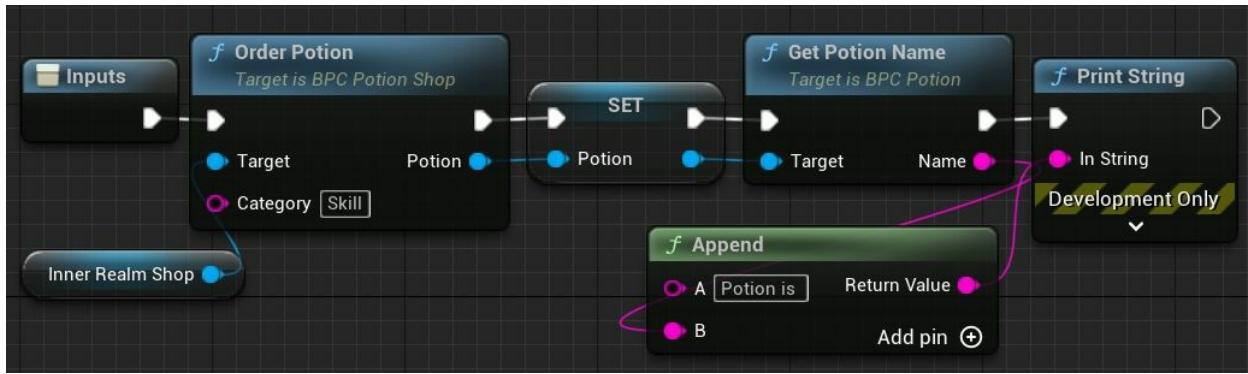
The *Create Inner PP* blueprint:



The *Create Outer SP* blueprint:



The *Create Inner SP* blueprint:



Basically, we are ordering different potions from the Potion Shops and then printed information related to the potion we ordered.

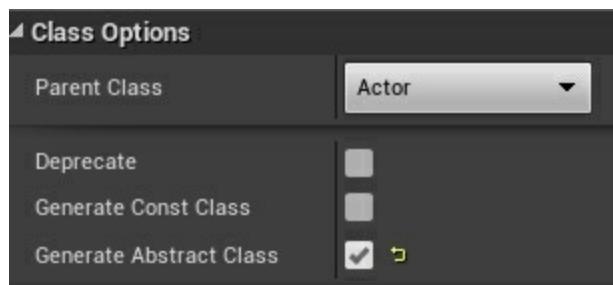
These graphs are equivalent to:

```
Potion = InnerRealmShop->OrderPotion("Health");
```

```
GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
FString::Printf(TEXT("Potion is %s"), *Potion->GetPotionName()));
```

The create inner/outer potion collapsed graphs all contain the same functionality with a different potion being ordered:

This concludes the FactoryMethod\_Main blueprint. Next, we move on to the *Potion* and *PotionShop* classes. Both of these classes are abstract. We can create abstract classes in blueprints by navigating to class settings and checking “Generate Abstract Class.” As pictured here:



Our *PotionShop* class code implementation:

## **PotionShop.h**

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "Potion.h"
#include "PotionShop.generated.h"

UCLASS()
class DESIGN_PATTERNS_API APotionShop : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    APotionShop();

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

    //Create Potion and returns it. It's pure virtual, so it doesn't need an
    implementation in this class
    virtual APotion* ConcoctPotion(FString PotionSKU)
PURE_VIRTUAL(APotionShop::ConcoctPotion, return nullptr;);

    //Order, concoct and returns a Potion of a specific Category
    APotion* OrderPotion(FString Category);

};
```

## **PotionShop.cpp**

```
#include "PotionShop.h"

// Sets default values
APotionShop::APotionShop()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void APotionShop::BeginPlay()
{
    Super::BeginPlay();

}

// Called every frame
void APotionShop::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

}

APotion* APotionShop::OrderPotion(FString Category)
{
    //Create the Potion and log its name
    APotion* Potion = ConcoctPotion(Category);
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
    FString::Printf(TEXT("Concocting %s"), *Potion->GetPotionName()));

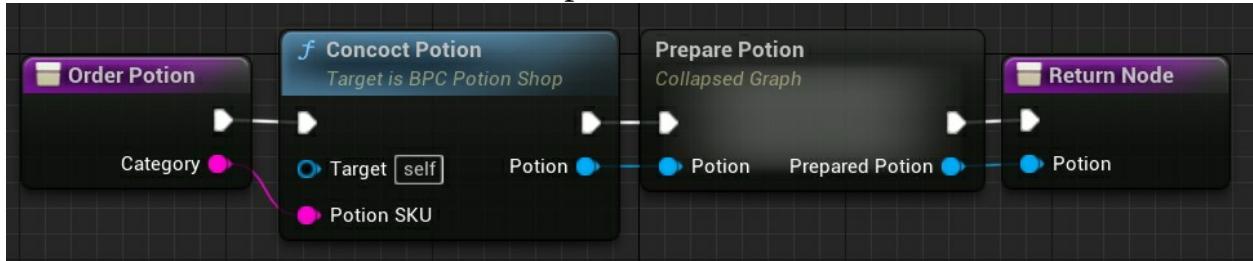
    //Start the concoct process
    Potion->Boil();
    Potion->Bubble();
    Potion->Brew();
```

```
Potion->Bottle();  
//Returns the created potion  
    return Potion;  
}
```

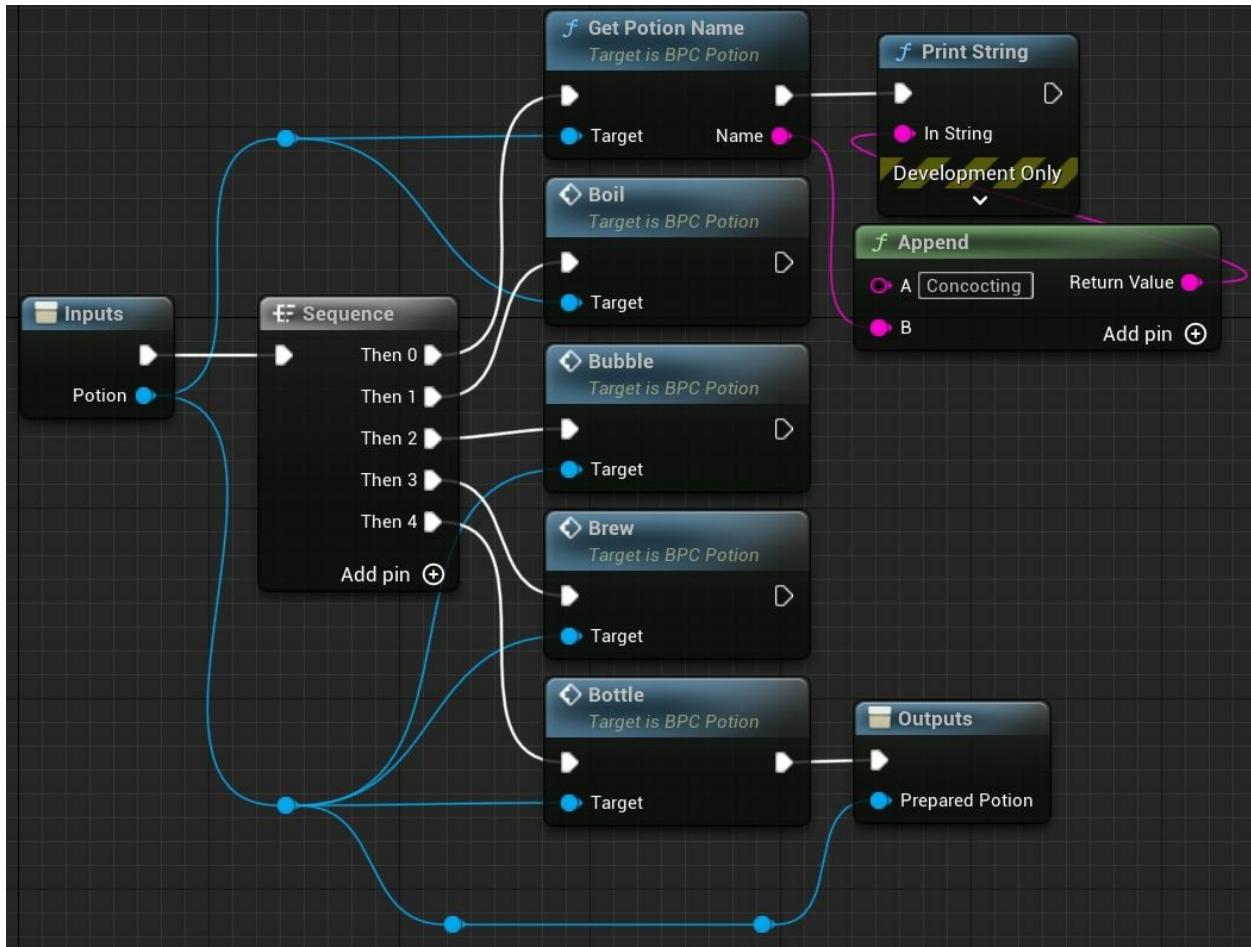
Here we must deviate a bit from traditional code. There does not seem to be a way to explicitly declare a function as “abstract” in blueprints. It is possible that the compiler may be smart enough to do this under the hood, but there is no public information stating this feature. Our *ConcoctPotion* equivalent blueprint:



The *OrderPotion* function blueprint:



The *Prepare Potion* collapsed graph:



The *GetPotionName*, *Boil*, *Bubble*, *Brew* and *Bottle* are in the *Potion* class. Here is the *Potion* class code implementation:

## **Potion.h**

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "Potion.generated.h"

UCLASS()
class DESIGN_PATTERNS_API APotion : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    APotion();

protected:

    //The name of this Potion
    FString PotionName;

    //The Gooeyness of this Potion
    FString Gooeyness;

    //The Blood of this Potion
    FString Blood;

    //The herbs contained in this Potion
    TArray<FString> Herbs;

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;
```

```
//Boild this potion
void Boil();

//Bubble this potion
void Bubble();

//Brew this potion
virtual void Brew();

//Bottle this potion
void Bottle();

//Return the Potion Name
FString GetPotionName();
};
```

## **Potion.cpp**

```
#include "Potion.h"

// Sets default values
APotion::APotion()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

    Herbs = TArray<FString>();

}

// Called when the game starts or when spawned
void APotion::BeginPlay()
{
    Super::BeginPlay();
}

// Called every frame
void APotion::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

}

void APotion::Boil()
{
    //Log the Boil procedure
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
    FString::Printf(TEXT("Boil %s"), *GetPotionName()));
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
    TEXT("Drop in blood..."));
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
    TEXT("Drop in herbs: "));
}
```

```
}

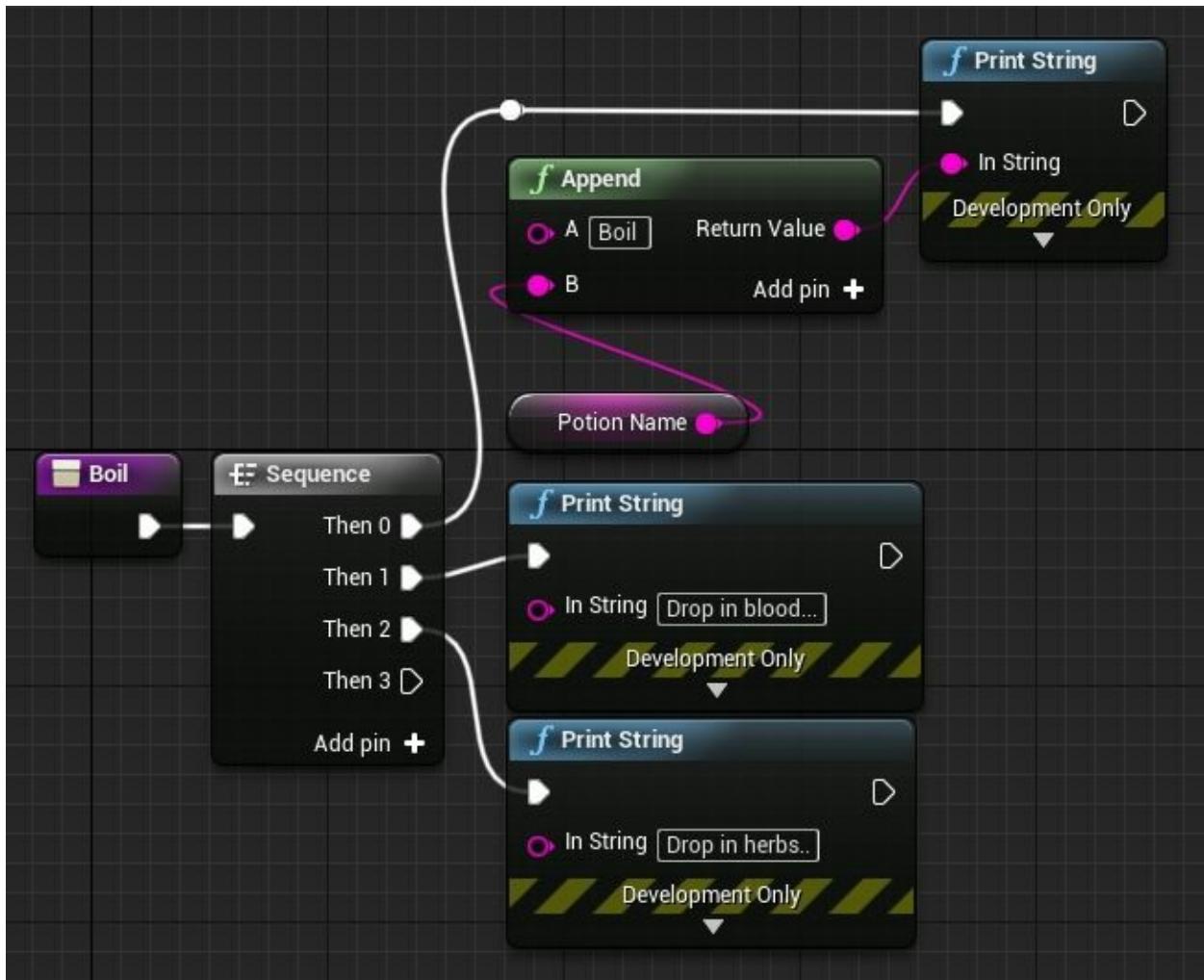
void APotion::Bubble()
{
    //Log the Bubble procedure
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("Bubble for 3 moons and a sunset"));
}

void APotion::Brew()
{
    //Log the Brew procedure
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("Brewing at Low Temperature"));
}

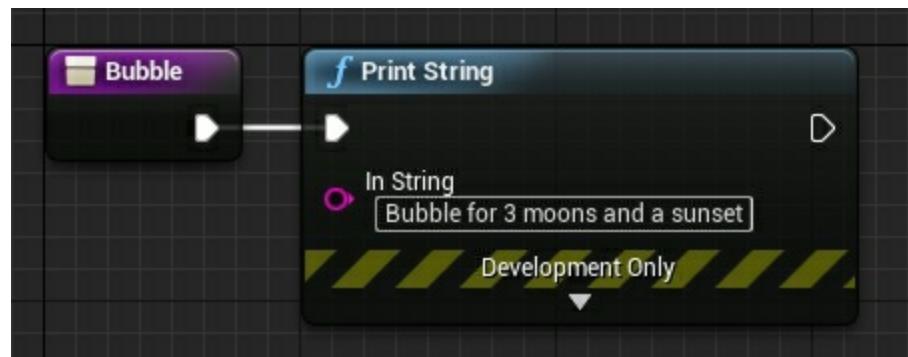
void APotion::Bottle()
{
    //Log the Bottle procedure
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("Bottle concoction in flask"));
}

FString APotion::GetPotionName()
{
    //Return the name of this Potion
    return PotionName;
}
```

The *Boil* method equivalent blueprint function:



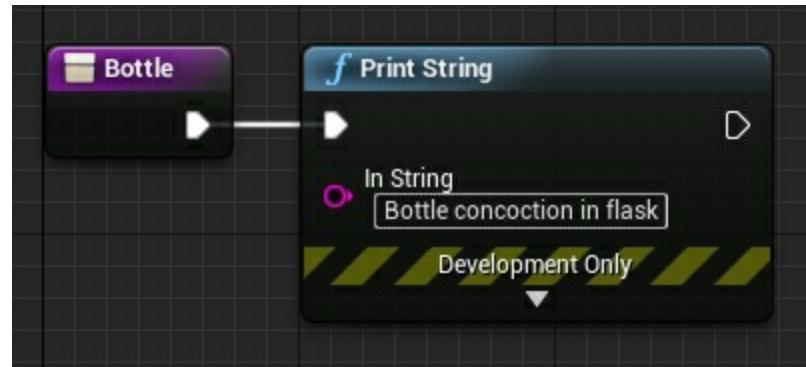
The *Bubble* method equivalent blueprint function:



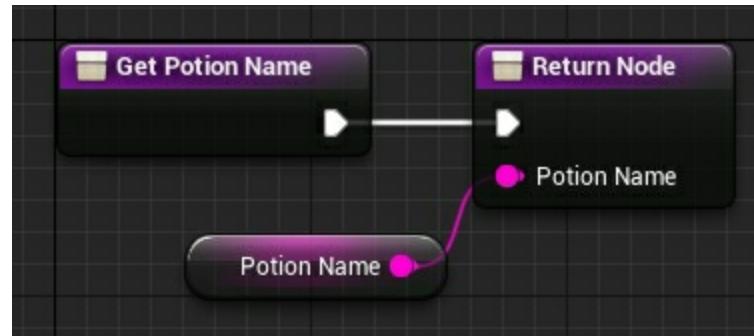
The *Brew* method equivalent blueprint function:



The *Bottle* method equivalent blueprint function:



The *GetPotionName* method equivalent blueprint function:



Now that we have our abstract parent classes defined let's focus on the concrete creator

The *InnerRealmPotionShop* concrete creator class:

## ***InnerRealmPotionShop.h***

```
#pragma once

#include "CoreMinimal.h"
#include "PotionShop.h"
#include "InnerRealmPotionShop.generated.h"

UCLASS()
class DESIGN_PATTERNS_API AInnerRealmPotionShop : public APotionShop
{
    GENERATED_BODY()

public:

    //Concoct the selected potion
    virtual APotion* ConcoctPotion(FString PotionSKU) override;

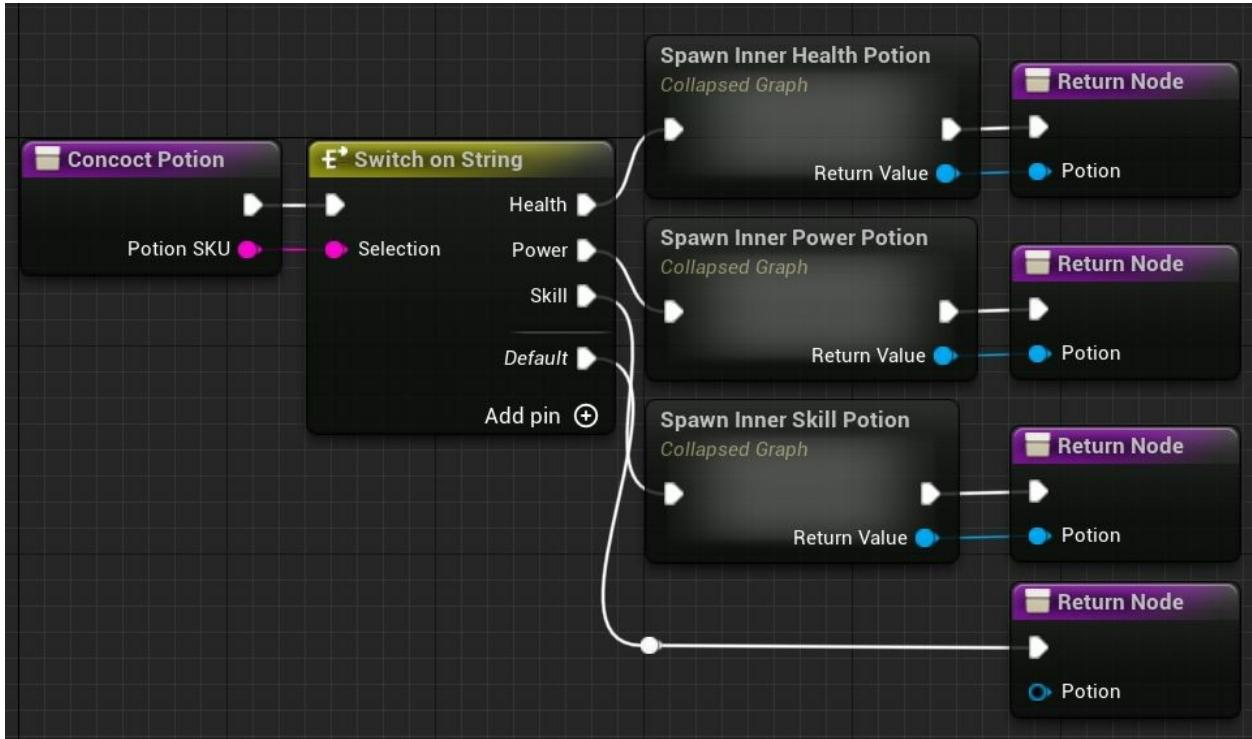
};
```

## ***InnerRealmPotionShop.cpp***

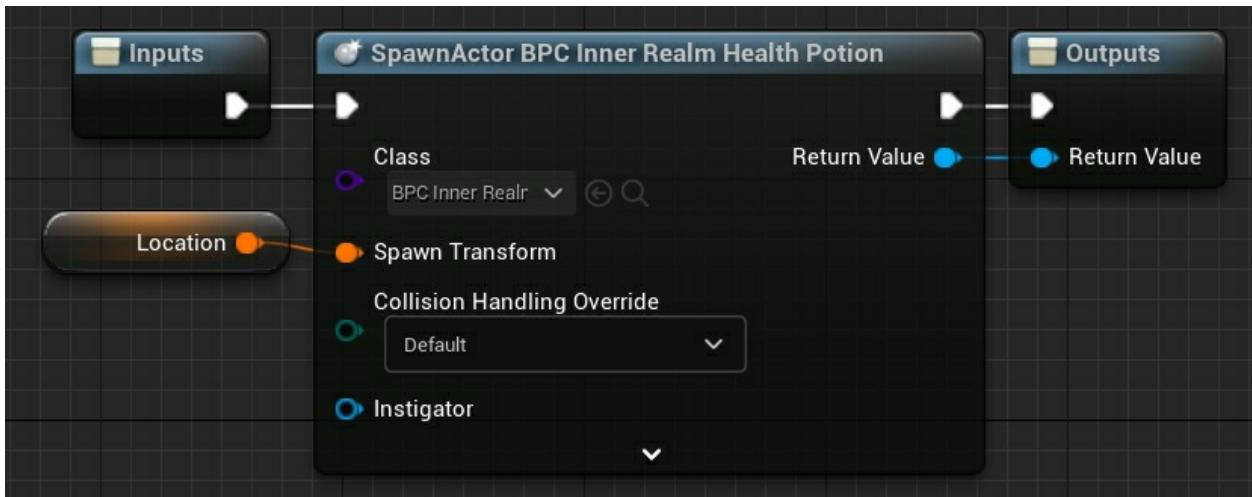
```
#include "InnerRealmPotionShop.h"
#include "InnerRealmHealthPotion.h"
#include "InnerRealmPowerPotion.h"
#include "InnerRealmSkillPotion.h"

APotion* AInnerRealmPotionShop::ConcoctPotion(FString PotionSKU)
{
    //Select which potion to spawn depending on the passed string
    if (PotionSKU.Equals("Health")) {
        return GetWorld()->SpawnActor<AIInnerRealmHealthPotion>
(AInnerRealmHealthPotion::StaticClass());
    }
    else if (PotionSKU.Equals("Power")) {
        return GetWorld()->SpawnActor<AIInnerRealmPowerPotion>
(AInnerRealmPowerPotion::StaticClass());
    }
    else if (PotionSKU.Equals("Skill")) {
        return GetWorld()->SpawnActor<AIInnerRealmSkillPotion>
(AInnerRealmSkillPotion::StaticClass());
    }
    else return nullptr; //Return null if the string isn't valid
}
```

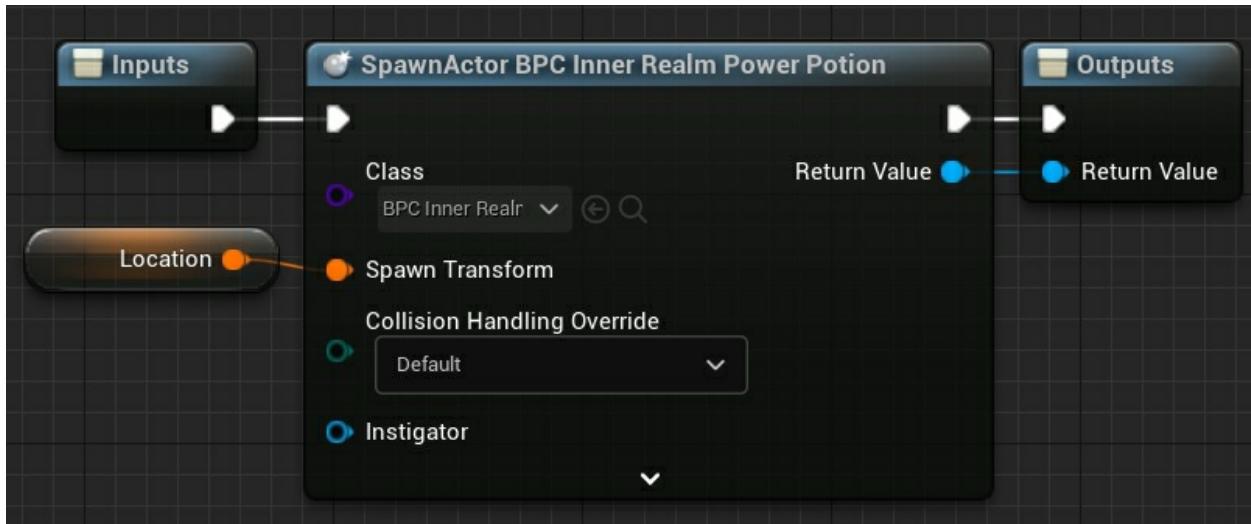
The *InnerRealmPotionShop* blueprint consists of only the *ConcoctPotion* function:



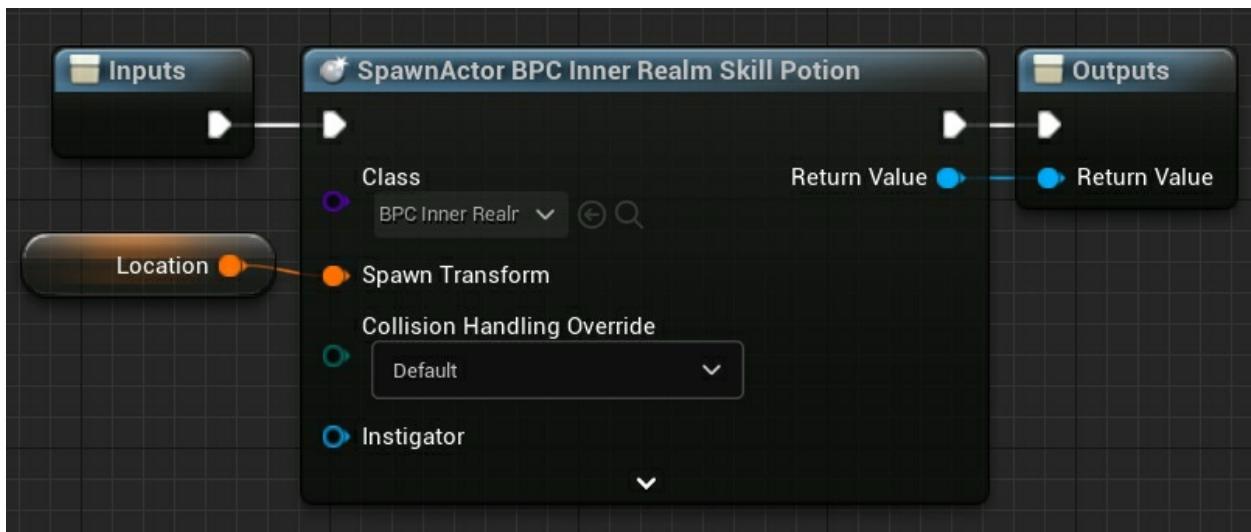
The *Spawn Inner Health Potion* collapsed graph:



The *Spawn Inner Power Potion* collapsed graph:



The *Spawn Inner Skill Potion* collapsed graph:



Now that we have our concrete creator class defined let's focus on the concrete products this creator can create.

The *InnerRealmHealthPotion* concrete product class:

## ***InnerRealmHealthPotion.h***

```
#pragma once

#include "CoreMinimal.h"
#include "Potion.h"
#include "InnerRealmHealthPotion.generated.h"

UCLASS()
class DESIGN_PATTERNS_API AInnerRealmHealthPotion : public APotion
{
    GENERATED_BODY()

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    //Brew the potion
    virtual void Brew() override;

};
```

## ***InnerRealmHealthPotion.cpp***

```
#include "InnerRealmHealthPotion.h"

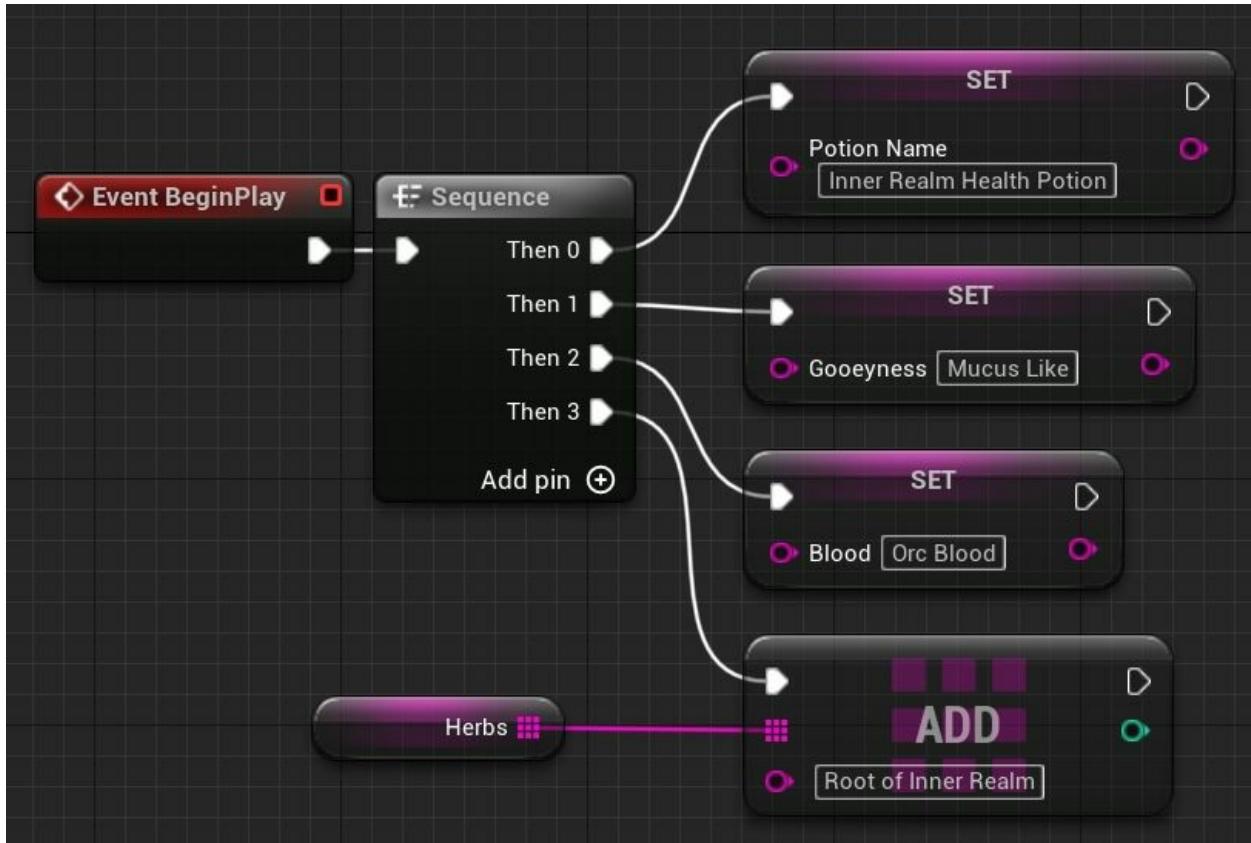
void AInnerRealmHealthPotion::BeginPlay()
{
    Super::BeginPlay();

    //Set the ingredients
    PotionName = "Inner Realm Health Potion";
    Gooeyness = "Mucus Like";
    Blood = "Orc Blood";

    //Add the herbs
    Herbs.Add("Root of Inner Realm");
}

void AInnerRealmHealthPotion::Brew()
{
    //Log the brewing type
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("Brewing at High Temperature"));
}
```

The *InnerRealmHealthPotion* blueprint event graph and a brew event:



The *InnerRealmPowerPotion* concrete product class:

## ***InnerRealmPowerPotion.h***

```
#pragma once

#include "CoreMinimal.h"
#include "Potion.h"
#include "InnerRealmPowerPotion.generated.h"

UCLASS()
class DESIGN_PATTERNS_API AInnerRealmPowerPotion : public APotion
{
    GENERATED_BODY()

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    //Brew the potion
    virtual void Brew() override;
};
```

## ***InnerRealmPowerPotion.cpp***

```
#include "InnerRealmPowerPotion.h"

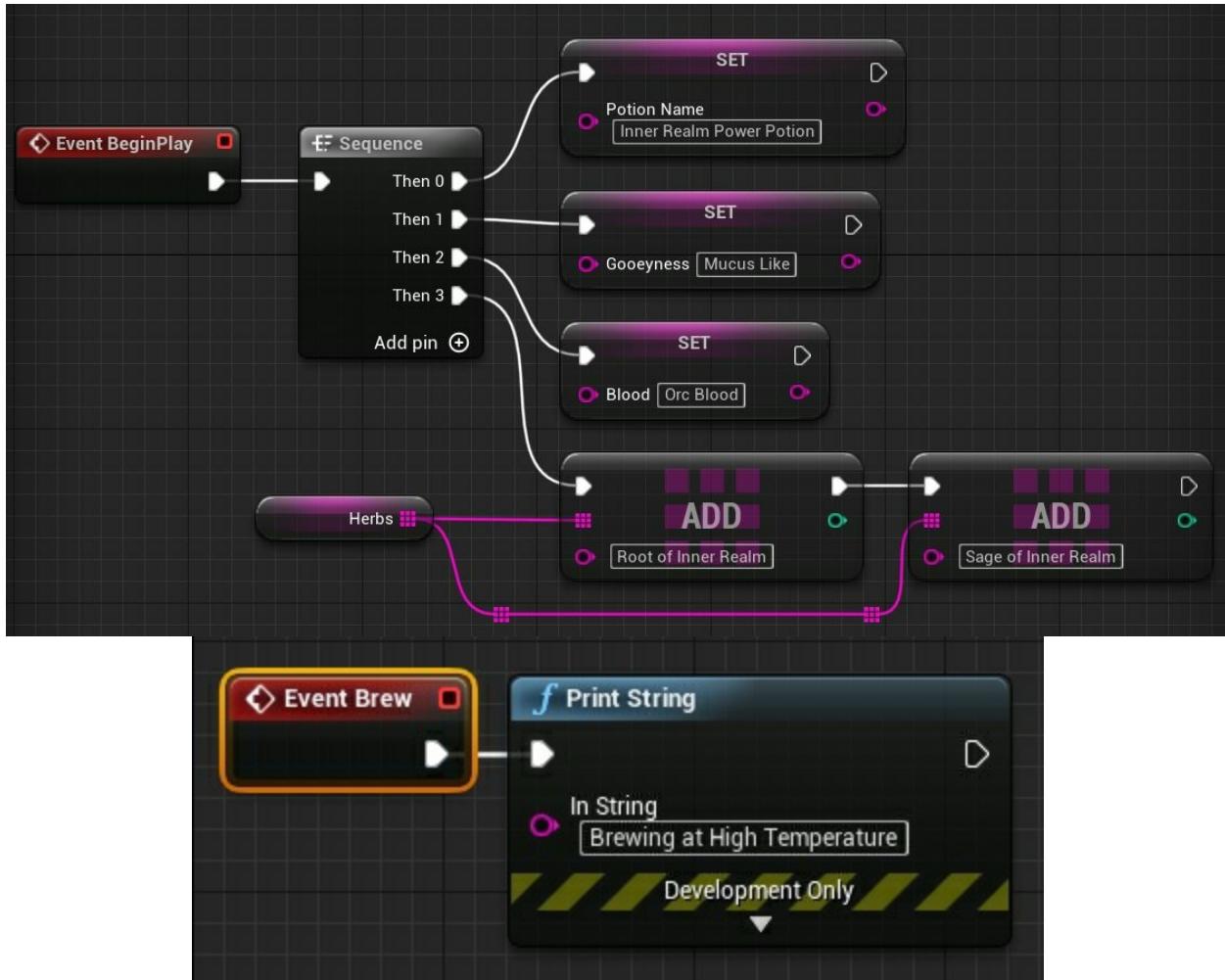
void AInnerRealmPowerPotion::BeginPlay()
{
    Super::BeginPlay();

    //Set the ingredients
    PotionName = "Inner Realm Power Potion";
    Gooeyness = "Mucus Like";
    Blood = "Orc Blood";

    //Add the herbs
    Herbs.Add("Root of Inner Realm");
    Herbs.Add("Sage of Inner Realm");
}

void AInnerRealmPowerPotion::Brew()
{
    //Log the brewing type
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("Brewing at High Temperature"));
}
```

The *InnerRealmPowerPotion* blueprint event graph and a brew event:



The *InnerRealmSkillPotion* concrete product class:

## ***InnerRealmSkillPotion.h***

```
#pragma once

#include "CoreMinimal.h"
#include "Potion.h"
#include "InnerRealmSkillPotion.generated.h"

UCLASS()
class DESIGN_PATTERNS_API AInnerRealmSkillPotion : public APotion
{
    GENERATED_BODY()

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    //Brew the potion
    virtual void Brew() override;

};
```

## ***InnerRealmSkillPotion.cpp***

```
#include "InnerRealmSkillPotion.h"

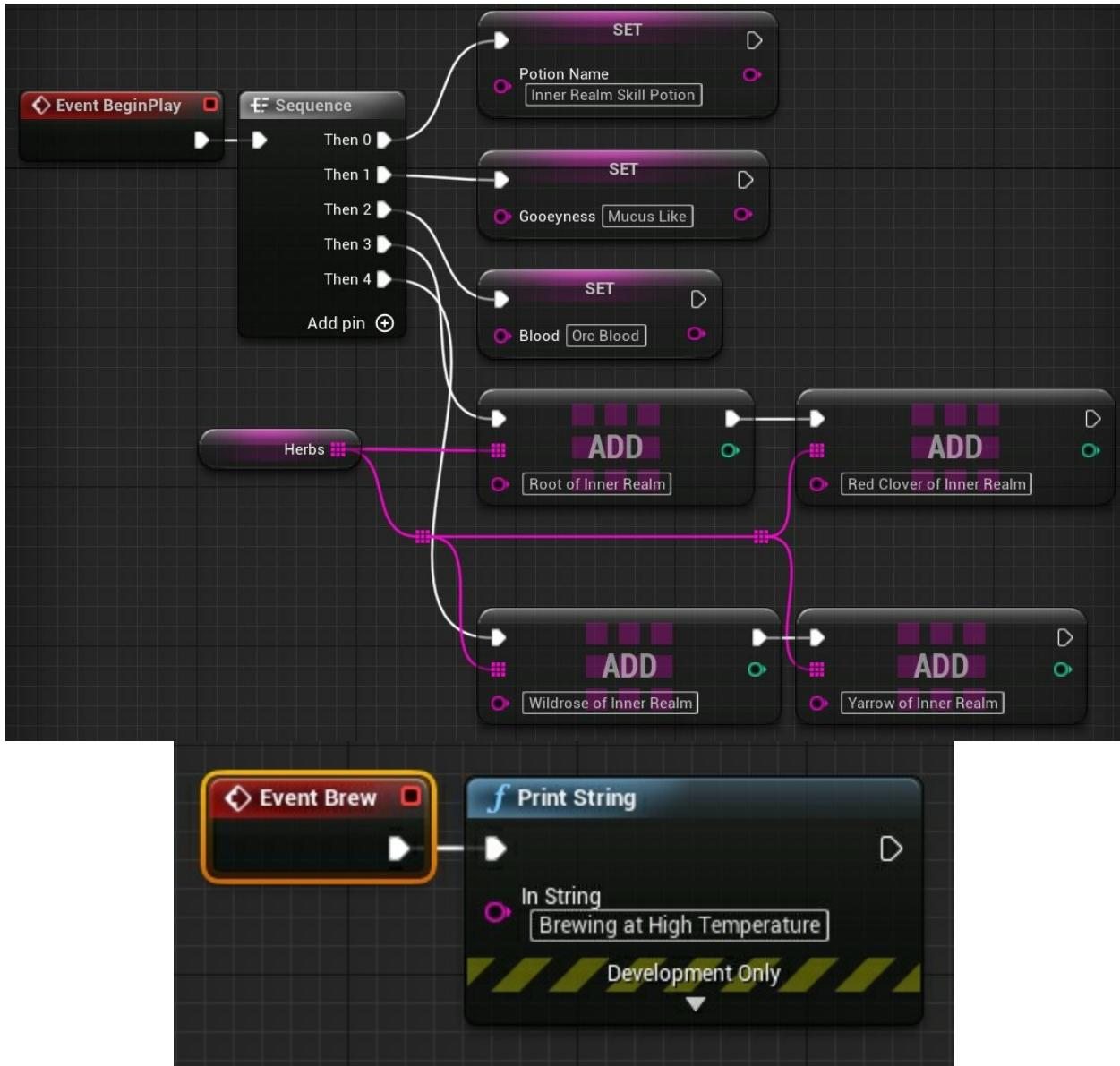
void AInnerRealmSkillPotion::BeginPlay()
{
    Super::BeginPlay();

    //Set the ingredients
    PotionName = "Inner Realm Skill Potion";
    Gooeyness = "Mucus Like";
    Blood = "Orc Blood";

    //Add the herbs
    Herbs.Add("Root of Inner Realm");
    Herbs.Add("Red Clover of Inner Realm");
    Herbs.Add("Wildrose of Inner Realm");
    Herbs.Add("Yarrow of Inner Realm");
}

void AInnerRealmSkillPotion::Brew()
{
    //Log the brewing type
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("Brewing at High Temperature"));
}
```

The *InnerRealmSkillPotion* blueprint event graph and a brew event:



We can easily add another concrete creator class like *OuterRealmPotionShop* and create more concrete products using the same inheritance methods as the *InnerRealmPotionShop*:

The *OuterRealmPotionShop* class:

### ***OuterRealmPotionShop.h***

```
#pragma once
```

```
#include "CoreMinimal.h"
#include "PotionShop.h"
#include "OuterRealmPotionShop.generated.h"
```

```
UCLASS()
```

```
class DESIGN_PATTERNS_API AOuterRealmPotionShop : public
```

```
APotionShop
```

```
{
```

```
GENERATED_BODY()
```

```
public:
```

```
//Concoct the selected potion
```

```
virtual APotion* ConcoctPotion(FString PotionSKU) override;
```

```
};
```

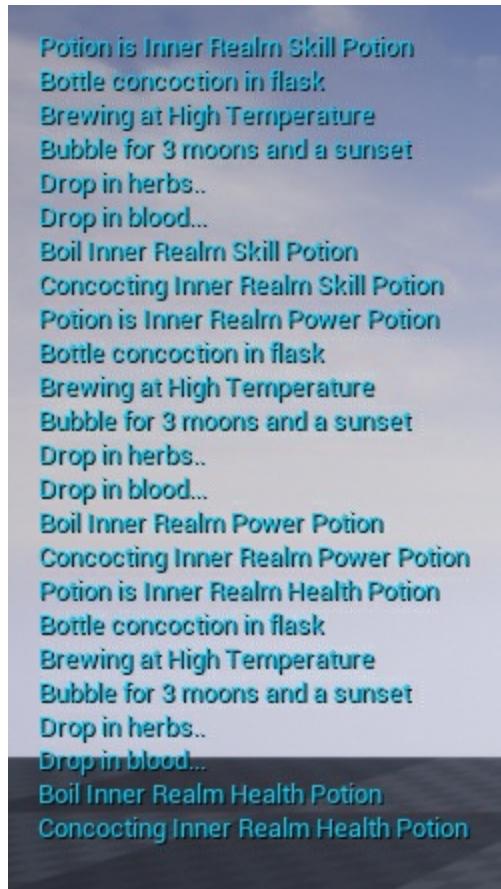
## ***OuterRealmPotionShop.cpp***

```
#include "OuterRealmPotionShop.h"
#include "OuterRealmHealthPotion.h"
#include "OuterRealmPowerPotion.h"
#include "OuterRealmSkillPotion.h"

APotion* AOuterRealmPotionShop::ConcoctPotion(FString PotionSKU)
{
    //Select which potion to spawn depending on the passed string
    if (PotionSKU.Equals("Health")) {
        return GetWorld()->SpawnActor<AOuterRealmHealthPotion>
(AOuterRealmHealthPotion::StaticClass());
    }
    else if (PotionSKU.Equals("Power")) {
        return GetWorld()->SpawnActor<AOuterRealmPowerPotion>
(AOuterRealmPowerPotion::StaticClass());
    }
    else if (PotionSKU.Equals("Skill")) {
        return GetWorld()->SpawnActor<AOuterRealmSkillPotion>
(AOuterRealmSkillPotion::StaticClass());
    }
    else return nullptr; //Return null if the string isn't valid
}
```

We would implement some *OuterRealmPotionShop* concrete products in a similar fashion as the concrete products of the *InnererRealmPotionShop* :

The Factory Method viewport print:



# Programming Elitists' Bane... Singleton Pattern

The Singleton pattern is often singled out for being one of, if not the worst commonly used design pattern. Singletons are like credit cards. Everyone uses them, but you can get into a real predicament if you use them irresponsibly. Programming purists loathe the Singleton pattern. In fact, I have never seen any modern published literature discuss the Singleton pattern without telling the reader avoid its use, ironically. The gang of four describe a Singleton as a class possessing only one instance, and provides a global point of access (Gamma et.al, 1995).

## **The Good**

- Encapsulates sole instance
- Global (ease of access)

## **The Not So Good**

- Global (ease of modification)
- May create unknown dependencies

## **Singleton Pattern Implementation**

There is no clear-cut way to create a custom Singleton blueprint in Unreal. However, it appears the Game Instance is a first party solution to the Singleton. An inventory system is a great example for demonstrating the Singleton design pattern. We only want one inventory instance at all times. Additionally, we want to be able to easily access that inventory system throughout our game.

The main Singleton class *Singleton\_Main*:

## **Singleton\_Main.h**

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "Singleton_Main.generated.h"

UCLASS()
class DESIGN_PATTERNS_API ASingleton_Main : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    ASingleton_Main();

private:
    //The Inventory of this Actor
    UPROPERTY()
        class AInventory* Inventory;

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

};
```

## Singleton\_Main.cpp

```
#include "Singleton_Main.h"
#include "Inventory.h"

// Sets default values
ASingleton_Main::ASingleton_Main()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void ASingleton_Main::BeginPlay()
{
    Super::BeginPlay();

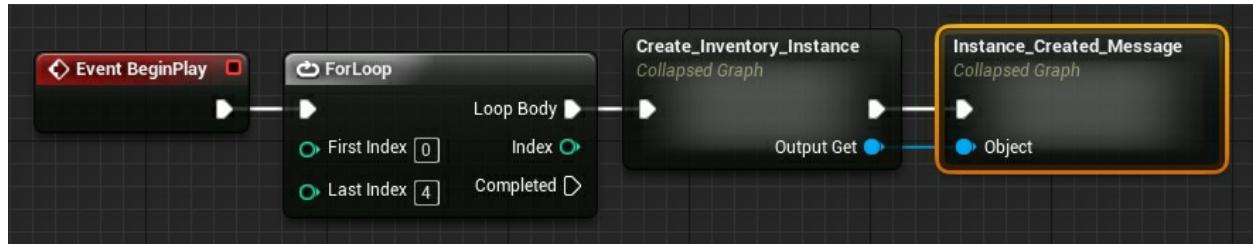
    //Create 4 Inventory
    for (int i = 0; i <= 4; i++)
    {
        AInventory* SpawnerInventory = GetWorld()->SpawnActor<AInventory>(AInventory::StaticClass());
        if (SpawnerInventory)
        {
            //If the Spawn succeeds, set the Spawner inventory to the local one
            //and log the success string
            SpawnerInventory = SpawnerInventory;
            GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
            FString::Printf(TEXT("%s has been created"), *SpawnerInventory->GetName()));
        }
    }
}

// Called every frame
void ASingleton_Main::Tick(float DeltaTime)
{
```

```
Super::Tick(DeltaTime);
```

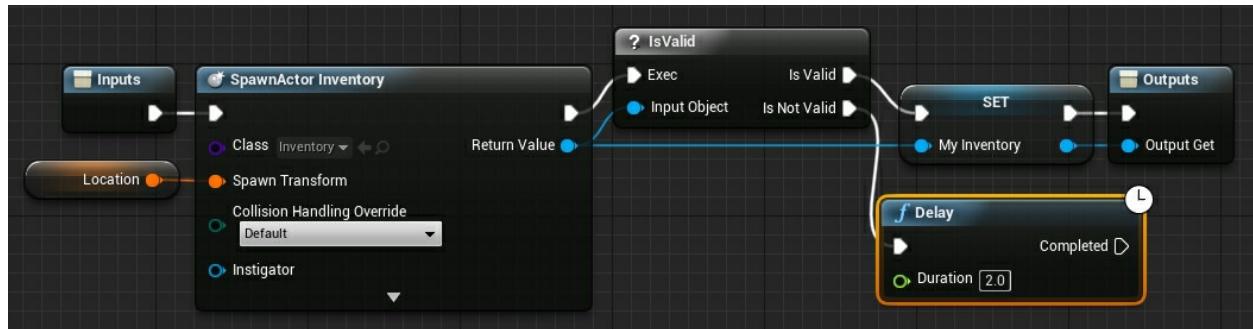
```
}
```

The *Singleton\_Main* blueprint event graph:

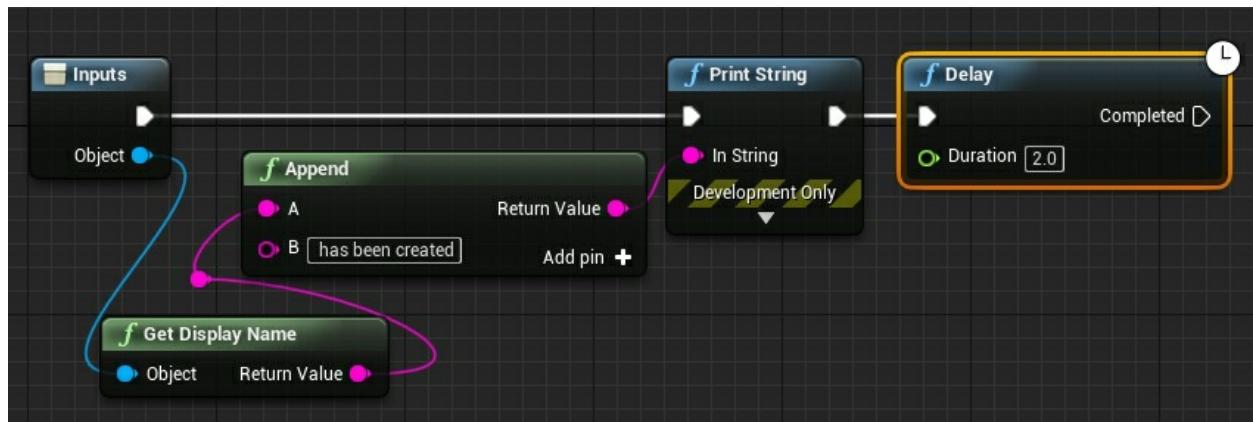


The for loop is present to demonstrate attempts to try to create more than one instance of our designated Singleton object. We should never be allowed to create multiple instances of our inventory If everything is set up correctly.

The *Create\_Inventory\_Instance* collapsed graph:



The *Instance\_Created\_Message* collapsed graph:



The *Inventory* class:

## ***Inventory.h***

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "Inventory.generated.h"

UCLASS()
class DESIGN_PATTERNS_API AInventory : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AInventory();

    //The instance of this Class
    UPROPERTY()
        AInventory* Instance;

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

};
```

## ***Inventory.cpp***

```
#include "Inventory.h"
#include "Kismet/GameplayStatics.h"

// Sets default values
AInventory::AInventory()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

    //Search for existing Instances of this class
    TArray<AActor*> Instances;
    UGameplayStatics::GetAllActorsOfClass(GetWorld(),
AInventory::StaticClass(), Instances);

    if (Instances.Num() > 1)
    {
        //If exist at least one of them, set the instance with the first found one
        Instance = Cast<AInventory>(Instances[0]);
        GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
FString::Printf(TEXT("%s already exists"), *Instance->GetName())));
    }

    //Then Destroy this Actor
    Destroy();
}

// Called when the game starts or when spawned
void AInventory::BeginPlay()
{
    Super::BeginPlay();

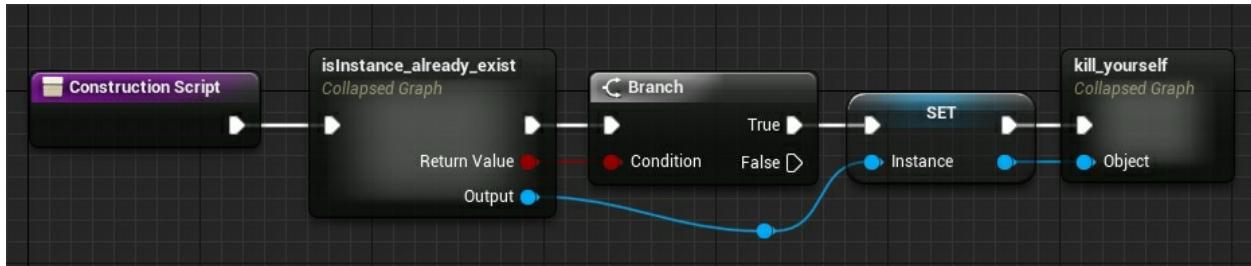
}

// Called every frame
```

```
void AInventory::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

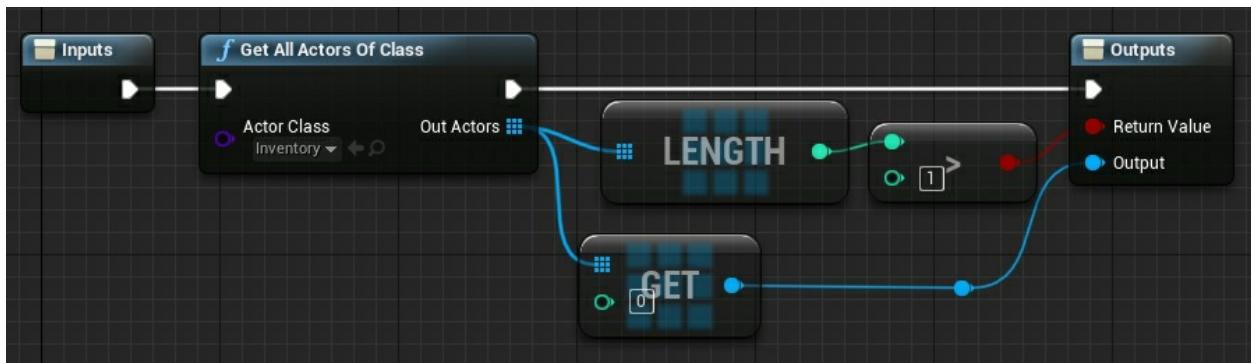
}
```

The *Inventory* blueprint construction script:

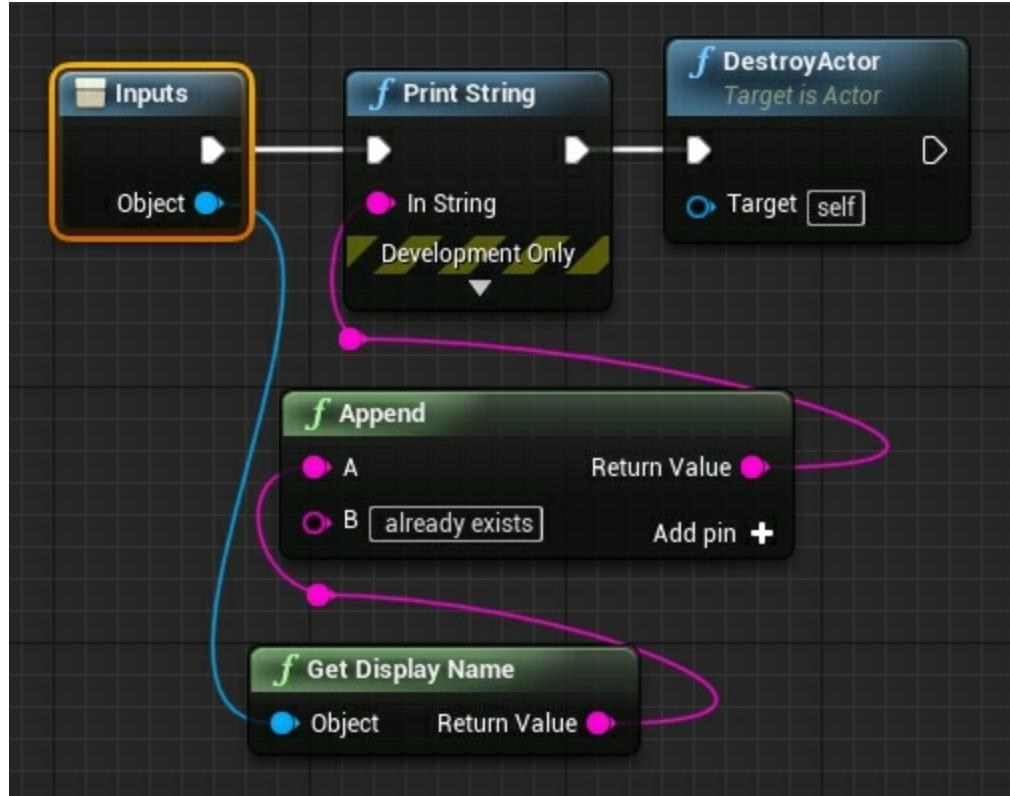


The concept of a private construction does not exist in Unreal. Therefore, we must create another way of keeping new Singleton objects from being created. We do this by checking if an inventory instance already exists. If an inventory instance does in fact already exists, the attempted new Singleton object is destroyed.

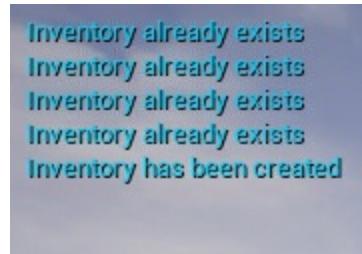
The *isInstance\_already\_exist* collapsed graph:



The *kill\_yourself* collapsed graph:



The Singleton pattern viewport screen print:



# Structural Patterns

The structural patterns we discuss help us to compose larger structures from classes and objects. One the most important goals of this pattern category is to be able to alter object composition at runtime.

# Adapt and Overcome... Adapter Pattern

A large amount of modern technology uses the idea of the wrapper. One can think of high-level programming languages as wrappers around lower-level programming languages. But how do these wrapped languages and their client languages communicate? We need some sort of adapter. GoF tells us the Adapter pattern facilitates the ability for classes to work together that couldn't otherwise because of incompatible interfaces (Gamma et.al, 1995).

## The Good

- Reuse complex code that is otherwise not compatible with a new code base
- Fairly easy to implement

## The Not So Good

- Can be overuse just to get a square peg into a round hole
- Difficulty in recognizing necessity for new implementation or using adapter

## Adapter Pattern Implementation

In our Adapter pattern implementation, we have a shooter who uses a sling shot. We want to be able to allow our shooter to fire a gun. However, we want to use the existing code we have already implemented by using the Adapter pattern.

The main adapter class:

## ***Adapter\_Main.h***

```
#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "Adapter_Main.generated.h"

UCLASS()
class DESIGN_PATTERNS_API AAdapter_Main : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AAdapter_Main();

private:
    //The Shooter Actor that holds the Gun Adapter
    UPROPERTY()
        class AShooter* Shooter;

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

};
```

## ***Adapter\_Main.cpp***

```
#include "Adapter_Main.h"
#include "GunAdapter.h"
#include "Shooter.h"

// Sets default values
AAdapter_Main::AAdapter_Main()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void AAdapter_Main::BeginPlay()
{
    Super::BeginPlay();

    //Spawn the Gun Adapter
    AGunAdapter* GunAdapter = GetWorld()->SpawnActor<AGunAdapter>
    (AGunAdapter::StaticClass());

    //Spawn the Shooter and set the Gun Adapter
    Shooter = GetWorld()->SpawnActor<AShooter>(AShooter::StaticClass());
    Shooter->SetSlingShot(GunAdapter);

    //Shoot
    Shooter->Sling();
}

// Called every frame
void AAdapter_Main::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);
}
```

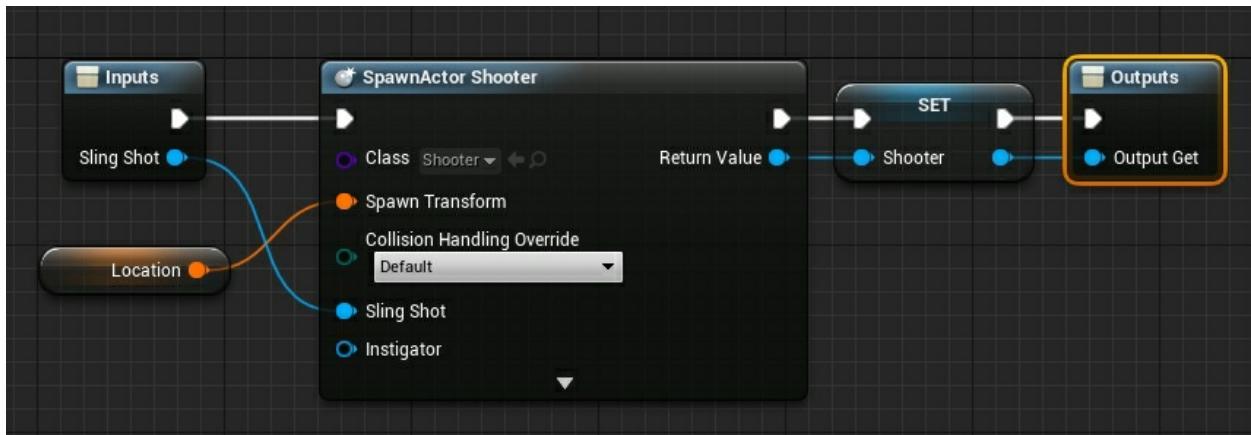
The *Adapter\_Main* blueprint:



The *Create\_Gun\_Adapter* collapsed graph:



The *Create\_Shooter* collapsed graph:



The *SlingShot* interface:

## *SlingShot.h*

```
#pragma once

#include "CoreMinimal.h"
#include "UObject/Interface.h"
#include "SlingShot.generated.h"

// This class does not need to be modified.
UINTERFACE(MinimalAPI)
class USlingShot : public UInterface
{
    GENERATED_BODY()
};

class DESIGN_PATTERNS_API ISlingShot
{
    GENERATED_BODY()

    // Add interface functions to this class. This is the class that will be
    // inherited to implement this interface.
public:

    //The pure virtual function of the SlingShot
    virtual void Sling() = 0;

};
```

## *SlingShot.cpp*

```
#include "SlingShot.h"

// Add default functionality here for any ISlingShot functions that are not
// pure virtual.

    SlingShot blueprint interface is omitted for brevity because it's just a
single function. The GunAdapter class:
```

## ***GunAdapter.h***

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "SlingShot.h"
#include "GunAdapter.generated.h"

class AGun;

UCLASS()
class DESIGN_PATTERNS_API AGunAdapter : public AActor, public
ISlingShot
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AGunAdapter();

private:
    //The Weapon Actor
    UPROPERTY();
    AGun* Weapon;

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

    //Fire the Weapon
    void Sling() override;

};
```

## ***GunAdapter.cpp***

```
#include "GunAdapter.h"
#include "Gun.h"

// Sets default values
AGunAdapter::AGunAdapter()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void AGunAdapter::BeginPlay()
{
    Super::BeginPlay();

    //Spawn the Weapon
    Weapon = GetWorld()->SpawnActor<AGun>(AGun::StaticClass());
}

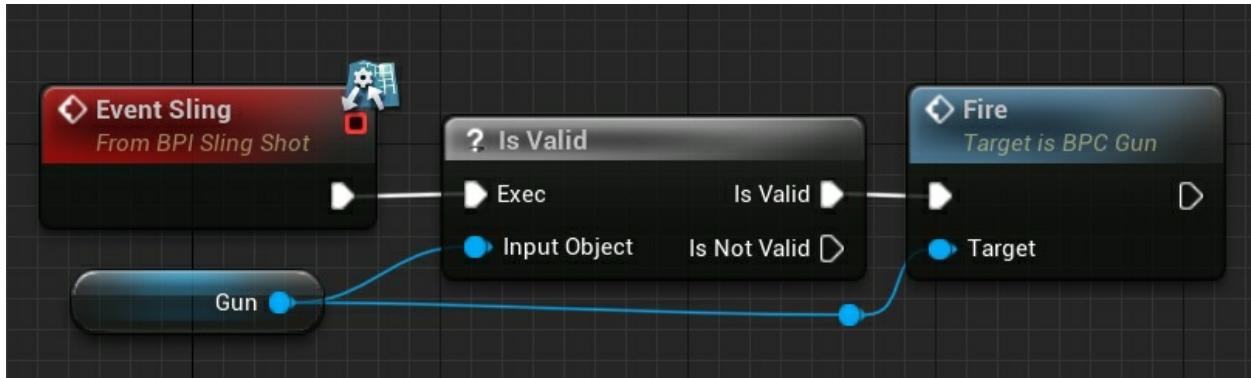
// Called every frame
void AGunAdapter::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

}

void AGunAdapter::Sling()
{
    if (!Weapon) { UE_LOG(LogTemp, Error, TEXT("Sling(): Weapon is
NULL, make sure it's initialized.")); return; }

    //Call the Fire function
    Weapon->Fire();
}
```

The *GunAdapter* blueprint:



The *Gun* class:

## ***Gun.h***

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "Gun.generated.h"

UCLASS()
class DESIGN_PATTERNS_API AGun : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AGun();

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

    //Fire with the gun
    void Fire();

};
```

## ***Gun.cpp***

```
#include "Gun.h"

// Sets default values
AGun::AGun()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void AGun::BeginPlay()
{
    Super::BeginPlay();

}

// Called every frame
void AGun::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

}

void AGun::Fire()
{
    //Print Fire log
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
    TEXT("Our gun is firing"));
}
```

The *Gun* blueprint:



The *Shooter* class:

## ***Shooter.h***

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "SlingShot.h"
#include "Shooter.generated.h"

UCLASS()
class DESIGN_PATTERNS_API AShooter : public AActor, public
ISlingShot
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AShooter();

private:
    //The Weapon of the Shooter, that must be a SlingShot
    ISlingShot* SlingShot;

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

    //Set the Weapon Actor
    void SetSlingShot(AActor* SlingShotObj);

    //Fire with the SlingShot
    void Sling();

};
```

## *Shooter.cpp*

```
#include "Shooter.h"

// Sets default values
AShooter::AShooter()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void AShooter::BeginPlay()
{
    Super::BeginPlay();

}

// Called every frame
void AShooter::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

}

void AShooter::SetSlingShot(AActor* SlingShotObj)
{
    //Cast the passed Actor and set the Weapon
    SlingShot = Cast<ISlingShot>(SlingShotObj);

    if (!SlingShot) //Log Error if cast fails
    {
        GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Red,
        TEXT("Invalid Cast! See Output log for more details"));
        UE_LOG(LogTemp, Error, TEXT("SetSlingShot(): The Actor is not a
        SlingShot! Are you sure that the Actor implements that interface?"));
    }
}
```

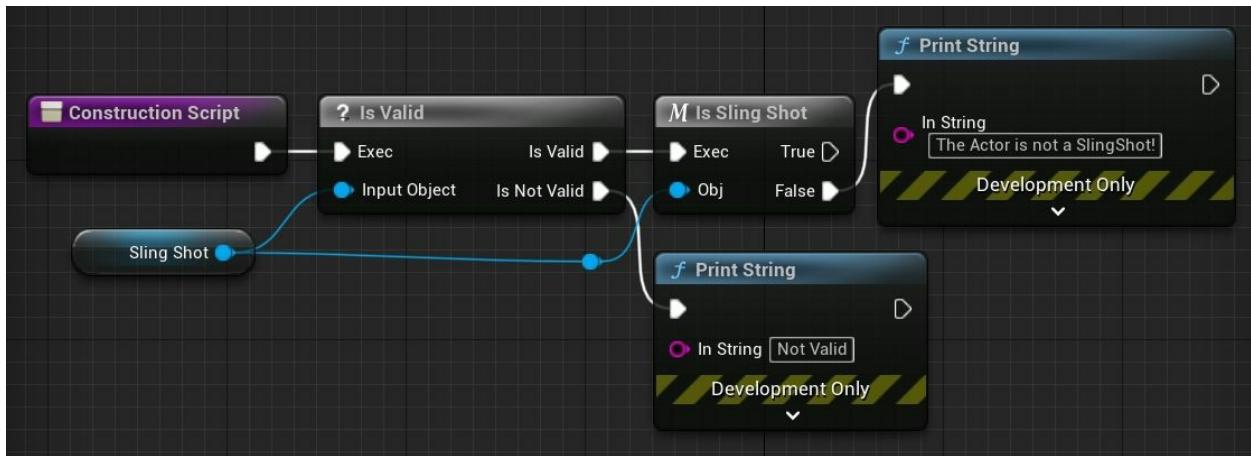
```
        }
    }

void AShooter::Sling()
{
    if (!SlingShot) { UE_LOG(LogTemp, Error, TEXT("Sling(): SlingShot is
NULL, make sure it's initialized.")); return; }

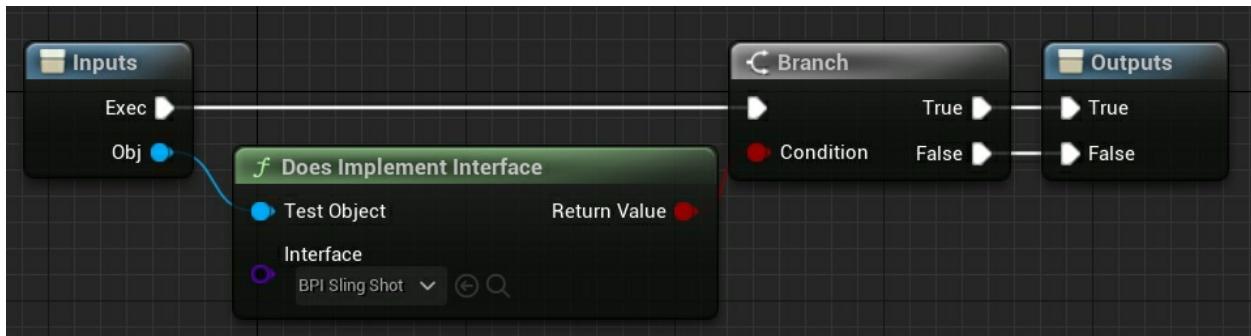
    //Fire
    SlingShot->Sling();
}
```

We can see that the *Shooter* class has no functionality to fire a gun. However, it does have functionality to sling a slingshot. Both skills are predicated on emitting a projectile from a weapon. We would like to reuse the slingshot functionality and adapt it for the use of a gun.

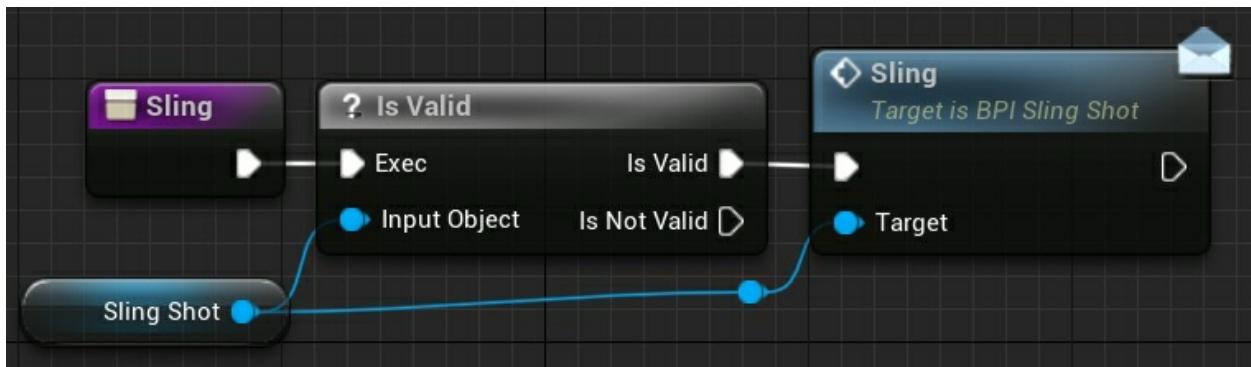
The *Shooter* construction script:



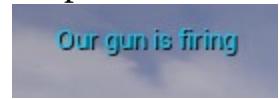
The Is Sling Shot Macro:



The *Sling* function:



The Adapter pattern viewport print:



# Decorating Overpowered Enemies...

## Decorator Pattern

The Decorator pattern allows for the possibility of adding features and functionality to objects at runtime. GoF states that, “Decorators provide a flexible alternative to sub-classing for extending functionality” (Gamma et.al, 1995).

### **The Good**

- Easily add functionality to individual objects
- Avoids affecting objects of the same class

### **The Not So Good**

- May end up creating what GoF calls “Lots of little objects”. Basically, all these little semi-identical objects may become difficult to debug.

## **Decorator Pattern Implementation**

Our implementation consists of an enemy who is initially a melee enemy. We want to “Decorate” the enemy and make an enemy who can use projectiles. Thus, the added functionality is the projectile ability.

The *Decorator\_Main* class:

## ***Decorator\_Main.h***

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "Enemy.h"
#include "Decorator_Main.generated.h"

UCLASS()
class DESIGN_PATTERNS_API ADecorator_Main : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    ADecorator_Main();

public:
    //The main Enemy Actor
    IEnemy* Enemy;

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

};
```

## ***Decorator\_Main.cpp***

```
#include "Decorator_Main.h"
#include "ConcreteEnemy.h"
#include "MeleeEnemy.h"
#include "ProjectileEnemy.h"

// Sets default values
ADecorator_Main::ADecorator_Main()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void ADecorator_Main::BeginPlay()
{
    Super::BeginPlay();

    //Spawn a Concrete Enemy
    AConcreteEnemy*    ConcreteEnemy    =    GetWorld()->SpawnActor<AConcreteEnemy>(AConcreteEnemy::StaticClass());
    //Spawn a Melee Enemy and set its Enemy to the Concrete one
    AMeleeEnemy*    MeleeEnemy    =    GetWorld()->SpawnActor<AMeleeEnemy>(AMeleeEnemy::StaticClass());
    MeleeEnemy->SetEnemy(ConcreteEnemy);

    //Spawn a Projectile Enemy and set its Enemy to the Melee one
    AProjectileEnemy*    ProjectileEnemy    =    GetWorld()->SpawnActor<AProjectileEnemy>(AProjectileEnemy::StaticClass());
    ProjectileEnemy->SetEnemy(MeleeEnemy);

    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("Melee Enemies are on the horizon"));
    Enemy = MeleeEnemy;
```

```

        Enemy->Fight();
        GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
FString::Printf(TEXT("Melee Enemies cause %i damage."), Enemy-
>GetDamage())));
        Enemy->Die();

        GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("Enemies are now armed with guns")));
        Enemy = ProjectileEnemy;
        Enemy->Fight();
        GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
FString::Printf(TEXT("Projectile Enemies cause %i damage."), Enemy-
>GetDamage())));
        Enemy->Die();

    }

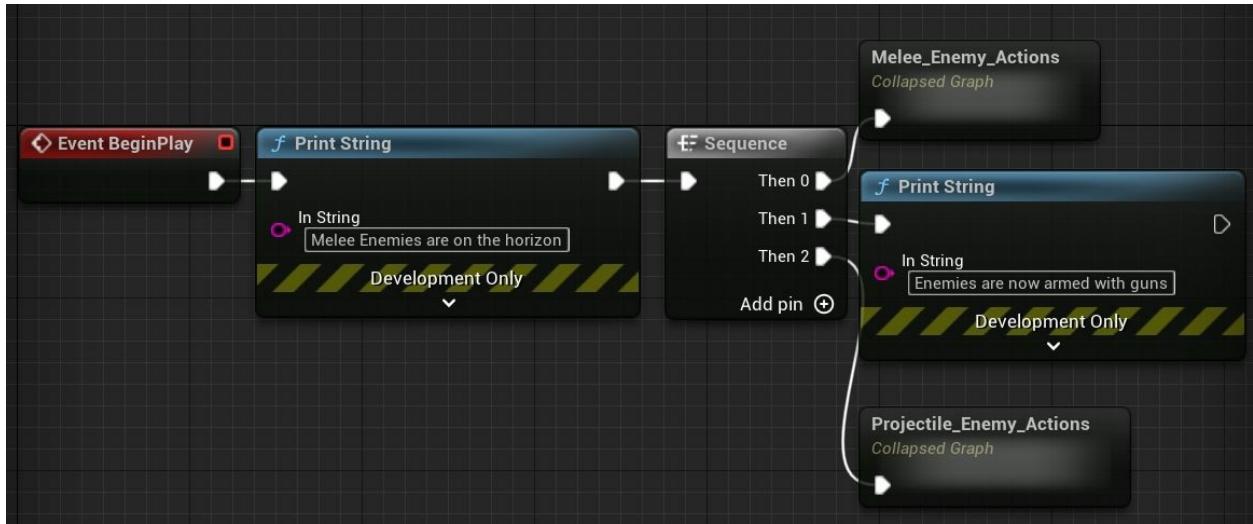
// Called every frame
void ADecorator_Main::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

}

```

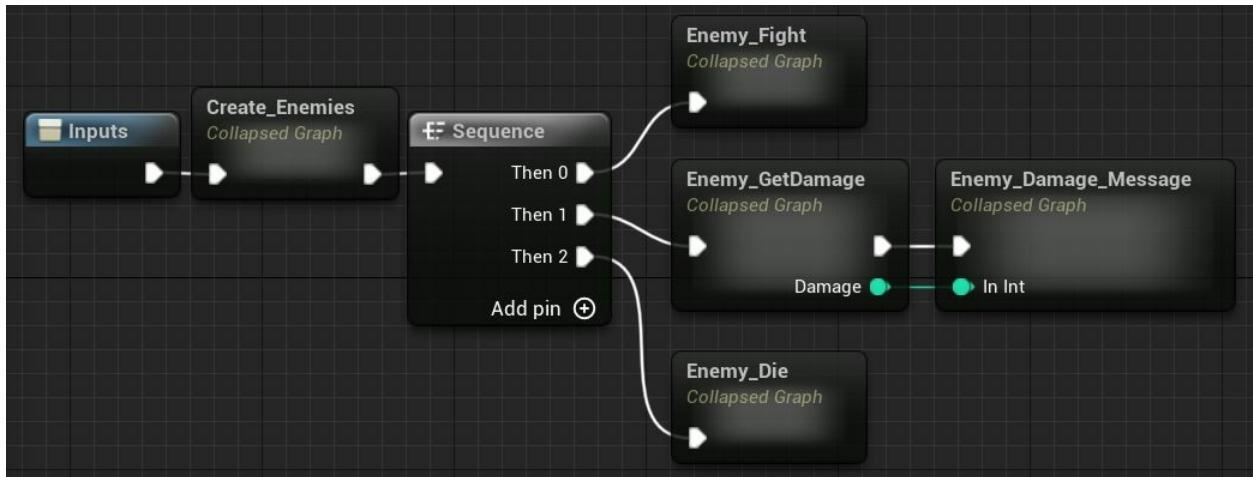
We can see from the *Decorator\_Main* that we do not need to make a separate *Enemy* class to add features. We merely “Decorate” the enemy, and now we have a projectile enemy. Effectively, we changed the behavior of the enemy at runtime by using a decorator.

The *Decorator\_Main* blueprint event graph:

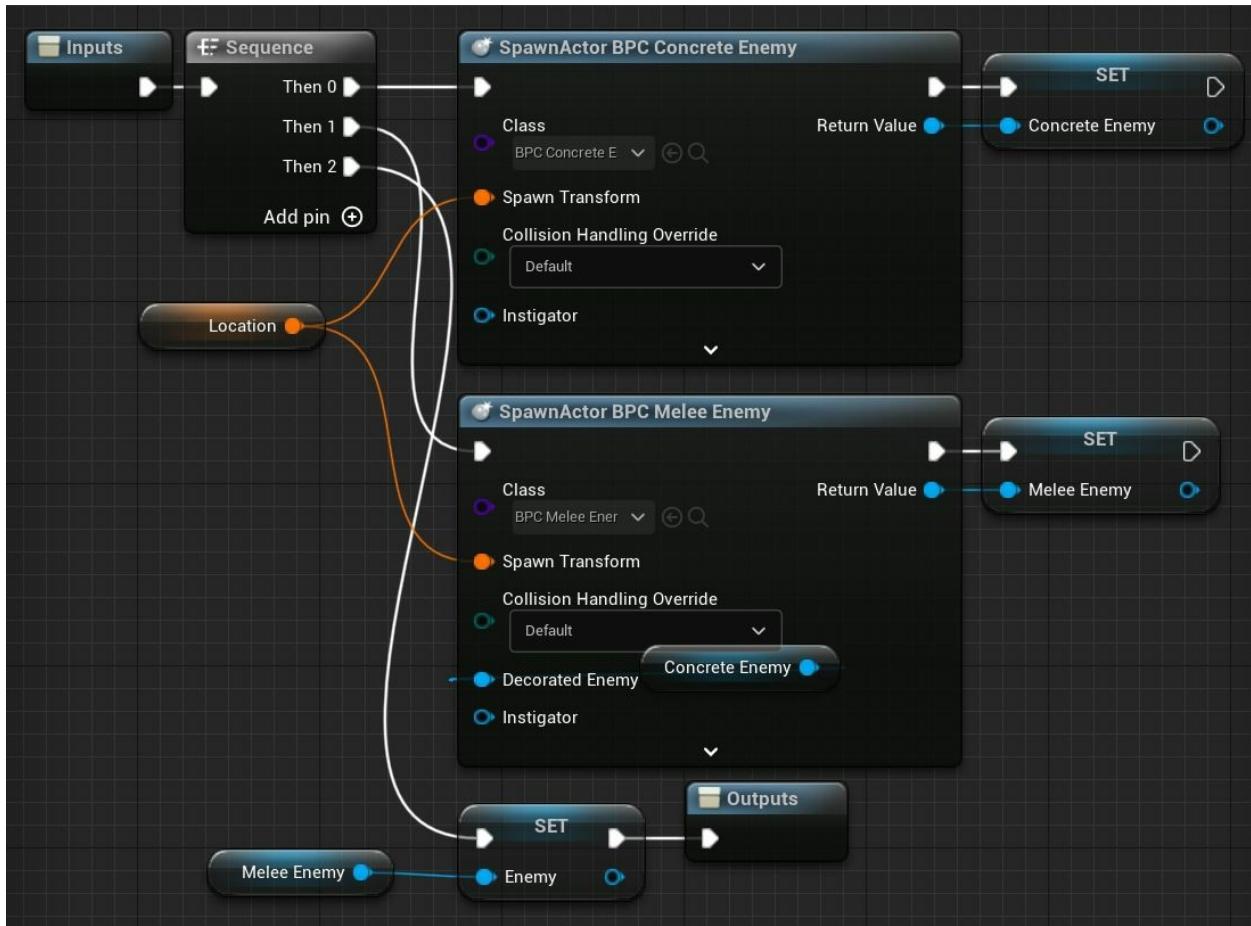


The melee enemy and the decorated projectile enemy are broken down into separate collapsed graphs for readability.

The *Melee\_Enemy\_Actions* collapsed graph:



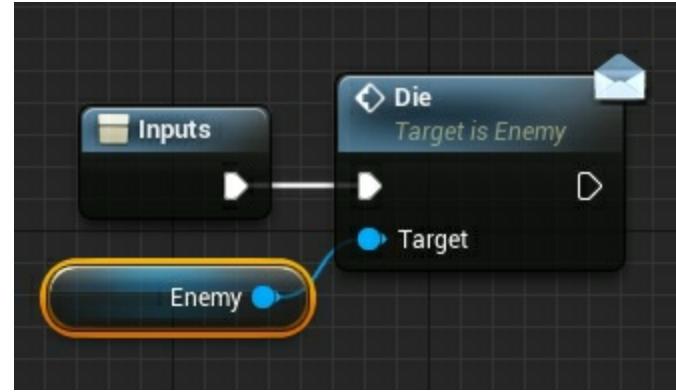
The *Create\_Enemy* collapsed graph:



The *Enemy\_Fight* collapsed graph:



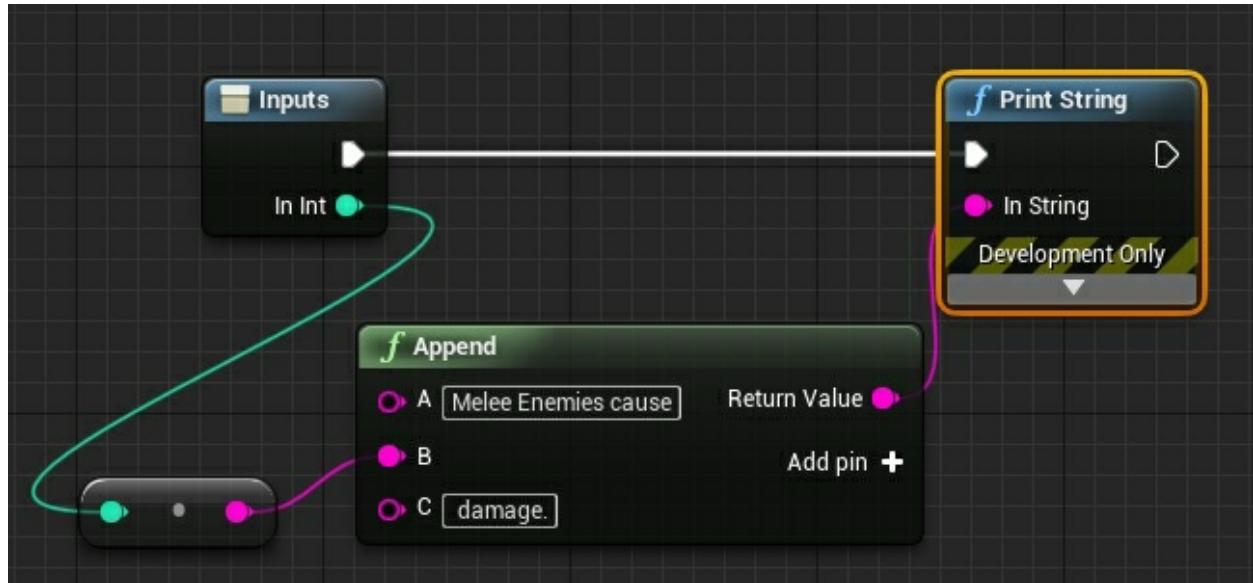
The *Enemy\_Die* collapsed graph:



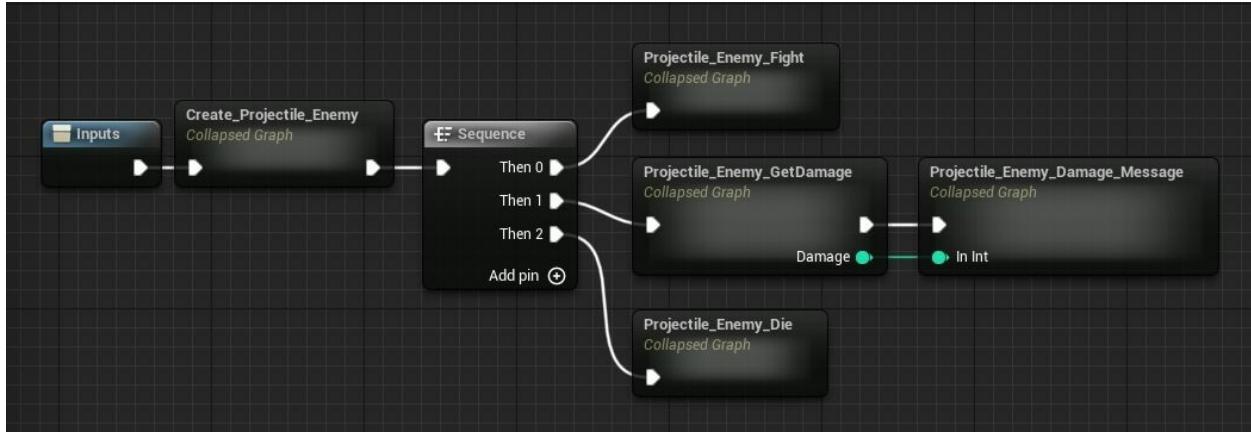
The `Enemy_Die` collapsed graph:



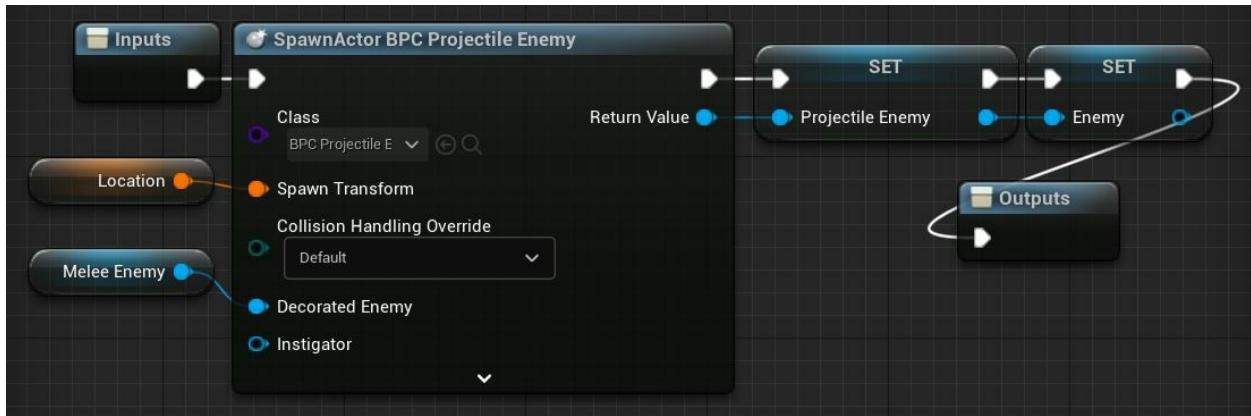
The `Enemy_Damage_Message` collapsed graph:



The `Projectile_Enemy_Actions` collapsed graph:



The *Create\_Projectile\_Enemy* collapsed graph:



The notable difference between the melee enemy and the projectile enemy is construction. The projectile enemy constructor accepts the previously created enemy as an argument. This is so we can reuse it to decorate an enemy object.

The *Projectile\_Enemy\_Fight* collapsed graph:



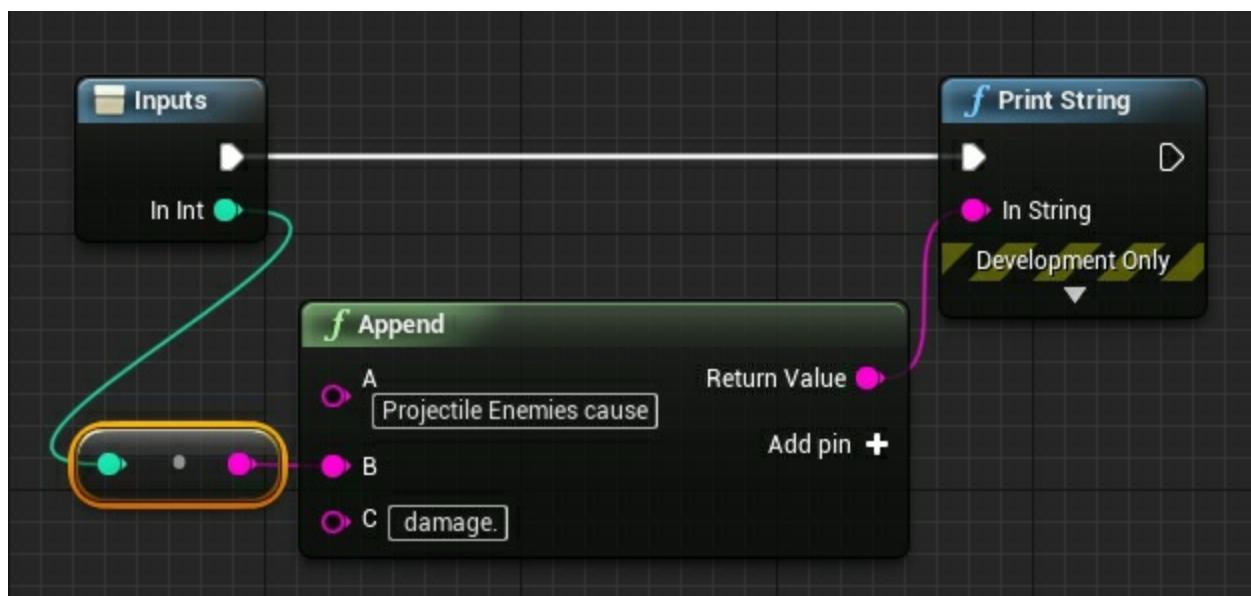
The *Projectile\_Enemy\_Die* collapsed graph:



The *Projectile\_Enemy\_GetDamage* collapsed graph:



The *Projectile\_Enemy\_Damage\_Message* collapsed graph:



The *Enemy* Interface:

## ***Enemy.h***

```
#pragma once

#include "CoreMinimal.h"
#include "UObject/Interface.h"
#include "Enemy.generated.h"

// This class does not need to be modified.
UINTERFACE(MinimalAPI)
class UEnemy : public UInterface
{
    GENERATED_BODY()
};

/*
 * Component
 * defines an interface for objects that can have responsibilities
 * added to them dynamically
 */
class DESIGN_PATTERNS_API IEnemy
{
    GENERATED_BODY()

    // Add interface functions to this class. This is the class that will be
    // inherited to implement this interface.
public:

    //The pure virtual functions of the Enemy
    virtual void Fight() = 0;
    virtual int GetDamage() = 0;
    virtual void Die() = 0;

};
```

## ***Enemy.cpp***

```
#include "Enemy.h"

// Add default functionality here for any IEnemy functions that are not pure
virtual.
```

The *Enemy* interface is shared across all enemy objects. The *MeleeEnemy* class:

## ***MeleeEnemy.h***

```
#pragma once

#include "CoreMinimal.h"
#include "Decorator.h"
#include "MeleeEnemy.generated.h"

/*
 * Concrete Decorators
 * add responsibilities to the component (can extend the state
 * of the component)
 */
UCLASS()
class DESIGN_PATTERNS_API AMeleeEnemy : public ADecorator
{
    GENERATED_BODY()

public:

    //Start Fighting
    virtual void Fight() override;

    //Returns how much damage this enemy has taken
    virtual int GetDamage() override;

    //Kill this enemy
    virtual void Die() override;
};
```

## ***MeleeEnemy.cpp***

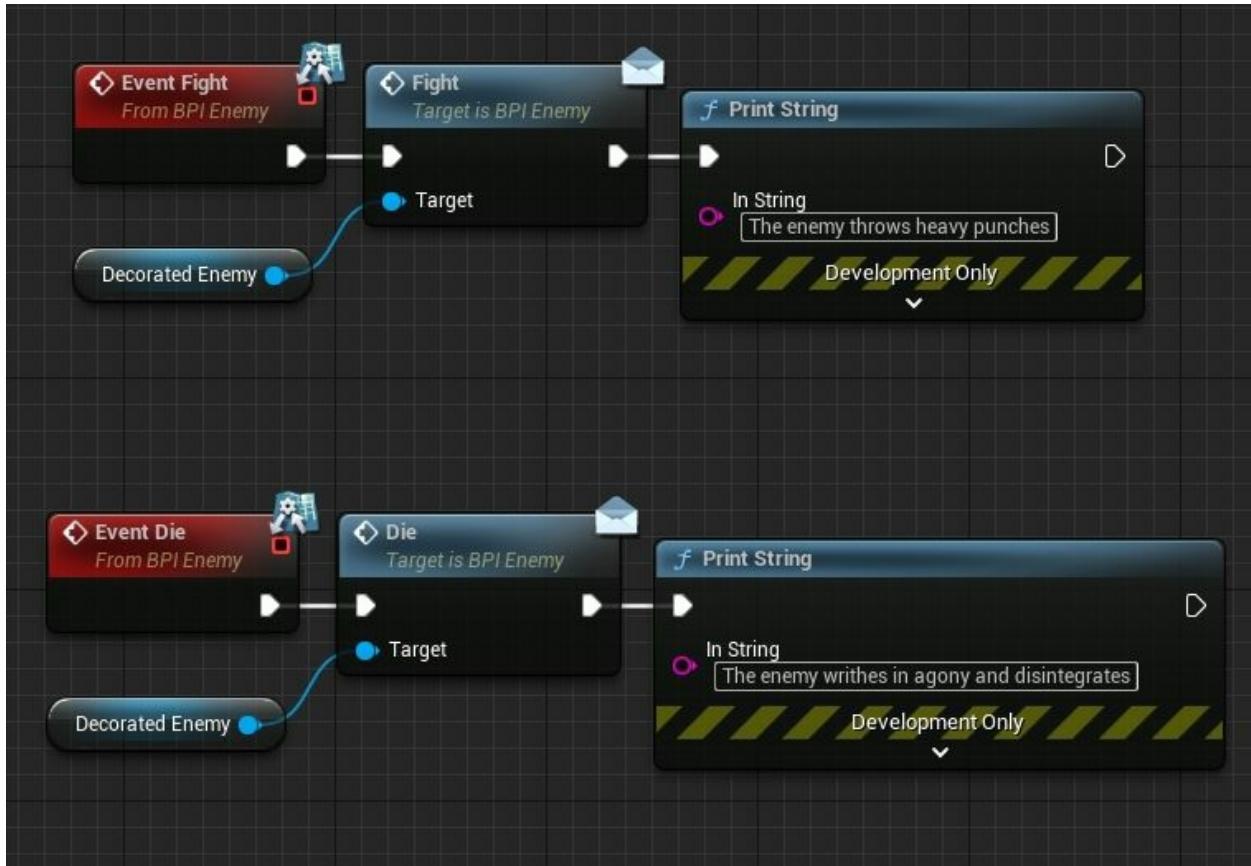
```
#include "MeleeEnemy.h"

void AMeleeEnemy::Fight()
{
    //Call the parent Fight function and log a message
    Super::Fight();
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("The enemy throws heavy punches"));
}

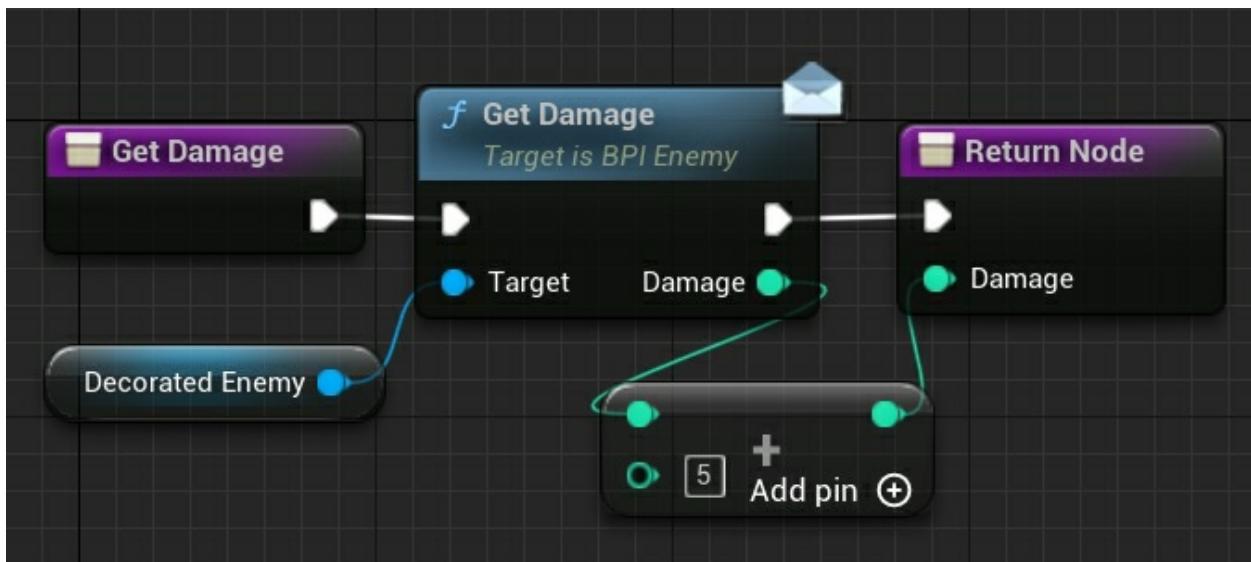
int AMeleeEnemy::GetDamage()
{
    //Returns the base Damage + 5
    return Super::GetDamage() + 5;
}

void AMeleeEnemy::Die()
{
    //Call the parent Die function and log a message
    Super::Die();
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("The enemy writhes in agony and disintegrates"));
}
```

The *MeleeEnemy* blueprint event graph:



The *GetDamage* function in the *MeleeEnemy* blueprint:



The *ProjectileEnemy* class:

## ***ProjectileEnemy.h***

```
#pragma once

#include "CoreMinimal.h"
#include "Decorator.h"
#include "ProjectileEnemy.generated.h"

UCLASS()
class DESIGN_PATTERNS_API AProjectileEnemy : public ADecorator
{
    GENERATED_BODY()

public:

    //Start Fighting
    virtual void Fight() override;

    //Returns how much damage this enemy has taken
    virtual int GetDamage() override;

    //Kill this enemy
    virtual void Die() override;
};
```

## **ProjectileEnemy.cpp**

```
#include "ProjectileEnemy.h"

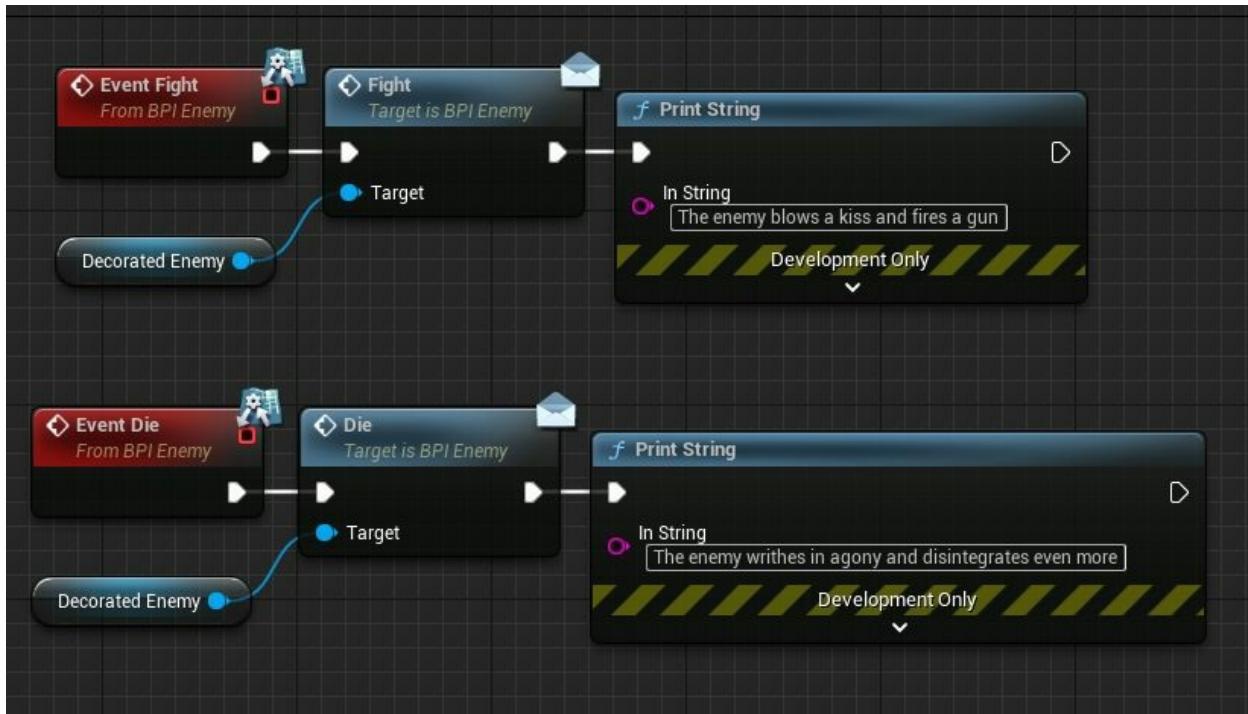
void AProjectileEnemy::Fight()
{
    //Call the parent Fight function and log a message
    Super::Fight();
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
    TEXT("The enemy blows a kiss and fires a gun"));
}

int AProjectileEnemy::GetDamage()
{
    //Returns the base Damage + 95
    return Super::GetDamage() + 95;
}

void AProjectileEnemy::Die()
{
    //Call the parent Die function and log a message
    Super::Die();
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
    TEXT("The enemy writhes in agony and disintegrates even more"));
}
```

We can see that the projectile enemy causes more damage than the melee enemy. We use the melee damage causing attribute and we increase it by 95 for the projectile enemy.

The *ProjectileEnemy* blueprint event graph:



The *GetDamage* function in the *ProjectileEnemy* blueprint:



Decorator Pattern viewport print:

Projectile Enemies cause 100 damage.  
The enemy writhes in agony and disintegrates  
The enemy blows a kiss and fires a gun  
The enemy throws heavy punches  
MeleeEnemies Implements Enemy  
Melee Enemies cause 5 damage.  
The enemy writhes in agony and disintegrates  
The enemy throws heavy punches  
Melee Enemies are on the horizon

We can see the melee enemies coming. The melee enemy fights us with punches. The enemy then dies. However, we take a 5-count damage from the melee enemy. The melee enemy is then given new functionality in the form of a projectile. The enemy can punch and shoot a gun causing damage of 100.

# Space Travel Looks Easy... Façade Pattern

GoF defines the Façade pattern as providing a central interface to a set of interfaces (Gamma et.al, 1995). The Façade pattern is great when we want to simplify the interaction between a client and a very complex system.

## The Good

- Simplifies the use of a very complex system
- Reduces coupling between clients and various subsystems

## The Not So Good

- Hides possibly overly complex characteristics of a subsystem

## Façade Pattern Implementation

Space travel is not easy. The most important part of space travel is the starship. However, the starship has complexity of its own. This is where the *StarShipFacade* comes into play.

The Façade main class:

## ***StarShipFacade.h***

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "StarShipCrewMember.h"
#include "StarShipFacade_Main.generated.h"

UCLASS()
class DESIGN_PATTERNS_API AStarShipFacade_Main : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AStarShipFacade_Main();

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

};
```

## *StarShipFacade.cpp*

```
#include "StarShipFacade_Main.h"
#include "StarShipFacade.h"

// Sets default values
AStarShipFacade_Main::AStarShipFacade_Main()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void AStarShipFacade_Main::BeginPlay()
{
    Super::BeginPlay();

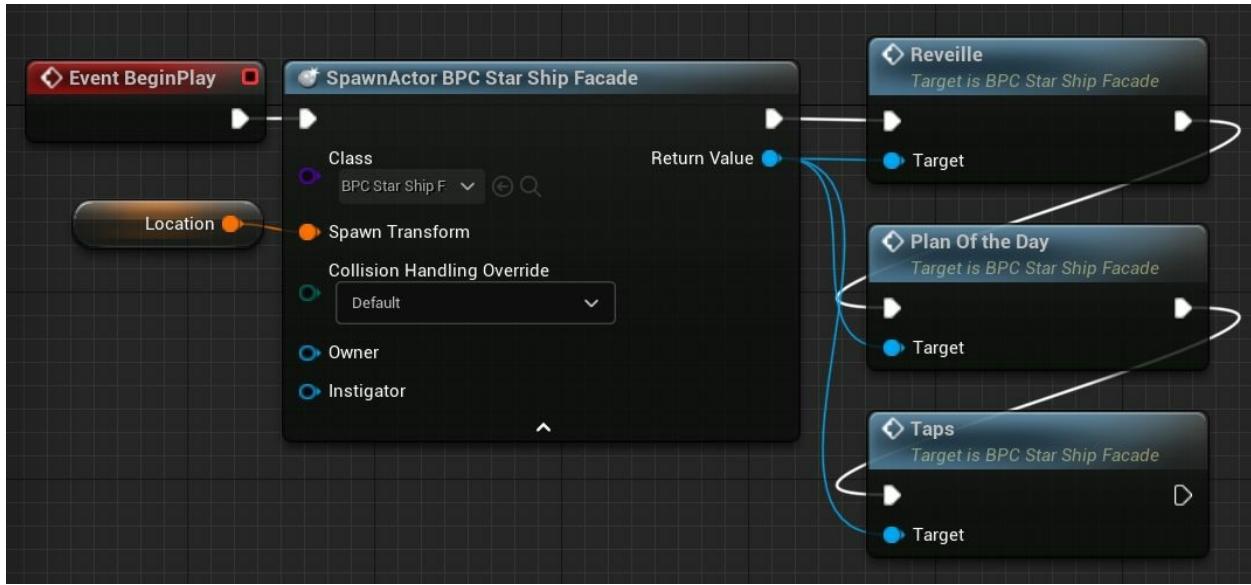
    //Create the Ship Facade Actor
    AStarShipFacade*      ShipFacade      =      GetWorld()->SpawnActor<AStarShipFacade>(AStarShipFacade::StaticClass());

    //Execute the needed tasks
    ShipFacade->Reveille();
    ShipFacade->PlanOfTheDay();
    ShipFacade->Taps();
}

// Called every frame
void AStarShipFacade_Main::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);
}
```

From *Façade\_Main* we can see how easy it is to run a starship. This is merely a façade using the Façade pattern. The complexity lies ahead.

The *Façade\_Main* event graph:



The *StarShipFacade* class:

## ***StarShipFacade.h***

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "StarShipCrewMember.h"
#include "StarShipFacade.generated.h"

UCLASS()
class DESIGN_PATTERNS_API AStarShipFacade : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AStarShipFacade();

private:
    //The Crew list
    TArray<AStarShipCrewMember*> Crew;

    //The Tasks to execute
    UPROPERTY()
    TArray<FString> Tasks;

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

private:
    //Execute the tasks for a specific Crew
    void PerformTasks(TArray<AStarShipCrewMember*> myCrew,
        TArray<FString> myTasks);

public:
    // Called every frame
```

```
virtual void Tick(float DeltaTime) override;  
  
//Execute the Reveille tasks  
void Reveille();  
  
//Execute the PlanOfTheDay tasks  
void PlanOfTheDay();  
  
//Execute the Taps tasks  
void Taps();  
};
```

## *StarShipFacade.cpp*

```
#include "StarShipFacade.h"
#include "StarShipNavigationOfficer.h"
#include "StarShipOperationsOfficer.h"
#include "StarShipSupplyOfficer.h"

// Sets default values
AStarShipFacade::AStarShipFacade()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void AStarShipFacade::BeginPlay()
{
    Super::BeginPlay();

    //Initialize the lists
    Crew = TArray<AStarShipCrewMember*>();
    Tasks = TArray<FString>();

    //Spawn the members
    AStarShipNavigationOfficer* NavOfficer = GetWorld()->SpawnActor<AStarShipNavigationOfficer>(AStarShipNavigationOfficer::StaticClass());
    AStarShipOperationsOfficer* OpsOfficer = GetWorld()->SpawnActor<AStarShipOperationsOfficer>(AStarShipOperationsOfficer::StaticClass());
    AStarShipSupplyOfficer* SupOfficer = GetWorld()->SpawnActor<AStarShipSupplyOfficer>(AStarShipSupplyOfficer::StaticClass());

    //Add the member to the crew
    Crew.Add(NavOfficer);
```

```
Crew.Add(OpsOfficer);
Crew.Add(SupOfficer);
}

// Called every frame
void AStarShipFacade::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

}

void AStarShipFacade::Reveille()
{
    //Empty the task list
    Tasks.Empty();

    //Add the tasks to execute
    Tasks.Add("trice_up");
    Tasks.Add("muster");

    //Execute the tasks
    PerformTasks(Crew, Tasks);
}

void AStarShipFacade::PlanOfTheDay()
{
    //Empty the task list
    Tasks.Empty();

    //Add the task to execute
    Tasks.Add("duty");

    //Execute the tasks
    PerformTasks(Crew, Tasks);
}

void AStarShipFacade::Taps()
{
```

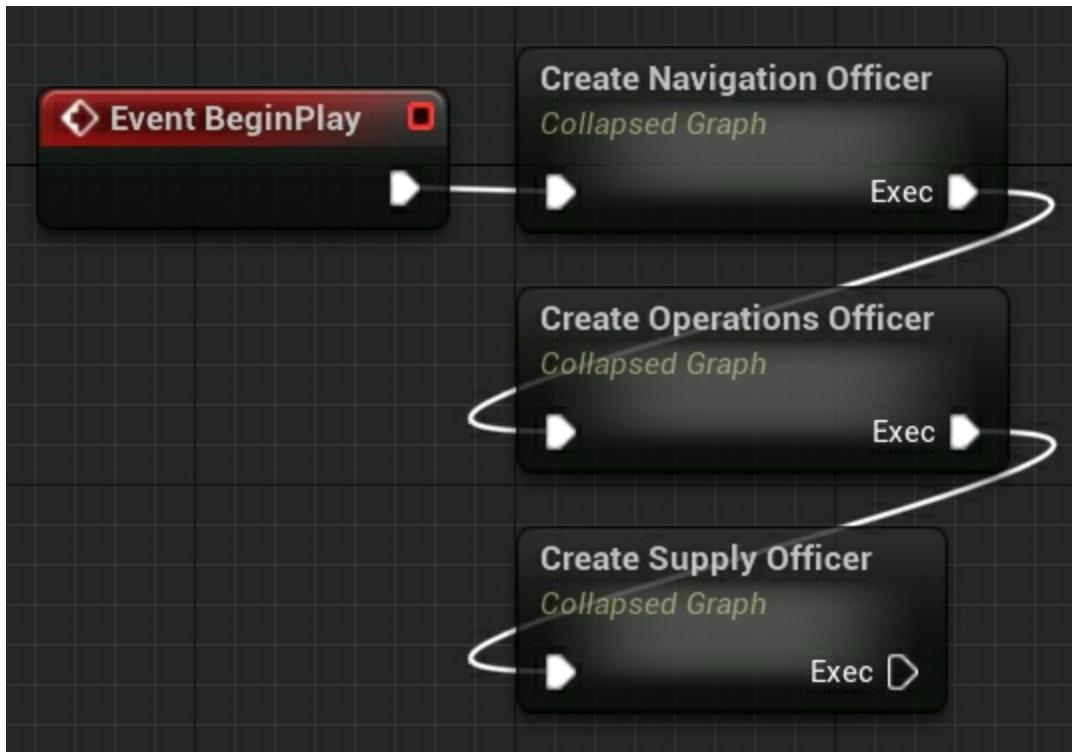
```
//Empty the task list
Tasks.Empty();

//Add the tasks to execute
Tasks.Add("liberty_call");
Tasks.Add("lights_out");

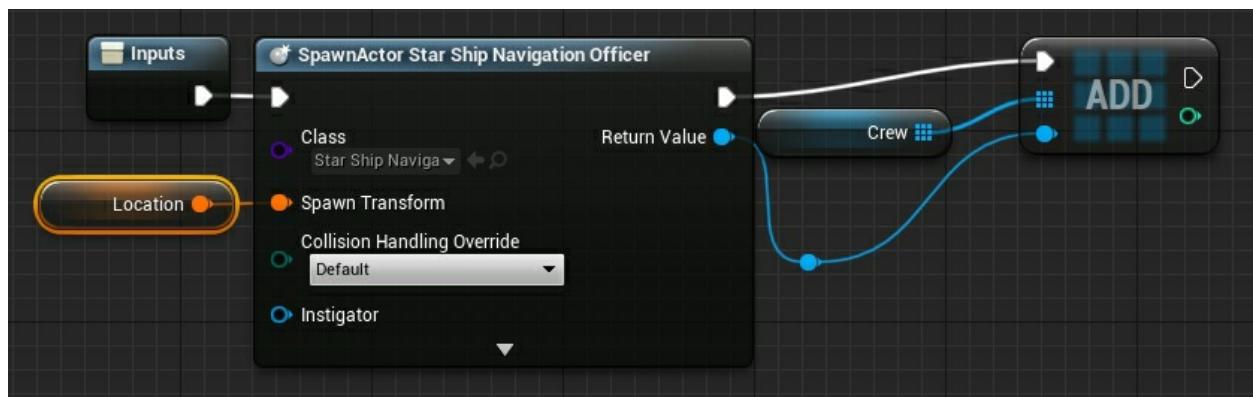
//Execute the tasks
PerformTasks(Crew, Tasks);
}

void AStarShipFacade::PerformTasks(TArray<AStarShipCrewMember*>
myCrew, TArray<FString> myTasks)
{
    //Execute the passed tasks for each crew member
    for (AStarShipCrewMember* Member : myCrew)
    {
        //Execute the task
        Member->Task(myTasks);
    }
}
```

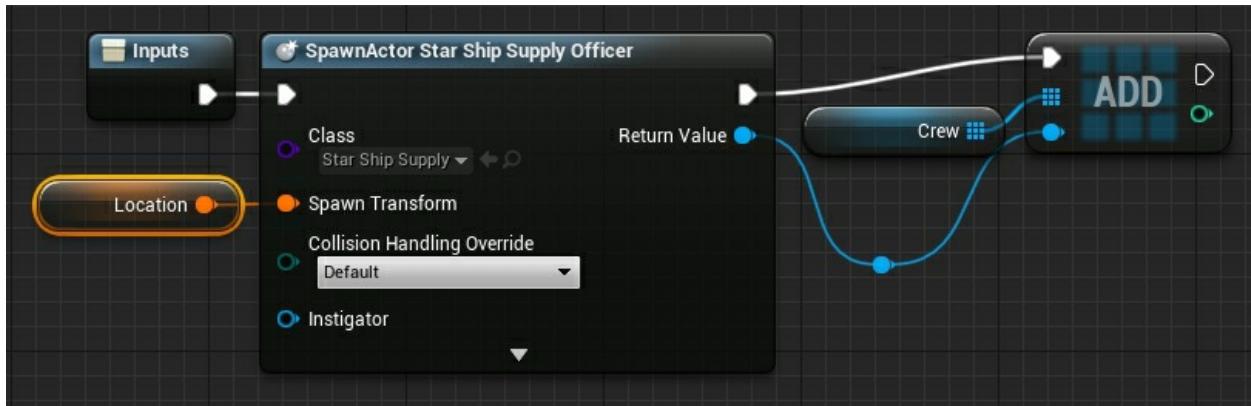
The *StarShipFacade* class is where the essence of this pattern exists. The *StarshipFacade* event graph:



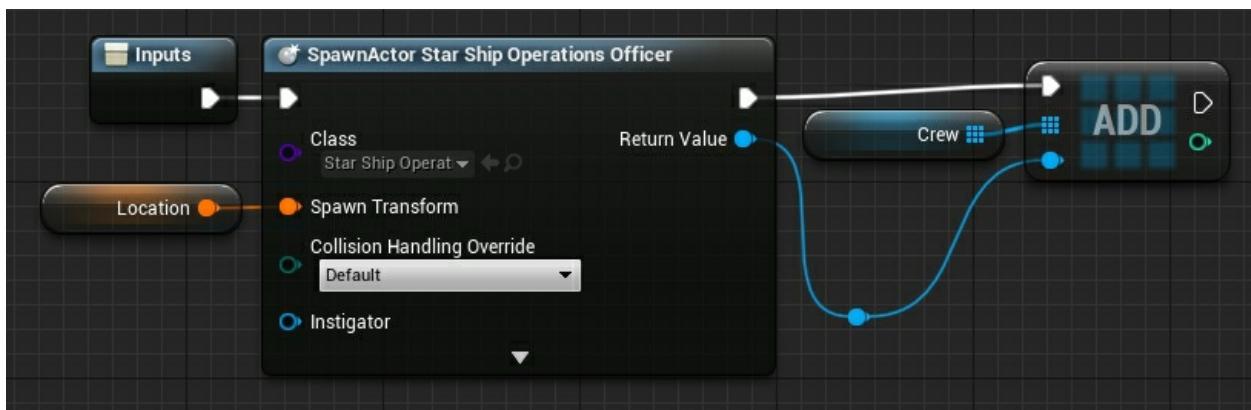
The *Add Navigation Officer to crew* collapsed graph:



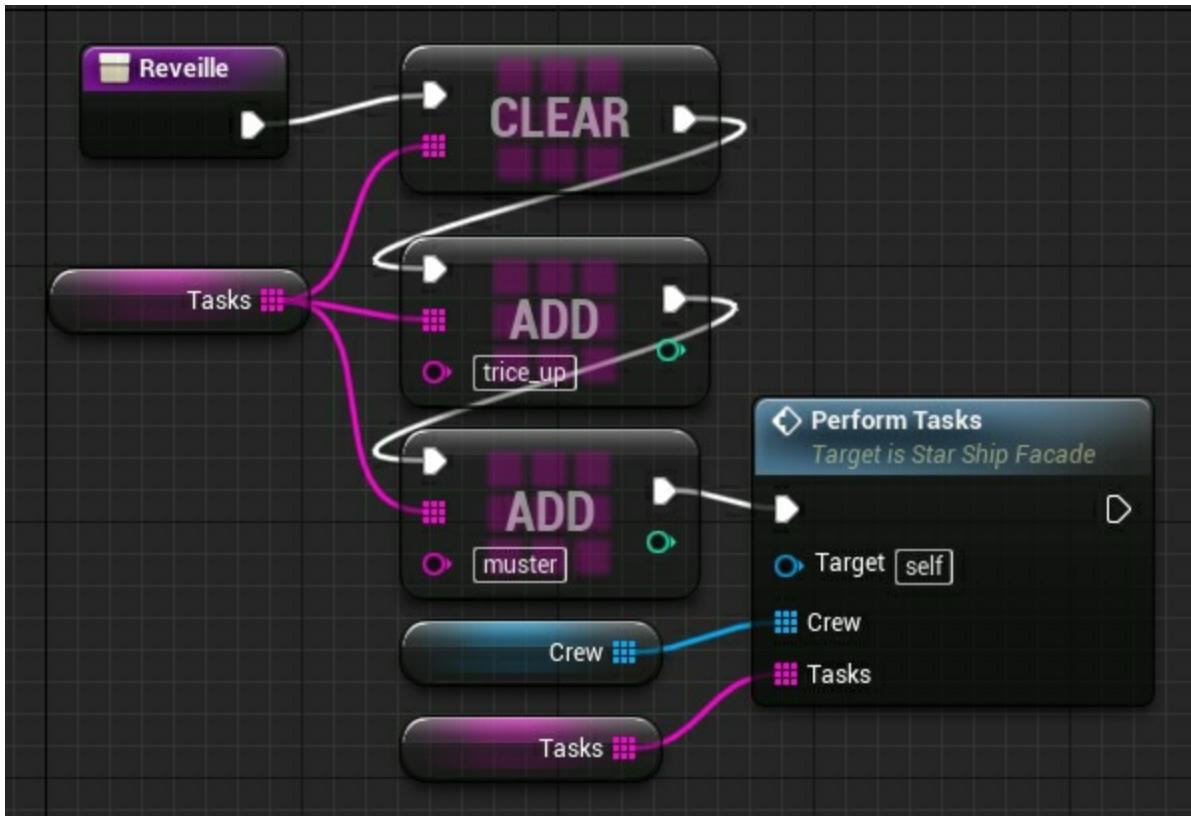
The *Add Supply Officer to crew* collapsed graph:



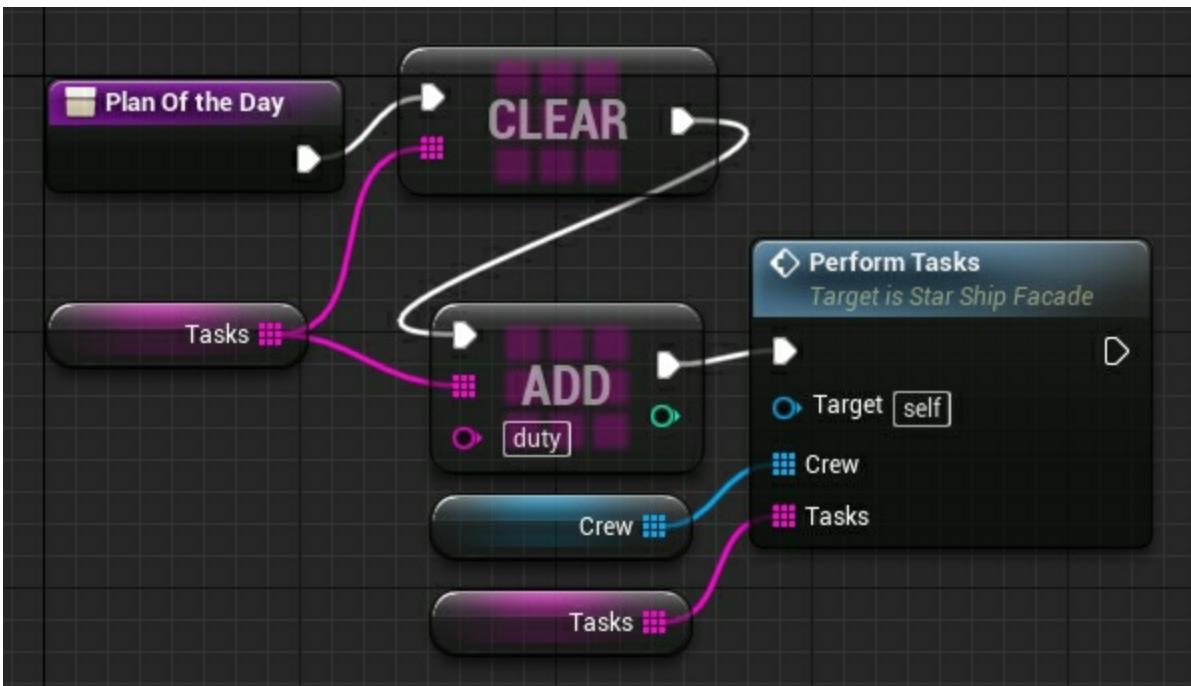
The *Add Operations Officer to Crew* collapsed graph:



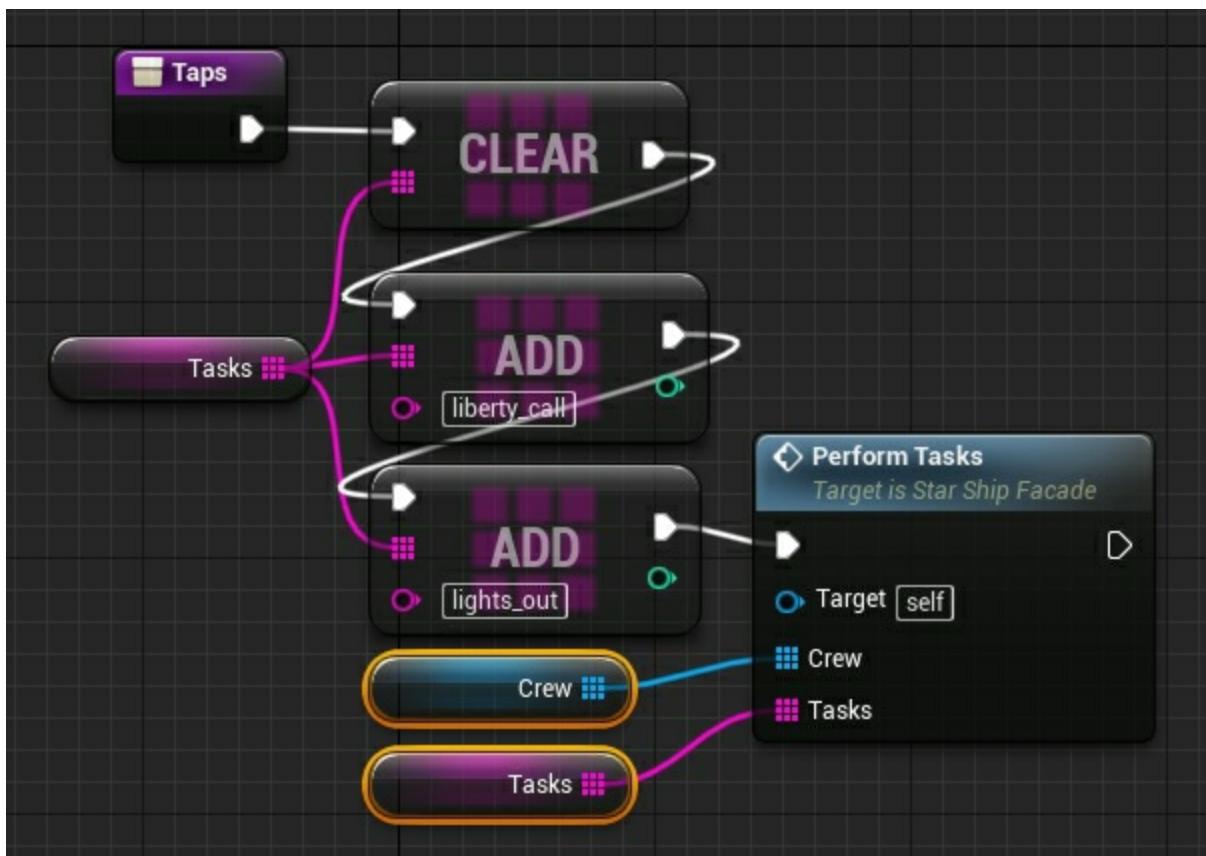
The *Reveille* blueprint function:



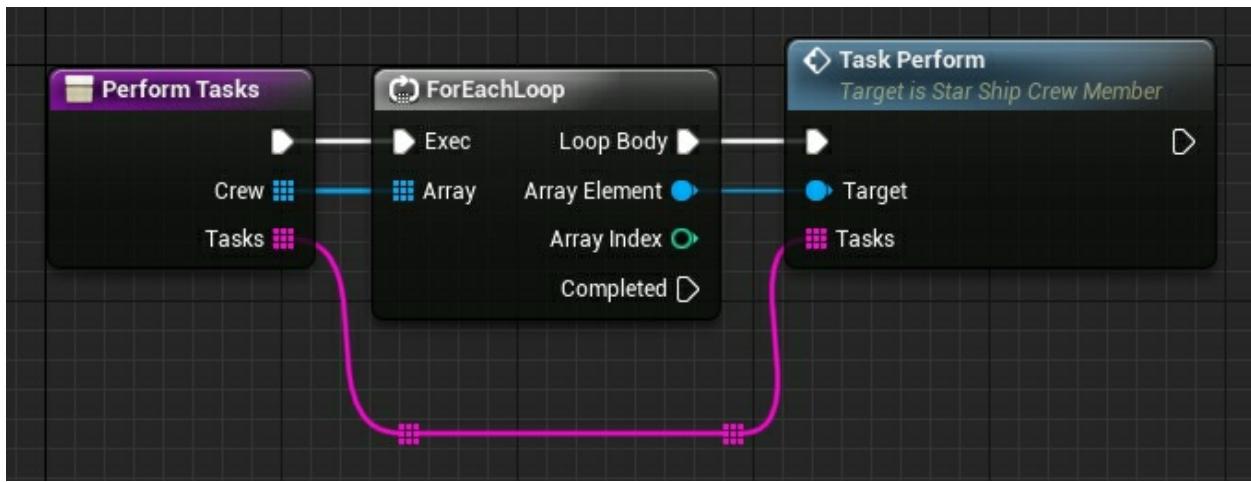
The *PlanOfTheDay* blueprint function:



The *Taps* blueprint function:



The *Perform Tasks* blueprint function:



The abstract class *StarShipCrewMember*:

## *StarShipCrewMember.h*

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "StarShipCrewMember.generated.h"

UCLASS()
class DESIGN_PATTERNS_API AStarShipCrewMember : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AStarShipCrewMember();

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

private:
    //Execute the passed Task
    void Task(const FString& Task);

public:
    //Execute the TurnLightsOut task
    void TurnLightsOut();

    //Execute the HeaveOutTriceUp task
    void HeaveOutTriceUp();

    //Execute the LibertyCall task
    void LibertyCall();

    //Execute the CrewMuster task
    void CrewMuster();
```

```
//Execute the passed Tasks
void Task(const TArray<FString>& Tasks);

    //Execute the member duty. It's pure virtual, so it doesn't need an
implementation in this class
    virtual void Duty() PURE_VIRTUAL(AStarShipCrewMember::Duty, {});

    //Return the member Title. It's pure virtual, so it doesn't need an
implementation in this class
    virtual FString CrewTitle()
PURE_VIRTUAL(AStarShipCrewMember::CrewTitle, return "");
```

## *StarShipCrewMember.cpp*

```
#include "StarShipCrewMember.h"

AStarShipCrewMember::AStarShipCrewMember()
{
}

void AStarShipCrewMember::BeginPlay()
{
    Super::BeginPlay();
}

void AStarShipCrewMember::TurnLightsOut()
{
    // Print lights out string
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
FString::Printf(TEXT("%s taps,taps lights out."), *CrewTitle));
}

void AStarShipCrewMember::HeaveOutTriceUp()
{
    // Print trice up string
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
FString::Printf(TEXT("%s all hands heave out and trice up."),
*CrewTitle));
}

void AStarShipCrewMember::LibertyCall()
{
    // Print liberty call string
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
FString::Printf(TEXT("%s liberty call, liberty call."), *CrewTitle));
}

void AStarShipCrewMember::CrewMuster()
{
```

```

// Print crew muster string
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
FString::Printf(TEXT("%s time for muster."), *CrewTitle()));
}

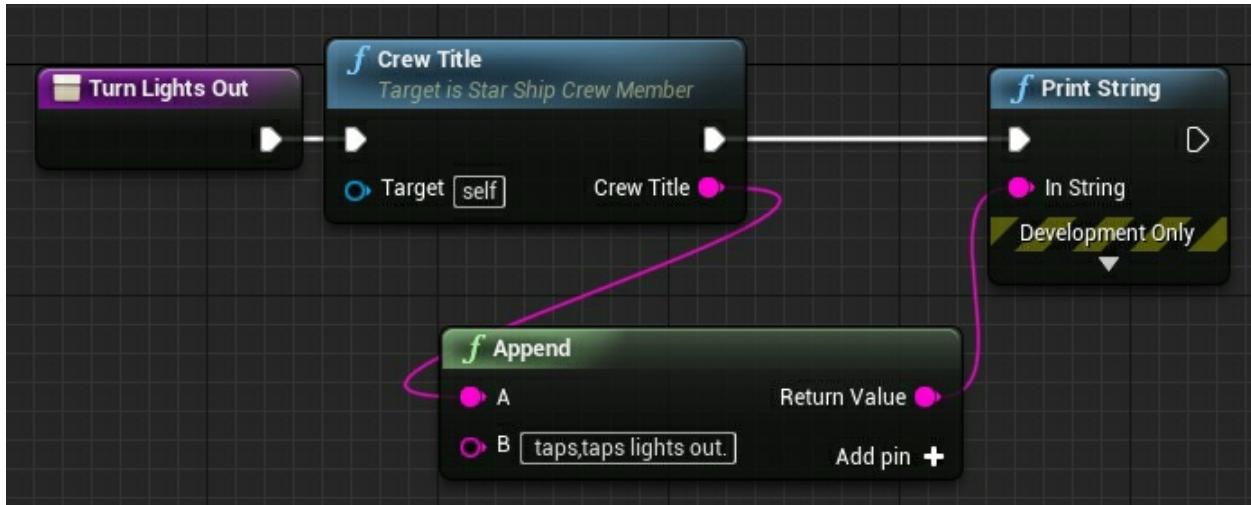
void AStarShipCrewMember::Task(const TArray<FString>& Tasks)
{
    //Loop the Tasks array and call the Task() function for each of them
    for (const FString& myTask : Tasks)
    {
        Task(myTask);
    }
}

void AStarShipCrewMember::Task(const FString& Task)
{
    //Checks which task must be executed
    if(Task.Equals("lights_out"))
    {
        TurnLightsOut();
    }
    else if(Task.Equals("trice_up"))
    {
        HeaveOutTriceUp();
    }
    else if (Task.Equals("liberty_call"))
    {
        LibertyCall();
    }
    else if (Task.Equals("muster"))
    {
        CrewMuster();
    }
    else if (Task.Equals("duty"))
    {
        Duty();
    }
}

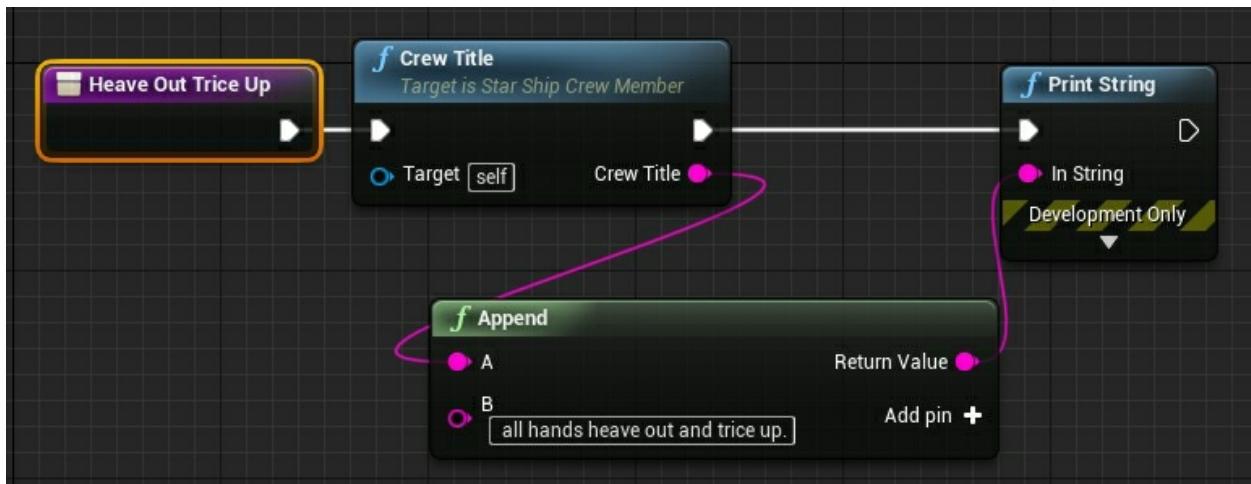
```

```
else
{
    //In case the passed Task doesn't exist
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Red,
TEXT("Undefined task"));
}
```

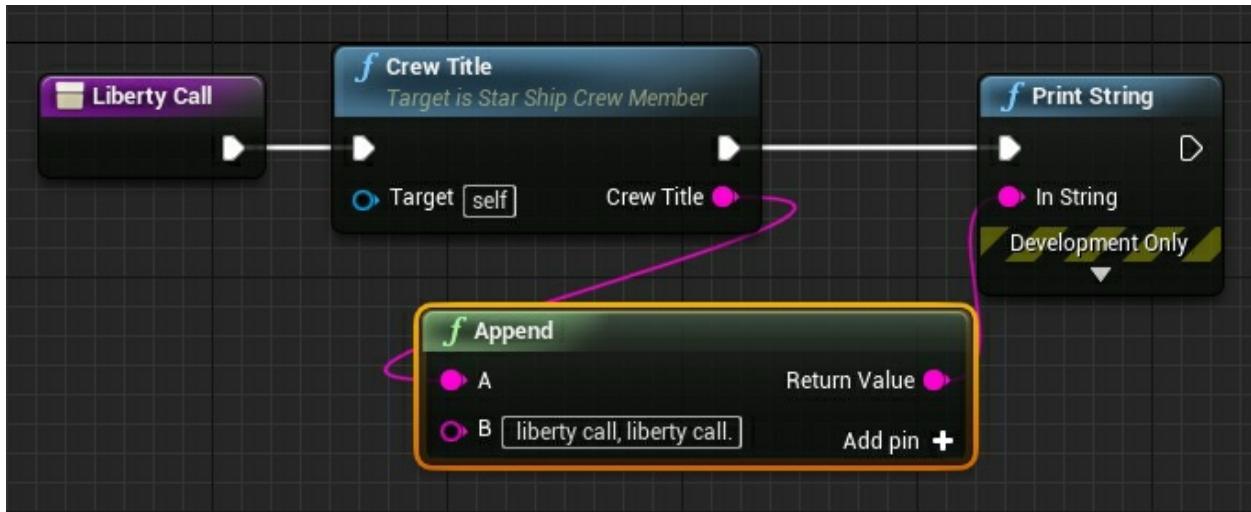
The *Turn Lights Out* blueprint function:



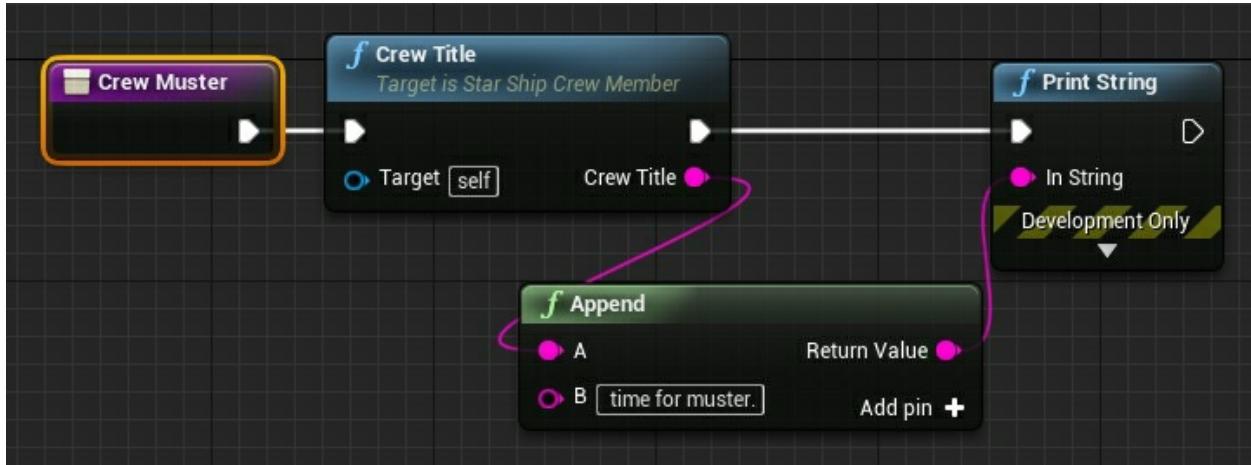
The *Heave Out Trice Up* blueprint function:



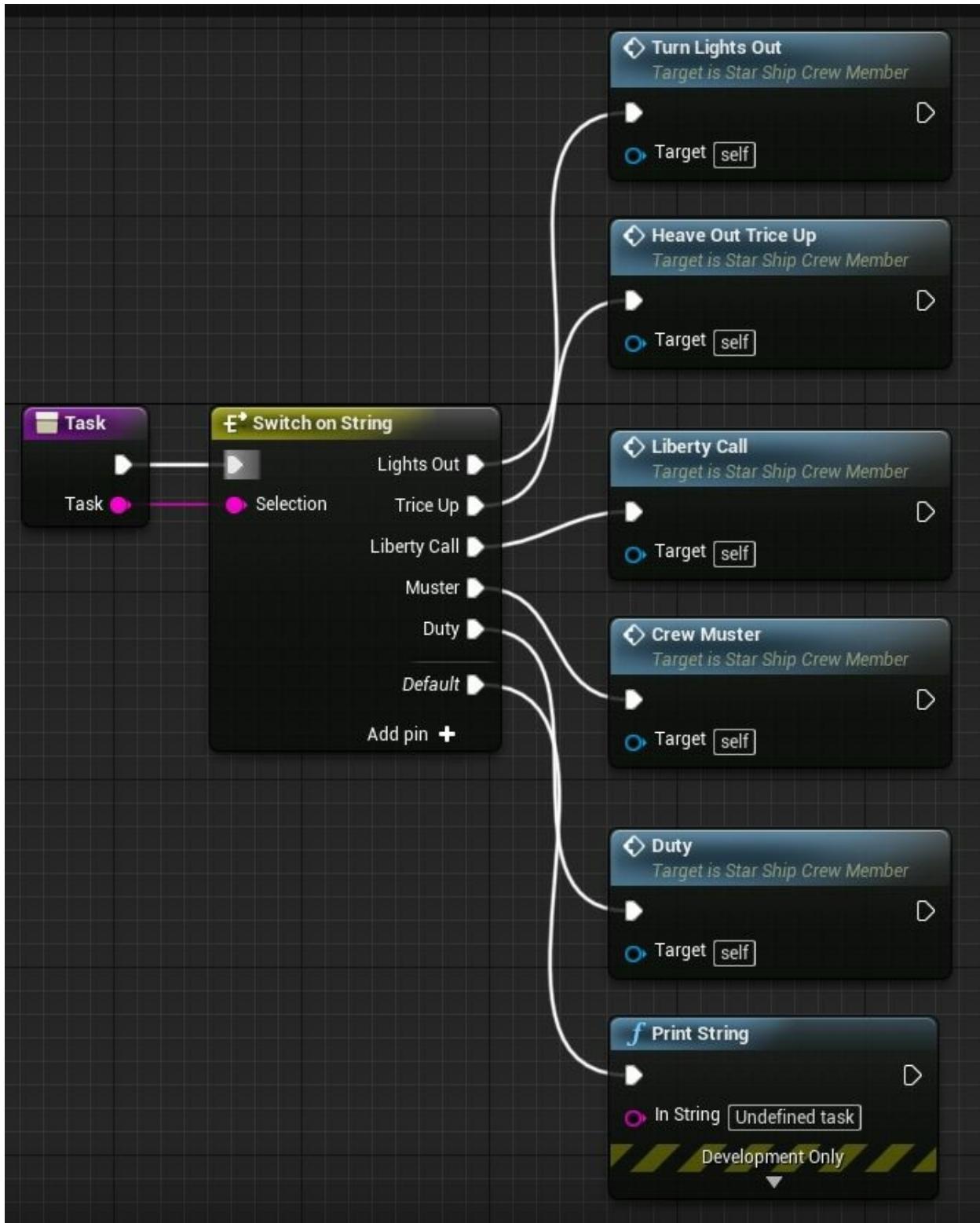
The *Liberty Call* blueprint function:



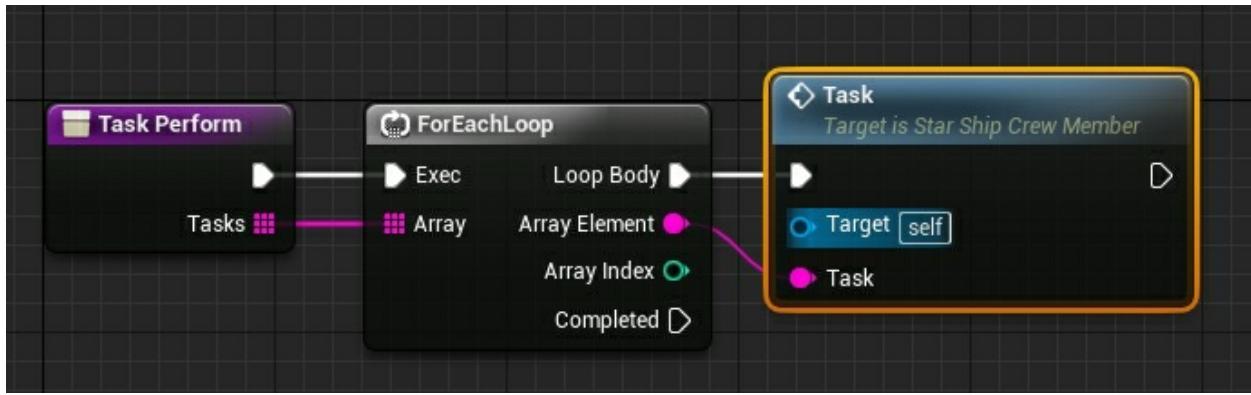
The *Crew Muster* blueprint function:



The *Task* blueprint function:



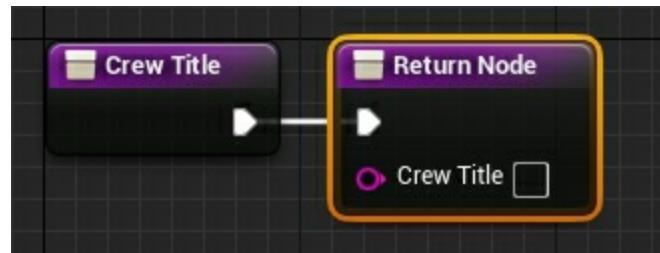
The *Task Perform* blueprint function:



The *Duty* blueprint function:



The *Crew Title* blueprint function:



The *StarShipNavigationOfficer* class:

## *StarShipNavigationOfficer.h*

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "StarShipCrewMember.h"
#include "StarShipNavigationOfficer.generated.h"

UCLASS()
class DESIGN_PATTERNS_API AStarShipNavigationOfficer : public
AStarShipCrewMember
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AStarShipNavigationOfficer();

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

    //Interface Duty function - Prints the Duty log
    virtual void Duty() override;

    //Interface Duty function - Returns the title
    virtual FString CrewTitle() override;

};
```

## *StarShipNavigationOfficer.cpp*

```
#include "StarShipNavigationOfficer.h"

// Sets default values
AStarShipNavigationOfficer::AStarShipNavigationOfficer()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void AStarShipNavigationOfficer::BeginPlay()
{
    Super::BeginPlay();

}

// Called every frame
void AStarShipNavigationOfficer::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

}

void AStarShipNavigationOfficer::Duty()
{
    //Print Duty string
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
    FString::Printf(TEXT("%s navigates the ship."), *CrewTitle()));
}

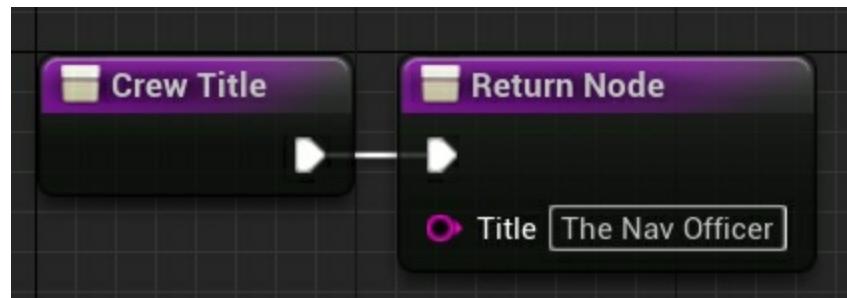
FString AStarShipNavigationOfficer::CrewTitle()
{
    //Return title
```

```
return "The Nav Officer";  
}
```

The *StarShipNavigationOfficer* Duty function:



The *StarShipNavigationOfficer* Crew Title function:



The *StarShipOperationsOfficer* class:

## *StarShipOperationsOfficer.h*

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "StarShipCrewMember.h"
#include "StarShipOperationsOfficer.generated.h"

UCLASS()
class DESIGN_PATTERNS_API AStarShipOperationsOfficer : public
AStarShipCrewMember
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AStarShipOperationsOfficer();

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

    //Interface Duty function - Prints the Duty log
    virtual void Duty() override;

    //Interface Duty function - Returns the title
    virtual FString CrewTitle() override;
};
```

## *StarShipOperationsOfficer.cpp*

```
#include "StarShipOperationsOfficer.h"

// Sets default values
AStarShipOperationsOfficer::AStarShipOperationsOfficer()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void AStarShipOperationsOfficer::BeginPlay()
{
    Super::BeginPlay();

}

// Called every frame
void AStarShipOperationsOfficer::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

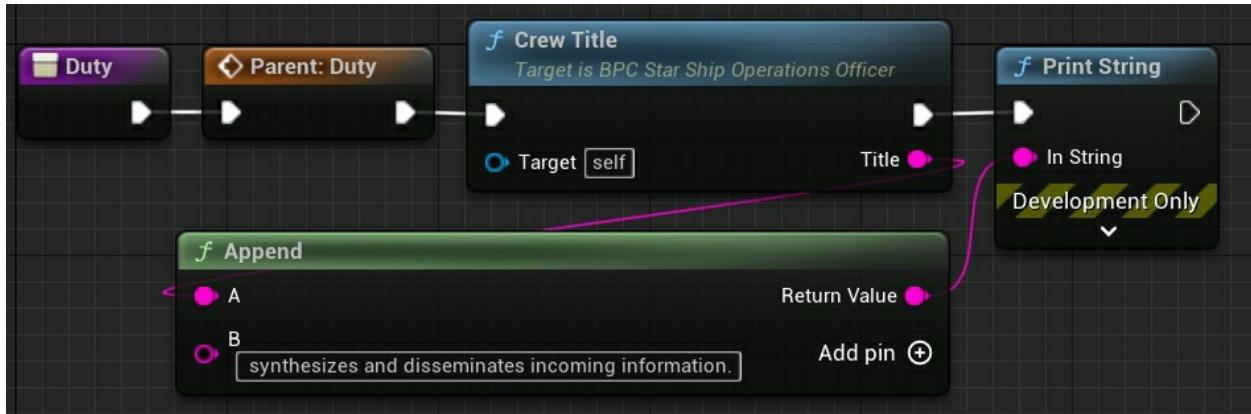
}

void AStarShipOperationsOfficer::Duty()
{
    //Print Duty string
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
    FString::Printf(TEXT("%s synthesizes and disseminates incoming
information."), *CrewTitle())));
}

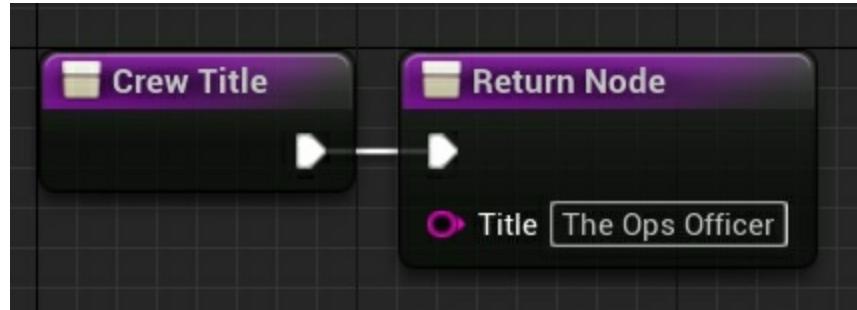
FString AStarShipOperationsOfficer::CrewTitle()
{
    //Return title
```

```
    return "The Ops Officer";  
}
```

The *StarShipOperationsOfficer* Duty function:



The *StarShipOperationsOfficer* Crew Title function:



The *StarShipSupplyOfficer* class:

## *StarShipSupplyOfficer.h*

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "StarShipCrewMember.h"
#include "StarShipSupplyOfficer.generated.h"

UCLASS()
class DESIGN_PATTERNS_API AStarShipSupplyOfficer : public AStarShipCrewMember
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AStarShipSupplyOfficer();

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

    //Interface Duty function - Prints the Duty log
    virtual void Duty() override;

    //Interface Duty function - Returns the title
    virtual FString CrewTitle() override;
};
```

## *StarShipSupplyOfficer.cpp*

```
#include "StarShipSupplyOfficer.h"

// Sets default values
AStarShipSupplyOfficer::AStarShipSupplyOfficer()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void AStarShipSupplyOfficer::BeginPlay()
{
    Super::BeginPlay();

}

// Called every frame
void AStarShipSupplyOfficer::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

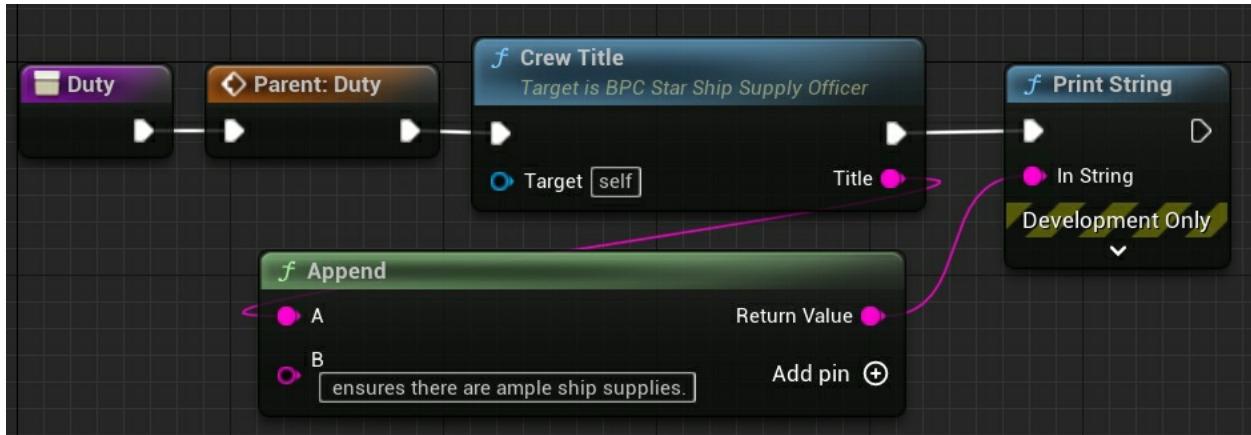
}

void AStarShipSupplyOfficer::Duty()
{
    //Print Duty string
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
    FString::Printf(TEXT("%s ensures there are ample ship supplies."),
    *CrewTitle())));
}

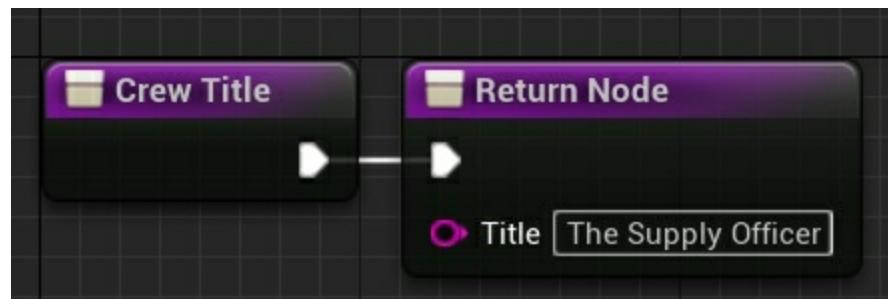
FString AStarShipSupplyOfficer::CrewTitle()
{
    //Return title
```

```
    return "The Supply Officer";  
}
```

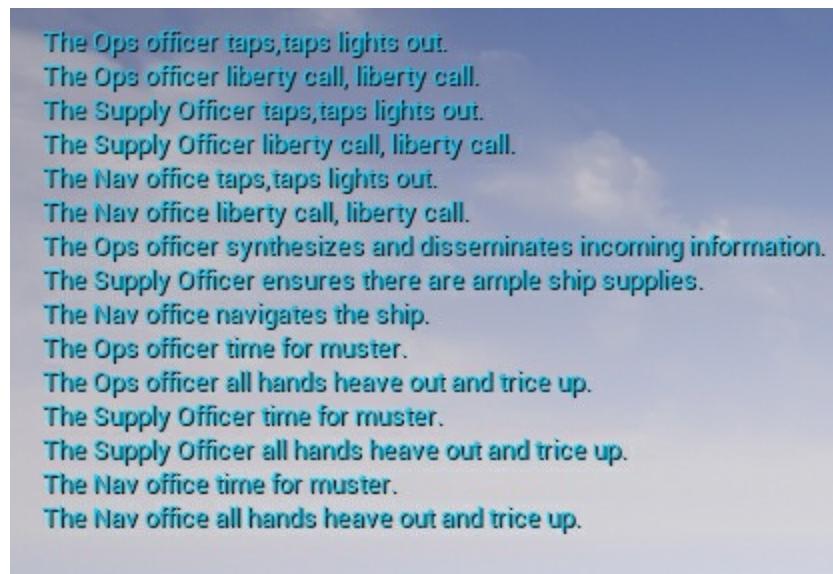
The *StarShipSupplyOfficer* Duty function:



The *StarShipSupplyOfficer* Crew Title function:



The Façade pattern viewport print:



# Behavioral Patterns

The behavioral patterns we discuss help us determine how objects interact. Additionally, object responsibilities are defined in behavioral patterns as well. We must mention communication in are behavioral pattern overview. Runtime communication between objects is a staple in behavioral pattern implementations.

# The Freaks Come out at night...

## Observer Pattern

The Observer pattern is a very popular pattern used throughout many different software development paradigms. Another name for this pattern is the publisher subscriber pattern. We have a publisher who broadcasts state changes and subscribers who listen for said broadcasts.

### **The Good**

- Great way to abstract publisher away from subscribers
- Publisher need not know anything about subscribers

### **The Not So Good**

- Special care must be taken to unsubscribe due to what GoF calls “dangling references.”

### **Observer Pattern Implementation**

A clock tower is our publisher for the Observer pattern example. The subscribers consist of a group of freaks who change their behavior based on the time of day. The freaks receive notifications from the clock tower when the clock towers time state has changed. The *Observer\_Main* class:

## ***Observer\_Main.h***

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "Observer_Main.generated.h"

UCLASS()
class DESIGN_PATTERNS_API AObserver_Main : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AObserver_Main();

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

};
```

## ***Observer\_Main.cpp***

```
#include "Observer_Main.h"
#include "ClockTower.h"
#include "FreakyAllen.h"
#include "FreakyJeff.h"
#include "FreakySue.h"

// Sets default values
AObserver_Main::AObserver_Main()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void AObserver_Main::BeginPlay()
{
    Super::BeginPlay();

    //Spawn the Clock Tower
    AClockTower* ClockTower = GetWorld()->SpawnActor<AClockTower>
    (AClockTower::StaticClass());

    //Spawn the first Subscriber and set its Clock Tower
    AFreakyAllen* FreakyAllen = GetWorld()->SpawnActor<AFreakyAllen>
    (AFreakyAllen::StaticClass());
    FreakyAllen->SetClockTower(ClockTower);

    //Spawn the second Subscriber and set its Clock Tower
    AFreakyJeff* FreakyJeff = GetWorld()->SpawnActor<AFreakyJeff>
    (AFreakyJeff::StaticClass());
    FreakyJeff->SetClockTower(ClockTower);

    //Spawn the third Subscriber and set its Clock Tower
    AFreakySue* FreakySue = GetWorld()->SpawnActor<AFreakySue>
```

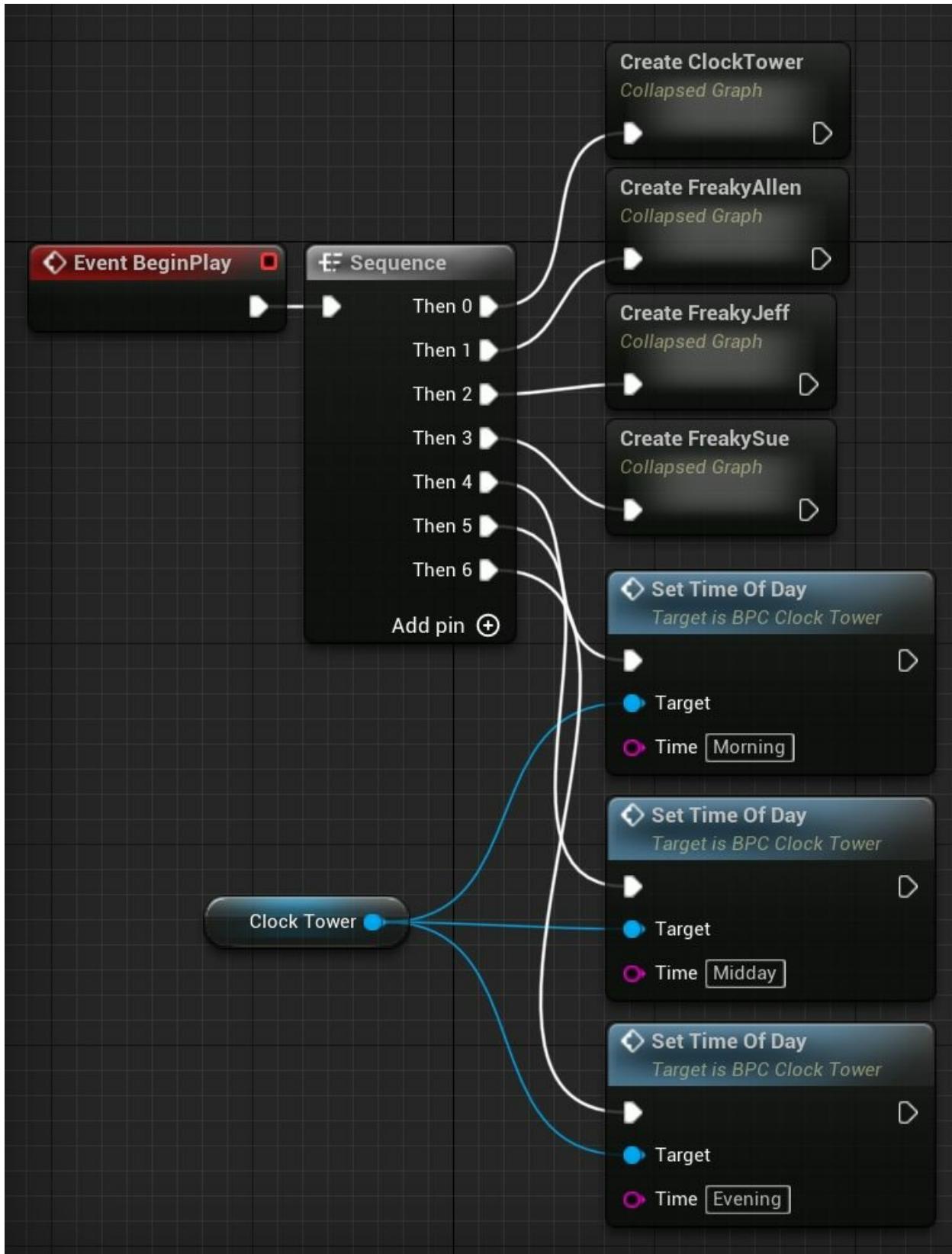
```
(AFreakySue::StaticClass());
FreakySue->SetClockTower(ClockTower);

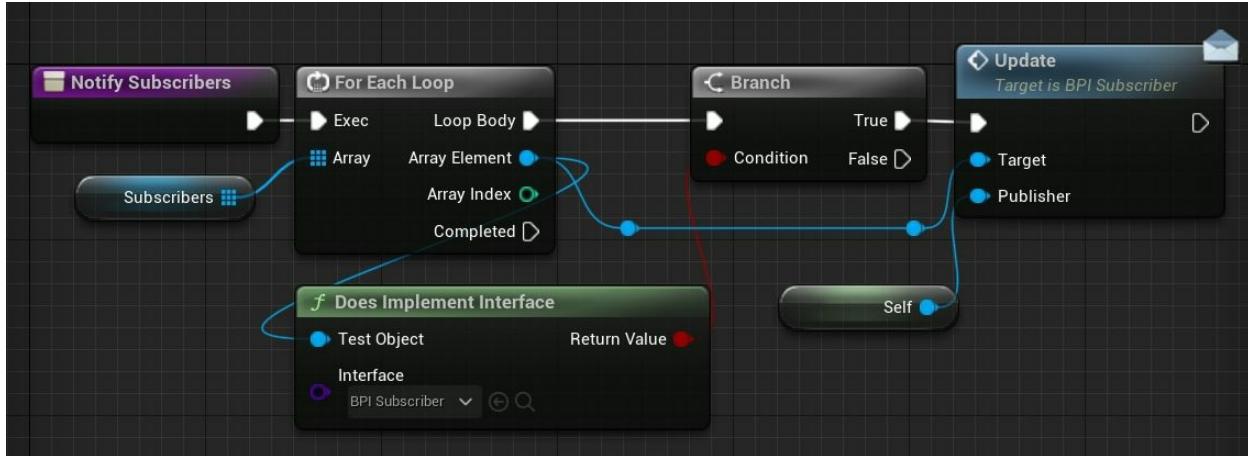
//Change the time of the Clock Tower, so the Subscribers can execute their
own routine
ClockTower->SetTimeOfDay("Morning");
ClockTower->SetTimeOfDay("Midday");
ClockTower->SetTimeOfDay("Evening");
}

// Called every frame
void AObserver_Main::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

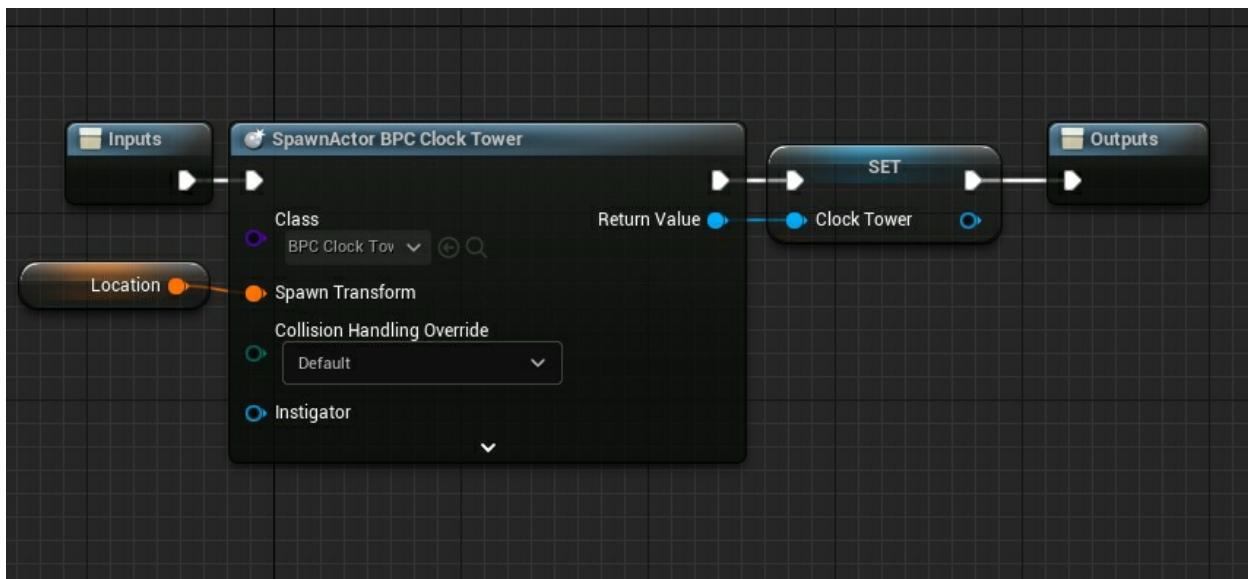
}
```

The *Observer\_Main* blueprint event graph:





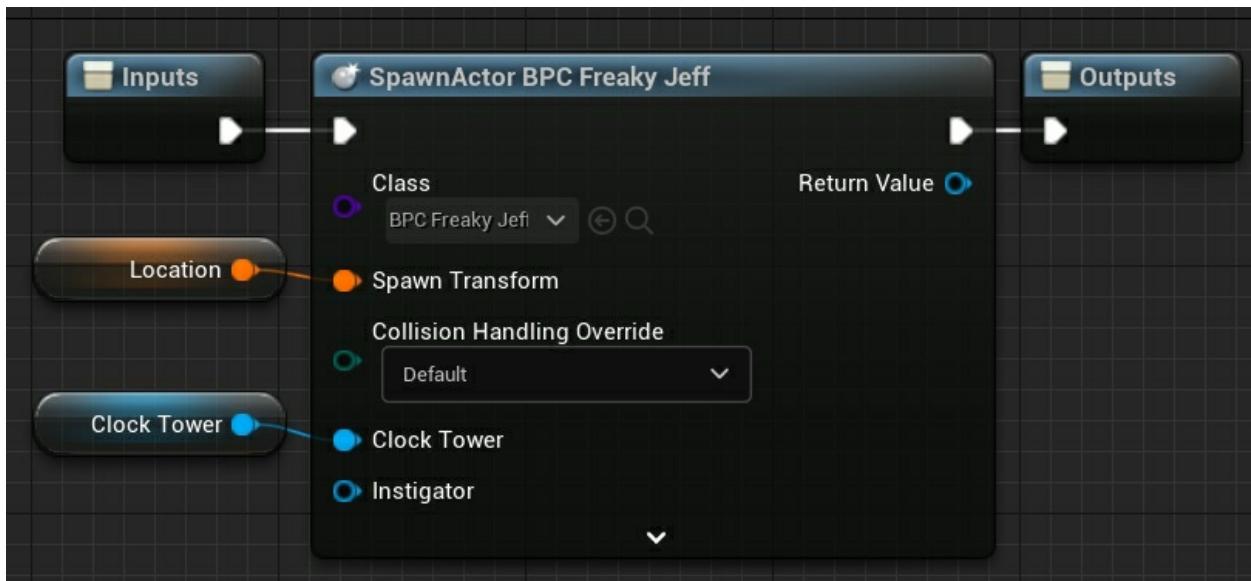
The *Create ClockTower* collapsed graph:



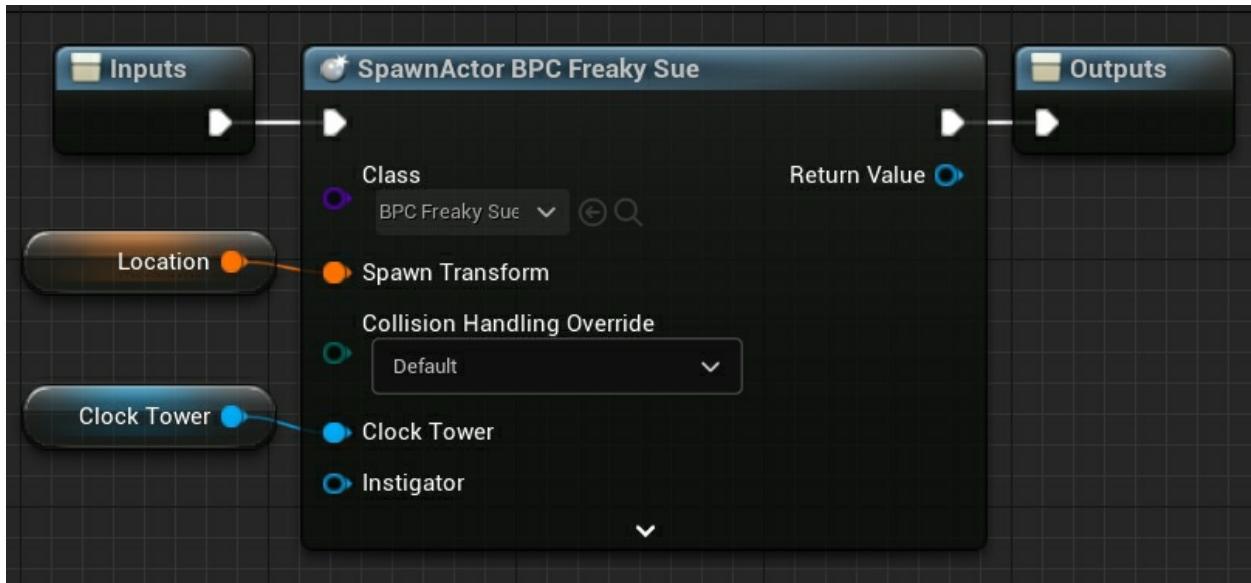
The *Create FreakyAllen* collapsed graph:



The *Create FreakyJeff* collapsed graph:



The *Create FreakySue* collapsed graph:



Let's implement the *Subscriber*:

## ***Subscriber.h***

```
#pragma once

#include "CoreMinimal.h"
#include "UObject/Interface.h"
#include "Subscriber.generated.h"

// This class does not need to be modified.
UINTERFACE(MinimalAPI)
class USubscriber : public UInterface
{
    GENERATED_BODY()
};

class DESIGN_PATTERNS_API ISubscriber
{
    GENERATED_BODY()

    // Add interface functions to this class. This is the class that will be
    // inherited to implement this interface.
public:

    //The pure virtual functions of the Subscriber
    virtual void Update(class APublisher* Publisher) = 0;
};
```

## ***Subscriber.cpp***

```
#include "Subscriber.h"

// Add default functionality here for any ISubscriber functions that are not
// pure virtual.

Lets implement the Publisher:
```

## **Publisher.h**

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "Subscriber.h"
#include "Publisher.generated.h"

UCLASS()
class DESIGN_PATTERNS_API APublisher : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    APublisher();

private:
    //The Subscribers of this Publisher
    UPROPERTY()
    TArray<AActor*> Subscribers = TArray<AActor*>();

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

public:
    //Add the passed Subscriber to the list
    virtual void Subscribe(AActor* Subscriber);

    //Remove the passed Subscriber from the list
    virtual void UnSubscribe(AActor* SubscriberToRemove);
```

```
//Notify all the Subscribers that something has changed  
virtual void NotifySubscribers();  
  
};
```

## **Publisher.cpp**

```
#include "Publisher.h"

// Sets default values
APublisher::APublisher()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void APublisher::BeginPlay()
{
    Super::BeginPlay();

}

// Called every frame
void APublisher::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

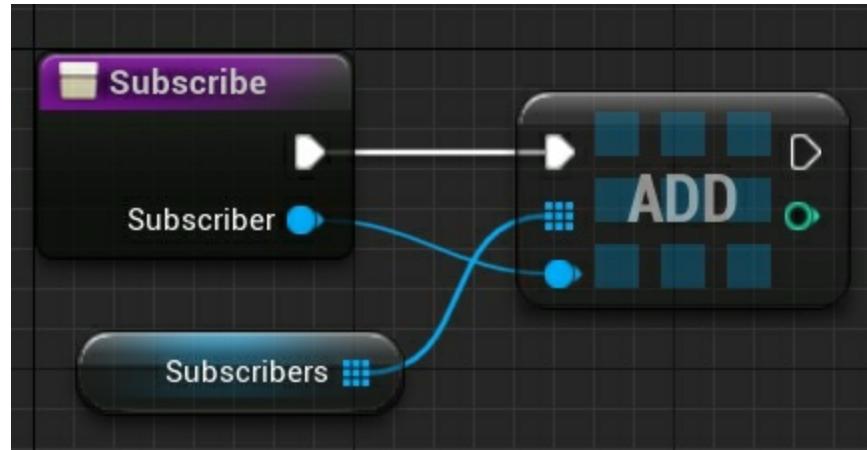
}

void APublisher::Subscribe(AActor* Subscriber)
{
    //Add the passed Subscriber
    Subscribers.Add(Subscriber);
}

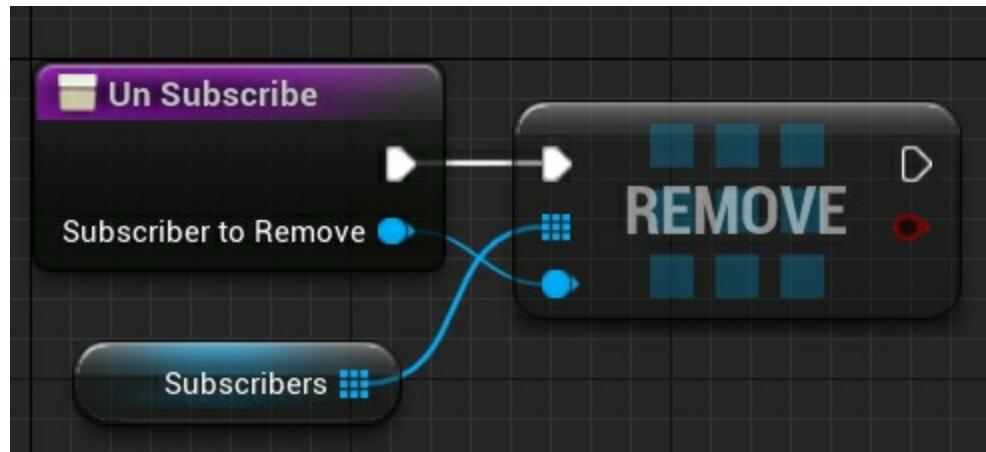
void APublisher::UnSubscribe(AActor* SubscriberToRemove)
{
    //Remove the passed Subscriber
    Subscribers.Remove(SubscriberToRemove);
}
```

```
void APublisher::NotifySubscribers()
{
    //Loop for each Subscriber
    for (AActor* Actor : Subscribers)
    {
        //Cast each of them to a concrete Subscriber
        ISubscriber* Sub = Cast<ISubscriber>(Actor);
        if (Sub)
        {
            //Notify each of them that something has changed, so they can execute
            their own routine
            Sub->Update(this);
        }
    }
}
```

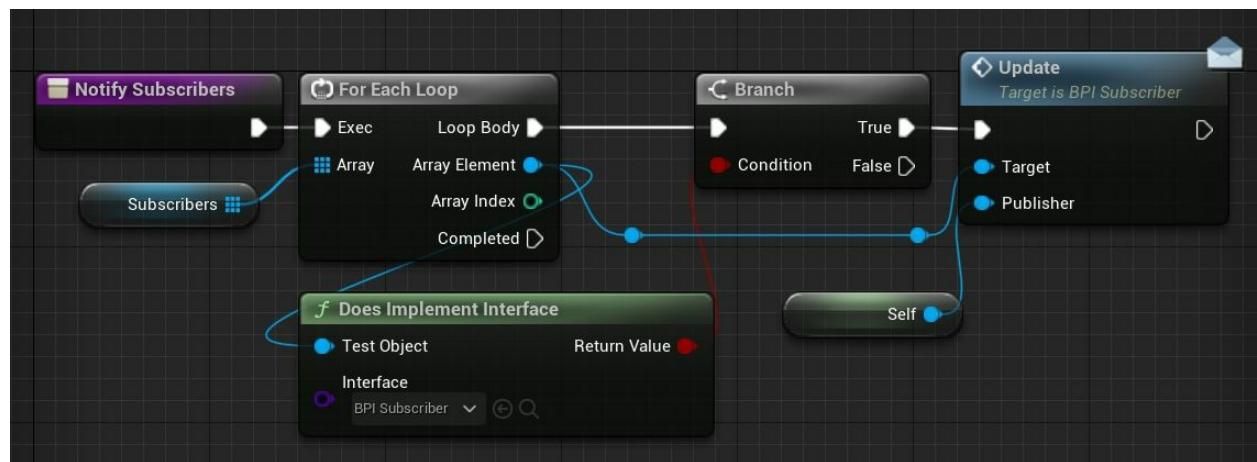
Here is the publisher's *Subscribe* function:



Here is the publisher's *UnSubscribe* function:



Here is the publisher's *NotifySubscribers* function:



The *ClockTower* class:

## **ClockTower.h**

```
#pragma once

#include "CoreMinimal.h"
#include "Publisher.h"
#include "ClockTower.generated.h"

UCLASS()
class DESIGN_PATTERNS_API AClockTower : public APublisher
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AClockTower();

private:
    //The current time of this Clock Tower
    FString Time;

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

public:
    //Called when the time of this Clock Tower has changed
    void TimeChanged();

    //Set the time of this Clock Tower
    void SetTimeOfDay(FString myTime);

    //Return the time of this Clock Tower
```

```
FORCEINLINE FString GetTime() { return Time; };  
};
```

## **ClockTower.cpp**

```
#include "ClockTower.h"

// Sets default values
AClockTower::AClockTower()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void AClockTower::BeginPlay()
{
    Super::BeginPlay();

}

// Called every frame
void AClockTower::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

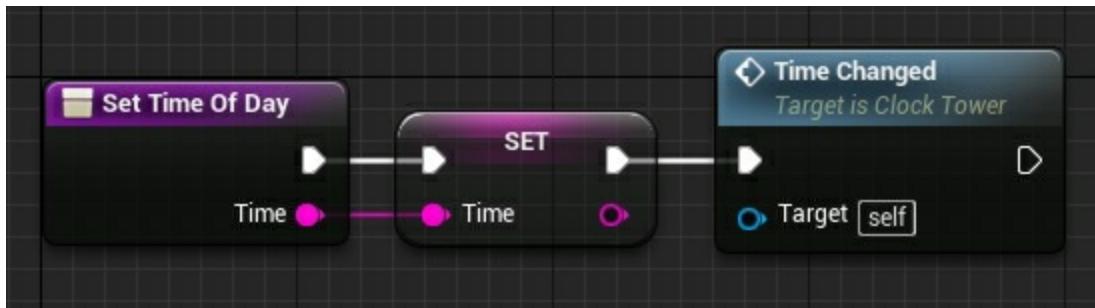
}

void AClockTower::TimeChanged()
{
    //When the time has changed, this Clock Tower (that is a Publisher)
    //notifies to all the subscribers that the time has changed
    NotifySubscribers();
}

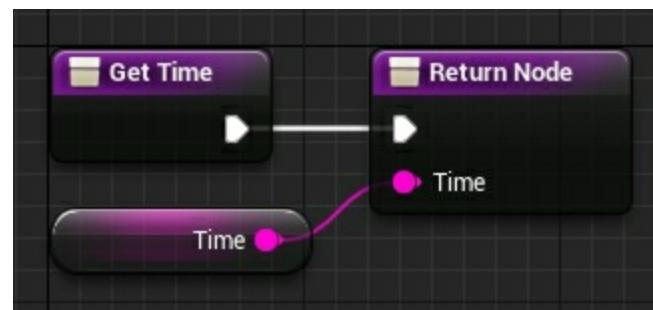
void AClockTower::SetTimeOfDay(FString myTime)
{
    //Set the time using the passed parameter and warn that it's changed
    Time = myTime;
```

```
    TimeChanged();  
}
```

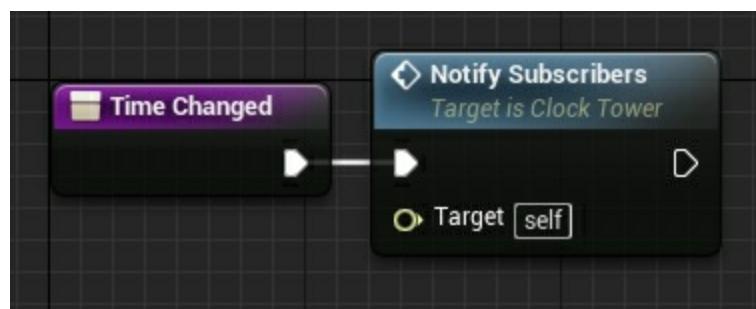
The *SetTimeOfDay* function:



The *GetTime* function:



The *TimeChanged* function:



The *FreakyAllen* class:

Let's implement the *Morph* interface our freaks will all share:

### ***Morph.h***

```
#pragma once

#include "CoreMinimal.h"
#include "UObject/Interface.h"
#include "Morph.generated.h"

// This class does not need to be modified.
UINTERFACE(MinimalAPI)
class UMorph : public UInterface
{
    GENERATED_BODY()
};

class DESIGN_PATTERNS_API IMorph
{
    GENERATED_BODY()

    // Add interface functions to this class. This is the class that will be
    // inherited to implement this interface.
public:

    //The pure virtual functions of the Morph
    virtual void Morph() = 0;
};
```

### ***Morph.cpp***

```
#include "Morph.h"

// Add default functionality here for any IMorph functions that are not pure
// virtual.
```

## *FreakyAllen.h*

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "Subscriber.h"
#include "Morph.h"
#include "FreakyAllen.generated.h"

class AClockTower;

UCLASS()
class DESIGN_PATTERNS_API AFreakyAllen : public AActor, public
ISubscriber, public IMorph
{
GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AFreakyAllen();

private:
    //The Clock Tower of this Subscriber
UPROPERTY()
    AClockTower* ClockTower;

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

    //Called when this Subscriber is destroyed, it will unsubscribe this from the
Clock Tower
```

```
virtual void Destroyed() override;  
  
public:  
    //Called when the Publisher changed its state, it will execute this  
    //Subscriber routine  
    virtual void Update(class APublisher* Publisher) override;  
  
    //Execute this Subscriber routine  
    virtual void Morph();  
  
    //Set the Clock Tower of this Subscriber  
    void SetClockTower(AClockTower* myClockTower);  
};
```

## *FreakyAllen.cpp*

```
#include "FreakyAllen.h"
#include "Publisher.h"
#include "ClockTower.h"

// Sets default values
AFreakyAllen::AFreakyAllen()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void AFreakyAllen::BeginPlay()
{
    Super::BeginPlay();
}

// Called every frame
void AFreakyAllen::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);
}

void AFreakyAllen::Destroyed()
{
    Super::Destroyed();

    //Log Error if its Clock Tower is NULL
    if (!ClockTower) { UE_LOG(LogTemp, Error, TEXT("Destroyed():\nClockTower is NULL, make sure it's initialized.")); return; }

    //Unsubscribe from the Clock Tower if it's destroyed
```

```

ClockTower->UnSubscribe(this);
}

void AFreakyAllen::Update(APublisher* Publisher)
{
    //Execute the routine
    Morph();
}

void AFreakyAllen::Morph()
{
    //Log Error if its Clock Tower is NULL
    if (!ClockTower) { UE_LOG(LogTemp, Error, TEXT("Morph():\nClockTower is NULL, make sure it's initialized.")); return; }

    //Get the current time of the Clock Tower
    FString Time = ClockTower->GetTime();

    if (!Time.Compare("Morning"))
    {
        //Execute the Morning routine
        GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
FString::Printf(TEXT("It is %s, so FreakyAllen makes a bowl of cereal"),
*Time));
    }
    else if (!Time.Compare("Midday"))
    {
        //Execute the Midday routine
        GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
FString::Printf(TEXT("It is %s, so FreakyAllen's right eye starts to twitch"),
*Time));
    }
    else if (!Time.Compare("Evening"))
    {
        //Execute the Evening routine
        GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
FString::Printf(TEXT("It is %s, so FreakyAllen morphs into a blood sucking\nwogglesnort"), *Time));
    }
}

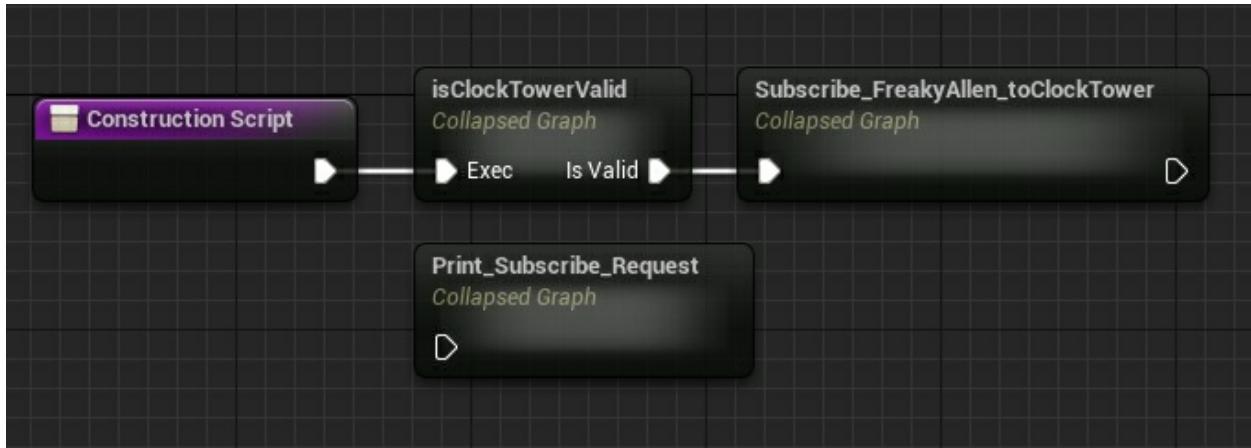
```

```
        }
    }

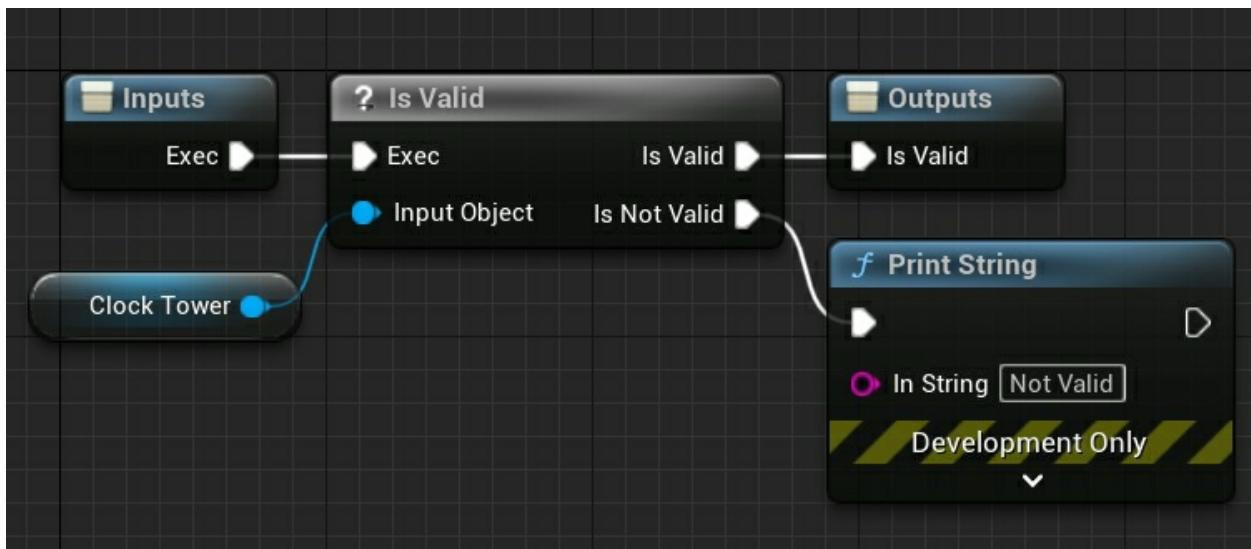
void AFreakyAllen::SetClockTower(AClockTower* myClockTower)
{
    //Log Error if the passed Clock Tower is NULL
    if (!myClockTower) { UE_LOG(LogTemp, Error,
TEXT("SetClockTower(): myClockTower is NULL, make sure it's
initialized.")); return; }

    //Set the Clock Tower and Subscribe to that
    ClockTower = myClockTower;
    ClockTower->Subscribe(this);
}
```

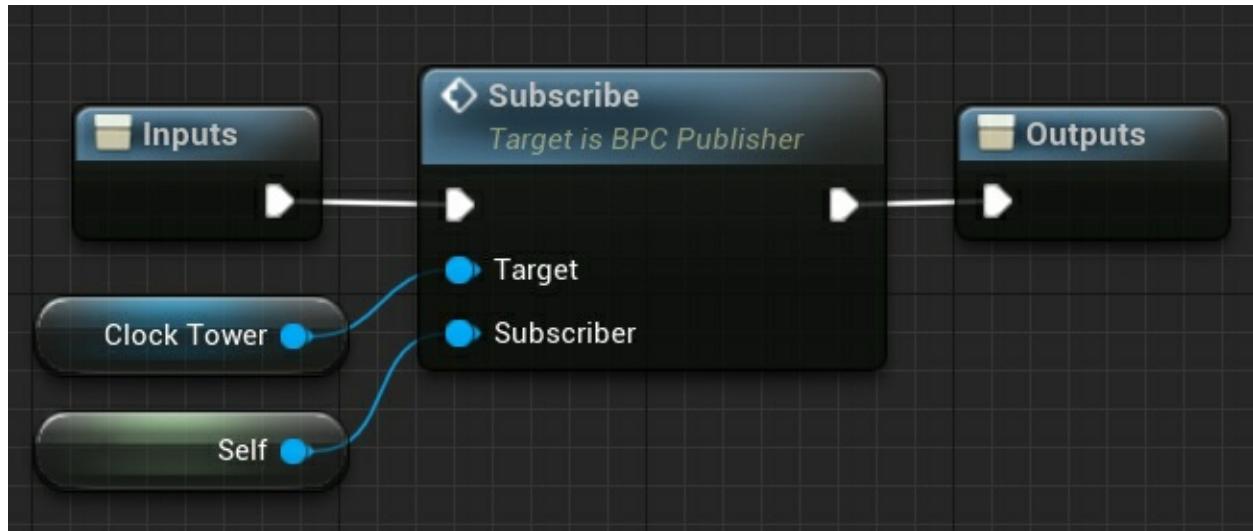
The *FreakyAllen* construction script:



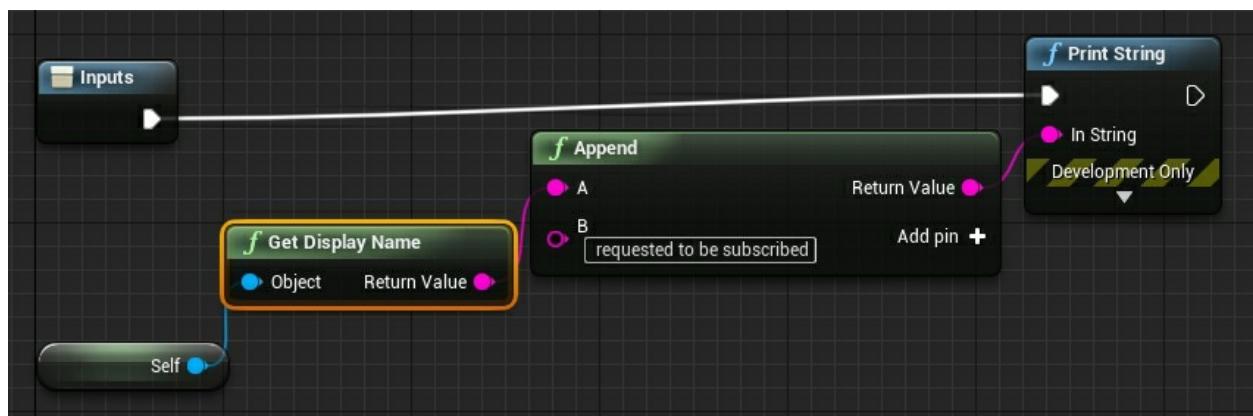
The *isClockTowerValid* collapsed graph:



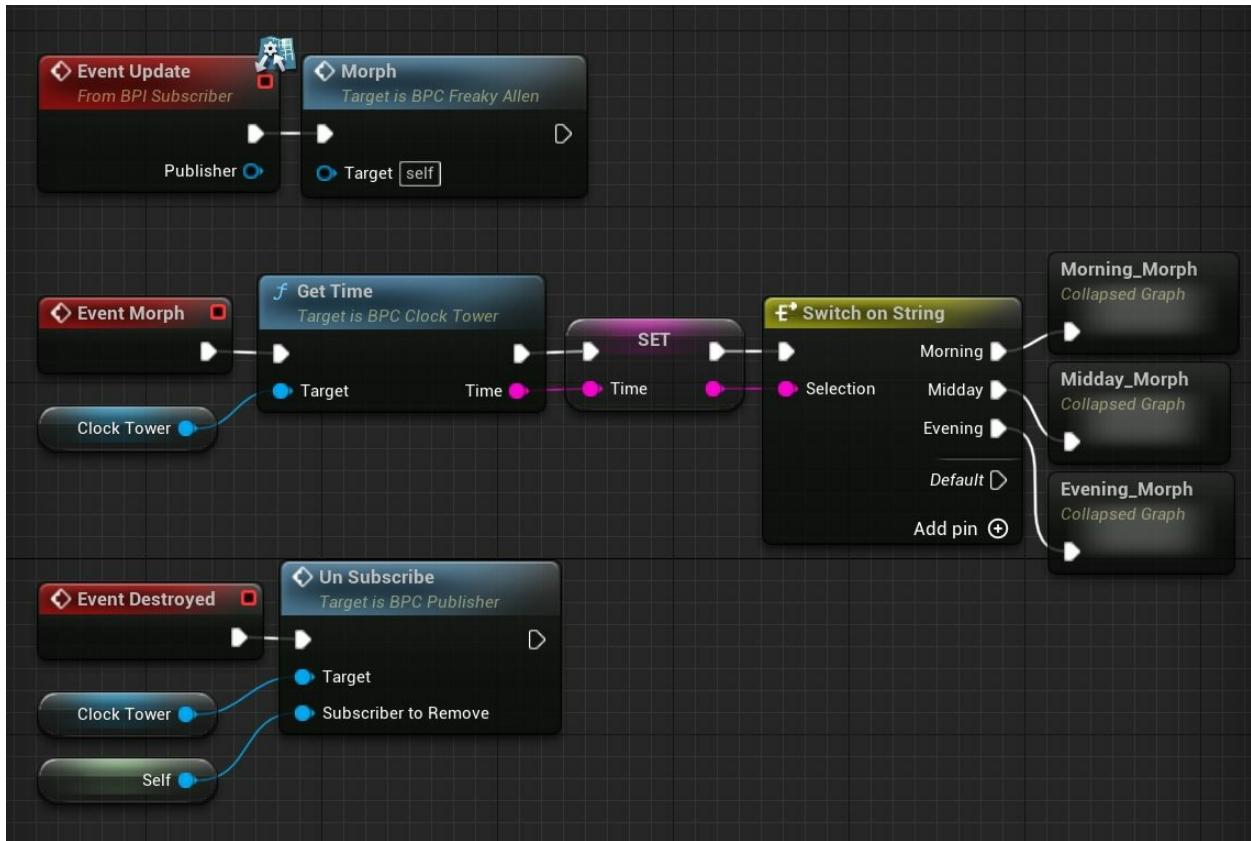
The *Subscribe\_FreakyAllen\_toClockTower* collapsed graph:



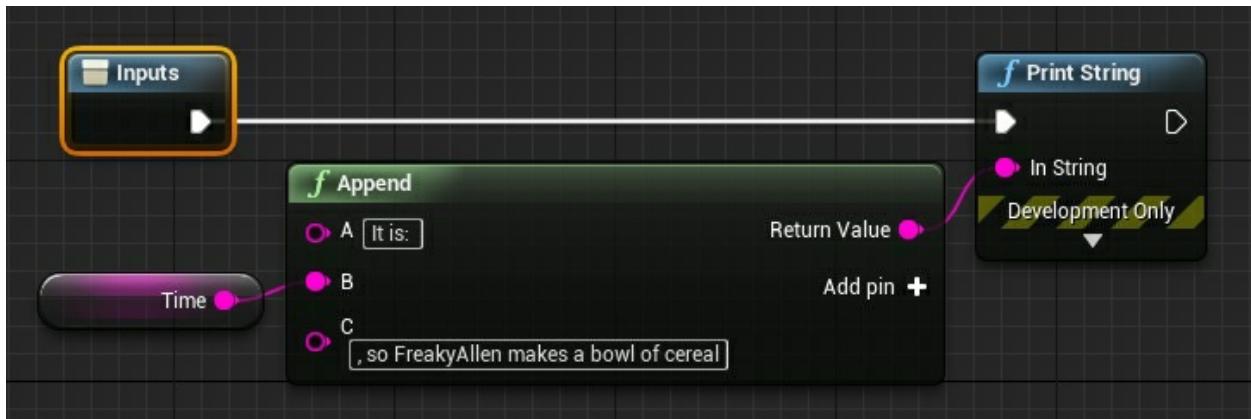
The *Print\_Subscribe\_Request* collapsed graph:



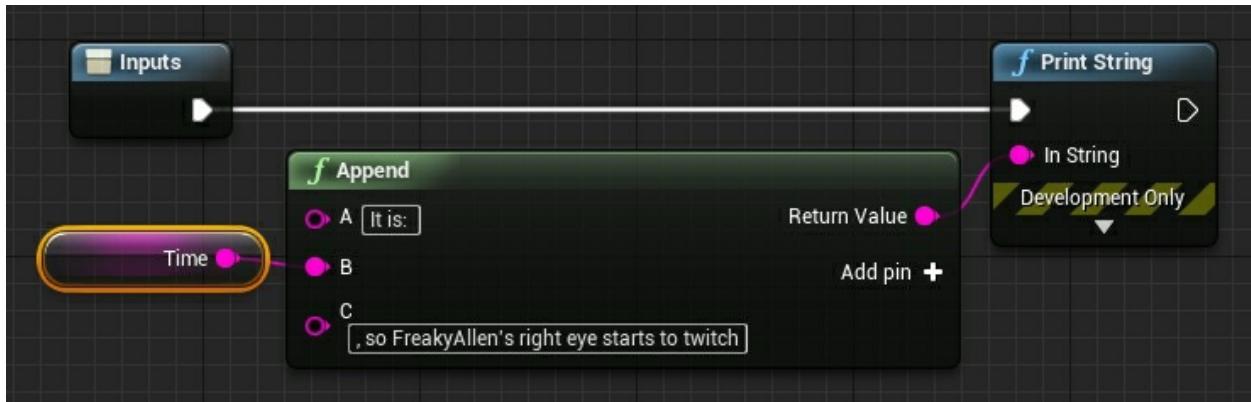
The *FreakyAllen* event graph:



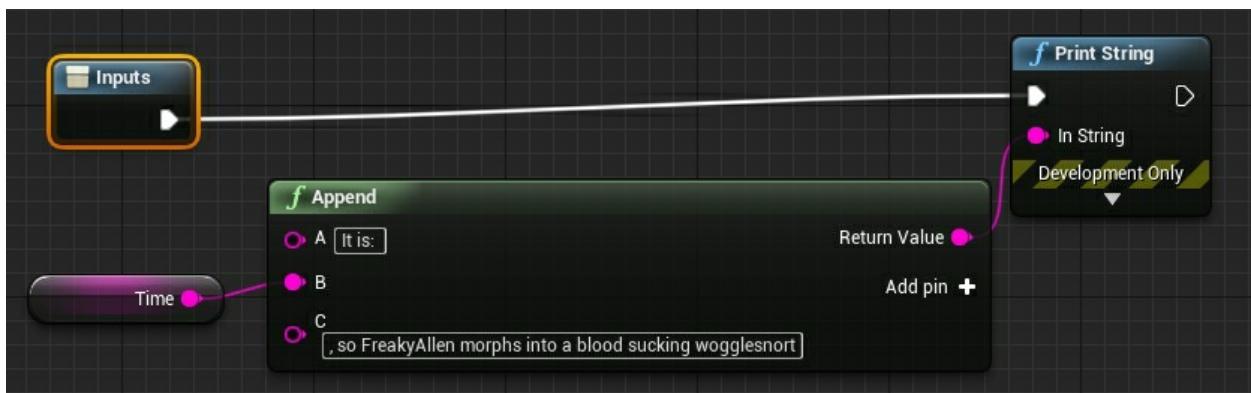
The *Morning\_Morph* collapsed graph:



The *Midday\_Morph* collapsed graph:

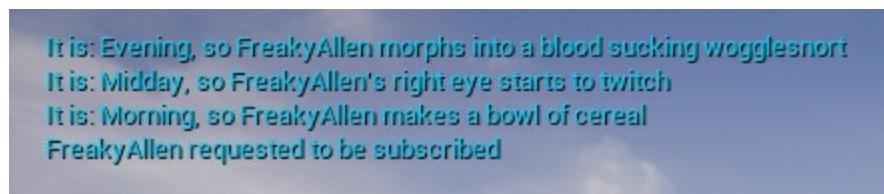


The *Evening\_Morph* collapsed graph:



The other freaks follow the same implementation with the exception of the strings that define their individual states and thus removed for brevity.

The Observer pattern viewport print:



# Make It Rain... State Pattern

The GoF describes the State pattern as allowing object to change their behavior when their internal state changes (Gamma et.al, 1995).

## The Good

- Localizes state specific behavior (Gamma et.al, 1995).

## The Not So Good

- Fairly similar to the Strategy pattern

## State Pattern Implementation

We are going to implement an old school slot machine. As we know, there are a few states the slot machine can assume. These states are no coin inserted, coin inserted, no dollars, won dollars. The no dollars state indicates the lack of funds inside of a slot machine. The won dollars state indicate a player winning. The *State\_Main* class:

## ***State\_Main.h***

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "State_Main.generated.h"

UCLASS()
class DESIGN_PATTERNS_API AState_Main : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AState_Main();

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

};
```

## ***State\_Main.cpp***

```
#include "State_Main.h"
#include "OldSchoolSlotMachine.h"

// Sets default values
AState_Main::AState_Main()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void AState_Main::BeginPlay()
{
    Super::BeginPlay();

    //Create the Slot Machine and set its Dollars Amount to 100
    AOldSchoolSlotMachine* OldSchoolSlotMachine = GetWorld()->SpawnActor<AOldSchoolSlotMachine>(AOldSchoolSlotMachine::StaticClass());
    OldSchoolSlotMachine->Initialize(100);

    //Log the current Slot Machine state
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
FString::Printf(TEXT("%s"), *OldSchoolSlotMachine->GetState()->ToString()));

    //Insert coin and Pull the lever
    OldSchoolSlotMachine->InsertCoin();
    OldSchoolSlotMachine->PullLever();

    //Log the current Slot Machine state
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
FString::Printf(TEXT("%s"), *OldSchoolSlotMachine->GetState()->ToString()));
}
```

```

//Insert coin, Pull the lever, then insert coin again
OldSchoolSlotMachine->InsertCoin();
OldSchoolSlotMachine->PullLever();
OldSchoolSlotMachine->InsertCoin();

//Log the current Slot Machine state
GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
FString::Printf(TEXT("%s"), *OldSchoolSlotMachine->GetState()-
>ToString()));

//Pull the lever
OldSchoolSlotMachine->PullLever();

//Add 100 Dollars
OldSchoolSlotMachine->RestockDollars(100);

//Insert coin and Pull the lever
OldSchoolSlotMachine->InsertCoin();
OldSchoolSlotMachine->PullLever();

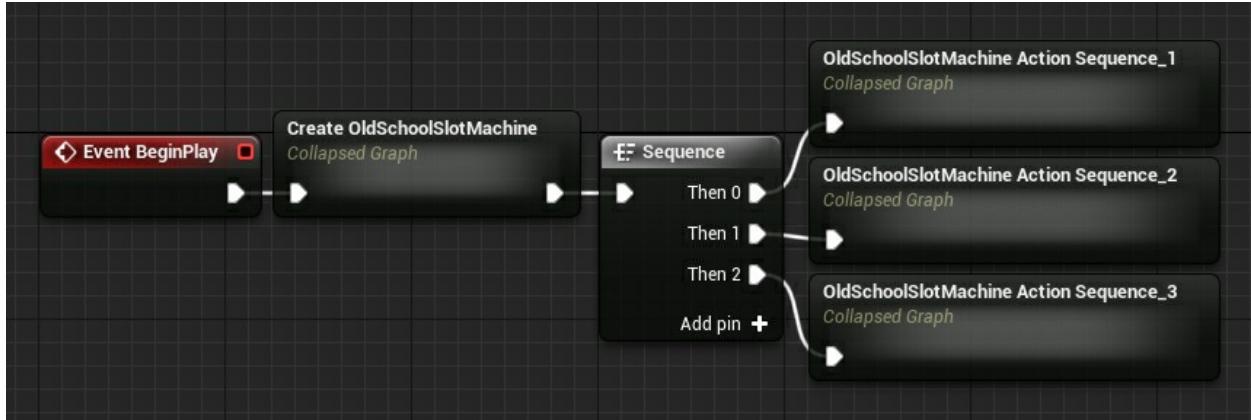
//Log the current Slot Machine state
GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
FString::Printf(TEXT("%s"), *OldSchoolSlotMachine->GetState()-
>ToString()));
}

// Called every frame
void AState_Main::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

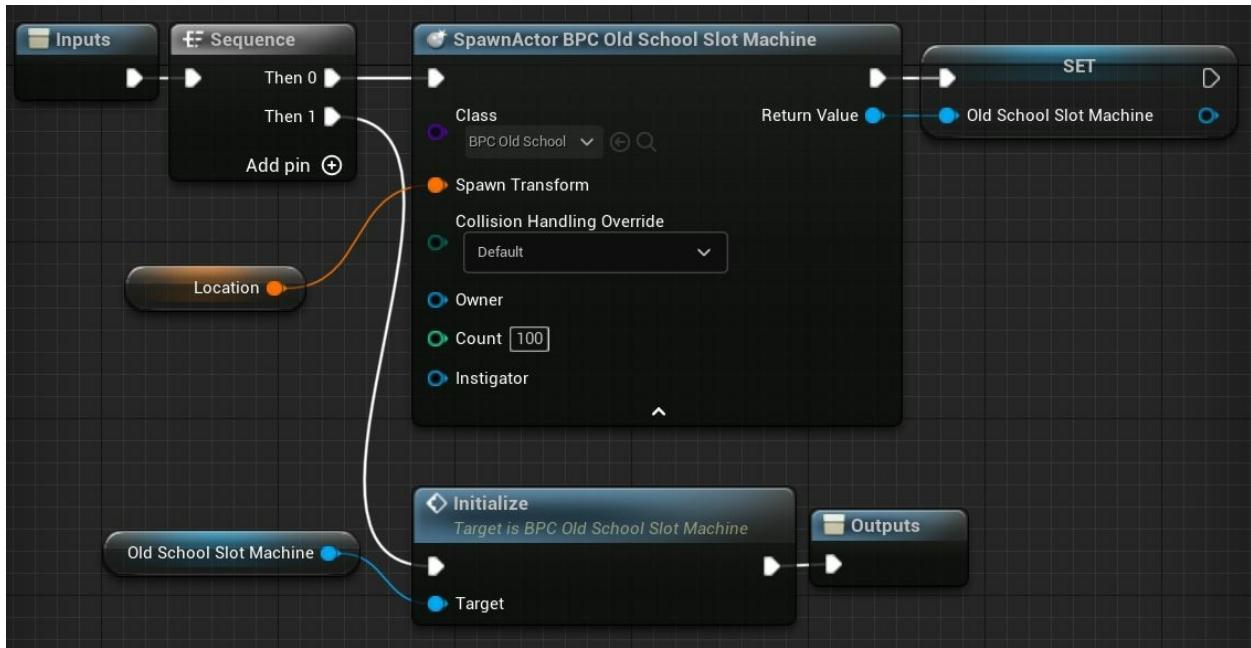
}

```

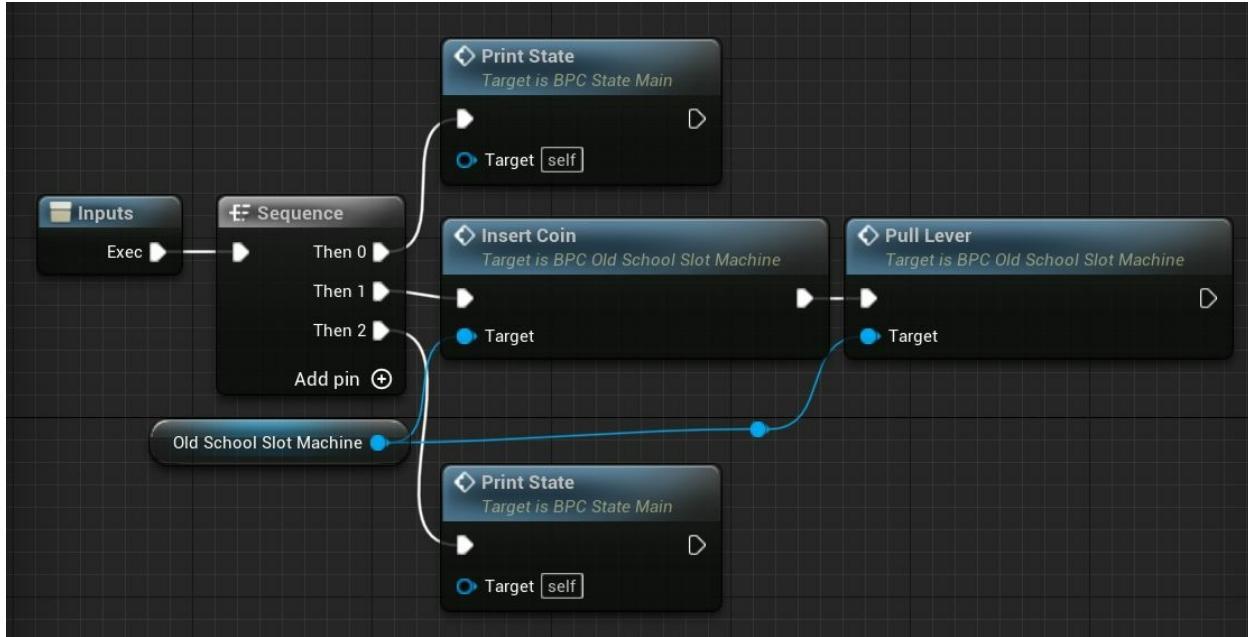
The *State\_Main* blueprint event graph:



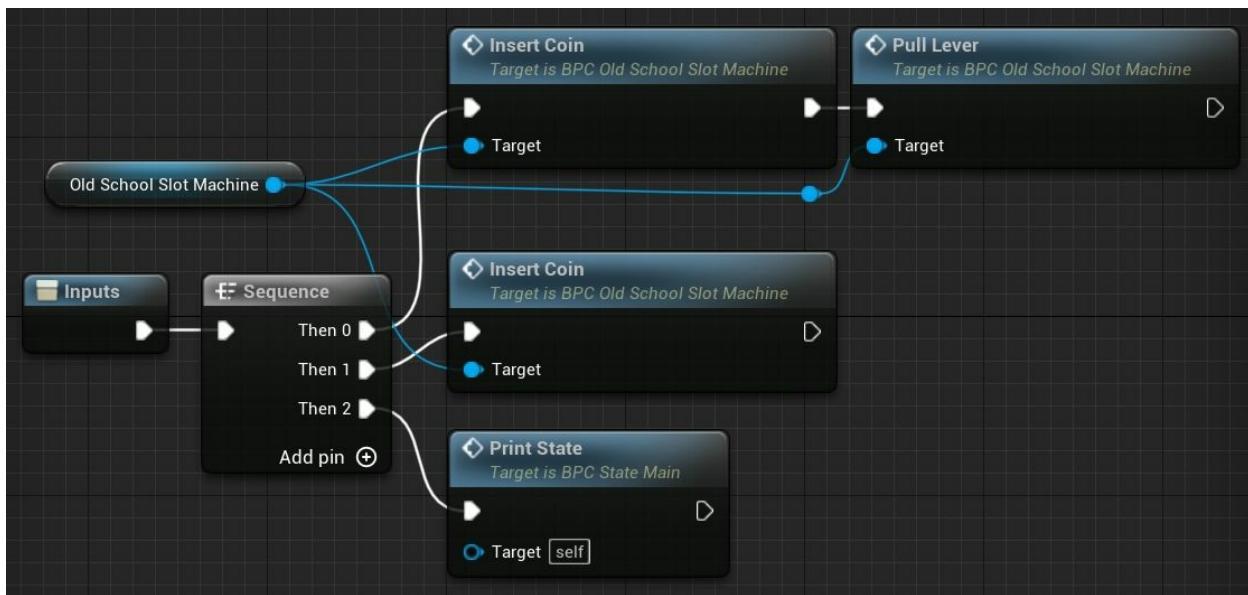
The *Create OldSchoolSlotMachine* collapsed node:



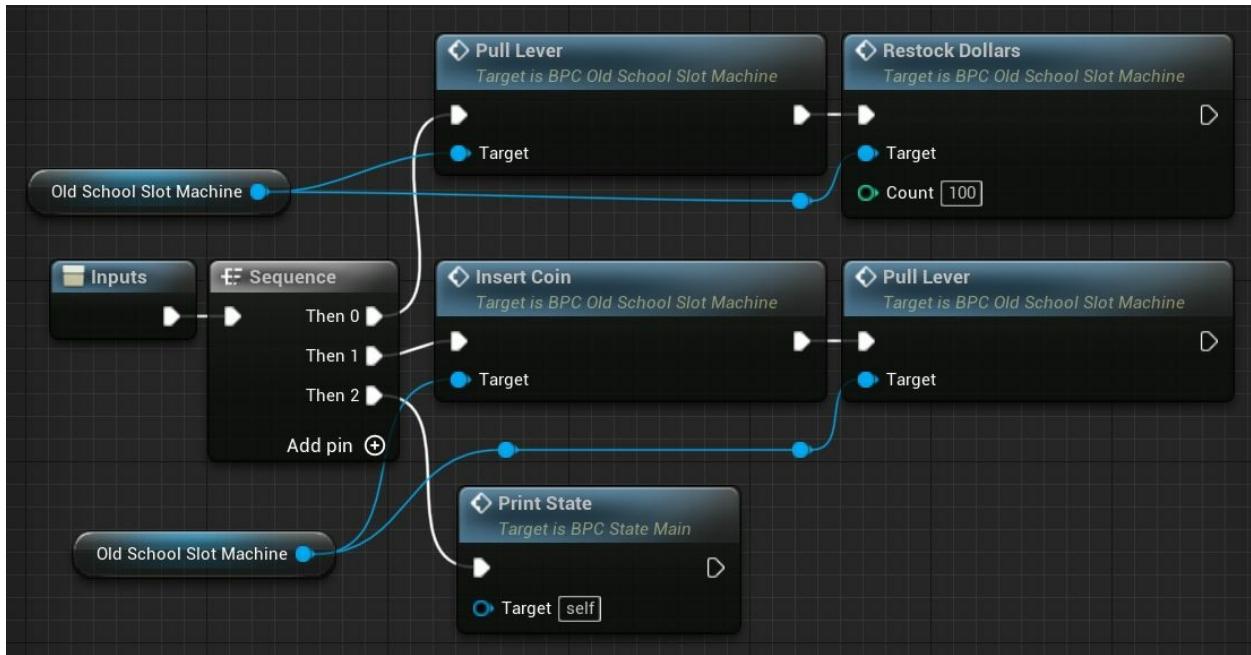
The *OldSchoolSlotMachine Action Sequence\_1* collapsed graph:



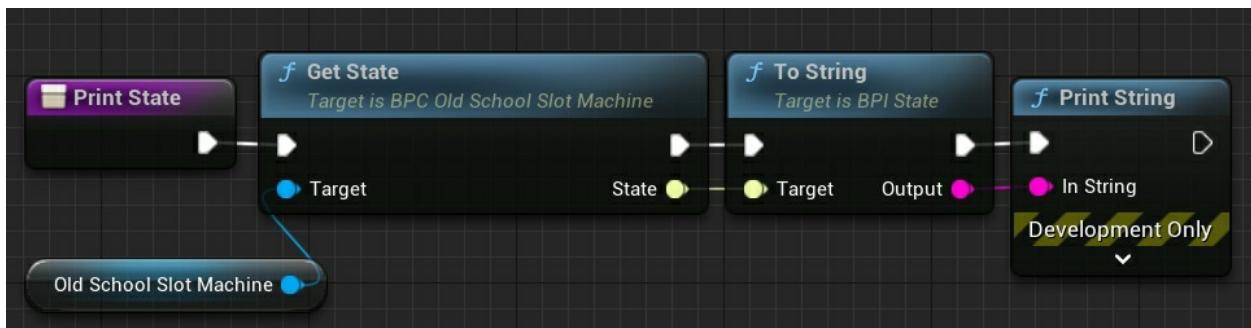
The *OldSchoolSlotMachine Action Sequence\_2* collapsed graph:



The *OldSchoolSlotMachine Action Sequence\_3* collapsed graph:



The *Print Current State* function:



The *OldSchoolSlotMachine* class:

## ***OldSchoolSlotMachine.h***

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "State.h"
#include "OldSchoolSlotMachine.generated.h"

UCLASS()
class DESIGN_PATTERNS_API AOldSchoolSlotMachine : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AOldSchoolSlotMachine();

private:
    //The No Dollars State of this Slot Machine
    IState* NoDollarsState;

    //The No Coin State of this Slot Machine
    IState* NoCoinState;

    //The Coin Inserted State of this Slot Machine
    IState* CoinInsertedState;

    //The Won Dollars State of this Slot Machine
    IState* WonDollarsState;

    //The State of this Slot Machine
    IState* State;

    //The amount of Dollars of this Slot Machine
    int Count = 0;

protected:
```

```
// Called when the game starts or when spawned
virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

    //Initialize this Slot Machine
    void Initialize(int NumberOfDollars);

    //Insert a Coin
    void InsertCoin();

    //Reject a Coin
    void RejectCoin();

    //Pull the lever of this Slot Machine
    void PullLever();

    //Get the current State of this Slot Machine
    IState* GetState();

    //Get the No Dollars State of this Slot Machine
    IState* GetNoDollarsState();

    //Get the No Coin State of this Slot Machine
    IState* GetNoCoinState();

    //Get the Coin Inserted State of this Slot Machine
    IState* GetCoinInsertedState();

    //Get the Won Dollars State of this Slot Machine
    IState* GetWonDollarsState();

    //Get the String of the current State
    FString GetCurrentState();

    //Set the Current State with the passed one
    void SetState(IState* myState);
```

```
//Emit Dollars from this Slot Machine
void EmitDollars();

//Get the amount of dollars in this Slot Machine
int GetCount();

//Refill the dollars by the passed amount
void RestockDollars(int myCount);
};
```

## *OldSchoolSlotMachine.cpp*

```
#include "OldSchoolSlotMachine.h"
#include "NoDollarsState.h"
#include "NoCoinState.h"
#include "CoinInsertedState.h"
#include "WonDollarsState.h"

// Sets default values
AOldSchoolSlotMachine::AOldSchoolSlotMachine()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void AOldSchoolSlotMachine::BeginPlay()
{
    Super::BeginPlay();

}

// Called every frame
void AOldSchoolSlotMachine::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

}

void AOldSchoolSlotMachine::Initialize(int NumberOfDollars)
{
    //Spanw the No Dollars State and set this Slot Machine to it
    NoDollarsState = GetWorld()->SpawnActor<ANoDollarsState>
    (ANoDollarsState::StaticClass());
    NoDollarsState->SetSlotMachine(this);
```

```

//Spanw the No Coin State and set this Slot Machine to it
NoCoinState      =  GetWorld()->SpawnActor<ANoCoinState>
(ANoCoinState::StaticClass());
NoCoinState->SetSlotMachine(this);

//Spanw the Coin Inserted State and set this Slot Machine to it
CoinInsertedState = GetWorld()->SpawnActor<ACoinInsertedState>
(ACoinInsertedState::StaticClass());
CoinInsertedState->SetSlotMachine(this);

//Spanw the Won Dollars State and set this Slot Machine to it
WonDollarsState = GetWorld()->SpawnActor<AWonDollarsState>
(AWonDollarsState::StaticClass());
WonDollarsState->SetSlotMachine(this);

//Set the amount of dollars
Count = NumberOfDollars;

//If its greater than 0, set the current State to No Coin State
if (NumberOfDollars > 0) {
    State = NoCoinState;
}
else {
    State = NoDollarsState;
}
}

void AOldSchoolSlotMachine::InsertCoin()
{
    //Execute the Insert Coin routine based on the current state
    State->InsertCoin();
}

void AOldSchoolSlotMachine::RejectCoin()
{
    //Execute the Reject Coin routine based on the current state
    State->RejectCoin();
}

```

```

void AOldSchoolSlotMachine::PullLever()
{
    //Execute the Pull Lever and Payout routine based on the current state
    State->PullLever();
    State->Payout();
}

void AOldSchoolSlotMachine::EmitDollars()
{
    //You won!
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("Make it Rain!"));

    //If the Slot Machine still have some dollars, decrease them by 50
    if (Count != 0) {
        Count = Count - 50;
    }
}

int AOldSchoolSlotMachine::GetCount()
{
    //Returns the Dollars count
    return Count;
}

void AOldSchoolSlotMachine::RestockDollars(int myCount)
{
    //Add the passed count to the Dollars amount and log it
    Count += myCount;
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
FString::Printf(TEXT("The Old School Slot Machine was just filled and new
dollar count is: %i"), Count));

    //Then execute the Restock Dollars routine based on the current state
    State->RestockDollars();
}

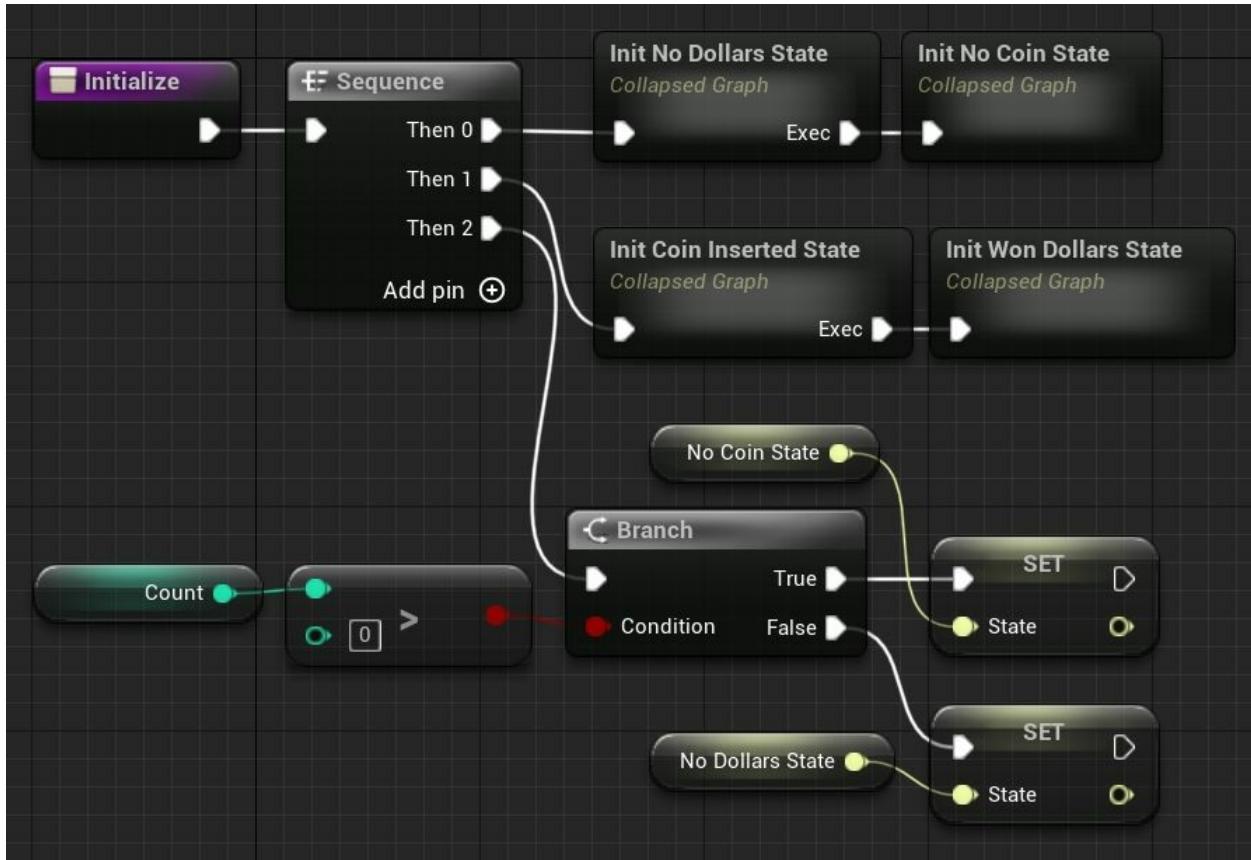
void AOldSchoolSlotMachine::SetState(IState* myState)

```

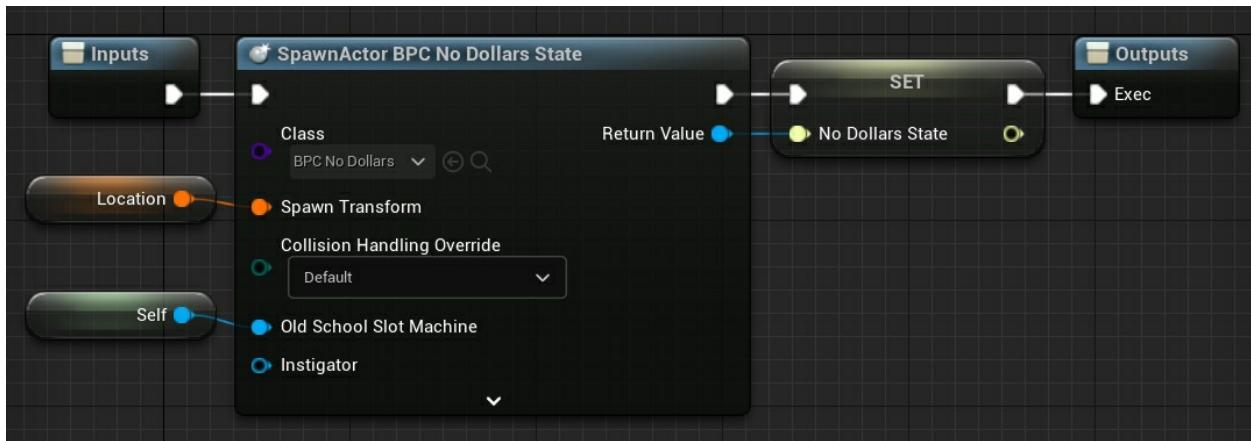
```
{  
    //Set the current state to the passed one  
    State = myState;  
}  
  
IState* AOldSchoolSlotMachine::GetState()  
{  
    //Returns the Current State  
    return State;  
}  
  
IState* AOldSchoolSlotMachine::GetNoDollarsState()  
{  
    //Returns the No Dollars State  
    return NoDollarsState;  
}  
  
IState* AOldSchoolSlotMachine::GetNoCoinState()  
{  
    //Returns the No Coin State  
    return NoCoinState;  
}  
  
IState* AOldSchoolSlotMachine::GetCoinInsertedState()  
{  
    //Returns the Coin Inserted State  
    return CoinInsertedState;  
}  
  
IState* AOldSchoolSlotMachine::GetWonDollarsState()  
{  
    //Returns the Won Dollars State  
    return WonDollarsState;  
}  
  
FString AOldSchoolSlotMachine::GetCurrentState()  
{  
    //Returns the 'Name' of current State
```

```
    return "Current Slot Machine State: " + State->ToString();  
}
```

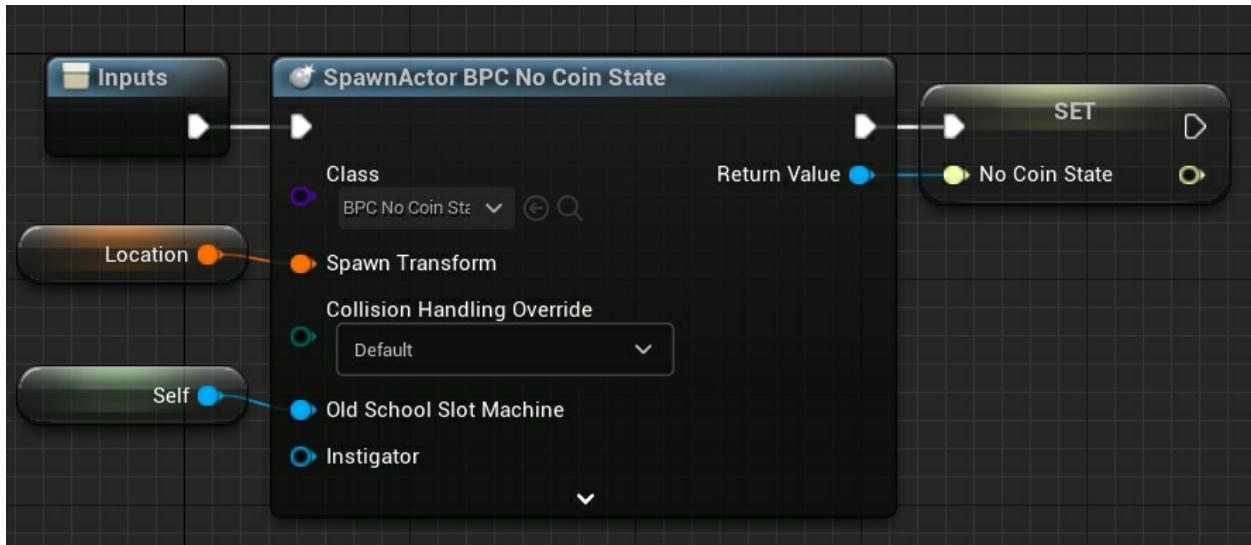
The *OldSchoolSlotMachine Initialize* function:



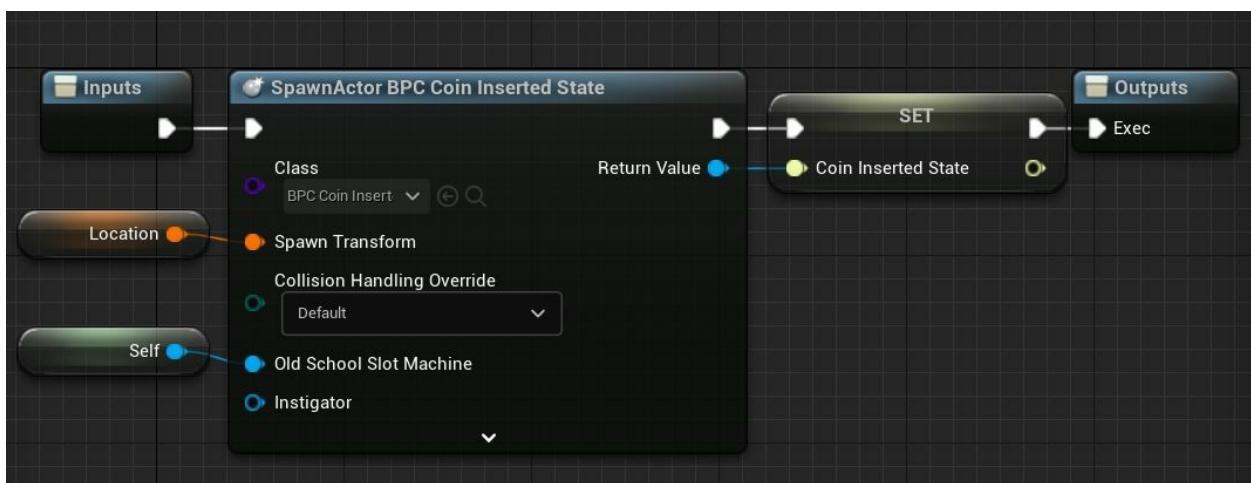
The *Init No Dollars State* collapsed graph:



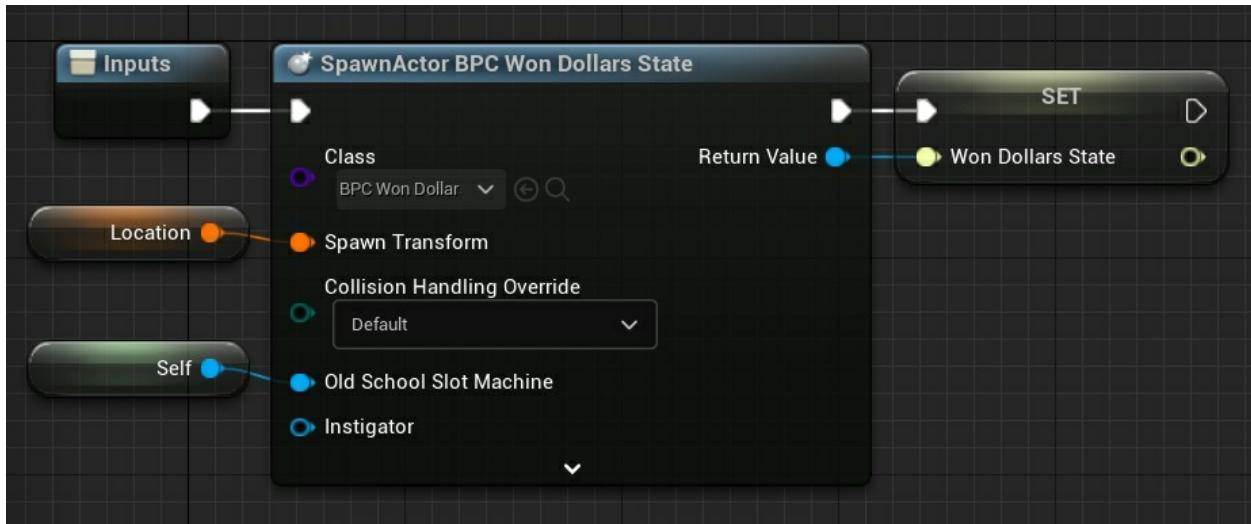
The *Init No Coin State* collapsed graph:



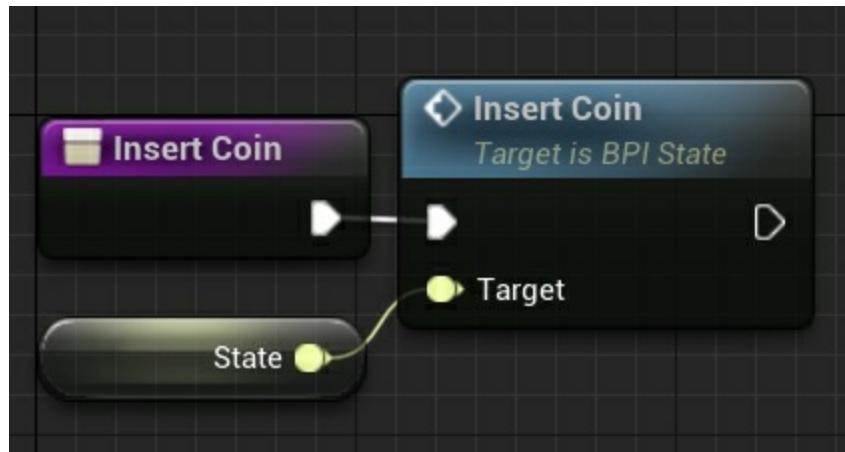
The *Init Coin Inserted State* collapsed graph:



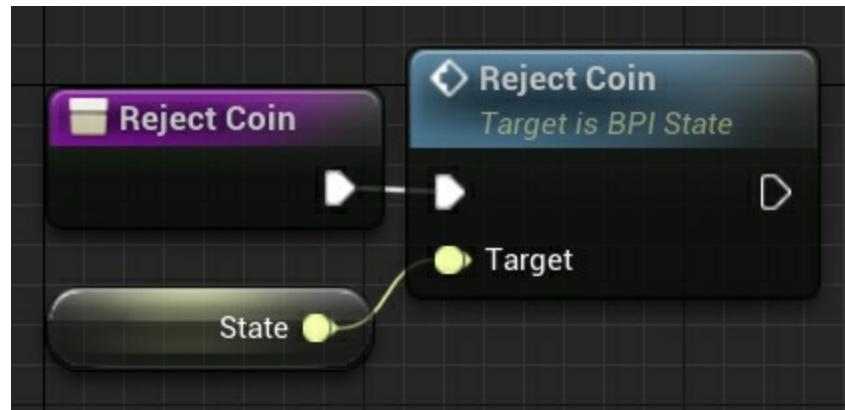
The *Init Won Dollars State* collapsed graph:



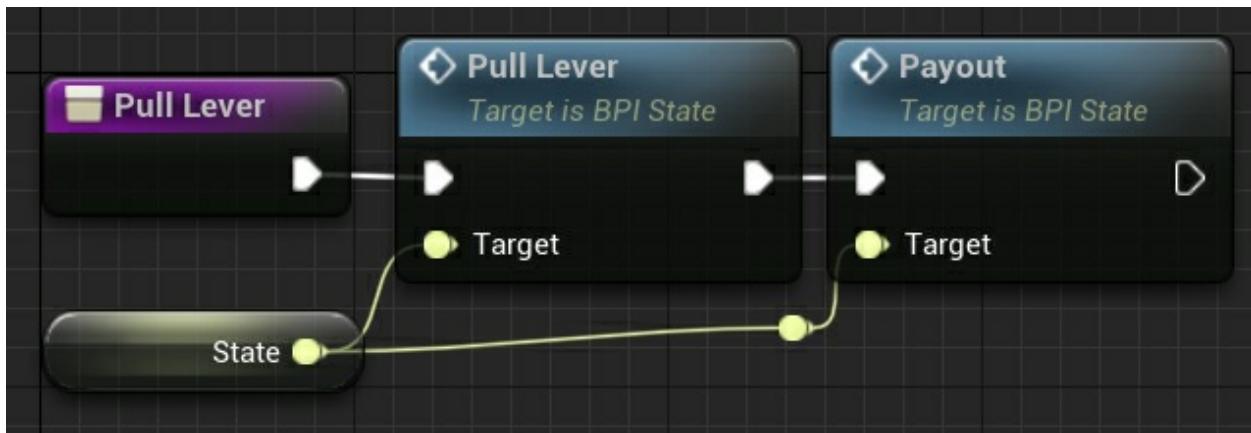
The *InsertCoin* function:



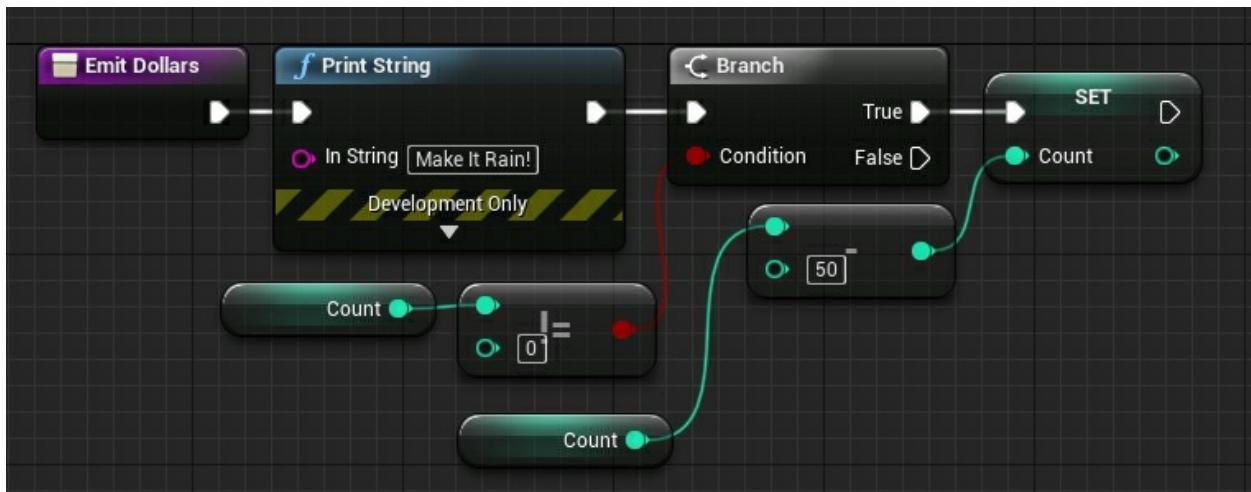
The *RejectCoin* function:



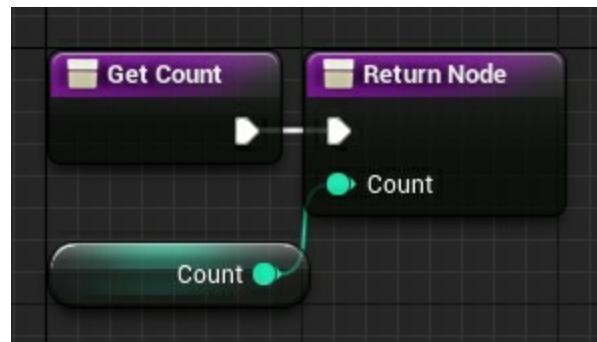
The *PullLever* function:



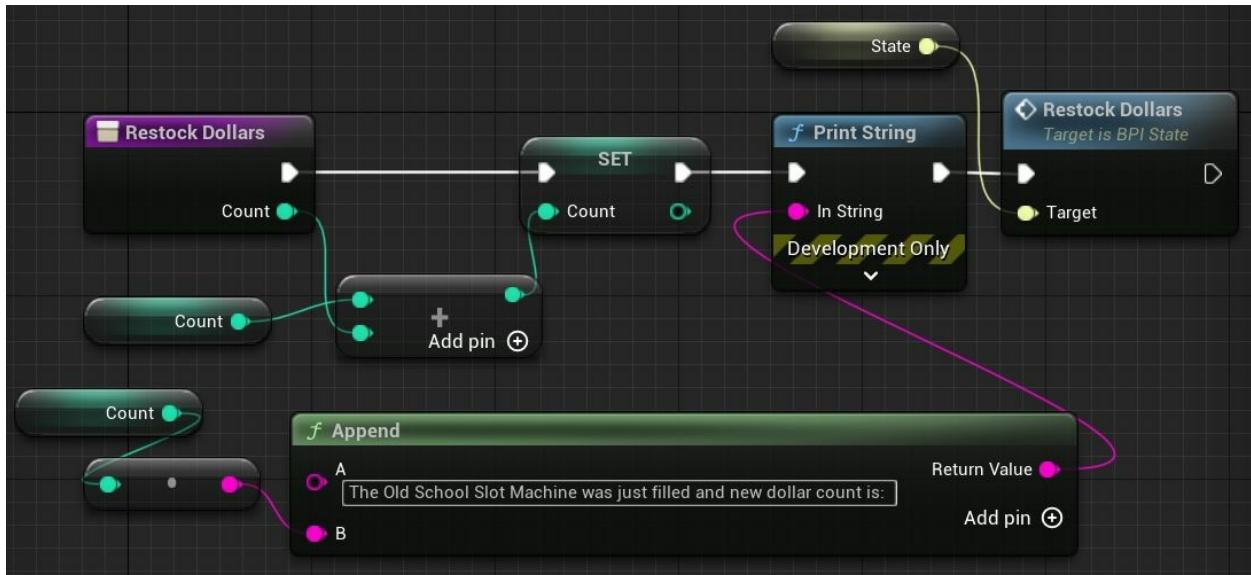
The *EmitDollars* function:



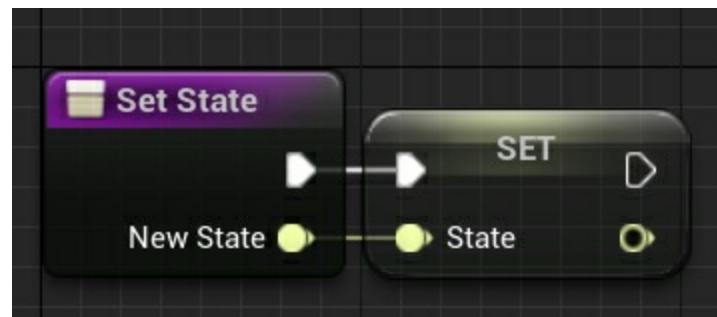
The *GetCount* function:



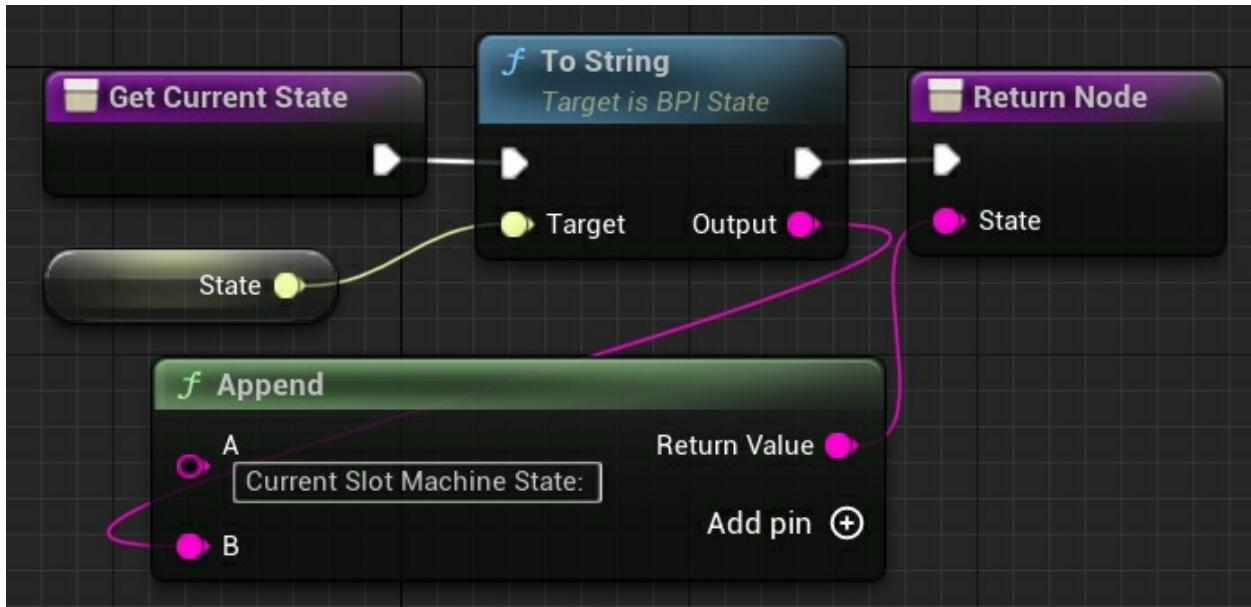
The *RestockDollars* function:



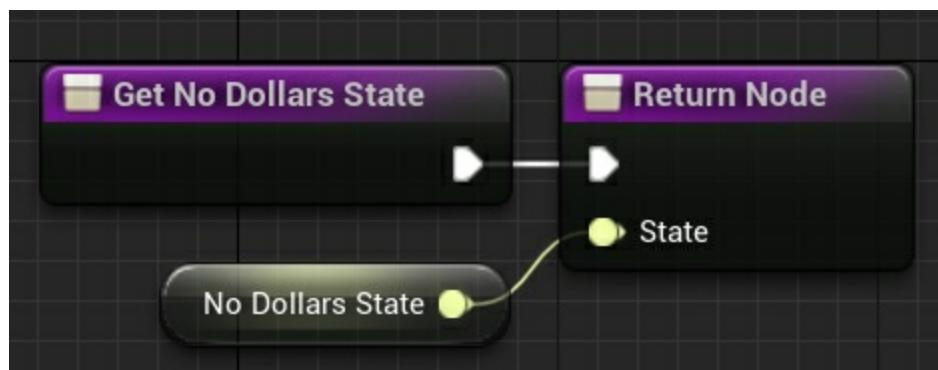
The *setState* function:



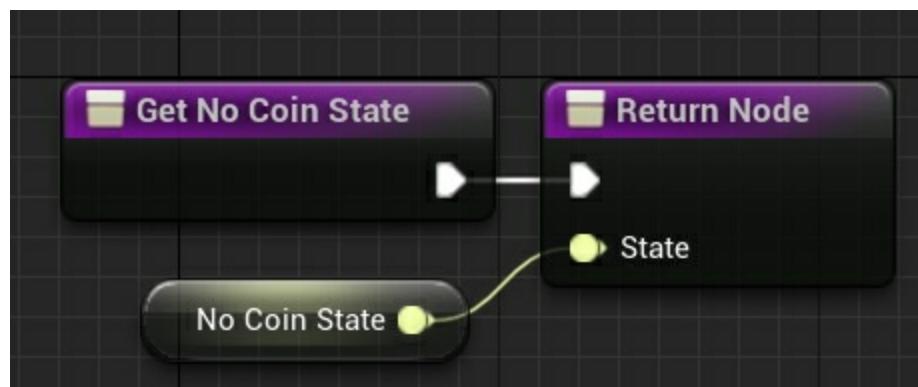
The *GetCurrentState* function:



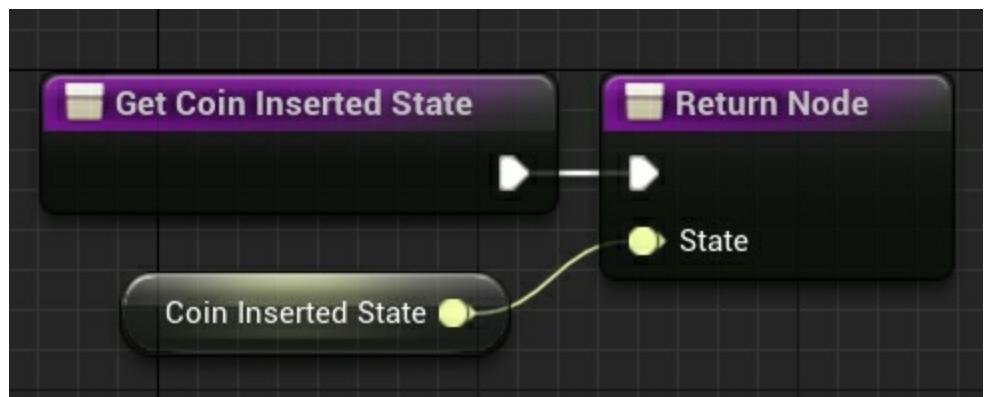
The *GetNoDollarsState* function:



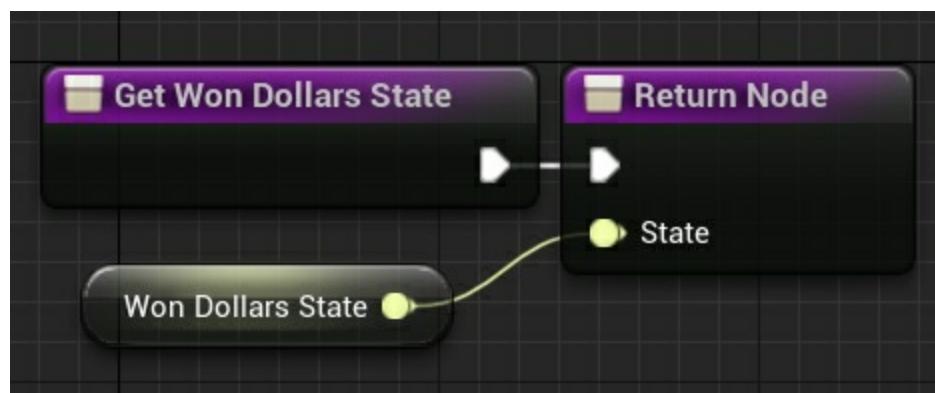
The *GetNoCoinState* function:



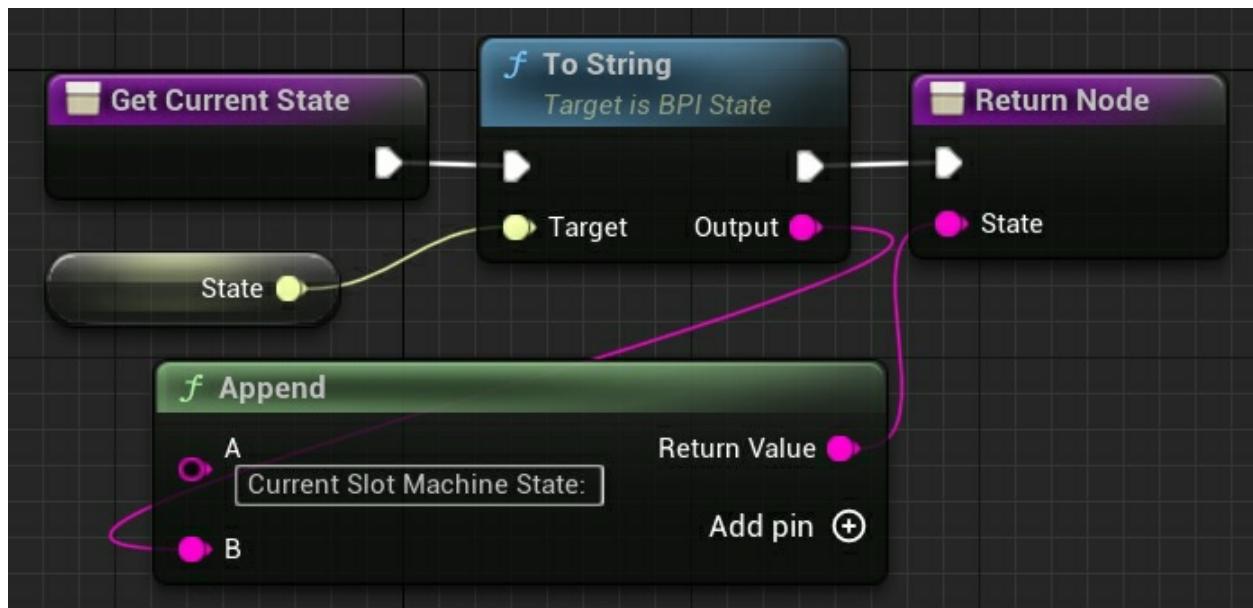
The *GetCoinInsertedState* function:



The *GetWonDollarsState* function:



The *GetCurrentState* function:



The *NoDollarsState* class:

***NoDollarsState.h***

```
#pragma once
```

```
#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "State.h"
#include "NoDollarsState.generated.h"
```

```
UCLASS()
class DESIGN_PATTERNS_API ANoDollarsState : public AActor, public
```

```
IState
```

```
{
```

```
    GENERATED_BODY()
```

```
public:
```

```
    // Sets default values for this actor's properties
```

```
    ANoDollarsState();
```

```
private:
```

```
    //The Slot Machine of this State
```

```
    UPROPERTY()
        class AOldSchoolSlotMachine* OldSchoolSlotMachine;
```

```
protected:
```

```
    // Called when the game starts or when spawned
```

```
    virtual void BeginPlay() override;
```

```
public:
```

```
    // Called every frame
```

```
    virtual void Tick(float DeltaTime) override;
```

```
    //Execute the routine when a coin is inserted
```

```
    virtual void InsertCoin() override;
```

```
    //Execute the routine when a coin is rejected
```

```
virtual void RejectCoin() override;

//Execute the routine when the lever is pulled
virtual void PullLever() override;

//Execute the routine of the payout
virtual void Payout() override;

//Restock the Slot Machine
virtual void RestockDollars() override;

//Get the String this State
virtual FString ToString() override;

//Set the Slot Machine of this state
virtual void SetSlotMachine(class AOldSchoolSlotMachine* SlotMachine)
override;
};
```

## **NoDollarsState.cpp**

```
#include "NoDollarsState.h"
#include "OldSchoolSlotMachine.h"

// Sets default values
ANoDollarsState::ANoDollarsState()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void ANoDollarsState::BeginPlay()
{
    Super::BeginPlay();

}

// Called every frame
void ANoDollarsState::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

}

void ANoDollarsState::SetSlotMachine(class AOldSchoolSlotMachine*
SlotMachine)
{
    //Set the Slot Machine of this state
    OldSchoolSlotMachine = SlotMachine;
}

void ANoDollarsState::InsertCoin()
{
    //Log the Insert Coin string
```

```

    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("You cannot insert a casino coin, the machine is out of money"));
}

void ANoDollarsState::RejectCoin()
{
    //Log the Reject Coin string
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("You have not inserted a casino coin"));
}

void ANoDollarsState::PullLever()
{
    //Log the Pull Lever string
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("You turned, but there is NO money in the machine"));
}

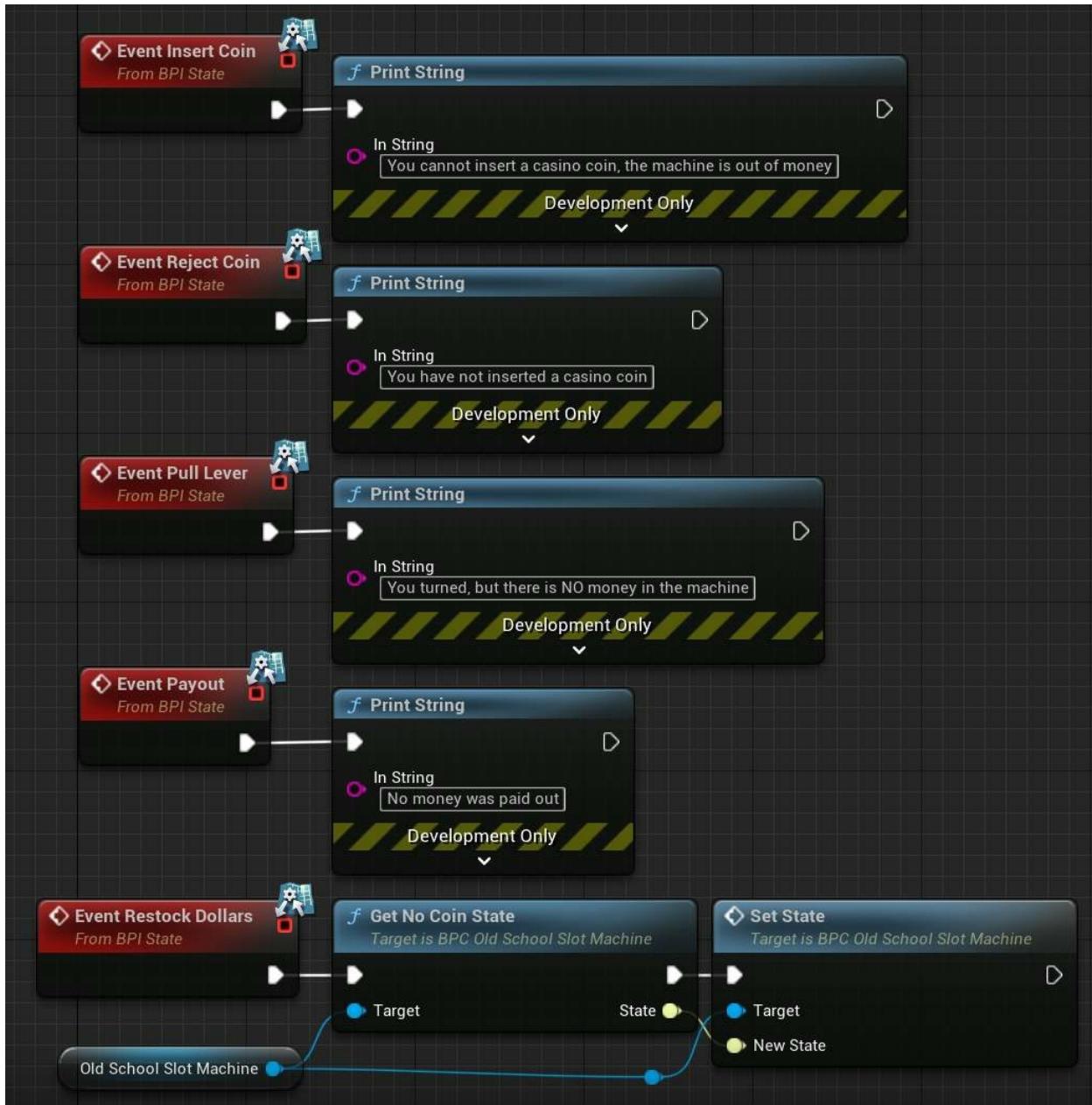
void ANoDollarsState::Payout()
{
    //Log the Payout string
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("No money was paid out"));
}

void ANoDollarsState::RestockDollars()
{
    //Set the state to "No Coin State"
    OldSchoolSlotMachine->SetState(OldSchoolSlotMachine-
>GetNoCoinState());
}

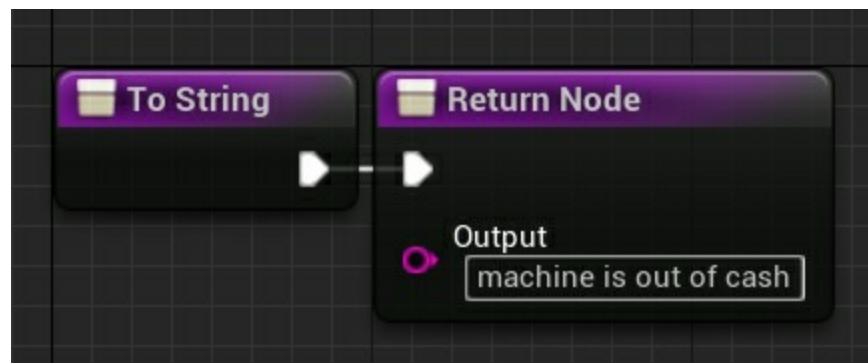
FString ANoDollarsState::ToString()
{
    //Return the String of this state
    return "machine is out of cash";
}

```

The *NoDollarsState* event graph:



The *ToString* function:



The *NoCoinState* class:

## **NoCoinState.h**

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "State.h"
#include "NoCoinState.generated.h"

UCLASS()
class DESIGN_PATTERNS_API ANoCoinState : public AActor, public IState
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    ANoCoinState();

private:
    //The Slot Machine of this State
    UPROPERTY()
        class AOldSchoolSlotMachine* OldSchoolSlotMachine;

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

    //Execute the routine when a coin is inserted
    virtual void InsertCoin() override;

    //Execute the routine when a coin is rejected
    virtual void RejectCoin() override;
```

```
//Execute the routine when the lever is pulled
virtual void PullLever() override;

//Execute the routine of the payout
virtual void Payout() override;

//Restock the Slot Machine
virtual void RestockDollars() override;

//Get the String this State
virtual FString ToString() override;

//Set the Slot Machine of this state
virtual void SetSlotMachine(class AOldSchoolSlotMachine* SlotMachine)
override;
};
```

## **NoCoinState.cpp**

```
#include "NoCoinState.h"
#include "OldSchoolSlotMachine.h"

// Sets default values
ANoCoinState::ANoCoinState()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void ANoCoinState::BeginPlay()
{
    Super::BeginPlay();

}

// Called every frame
void ANoCoinState::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

}

void ANoCoinState::SetSlotMachine(class AOldSchoolSlotMachine*
SlotMachine)
{
    //Set the Slot Machine of this state
    OldSchoolSlotMachine = SlotMachine;
}

void ANoCoinState::InsertCoin()
{
    //Log the Insert Coin string and set the state to "Coin Inserted State"
```

```

        GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("You inserted a casino coin"));
        OldSchoolSlotMachine->SetState(OldSchoolSlotMachine-
>GetCoinInsertedState());
}

void ANoCoinState::RejectCoin()
{
    //Log the Reject Coin string
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("You haven't inserted a casino coin"));
}

void ANoCoinState::PullLever()
{
    //Log the Pull Lever string
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("You turned, but there's no casino coin"));
}

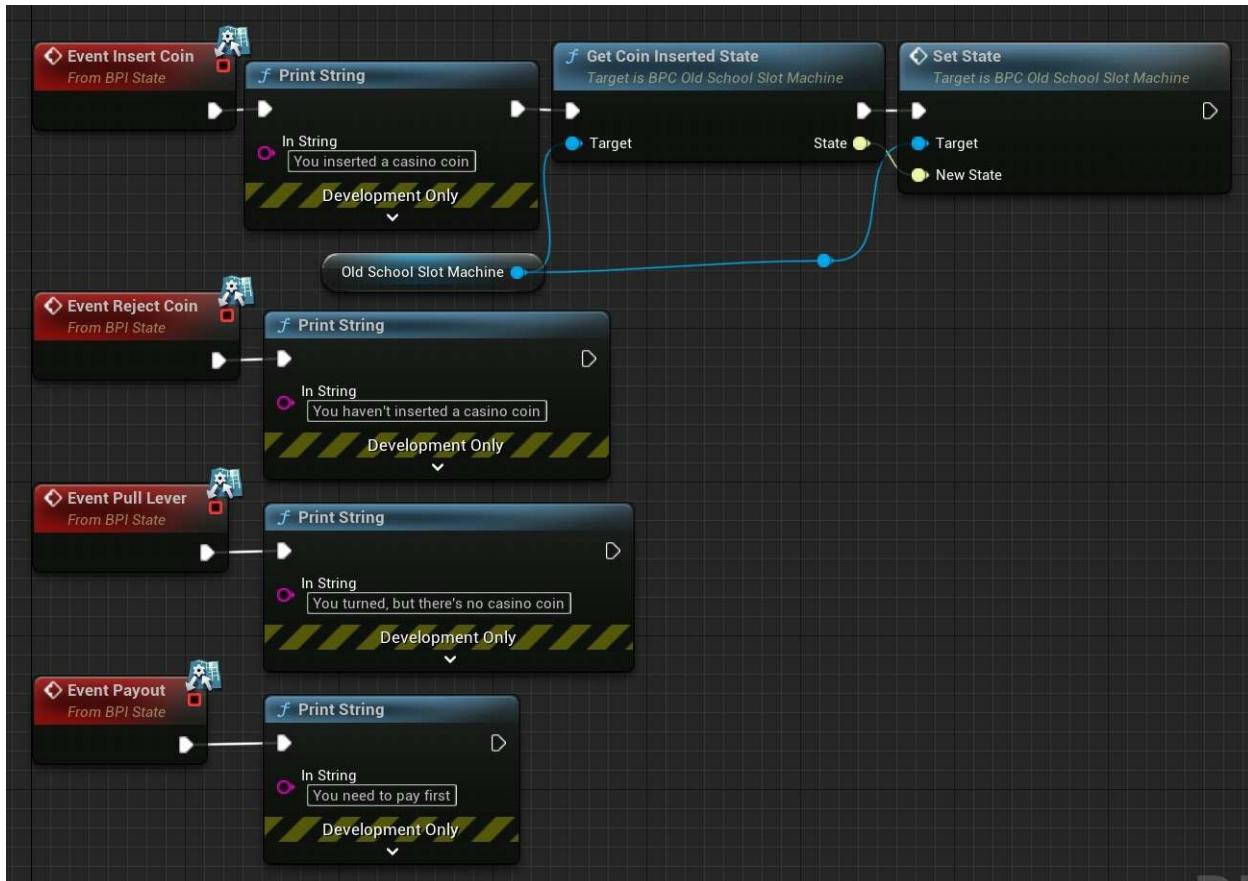
void ANoCoinState::Payout()
{
    //Log the Payout string
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("You need to pay first"));
}

void ANoCoinState::RestockDollars() { /*Nothing*/ }

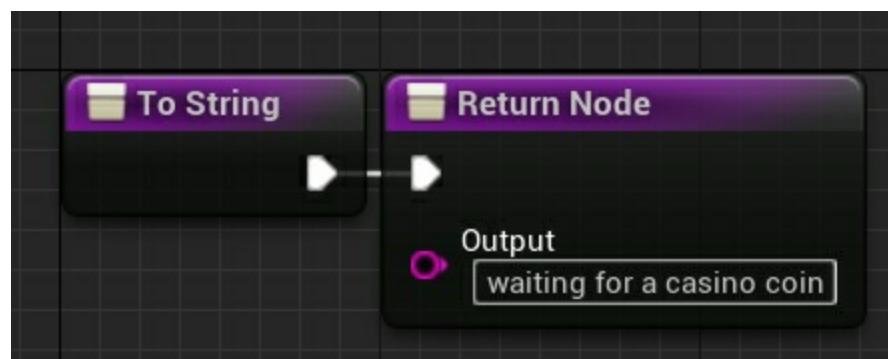
FString ANoCoinState::ToString()
{
    //Return the String of this state
    return "waiting for a casino coin";
}

```

The *NoCoinState* event graph:



The *ToString* function:



The *CoinInsertedState* class:

## ***CoinInsertedState.h***

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "State.h"
#include "CoinInsertedState.generated.h"

UCLASS()
class DESIGN_PATTERNS_API ACoinInsertedState : public AActor,
public IState
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    ACoinInsertedState();

private:
    //The Slot Machine of this State
    UPROPERTY()
        class AOldSchoolSlotMachine* OldSchoolSlotMachine;

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

    //Execute the routine when a coin is inserted
    virtual void InsertCoin() override;

    //Execute the routine when a coin is rejected
    virtual void RejectCoin() override;
```

```
//Execute the routine when the lever is pulled
virtual void PullLever() override;

//Execute the routine of the payout
virtual void Payout() override;

//Restock the Slot Machine
virtual void RestockDollars() override;

//Get the String this State
virtual FString ToString() override;

//Set the Slot Machine of this state
virtual void SetSlotMachine(class AOldSchoolSlotMachine* SlotMachine)
override;
};
```

## ***CoinInsertedState.cpp***

```
#include "CoinInsertedState.h"
#include "OldSchoolSlotMachine.h"

// Sets default values
ACoinInsertedState::ACoinInsertedState()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void ACoinInsertedState::BeginPlay()
{
    Super::BeginPlay();

}

// Called every frame
void ACoinInsertedState::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

}

void ACoinInsertedState::SetSlotMachine(AOldSchoolSlotMachine*
SlotMachine)
{
    //Set the Slot Machine of this state
    OldSchoolSlotMachine = SlotMachine;
}

void ACoinInsertedState::InsertCoin()
{
    //Log the Insert Coin string
```

```

GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("You cannot insert another casino coin"));
}

void ACoinInsertedState::RejectCoin()
{
    //Log the Reject Coin string and set the state to "No Coin State"
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("Casino coin returned"));
    OldSchoolSlotMachine->SetState(OldSchoolSlotMachine-
>GetNoCoinState());
}

void ACoinInsertedState::PullLever()
{
    //Log the Pull Lever string and set the state to "Won Dollars State"
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("You pulled the lever..."));
    OldSchoolSlotMachine->SetState(OldSchoolSlotMachine-
>GetWonDollarsState());
}

void ACoinInsertedState::Payout()
{
    //Log the Payout string
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("No dollars dispensed"));
}

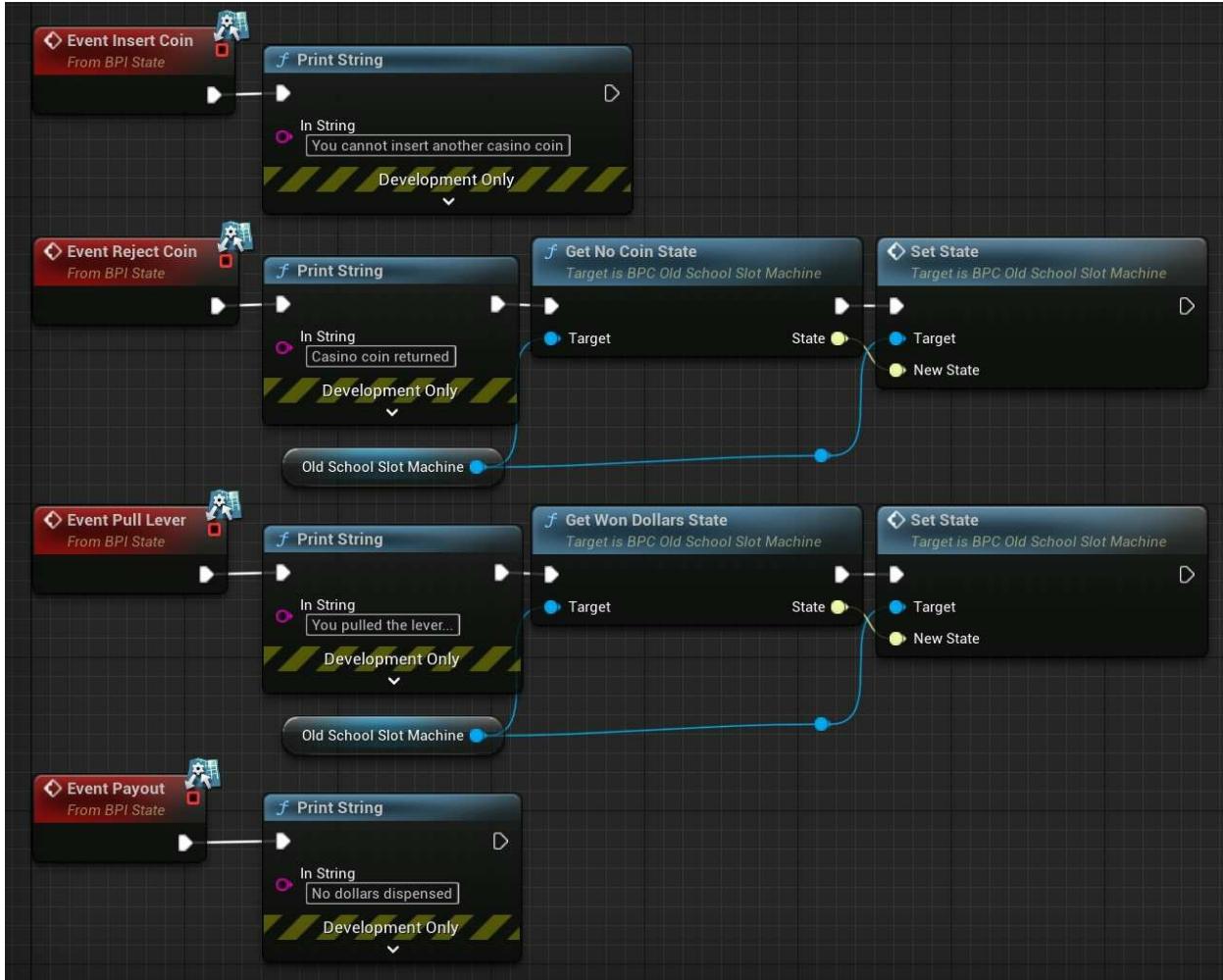
void ACoinInsertedState::RestockDollars() /*Nothing*/
}

FString ACoinInsertedState::ToString()
{
    //Return the String of this state
    return "waiting for lever pull";
}

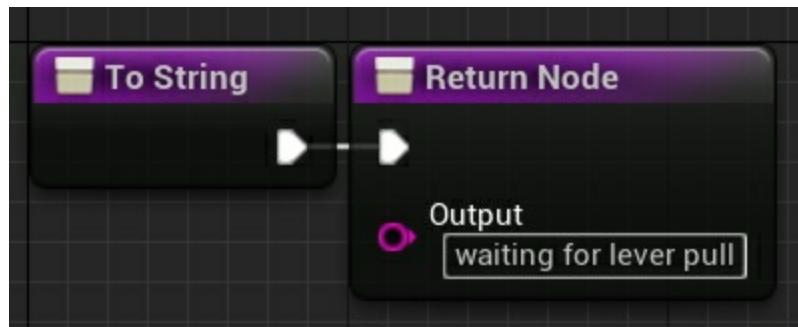
```

The *CoinInsertedState* blueprint event graph:





The *ToString* function:



The *WonDollarsState* class:

## ***WonDollarsState.h***

```
#pragma once
```

```
#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "State.h"
#include "WonDollarsState.generated.h"
```

```
UCLASS()
```

```
class DESIGN_PATTERNS_API AWonDollarsState : public AActor, public
IState
{
    GENERATED_BODY()
```

```
public:
```

```
    // Sets default values for this actor's properties
    AWonDollarsState();
```

```
private:
```

```
    //The Slot Machine of this State
```

```
    UPROPERTY()
```

```
        class AOldSchoolSlotMachine* OldSchoolSlotMachine;
```

```
protected:
```

```
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;
```

```
public:
```

```
    // Called every frame
```

```
    virtual void Tick(float DeltaTime) override;
```

```
    //Execute the routine when a coin is inserted
```

```
    virtual void InsertCoin() override;
```

```
    //Execute the routine when a coin is rejected
```

```
    virtual void RejectCoin() override;
```

```
//Execute the routine when the lever is pulled
virtual void PullLever() override;

//Execute the routine of the payout
virtual void Payout() override;

//Restock the Slot Machine
virtual void RestockDollars() override;

//Get the String this State
virtual FString ToString() override;

//Set the Slot Machine of this state
virtual void SetSlotMachine(class AOldSchoolSlotMachine* SlotMachine)
override;
};
```

## ***WonDollarsState.cpp***

```
#include "WonDollarsState.h"
#include "OldSchoolSlotMachine.h"

// Sets default values
AWonDollarsState::AWonDollarsState()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void AWonDollarsState::BeginPlay()
{
    Super::BeginPlay();

}

// Called every frame
void AWonDollarsState::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

}

void AWonDollarsState::SetSlotMachine(AOldSchoolSlotMachine*
SlotMachine)
{
    //Set the Slot Machine of this state
    OldSchoolSlotMachine = SlotMachine;
}

void AWonDollarsState::InsertCoin()
{
    //Log the Insert Coin string
```

```

        GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("Please wait, machine is dispensing dollars"));
}

void AWonDollarsState::RejectCoin()
{
    //Log the Reject Coin string
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("Chill, you already pulled the lever"));
}

void AWonDollarsState::PullLever()
{
    //Log the Pull Lever string
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("Do not pull lever twice!"));
}

void AWonDollarsState::Payout()
{
    //Payout time! Emit the dollars from the Slot Machine
    OldSchoolSlotMachine->EmitDollars();

    if (OldSchoolSlotMachine->GetCount() > 0)
    {
        //If the Slot Machine still have dollars, set its state to No Coin State
        OldSchoolSlotMachine->SetState(OldSchoolSlotMachine-
>GetNoCoinState());
    }
    else
    {
        //If the Slot Machine doesn't have any dollars left, log it and set its state
        //to No Dollars State
        GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("Machine is now out of cash"));
        OldSchoolSlotMachine->SetState(OldSchoolSlotMachine-
>GetNoDollarsState());
    }
}

```

}

void AWonDollarsState::RestockDollars() { /\*Nothing\*/ }

FString AWonDollarsState::ToString()

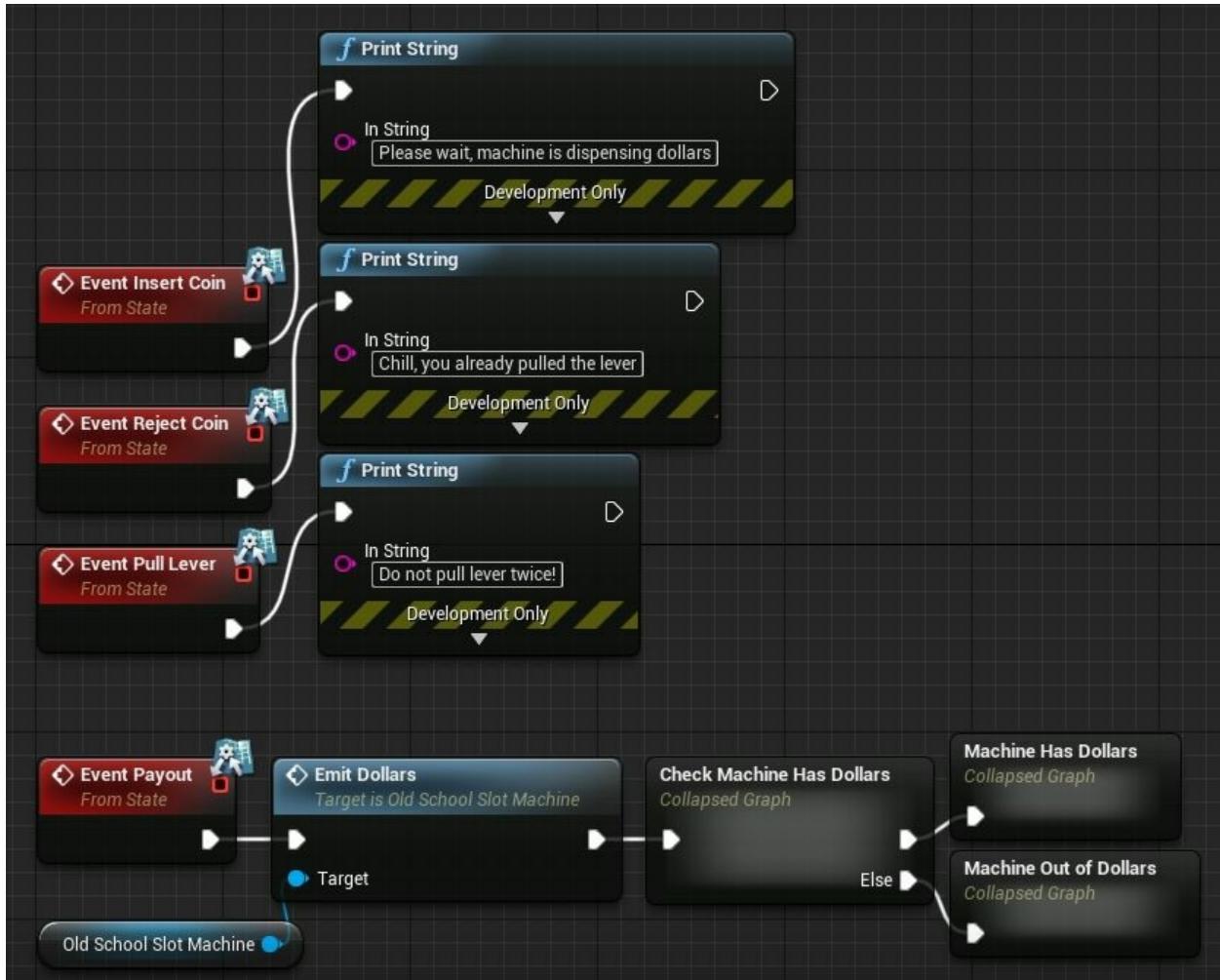
{

//Return the String of this state

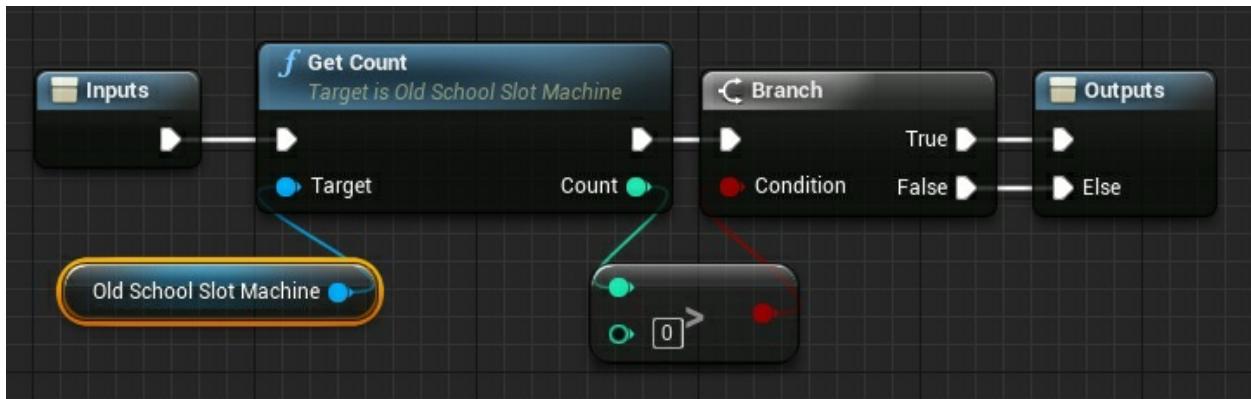
return "dispensing cash";

}

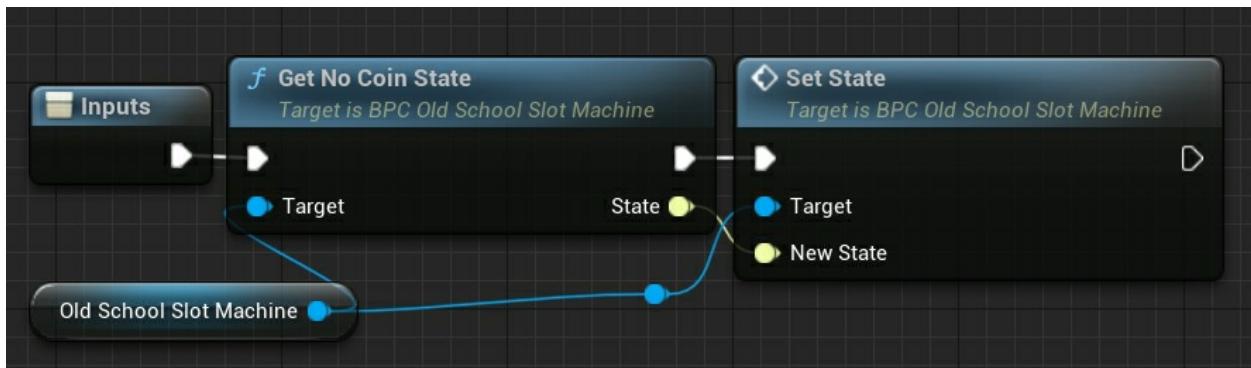
The *WonDollarsState* event graph:



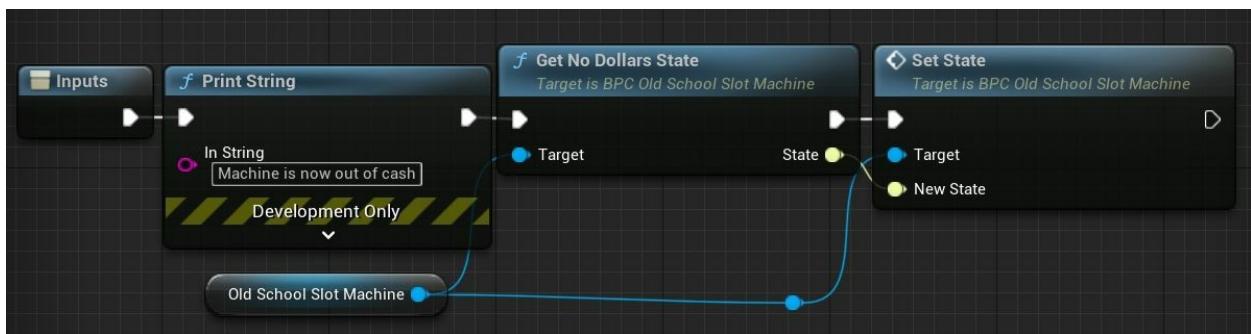
The *Check Machine Has Dollars* collapsed graph:



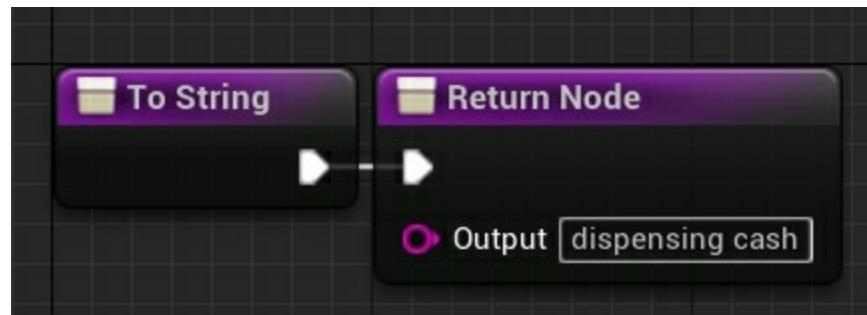
The *Machine Has Dollars* collapsed graph:



The *Machine Out of Dollars* collapsed graph collapsed graph:



The *ToString* function:



The *State* interface:

## ***State.h***

```
#pragma once

#include "CoreMinimal.h"
#include "UObject/Interface.h"
#include "State.generated.h"

// This class does not need to be modified.
UINTERFACE(MinimalAPI)
class UState : public UInterface
{
    GENERATED_BODY()
};

class DESIGN_PATTERNS_API IState
{
    GENERATED_BODY()

    // Add interface functions to this class. This is the class that will be
    // inherited to implement this interface.

public:

    //The pure virtual functions of the State
    virtual void InsertCoin() = 0;
    virtual void RejectCoin() = 0;
    virtual void PullLever() = 0;
    virtual void Payout() = 0;
    virtual void RestockDollars() = 0;
    virtual FString ToString() = 0;

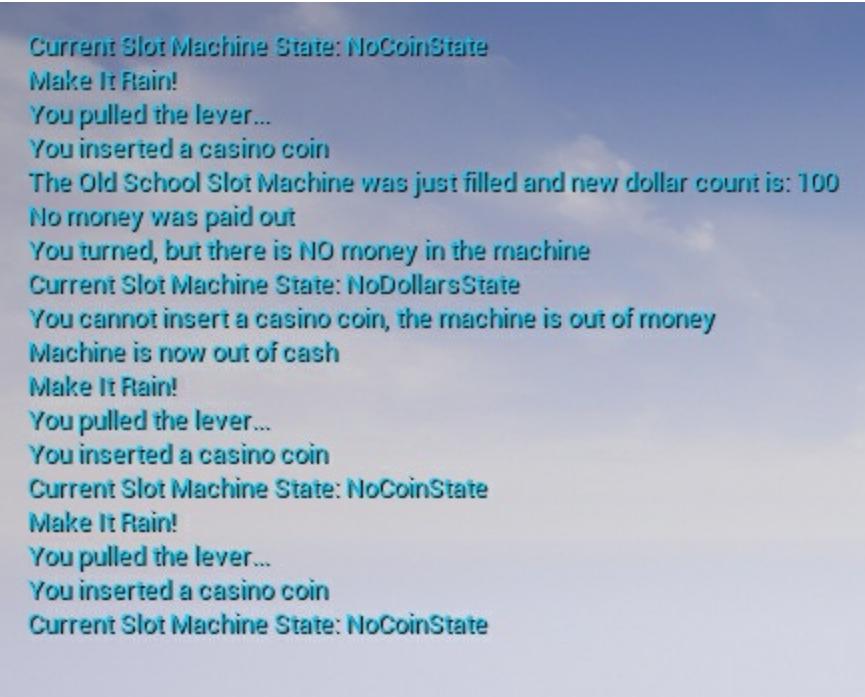
    virtual void SetSlotMachine(class AOldSchoolSlotMachine* SlotMachine)
= 0;
};
```

## *State.cpp*

```
#include "State.h"

// Add default functionality here for any IState functions that are not pure
virtual.
```

The State pattern viewport print:



```
Current Slot Machine State: NoCoinState
Make It Rain!
You pulled the lever...
You inserted a casino coin
The Old School Slot Machine was just filled and new dollar count is: 100
No money was paid out
You turned, but there is NO money in the machine
Current Slot Machine State: NoDollarsState
You cannot insert a casino coin, the machine is out of money
Machine is now out of cash
Make It Rain!
You pulled the lever...
You inserted a casino coin
Current Slot Machine State: NoCoinState
Make It Rain!
You pulled the lever...
You inserted a casino coin
Current Slot Machine State: NoCoinState
```

# You Sunk My Battleship... Strategy Pattern

The GoF defines the Strategy pattern as a “policy” (Gamma et.al, 1995). We would like to have a family of algorithms be interchangeable and vary independently. This will become more clear with the following example.

## **The Good**

- Provides an alternative to subclassing (Gamma et.al, 1995).

## **The Not So Good**

- The numbers of object may increase undesirably.

## **Strategy Pattern Implementation**

We have a strategy for combating different enemies at sea. The brute force strategy is used if the threat is small. We use the divide and conquer strategy if there are many enemies. The retreat strategy is executed if the battle is deemed futile. The *Strategy\_Main* class:

## ***Strategy\_Main.h***

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "Strategy_Main.generated.h"

UCLASS()
class DESIGN_PATTERNS_API AStrategy_Main : public AActor
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    AStrategy_Main();

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

};
```

## **Strategy\_Main.cpp**

```
#include "Strategy_Main.h"
#include "BattleShip.h"
#include "BruteForceStrategy.h"
#include "DivideConquerStrategy.h"
#include "RetreatStrategy.h"

// Sets default values
AStrategy_Main::AStrategy_Main()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void AStrategy_Main::BeginPlay()
{
    Super::BeginPlay();

    //Enemies alert log
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("A tiny frigate wants some trouble"));

    //Spawn the Battle Ship
    ABattleShip* BattleShip = GetWorld()->SpawnActor<ABattleShip>
(ABattleShip::StaticClass());

    //Create the Brute Force Strategy and set it to the Battle Ship
    ABruteForceStrategy* BruteForceStrategy = GetWorld()->SpawnActor<ABruteForceStrategy>(ABruteForceStrategy::StaticClass());
    BattleShip->AlterManeuvers(BruteForceStrategy);

    //Engage with the current Strategy
    BattleShip->Engage();
}
```

```

//Enemies alert log
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("Four tiny frigates want some trouble"));

//Create the Divide Conquer Strategy and set it to the Battle Ship
    ADivideConquerStrategy* DivideConquerStrategy = GetWorld()->SpawnActor<ADivideConquerStrategy>(ADivideConquerStrategy::StaticClass());
    BattleShip->AlterManeuvers(DivideConquerStrategy);

//Engage with the current Strategy
BattleShip->Engage();

//Enemies alert log
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
TEXT("An aircraft carrier group wants some trouble"));

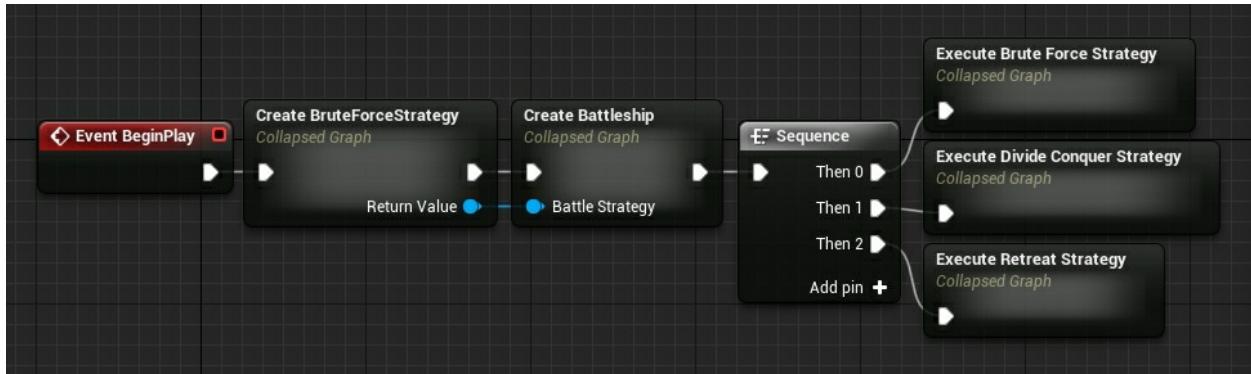
//Create the Retreat Strategy and set it to the Battle Ship
    ARetreatStrategy* RetreatStrategy = GetWorld()->SpawnActor<ARetreatStrategy>(ARetreatStrategy::StaticClass());
    BattleShip->AlterManeuvers(RetreatStrategy);

//Engage with the current Strategy
BattleShip->Engage();
}

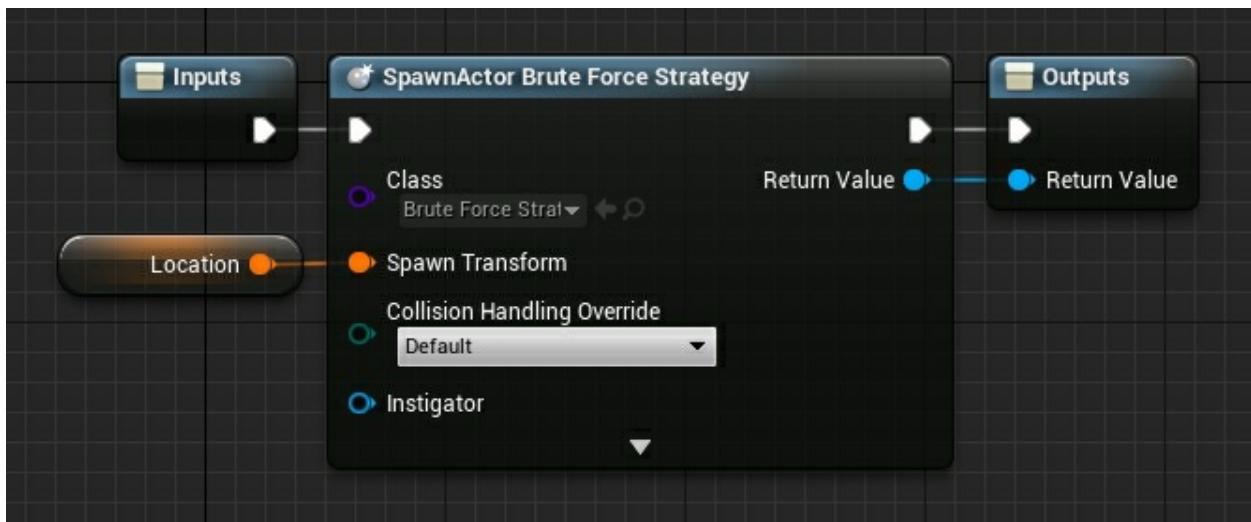
// Called every frame
void AStrategy_Main::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);
}

```

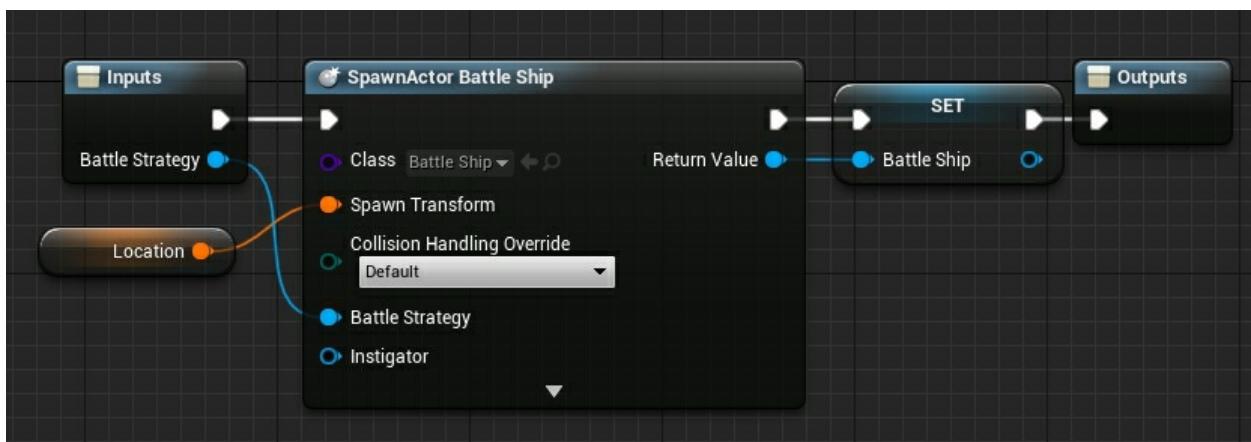
The *Strategy\_Main* blueprint event graph:



The *Create BruteForceStrategy* collapsed graph:



The *Create Battleship* collapsed graph:



The *Execute Brute Force Strategy* collapsed graph:



The *Execute Divide Conquer Strategy* collapsed graph:



The *Execute Retreat Strategy* collapsed graph:



The *BattleShip* class:

## ***BattleShip.h***

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "BattleShipStrategy.h"
#include "BattleShip.generated.h"

UCLASS()
class DESIGN_PATTERNS_API ABattleShip : public AActor
{
    GENERATED_BODY()

public:
    ABattleShip();

private:

//The current Battle Strategy
IBattleShipStrategy* BattleStrategy;

protected:
    virtual void BeginPlay() override;

public:
    virtual void Tick(float DeltaTime) override;

    // Change the Maneuver and set "BattleStrategy" variable
    void AlterManeuvers(AActor* myBattleStrategy);

    // Engage with the current Battle Strategy
    void Engage();

};

};
```

## **BattleShip.cpp**

```
#include "BattleShip.h"

// Sets default values
ABattleShip::ABattleShip()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void ABattleShip::BeginPlay()
{
    Super::BeginPlay();

}

// Called every frame
void ABattleShip::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

}

void ABattleShip::AlterManeuvers(AActor* myBattleStrategy)
{
    //Try to cast the passed Strategy and set it to the current one
    BattleStrategy = Cast<IBattleShipStrategy>(myBattleStrategy);

    //Log Error if the cast failed
    if (!BattleStrategy)
    {
        GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Red,
        TEXT("Invalid Cast! See Output log for more details"));
        UE_LOG(LogTemp, Error, TEXT("AlterManeuvers(): The Actor is not
```

```
a BattleShipStrategy! Are you sure that the Actor implements that  
interface?"));
```

```
}
```

```
void ABattleShip::Engage()
```

```
{
```

```
    //Log Error if the current Strategy is NULL
```

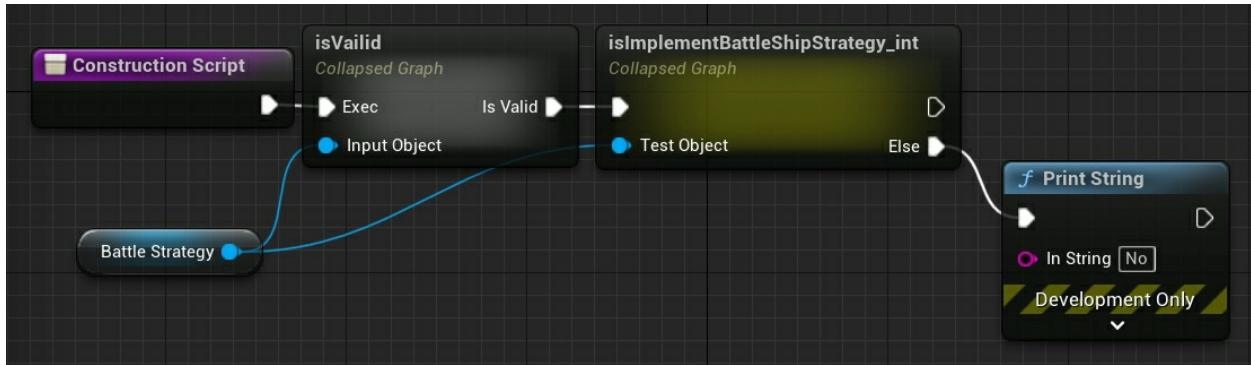
```
    if (!BattleStrategy) { UE_LOG(LogTemp, Error, TEXT("Engage():  
BattleStrategy is NULL, make sure it's initialized.")); return; }
```

```
    //Execute the current Strategy Maneuver
```

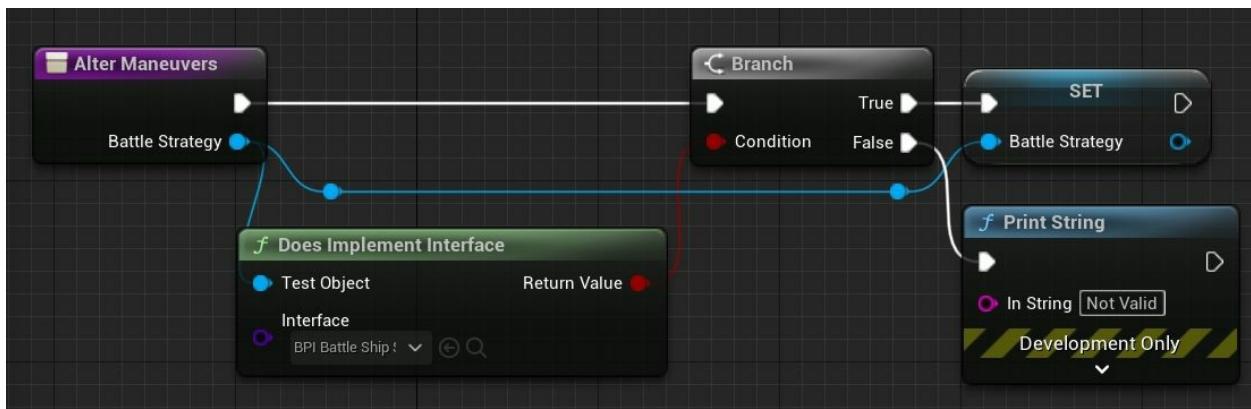
```
    BattleStrategy->Maneuver();
```

```
}
```

The *BattleShip* construction script:

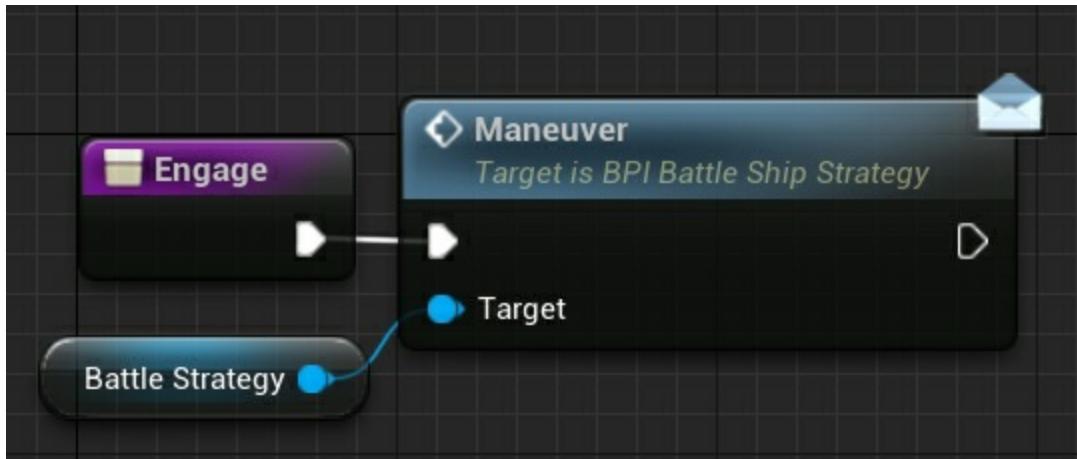


The *AlterManeuvers* blueprint function:



Is identical to the *BattleShip* construction with the exception of actually setting the *BattleStrategy* variable manually inside the function because it is not exposed on spawn.

The *engage* blueprint function:



The *BattleShipStrategy* interface:

## ***BattleShipStrategy.h***

```
#pragma once

#include "CoreMinimal.h"
#include "UObject/Interface.h"
#include "BattleShipStrategy.generated.h"

// This class does not need to be modified.
UINTERFACE(MinimalAPI)
class UBattleShipStrategy : public UInterface
{
    GENERATED_BODY()
};

class DESIGN_PATTERNS_API IBattleShipStrategy
{
    GENERATED_BODY()

    // Add interface functions to this class. This is the class that will be
    // inherited to implement this interface.

public:

    //The pure virtual functions of the BattleShipStrategy
    virtual void Maneuver() = 0;
};
```

## ***BattleShipStrategy.cpp***

```
#include "BattleShipStrategy.h"

// Add default functionality here for any IBattleShipStrategy functions that
// are not pure virtual.

The BruteForceStrategy class:
```

## ***BruteForceStrategy.h***

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "BattleShipStrategy.h"
#include "BruteForceStrategy.generated.h"

UCLASS()
class DESIGN_PATTERNS_API ABruteForceStrategy : public AActor,
public IBattleShipStrategy
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    ABruteForceStrategy();

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

    //Execute the Maneuver of this Strategy
    virtual void Maneuver() override;
};
```

## ***BruteForceStrategy.cpp***

```
#include "BruteForceStrategy.h"

// Sets default values
ABruteForceStrategy::ABruteForceStrategy()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void ABruteForceStrategy::BeginPlay()
{
    Super::BeginPlay();

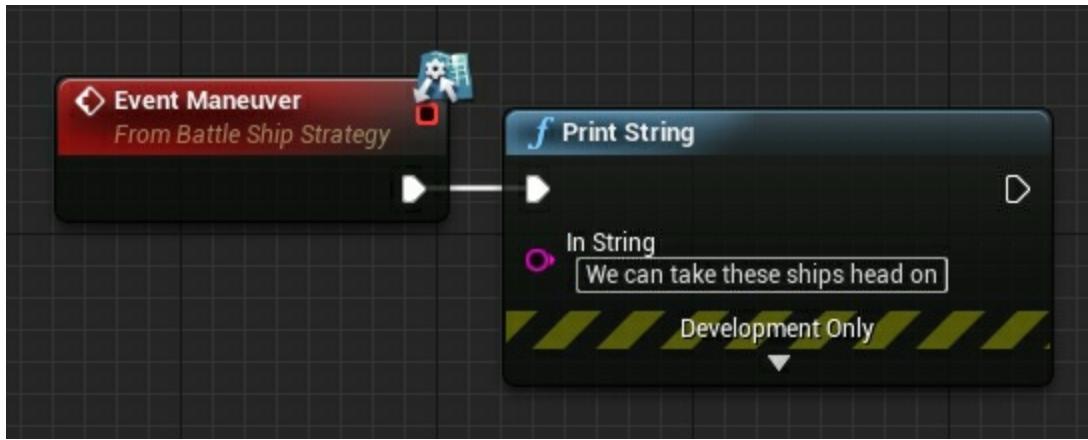
}

// Called every frame
void ABruteForceStrategy::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

}

void ABruteForceStrategy::Maneuver()
{
    //Execute the routine of this type of Strategy
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
    TEXT("We can take these ships head on"));
}
```

The *BruteForceStrategy* blueprint event graph:



The *DivideConquerStrategy* class:

## ***DivideConquerStrategy.h***

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "BattleShipStrategy.h"
#include "DivideConquerStrategy.generated.h"

UCLASS()
class DESIGN_PATTERNS_API ADivideConquerStrategy : public AActor,
public IBattleShipStrategy
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    ADivideConquerStrategy();

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

    //Execute the Maneuver of this Strategy
    virtual void Maneuver() override;

};
```

## ***DivideConquerStrategy.cpp***

```
#include "DivideConquerStrategy.h"

// Sets default values
ADivideConquerStrategy::ADivideConquerStrategy()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void ADivideConquerStrategy::BeginPlay()
{
    Super::BeginPlay();

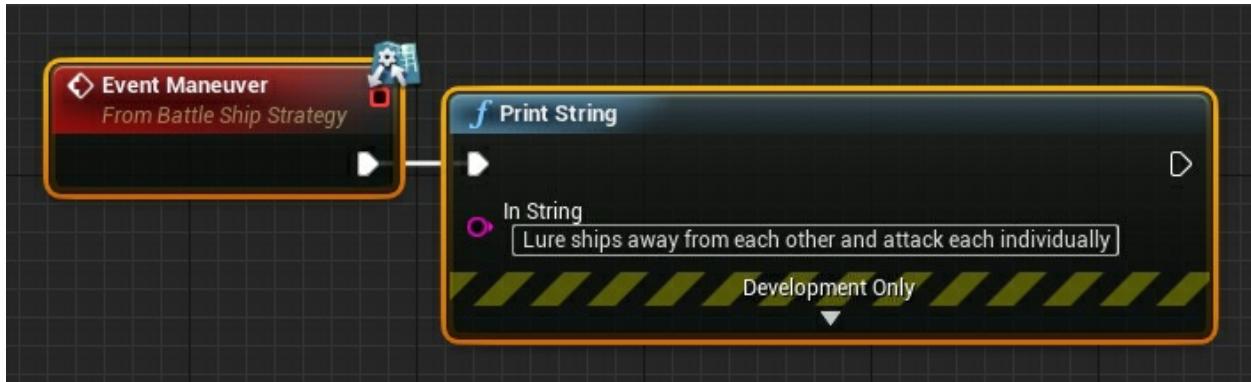
}

// Called every frame
void ADivideConquerStrategy::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

}

void ADivideConquerStrategy::Maneuver()
{
    //Execute the routine of this type of Strategy
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
    TEXT("Lure ships away from each other and attack each individually"));
}
```

The *DivideConquerStrategy* blueprint event graph:



The *RetreatStrategy* class:

## ***RetreatStrategy.h***

```
#pragma once

#include "CoreMinimal.h"
#include "GameFramework/Actor.h"
#include "BattleShipStrategy.h"
#include "RetreatStrategy.generated.h"

UCLASS()
class DESIGN_PATTERNS_API ARetreatStrategy : public AActor, public
IBattleShipStrategy
{
    GENERATED_BODY()

public:
    // Sets default values for this actor's properties
    ARetreatStrategy();

protected:
    // Called when the game starts or when spawned
    virtual void BeginPlay() override;

public:
    // Called every frame
    virtual void Tick(float DeltaTime) override;

    //Execute the Maneuver of this Strategy
    virtual void Maneuver() override;

};
```

## ***RetreatStrategy.cpp***

```
#include "RetreatStrategy.h"

// Sets default values
ARetreatStrategy::ARetreatStrategy()
{
    // Set this actor to call Tick() every frame. You can turn this off to
    // improve performance if you don't need it.
    PrimaryActorTick.bCanEverTick = true;

}

// Called when the game starts or when spawned
void ARetreatStrategy::BeginPlay()
{
    Super::BeginPlay();

}

// Called every frame
void ARetreatStrategy::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);

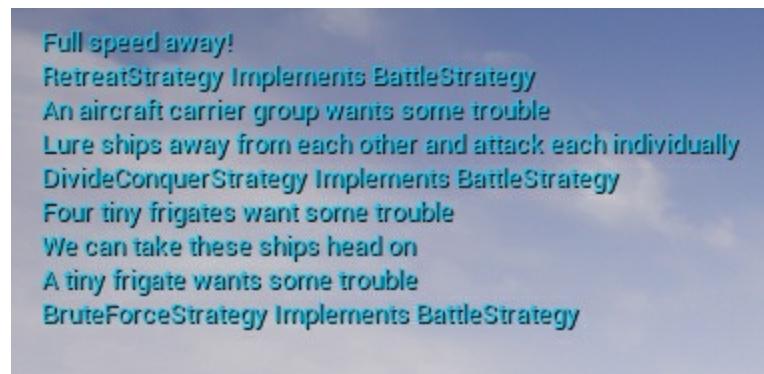
}

void ARetreatStrategy::Maneuver()
{
    //Execute the routine of this type of Strategy
    GEngine->AddOnScreenDebugMessage(-1, 15.f, FColor::Yellow,
    TEXT("Full speed away!"));
}
```

The *RetreatStrategy* blueprint event graph:



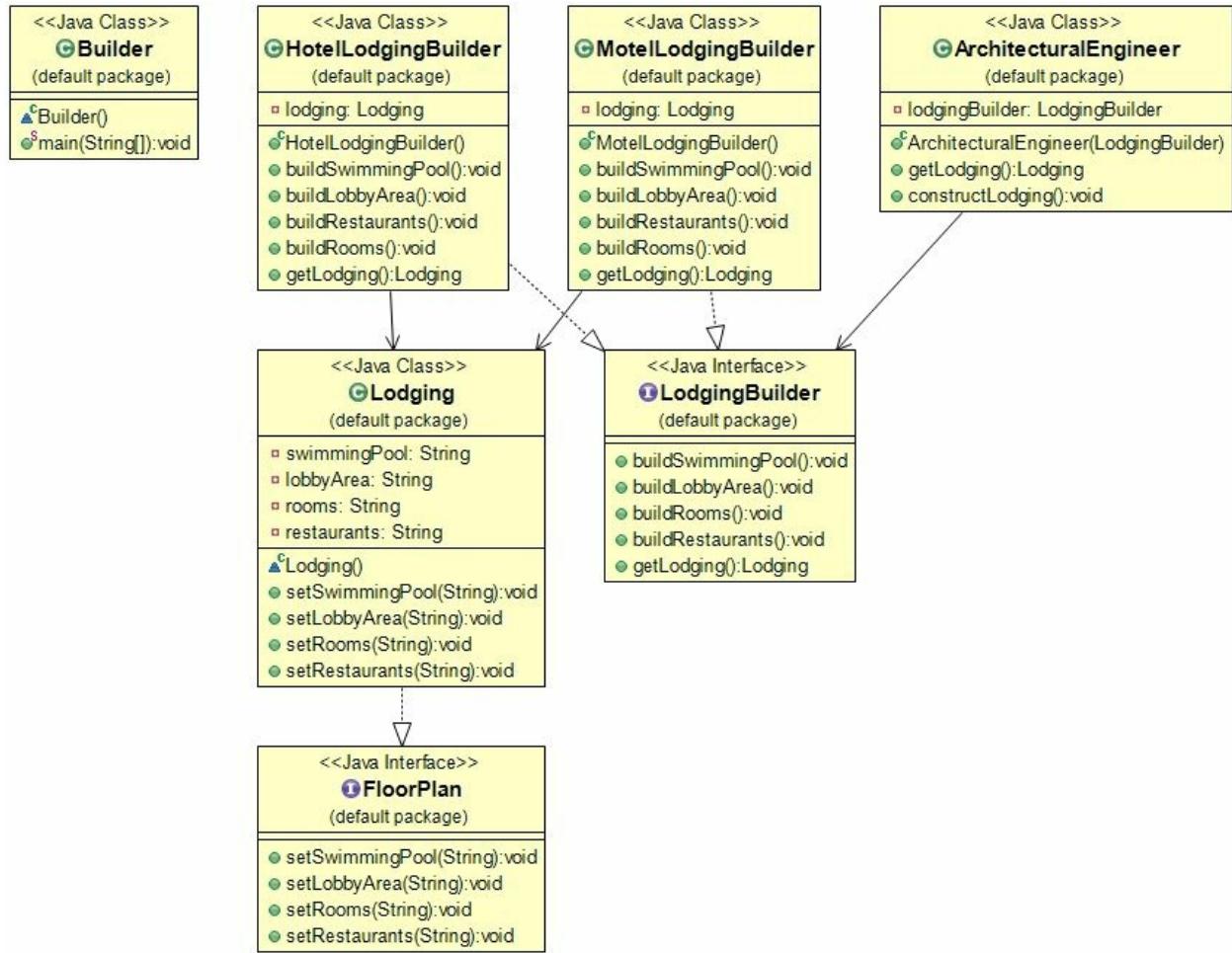
The Strategy pattern viewport print:



# Creational Patterns (Java)

# Hotel Motel Holiday Inn... Builder Pattern (Java)

**Builder Pattern UML Diagram**



## **Builder Pattern Implementation**

The *Builder* class contains the main function in Java. In terms of Blueprints, we can think of this class as the one that actually needs to be physically dragged into the game world. The other “concrete” classes do not need to be physically dragged into the world.

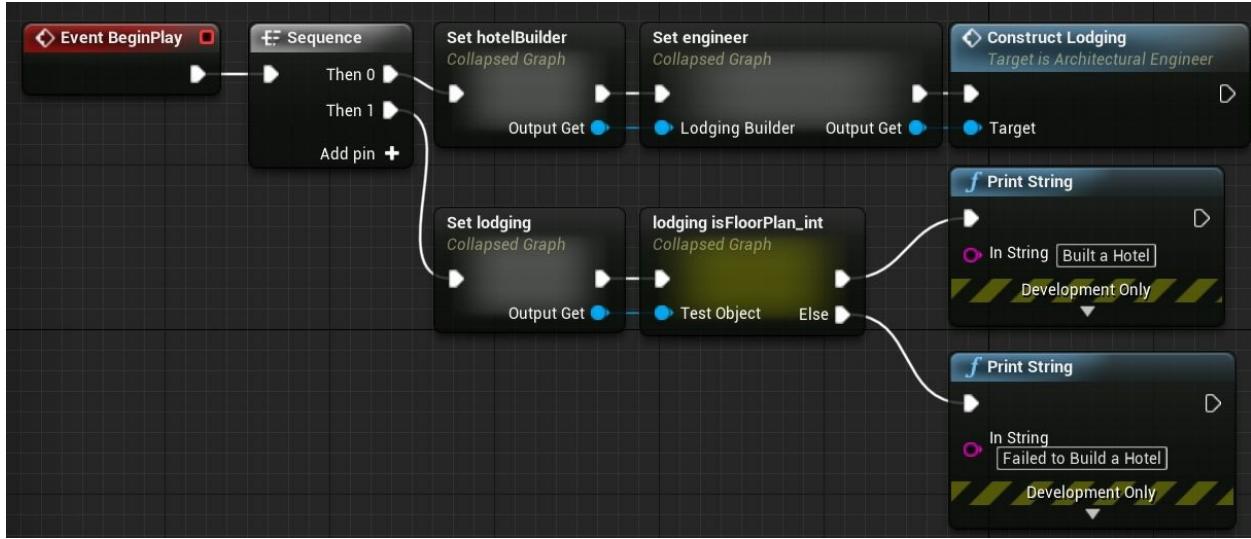
```
class Builder_Main
{
    public static void main(String[] args)
    {
        LodgingBuilder hotelBuilder = new HotelLodgingBuilder();
        ArchitecturalEngineer engineer = new
        ArchitecturalEngineer(hotelBuilder);

        engineer.constructLodging();

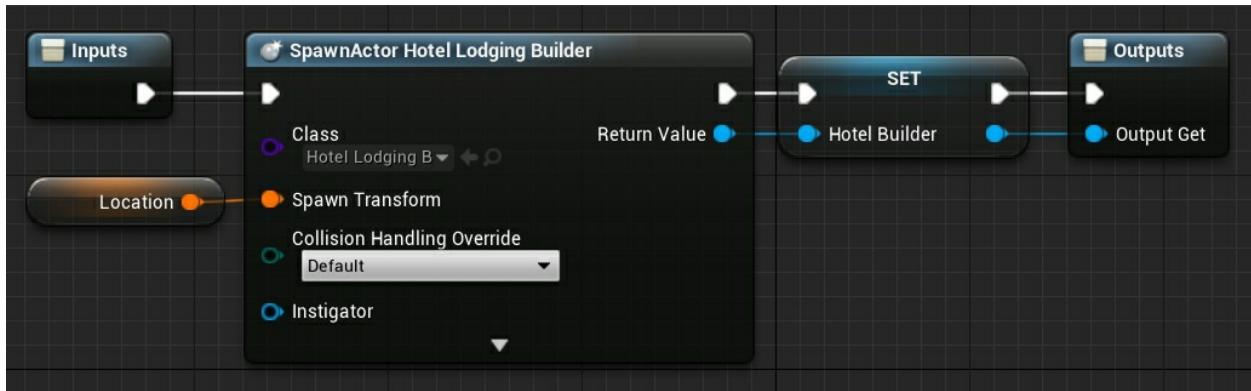
        Lodging lodging = engineer.getLodging();

        lodging.lodgingCharacteristics();
    }
}
```

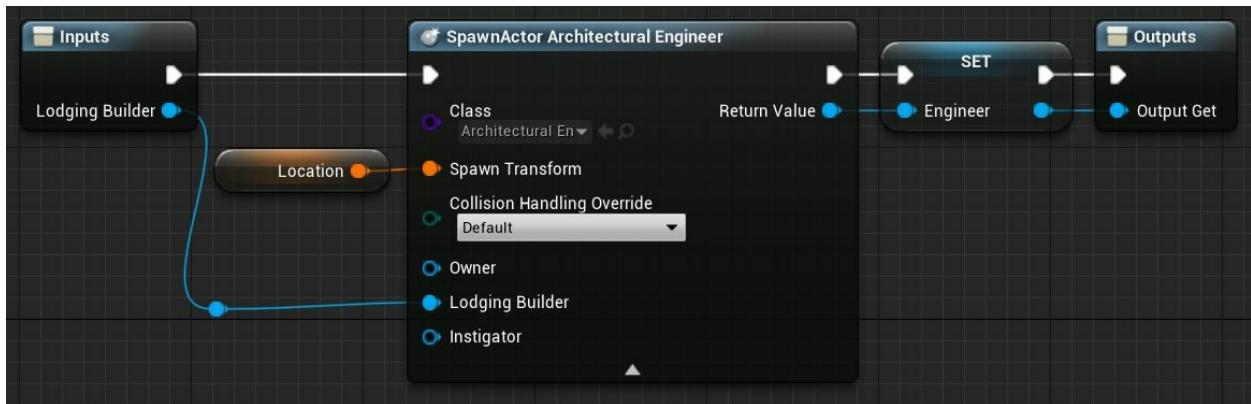
The blueprint equivalent of the *Builder\_Main* class:



The *Set hotelBuilder* and *Set engineer* collapsed graph are respectively:



and



The *Set hotelBuilder* collapsed graph represents this line of code:  
LodgingBuilder hotelBuilder = **new** HotelLodgingBuilder();

The *Spawn Actor From Class* node is akin to the *new* keyword in Java and other languages. The class to spawn is chosen using the dropdown box under the word “Class”. The *Spawn Transform* is to one’s own specific scenario. For simplicity, we can just right click on the *Spawn Transform* and choose “Promote to Variable”. If there is no *Spawn Transform* then the blueprint will compile with errors. We set (or “initialize”) our *hotelBuilder* variable once the desired actor is spawned.

The same basic idea applies to the *Set engineer* collapsed graph. However, *Architectural Engineer* needs a *LodgingBuilder* to be passed into its constructor as expressed in this line of code:

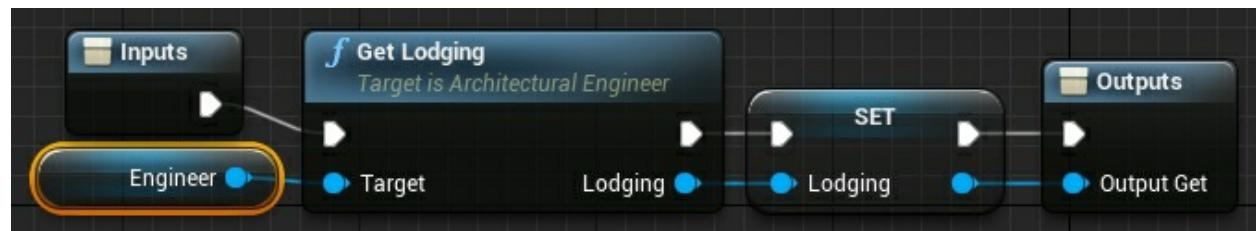
ArchitecturalEngineer engineer = **new** ArchitecturalEngineer(hotelBuilder);

Later, we will not be going into such detail on every spawning activity. Trust that there are many, many spawn actor requirements, and discussing each one will become an exercise in tedium.

*The Builder\_Main* blueprint continues with the *Construct Lodging* function call node. This function is called on the *engineer* we spawned previously. The equivalent line of code:

engineer.constructLodging();

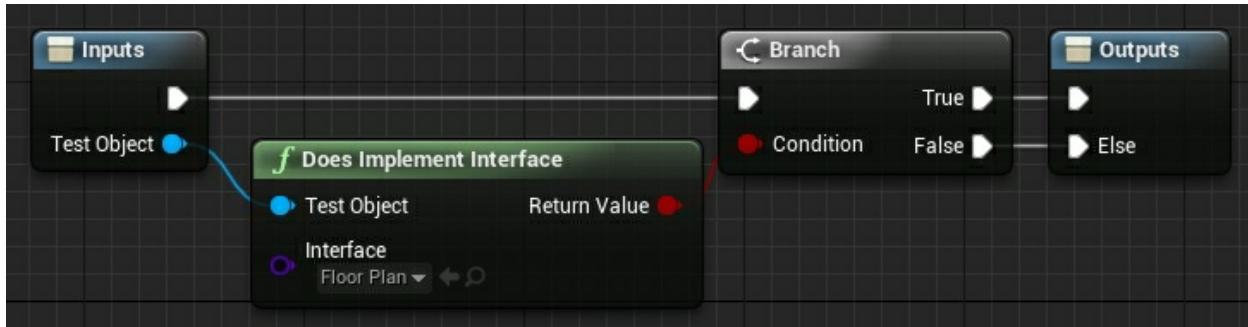
Note the *Sequence* node in *Builder\_Main*. Sequence 0 finishes once the *Construct Lodging* function call executes. Sequence 1 then starts executing. The *Set lodging* collapsed graph:



Now, we are equivalently executing this line of code:

```
Lodging lodging = engineer.getLodging();
```

This is the last line of code in the *Builder* class. However, we need to make some validation checks and make sure we built a hotel. We want to make sure the *lodging* actor implements the *FloorPlan* interface. The *lodging isFloorPlan\_int* collapsed graph:



Going forward we will not be showing this interface check collapsed graph. Note, the collapsed graph for this check always possesses a yellow hue. The only item that changes inside the collapsed node is the target interface we are checking the object against. The target interface is always part of the collapsed graph name.

The *Builder\_Main* blueprint ends execution with a print string to let us know if the hotel is actually built. “Failed to Build a Hotel” prints If an error in logic occurs somewhere in any of the blueprints.

Now we can discuss the interfaces we use in this pattern. First, we create a floor plan interface for all lodgings to implement. Most commercial lodgings in the USA contain a pool, lobby area, rooms, and restaurants. We also create a lodging builder interface for our specific lodging builder classes to implement. The *FloorPlan* interface:

```
interface FloorPlan{  
  
    public void setSwimmingPool(String swimmingPool);  
    public void setLobbyArea(String lobbyArea);  
    public void setRooms(String rooms);  
    public void setRestaurants(String restaurants);  
  
}
```

The equivalent blueprint interface is very mundane. The following are the nodes representing the functions in the *FloorPlan* interface.



The saying goes, “Program to an Interface, not an implementation” (Gamma et.al, 1995). In fact, it is a core principle in object-oriented design. The main benefit of programming to an interface is that it reduces implementation dependencies between subsystems (Gamma et.al, 1995). We see this first hand in our Builder Pattern example. Specifically, The Builder pattern seeks to decouple the construction of complex objects from its actual representation (Gamma et.al, 1995).

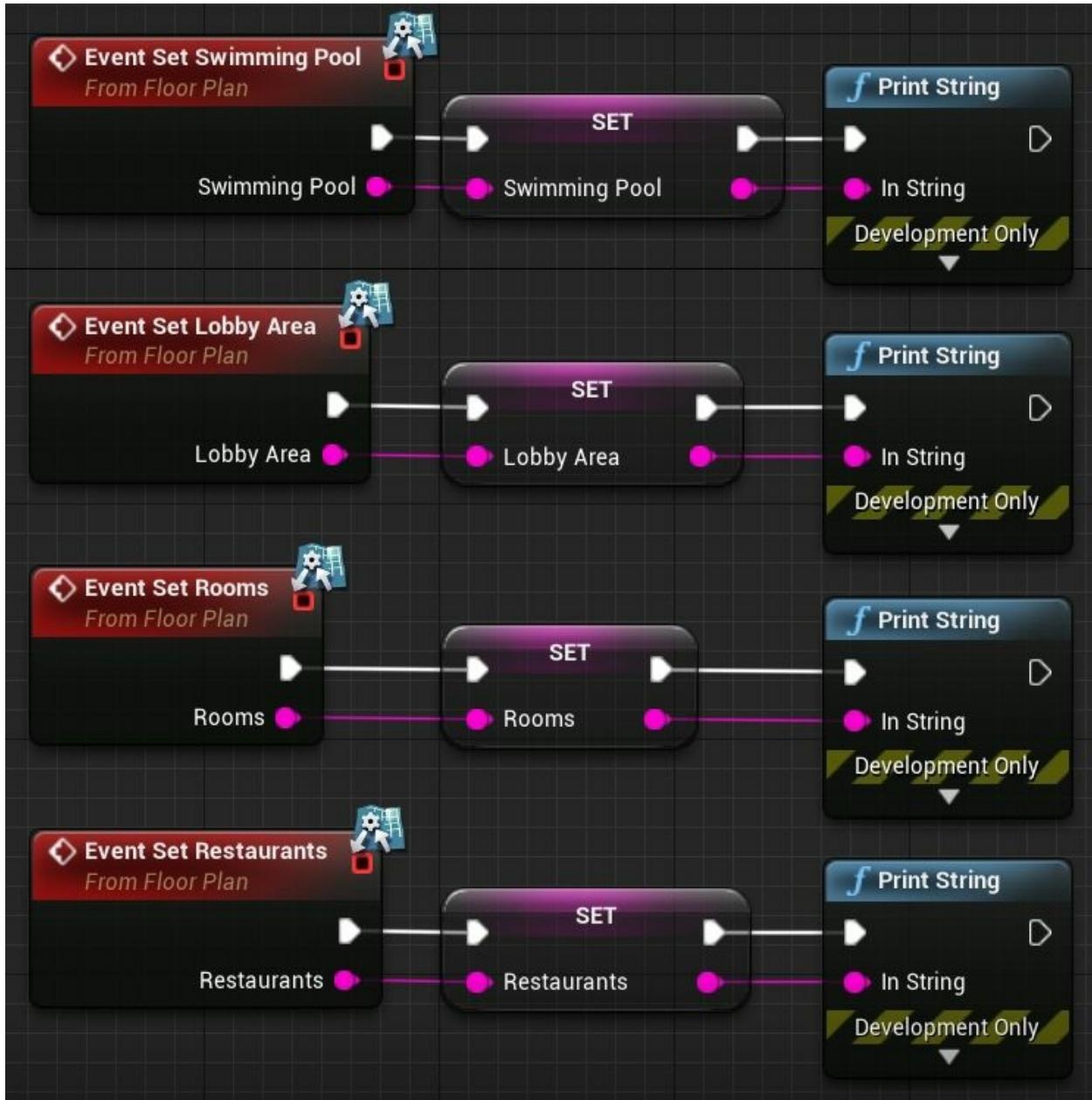
```
interface LodgingBuilder
{
    public void buildSwimmingPool();
    public void buildLobbyArea();
    public void buildRooms();
    public void buildRestaurants();
    public Lodging getLodging();
}
```

The screenshot of the *LodgingBuilder* interface blueprint follows the same design as the *FloorPlan* interface blueprint and is thus removed for the sake of brevity. Going forward, showing interface blueprint images are minimized unless considered necessary to deepen understanding for the reader.

Next, we create the *Lodging* class which implements the *FloorPlan* interface.

```
class Lodging implements FloorPlan{  
  
    private String swimmingPool;  
    private String lobbyArea;  
    private String rooms;  
    private String restaurants;  
  
    @Override  
    public void setSwimmingPool(String swimmingPool) {  
        this.swimmingPool = swimmingPool;  
    }  
  
    @Override  
    public void setLobbyArea(String lobbyArea) {  
        this.lobbyArea = lobbyArea;  
    }  
  
    @Override  
    public void setRooms(String rooms) {  
        this.rooms = rooms;  
    }  
  
    @Override  
    public void setRestaurants(String restaurants) {  
        this.restaurants = restaurants;  
    }  
  
    public void lodgingCharacteristics(){  
  
        System.out.println(swimmingPool);  
        System.out.println(lobbyArea);  
        System.out.println(rooms);  
        System.out.println(restaurants);  
  
    }  
}
```

The *Lodging* blueprint:



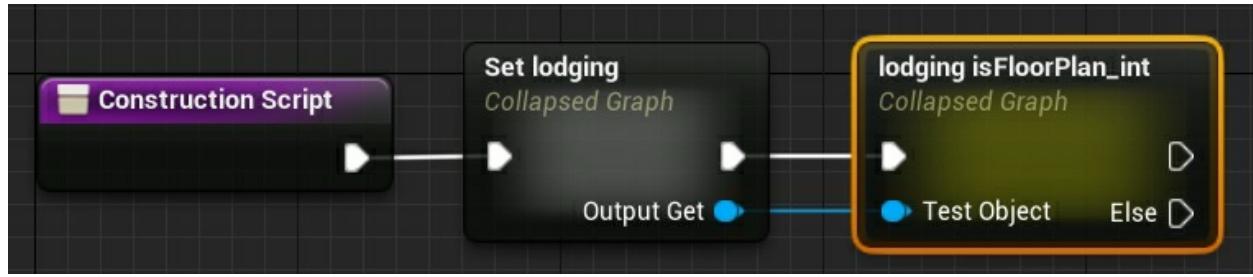
Blueprints use *Event* nodes when calling void interface functions. The blueprint version of the *Lodging* class prints the string value passed to the function in addition to setting local variables. Also, we must make sure this blueprint class implements the *FloorPlan* interface by adding it in the class settings Interfaces sections.

The next two classes, *HotelLodgingBuilder* and *MotelLodgingBuilder*,

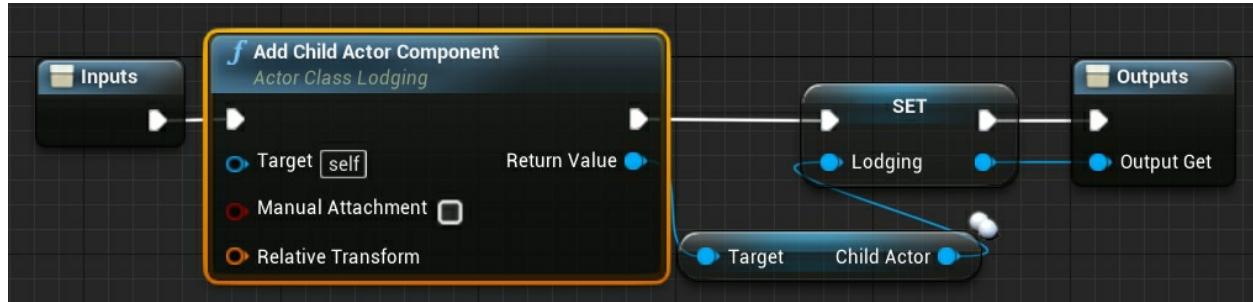
implement the lodging builder interface and provide the custom specs for each type of lodging.

```
class HotelLodgingBuilder implements LodgingBuilder {  
  
    private Lodging lodging;  
  
    public HotelLodgingBuilder(){  
        this.lodging = new Lodging();  
    }  
  
    @Override  
    public void buildLobbyArea() {  
        lodging.setLobbyArea("Grand Hall");  
    }  
  
    @Override  
    public void buildRestaurants() {  
        lodging.setRestaurants("5 star Buffet");  
    }  
  
    @Override  
    public void buildRooms() {  
        lodging.setRooms("Luxury Suites");  
    }  
  
    @Override  
    public void buildSwimmingPool() {  
        lodging.setSwimmingPool("Indoor Lagoon");  
    }  
  
    @Override  
    public Lodging getLodging() {  
        return this.lodging;  
    }  
}
```

The *HotelLodgingBuilder* blueprint construction script event graph:



The *Set lodging* collapsed graph node:



The *Add Child Actor Component* is a work around for not being able to use the *Spawn Actor of Class* node. We can see the *HotelLodgingBuilder* constructor instantiates a new *Lodging* object in our code:

```
public HotelLodgingBuilder(){  
    this.lodging = new Lodging();  
}
```

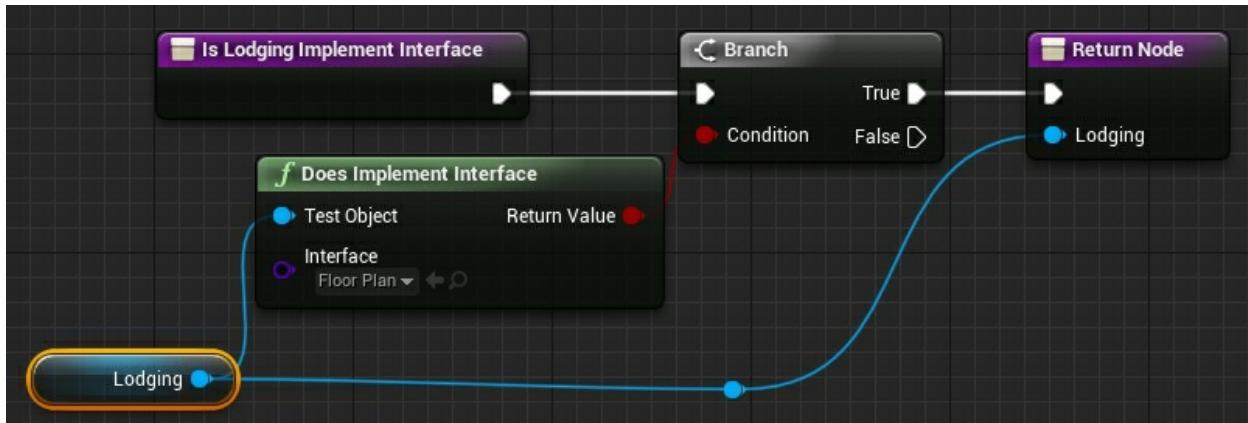
We will see an even more dynamic way to initialize objects around construction script constraints when we get to the State Pattern.

*HotelLodgingBuilder* event graph:



The *isLodgingImplementInterface* function is a custom interface check function similar the yellow hued collapsed graphs.

The *isLodgingImplementInterface* function:



The *MotelLodgingBuilder* is very similar to the *HotelLodgingBuilder*. The difference is in the attributes for the pool, lobby area, rooms, and restaurant.

```
class MotelLodgingBuilder implements LodgingBuilder
{
    private Lodging lodging;

    public MotelLodgingBuilder()
    {
        this.lodging = new Lodging();
    }

    public void buildSwimmingPool()
    {
        lodging.setSwimmingPool("Green Slim");
    }

    public void buildLobbyArea()
    {
        lodging.setLobbyArea("Economical");
    }

    public void buildRestaurants()
    {
        lodging.setRestaurants("Vending Machines");
    }

    public void buildRooms()
    {
        lodging.setRooms("Bates Style");
    }

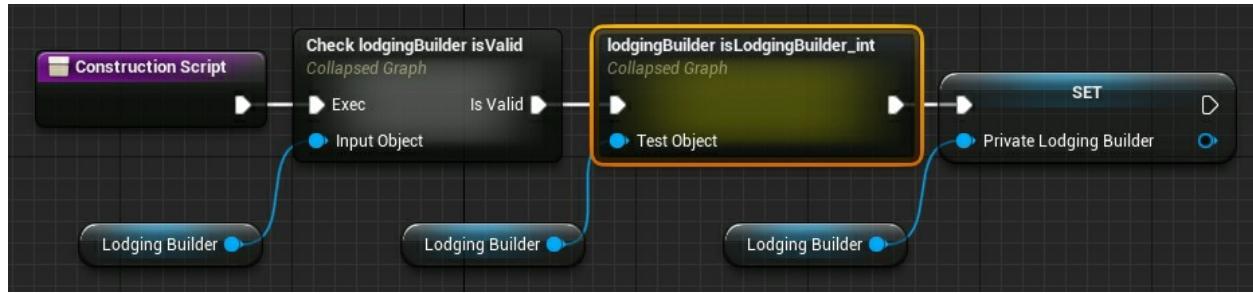
    public Lodging getLodging()
    {
        return this.lodging;
    }
}
```

The blueprint screenshot for *MotelLodgingBuilder* is omitted for brevity. It basically mimics that of the *HotelLodgingBuilder*.

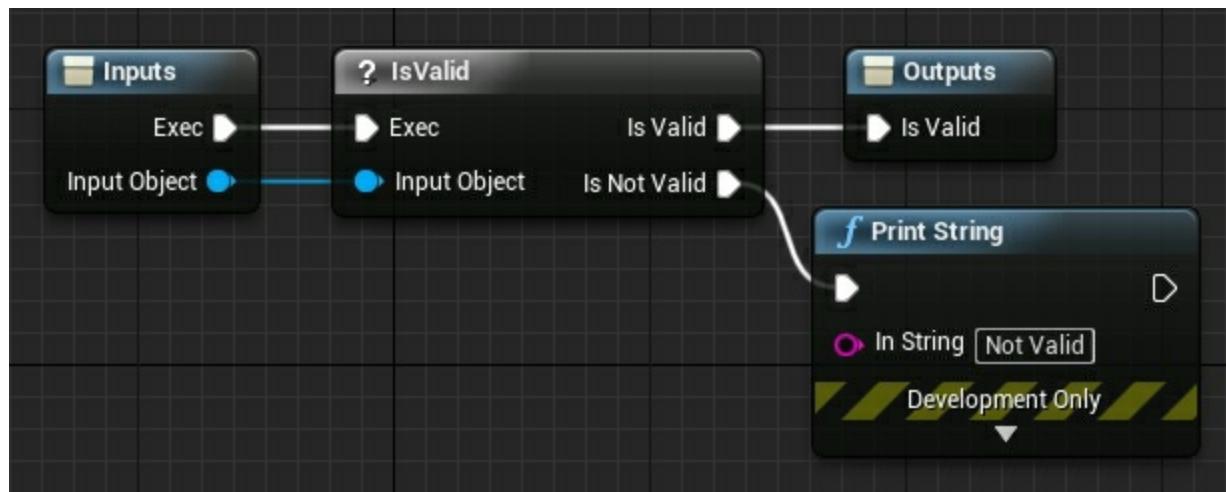
Now we need an architectural engineer to coordinate the step by step construction of the lodging.

```
class ArchitecturalEngineer{  
  
    private LodgingBuilder lodgingBuilder;  
  
    public ArchitecturalEngineer(LodgingBuilder lodgingBuilder){  
        this.lodgingBuilder = lodgingBuilder;  
    }  
  
    public Lodging getLodging(){  
        return this.lodgingBuilder.getLodging();  
    }  
  
    public void constructLodging(){  
  
        this.lodgingBuilder.buildSwimmingPool();  
        this.lodgingBuilder.buildLobbyArea();  
        this.lodgingBuilder.buildRooms();  
        this.lodgingBuilder.buildRestaurants();  
    }  
}
```

The *ArchitecturalEngineer* construction script:



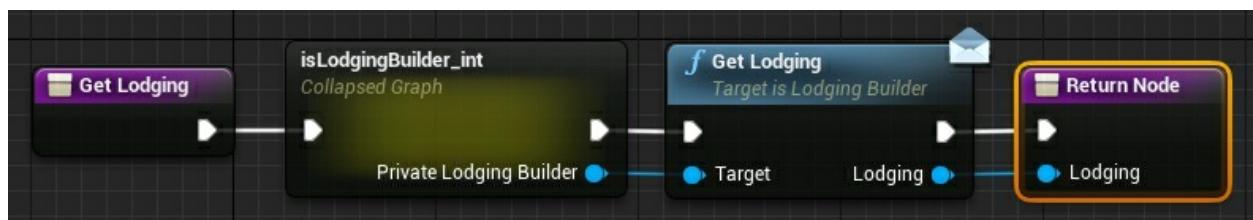
The *Check lodgingBuilder isValid* node:



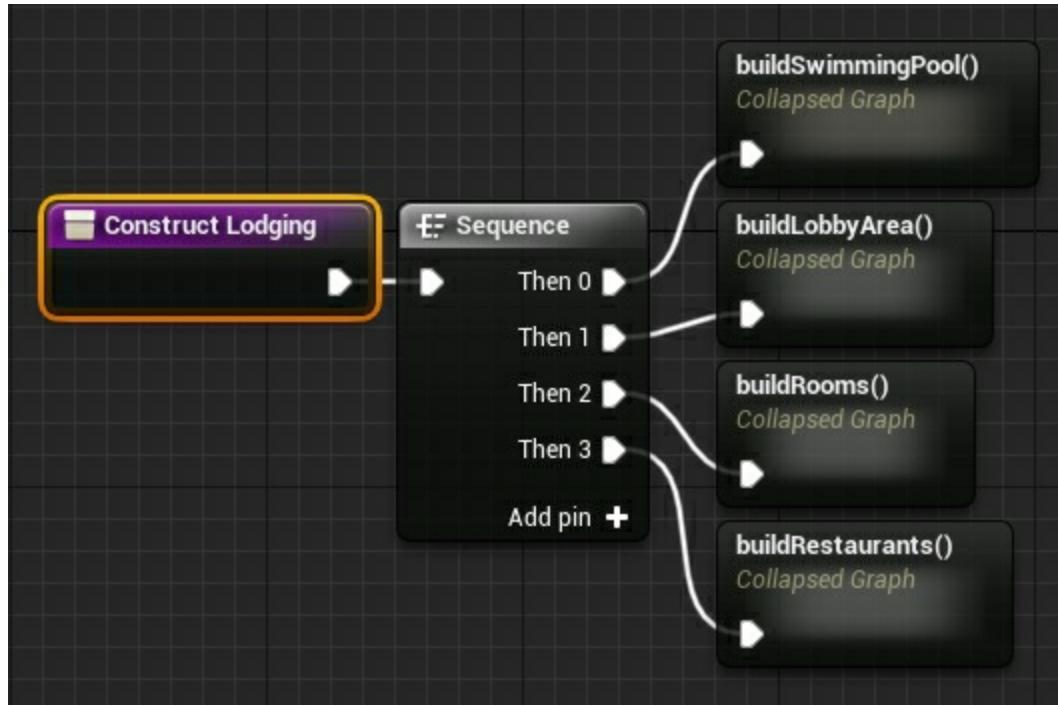
This validity check is executed as a safety mechanism. We want to make sure the object is not null before we start to perform operations on said object. A “Not Valid” string prints to the viewport If the object is indeed null.

The *ArchitecturalEngineer* contains a few important functions, namely *getLodging* and *constructLodging*.

The *getLodging* function graph:



The *constructLodging* function graph:



The collapsed graphs represent the body of the *constructLodging* function:

```
public void constructLodging(){  
    this.lodgingBuilder.buildSwimmingPool();  
    this.lodgingBuilder.buildLobbyArea();  
    this.lodgingBuilder.buildRooms();  
    this.lodgingBuilder.buildRestaurants();  
}
```

The *buildSwimmingPool*, *buildLobbyArea*, *buildRooms*, and *buildRestaurants* graphs are respectively:



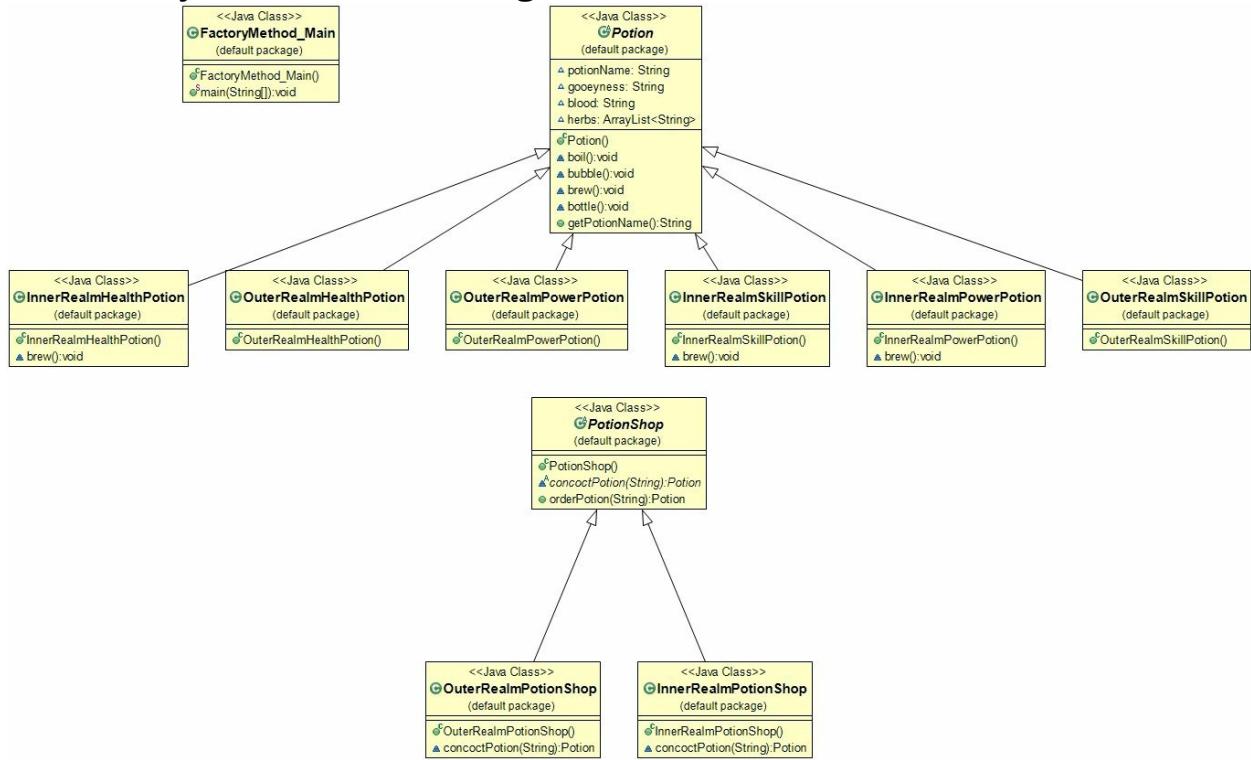
This is the final screenshot of the construction of a hotel.



Strings printed to the viewport are not impressive. However, these strings represent “what could be.” In development, one has the 3D or 2D artwork associated with these strings. Thanks to the Builder Pattern, we have a framework to build different versions of hotels piece by piece.

# Boil and Bubble, Brew and Bottle... Factory Method (Java)

Factory Method UML Diagram



## Factory Method Implementation

We have creator, concrete creator, and product participants in the Factory Method design pattern. Our example implementation has a *PotionShop* abstract class as a creator. The concrete creators are *InnerRealmPotionShop* and *OuterRealmPotionShop*. These concrete creators inherit functionality from the *PotionShop* abstract class. Our concrete products are:

*InnerRealmHealthPotion*  
*InnerRealmPowerPotion*  
*InnerRealmSkillPotion*  
*OuterRealmHealthPotion*  
*OuterRealmPowerPotion*  
*OuterRealmSkillPotion*

All these products inherit functionality from the *Potion* abstract class. Using blueprints, we are going to implement the *InnerRealmPotionShop* and its products, namely:

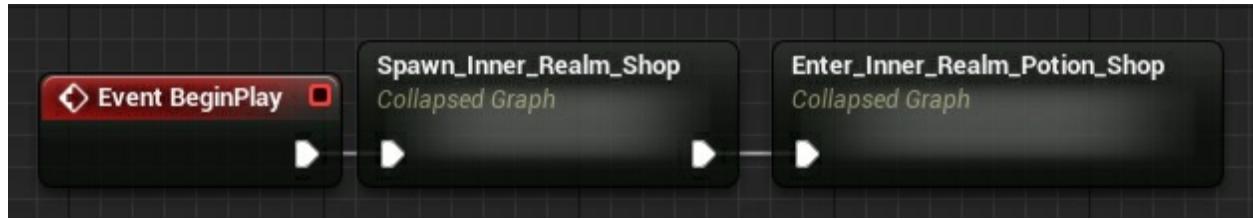
*InnerRealmHealthPotion*  
*InnerRealmPowerPotion*  
*InnerRealmSkillPotion*

The same ideas and principles apply to the creation of the *OuterRealmPotionShop* and its products.

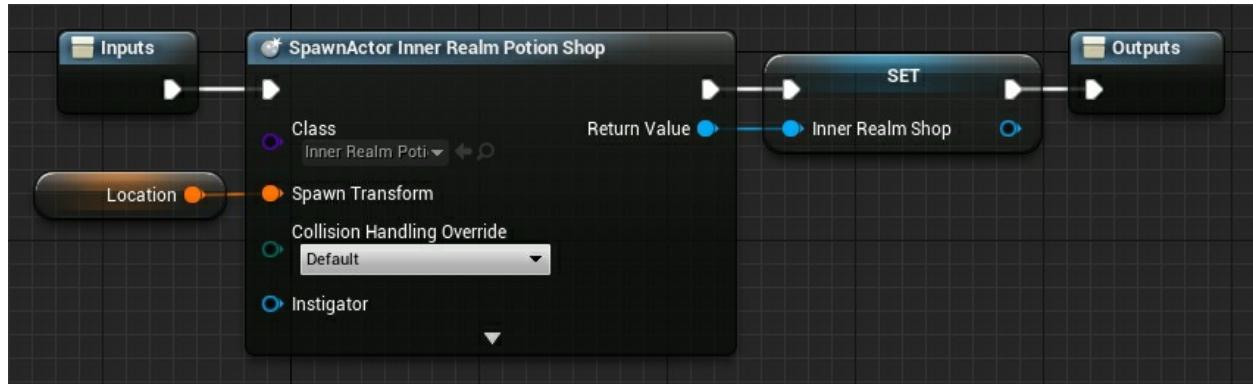
The *FactoryMethod* class contains the main function in Java. In terms of Blueprints, we can think of this class as the one that actually needs to be physically dragged into the game world. The other “concrete” classes do not need to be physically dragged into the world.

```
public class FactoryMethod_Main {  
  
    public static void main(String[] args) {  
        PotionShop outerRealmShop = new OuterRealmPotionShop();  
        PotionShop innerRealmShop = new InnerRealmPotionShop();  
  
        Potion potion = outerRealmShop.orderPotion("Health");  
        System.out.println("Potion is " + potion.getPotionName() + "\n");  
  
        potion = innerRealmShop.orderPotion("Health");  
        System.out.println("Potion is " + potion.getPotionName() + "\n");  
  
        potion = outerRealmShop.orderPotion("Power");  
        System.out.println("Potion is " + potion.getPotionName() + "\n");  
  
        potion = innerRealmShop.orderPotion("Power");  
        System.out.println("Potion is " + potion.getPotionName() + "\n");  
  
        potion = outerRealmShop.orderPotion("Skill");  
        System.out.println("Potion is " + potion.getPotionName() + "\n");  
        potion = innerRealmShop.orderPotion("Skill");  
        System.out.println("Potion is " + potion.getPotionName() + "\n");  
    }  
}
```

This is the top-level event graph for the *FactoryMethod\_Main* blueprint:



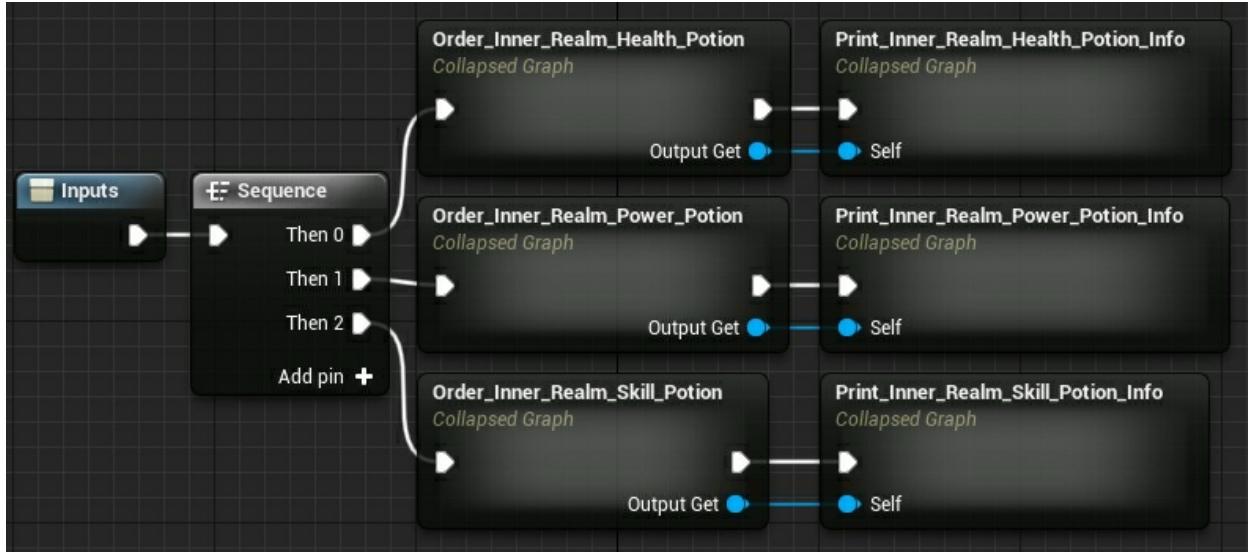
The *Spawn\_Inner\_Realm\_Shop* collapsed graph:



The *Spawn\_Inner\_Realm\_Shop* collapsed graph represents this line of code:

```
PotionShop innerRealmShop = new InnerRealmPotionShop();
```

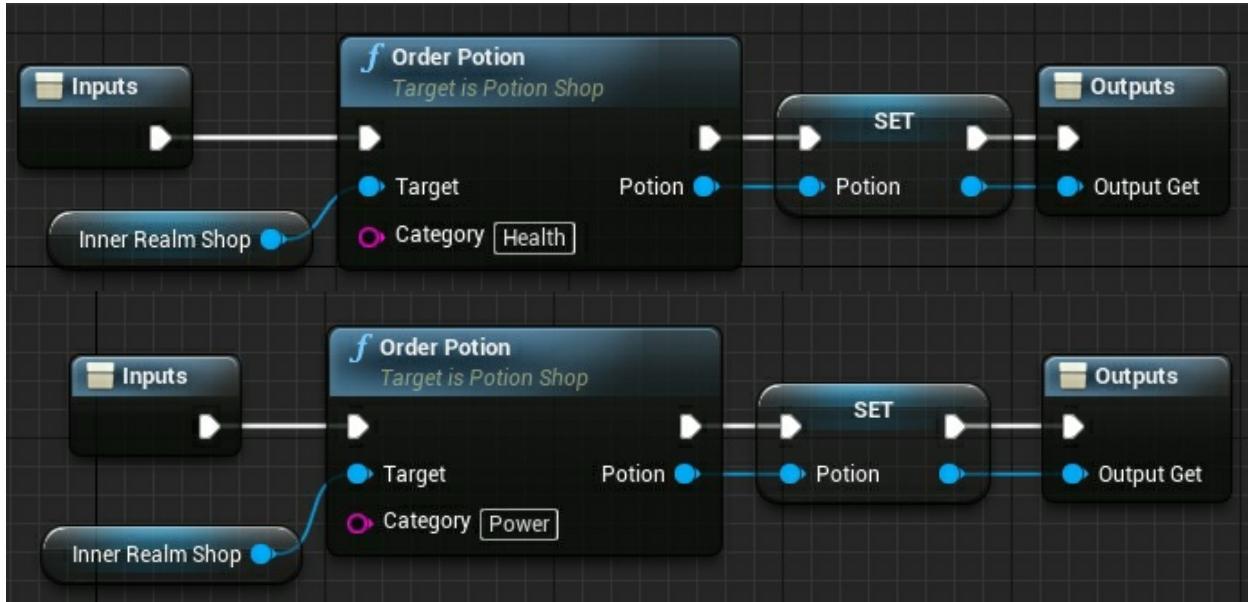
The bulk of the main class is collapsed inside of the *Enter\_Inner\_Region\_Potion\_Shop*:



Basically, we are ordering different potions from the Inner Realm Potion Shop and then printed information related to the potion we ordered.

The  
*Order\_Inner\_Realm\_Potion*,  
*Order\_Inner\_Realm\_Power\_Potion*,  
*Order\_Inner\_Realm\_Skill\_Potion*  
collapsed graphs respectively are:

*Order\_Inner\_Realm\_Health\_Potion*,  
*Order\_Inner\_Realm\_Power\_Potion*

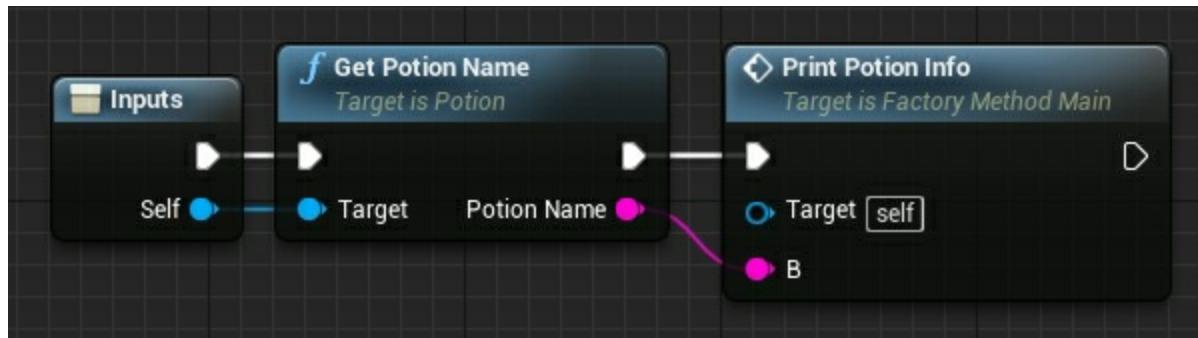




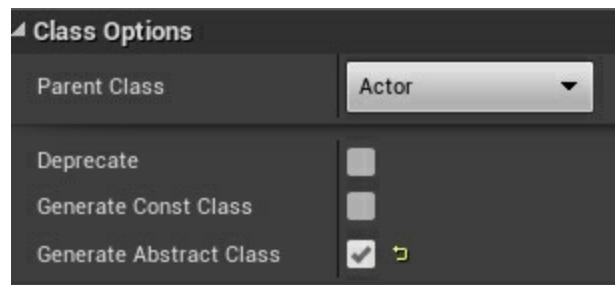
These three graphs are equivalent to:

```
potion = innerRealmShop.orderPotion("Health");
potion = innerRealmShop.orderPotion("Power");
potion = innerRealmShop.orderPotion("Skill");
```

The print potion info collapsed graphs all contain the same functionality with a different potion being passed to it:



This concludes the FactoryMethod\_Main blueprint. Next, we move on to the *Potion* and *PotionShop* classes. Both of these classes are abstract. We can create abstract classes in blueprints by navigating to class settings and checking “Generate Abstract Class.” As pictured here:



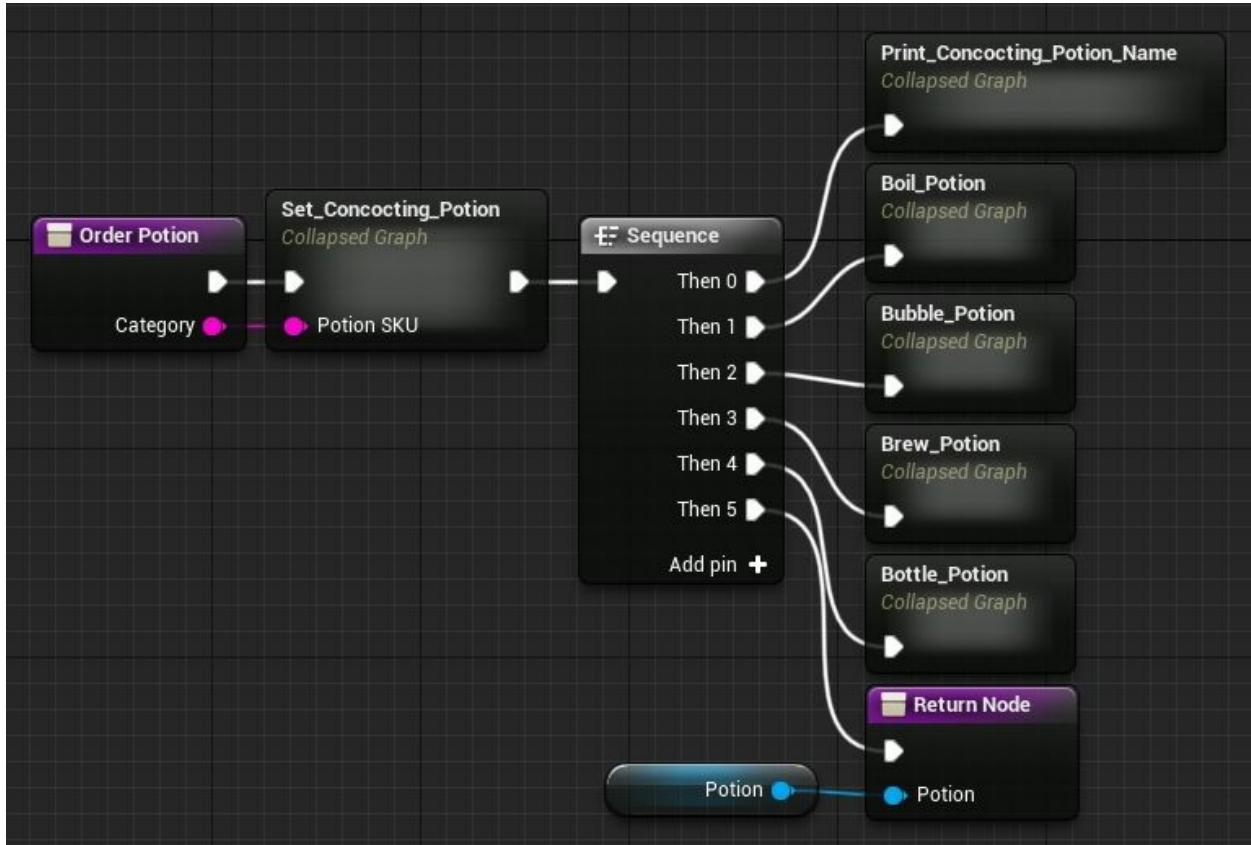
Our *PotionShop* class code implementation:

```
public abstract class PotionShop {  
  
    abstract Potion concoctPotion(String potionSKU);  
  
    public Potion orderPotion(String category) {  
        Potion potion = concoctPotion(category);  
        System.out.println("Concocting " + potion.getPotionName());  
        potion.boil();  
        potion.bubble();  
        potion.brew();  
        potion.bottle();  
        return potion;  
    }  
}
```

Here we must deviate a bit from traditional code. There does not seem to be a way to explicitly declare a function as “abstract” in blueprints. It is possible that the compiler may be smart enough to do this under the hood, but there is no public information stating this feature. Our *concoctPotion* equivalent blueprint:



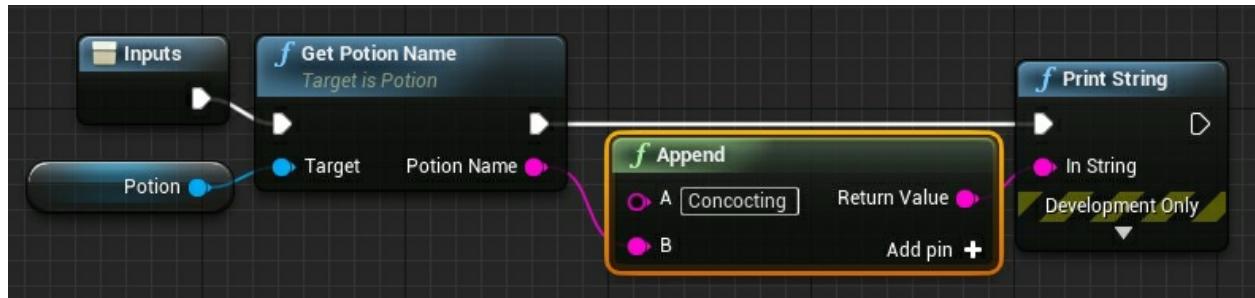
The *orderPotion* function blueprint:



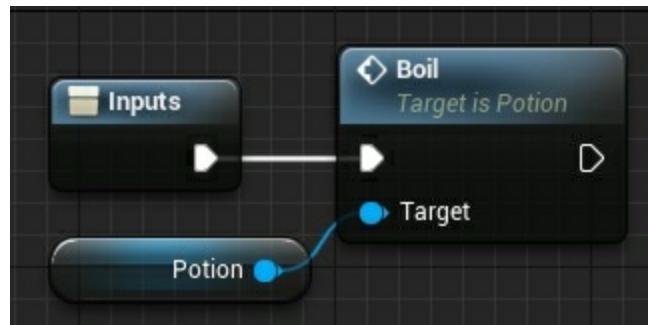
The *Set\_Concocting\_Potion* collapsed graph:



The `Print_Concocting_Potion_Name` collapsed graph:



The `Boil_Potion` collapsed graph:



The `Bubble_Potion` collapsed graph:



The *Brew\_Potion* collapsed graph:



The *Bottle\_Potion* collapsed graph:



The abstract *Potion* class code:

```
import java.util.ArrayList;

public abstract class Potion {
    String potionName;
    String gooeyness;
    String blood;
    ArrayList<String> herbs = new ArrayList<String>();

    void boil() {
        System.out.println("Boil " + potionName);
        System.out.println("Drop in blood...");
        System.out.println("Drop in herbs: ");
    }

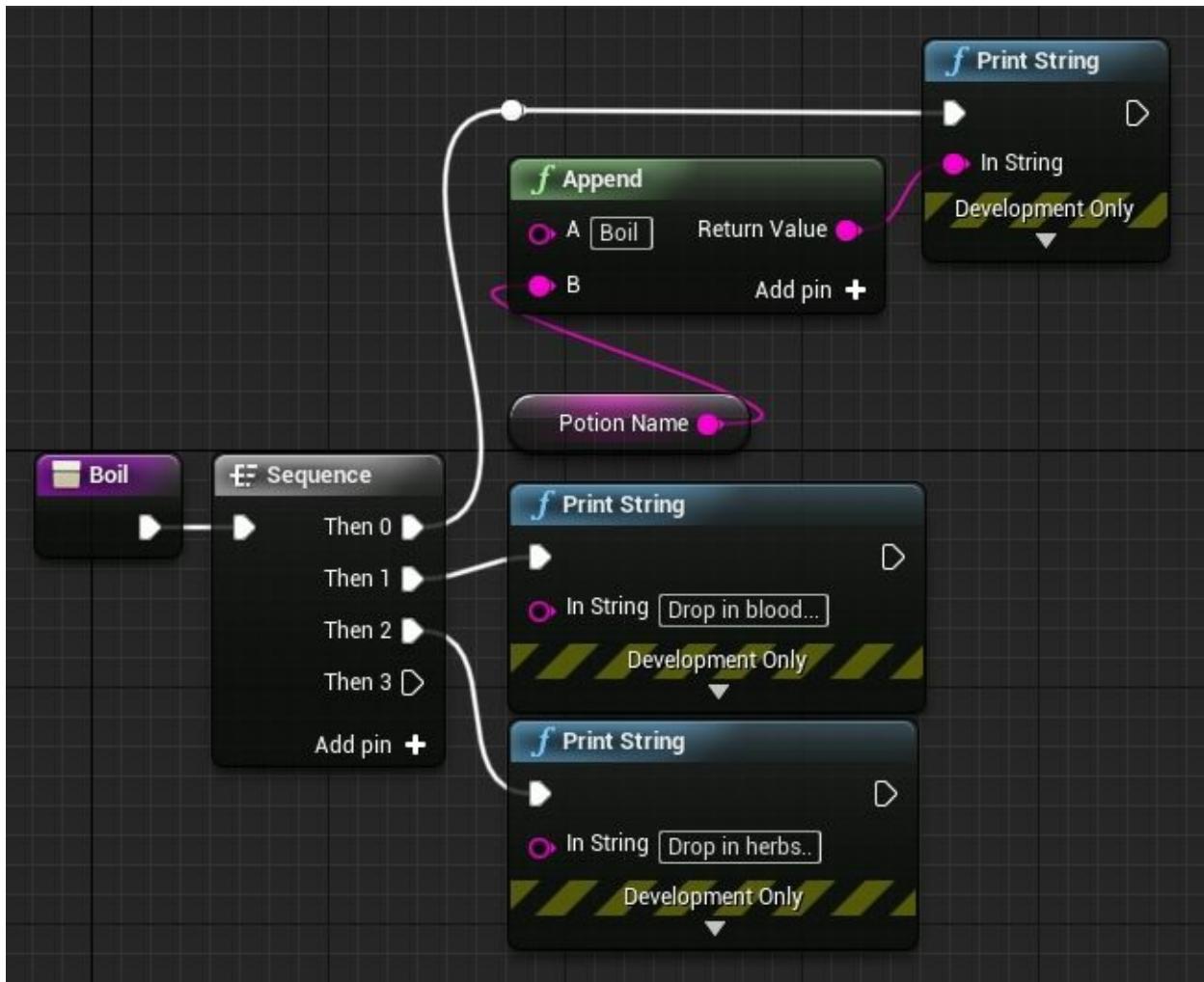
    void bubble() {
        System.out.println("Bubble for 3 moons and a sunset");
    }

    void brew() {
        System.out.println("Brewing at Low Temperature");
    }

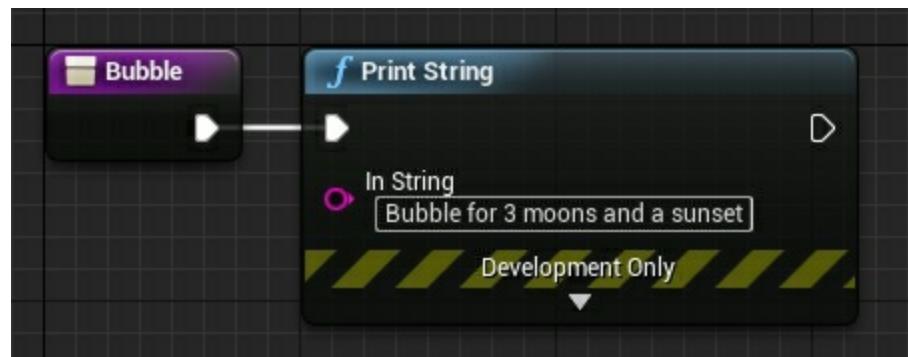
    void bottle() {
        System.out.println("Bottle concoction in flask");
    }

    public String getPotionName() {
        return potionName;
    }
}
```

The *boil* method equivalent blueprint function:



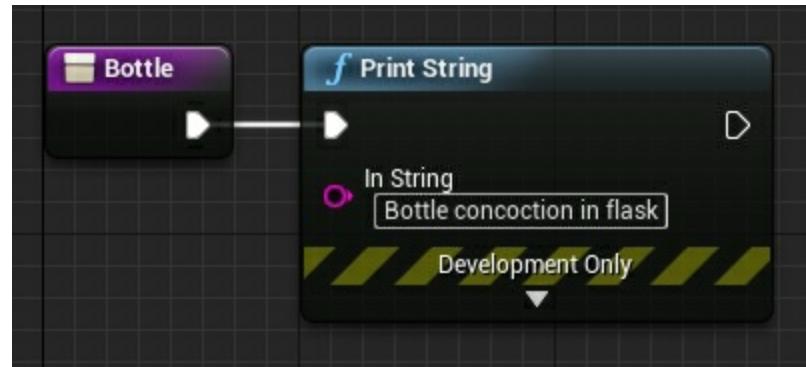
The *bubble* method equivalent blueprint function:



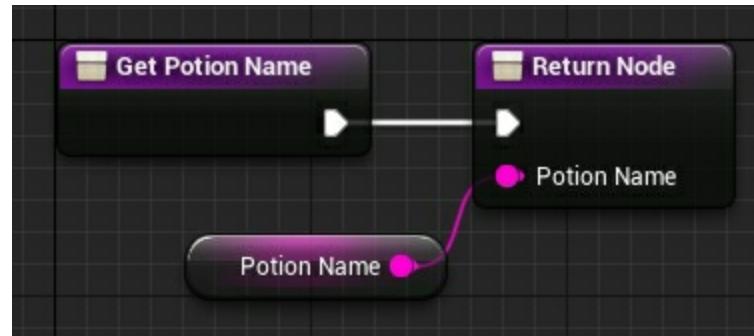
The *brew* method equivalent blueprint function:



The *bottle* method equivalent blueprint function:



The *getPotionName* method equivalent blueprint function:

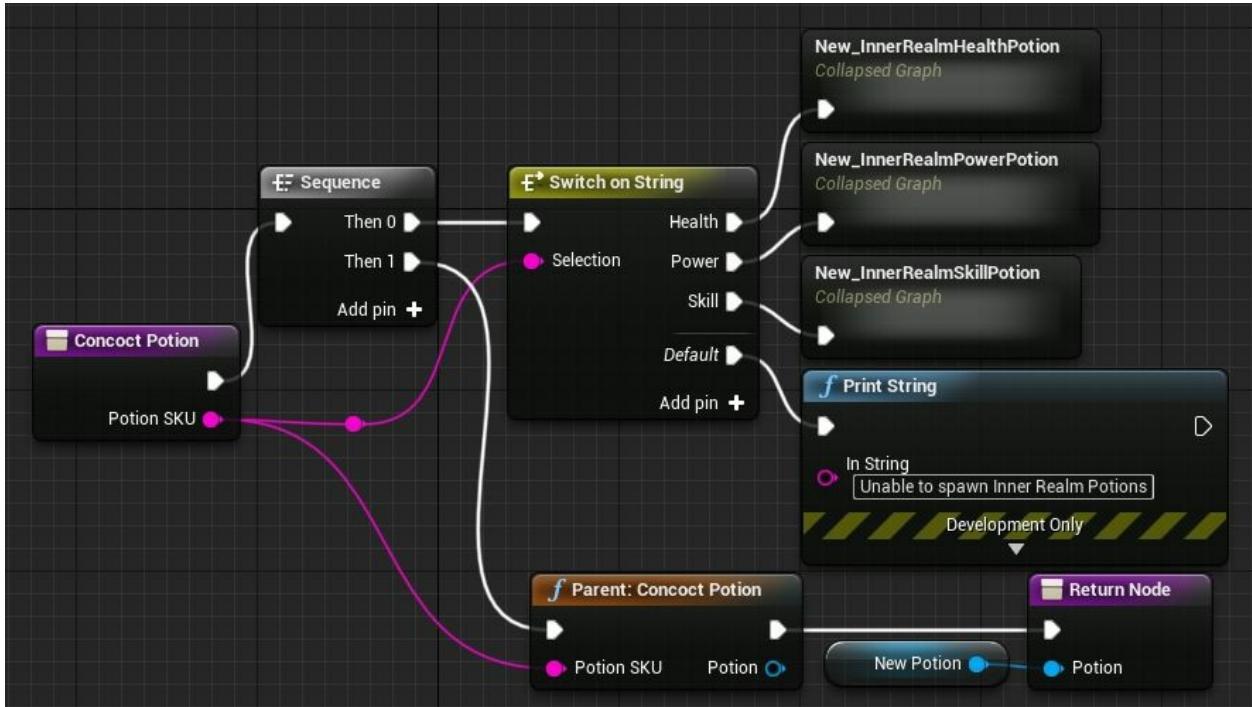


Now that we have our abstract parent classes defined let's focus on the concrete creator

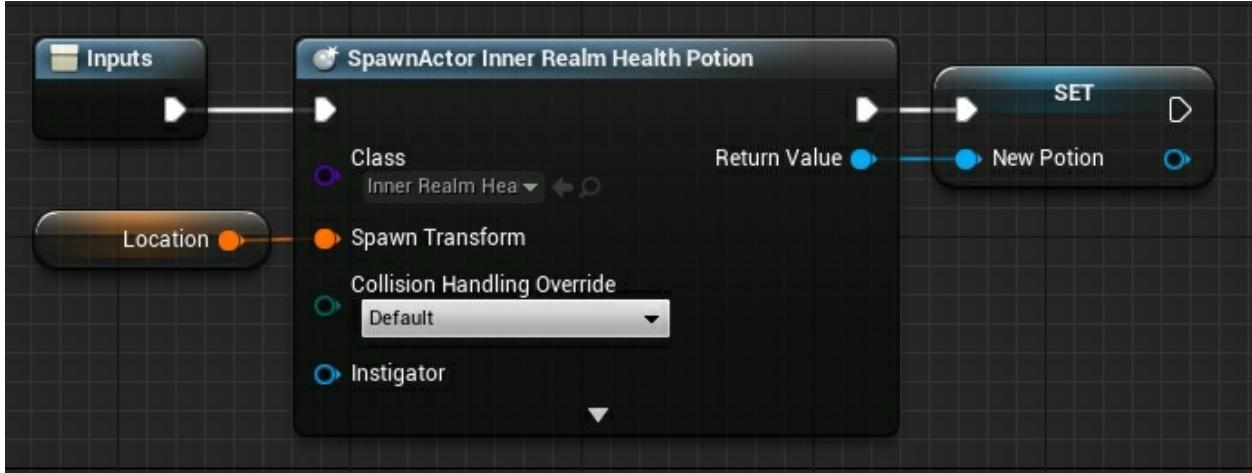
The *InnerRealmPotionShop* concrete creator class:

```
public class InnerRealmPotionShop extends PotionShop {  
  
    Potion concoctPotion(String potionSKU) {  
        if (potionSKU.equals("Health")) {  
            return new InnerRealmHealthPotion();  
        } else if (potionSKU.equals("Power")) {  
            return new InnerRealmPowerPotion();  
        } else if (potionSKU.equals("Skill")) {  
            return new InnerRealmSkillPotion();  
        } else return null;  
    }  
}
```

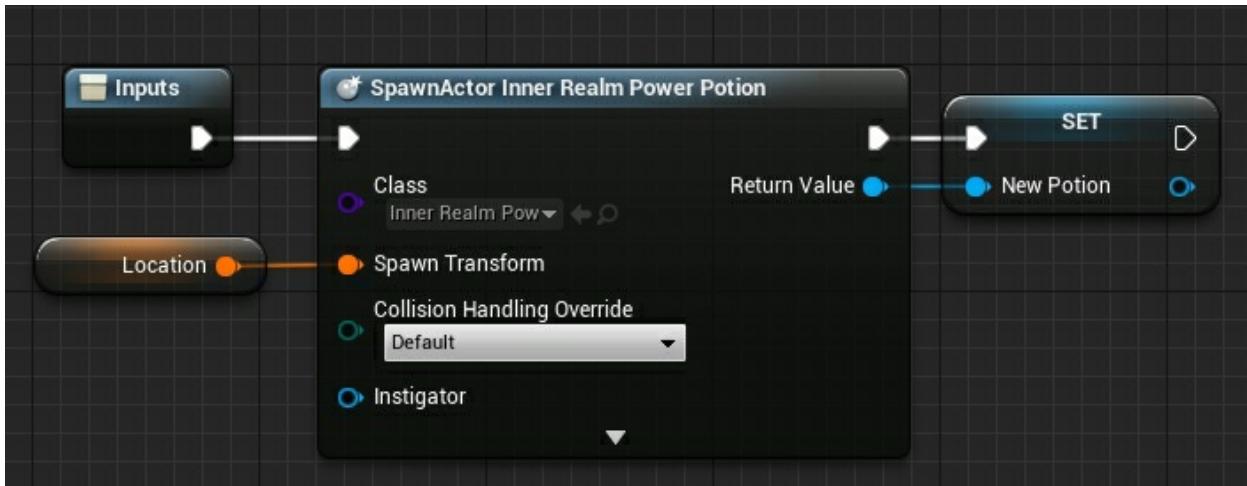
The *InnerRealmPotionShop* blueprint consists of only the *concotPotion* function:



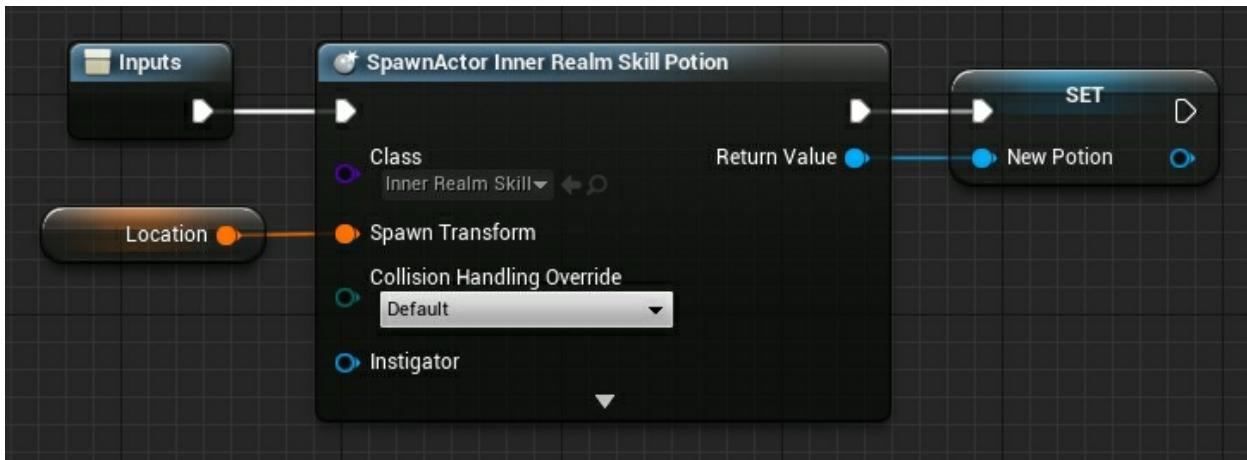
The *New\_InnerRealmHealthPotion* collapsed graph:



The *New\_InnerRealmPowerPotion* collapsed graph:



The `New_InnerRealmSkillPotion` collapsed graph:

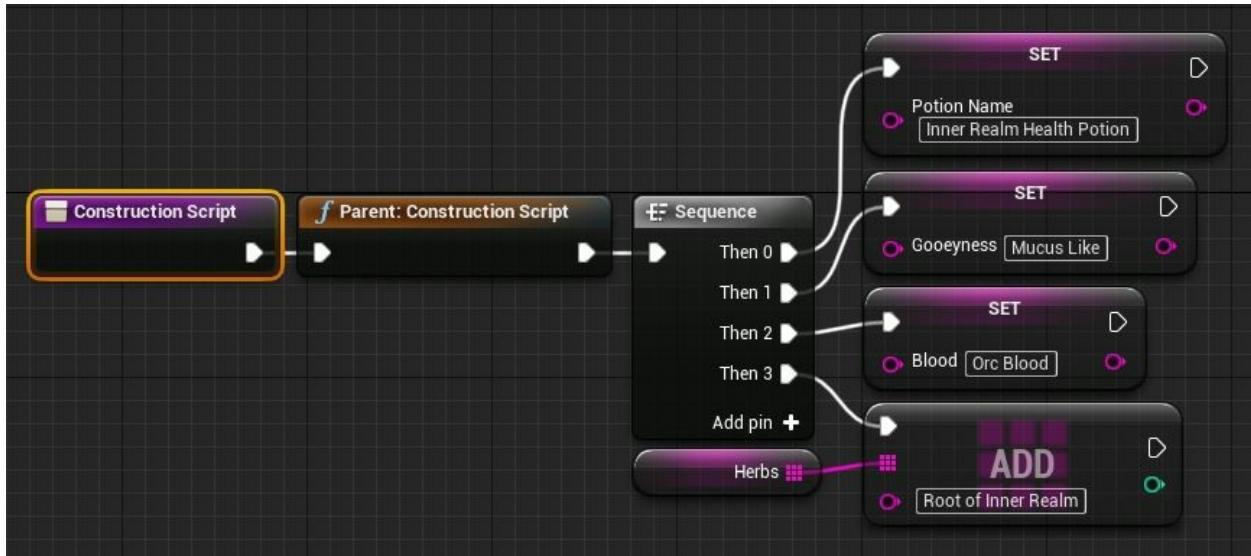


Now that we have our concrete creator class defined let's focus on the concrete products this creator can create.

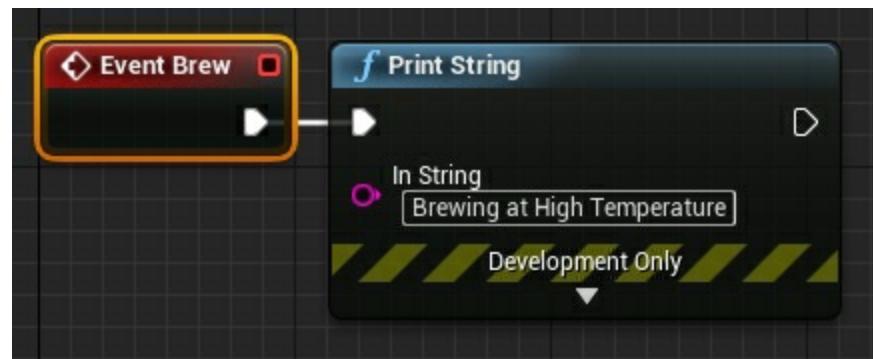
The `InnerRealmHealthPotion` concrete product class:

```
public class InnerRealmHealthPotion extends Potion {  
  
    public InnerRealmHealthPotion() {  
        potionName = "Inner Realm Health Potion";  
        gooeyness = "Mucus Like";  
        blood = "Orc Blood";  
  
        herbs.add("Root of Inner Realm");  
    }  
  
    void brew() {  
        System.out.println("Brewing at High Temperature");  
    }  
}
```

The *InnerRealmHealthPotion* blueprint construction script:



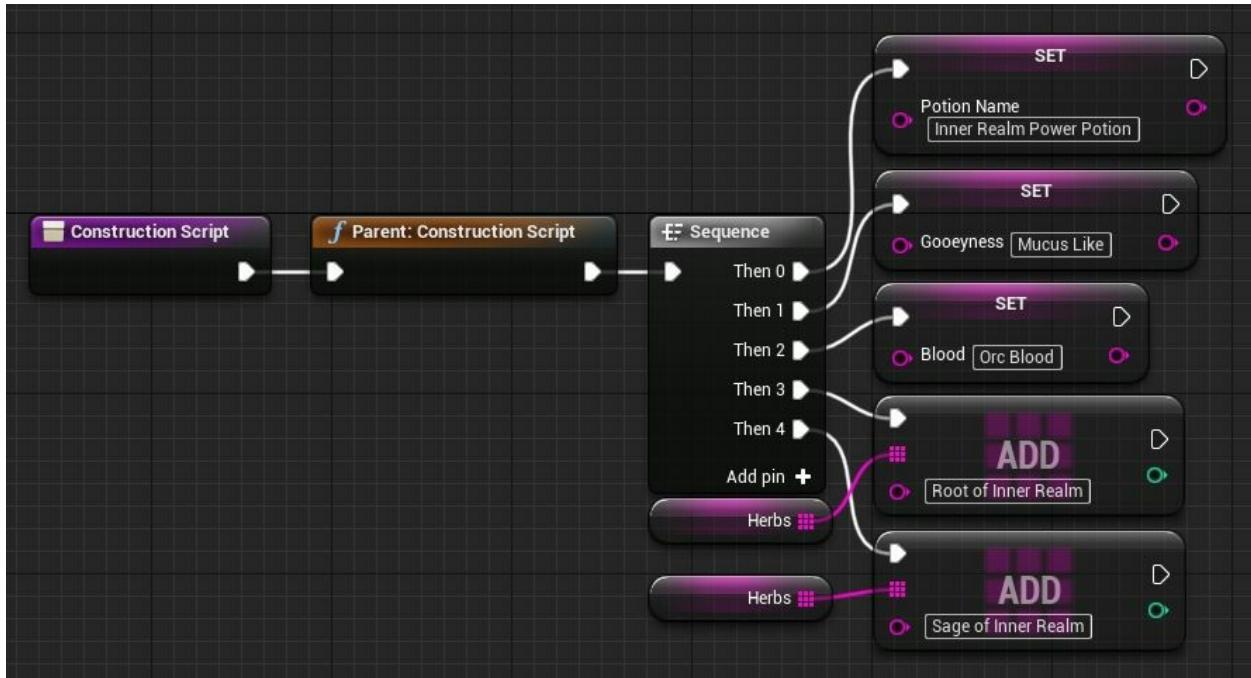
The *InnerRealmHealthPotion* blueprint event graph:



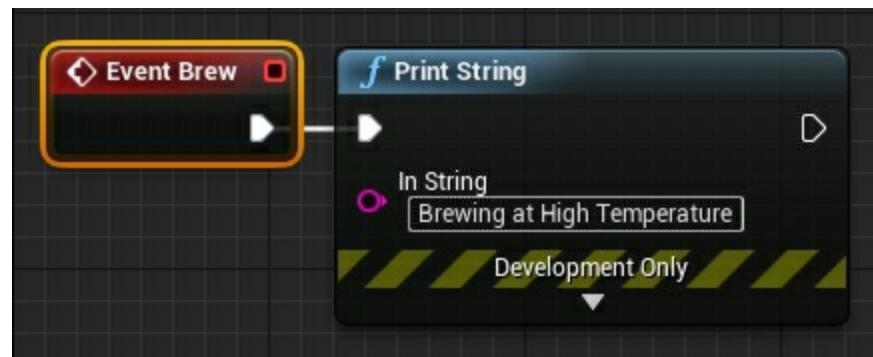
The *InnerRealmPowerPotion* concrete product class:

```
public class InnerRealmPowerPotion extends Potion {  
  
    public InnerRealmPowerPotion() {  
        potionName = "Inner Realm Power Potion";  
        gooeyness = "Mucus Like";  
        blood = "Orc Blood";  
  
        herbs.add("Root of Inner Realm");  
        herbs.add("Sage of Inner Realm");  
    }  
  
    void brew() {  
        System.out.println("Brewing at High Temperature");  
    }  
}
```

The *InnerRealmPowerPotion* blueprint construction script:



The *InnerRealmPowerPotion* blueprint event graph:



The *InnerRealmSkillPotion* concrete product class:

```
public class InnerRealmSkillPotion extends Potion {
```

```
    public InnerRealmSkillPotion() {
        potionName = "Inner Realm Skill Potion";
        gooeyness = "Mucus Like";
        blood = "Orc Blood";

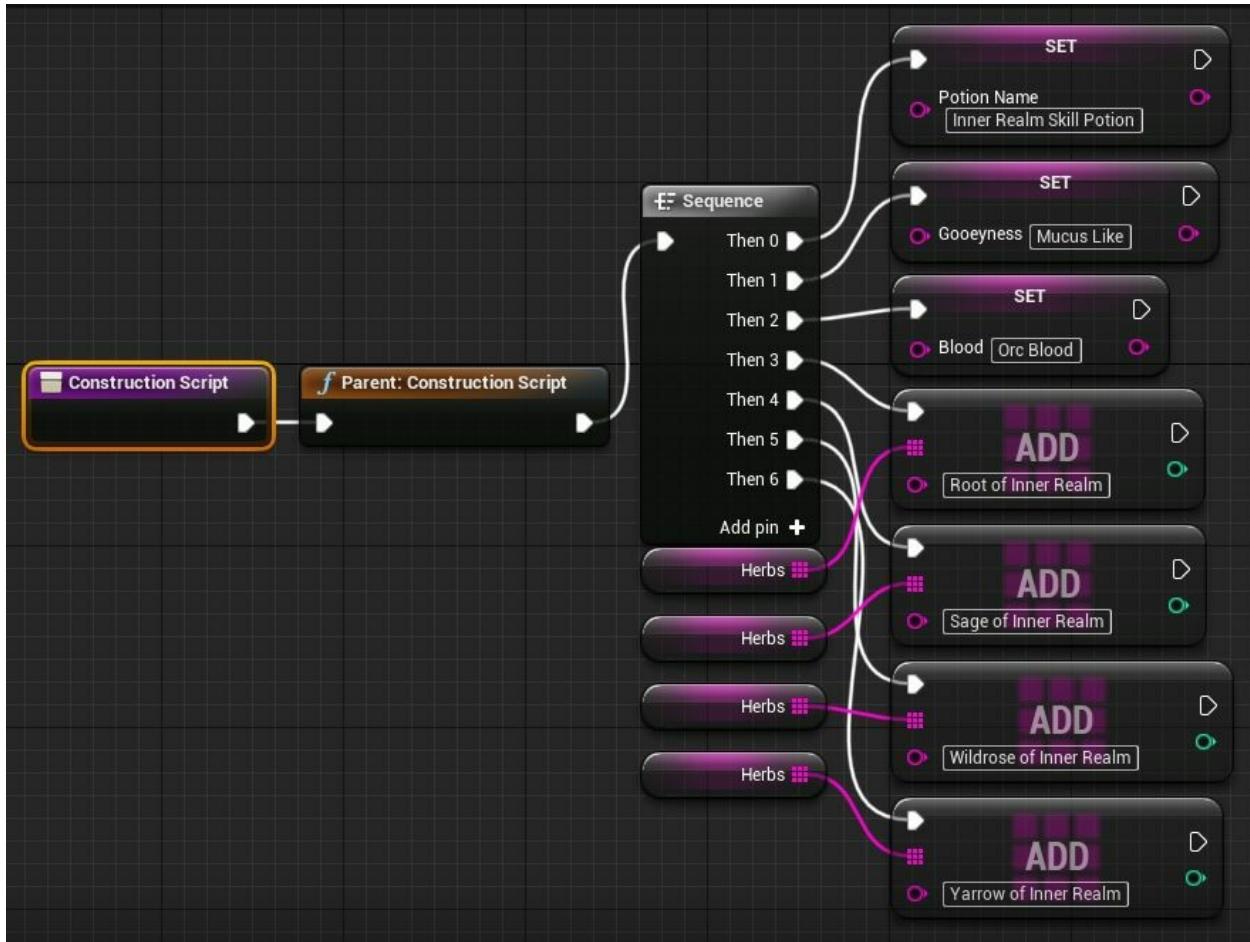
        herbs.add("Root of Inner Realm");
        herbs.add("Red Clover of Inner Realm");
```

```
herbs.add("Wildrose of Inner Realm");
herbs.add("Yarrow of Inner Realm");

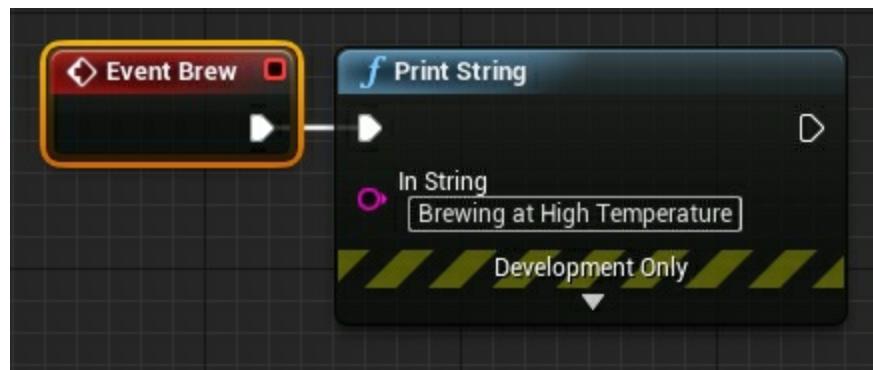
}

void brew() {
    System.out.println("Brewing at High Temperature");
}
}
```

The *InnerRealmSkillPotion* blueprint construction script:



The *InnerRealmSkillPotion* blueprint event graph:



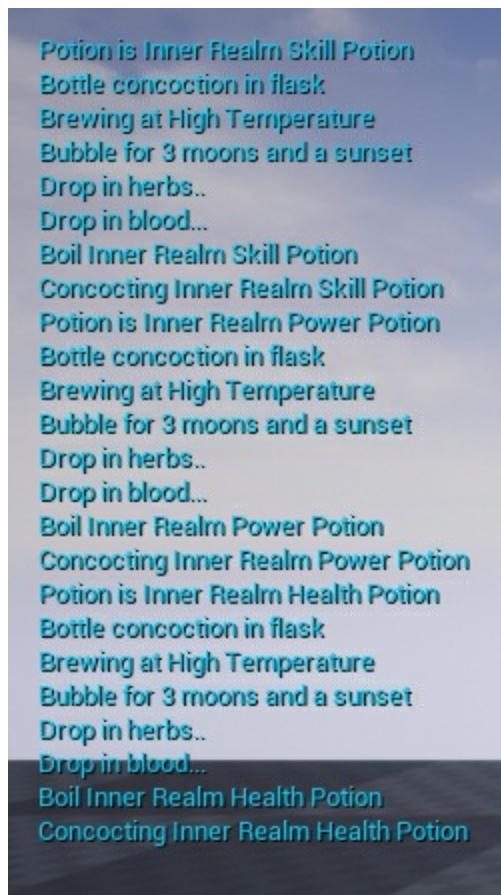
We can easily add another concrete creator class like *OuterRealmPotionShop* and create more concrete products using the same inheritance methods as the *InnerRealmPotionShop*:

```
public class OuterRealmPotionShop extends PotionShop {  
  
    Potion concoctPotion(String potionSKU) {  
        if (potionSKU.equals("Health")) {  
            return new OuterRealmHealthPotion();  
        } else if (potionSKU.equals("Power")) {  
            return new OuterRealmPowerPotion();  
        } else if (potionSKU.equals("Skill")) {  
            return new OuterRealmSkillPotion();  
        } else return null;  
    }  
}  
  
public class OuterRealmHealthPotion extends Potion {  
  
    public OuterRealmHealthPotion() {  
        potionName = "Outer Realm Health Potion";  
        gooeyness = "Watery Mess";  
        blood = "Dwarf Blood";  
  
        herbs.add("Root of Outer Realm");  
    }  
}  
  
public class OuterRealmPowerPotion extends Potion {  
  
    public OuterRealmPowerPotion() {  
        potionName = "Outer Realm Power Potion";  
        gooeyness = "Watery Mess";  
        blood = "Dwarf Blood";  
  
        herbs.add("Root of Outer Realm");  
        herbs.add("Sage of Outer Realm");  
    }  
}  
  
public class OuterRealmSkillPotion extends Potion {
```

```
public OuterRealmSkillPotion() {
    potionName = "Outer Realm Skill Potion";
    gooeyness = "Watery Mess";
    blood = "Dwarf Blood";

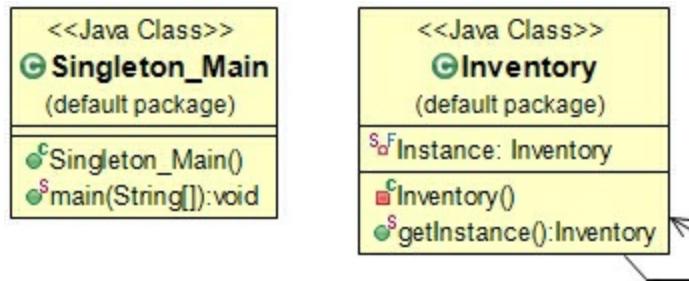
    herbs.add("Root of Outer Realm");
    herbs.add("Red Clover of Outer Realm");
    herbs.add("Wildrose of Outer Realm");
    herbs.add("Yarrow of Outer Realm");
}
```

The Factory Method viewport print:



# Programming Elitists' Bane... Singleton Pattern (Java)

## Singleton Pattern UML Diagram



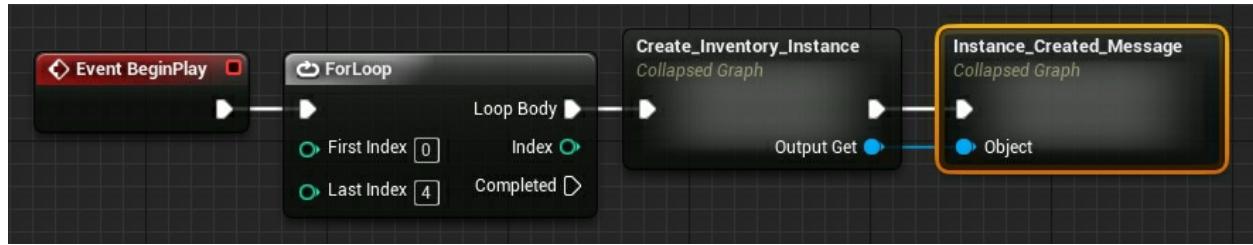
## **Singleton Pattern Implementation**

There is no clear-cut way to create a custom Singleton blueprint in Unreal. However, it appears the Game Instance is a first party solution to the Singleton. An inventory system is a great example for demonstrating the Singleton design pattern. We only want one inventory instance at all times. Additionally, we want to be able to easily access that inventory system throughout our game.

The main Singleton class *Singleton\_Main*:

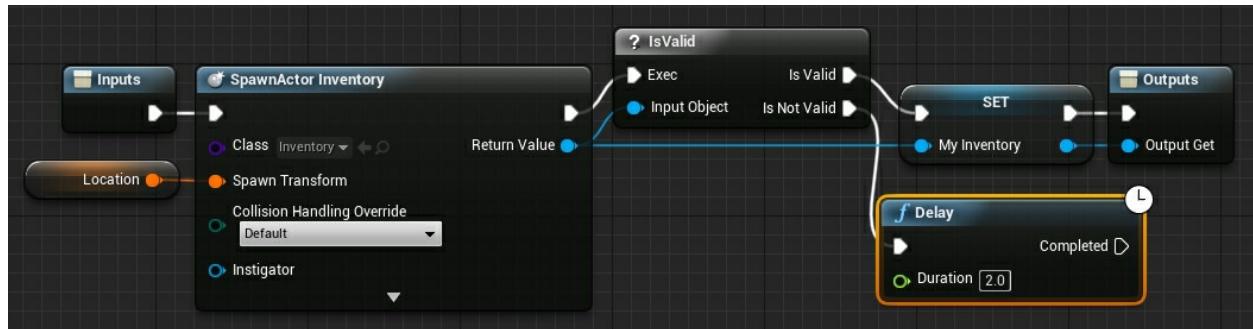
```
public class Singleton_Main {  
    public static void main(String[] args) {  
        Inventory myInventory = Inventory.getInstance();  
        System.out.println(myInventory);  
    }  
}
```

The *Singleton\_Main* blueprint event graph:

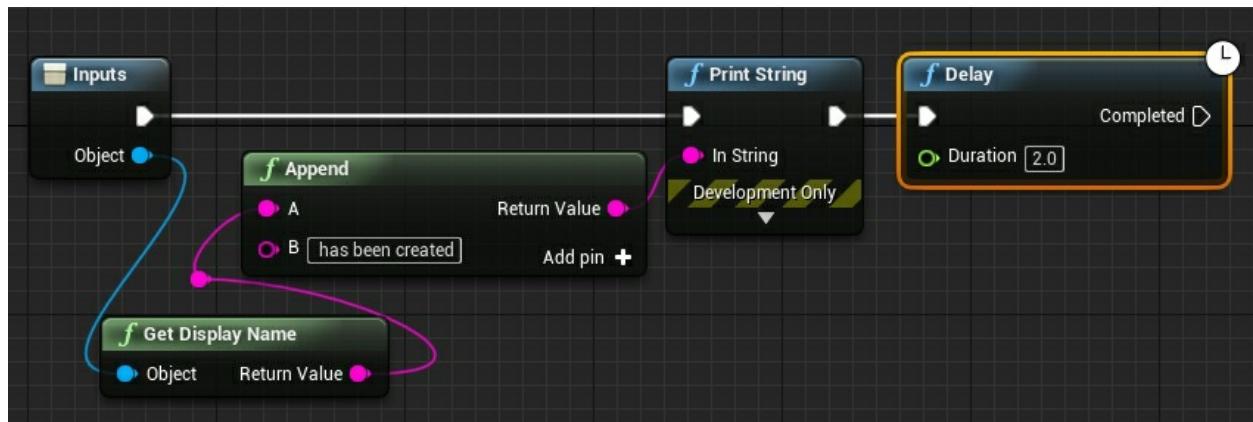


The for loop is present to demonstrate attempts to try to create more than one instance of our designated Singleton object. We should never be allowed to create multiple instances of our inventory If everything is set up correctly.

The *Create\_Inventory\_Instance* collapsed graph:



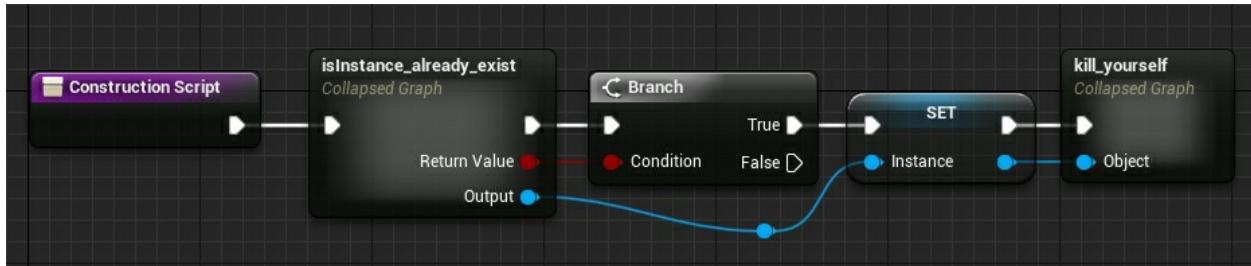
The *Instance\_Created\_Message* collapsed graph:



The *Inventory* class:

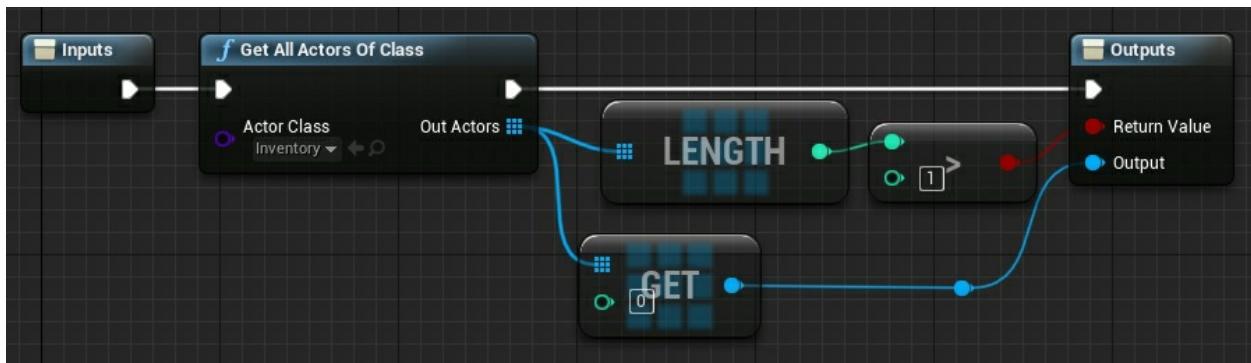
```
public final class Inventory {  
    private static final Inventory Instance = new Inventory();  
  
    private Inventory() {}  
  
    public static Inventory getInstance() {  
        return Instance;  
    }  
}
```

The *Inventory* blueprint construction script:

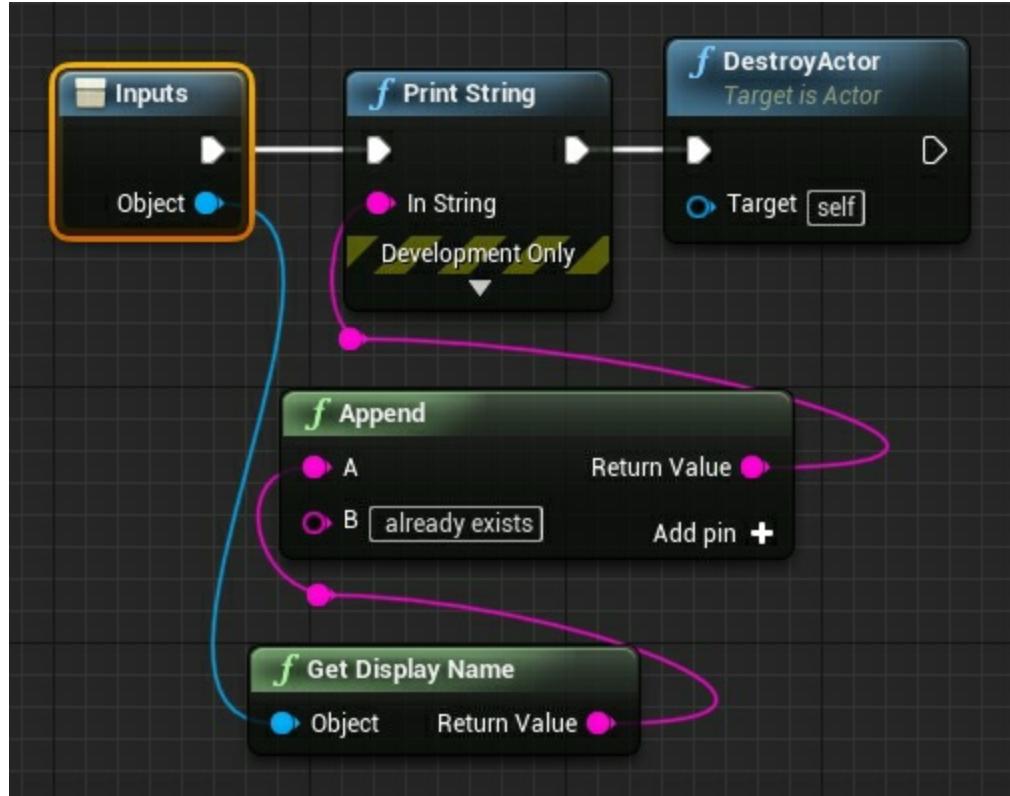


The concept of a private construction does not exist in Unreal. Therefore, we must create another way of keeping new Singleton objects from being created. We do this by checking if an inventory instance already exists. If an inventory instance does in fact already exists, the attempted new Singleton object is destroyed.

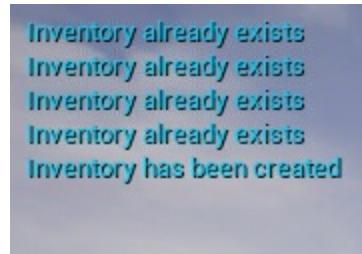
The *isInstance\_already\_exist* collapsed graph:



The *kill\_yourself* collapsed graph:



The Singleton pattern viewport screen print:

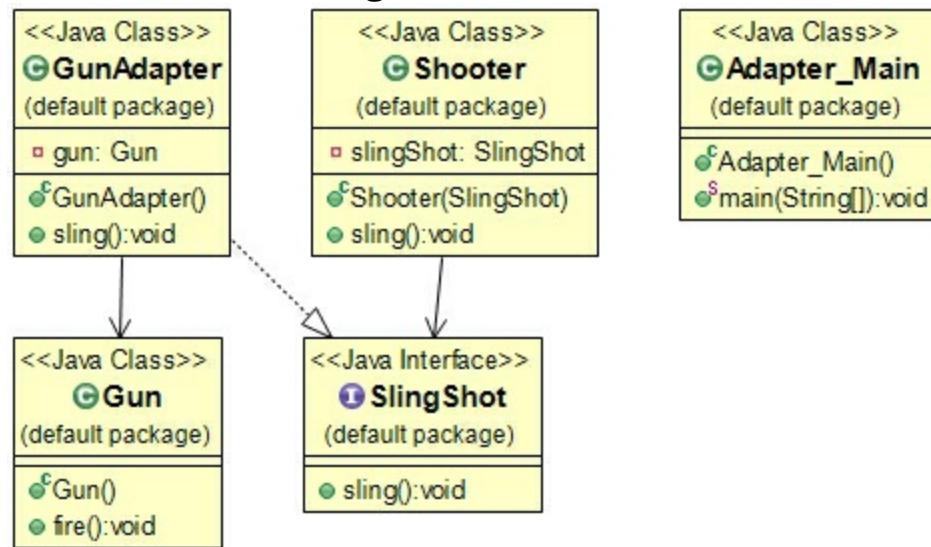


# Structural Patterns (Java)

The structural patterns we discuss help us to compose larger structures from classes and objects. One the most important goals of this pattern category is to be able to alter object composition at runtime.

# Adapt and Overcome... Adapter Pattern (Java)

Adapter Pattern UML Diagram



## **Adapter Pattern Implementation**

In our Adapter pattern implementation we have a shooter who uses a sling shot. We want to be able to allow our shooter to fire a gun. However, we want to use the existing code we have already implemented by using the Adapter pattern.

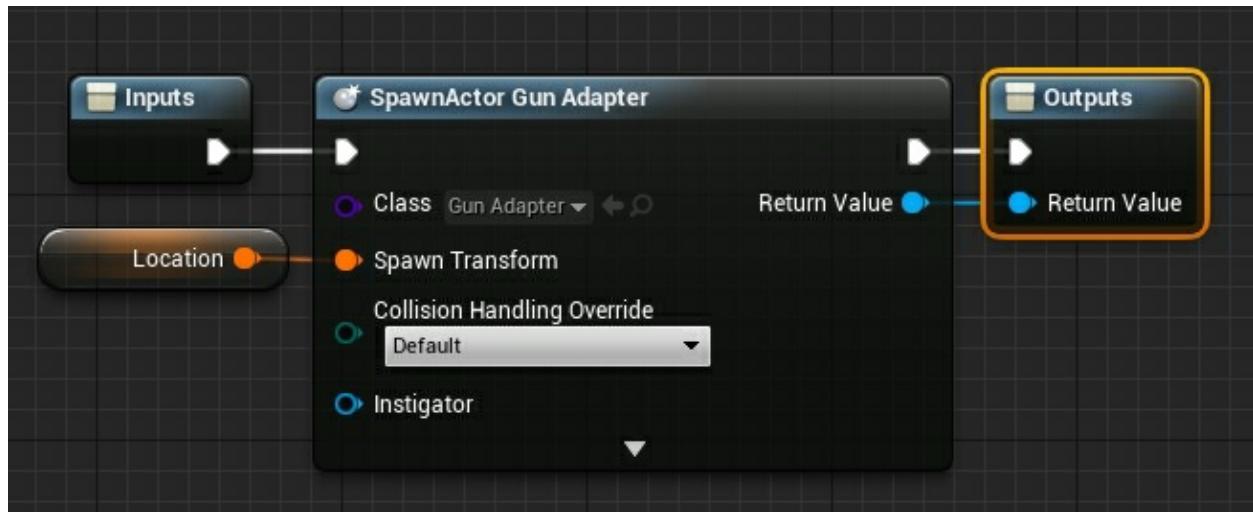
The main adapter class:

```
public class Adapter_Main {  
    public static void main(String[] args) {  
        Shooter shooter = new Shooter(new GunAdapter());  
        shooter.sling();  
    }  
}
```

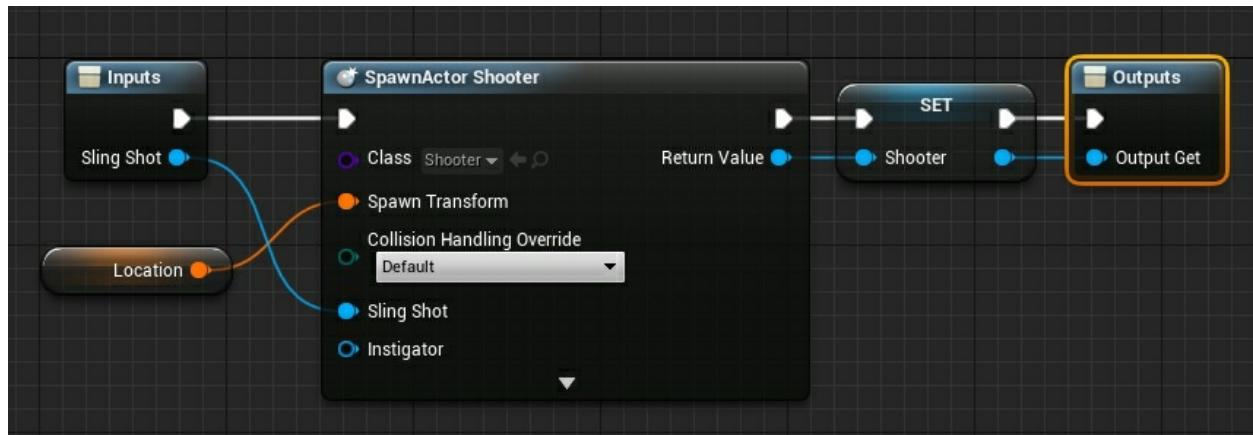
The *Adapter\_Main* blueprint:



The *Create\_Gun\_Adapter* collapsed graph:



The *Create\_Shooter* collapsed graph:



The *SlingShot* interface:

```
public interface SlingShot {
```

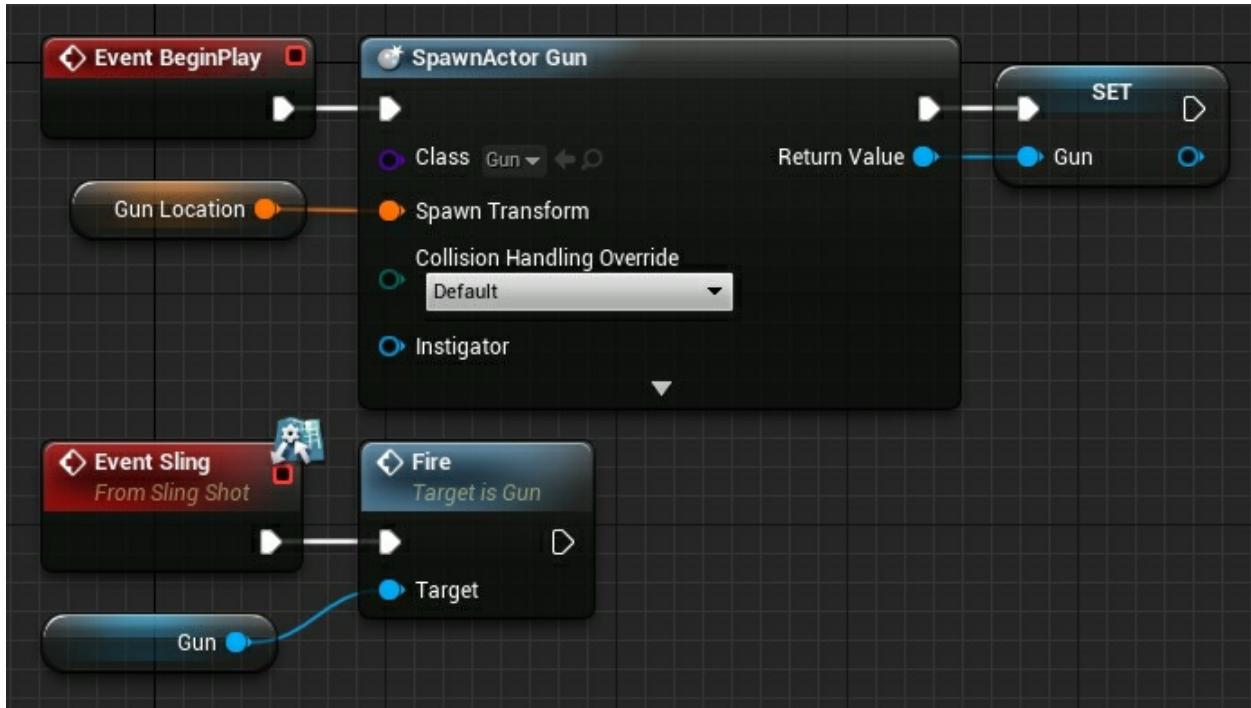
```
void sling();
```

}

The *GunAdapter* class:

```
public class GunAdapter implements SlingShot {  
    private Gun weapon;  
  
    public GunAdapter() {  
        weapon = new Gun();  
    }  
  
    @Override  
    public void sling() {  
        weapon.fire();  
    }  
}
```

The *GunAdapter* blueprint:



The *Gun* class:

```
public class Gun {  
  
    public void fire() {  
        System.out.println("Our gun is firing");  
    }  
  
}
```

The *Gun* blueprint:

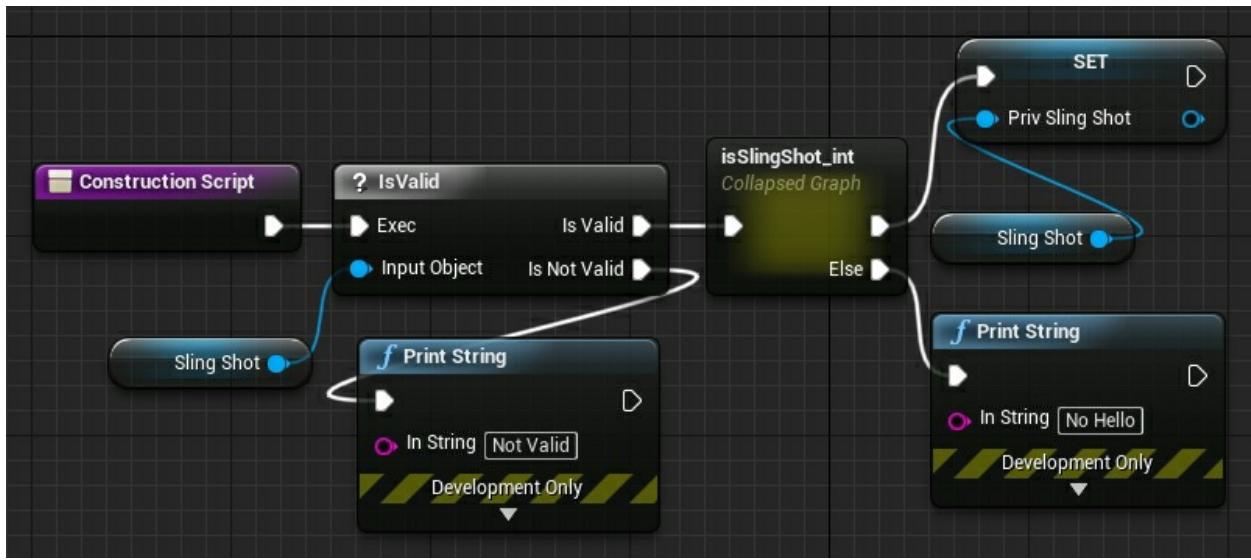


The *Shooter* class:

```
public class Shooter implements SlingShot {  
    private SlingShot slingShot;  
  
    public Shooter(SlingShot slingShot) {  
        this.slingShot = slingShot;  
    }  
  
    public void sling0 {  
        slingShot.sling0;  
    }  
}
```

We can see that the *Shooter* class has no functionality to fire a gun. However, it does functionality to sling a slingshot. Both of these skills are predicated on emitting a projectile from a weapon. We would like to reuse the slingshot functionality and adapt it for the use of a gun.

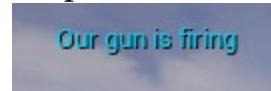
The *Shooter* construction script:



The *Sling* function:



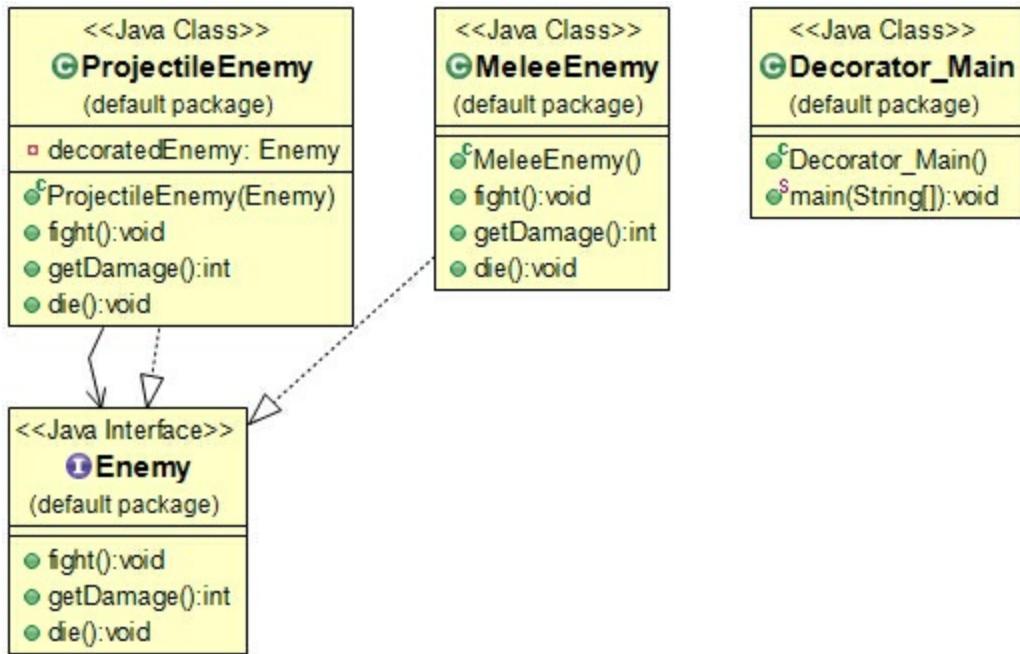
The Adapter pattern viewport print:



# Decorating Overpowered Enemies...

## Decorator Pattern (Java)

Decorator Pattern UML Diagram



## **Decorator Pattern Implementation**

Our implementation consists of an enemy who is initially a melee enemy. We want to “Decorate” the enemy and make an enemy who can use projectiles. Thus, the added functionality is the projectile ability.

The *Decorator\_Main* class:

```
public class Decorator_Main {  
    public static void main(String[] args) {  
        System.out.println("Melee Enemies are on the horizon");  
        Enemy enemy = new MeleeEnemy();  
        enemy.fight();  
        enemy.die();  
        System.out.println("Melee Enemies cause "+ enemy.getDamage()  
+" damage.");  
  
        System.out.println("Enemies are now armed with guns");  
        Enemy projectileEnemy = new ProjectileEnemy(enemy);  
        projectileEnemy.fight();  
        projectileEnemy.die();  
        System.out.println("Projectile Enemies cause "+  
projectileEnemy.getDamage()+" damage.");  
    }  
}
```

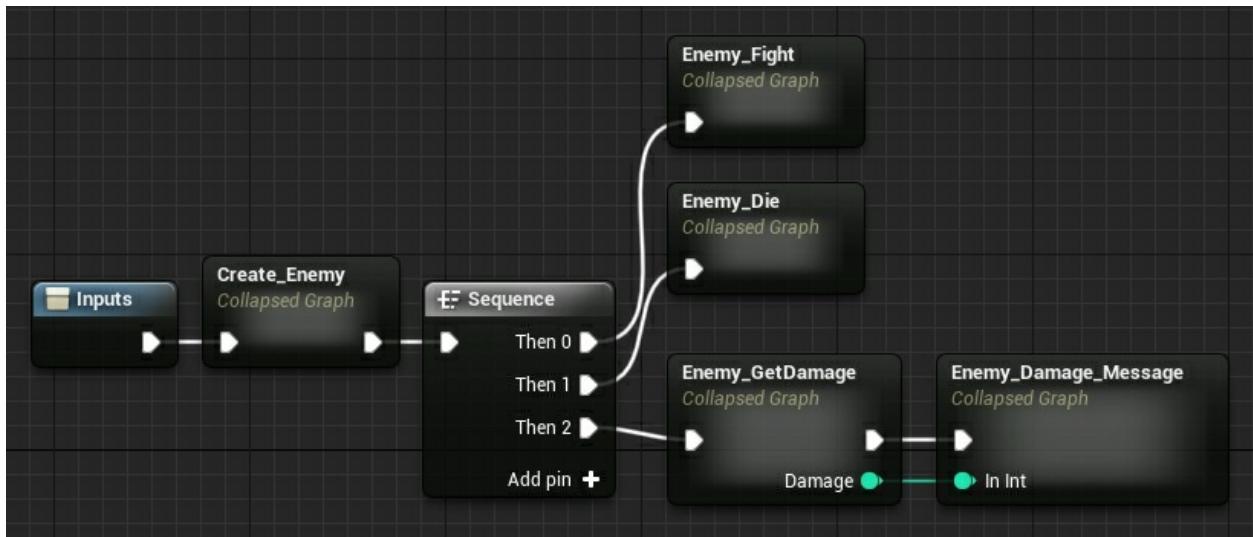
We can see from the *Decorator\_Main* that we do not need to make a separate *Enemy* class to add features. We merely “Decorate” the enemy, and now we have a projectile enemy. Effectively, we changed the behavior of the enemy at runtime by using a decorator.

The *Decorator\_Main* blueprint event graph:

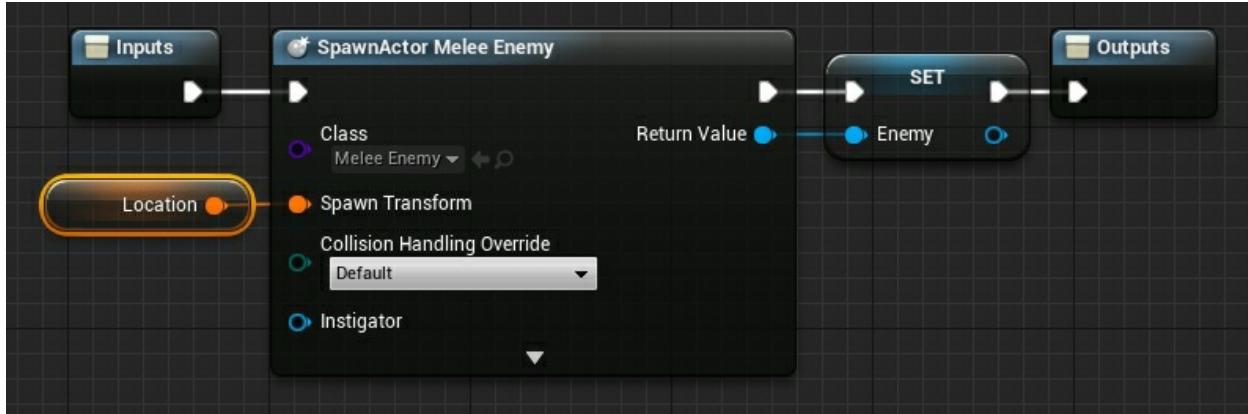


The melee enemy and the decorated projectile enemy are broken down into separate collapsed graphs for readability.

The *Melee\_Enemy\_Actions* collapsed graph:



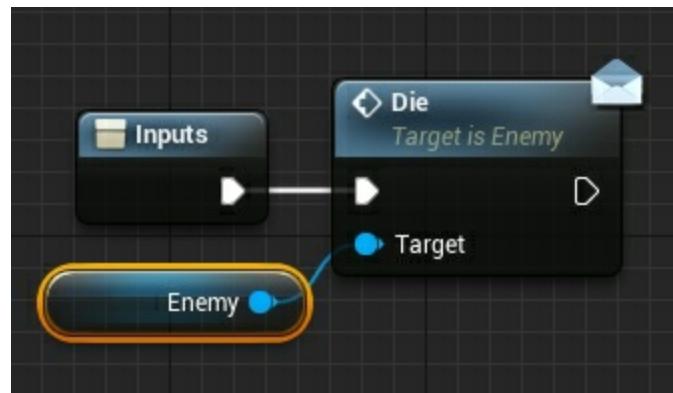
The *Create\_Enemy* collapsed graph:



The *Enemy\_Fight* collapsed graph:



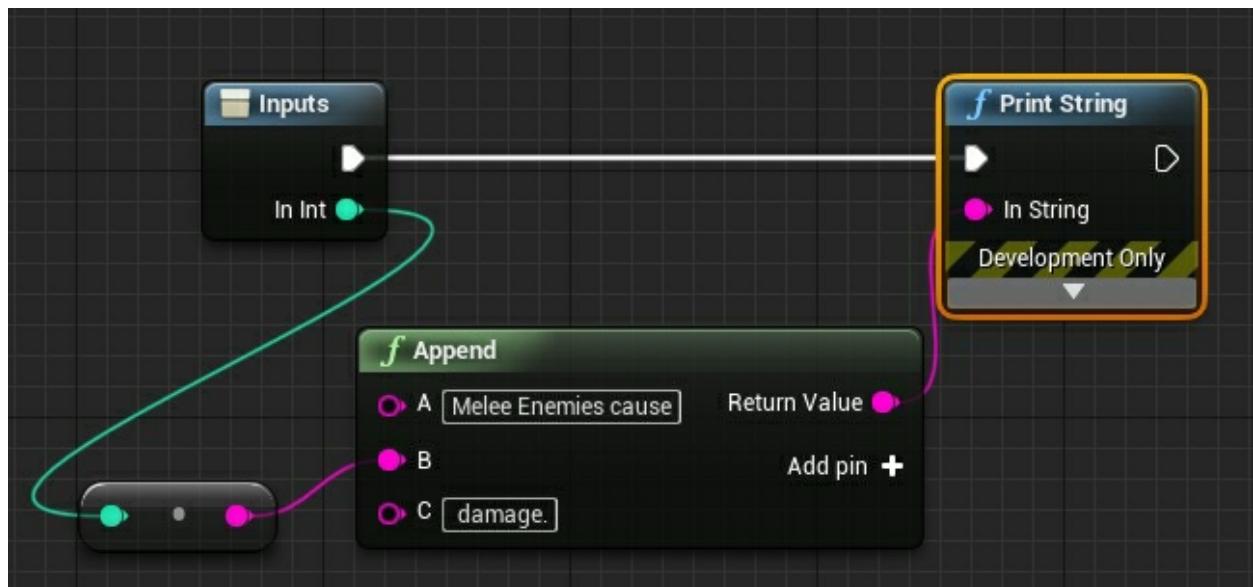
The *Enemy\_Die* collapsed graph:



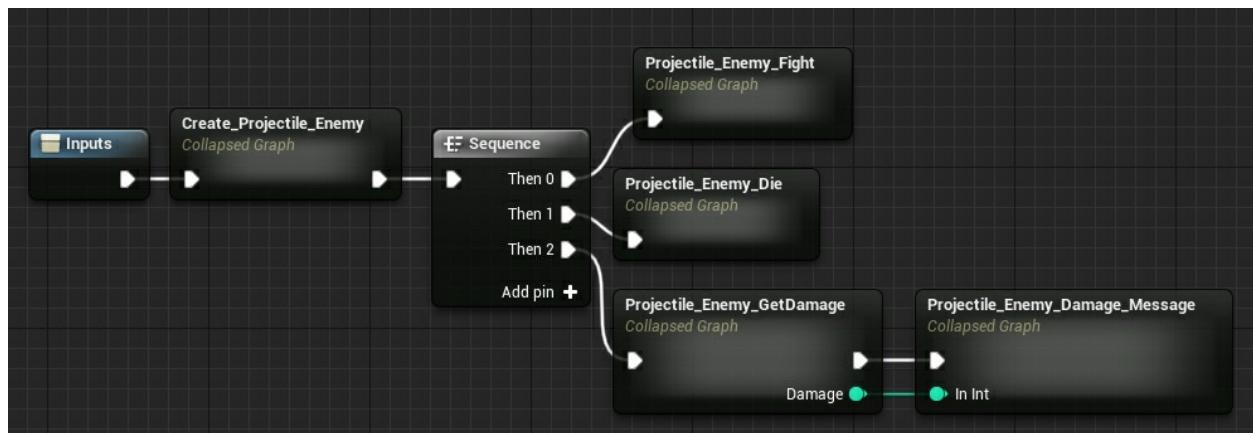
The *Enemy\_GetDamage* collapsed graph:



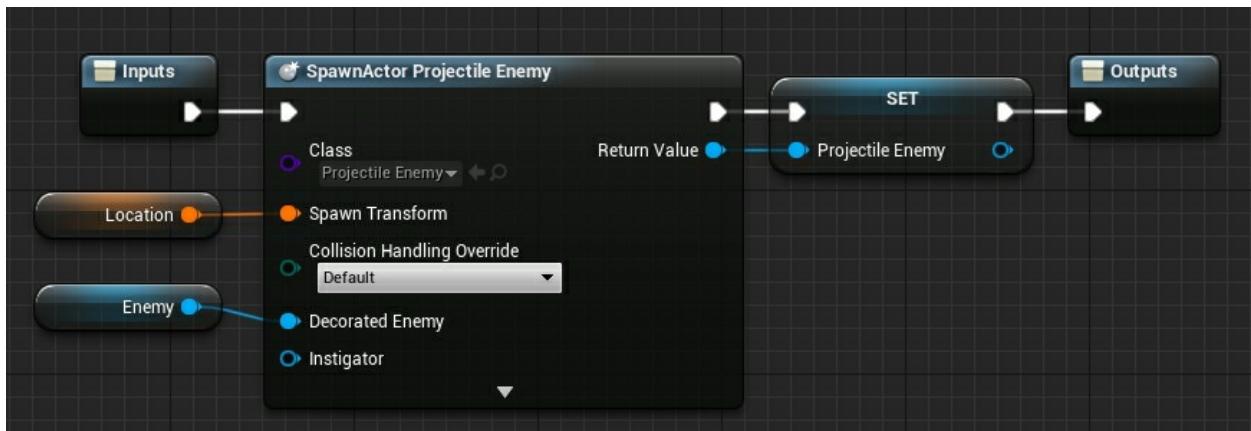
The *Enemy\_Damage\_Message* collapsed graph:



The *Projectile\_Enemy\_Actions* collapsed graph:



The *Create\_Projectile\_Enemy* collapsed graph:



The notable difference between the melee enemy and the projectile enemy is construction. The projectile enemy constructor accepts the previously created enemy as an argument. This is so we can reuse it to decorate an enemy object.

The *Projectile\_Enemy\_Fight* collapsed graph:



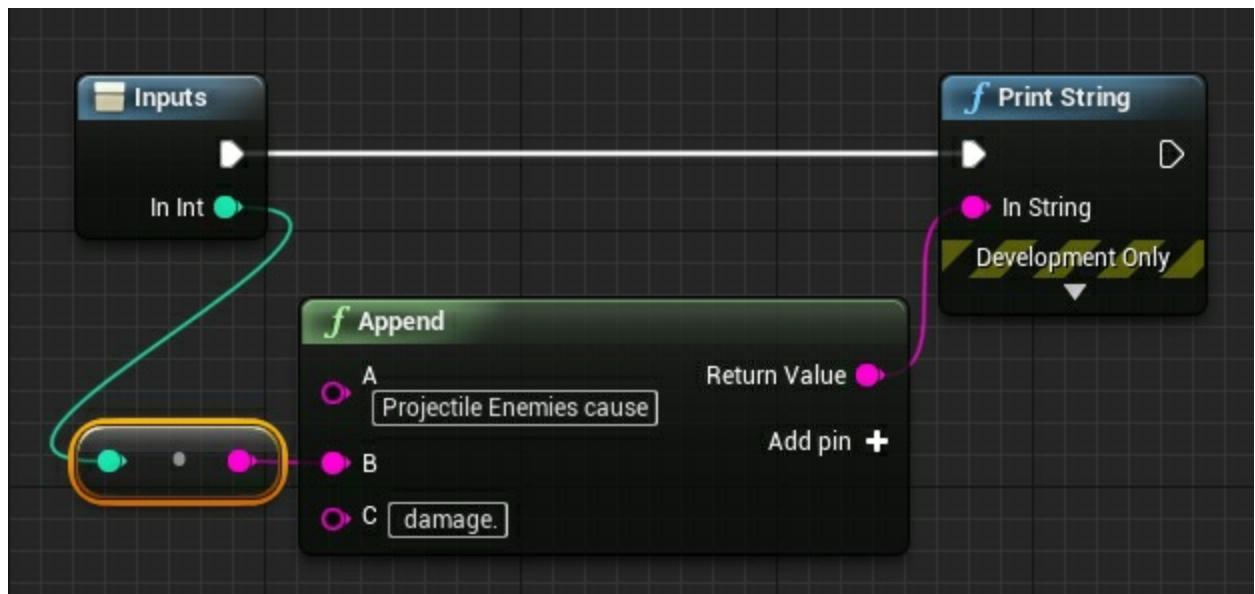
The *Projectile\_Enemy\_Die* collapsed graph:



The *Projectile\_Enemy\_GetDamage* collapsed graph:



The *Projectile\_Enemy\_Damage\_Message* collapsed graph:



The *Enemy* Interface:

```
public interface Enemy {  
    void fight();  
  
    //How much damage the enemy gives  
    int getDamage();  
  
    void die();
```

}

The *Enemy* interface is shared across all enemy objects

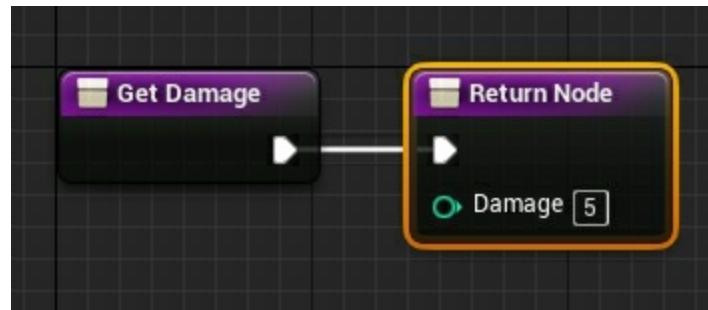
The *MeleeEnemy* class:

```
public class MeleeEnemy implements Enemy {  
  
    @Override  
    public void fight() {  
        System.out.println("The enemy throws heavy punches");  
    }  
  
    @Override  
    public int getDamage() {  
        return 5;  
    }  
  
    @Override  
    public void die() {  
        System.out.println("The enemy writhes in agony and disintegrates");  
    }  
}
```

The *MeleeEnemy* blueprint event graph:



The *GetDamage* function in the *MeleeEnemy* blueprint:

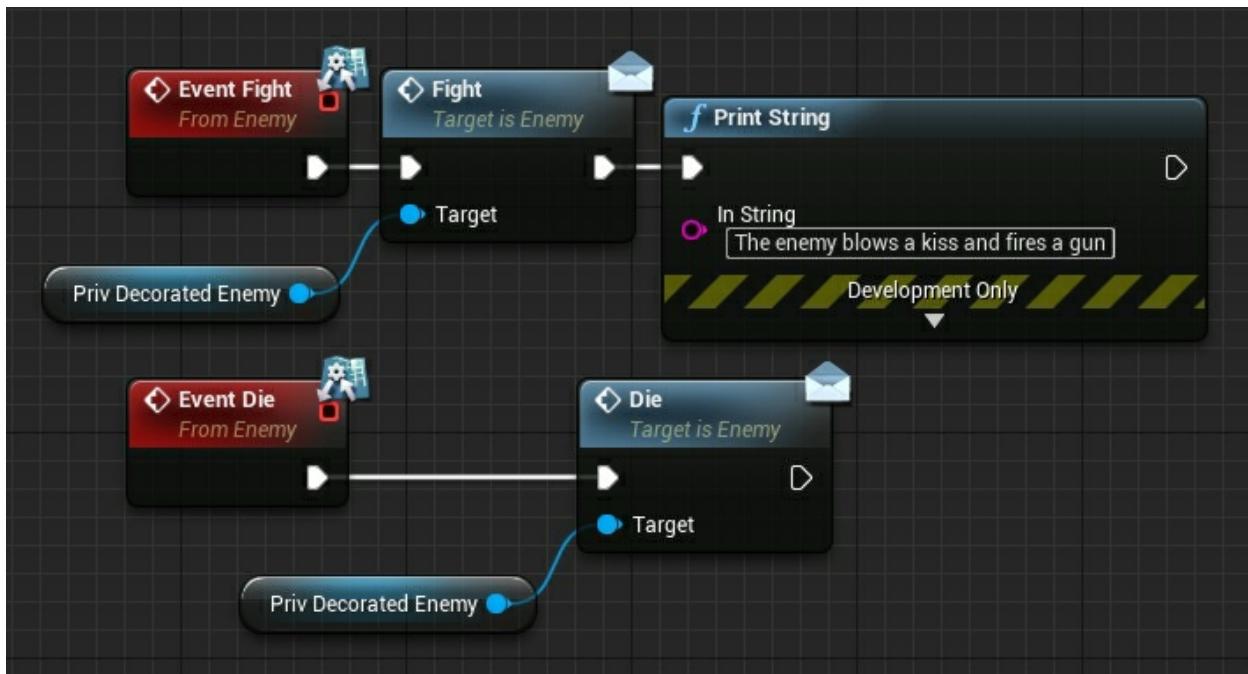


The *ProjectileEnemy* class:

```
public class ProjectileEnemy implements Enemy {  
  
    private Enemy decoratedEnemy;  
  
    public ProjectileEnemy(Enemy decoratedEnemy) {  
        this.decoratedEnemy = decoratedEnemy;  
    }  
  
    @Override  
    public void fight() {  
        decoratedEnemy.fight();  
        System.out.println("The enemy blows a kiss and fires a gun");  
    }  
  
    @Override  
    public int getDamage() {  
        return decoratedEnemy.getDamage() + 95;  
    }  
  
    @Override  
    public void die() {  
        decoratedEnemy.die();  
    }  
}
```

We can see that the projectile enemy causes more damage than the melee enemy. We use the melee damage causing attribute and we increase it by 95 for the projectile enemy.

The *ProjectileEnemy* blueprint event graph:



The *GetDamage* function in the *ProjectileEnemy* blueprint:



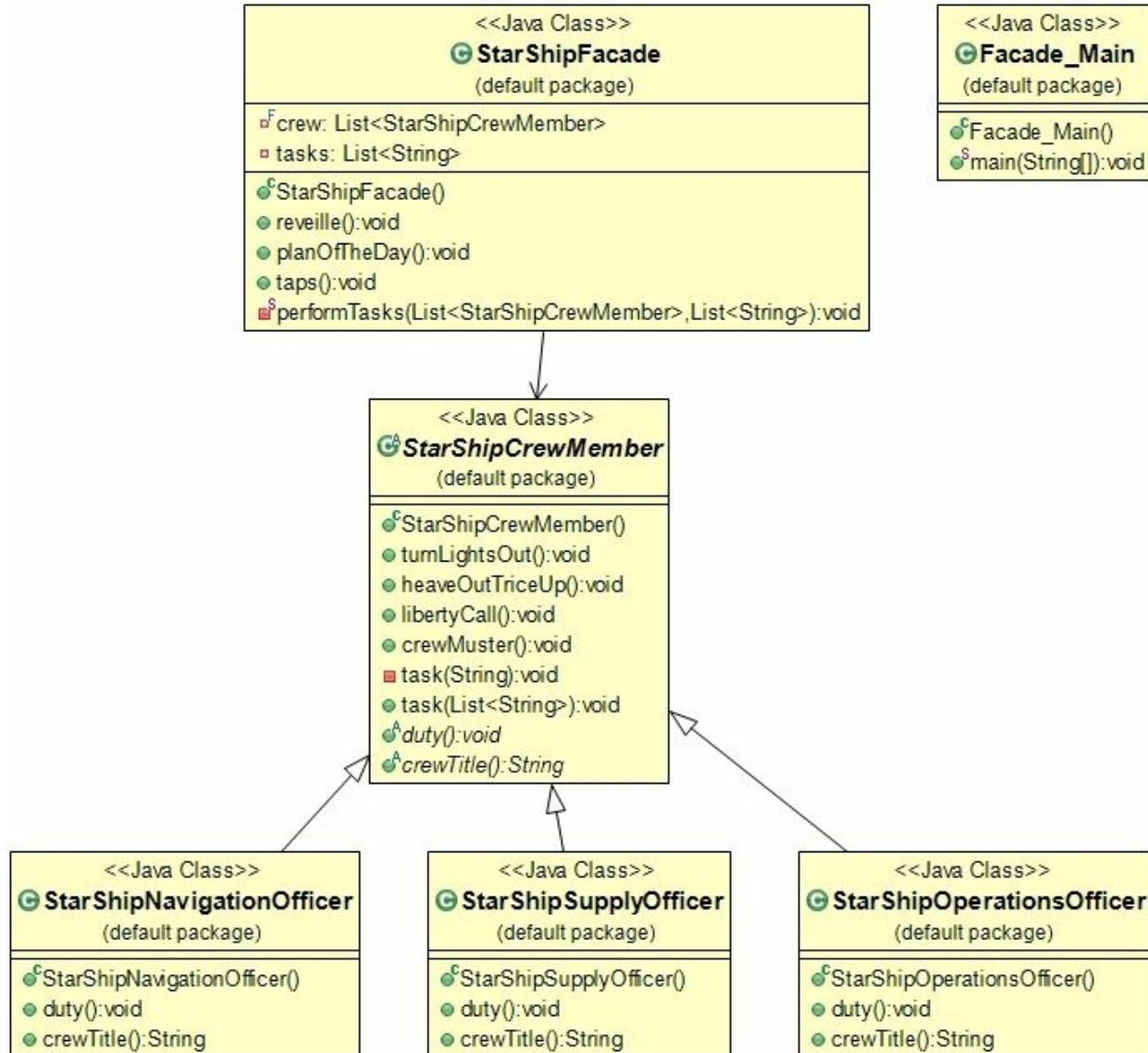
Decorator Pattern viewport print:

Projectile Enemies cause 100 damage.  
The enemy writhes in agony and disintegrates  
The enemy blows a kiss and fires a gun  
The enemy throws heavy punches  
MeleeEnemies Implements Enemy  
Melee Enemies cause 5 damage.  
The enemy writhes in agony and disintegrates  
The enemy throws heavy punches  
Melee Enemies are on the horizon

We can see the melee enemies coming. The melee enemy fights us with punches. The enemy then dies. However, we take a 5-count damage from the melee enemy. The melee enemy is then given new functionality in the form of a projectile. The enemy can punch and shoot a gun causing damage of 100.

# Space Travel Looks Easy... Façade Pattern (Java)

Façade Pattern UML Diagram



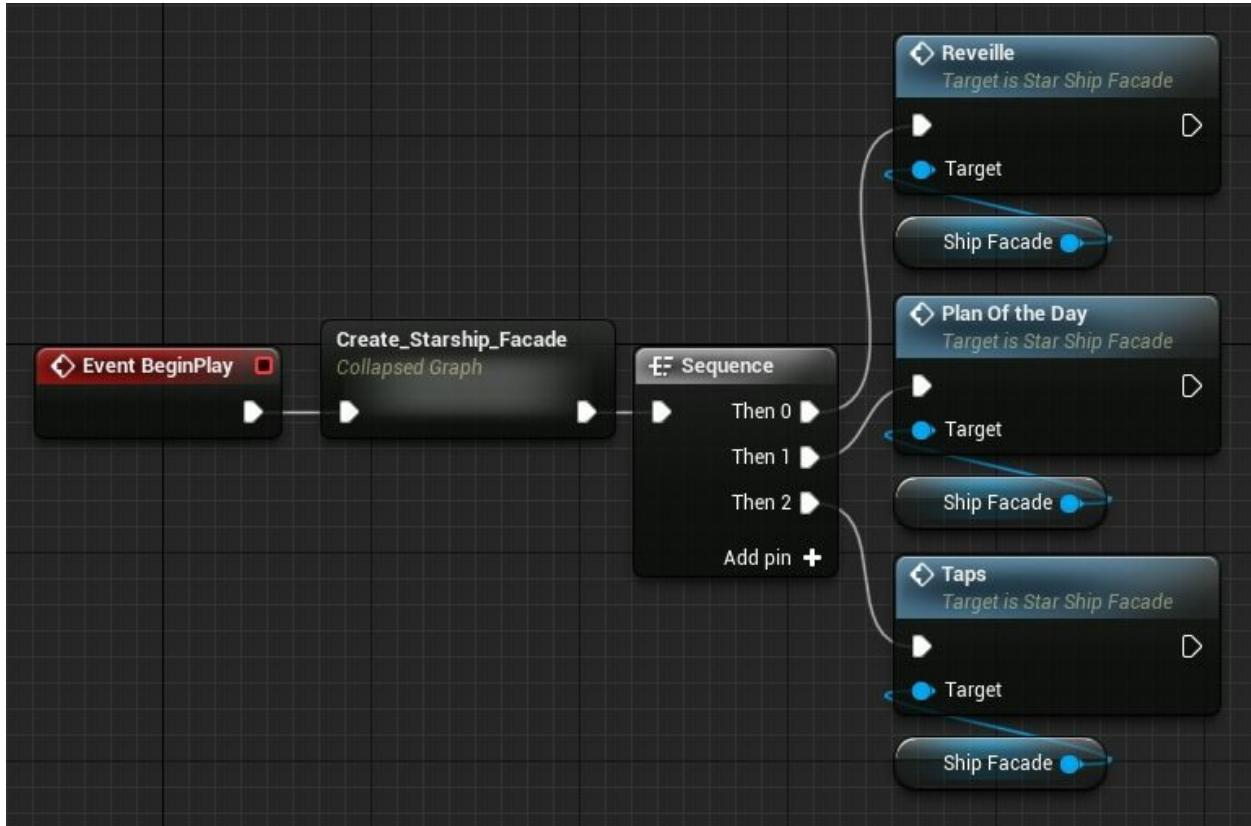
## Façade Pattern Implementation

Space travel is not easy. The most important part of space travel is the starship. However, the starship has complexity of its own. This is where the *StarShipFacade* comes into play. The Façade main class:

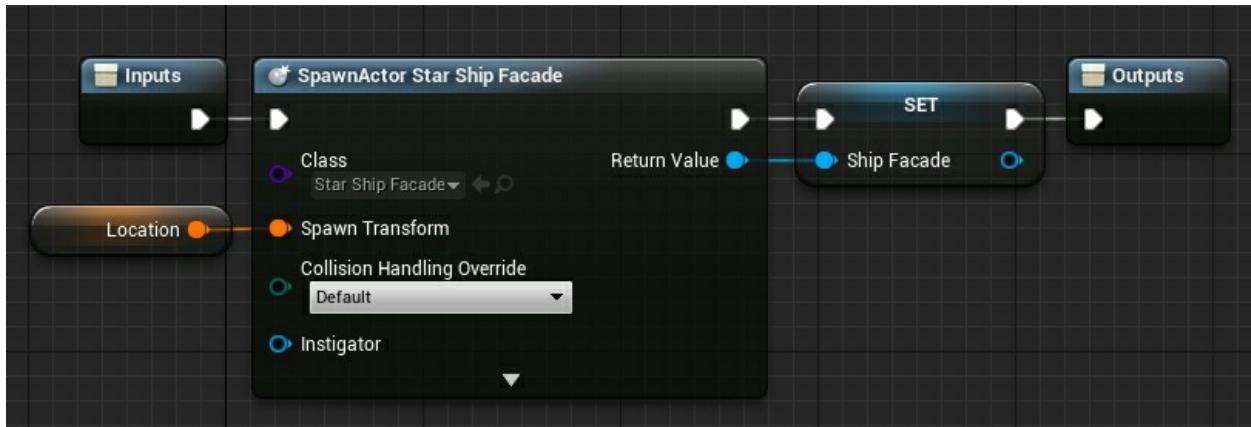
```
public class Facade_Main {  
  
    public static void main(String[] args) {  
        StarShipFacade shipFacade = new StarShipFacade();  
        shipFacade.reveille();  
        shipFacade.planOfTheDay();  
        shipFacade.taps();  
    }  
  
}
```

From *Façade\_Main* we can see how easy it is to run a starship. This is merely a façade using the Façade pattern. The complexity lies ahead.

The *Façade\_Main* event graph:



The *Create\_Starship\_Facade* collapsed graph:



The *StarShipFacade* class:

```
import java.util.ArrayList;
import java.util.List;

public class StarShipFacade {
    private final List<StarShipCrewMember> crew;

    private List<String> tasks;

    public StarShipFacade() {
        crew = new ArrayList<>();
        tasks= new ArrayList<>();
        crew.add(new StarShipNavigationOfficer());
        crew.add(new StarShipSupplyOfficer());
        crew.add(new StarShipOperationsOfficer());
    }

    public void reveille() {
        tasks.clear();
        tasks.add("trice_up");
        tasks.add("muster");
        performTasks(crew, tasks);
    }

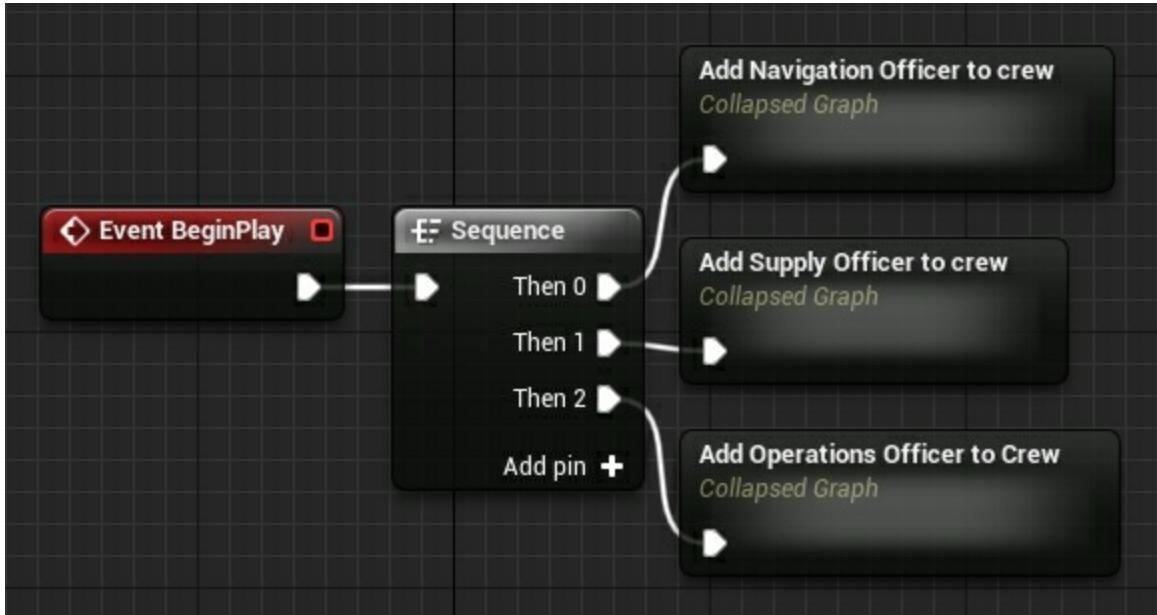
    public void planOfTheDay() {
        tasks.clear();
        tasks.add("duty");
        performTasks(crew, tasks);
    }

    public void taps() {
        tasks.clear();
        tasks.add("liberty_call");
        tasks.add("lights_out");
        performTasks(crew, tasks);
    }

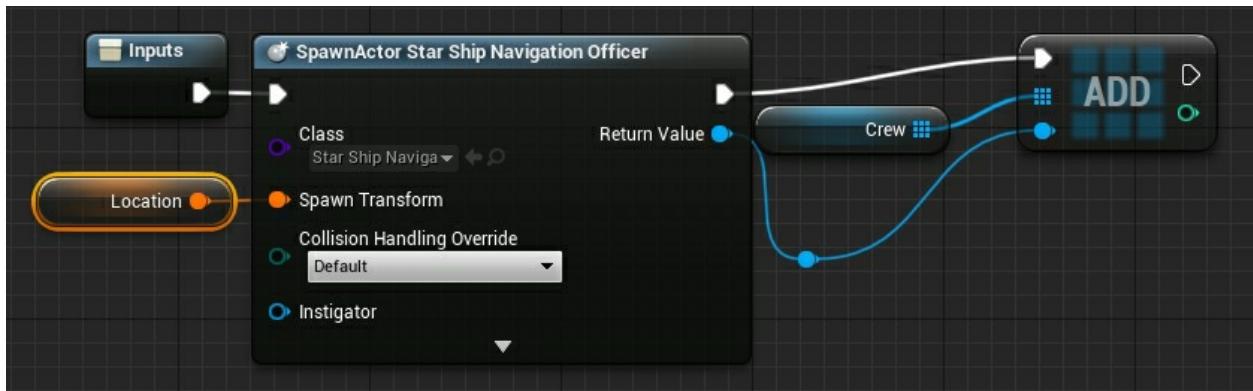
    private static void performTasks(List<StarShipCrewMember> crew,
```

```
List<String> tasks) {  
    for (StarShipCrewMember crewMember : crew) {  
        crewMember.task(tasks);  
    }  
}
```

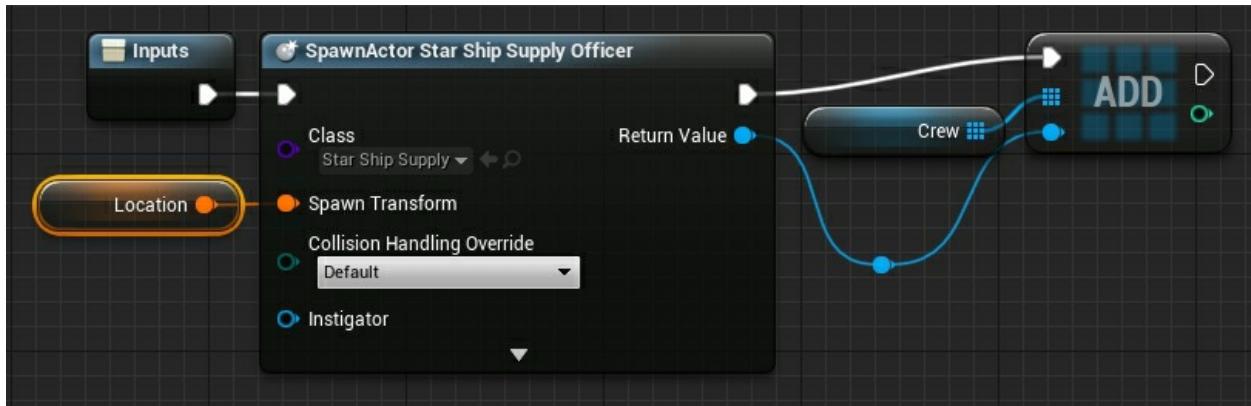
The *StarShipFacade* class is where the meat of this pattern lies. We did not use the construction script as demonstrated in the code. We placed the construction code inside of the event graph. The *StarshipFacade* event graph:



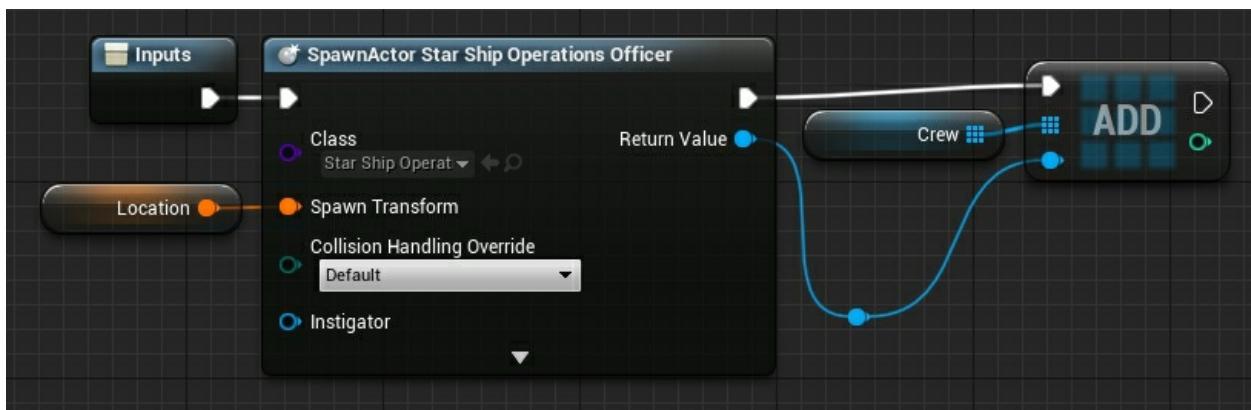
The *Add Navigation Officer to crew* collapsed graph:



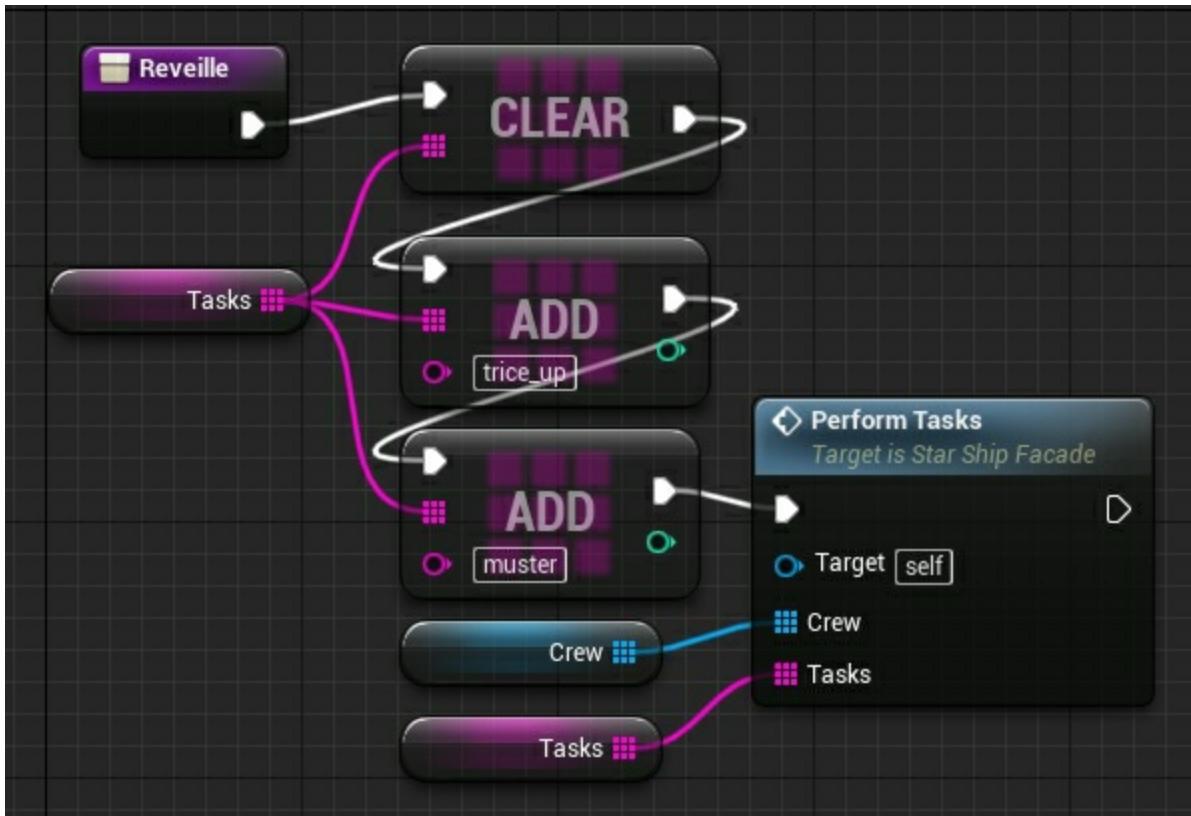
The *Add Supply Officer to crew* collapsed graph:



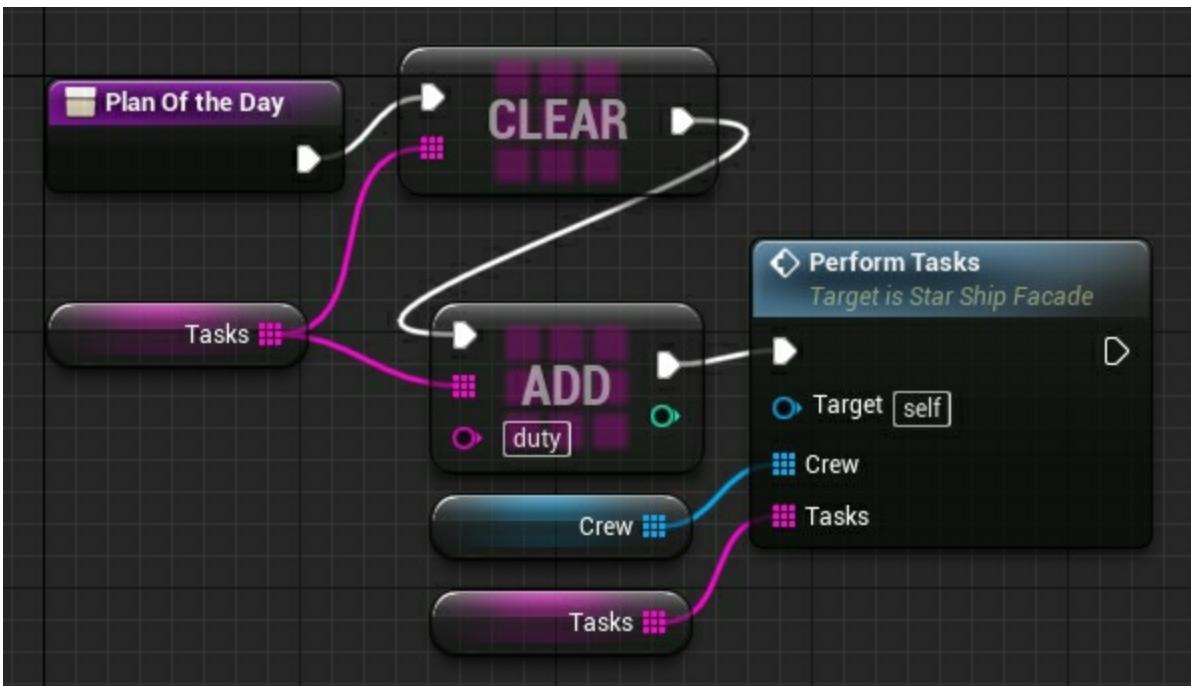
The *Add Operations Officer to Crew* collapsed graph:



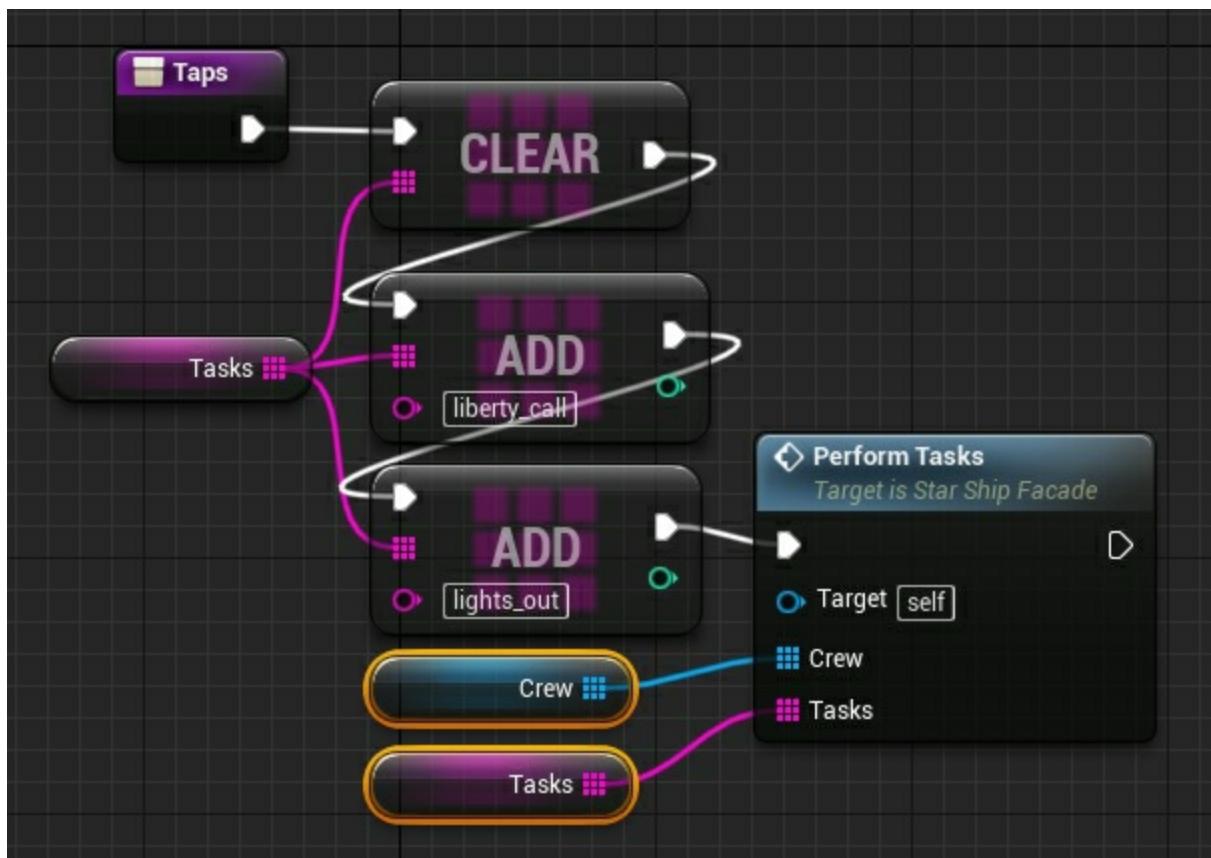
The *Reveille* blueprint function:



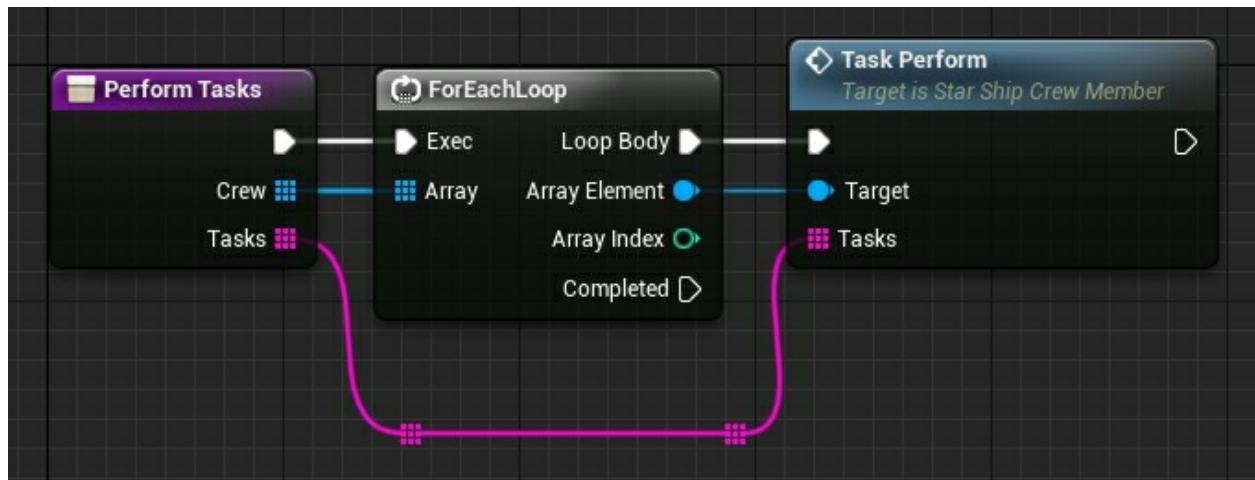
The *PlanOfTheDay* blueprint function:



The *Taps* blueprint function:



The *Perform Tasks* blueprint function:



The abstract class *StarShipCrewMember*:

```
import java.util.List;

public abstract class StarShipCrewMember {

    public void turnLightsOut() {
        System.out.println(crewTitle() +
            " taps,taps lights out.");
    }

    public void heaveOutTriceUp() {
        System.out.println(crewTitle() +
            " all hands heave out and trice up.");
    }

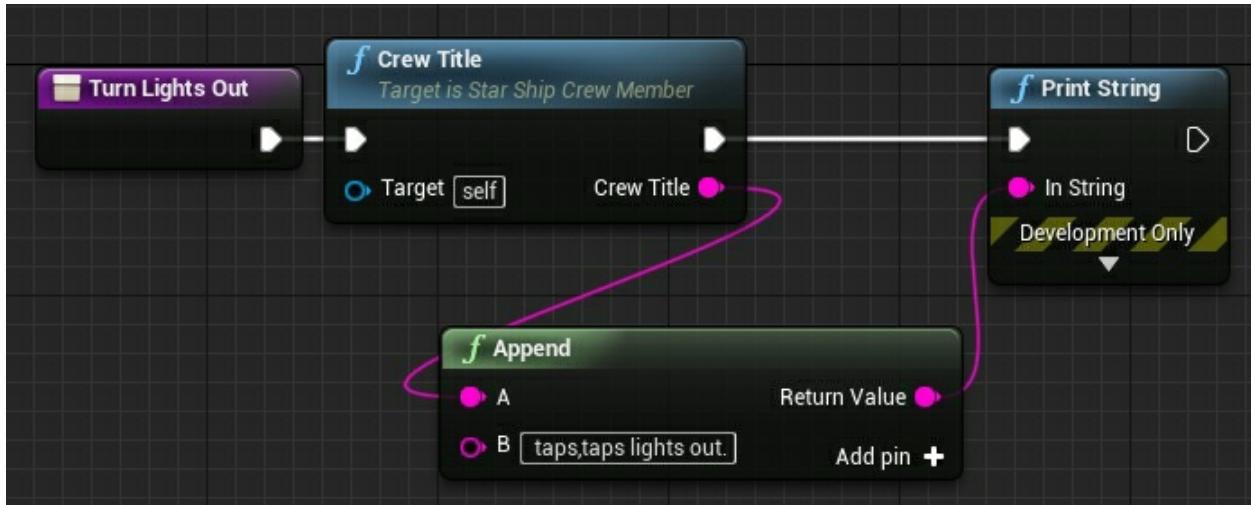
    public void libertyCall() {
        System.out.println(crewTitle() +
            " liberty call, liberty call.");
    }

    public void crewMuster() {
        System.out.println(crewTitle() +
            " time for muster.");
    }

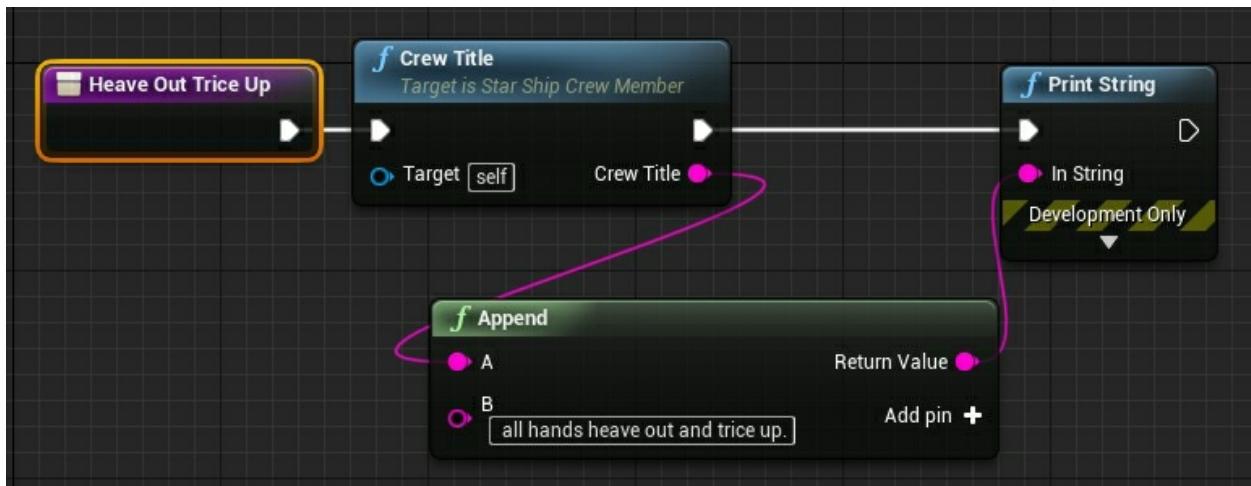
    private void task(String task) {
        switch (task) {
            case "lights_out":
                turnLightsOut();
                break;
            case "trice_up":
                heaveOutTriceUp();
                break;
            case "liberty_call":
                libertyCall();
                break;
            case "muster":
                crewMuster();
                break;
        }
    }
}
```

```
case "duty":  
    duty();  
    break;  
default:  
    System.out.println("Undefined task");  
    break;  
}  
}  
  
public void task(List<String> tasks) {  
    for (String task : tasks) {  
        task(task);  
    }  
}  
  
public abstract void duty();  
  
public abstract String crewTitle();  
}
```

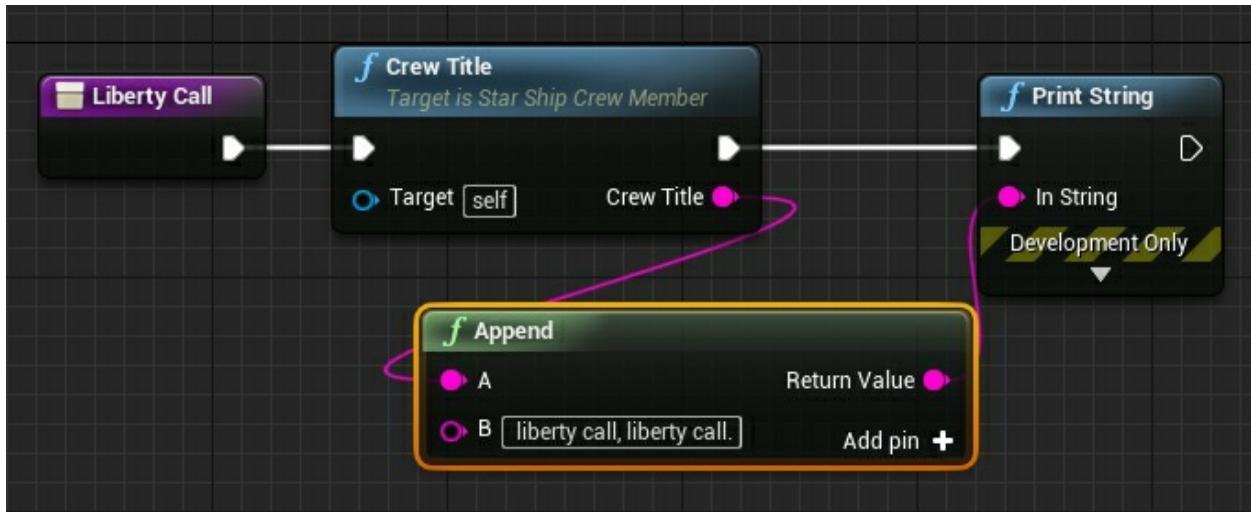
The *Turn Lights Out* blueprint function:



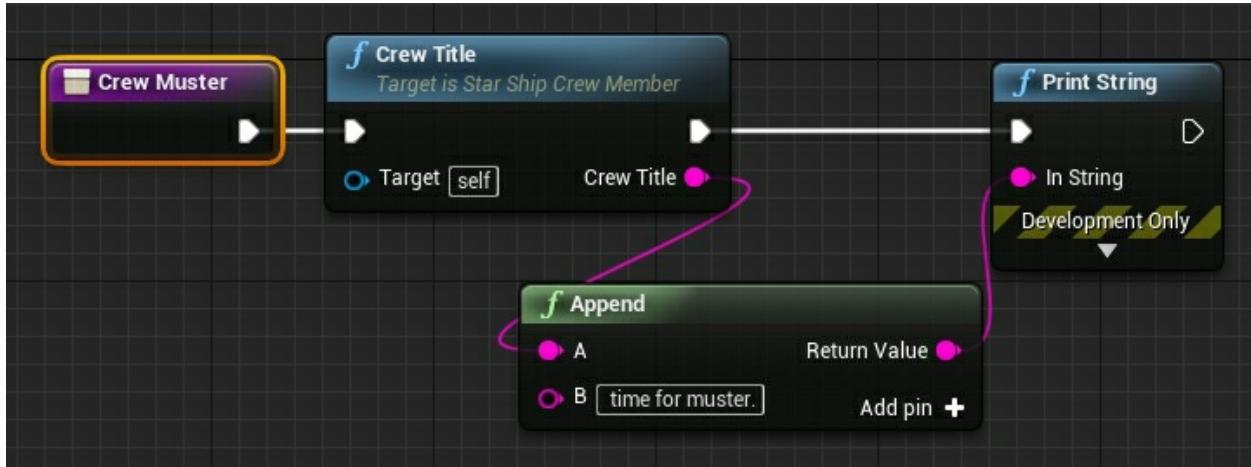
The *Heave Out Trice Up* blueprint function:



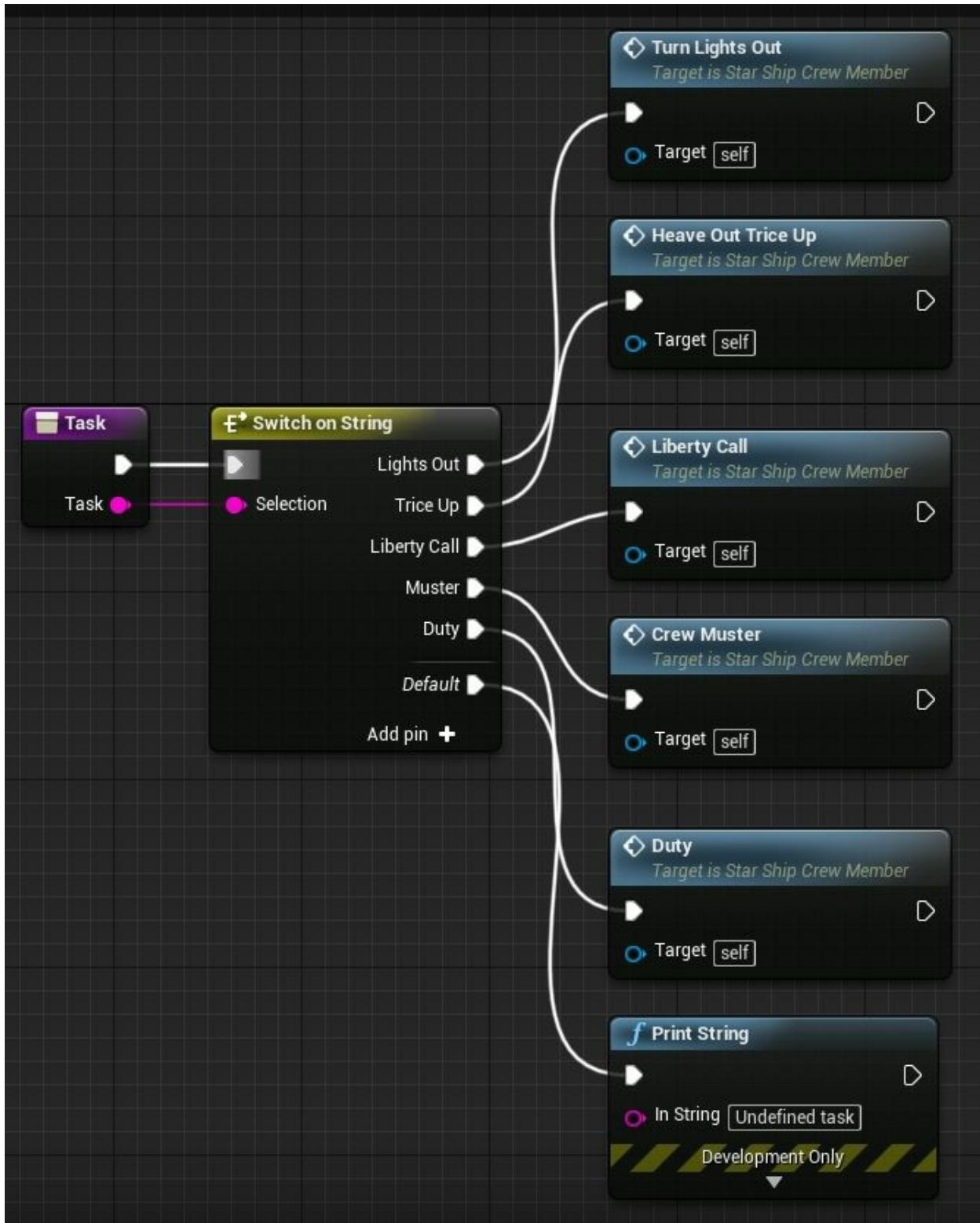
The *Liberty Call* blueprint function:



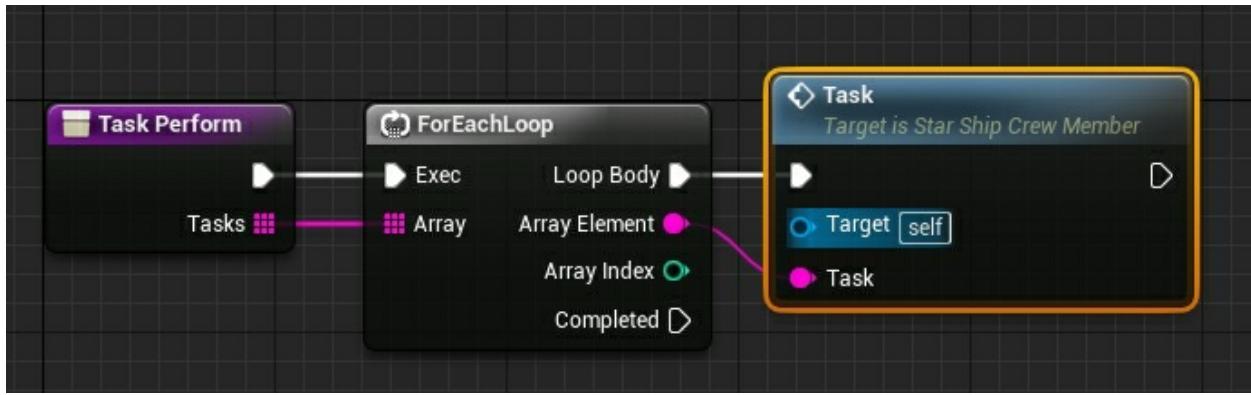
The *Crew Muster* blueprint function:



The *Task* blueprint function:



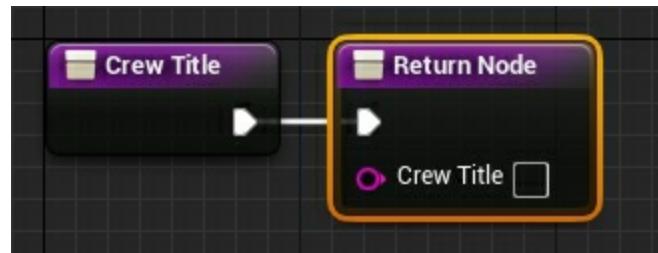
The *Task Perform* blueprint function:



The *Duty* blueprint function:



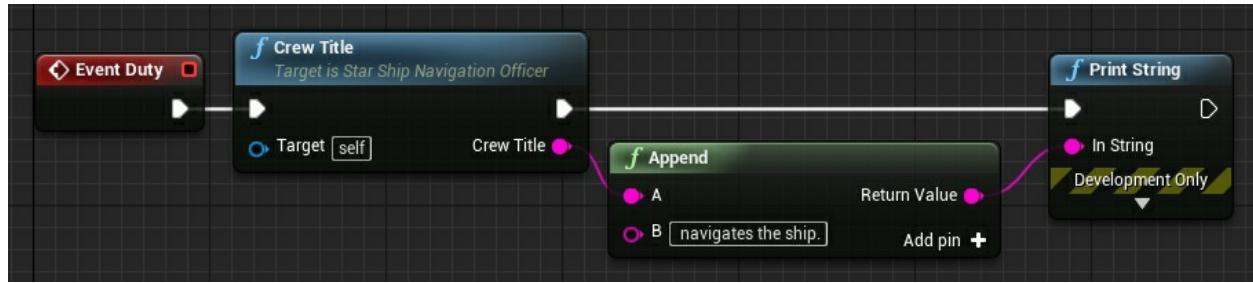
The *Crew Title* blueprint function:



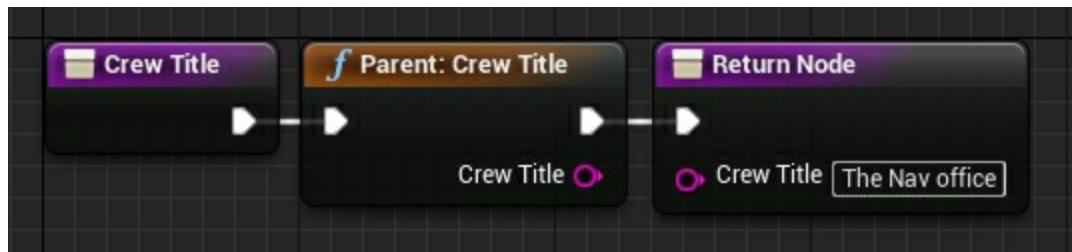
The *StarShipNavigationOfficer* class:

```
public class StarShipNavigationOfficer extends StarShipCrewMember {  
  
    @Override  
    public void duty() {  
        System.out.println(crewTitle() + " navigates the ship.");  
    }  
  
    @Override  
    public String crewTitle() {  
        return "The Nav Officer";  
    }  
}
```

The *StarShipNavigationOfficer* event graph:



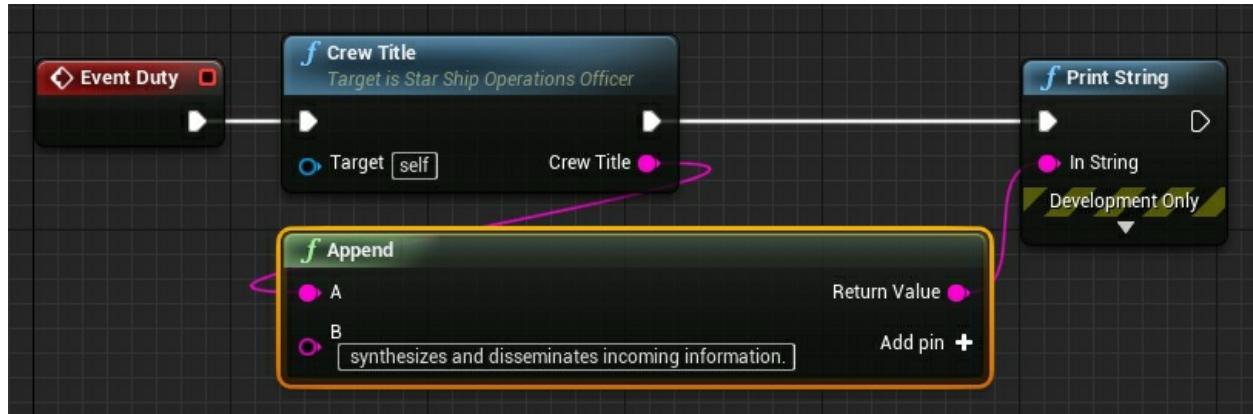
The *StarShipNavigationOfficer* crew title function:



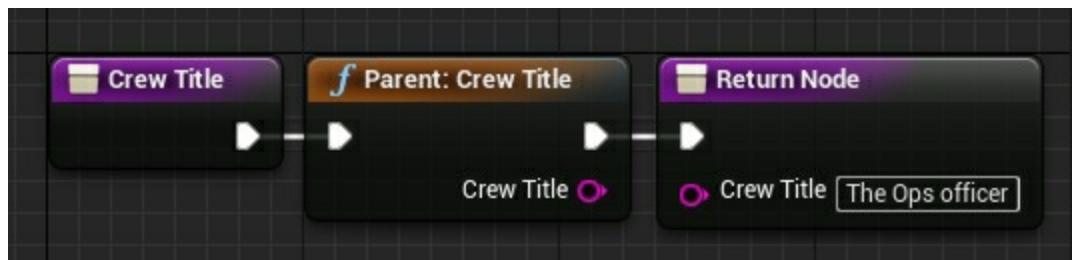
The *StarShipOperationsOfficer* class:

```
public class StarShipOperationsOfficer extends StarShipCrewMember {  
  
    @Override  
    public void duty() {  
        System.out.println(crewTitle() +  
            " synthesizes and disseminates incoming information.");  
    }  
  
    @Override  
    public String crewTitle() {  
        return "The Ops Officer";  
    }  
}
```

The *StarShipOperationsOfficer* event graph:



The *StarShipOperationsOfficer* crew title function:



The *StarShipSupplyOfficer* class:

```
public class StarShipSupplyOfficer extends StarShipCrewMember {  
  
    @Override  
    public void duty() {  
        System.out.println(crewTitle() +  
            " ensures there are ample ship supplies.");  
    }  
  
    @Override  
    public String crewTitle() {  
        return "The Supply Officer";  
    }  
}
```

The *StarShipSupplyOfficer* event graph:



The *StarShipSupplyOfficer* crew title function:



The Façade pattern viewport print:

The printed text is as follows:

```
The Ops officer taps,taps lights out.  
The Ops officer liberty call, liberty call.  
The Supply Officer taps,taps lights out.  
The Supply Officer liberty call, liberty call.  
The Nav office taps,taps lights out.  
The Nav office liberty call, liberty call.  
The Ops officer synthesizes and disseminates incoming information.  
The Supply Officer ensures there are ample ship supplies.  
The Nav office navigates the ship.  
The Ops officer time for muster.  
The Ops officer all hands heave out and trice up.  
The Supply Officer time for muster.  
The Supply Officer all hands heave out and trice up.  
The Nav office time for muster.  
The Nav office all hands heave out and trice up.
```

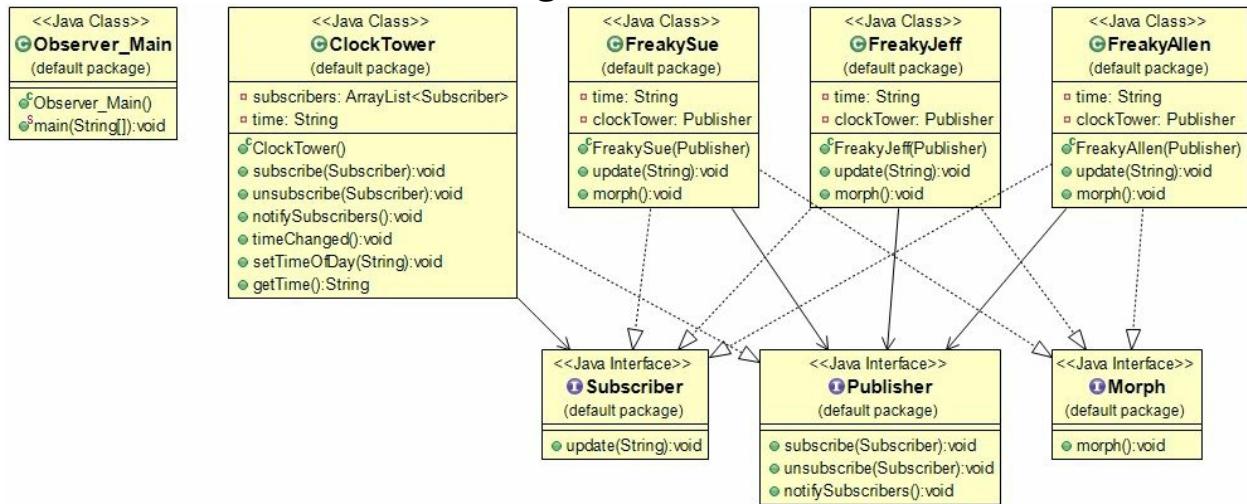
# Behavioral Patterns (Java)

The behavioral patterns we discuss help us determine how objects interact. Additionally, object responsibilities are defined in behavioral patterns as well. We must mention communication in are behavioral pattern overview. Runtime communication between objects is a staple in behavioral pattern implementations.

# The Freaks Come out at night...

## Observer Pattern (Java)

**Observer Pattern UML Diagram**

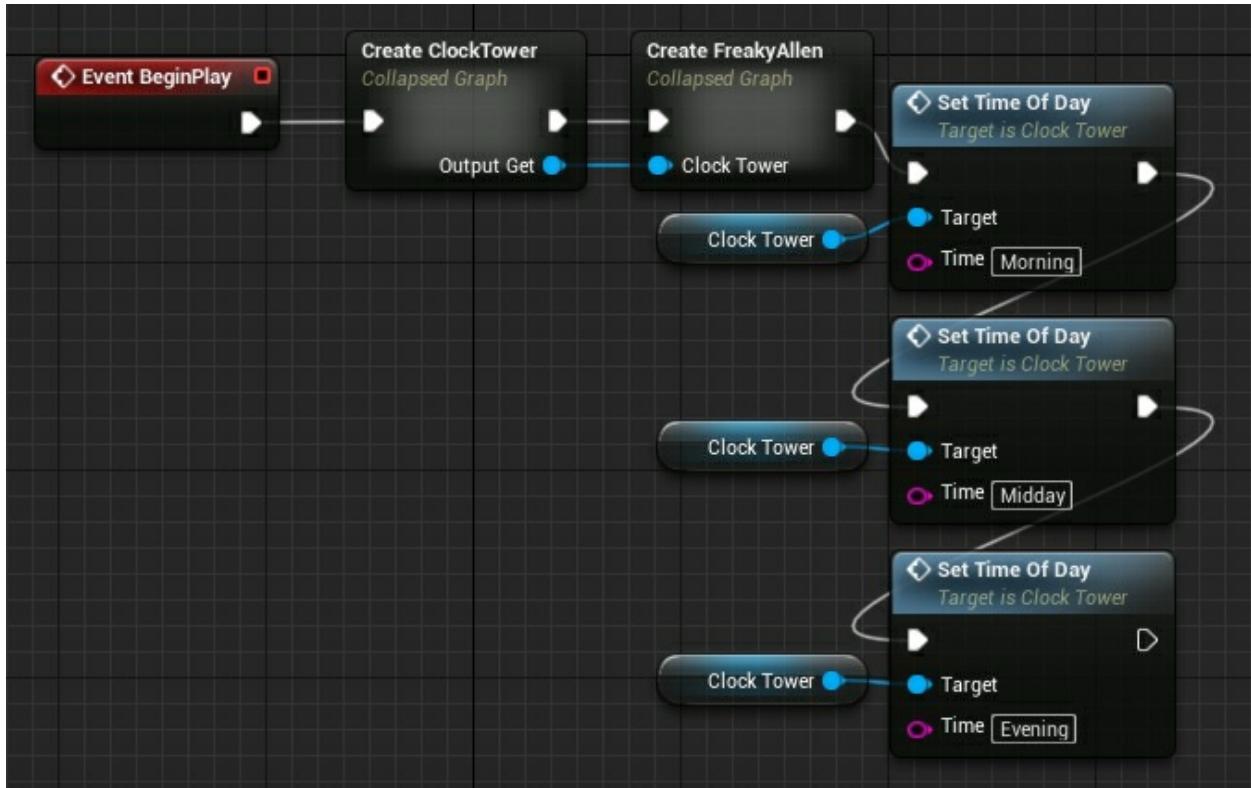


## **Observer Pattern Implementation**

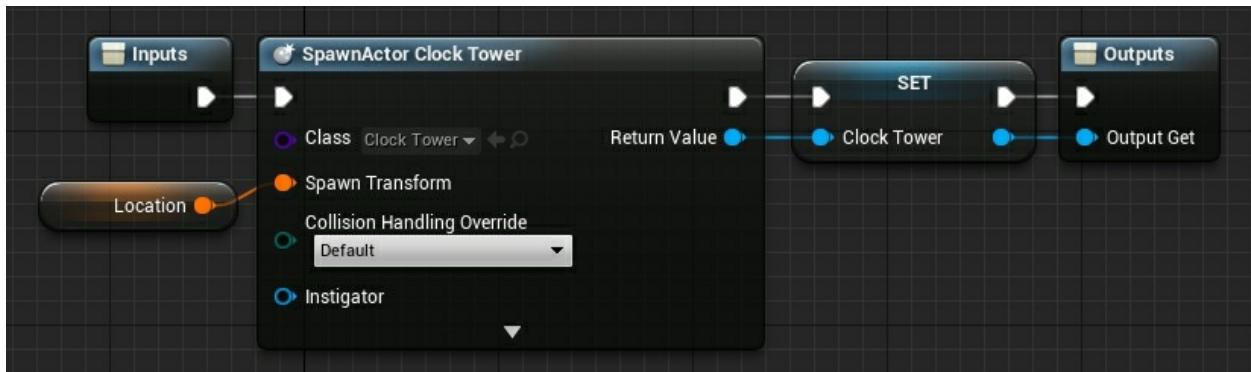
A clock tower is our publisher for the Observer pattern example. The subscribers consist of a group of freaks who change their behavior based on the time of day. The freaks receive notifications from the clock tower when the clock towers time state has changed. The *Observer\_Main* class:

```
public class Observer_Main {  
  
    public static void main(String[] args) {  
        ClockTower clockTower = new ClockTower();  
  
        FreakyAllen freakyAllen = new FreakyAllen(clockTower);  
        FreakySue freakySue = new FreakySue(clockTower);  
        FreakyJeff freakyJeff = new FreakyJeff(clockTower);  
  
        clockTower.setTimeOfDay("Morning");  
        clockTower.setTimeOfDay("Midday");  
        clockTower.setTimeOfDay("Evening");  
    }  
}
```

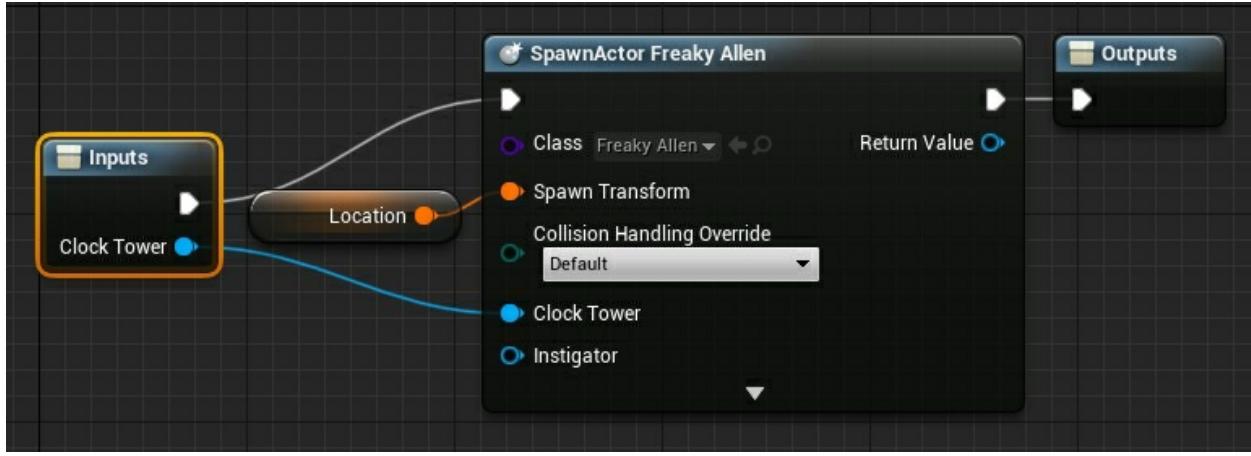
The *Observer\_Main* blueprint event graph:



The *Create ClockTower* collapsed graph:



The *Create FreakyAllen* collapsed graph:



Let's implement the *Subscriber* interface:

```
public interface Subscriber {
    public void update(String time);
}
```

Let's implement the *Publisher* interface:

```
public interface Publisher {
    public void subscribe(Subscriber s);
    public void unsubscribe(Subscriber s);
    public void notifySubscribers();
}
```

The *ClockTower* class:

```
import java.util.ArrayList;

public class ClockTower implements Publisher {
    private ArrayList<Subscriber> subscribers;
    private String time;

    public ClockTower() {
        subscribers = new ArrayList<Subscriber>();
    }

    public void subscribe(Subscriber s) {
        subscribers.add(s);
    }

    public void unsubscribe(Subscriber s) {
        int i = subscribers.indexOf(s);
        if (i >= 0) {
            subscribers.remove(i);
        }
    }

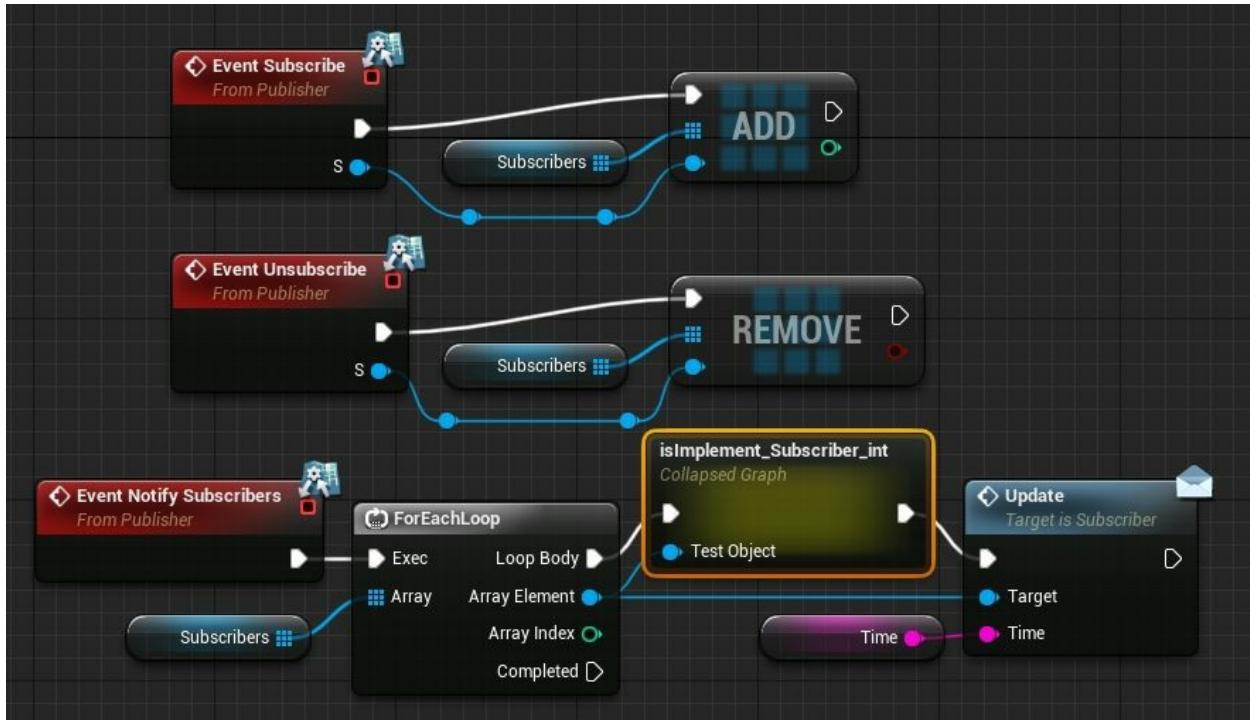
    public void notifySubscribers() {
        for (Subscriber subscriber : subscribers) {
            subscriber.update(time);
        }
    }

    public void timeChanged() {
        notifySubscribers();
    }

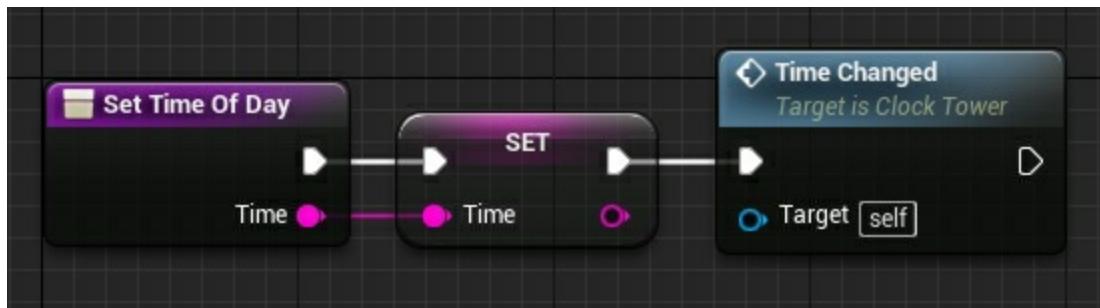
    public void setTimeOfDay(String time) {
        this.time = time;
        timeChanged();
    }
}
```

```
public String getTime() {  
    return time;  
}  
  
}
```

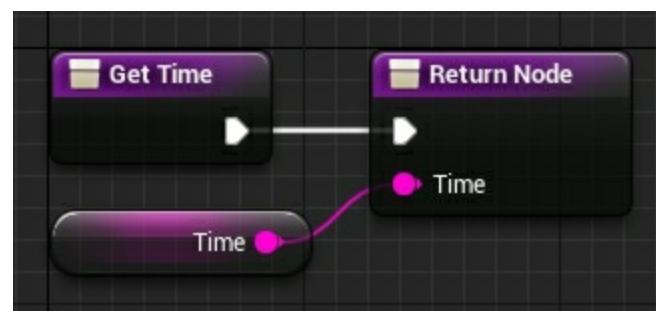
The *ClockTower* blueprint event graph:



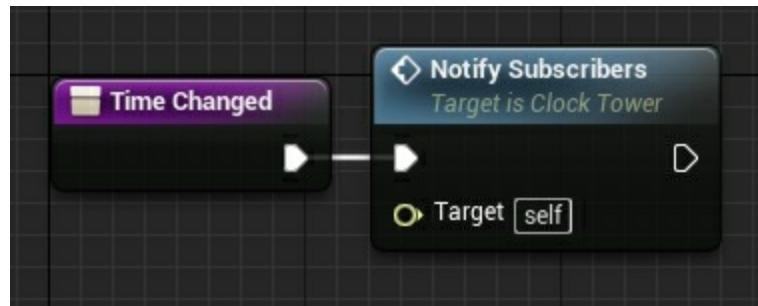
The *setTimeOfDay* function:



The *getTime* function:



The *timeChanged* function:



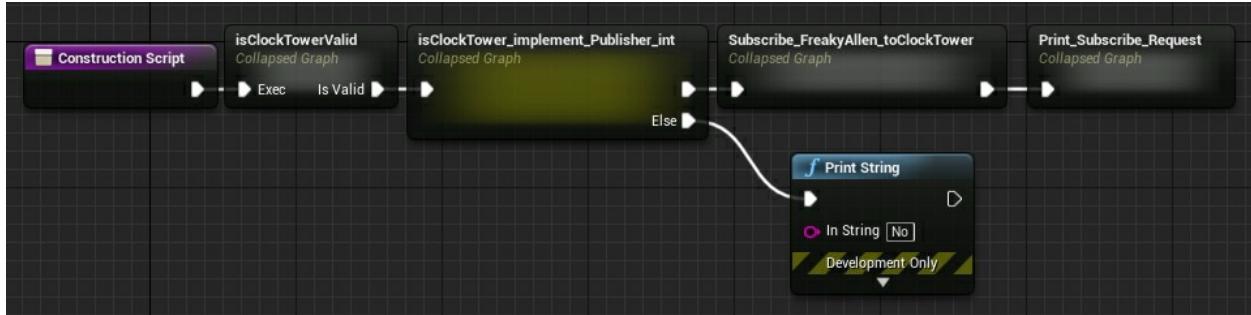
Let's implement the *Morph* interface our freaks will all share:

```
public interface Morph {  
    public void morph();  
}
```

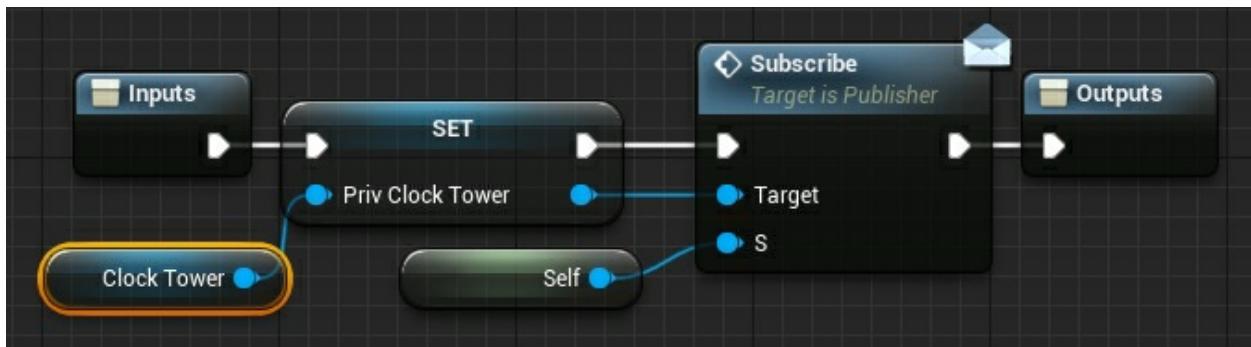
The *FreakyAllen* class:

```
public class FreakyAllen implements Subscriber, Morph {  
    private String time;  
    private Publisher clockTower;  
  
    public FreakyAllen(Publisher clockTower) {  
        this.clockTower = clockTower;  
        clockTower.subscribe(this);  
    }  
  
    public void update(String time) {  
        this.time = time;  
        morph();  
    }  
  
    public void morph() {  
        if (time.equals("Morning")) {  
            System.out.println("It is " + time  
                + ", so FreakyAllen makes a bowl of cereal");  
        } else if (time.equals("Midday")) {  
            System.out.println("It is: " + time  
                + ", so FreakyAllen's right eye starts to twitch");  
        } else if (time.equals("Evening")) {  
            System.out.println("It is: " + time  
                + ", so FreakyAllen morphs into a blood sucking wogglesnort");  
        }  
    }  
}
```

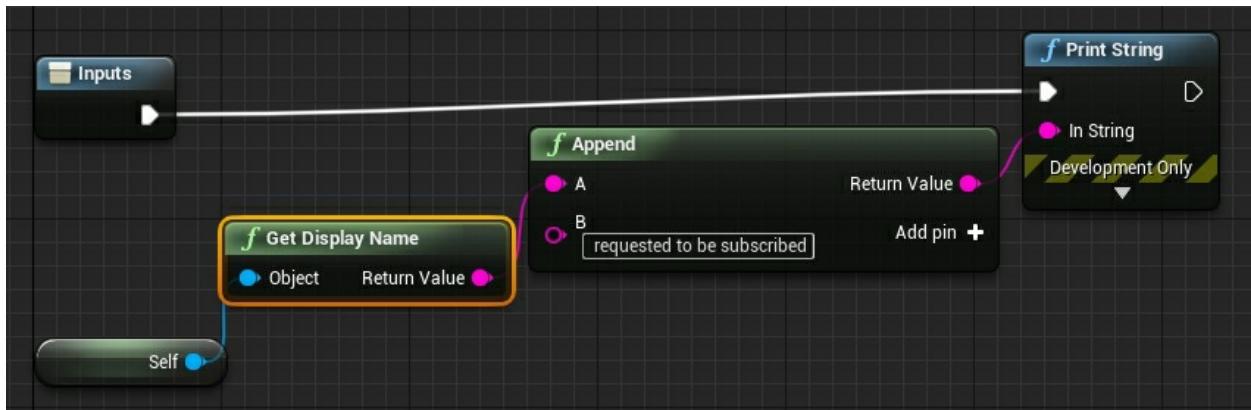
The *FreakyAllen* construction script:



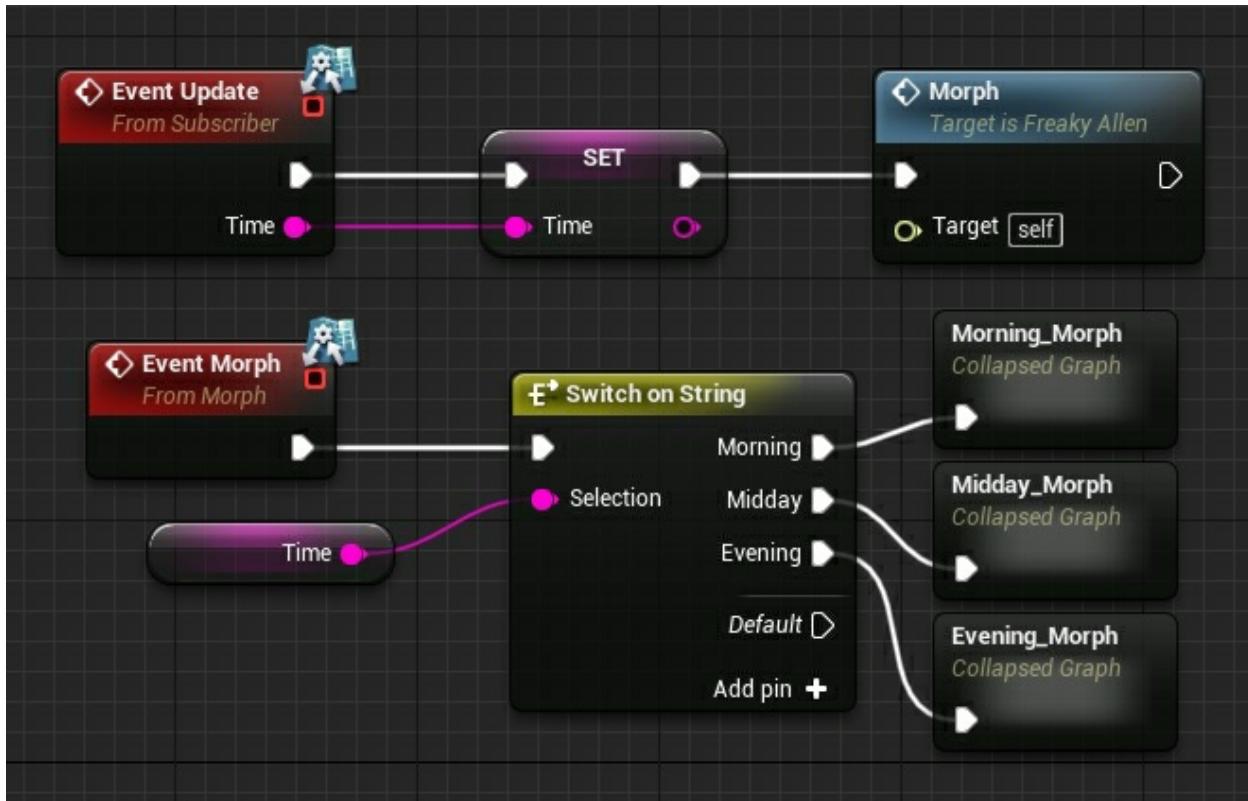
The *Subscribe\_FreakyAllen\_toClockTower* collapsed graph:



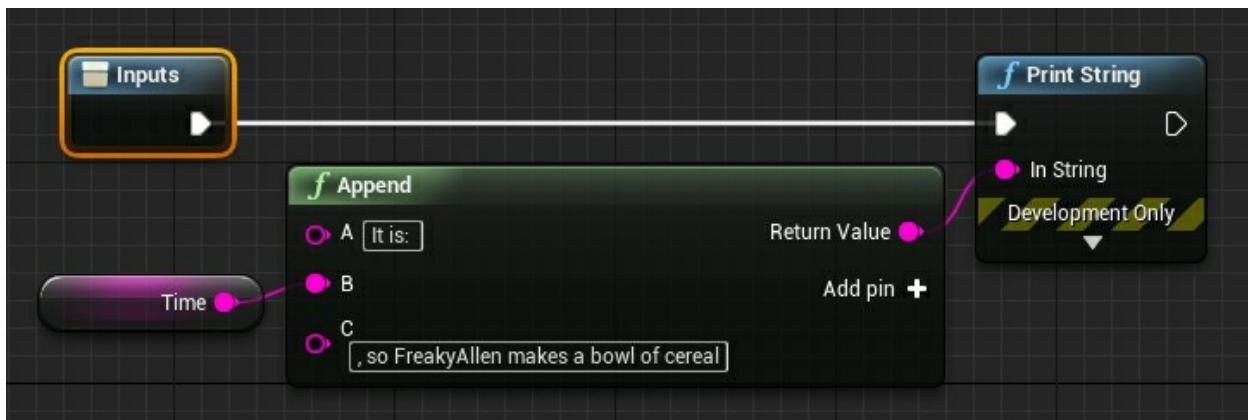
The *Print\_Subscribe\_Request* collapsed graph:



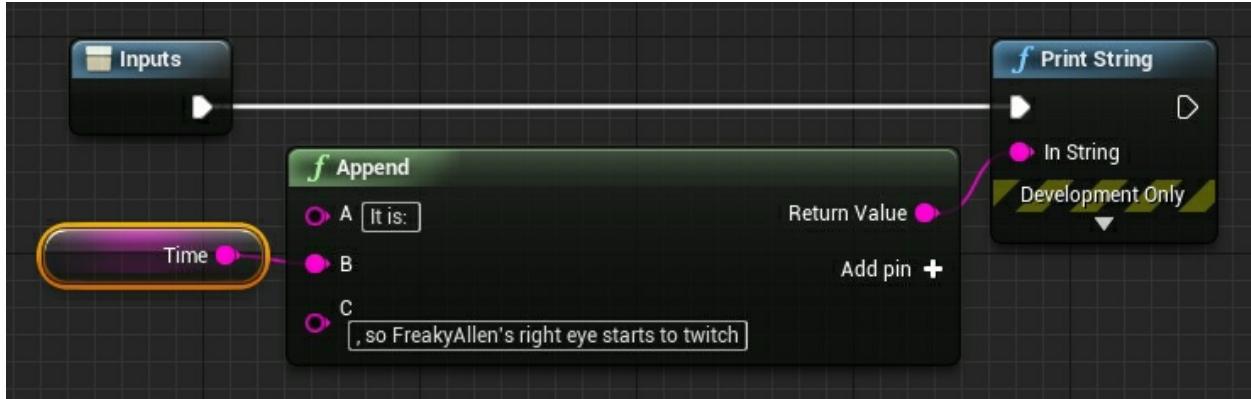
The *FreakyAllen* event graph:



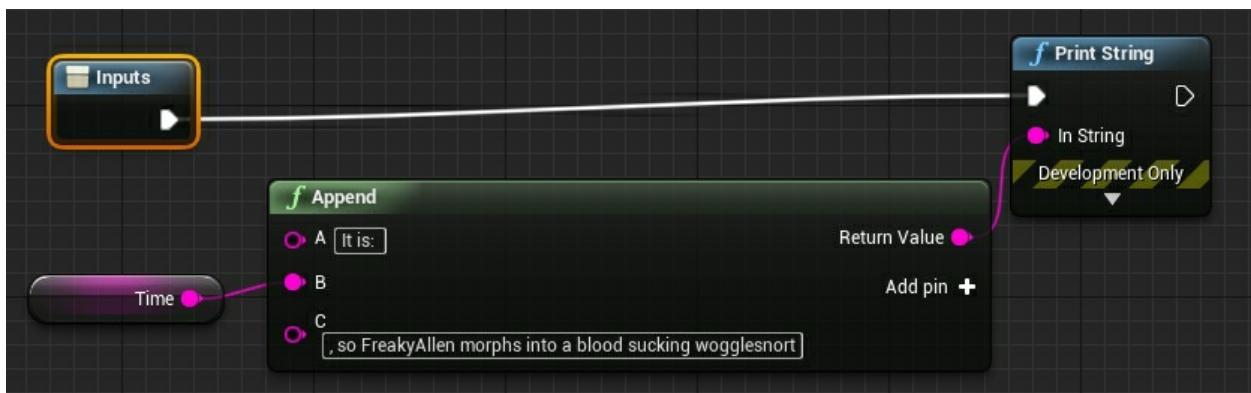
The *Morning\_Morph* collapsed graph:



The *Midday\_Morph* collapsed graph:

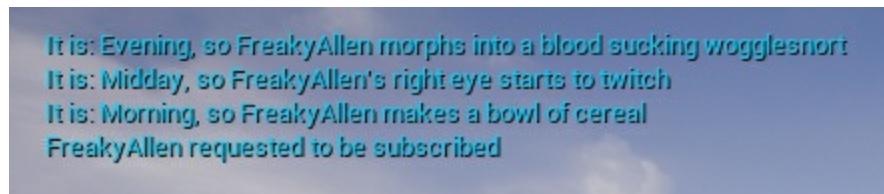


The *Evening\_Morph* collapsed graph:



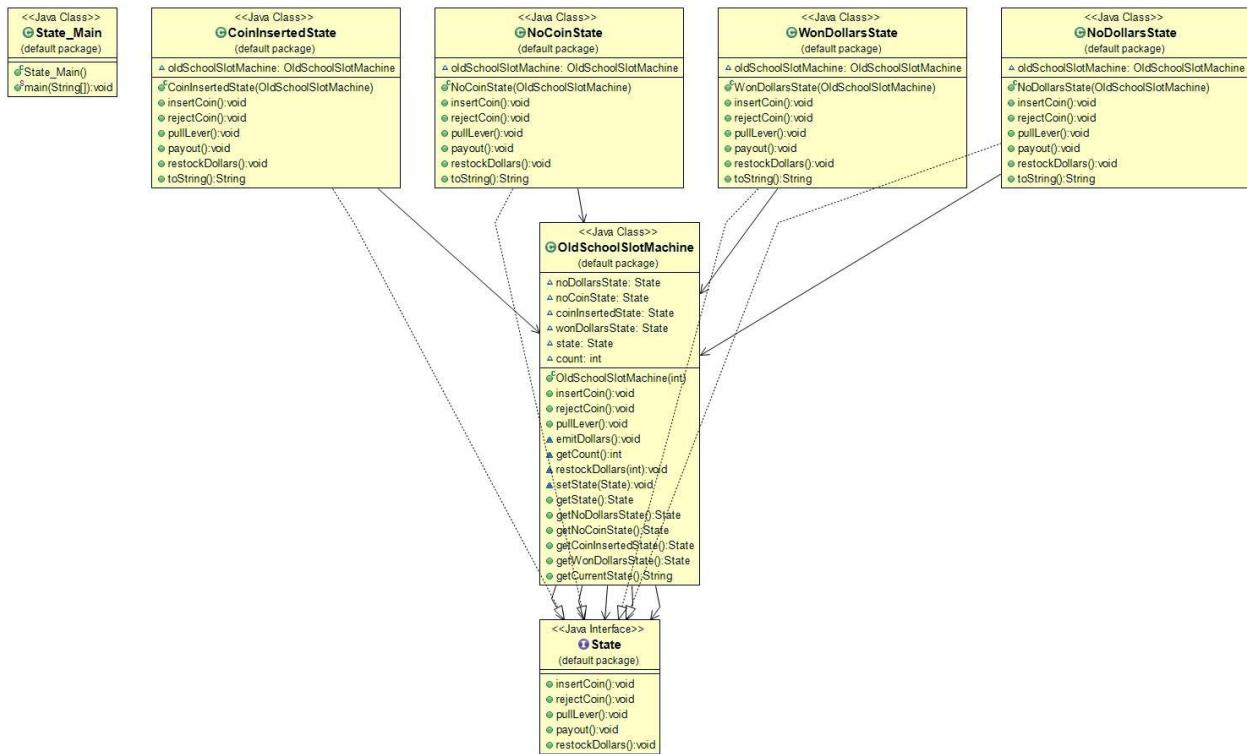
The other freaks follow the same implementation with the exception of the strings that define their individual states and thus removed for brevity.

The Observer pattern viewport print:



# Make It Rain... State Pattern (Java)

**State Pattern UML Diagram**

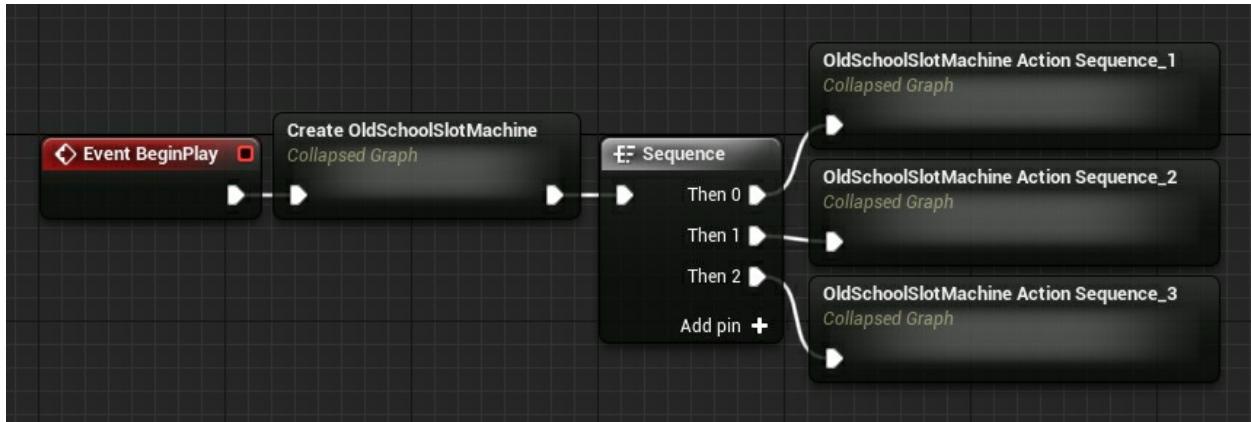


## **State Pattern Implementation**

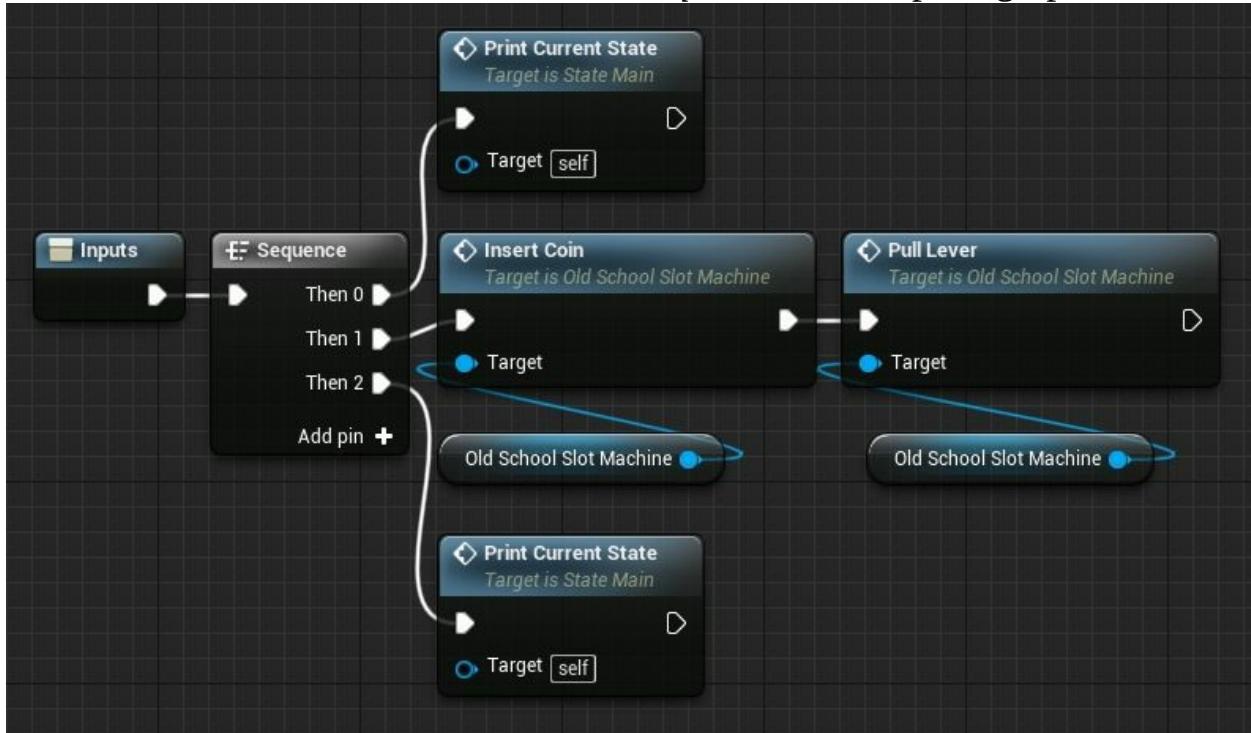
We are going to implement an old school slot machine. As we know, there are a few states the slot machine can assume. These states are: no coin inserted, coin inserted, no dollars, won dollars. The no dollars state indicates the lack of funds inside of a slot machine. The won dollars state indicate a player winning. The *State\_Main* class:

```
public class State_Main {  
    public static void main(String[] args) {  
        OldSchoolSlotMachine oldSchoolSlotMachine = new  
        OldSchoolSlotMachine(100);  
  
        System.out.println(oldSchoolSlotMachine.getCurrentState());  
  
        oldSchoolSlotMachine.insertCoin();  
        oldSchoolSlotMachine.pullLever();  
  
        System.out.println(oldSchoolSlotMachine.getCurrentState());  
  
        oldSchoolSlotMachine.insertCoin();  
        oldSchoolSlotMachine.pullLever();  
        oldSchoolSlotMachine.insertCoin();  
  
        System.out.println(oldSchoolSlotMachine.getCurrentState());  
  
        oldSchoolSlotMachine.pullLever();  
  
        oldSchoolSlotMachine.restockDollars(100);  
        oldSchoolSlotMachine.insertCoin();  
        oldSchoolSlotMachine.pullLever();  
  
        System.out.println(oldSchoolSlotMachine.getCurrentState());  
    }  
}
```

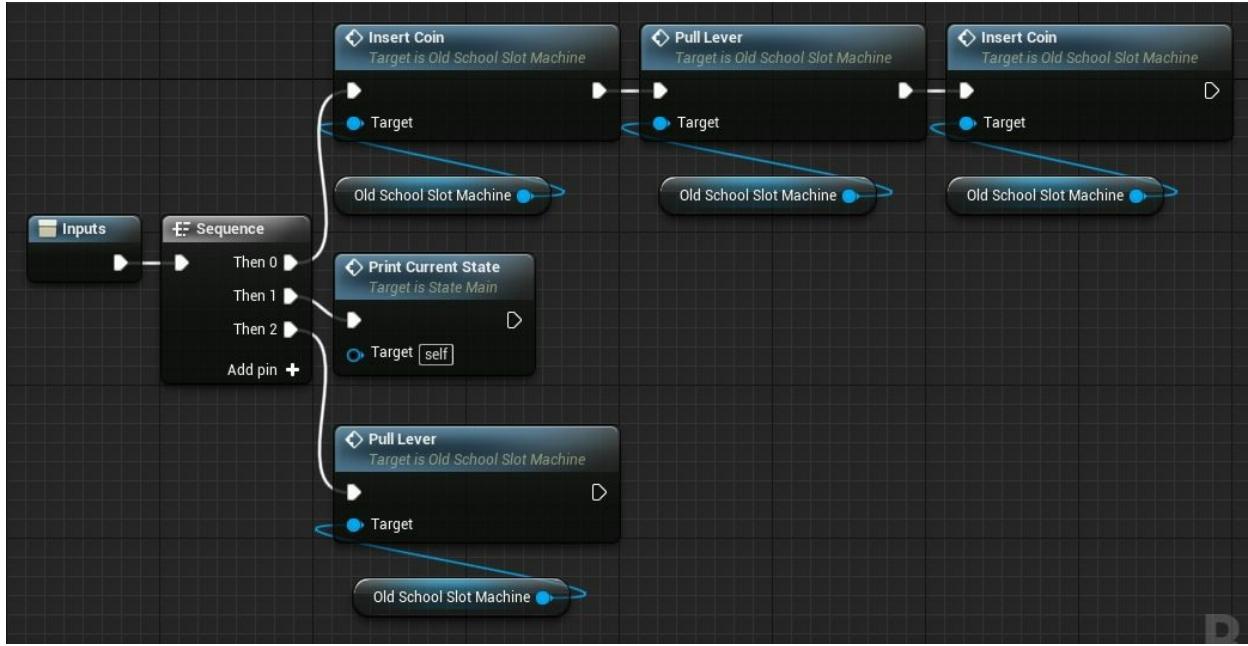
The *State\_Main* blueprint event graph:



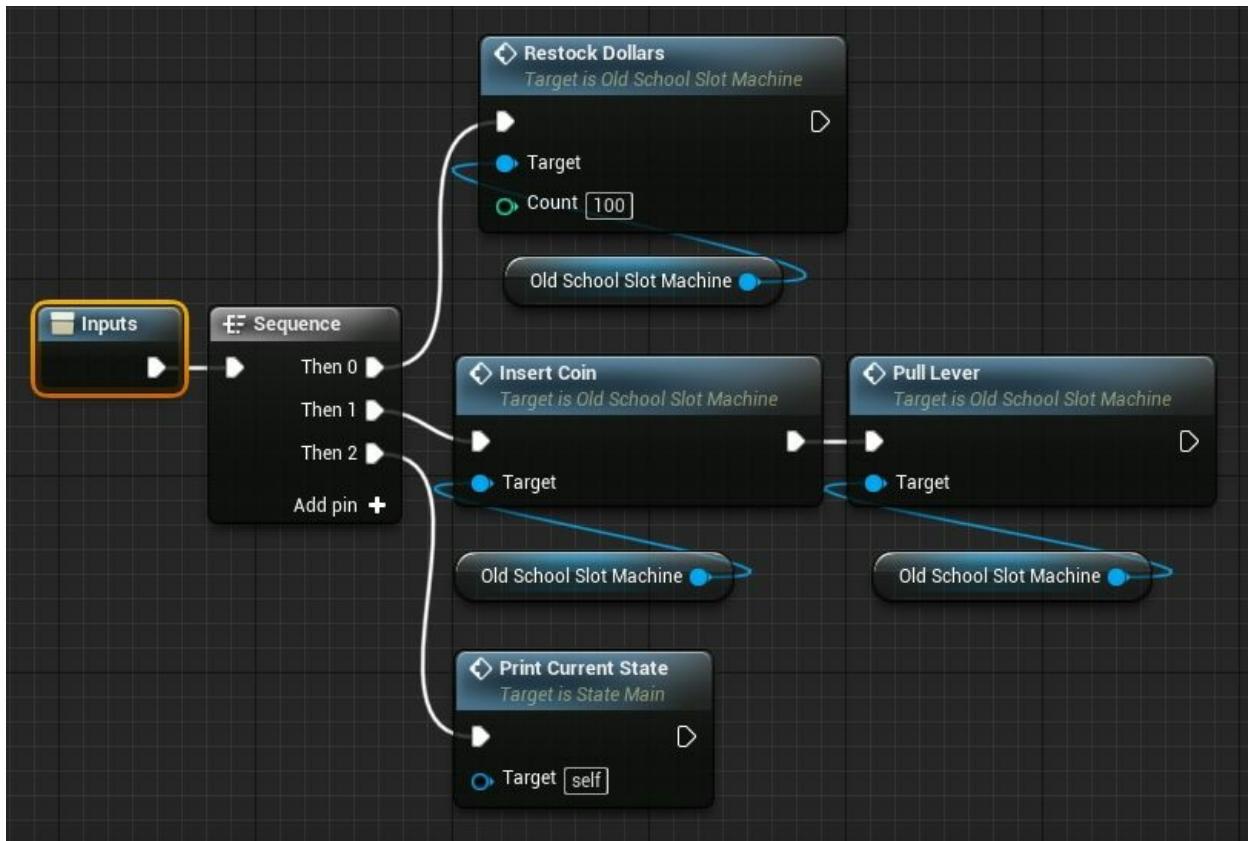
The *OldSchoolSlotMachine Action Sequence\_1* collapsed graph:



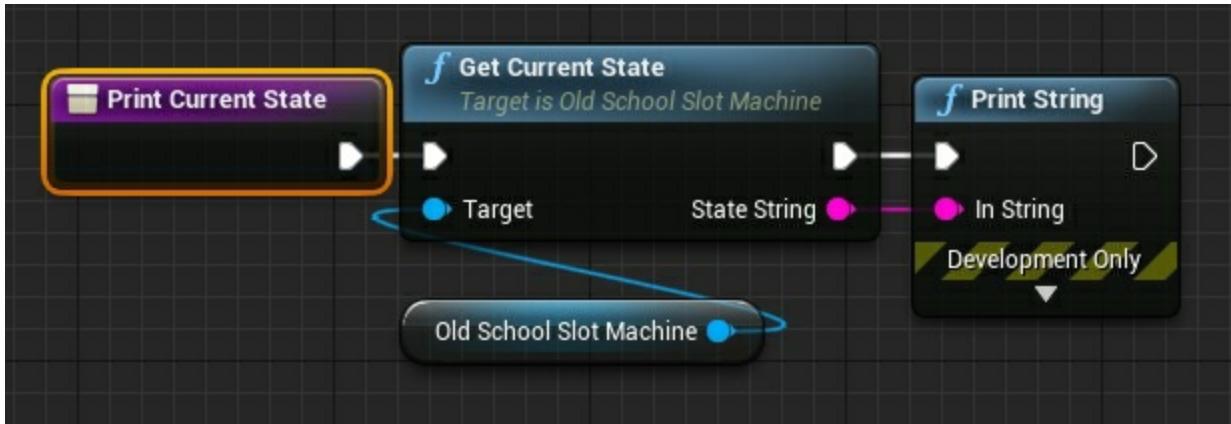
The *OldSchoolSlotMachine Action Sequence\_2* collapsed graph:



The *OldSchoolSlotMachine Action Sequence\_3* collapsed graph:



The *Print Current State* function:



The *OldSchoolSlotMachine* class:

```
public class OldSchoolSlotMachine {

    State noDollarsState;
    State noCoinState;
    State coinInsertedState;
    State wonDollarsState;

    State state;
    int count = 0;

    public OldSchoolSlotMachine(int numberOfDollars) {
        noDollarsState = new NoDollarsState(this);
        noCoinState = new NoCoinState(this);
        coinInsertedState = new CoinInsertedState(this);
        wonDollarsState = new WonDollarsState(this);

        this.count = numberOfDollars;
        if (numberOfDollars > 0) {
            state = noCoinState;
        } else {
            state = noDollarsState;
        }
    }

    public void insertCoin() {
```

```
        state.insertCoin();
    }

    public void rejectCoin() {
        state.rejectCoin();
    }

    public void pullLever() {
        state.pullLever();
        state.payout();
    }

    void emitDollars() {
        System.out.println("Make It Rain!");
        if (count != 0) {
            count = count - 50;
        }
    }

    int getCount() {
        return count;
    }

    void restockDollars(int count) {
        this.count += count;
        System.out.println(
            "The Old School Slot Machine was just filled and new dollar count is: "
            + this.count);
        state.restockDollars();
    }

    void setState(State state) {
        this.state = state;
    }

    public State getState() {
        return state;
    }
```

```
public State getNoDollarsState() {
    return noDollarsState;
}

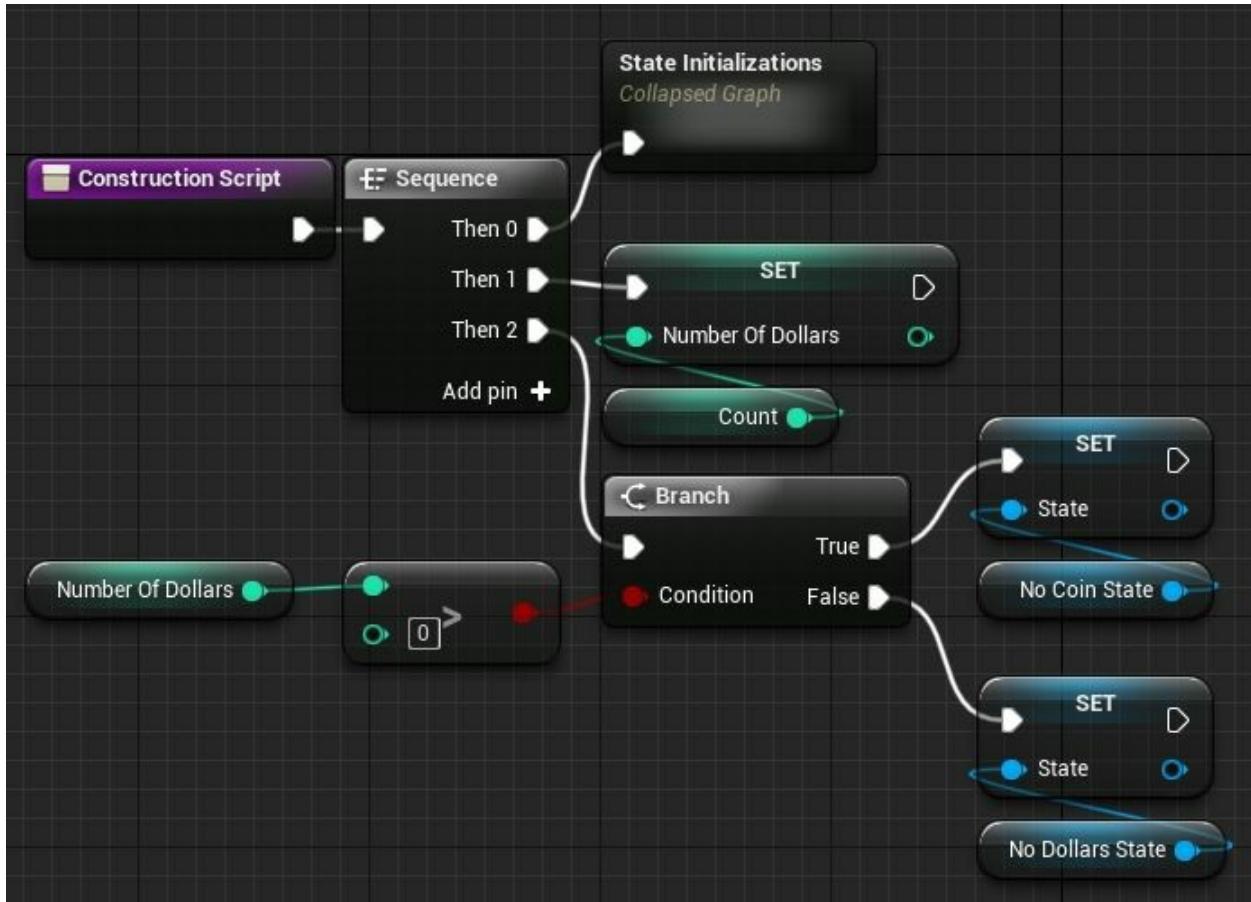
public State getNoCoinState() {
    return noCoinState;
}

public State getCoinInsertedState() {
    return coinInsertedState;
}

public State getWonDollarsState() {
    return wonDollarsState;
}

public String getCurrentState() {
    return "Current Slot Machine State: "
        + state + "\n";
}
```

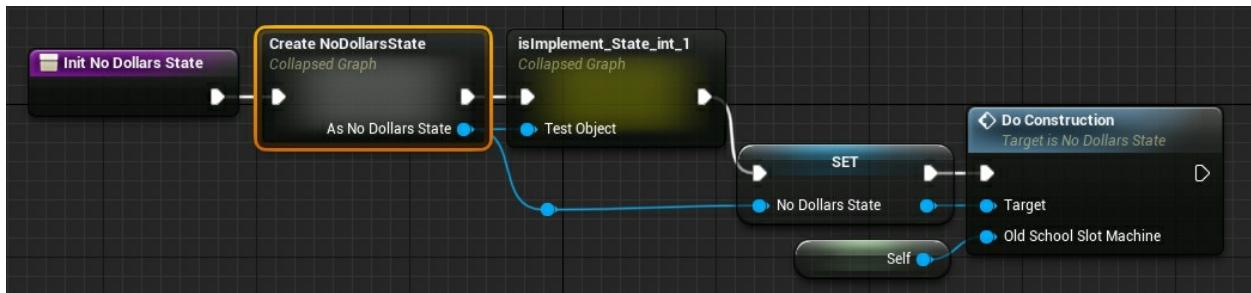
The *OldSchoolSlotMachine* construction script:



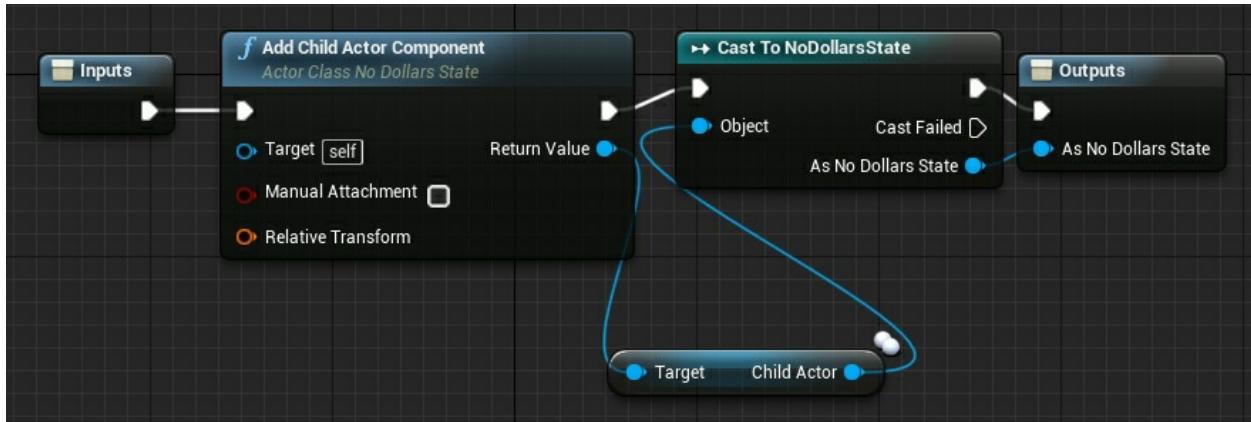
The *State Initializations* collapsed graph:



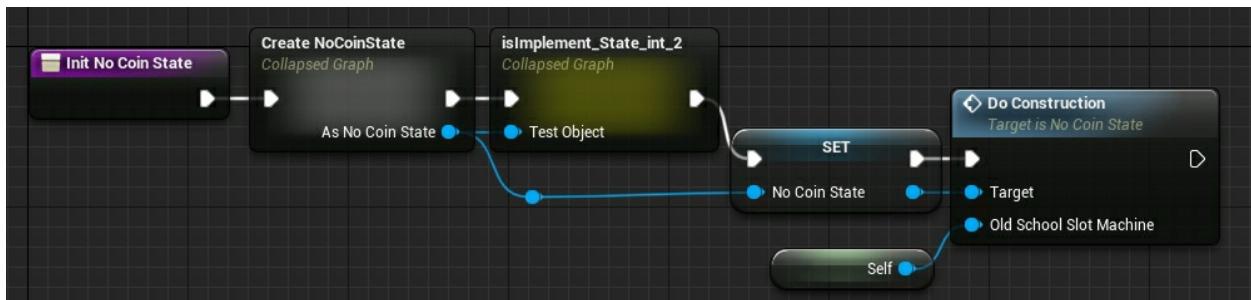
The *initNoDollarsState* function:



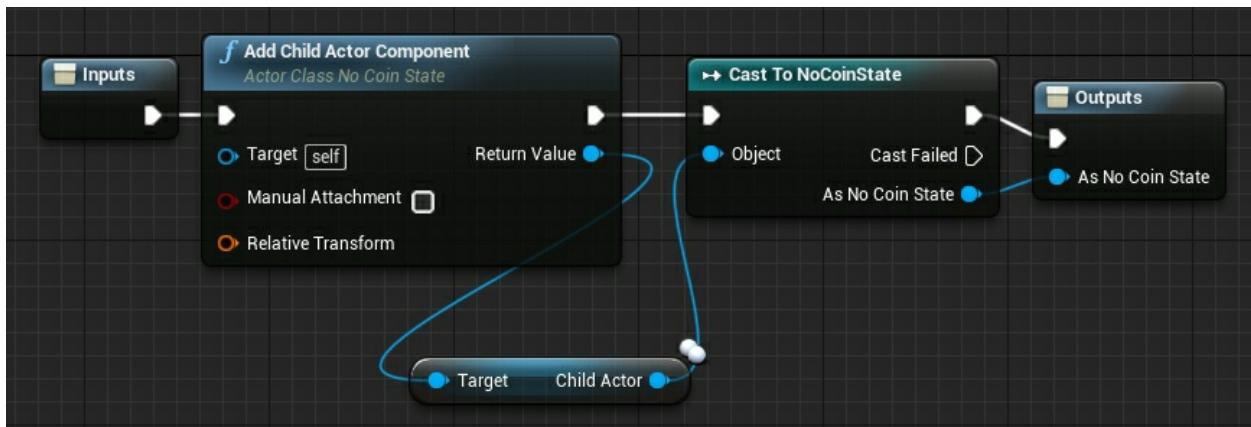
The *Create NoDollarsState* collapsed graph:



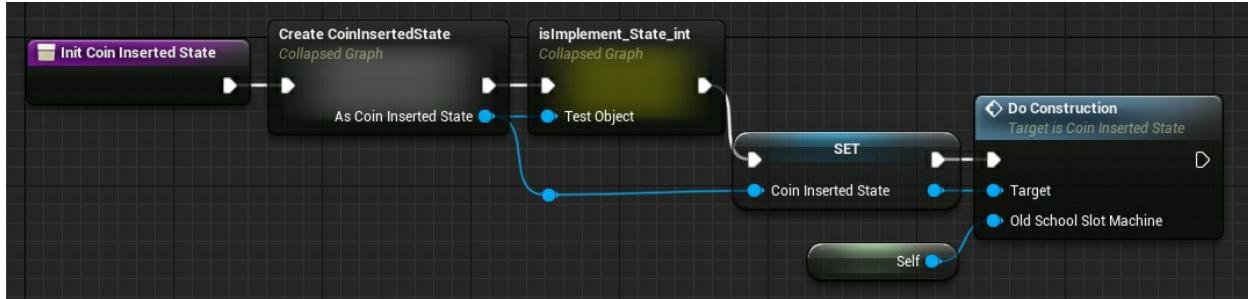
The *initNoCoinState* function:



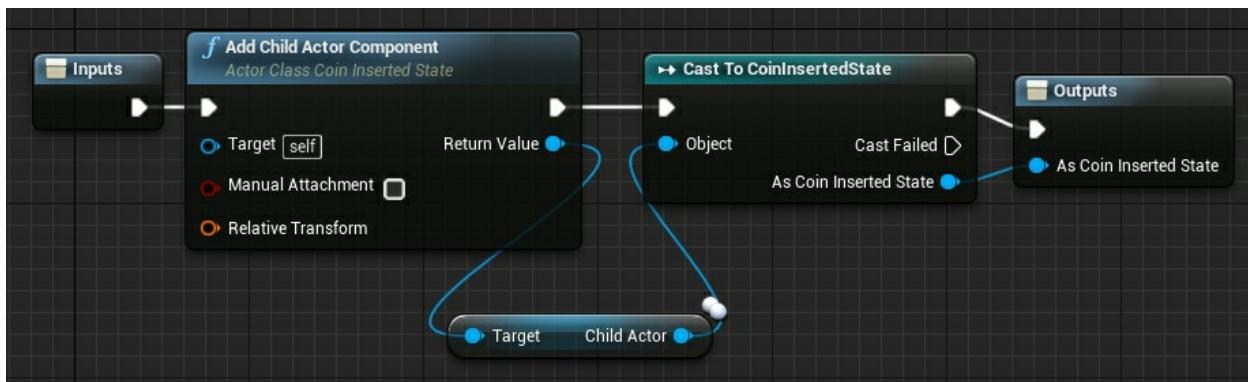
The *Create NoCoinState* collapsed graph:



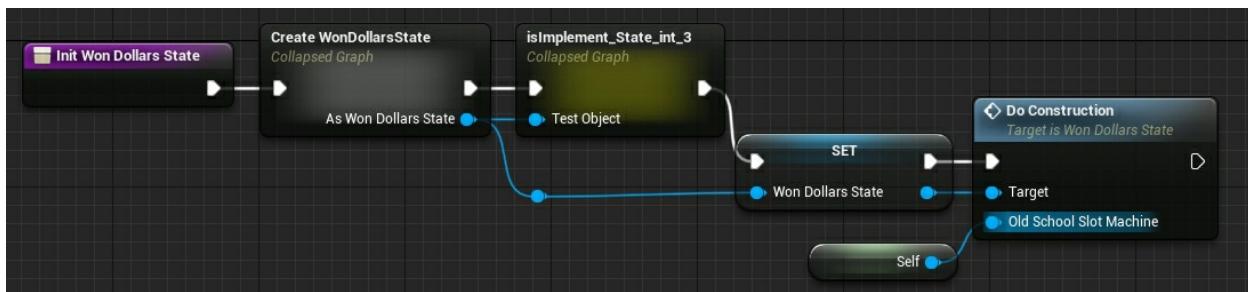
The *initCoinInsertedState* function:



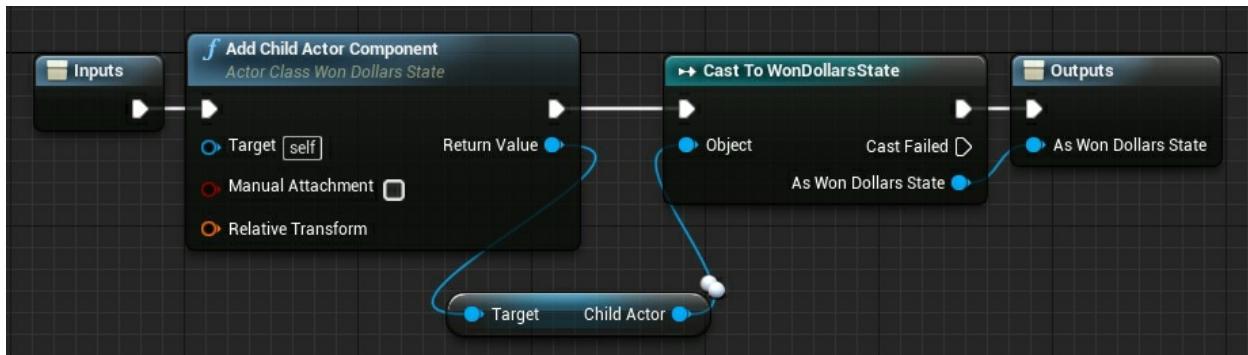
The *Create CoinInsertedState* collapsed graph:



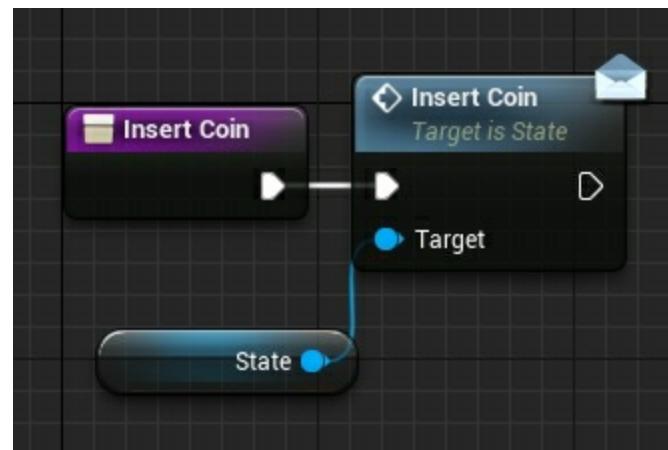
The *initWonDollarsState* function:



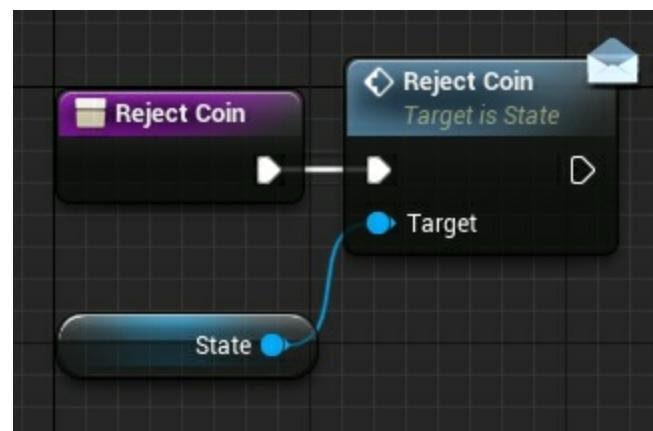
The *Create WonDollarsState* collapsed graph:



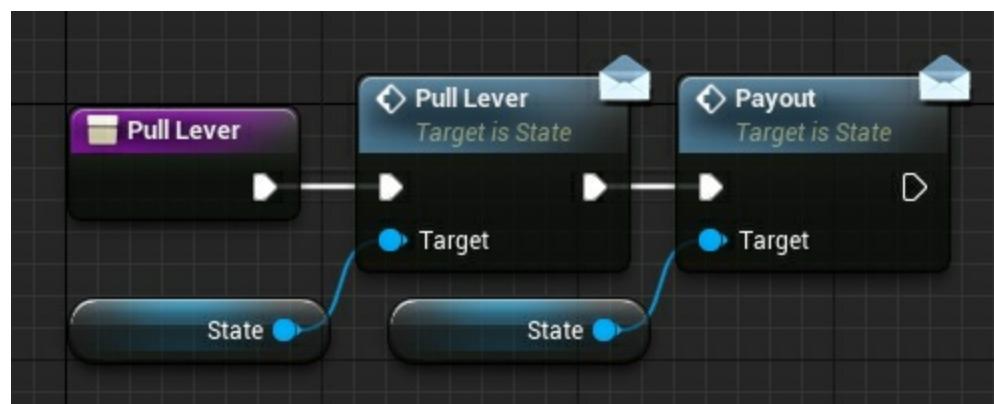
The *insertCoin* function:



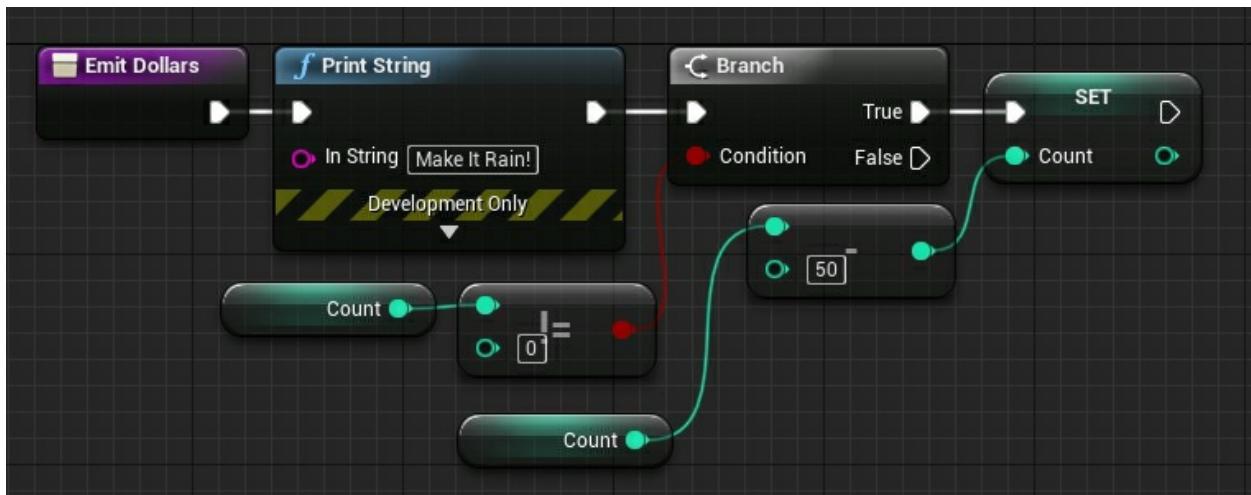
The *rejectCoin* function:



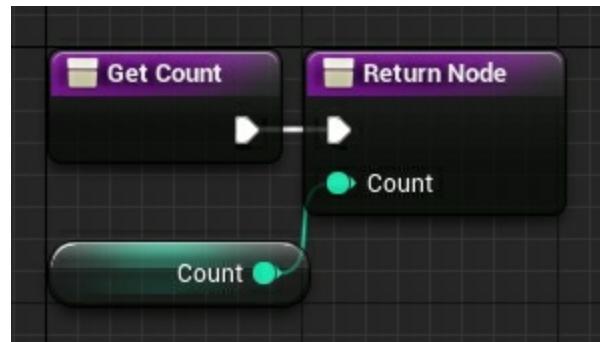
The *pullLever* function:



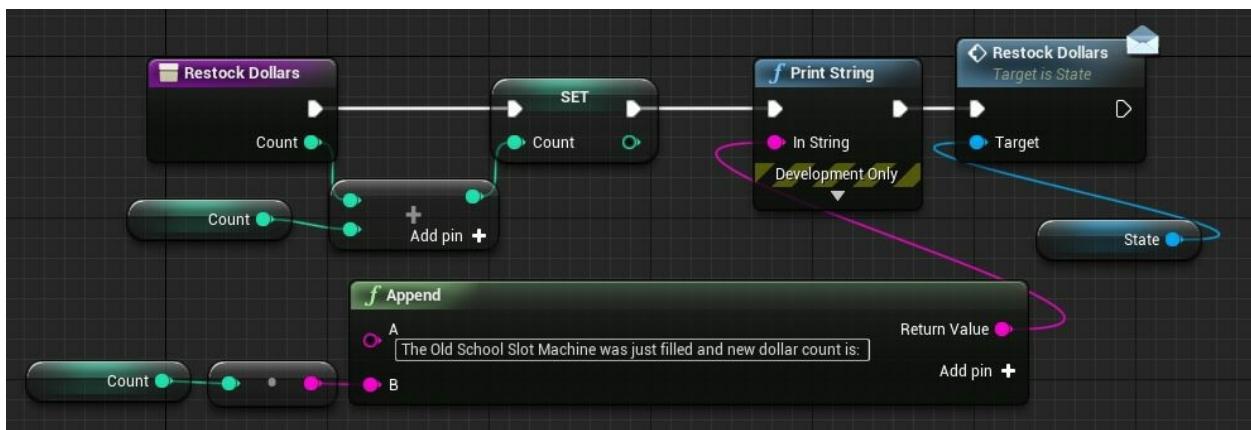
The *emitDollars* function:



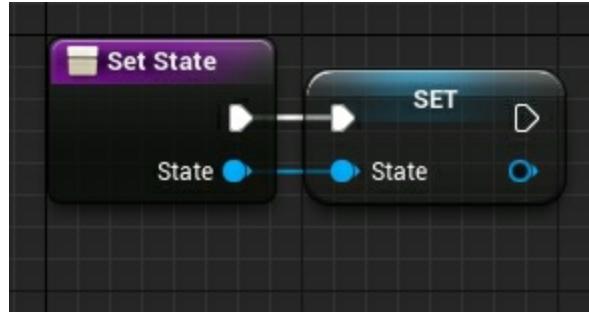
The *getCount* function:



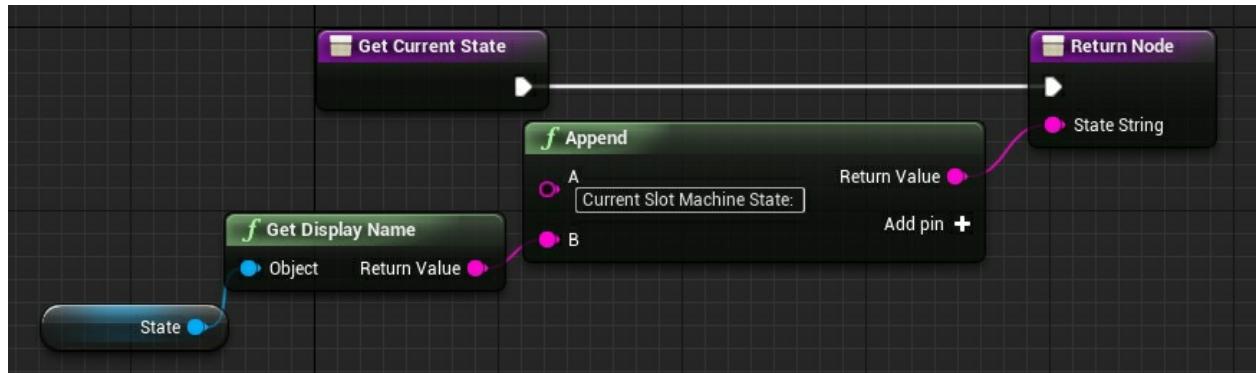
The *restockDollars* function:



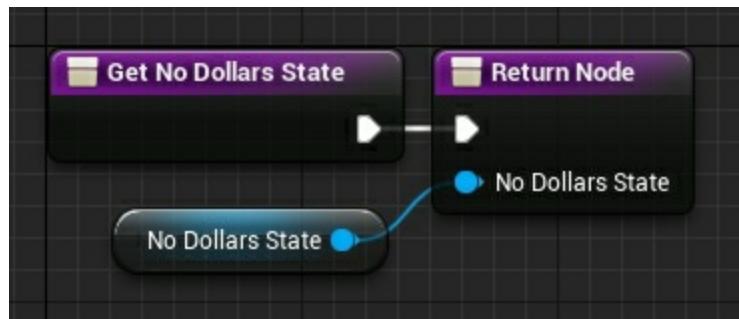
The *setState* function:



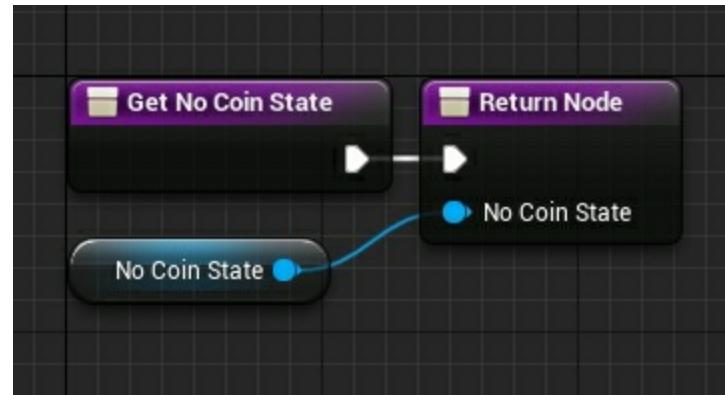
The *getCurrentState* function:



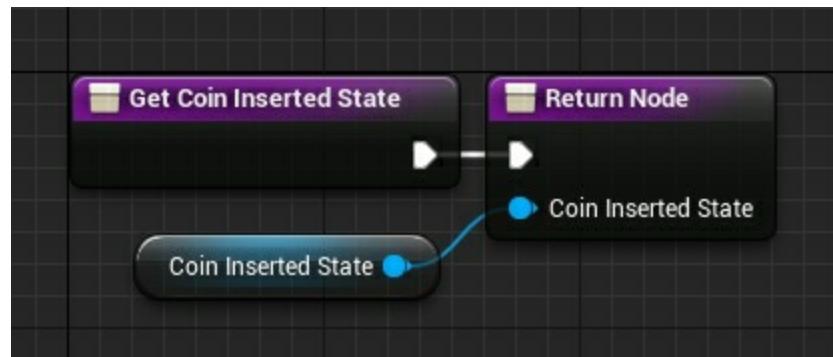
The *getNoDollarsState* function:



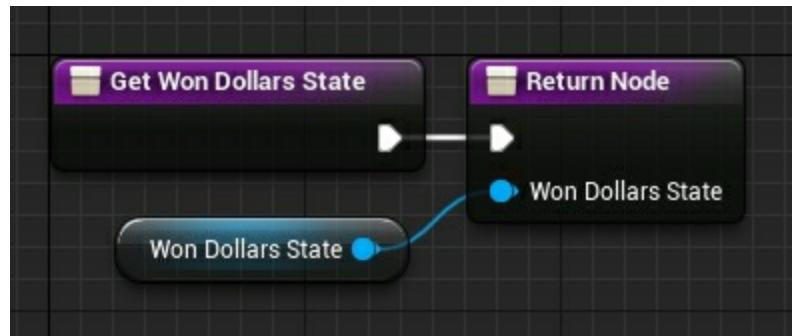
The *getNoCoinState* function:



The *getCoinInsertedState* function:



The *getWonDollarsState* function:



The *NoDollarsState* class:

```
public class NoDollarsState implements State {
    OldSchoolSlotMachine oldSchoolSlotMachine;

    public NoDollarsState(OldSchoolSlotMachine oldSchoolSlotMachine) {
        this.oldSchoolSlotMachine = oldSchoolSlotMachine;
    }

    public void insertCoin() {
        System.out.println(
            "You cannot insert a casino coin, the machine is out of money");
    }

    public void rejectCoin() {
        System.out.println("You have not inserted a casino coin");
    }

    public void pullLever() {
        System.out.println(
            "You turned, but there is NO money in the machine");
    }

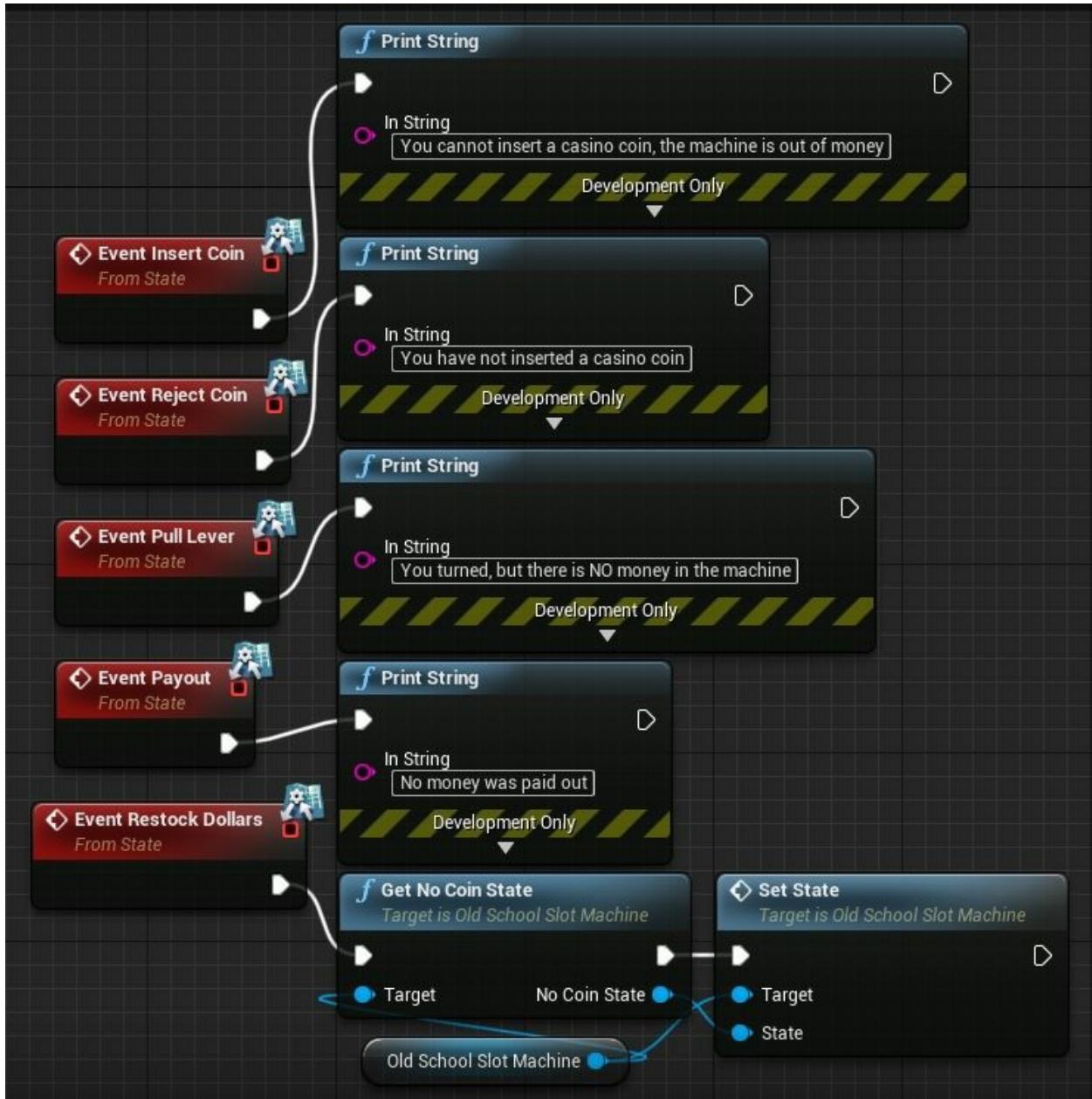
    public void payout() {
        System.out.println("No money was paid out");
    }

    public void restockDollars() {

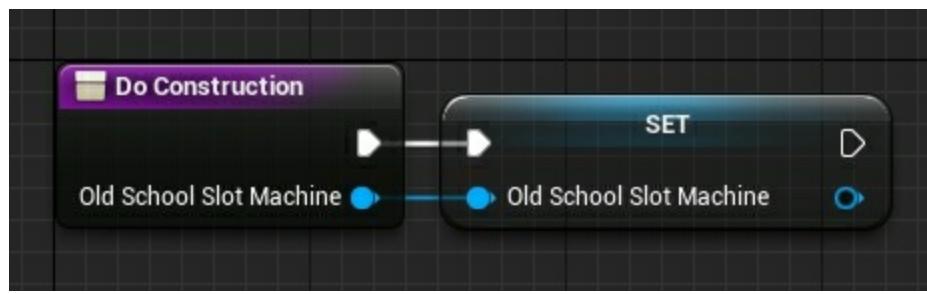
        oldSchoolSlotMachine.setState(oldSchoolSlotMachine.getNoCoinState());
    }

    public String toString() {
        return "machine is out of cash";
    }
}
```

The *NoDollarsState* event graph:



The *NoDollarsState doConstruction* function:



The *NoCoinState* class:

```
public class NoCoinState implements State {
    OldSchoolSlotMachine oldSchoolSlotMachine;

    public NoCoinState(OldSchoolSlotMachine oldSchoolSlotMachine) {
        this.oldSchoolSlotMachine = oldSchoolSlotMachine;
    }

    public void insertCoin() {
        System.out.println("You inserted a casino coin");

        oldSchoolSlotMachine.setState(oldSchoolSlotMachine.getCoinInsertedState(
    }

    public void rejectCoin() {
        System.out.println("You haven't inserted a casino coin");
    }

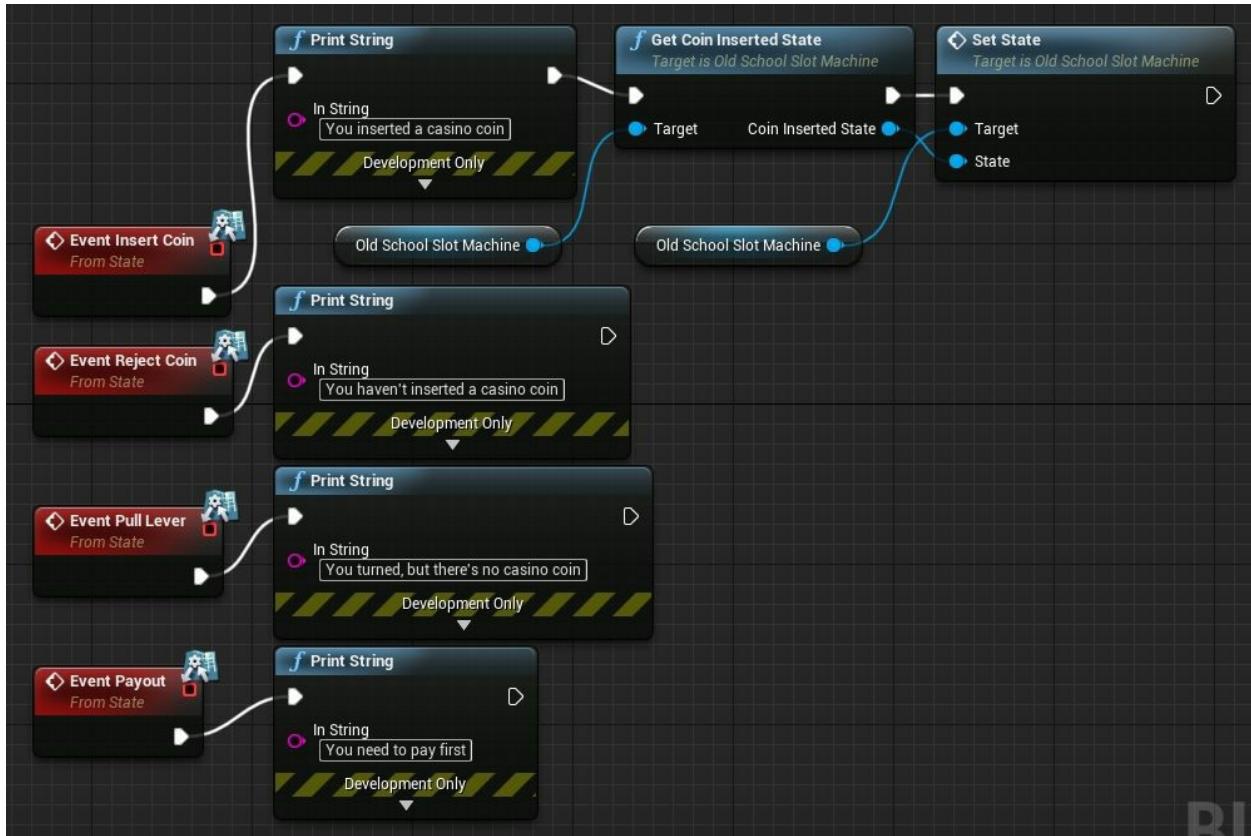
    public void pullLever() {
        System.out.println("You turned, but there's no casino coin");
    }

    public void payout() {
        System.out.println("You need to pay first");
    }

    public void restockDollars() {}

    public String toString() {
        return "waiting for a casino coin";
    }
}
```

The *NoCoinState* event graph:



The *NoCoinState doConstruction* function:



The *CoinInsertedState* class:

```
public class CoinInsertedState implements State {
    OldSchoolSlotMachine oldSchoolSlotMachine;

    public CoinInsertedState(OldSchoolSlotMachine oldSchoolSlotMachine) {
        this.oldSchoolSlotMachine = oldSchoolSlotMachine;
    }

    public void insertCoin() {
        System.out.println("You cannot insert another casino coin");
    }

    public void rejectCoin() {
        System.out.println("Casino coin returned");

        oldSchoolSlotMachine.setState(oldSchoolSlotMachine.getNoCoinState());
    }

    public void pullLever() {
        System.out.println("You pulled the lever...");

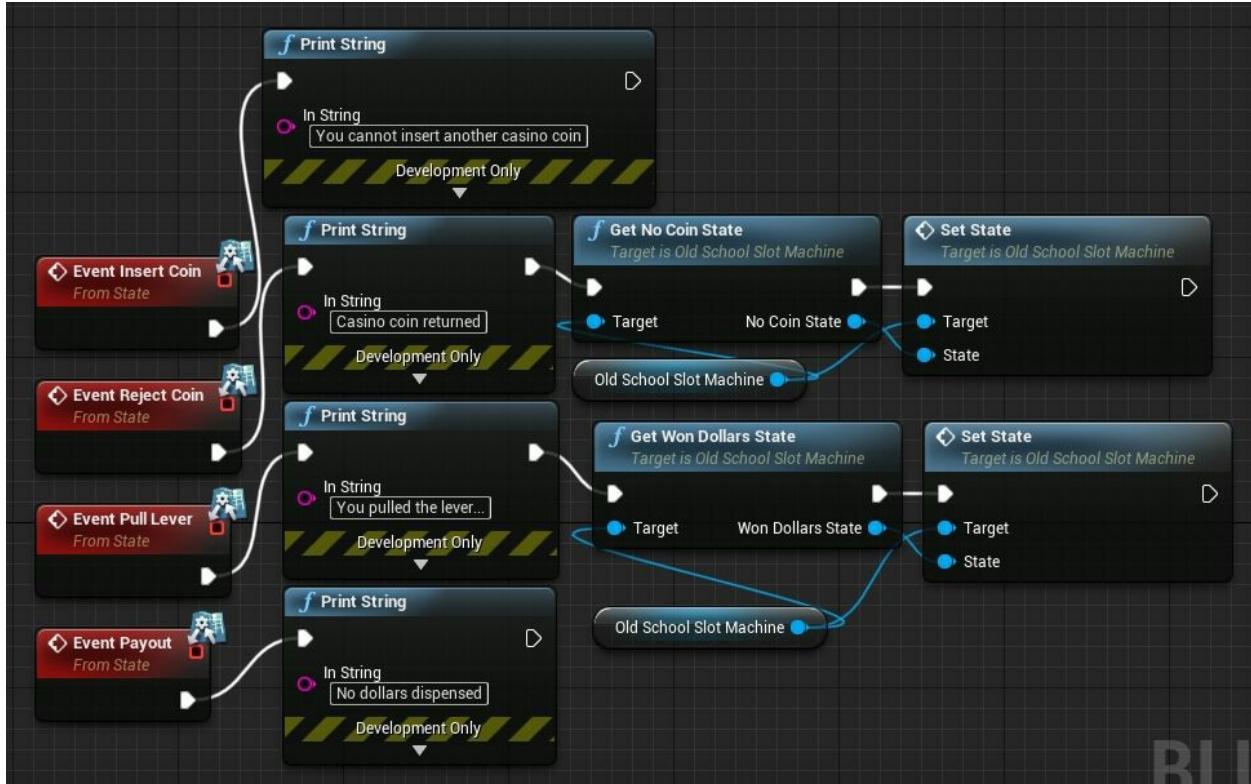
        oldSchoolSlotMachine.setState(oldSchoolSlotMachine.getWonDollarsState());
    }

    public void payout() {
        System.out.println("No dollars dispensed");
    }

    public void restockDollars() { }

    public String toString() {
        return "waiting for lever pull";
    }
}
```

The *CoinInsertedState* blueprint event graph:



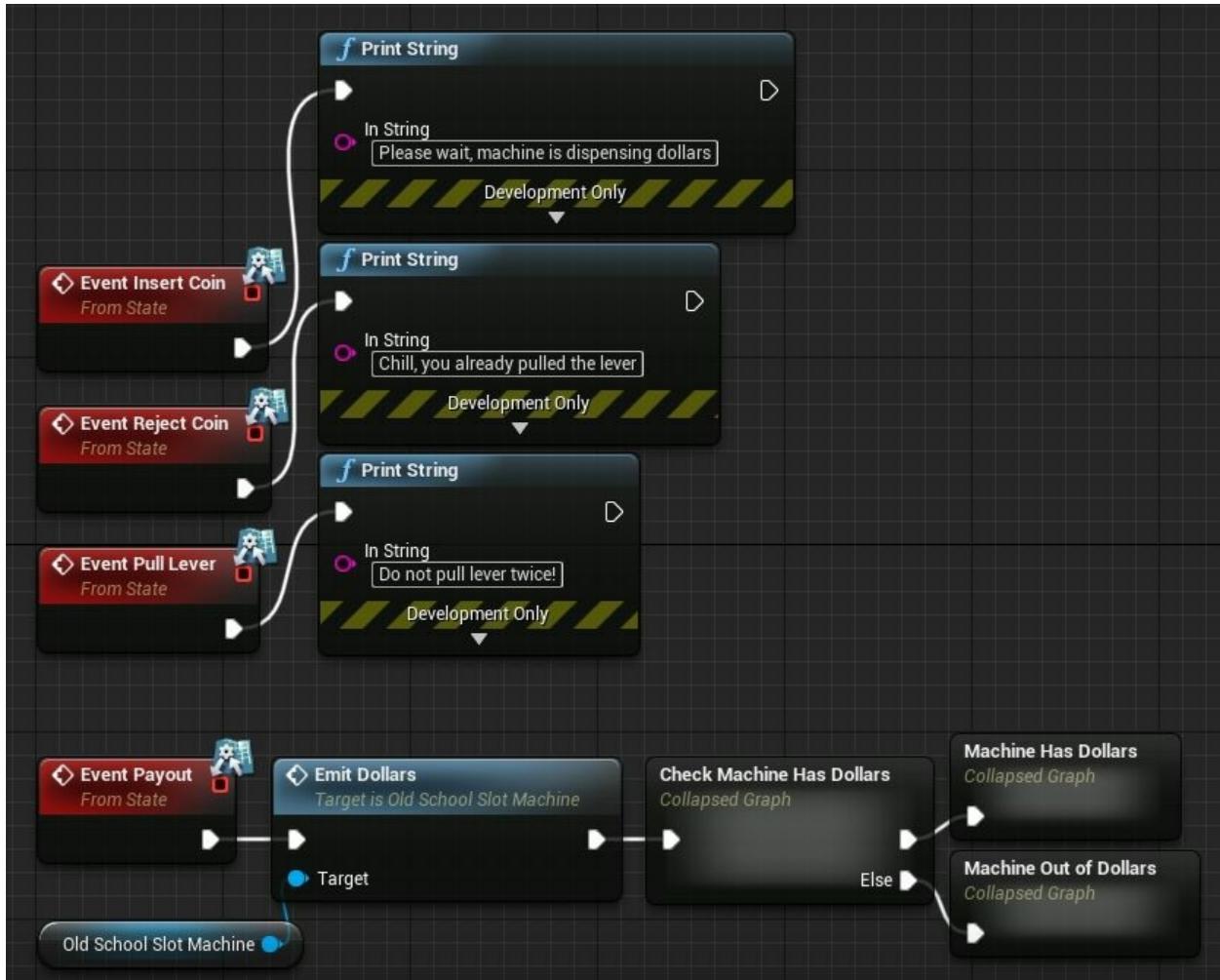
The *CoinInsertedState doConstruction* function:



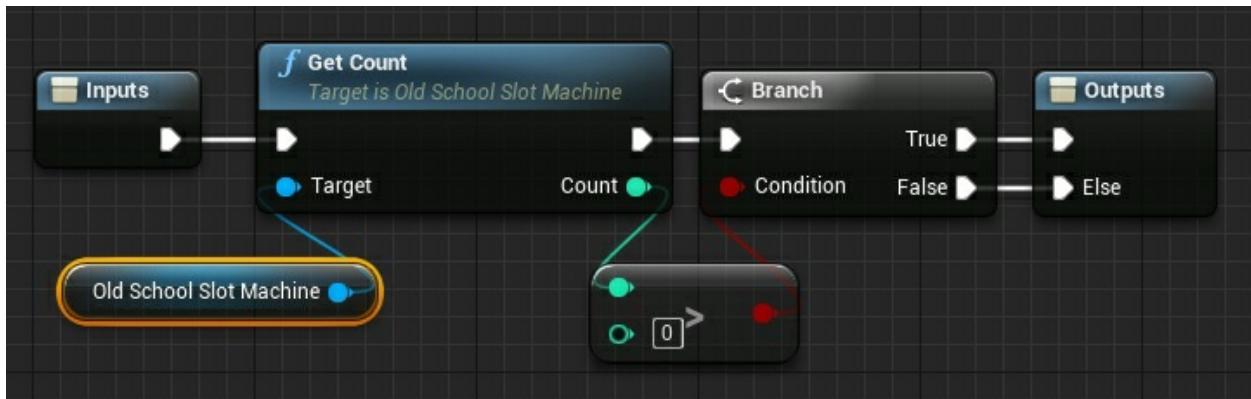
The *WonDollarsState* class:

```
public class WonDollarsState implements State {  
  
    OldSchoolSlotMachine oldSchoolSlotMachine;  
  
    public WonDollarsState(OldSchoolSlotMachine oldSchoolSlotMachine) {  
        this.oldSchoolSlotMachine = oldSchoolSlotMachine;  
    }  
  
    public void insertCoin() {  
        System.out.println("Please wait, machine is dispensing dollars");  
    }  
  
    public void rejectCoin() {  
        System.out.println("Chill, you already pulled the lever");  
    }  
  
    public void pullLever() {  
        System.out.println("Do not pull lever twice!");  
    }  
  
    public void payout() {  
        oldSchoolSlotMachine.emitDollars();  
        if (oldSchoolSlotMachine.getCount() > 0) {  
            oldSchoolSlotMachine.setState(  
                oldSchoolSlotMachine.getNoCoinState());  
        } else {  
            System.out.println("Machine is now out of cash");  
            oldSchoolSlotMachine.setState(  
                oldSchoolSlotMachine.getNoDollarsState());  
        }  
    }  
  
    public void restockDollars() {}  
  
    public String toString() {  
        return "dispensing cash";  
    }  
}
```

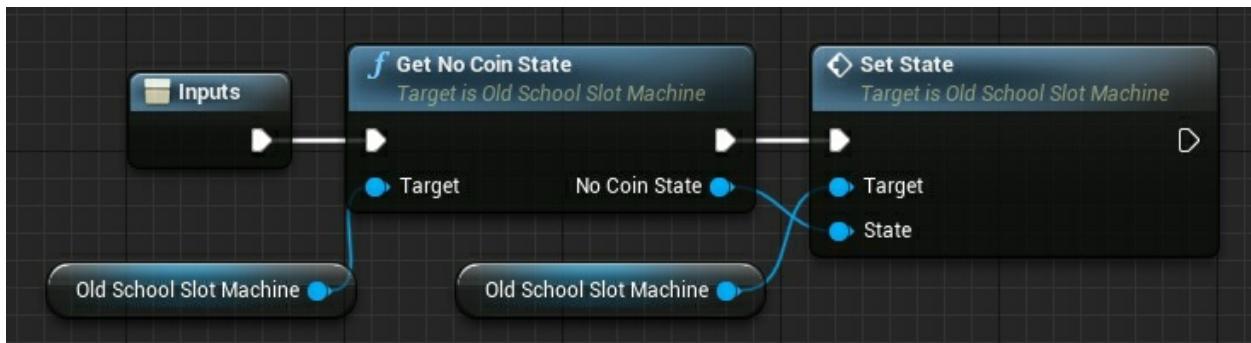
The *WonDollarsState* event graph:



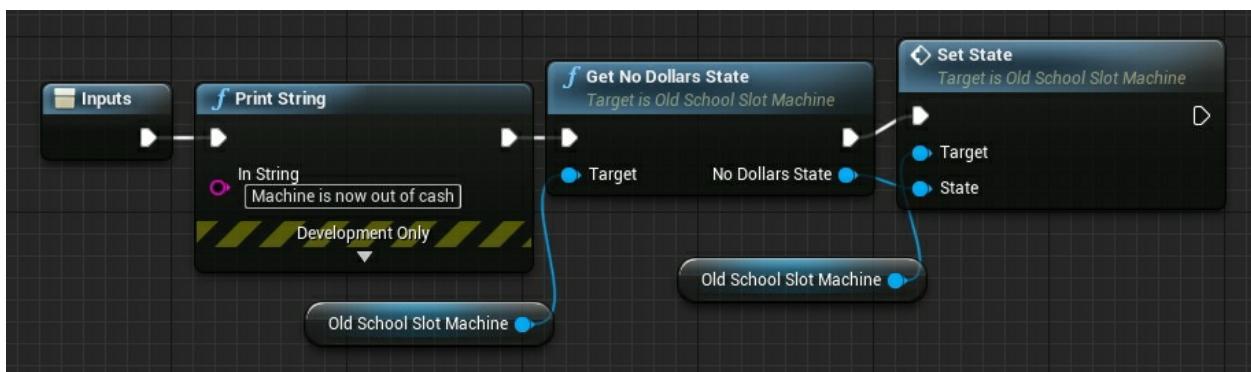
The *Check Machine Has Dollars* collapsed graph:



The *Machine Has Dollars* collapsed graph:



The *Machine Out of Dollars* collapsed graph collapsed graph:



The *WonDollarsState doConstruction* function:



The *State* interface:

```
public interface State {  
  
    public void insertCoin();  
    public void rejectCoin();  
    public void pullLever();  
    public void payout();  
    public void restockDollars();
```

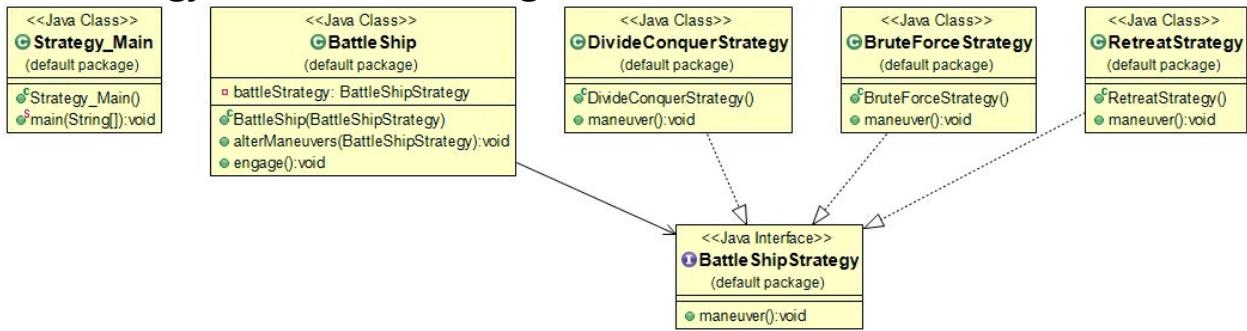
}

The State pattern viewport print:

```
Current Slot Machine State: NoCoinState
Make It Rain!
You pulled the lever...
You inserted a casino coin
The Old School Slot Machine was just filled and new dollar count is: 100
No money was paid out
You turned, but there is NO money in the machine
Current Slot Machine State: NoDollarsState
You cannot insert a casino coin, the machine is out of money
Machine is now out of cash
Make It Rain!
You pulled the lever...
You inserted a casino coin
Current Slot Machine State: NoCoinState
Make It Rain!
You pulled the lever...
You inserted a casino coin
Current Slot Machine State: NoCoinState
```

# You Sunk My Battleship... Strategy Pattern (Java)

Strategy Pattern UML Diagram

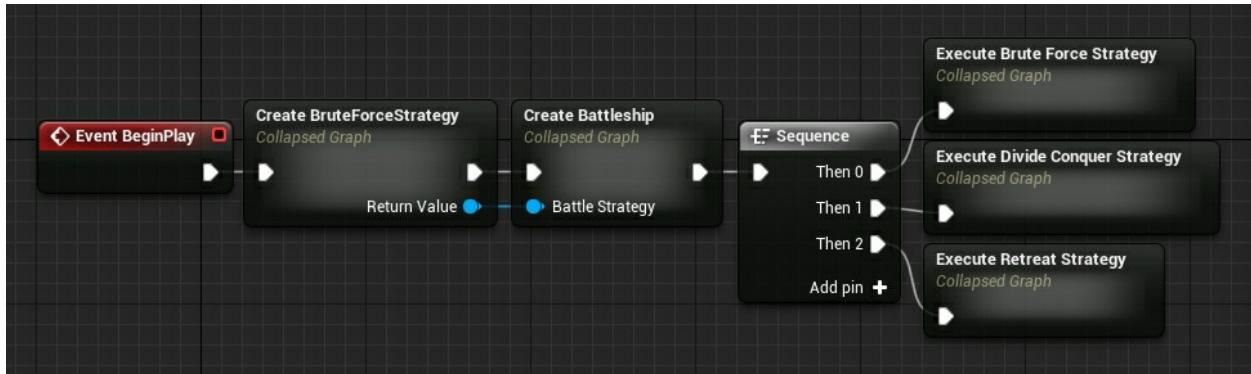


## **Strategy Pattern Implementation**

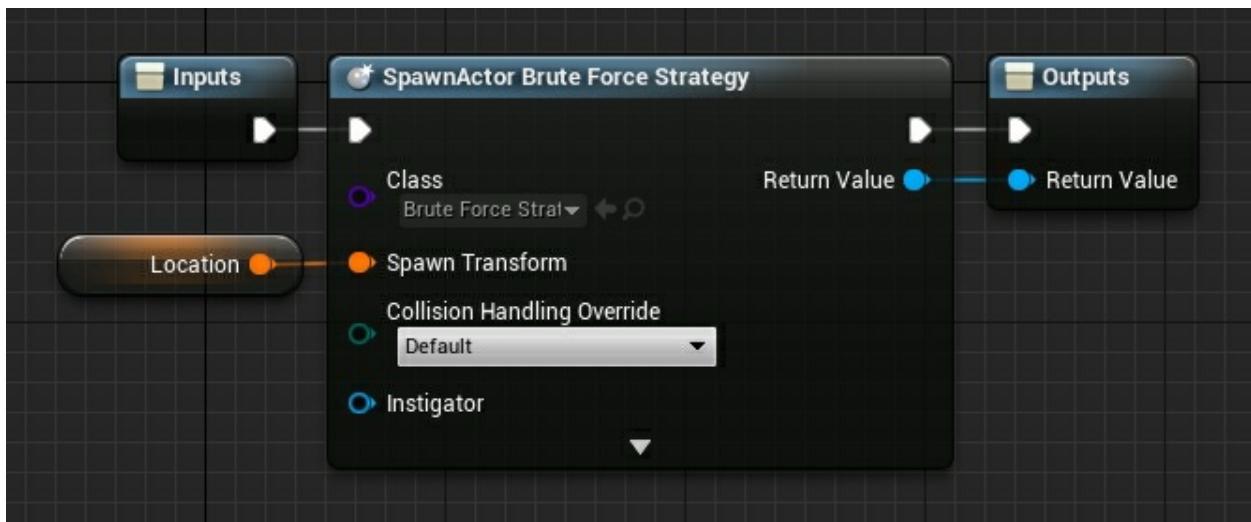
We have a strategy for combating different enemies at sea. The brute force strategy is used if the threat is small. We use the divide and conquer strategy if there are many enemies. The retreat strategy is executed if the battle is deemed futile. The *Strategy\_Main* class:

```
public class Strategy_Main {  
    public static void main(String[] args) {  
  
        System.out.println("A tiny frigate wants some trouble");  
        BattleShip battleShip = new BattleShip(new BruteForceStrategy());  
        battleShip.engage();  
        System.out.println("Four tiny frigates want some trouble");  
        battleShip.alterManeuvers(new DivideConquerStrategy());  
        battleShip.engage();  
        System.out.println("An aircraft carrier group wants some trouble");  
        battleShip.alterManeuvers(new RetreatStrategy());  
        battleShip.engage();  
    }  
}
```

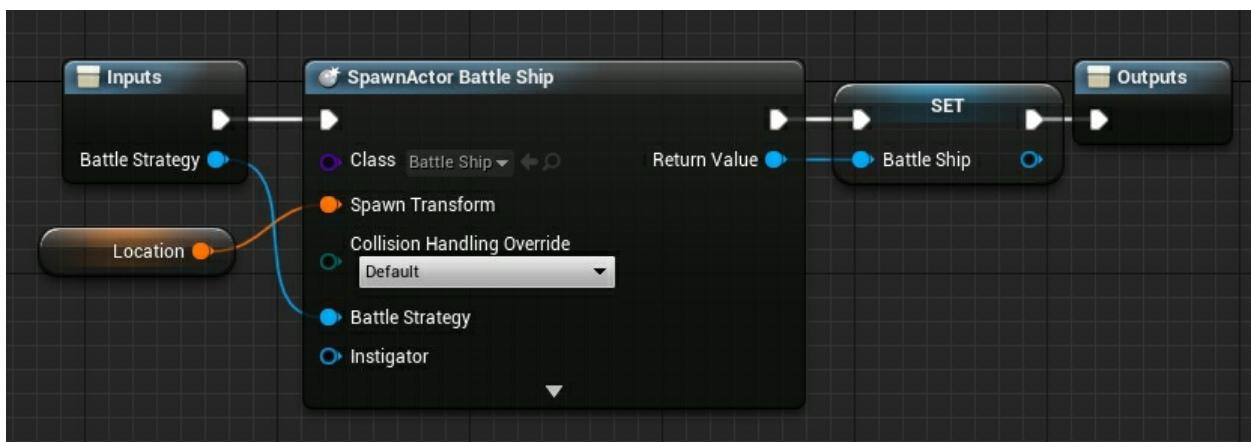
The *Strategy\_Main* blueprint event graph:



The *Create BruteForceStrategy* collapsed graph:



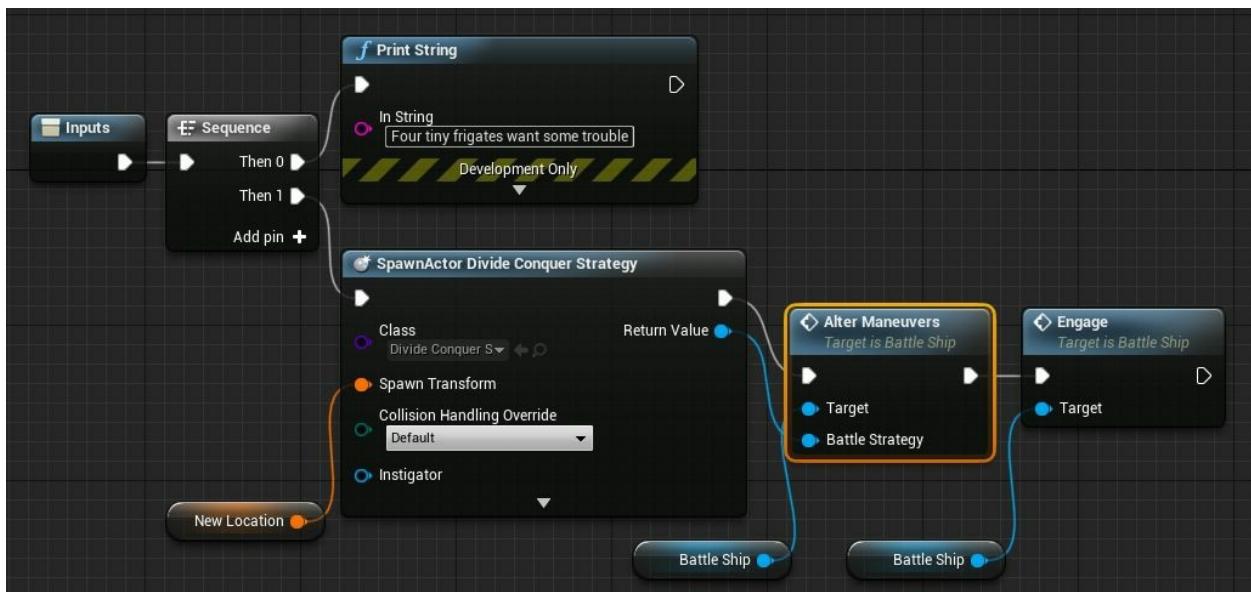
The *Create Battleship* collapsed graph:



The *Execute Brute Force Strategy* collapsed graph:



The *Execute Divide Conquer Strategy* collapsed graph:



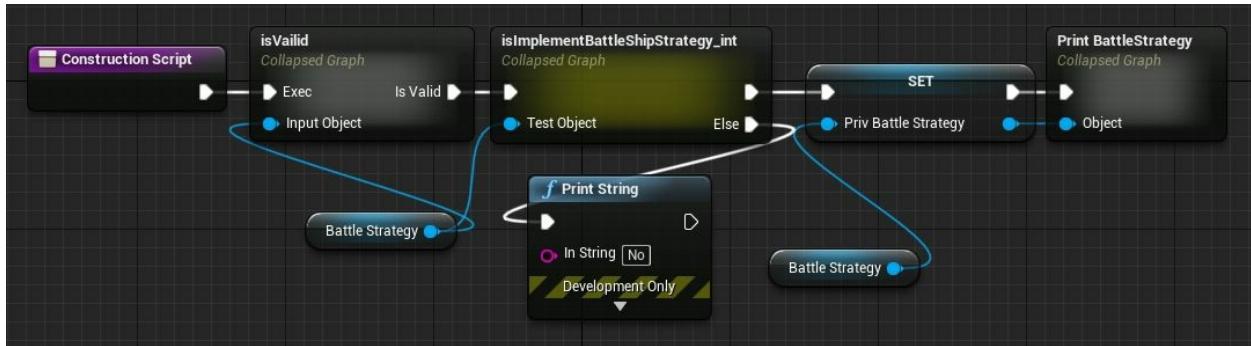
The *Execute Retreat Strategy* collapsed graph:



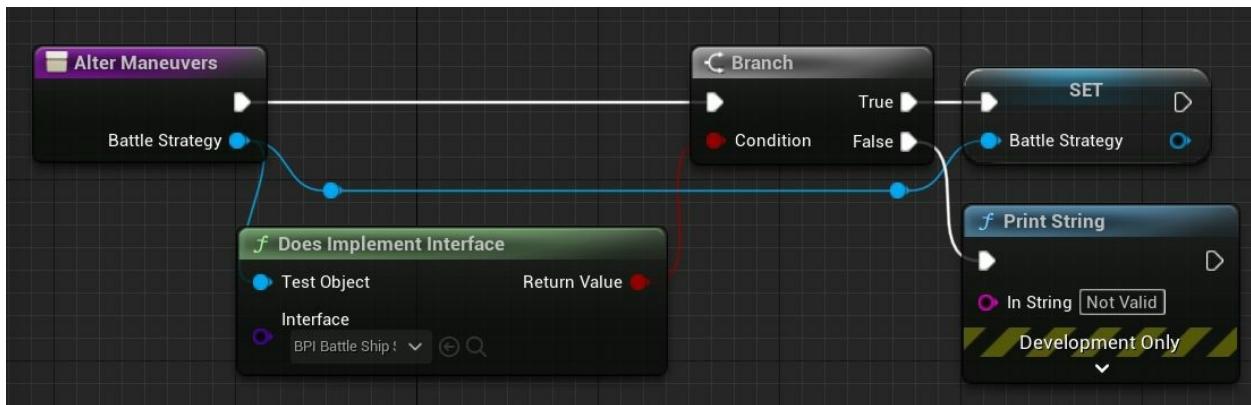
The *BattleShip* class:

```
public class BattleShip {  
    private BattleShipStrategy battleStrategy;  
  
    public BattleShip(BattleShipStrategy battleStrategy) {  
        this.battleStrategy = battleStrategy;  
    }  
  
    public void alterManeuvers(BattleShipStrategy battleStrategy) {  
        this.battleStrategy = battleStrategy;  
    }  
  
    public void engage() {  
        battleStrategy.maneuver();  
    }  
}
```

The *BattleShip* construction script:

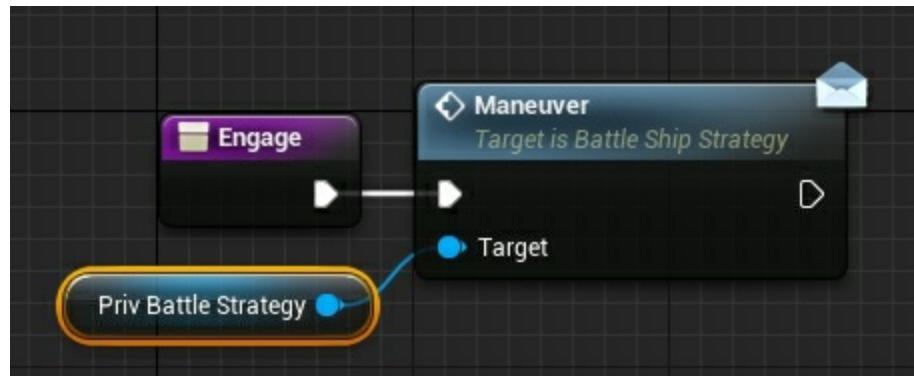


The *alterManeuvers* blueprint function:



Is identical to the *BattleShip* construction with the exception of actually setting the *battleStrategy* variable manually inside the function because it is not exposed on spawn like.

The *engage* blueprint function:



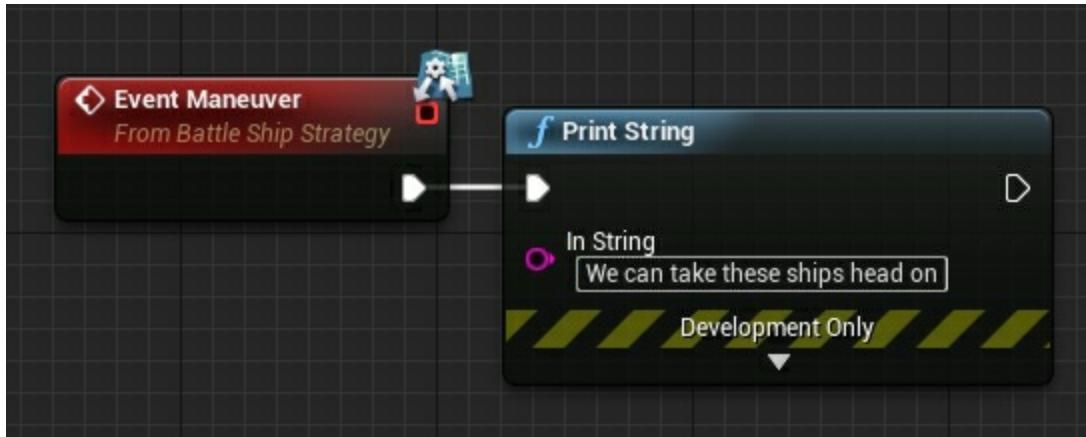
The *BattleShipStrategy* interface:

```
public interface BattleShipStrategy {  
    void maneuver();  
}
```

The *BruteForceStrategy* class:

```
public class BruteForceStrategy implements BattleShipStrategy {  
  
    @Override  
    public void maneuver() {  
        System.out.println("We can take these ships head on");  
    }  
}
```

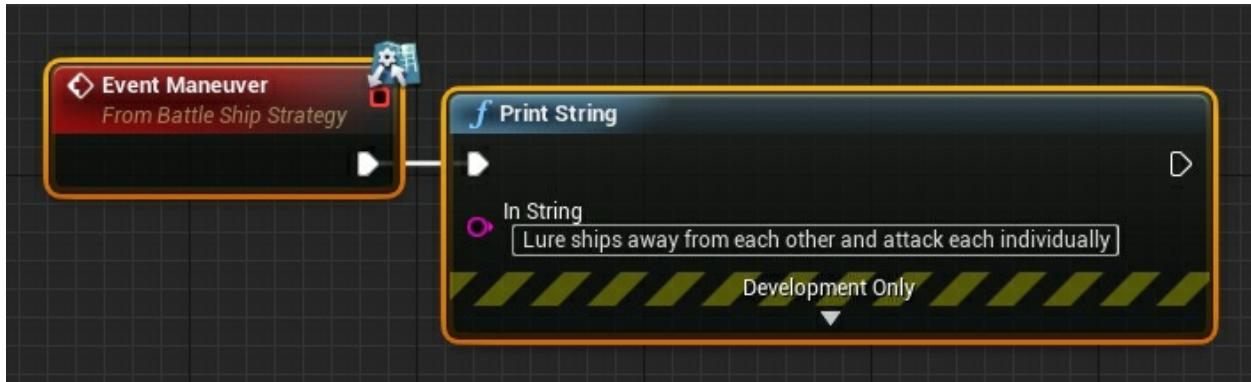
The *BruteForceStrategy* blueprint event graph:



The *DivideConquerStrategy* class:

```
public class DivideConquerStrategy implements BattleShipStrategy {  
  
    @Override  
    public void maneuver() {  
        System.out.println("Lure ships away from each other and attack each  
individually");  
    }  
}
```

The *DivideConquerStrategy* blueprint event graph:



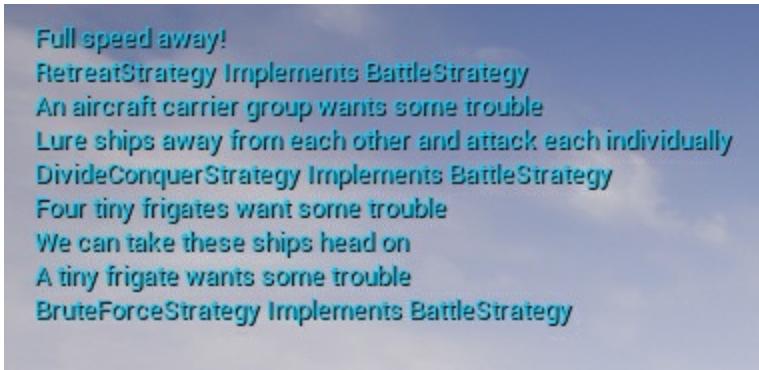
The *RetreatStrategy* class:

```
public class RetreatStrategy implements BattleShipStrategy {  
  
    @Override  
    public void maneuver() {  
        System.out.println("Full speed away!");  
    }  
  
}
```

The *RetreatStrategy* blueprint event graph:



The Strategy pattern viewport print:



Full speed away!  
RetreatStrategy Implements BattleStrategy  
An aircraft carrier group wants some trouble  
Lure ships away from each other and attack each individually  
DivideConquerStrategy Implements BattleStrategy  
Four tiny frigates want some trouble  
We can take these ships head on  
A tiny frigate wants some trouble  
BruteForceStrategy Implements BattleStrategy

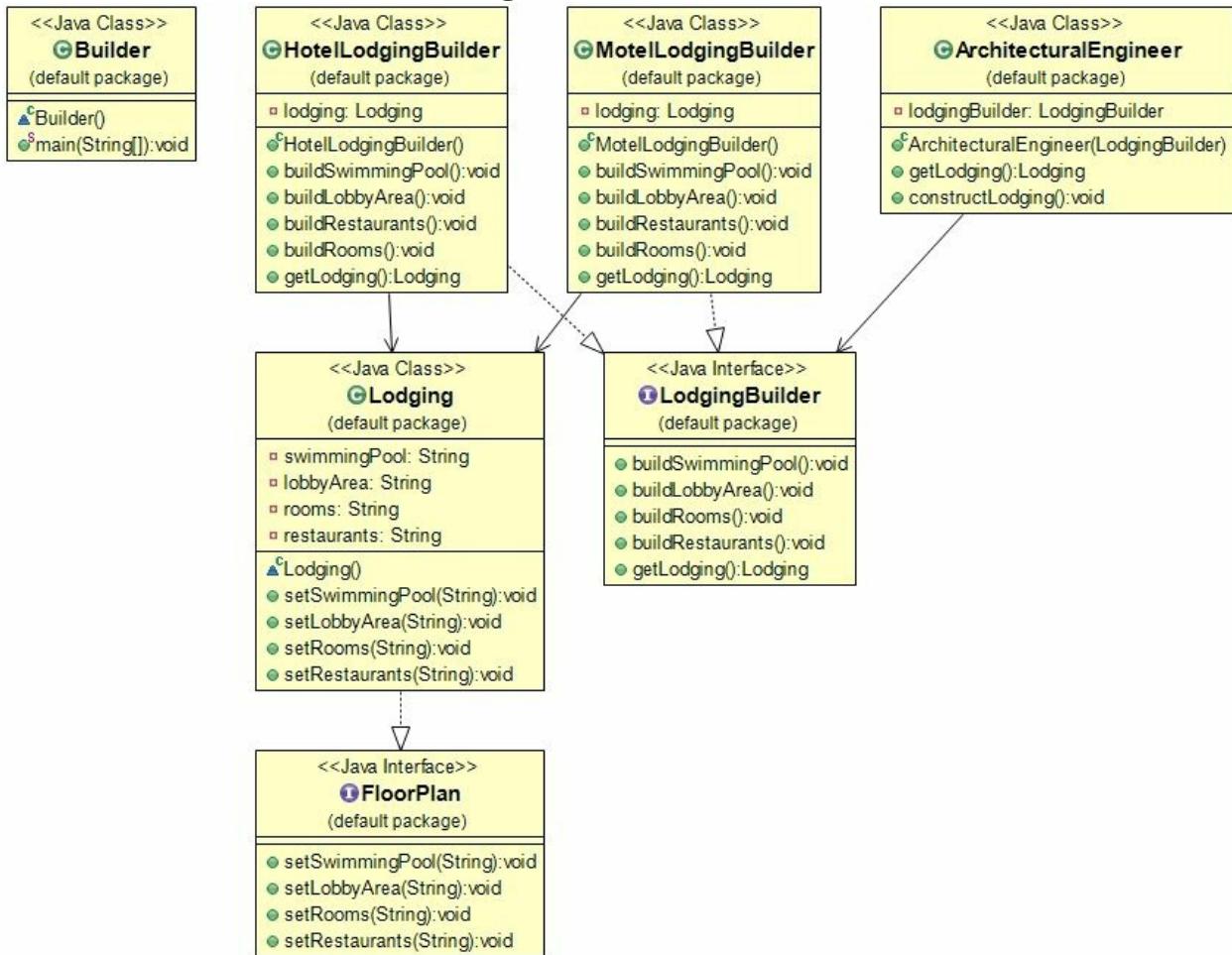
## The Patterns Using C#

In this section, we are going to implement most of the patterns we have covered using C#. The reader will notice that the blueprints equivalent to C# are identical to the blueprints shown in the Java section. This section will be most beneficial to those with a Unity programming background. C# is the lingua franca for high level scripting in the Unity game engine. Let's quickly call out the *using directive*. It will not have any performance impact on a final app or game. Directives as such are compiler “shortcuts” to deal with long type names. Also, a more in-depth discussion on the specific pattern and its implementation please refer to the C++ section. The discussion details remain the same regardless which language one is using.

# Hotel Motel Holiday Inn... Builder Pattern (C#)

The Builder Pattern in C# has a UML diagram identical to the Java implementation and thus that diagram is shown here. Ignore the “Java” class text.

**Builder Pattern UML Diagram**



## **Builder Pattern Implementation**

The *Builder* class contains the main function in C#. In terms of Blueprints, we can think of this class as the one that actually needs to be physically dragged into the game world. The other “concrete” classes do not need to be physically dragged into the world.

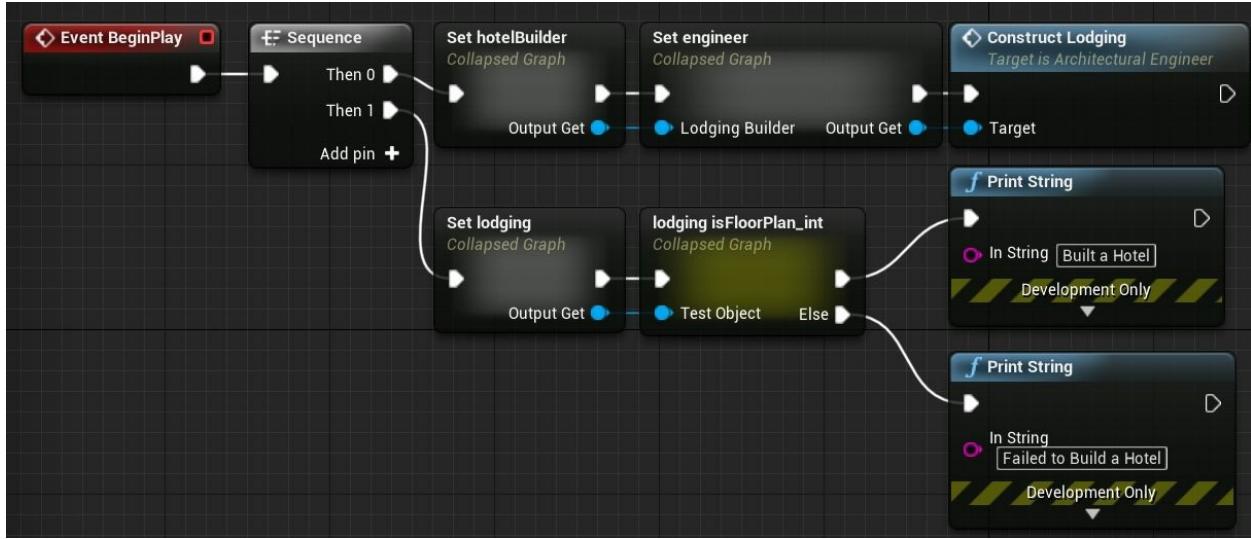
```
using System;

namespace Builder
{
    class Builder_Main
    {
        static void Main(string[] args)
        {
            LodgingBuilder hotelBuilder = new HotelLodgingBuilder();
            ArchitecturalEngineer engineer = new
                ArchitecturalEngineer(hotelBuilder);

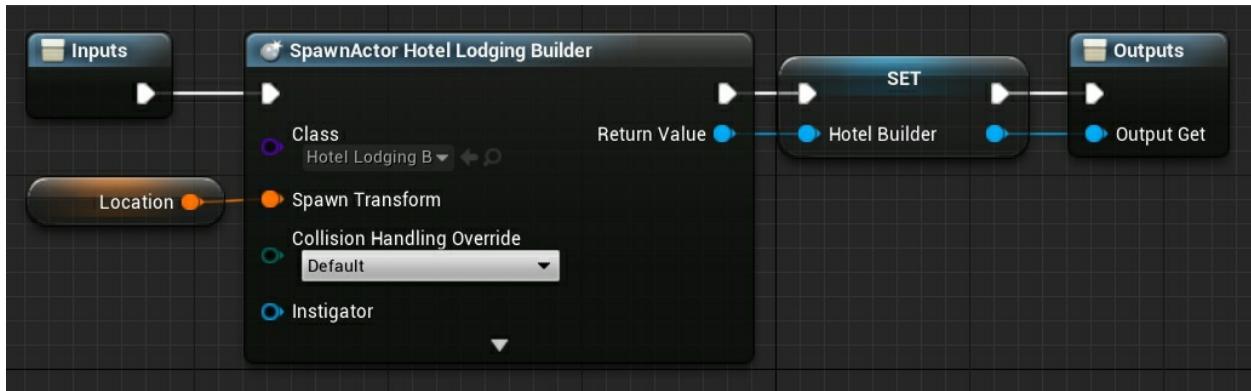
            engineer.constructLodging();

            Lodging lodging = engineer.getLodging();
            lodging.lodgingCharacteristics();
        }
    }
}
```

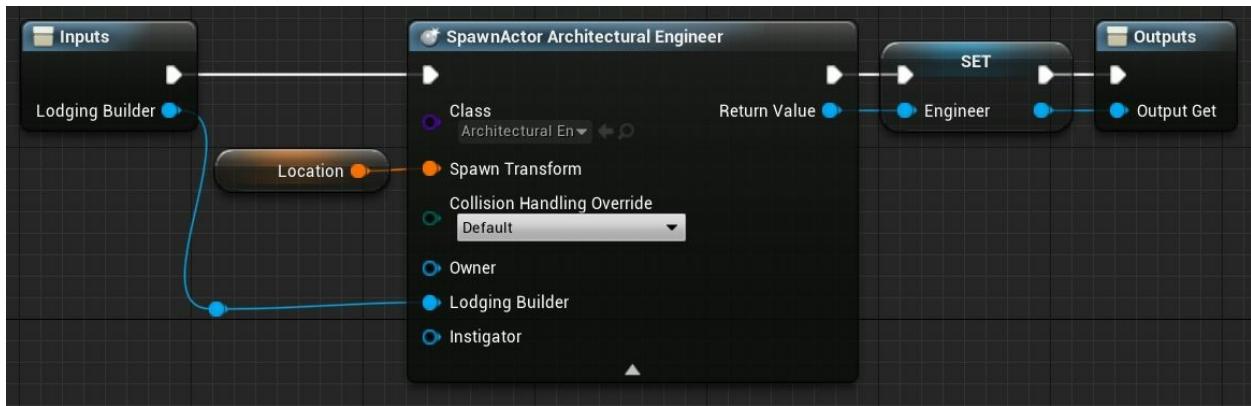
The blueprint equivalent of the *Builder\_Main* class:



The *Set hotelBuilder* and *Set engineer* collapsed graph are respectively:



and



The *Set hotelBuilder* collapsed graph represents this line of code:  
LodgingBuilder hotelBuilder = **new** HotelLodgingBuilder();

The *Spawn Actor From Class* node is akin to the *new* keyword in C# and other languages. The class to spawn is chosen using the dropdown box under the word “Class”. The Spawn Transform is to one’s own specific scenario. For simplicity, we can just right click on the Spawn Transform and choose “Promote to Variable”. If there is no Spawn Transform then the blueprint will compile with errors. We set (or “initialize”) our *hotelBuilder* variable once the desired actor is spawned.

The same basic idea applies to the *Set engineer* collapsed graph. However, *Architectural Engineer* needs a *LodgingBuilder* to be passed into its constructor as expressed in this line of code:

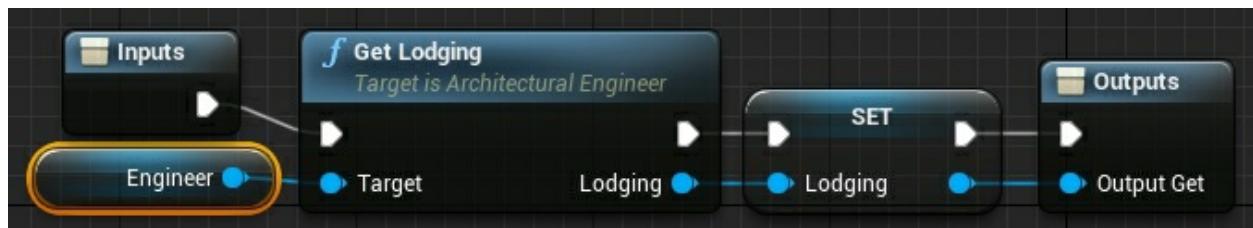
ArchitecturalEngineer engineer = **new** ArchitecturalEngineer(hotelBuilder);

Later, we will not be going into such detail on every spawning activity. Trust that there are many, many spawn actor requirements, and discussing each one will become an exercise in tedium.

*The Builder\_Main* blueprint continues with the *Construct Lodging* function call node. This function is called on the *engineer* we spawned previously. The equivalent line of code:

engineer.constructLodging();

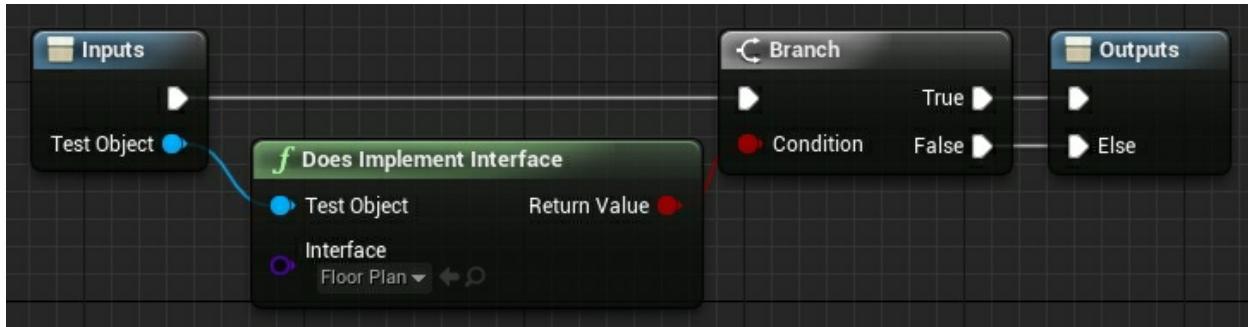
Note the *Sequence* node in *Builder\_Main*. Sequence 0 finishes once the *Construct Lodging* function call executes. Sequence 1 then starts executing. The *Set lodging* collapsed graph:



Now, we are equivalently executing this line of code:

```
Lodging lodging = engineer.getLodging();
```

This is the last line of code in the *Builder* class. However, we need to make some validation checks and make sure we built a hotel. We want to make sure the *lodging* actor implements the *FloorPlan* interface. The *lodging isFloorPlan\_int* collapsed graph:



Going forward we will not be showing this interface check collapsed graph. Note, the collapsed graph for this check always possesses a yellow hue. The only item that changes inside the collapsed node is the target interface we are checking the object against. The target interface is always part of the collapsed graph name.

The *Builder\_Main* blueprint ends execution with a print string to let us know if the hotel is actually built. “Failed to Build a Hotel” prints If an error in logic occurs somewhere in any of the blueprints.

Now we can discuss the interfaces we use in this pattern. First, we create a floor plan interface for all lodgings to implement. Most commercial lodgings in the USA contain a pool, lobby area, rooms, and restaurants. We also create a lodging builder interface for our specific lodging builder classes to implement. The *FloorPlan* interface:

```
using System;
interface FloorPlan
{
    public void setSwimmingPool(String swimmingPool);

    public void setLobbyArea(String lobbyArea);

    public void setRooms(String rooms);

    public void setRestaurants(String restaurants);
}
```

The equivalent blueprint interface is very mundane. The following are the nodes representing the functions in the *FloorPlan* interface.



The saying goes, “Program to an Interface, not an implementation” (Gamma et.al, 1995). In fact, it is a core principle in object-oriented design. The main benefit of programming to an interface is that it reduces implementation dependencies between subsystems (Gamma et.al, 1995). We see this first hand in our Builder Pattern example. Specifically, The Builder pattern seeks to decouple the construction of complex objects from its actual representation (Gamma et.al, 1995).

```
interface LodgingBuilder
{
    public void buildSwimmingPool();
    public void buildLobbyArea();
    public void buildRooms();
    public void buildRestaurants();
    public Lodging getLodging();
}
```

The screenshot of the *LodgingBuilder* interface blueprint follows the same design as the *FloorPlan* interface blueprint and is thus removed for the sake of brevity. Going forward, showing interface blueprint images are minimized unless considered necessary to deepen understanding for the reader.

Next, we create the *Lodging* class which implements the *FloorPlan* interface.

```
using System;
class Lodging : FloorPlan
{
    private String swimmingPool;
    private String lobbyArea;
    private String rooms;
    private String restaurants;

    public void setSwimmingPool(String swimmingPool)
    {
        this.swimmingPool = swimmingPool;
    }

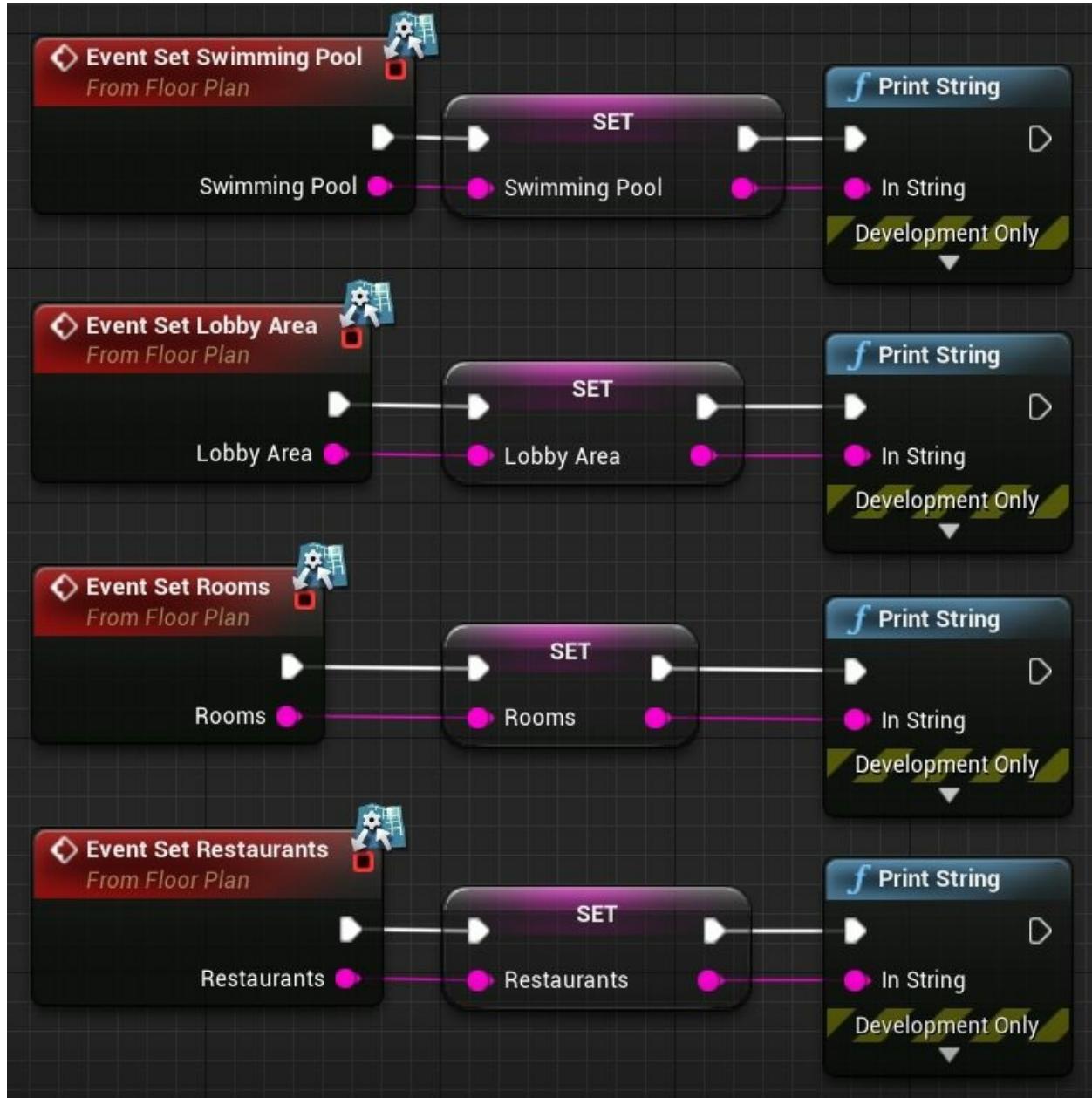
    public void setLobbyArea(String lobbyArea)
    {
        this.lobbyArea = lobbyArea;
    }

    public void setRooms(String rooms)
    {
        this.rooms = rooms;
    }

    public void setRestaurants(String restaurants)
    {
        this.restaurants = restaurants;
    }

    public void lodgingCharacteristics()
    {
        Console.WriteLine(swimmingPool);
        Console.WriteLine(lobbyArea);
        Console.WriteLine(rooms);
        Console.WriteLine(restaurants);
    }
}
```

The *Lodging* blueprint:



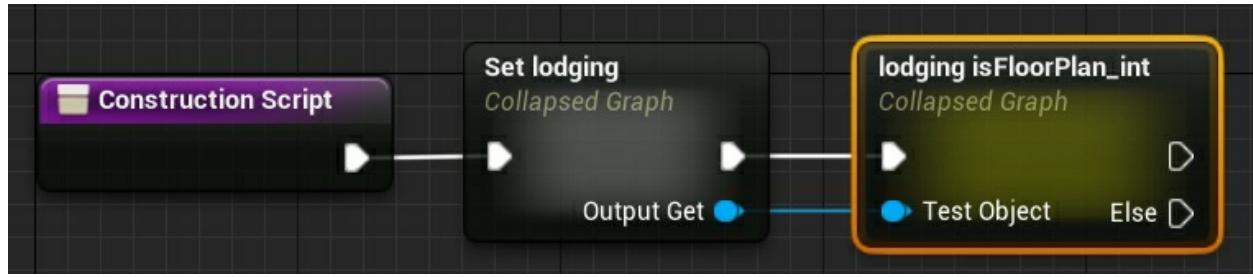
Blueprints use *Event* nodes when calling void interface functions. The blueprint version of the *Lodging* class prints the string value passed to the function in addition to setting local variables. Also, we must make sure this blueprint class implements the *FloorPlan* interface by adding it in the class settings Interfaces sections.

The next two classes, *HotelLodgingBuilder* and *MotelLodgingBuilder*,

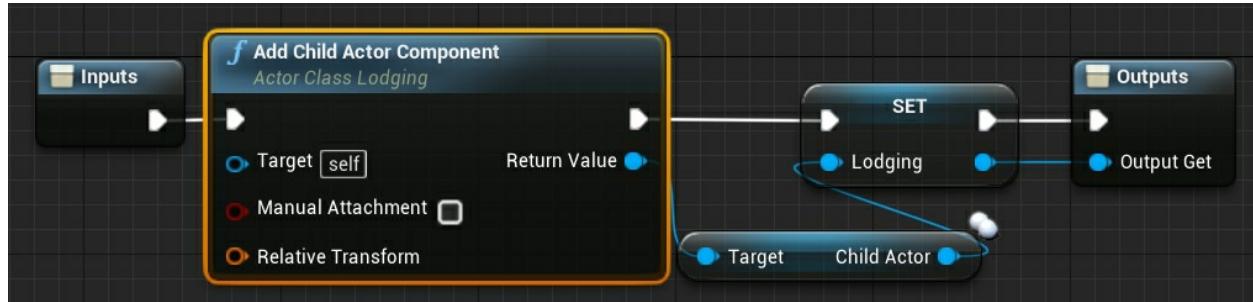
implement the lodging builder interface and provide the custom specs for each type of lodging.

```
class HotelLodgingBuilder implements LodgingBuilder {  
  
    private Lodging lodging;  
  
    public HotelLodgingBuilder(){  
        this.lodging = new Lodging();  
    }  
  
    @Override  
    public void buildLobbyArea() {  
        lodging.setLobbyArea("Grand Hall");  
    }  
  
    @Override  
    public void buildRestaurants() {  
        lodging.setRestaurants("5 star Buffet");  
    }  
  
    @Override  
    public void buildRooms() {  
        lodging.setRooms("Luxury Suites");  
    }  
  
    @Override  
    public void buildSwimmingPool() {  
        lodging.setSwimmingPool("Indoor Lagoon");  
    }  
  
    @Override  
    public Lodging getLodging() {  
        return this.lodging;  
    }  
}
```

The *HotelLodgingBuilder* blueprint construction script event graph:



The *Set lodging* collapsed graph node:



The *Add Child Actor Component* is a work around for not being able to use the *Spawn Actor of Class* node. We can see the *HotelLodgingBuilder* constructor instantiates a new *Lodging* object in our code:

```
public HotelLodgingBuilder()
{
    this.lodging = new Lodging();
}
```

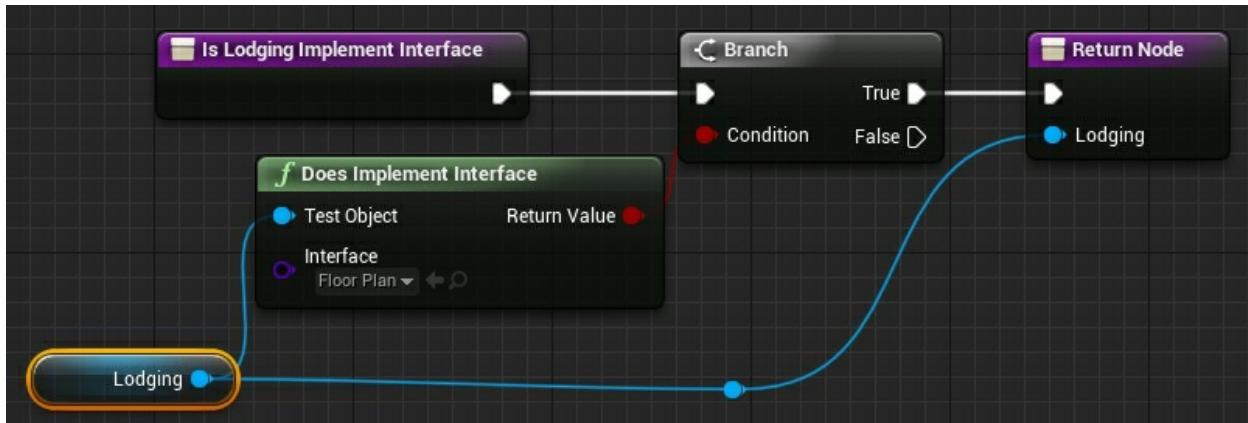
We will see an even more dynamic way to initialize objects around construction script constraints when we get to the State Pattern.

*HotelLodgingBuilder* event graph:



The *isLodgingImplementInterface* function is a custom interface check function similar the yellow hued collapsed graphs.

The *isLodgingImplementInterface* function:



The *MotelLodgingBuilder* is very similar to the *HotelLodgingBuilder*. The difference is in the attributes for the pool, lobby area, rooms, and restaurant.

```
class MotelLodgingBuilder : LodgingBuilder
{
    private Lodging lodging;

    public MotelLodgingBuilder()
    {
        this.lodging = new Lodging();
    }

    public void buildSwimmingPool()
    {
        lodging.setSwimmingPool("Green Slim");
    }

    public void buildLobbyArea()
    {
        lodging.setLobbyArea("Economical");
    }

    public void buildRestaurants()
    {
        lodging.setRestaurants("Vending Machines");
    }

    public void buildRooms()
    {
        lodging.setRooms("Bates Style");
    }

    public Lodging getLodging()
    {
        return this.lodging;
    }
}
```

The blueprint screenshot for *MotelLodgingBuilder* is omitted for brevity. It basically mimics that of the *HotelLodgingBuilder*.

Now we need an architectural engineer to coordinate the step by step construction of the lodging.

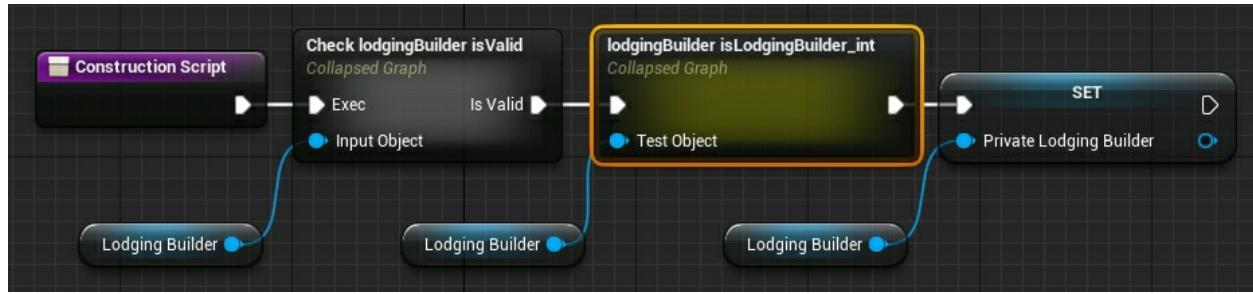
```
class ArchitecturalEngineer
{
    private LodgingBuilder lodgingBuilder;

    public ArchitecturalEngineer(LodgingBuilder lodgingBuilder)
    {
        this.lodgingBuilder = lodgingBuilder;
    }

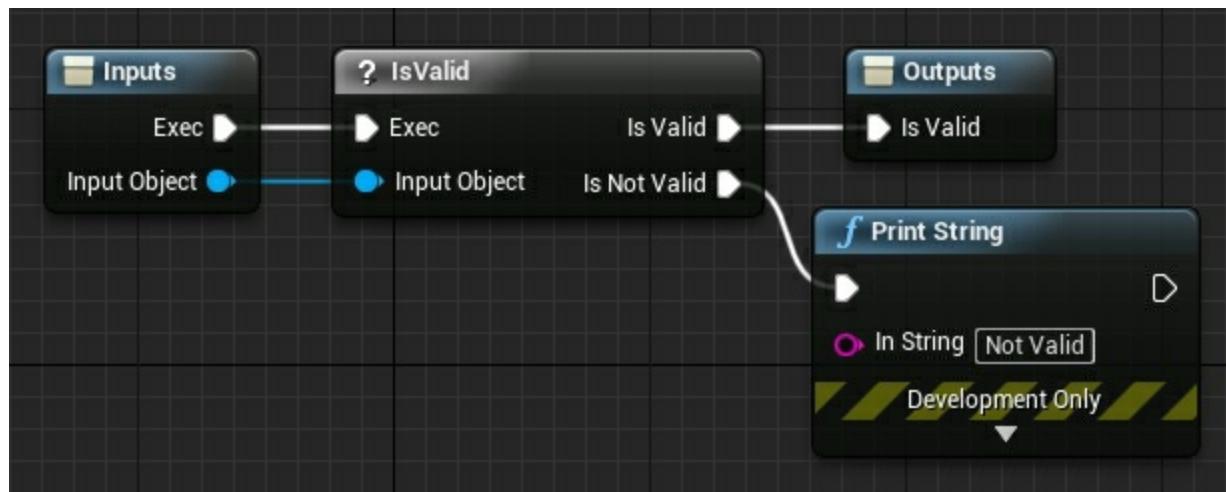
    public Lodging getLodging()
    {
        return this.lodgingBuilder.getLodging();
    }

    public void constructLodging()
    {
        this.lodgingBuilder.buildSwimmingPool();
        this.lodgingBuilder.buildLobbyArea();
        this.lodgingBuilder.buildRooms();
        this.lodgingBuilder.buildRestaurants();
    }
}
```

The *ArchitecturalEngineer* construction script:



The *Check lodgingBuilder isValid* node:



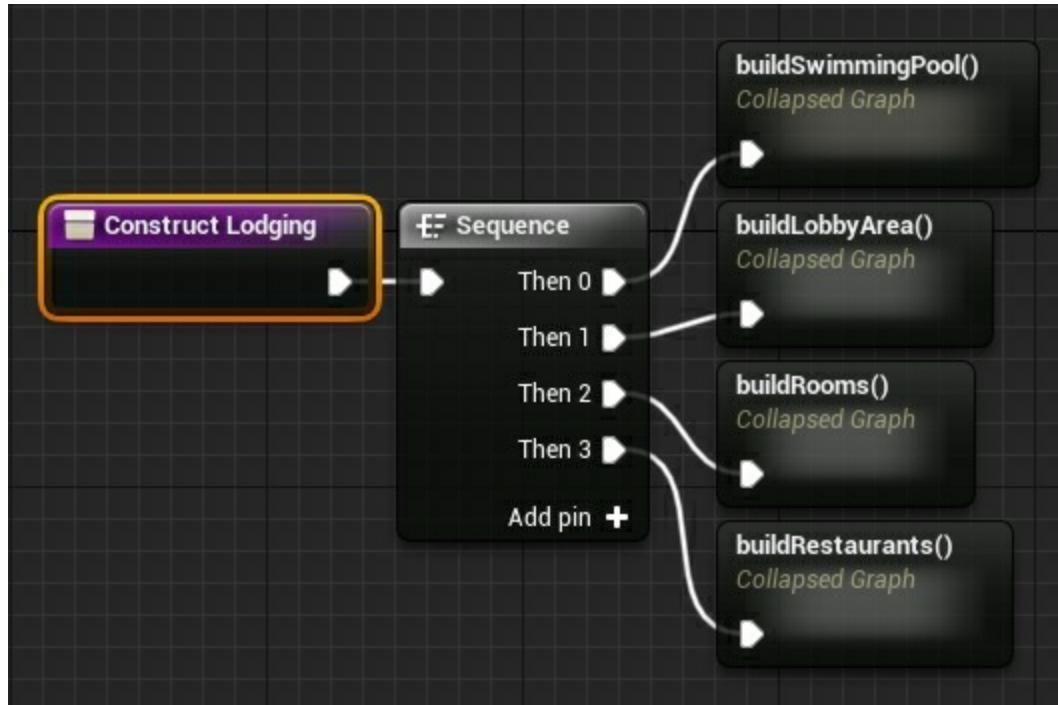
This validity check is executed as a safety mechanism. We want to make sure the object is not null before we start to perform operations on said object. A “Not Valid” string prints to the viewport If the object is indeed null.

The *ArchitecturalEngineer* contains a few important functions, namely *getLodging* and *constructLodging*.

The *getLodging* function graph:



The *constructLodging* function graph:



The collapsed graphs represent the body of the *constructLodging* function:

```
public void constructLodging()
{
    this.lodgingBuilder.buildSwimmingPool();
    this.lodgingBuilder.buildLobbyArea();
    this.lodgingBuilder.buildRooms();
    this.lodgingBuilder.buildRestaurants();
}
```

The *buildSwimmingPool*, *buildLobbyArea*, *buildRooms*, and *buildRestaurants* graphs are respectively:



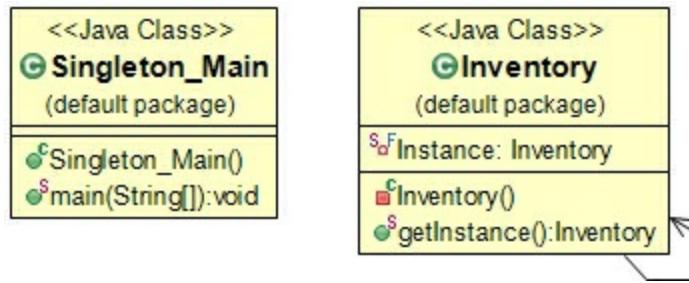
This is the final screenshot of the construction of a hotel.



Strings printed to the viewport are not impressive. However, these strings represent “what could be.” In development, one has the 3D or 2D artwork associated with these strings. Thanks to the Builder Pattern, we have a framework to build different versions of hotels piece by piece.

# Programming Elitists' Bane... Singleton Pattern (C#)

Singleton Pattern UML Diagram



## **Singleton Pattern Implementation**

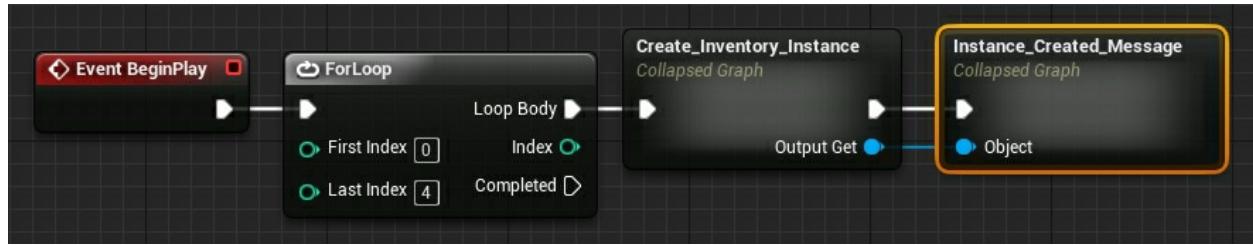
There is no clear-cut way to create a custom Singleton blueprint in Unreal. However, it appears the Game Instance is a first party solution to the Singleton. An inventory system is a great example for demonstrating the Singleton design pattern. We only want one inventory instance at all times. Additionally, we want to be able to easily access that inventory system throughout our game.

The main Singleton class *Singleton\_Main*:

```
using System;

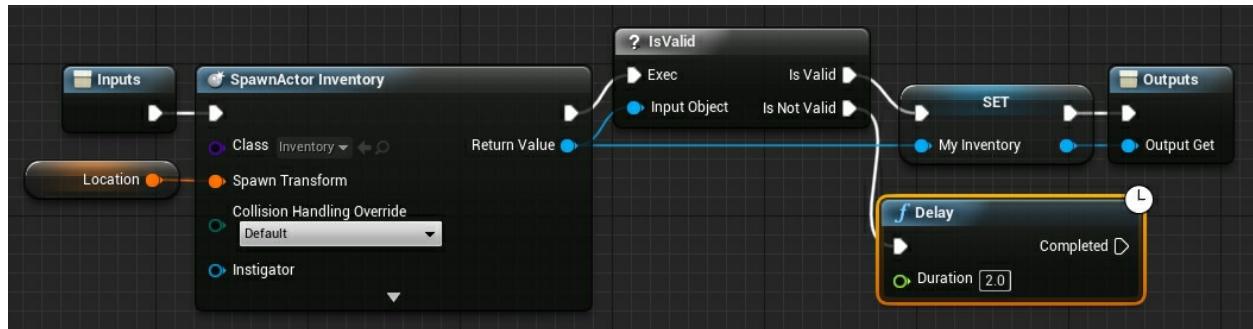
namespace Singleton
{
    class Singleton_Main
    {
        static void Main(string[] args)
        {
            Inventory myInventory = Inventory.getInstance();
            Console.WriteLine(myInventory);
        }
    }
}
```

The *Singleton\_Main* blueprint event graph:

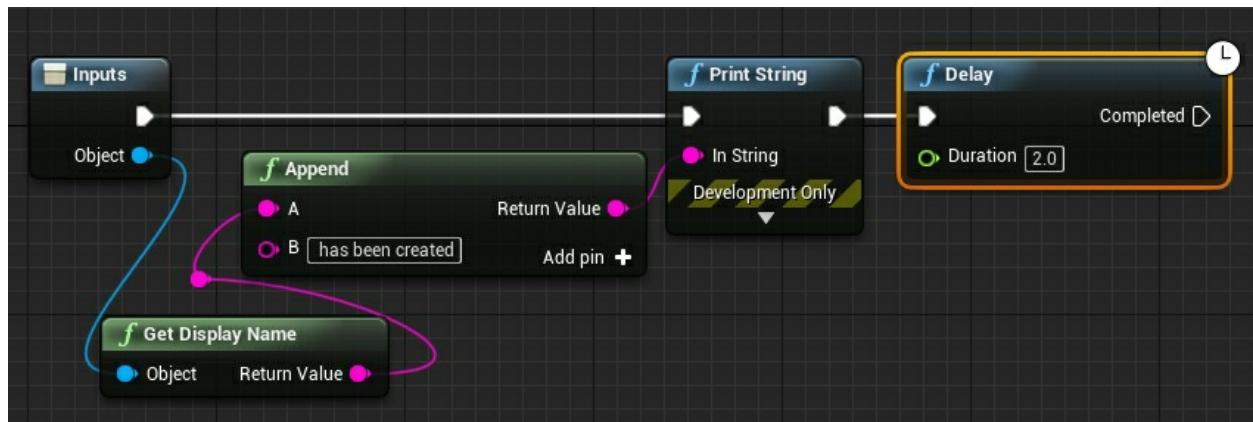


The for loop is present to demonstrate attempts to try to create more than one instance of our designated Singleton object. We should never be allowed to create multiple instances of our inventory If everything is set up correctly.

The *Create\_Inventory\_Instance* collapsed graph:



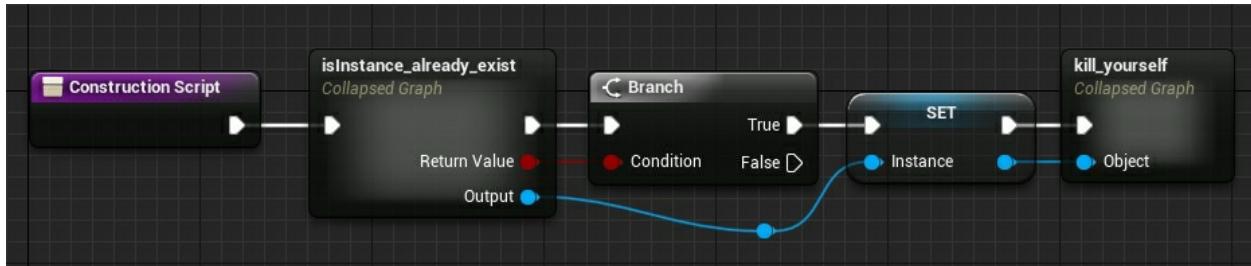
The *Instance\_Created\_Message* collapsed graph:



The *Inventory* class:

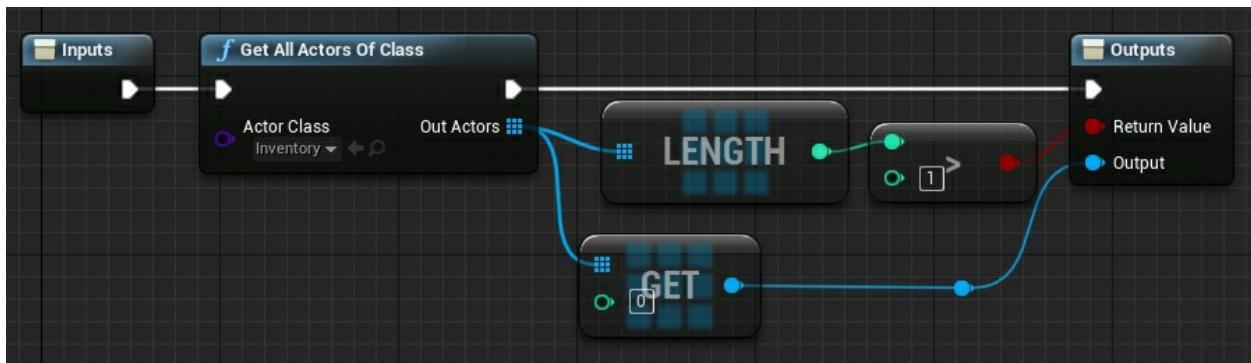
```
public class Inventory {  
    private static readonly Inventory Instance = new Inventory();  
  
    private Inventory() {}  
  
    public static Inventory getInstance() {  
        return Instance;  
    }  
}
```

The *Inventory* blueprint construction script:

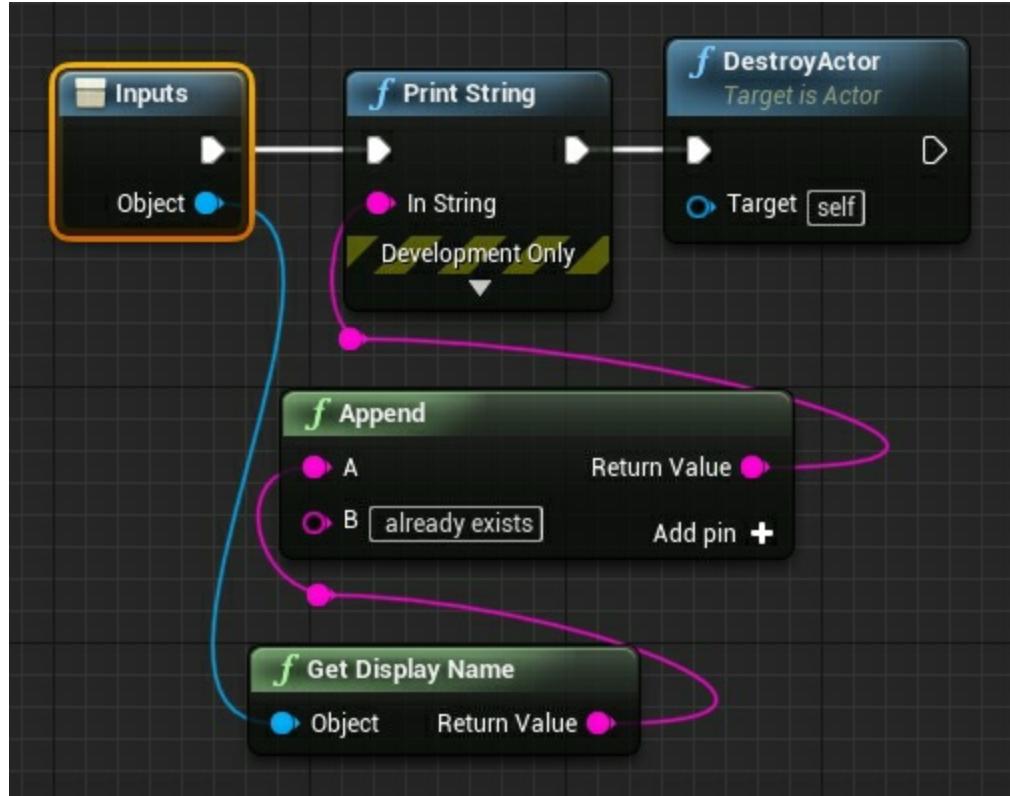


The concept of a private construction does not exist in Unreal. Therefore, we must create another way of keeping new Singleton objects from being created. We do this by checking if an inventory instance already exists. If an inventory instance does in fact already exists, the attempted new Singleton object is destroyed.

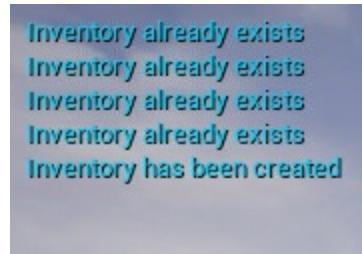
The *isInstance\_already\_exist* collapsed graph:



The *kill\_yourself* collapsed graph:

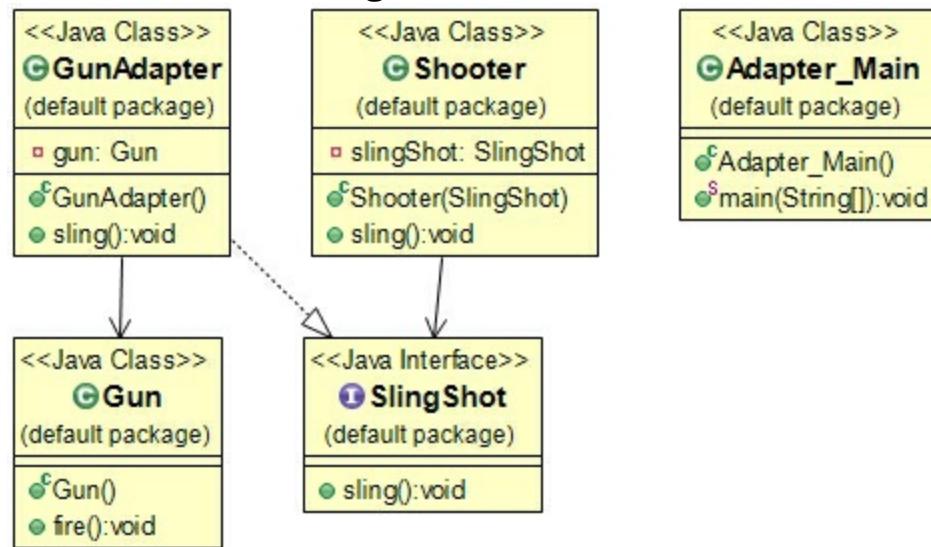


The Singleton pattern viewport screen print:



# Adapt and Overcome... Adapter Pattern (C#)

Adapter Pattern UML Diagram



## **Adapter Pattern Implementation**

In our Adapter pattern implementation, we have a shooter who uses a sling shot. We want to be able to allow our shooter to fire a gun. However, we want to use the existing code we have already implemented by using the Adapter pattern.

The main adapter class:

```
using System;

namespace Adapter
{
    class Adapter_Main
    {
        static void Main(string[] args)
        {
            Shooter shooter = new Shooter(new GunAdapter());
            shooter.sling();
        }
    }
}
```

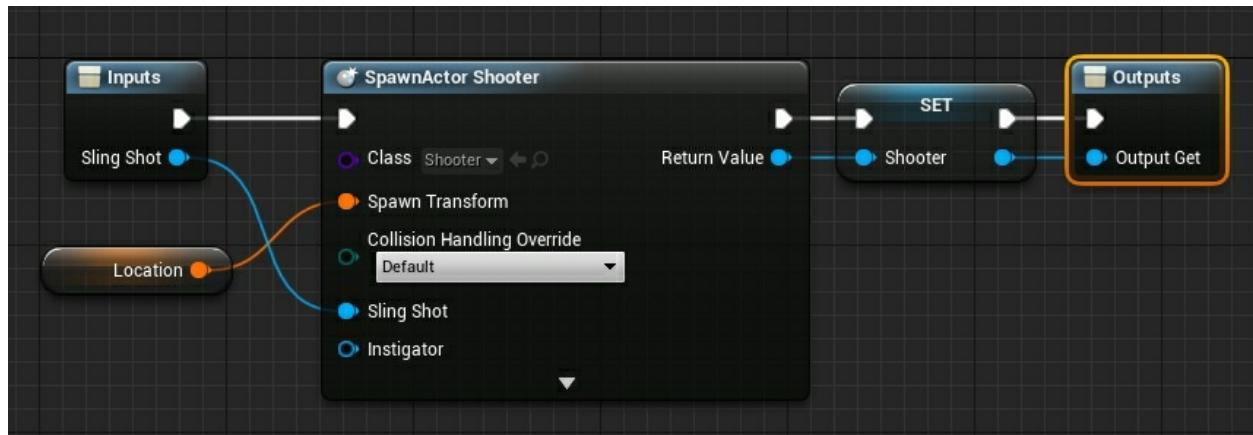
The *Adapter\_Main* blueprint:



The *Create\_Gun\_Adapter* collapsed graph:



The *Create\_Shooter* collapsed graph:



The *SlingShot* interface:

```
public interface SlingShot {
```

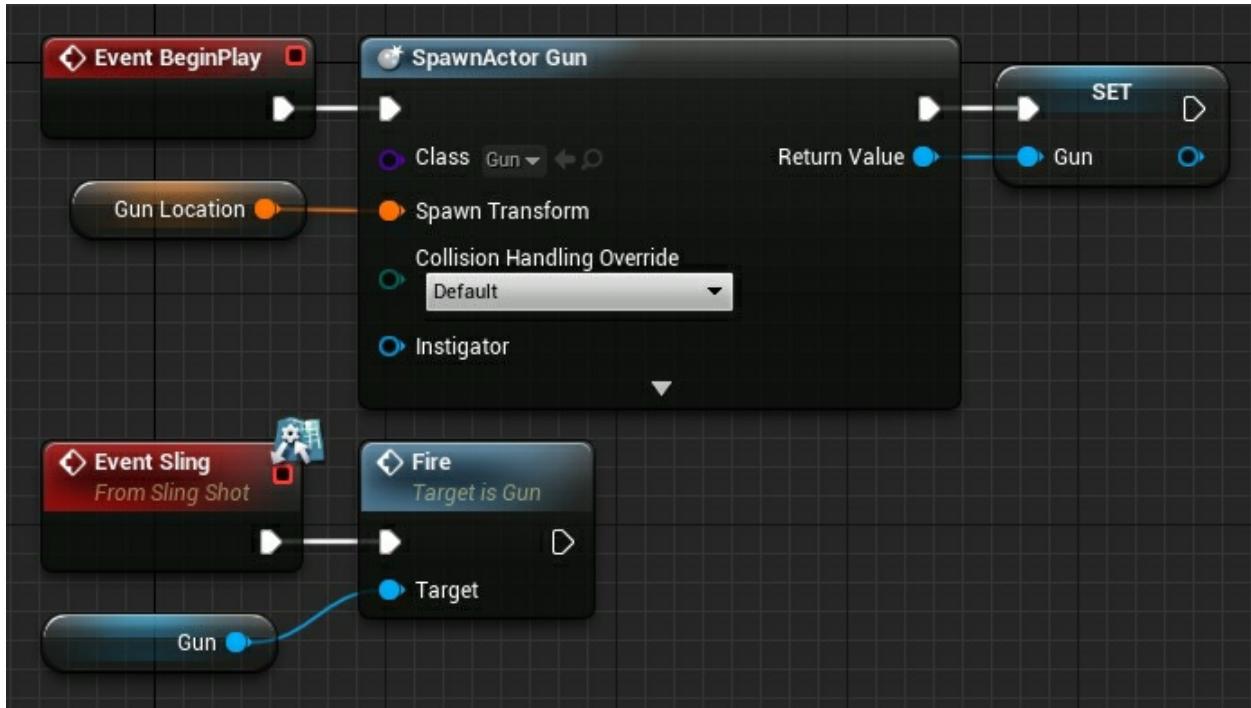
```
    void sling();
```

}

The *GunAdapter* class:

```
public class GunAdapter: SlingShot {  
  
    private Gun weapon;  
  
    public GunAdapter() {  
        weapon = new Gun();  
    }  
  
    public void sling() {  
        weapon.fire();  
    }  
}
```

The *GunAdapter* blueprint:



The *Gun* class:

```
using System;
public class Gun {

    public void fire() {
        Console.WriteLine("Our gun is firing");
    }
}
```

The *Gun* blueprint:

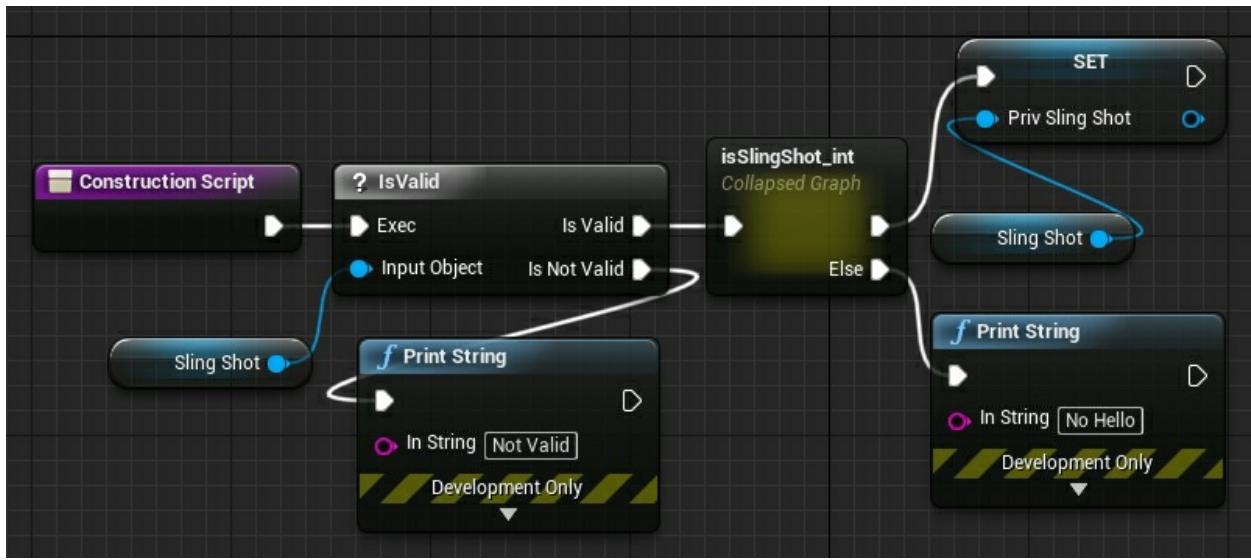


The *Shooter* class:

```
public class Shooter : SlingShot {  
    private SlingShot slingShot;  
  
    public Shooter(SlingShot slingShot) {  
        this.slingShot = slingShot;  
    }  
  
    public void sling0 {  
        slingShot.sling0;  
    }  
}
```

We can see that the *Shooter* class has no functionality to fire a gun. However, it does functionality to sling a slingshot. Both of these skills are predicated on emitting a projectile from a weapon. We would like to reuse the slingshot functionality and adapt it for the use of a gun.

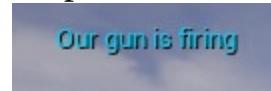
The *Shooter* construction script:



The *Sling* function:

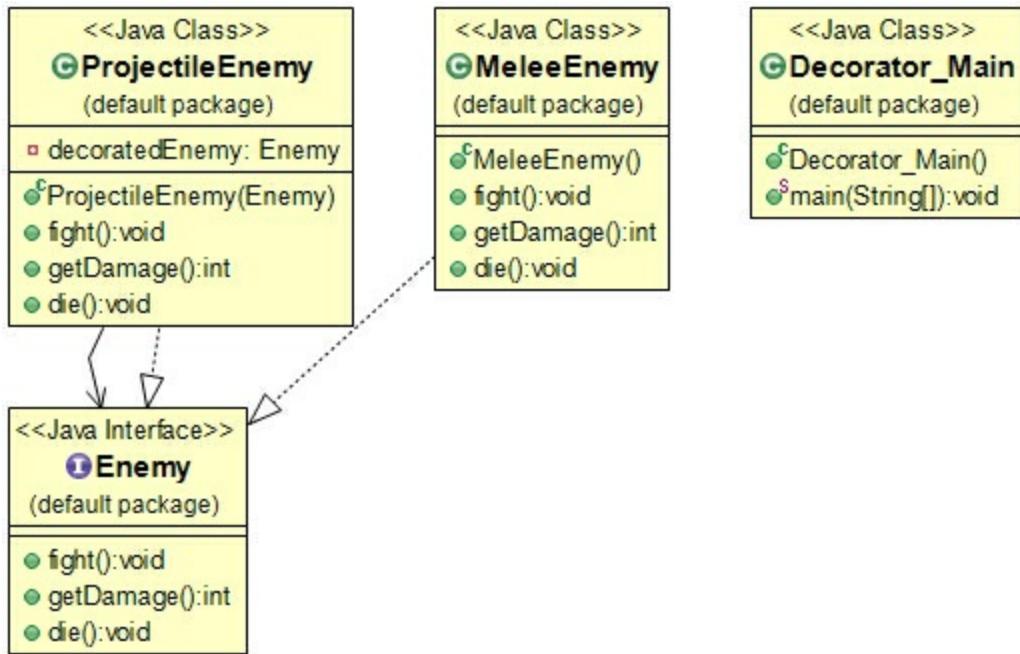


The Adapter pattern viewport print:



# Decorating Overpowered Enemies... Decorator Pattern (C#)

Decorator Pattern UML Diagram



## **Decorator Pattern Implementation**

Our implementation consists of an enemy who is initially a melee enemy. We want to “Decorate” the enemy and make an enemy who can use projectiles. Thus, the added functionality is the projectile ability.

The *Decorator\_Main* class:

```
using System;

namespace Decorator
{
    class Decorator_Main
    {
        static void Main(string[] args)
        {

            Console.WriteLine("Melee Enemies are on the horizon");
            Enemy enemy = new MeleeEnemy();
            enemy.fight();
            enemy.die();

            Console.WriteLine("Melee Enemies cause "+ enemy.getDamage() +" damage.");

            Console.WriteLine("Enemies are now armed with guns");
            Enemy projectileEnemy = new ProjectileEnemy(enemy);
            projectileEnemy.fight();
            projectileEnemy.die();

            Console.WriteLine("Projectile Enemies cause "+
projectileEnemy.getDamage()+" damage.");
        }
    }
}
```

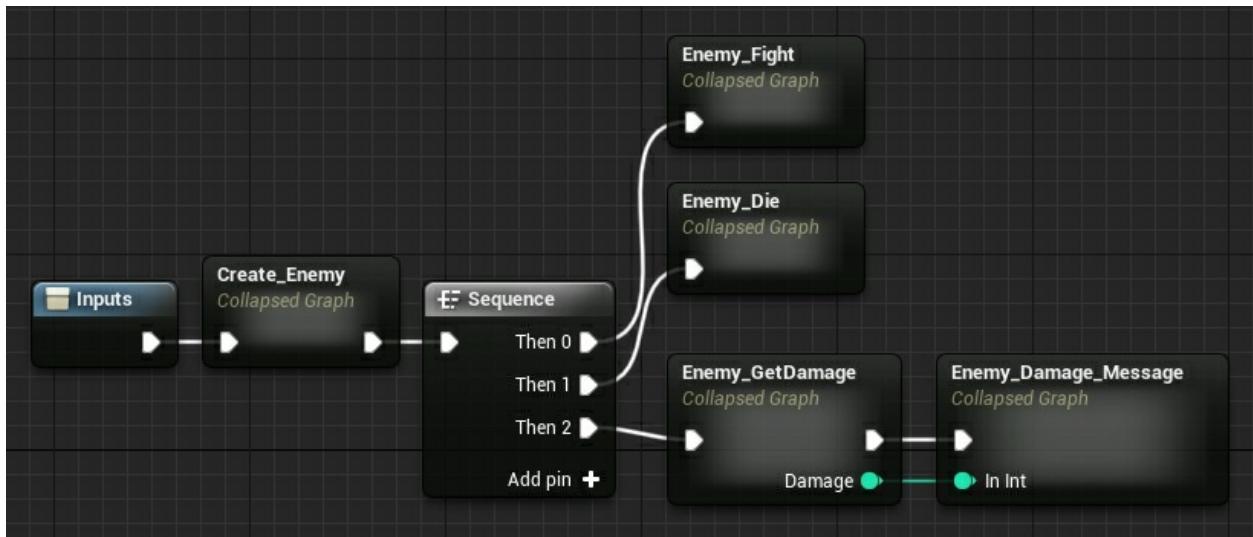
We can see from the *Decorator\_Main* that we do not need to make a separate *Enemy* class to add features. We merely “Decorate” the enemy, and now we have a projectile enemy. Effectively, we changed the behavior of the enemy at runtime by using a decorator.

The *Decorator\_Main* blueprint event graph:

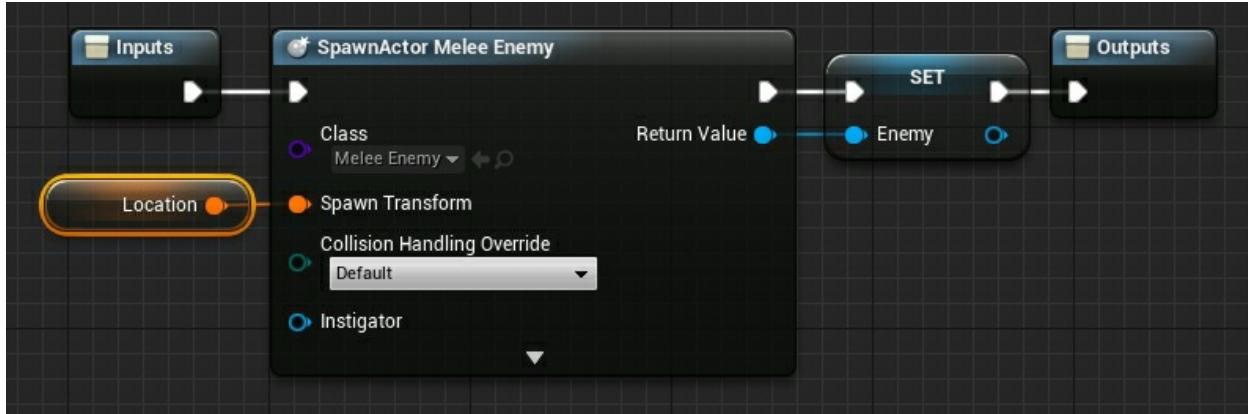


The melee enemy and the decorated projectile enemy are broken down into separate collapsed graphs for readability.

The *Melee\_Enemy\_Actions* collapsed graph:



The *Create\_Enemy* collapsed graph:



The *Enemy\_Fight* collapsed graph:



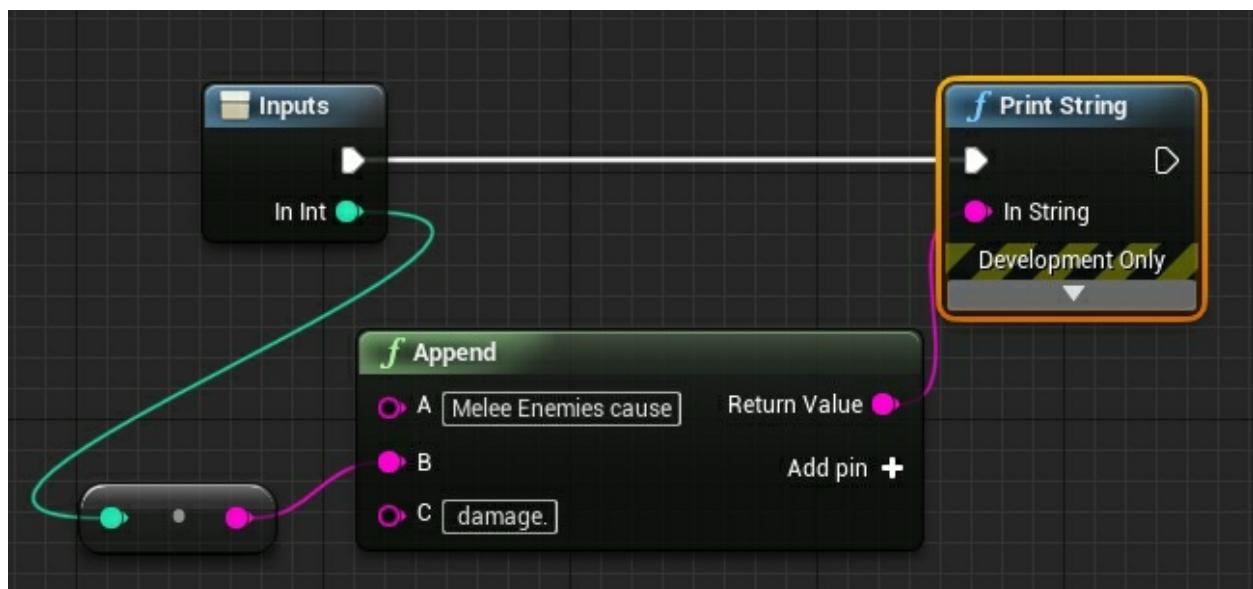
The *Enemy\_Die* collapsed graph:



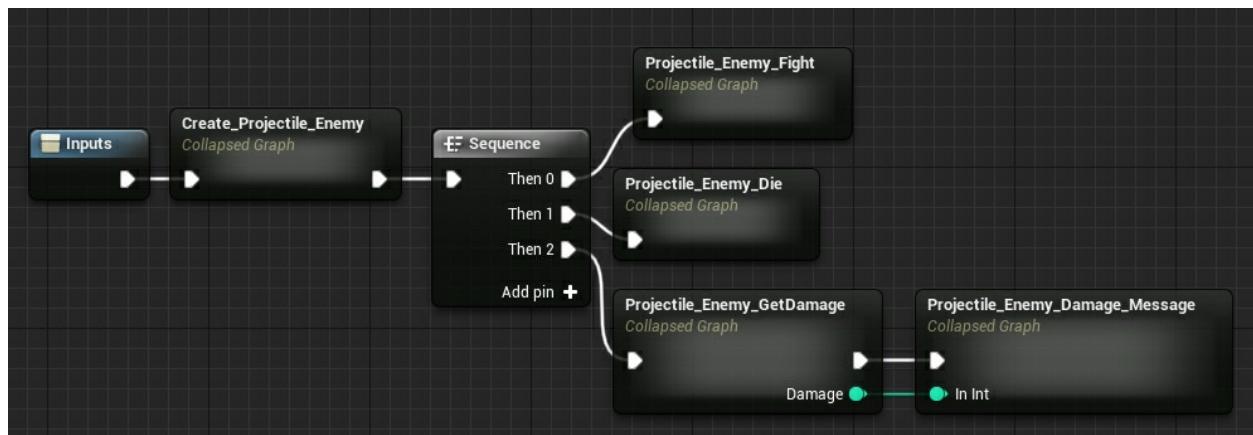
The *Enemy\_GetDamage* collapsed graph:



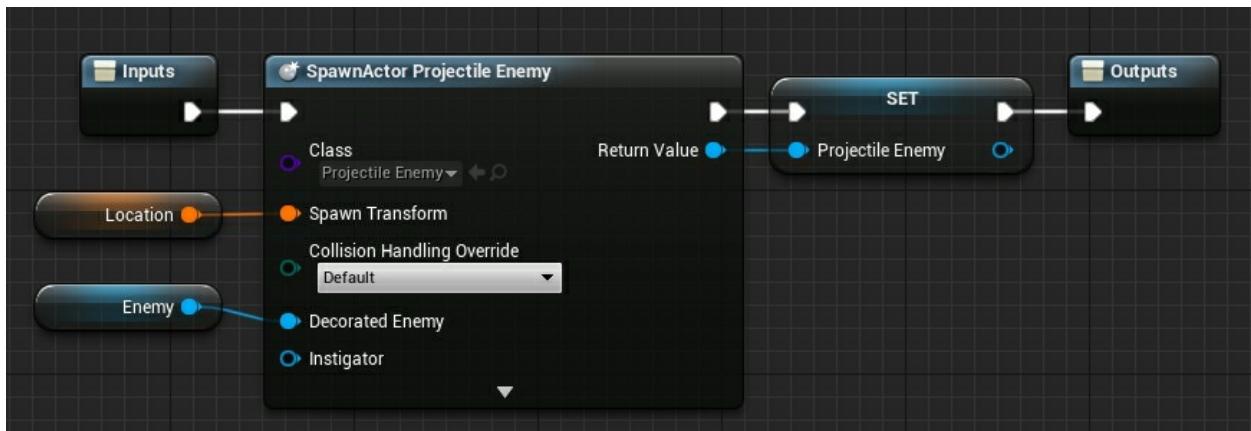
The *Enemy\_Damage\_Message* collapsed graph:



The *Projectile\_Enemy\_Actions* collapsed graph:



The *Create\_Projectile\_Enemy* collapsed graph:



The notable difference between the melee enemy and the projectile enemy is construction. The projectile enemy constructor accepts the previously created enemy as an argument. This is so we can reuse it to decorate an enemy object.

The *Projectile\_Enemy\_Fight* collapsed graph:



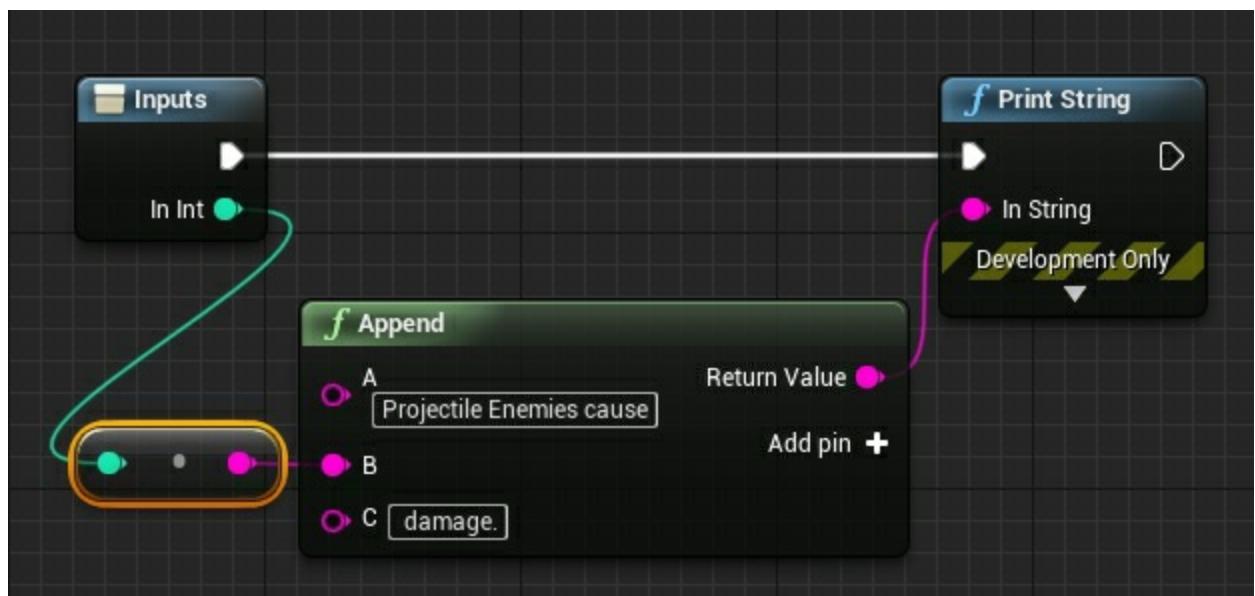
The *Projectile\_Enemy\_Die* collapsed graph:



The *Projectile\_Enemy\_GetDamage* collapsed graph:



The *Projectile\_Enemy\_Damage\_Message* collapsed graph:



The *Enemy* Interface:

```
public interface Enemy {  
    void fight();  
  
    //How much damage the enemy gives  
    int getDamage();  
  
    void die();
```

}

The *Enemy* interface is shared across all enemy objects

The *MeleeEnemy* class:

```
using System;
public class MeleeEnemy : Enemy {

    public void fight() {
        Console.WriteLine("The enemy throws heavy punches");
    }

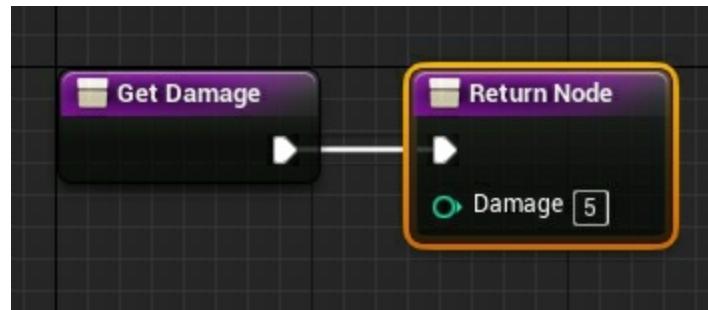
    public int getDamage() {
        return 5;
    }

    public void die() {
        Console.WriteLine("The enemy writhes in agony and disintegrates");
    }
}
```

The *MeleeEnemy* blueprint event graph:



The *GetDamage* function in the *MeleeEnemy* blueprint:



The *ProjectileEnemy* class:

```
using System;
public class ProjectileEnemy : Enemy {

    private Enemy decoratedEnemy;

    public ProjectileEnemy(Enemy decoratedEnemy) {
        this.decoratedEnemy = decoratedEnemy;
    }

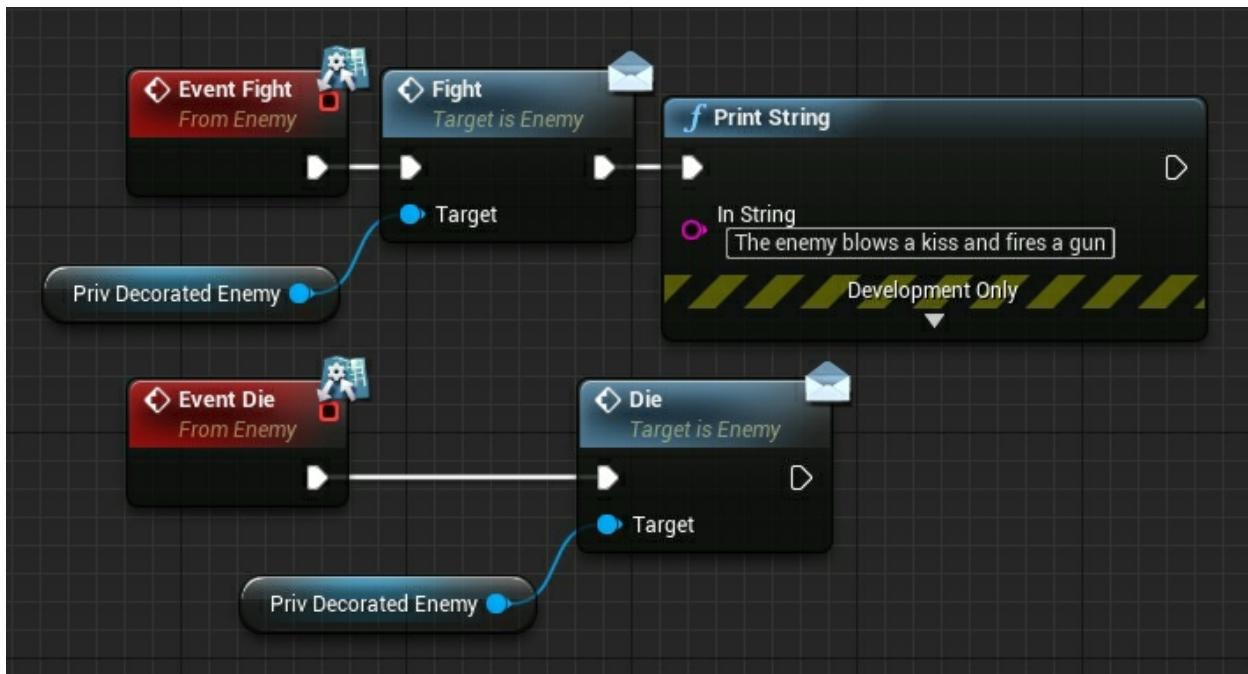
    public void fight() {
        decoratedEnemy.fight();
        Console.WriteLine("The enemy blows a kiss and fires a gun");
    }

    public int getDamage() {
        return decoratedEnemy.getDamage() + 95;
    }

    public void die() {
        decoratedEnemy.die();
    }
}
```

We can see that the projectile enemy causes more damage than the melee enemy. We use the melee damage causing attribute and we increase it by 95 for the projectile enemy.

The *ProjectileEnemy* blueprint event graph:



The *GetDamage* function in the *ProjectileEnemy* blueprint:



Decorator Pattern viewport print:

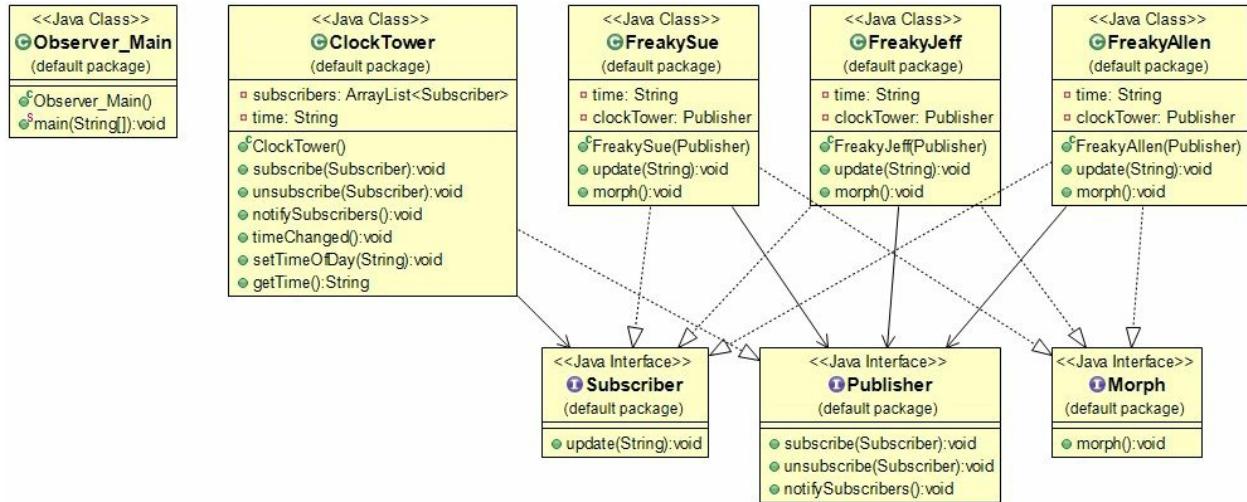
```
Projectile Enemies cause 100 damage.  
The enemy writhes in agony and disintegrates  
The enemy blows a kiss and fires a gun  
The enemy throws heavy punches  
MeleeEnemies Implements Enemy  
Melee Enemies cause 5 damage.  
The enemy writhes in agony and disintegrates  
The enemy throws heavy punches  
Melee Enemies are on the horizon
```

We can see the melee enemies coming. The melee enemy fights us with punches. The enemy then dies. However, we take a 5-count damage from the melee enemy. The melee enemy is then given new functionality in the form of a projectile. The enemy can punch and shoot a gun causing damage of 100.

# The Freaks Come out at night...

## Observer Pattern (C#)

Observer Pattern UML Diagram



## **Observer Pattern Implementation**

A clock tower is our publisher for the Observer pattern example. The subscribers consist of a group of freaks who change their behavior based on the time of day. The freaks receive notifications from the clock tower when the clock towers time state has changed. The *Observer\_Main* class:

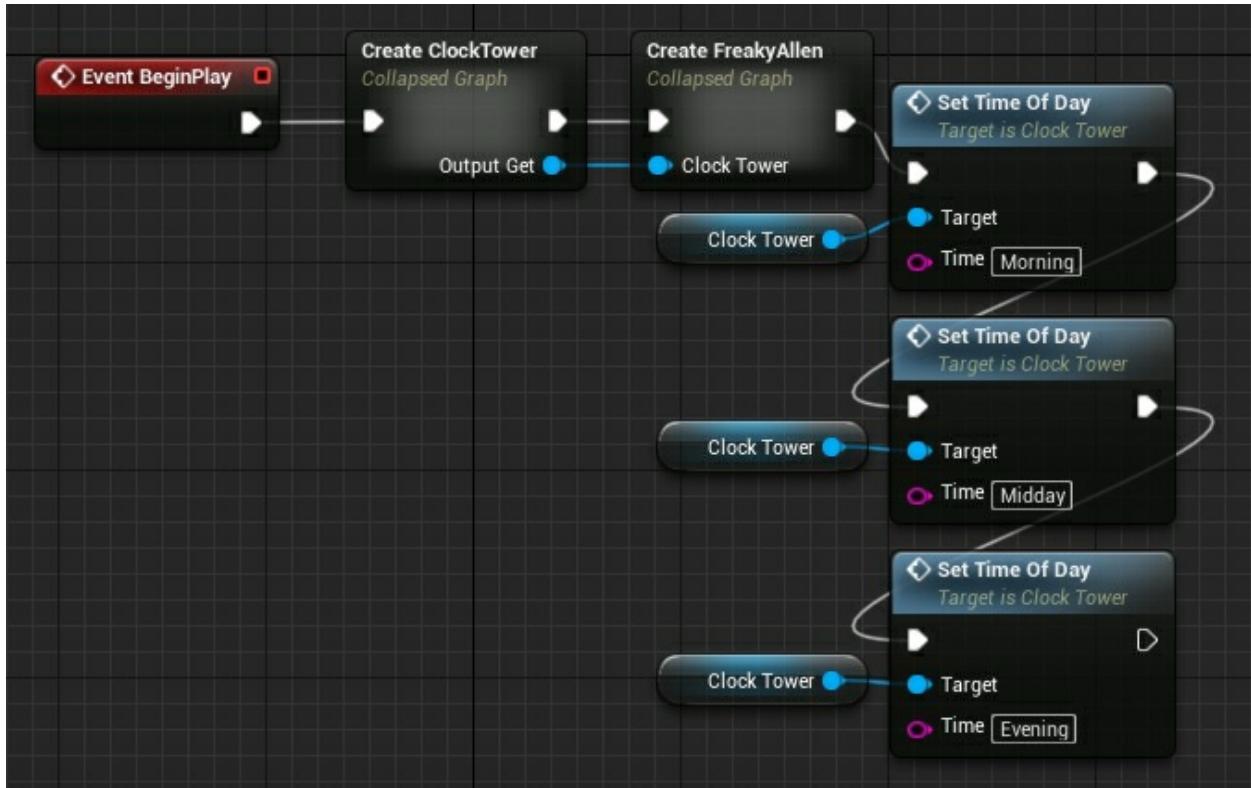
```
using System;

namespace Observer
{
    class Observer_Main
    {
        static void Main(string[] args)
        {
            ClockTower clockTower = new ClockTower();

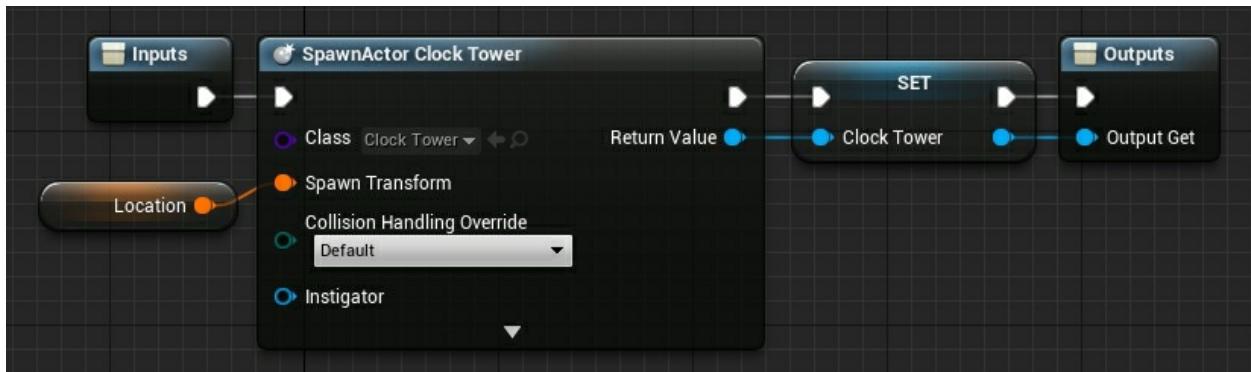
            FreakyAllen freakyAllen = new FreakyAllen(clockTower);
            FreakySue freakySue = new FreakySue(clockTower);
            FreakyJeff freakyJeff = new FreakyJeff(clockTower);

            clockTower.setTimeOfDay("Morning");
            clockTower.setTimeOfDay("Midday");
            clockTower.setTimeOfDay("Evening");
        }
    }
}
```

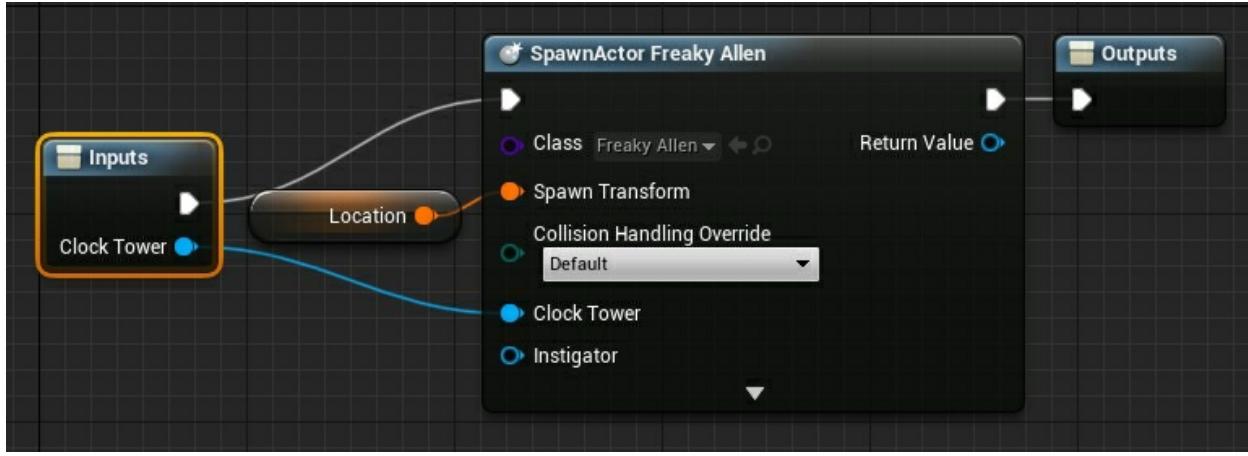
The *Observer\_Main* blueprint event graph:



The *Create ClockTower* collapsed graph:



The *Create FreakyAllen* collapsed graph:



Let's implement the *Subscriber* interface:  
**using System;**

```
public interface Subscriber {
    public void update(String time);
}
```

Let's implement the *Publisher* interface:  
**public interface** Publisher {
 **public void** subscribe(Subscriber s);
 **public void** unsubscribe(Subscriber s);
 **public void** notifySubscribers();
}

The *ClockTower* class:

```
using System;
using System.Collections.Generic;

public class ClockTower : Publisher {
    private List<Subscriber> subscribers;
    private String time;

    public ClockTower() {
        subscribers = new List<Subscriber>();
    }

    public void subscribe(Subscriber s) {
        subscribers.Add(s);
    }

    public void unsubscribe(Subscriber s) {
        int i = subscribers.IndexOf(s);
        if (i >= 0) {
            subscribers.RemoveAt(i);
        }
    }

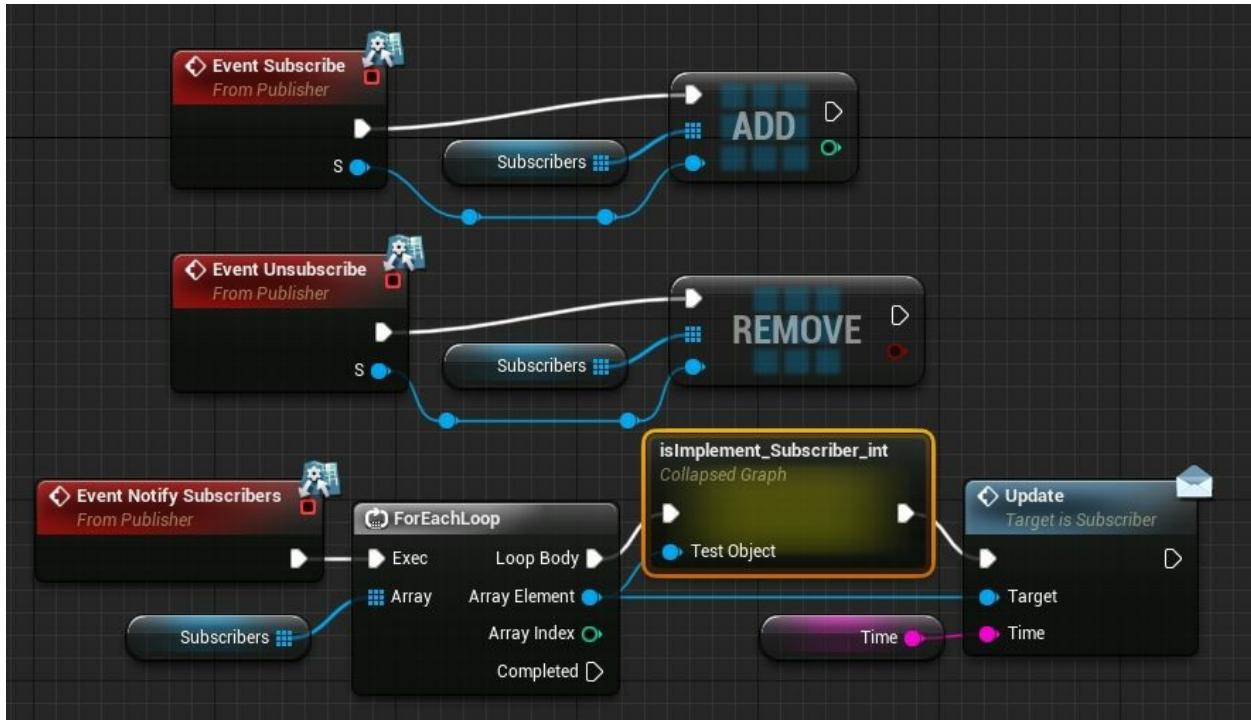
    public void notifySubscribers() {
        foreach (Subscriber subscriber in subscribers) {
            subscriber.update(time);
        }
    }

    public void timeChanged() {
        notifySubscribers();
    }

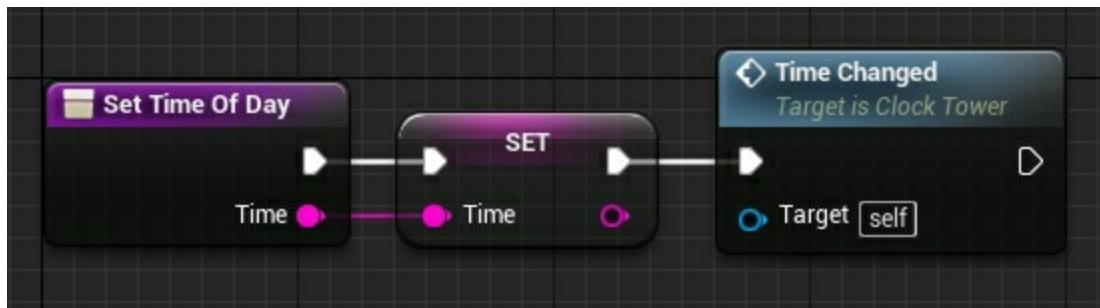
    public void setTimeOfDay(String time) {
        this.time = time;
        timeChanged();
    }
}
```

```
public String getTime() {  
    return time;  
}  
  
}
```

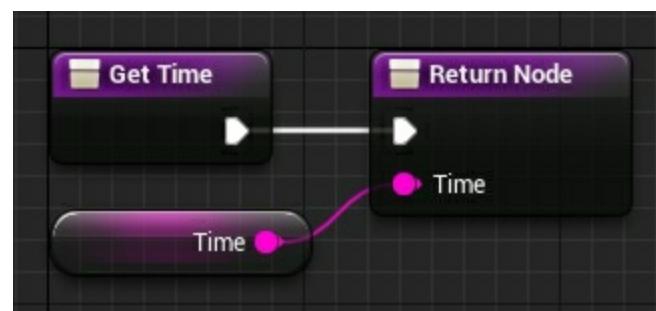
The *ClockTower* blueprint event graph:



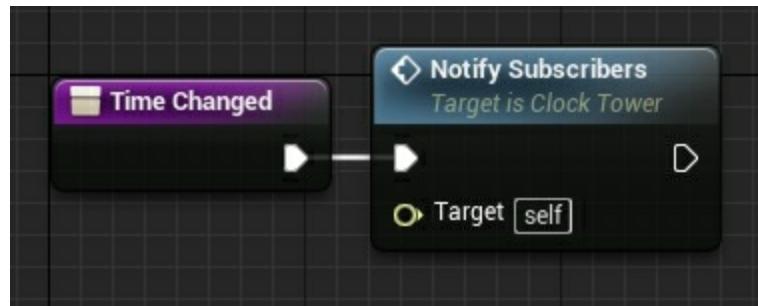
The *setTimeOfDay* function:



The *getTime* function:



The *timeChanged* function:



Let's implement the *Morph* interface our freaks will all share:

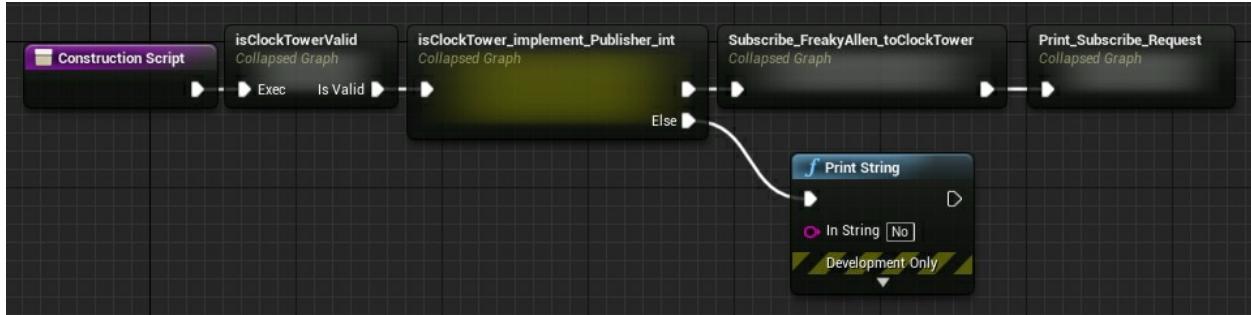
```
public interface Morph {  
    public void morph();  
}
```

The *FreakyAllen* class:

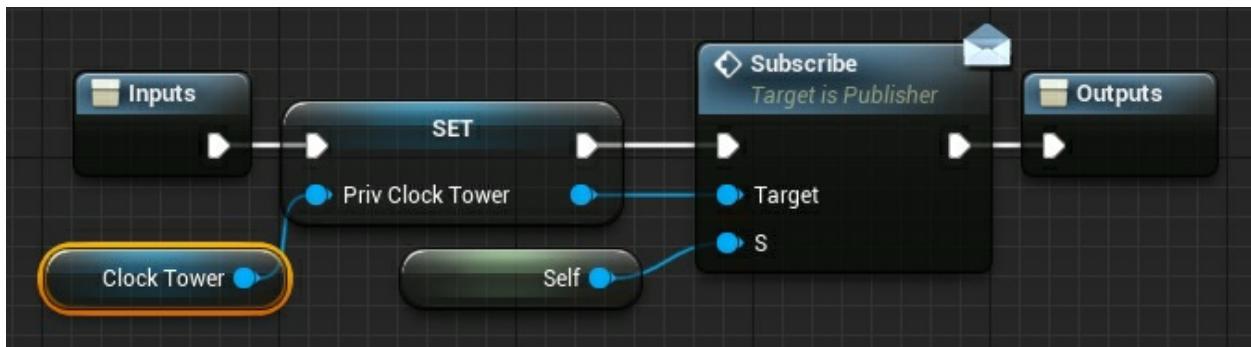
**using** System;

```
public class FreakyAllen : Subscriber, Morph {  
    private String time;  
    private Publisher clockTower;  
  
    public FreakyAllen(Publisher clockTower) {  
        this.clockTower = clockTower;  
        clockTower.subscribe(this);  
    }  
  
    public void update(String time) {  
        this.time = time;  
        morph();  
    }  
  
    public void morph() {  
        if (time.Equals("Morning")) {  
            Console.WriteLine("It is " + time  
                + ", so FreakyAllen makes a bowl of cereal");  
        } else if (time.Equals("Midday")) {  
            Console.WriteLine("It is: " + time  
                + ", so FreakyAllen's right eye starts to twitch");  
        } else if (time.Equals("Evening")) {  
            Console.WriteLine("It is: " + time  
                + ", so FreakyAllen morphs into a blood sucking wogglesnort");  
        }  
    }  
}
```

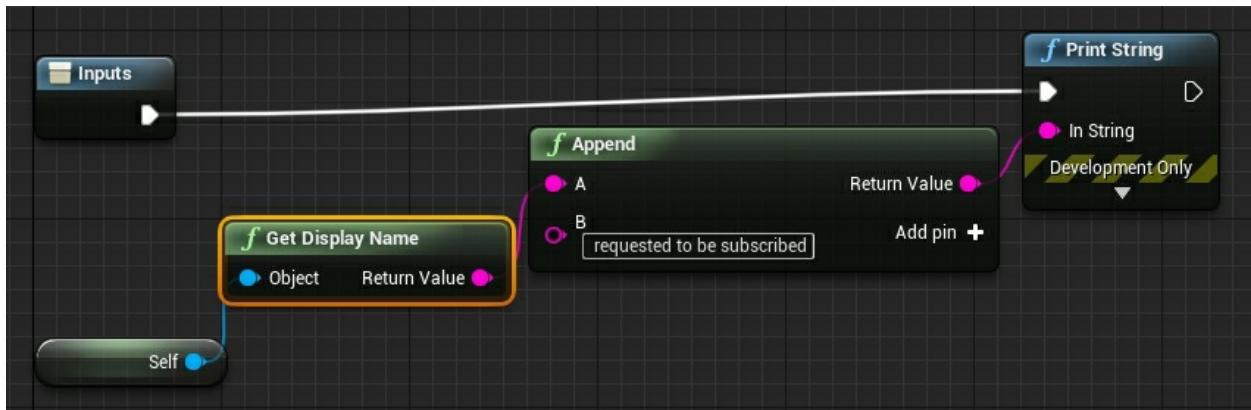
The *FreakyAllen* construction script:



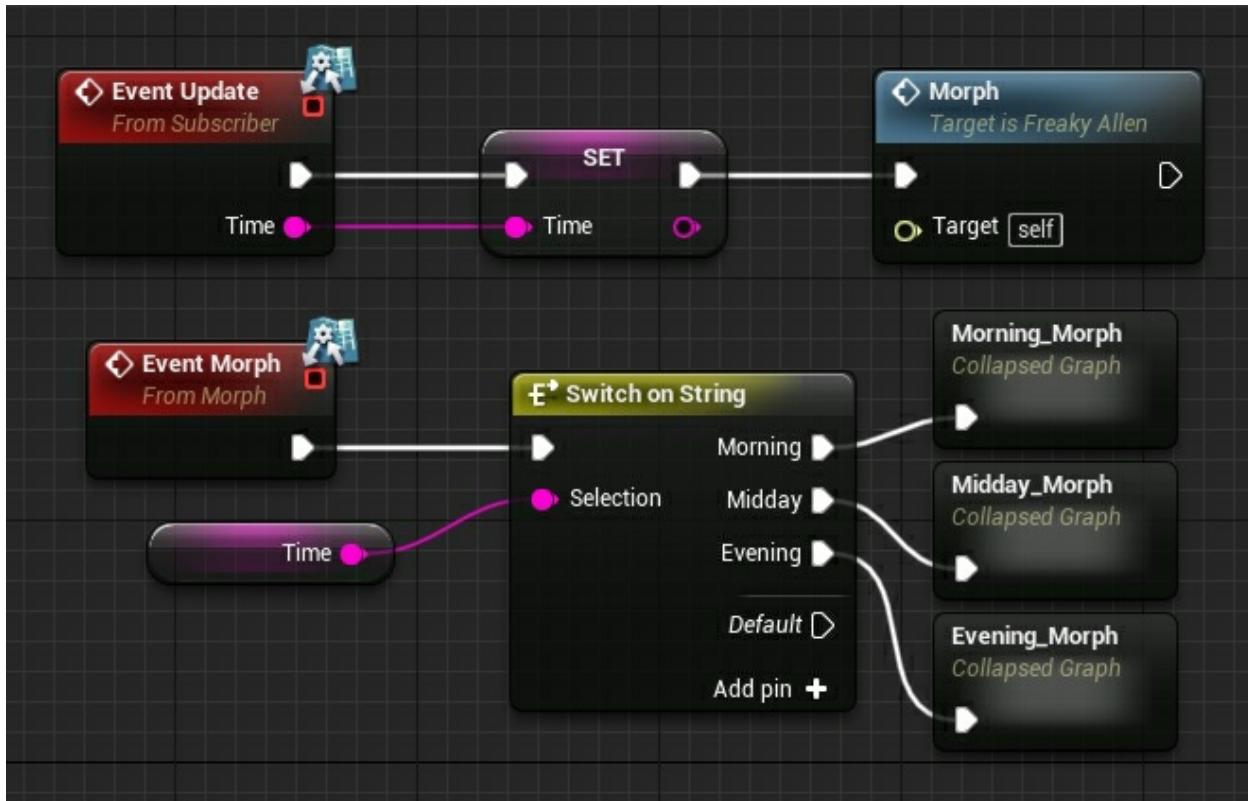
The *Subscribe\_FreakyAllen\_toClockTower* collapsed graph:



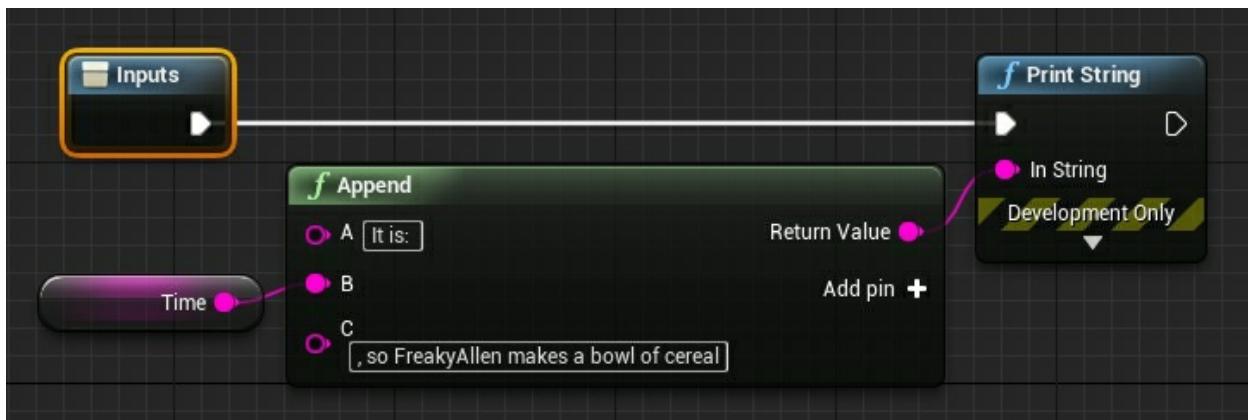
The *Print\_Subscribe\_Request* collapsed graph:



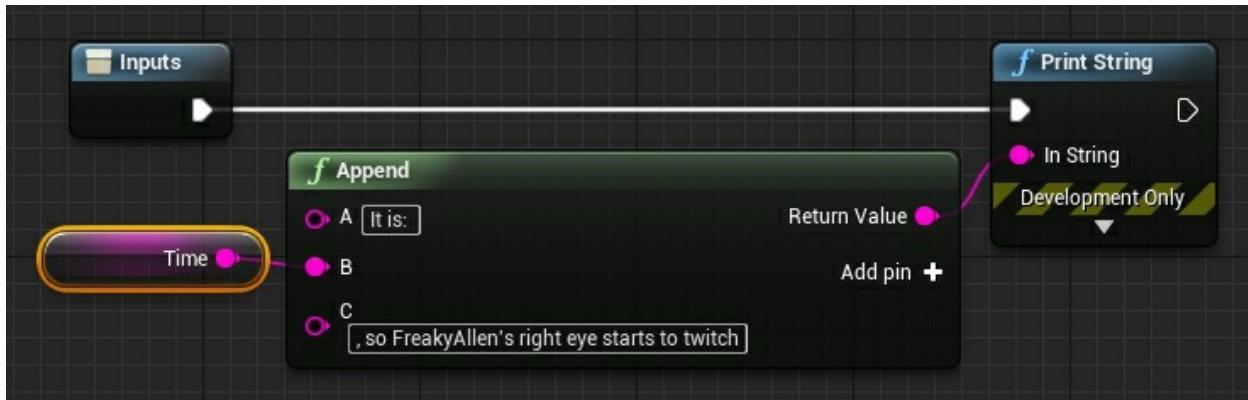
The *FreakyAllen* event graph:



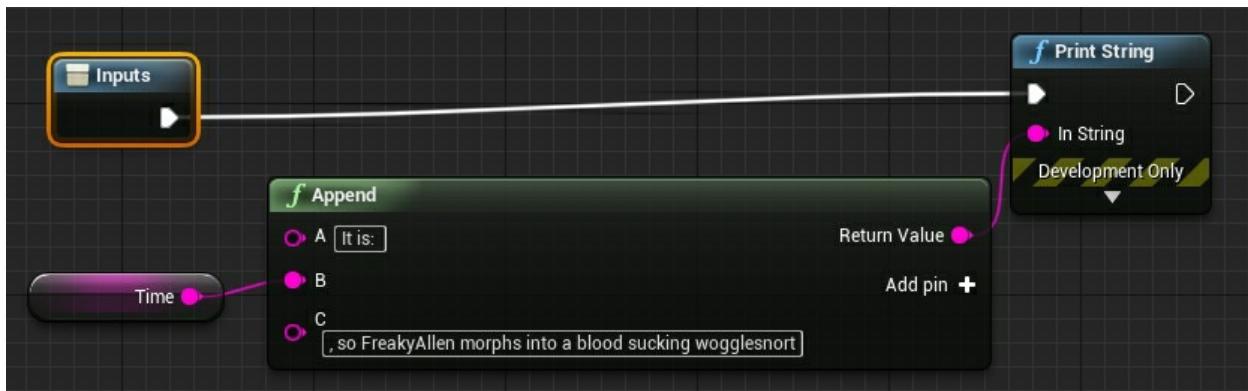
The *Morning\_Morph* collapsed graph:



The *Midday\_Morph* collapsed graph:

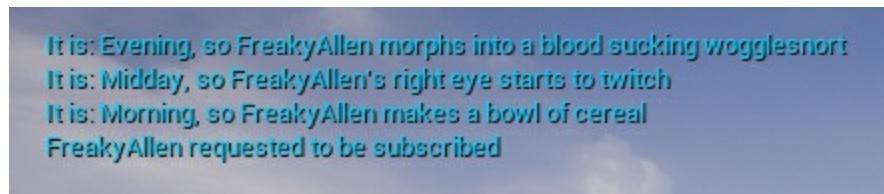


The *Evening\_Morph* collapsed graph:



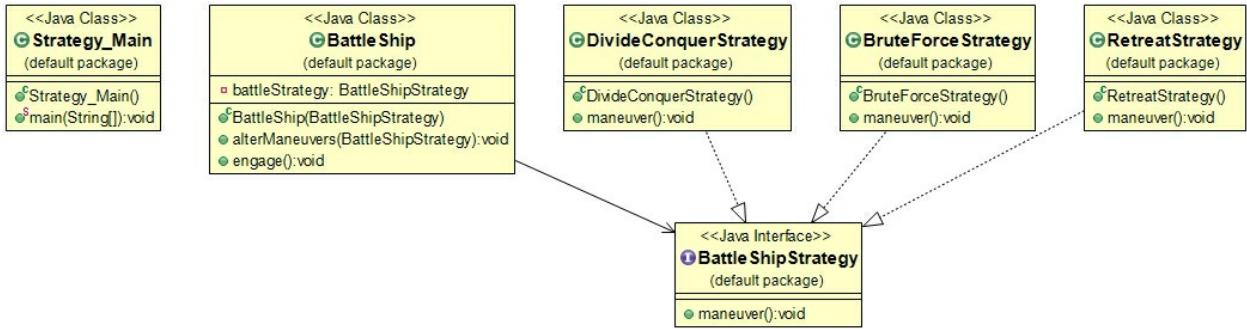
The other freaks follow the same implementation except for the strings that define their individual states and thus removed for brevity.

The Observer pattern viewport print:



# You Sunk My Battleship... Strategy Pattern (C#)

Strategy Pattern UML Diagram



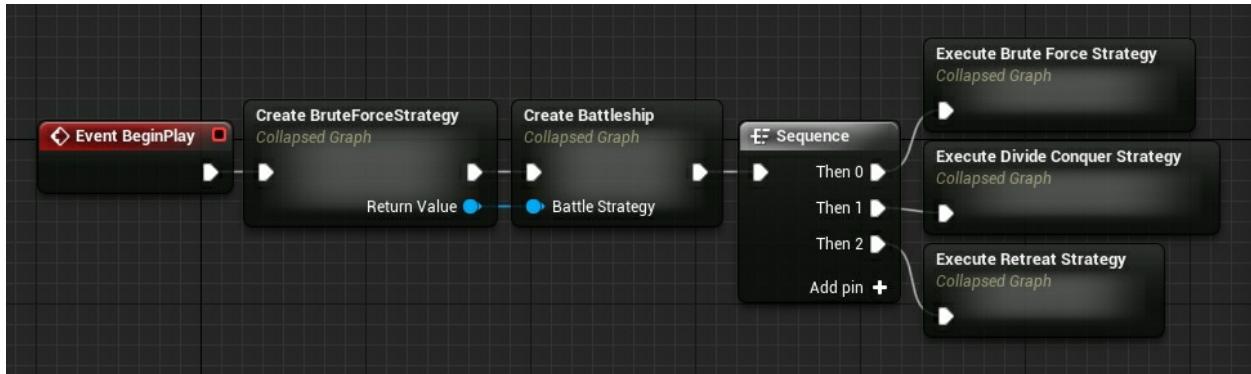
## **Strategy Pattern Implementation**

We have a strategy for combating different enemies at sea. The brute force strategy is used if the threat is small. We use the divide and conquer strategy if there are many enemies. The retreat strategy is executed if the battle is deemed futile. The *Strategy\_Main* class:

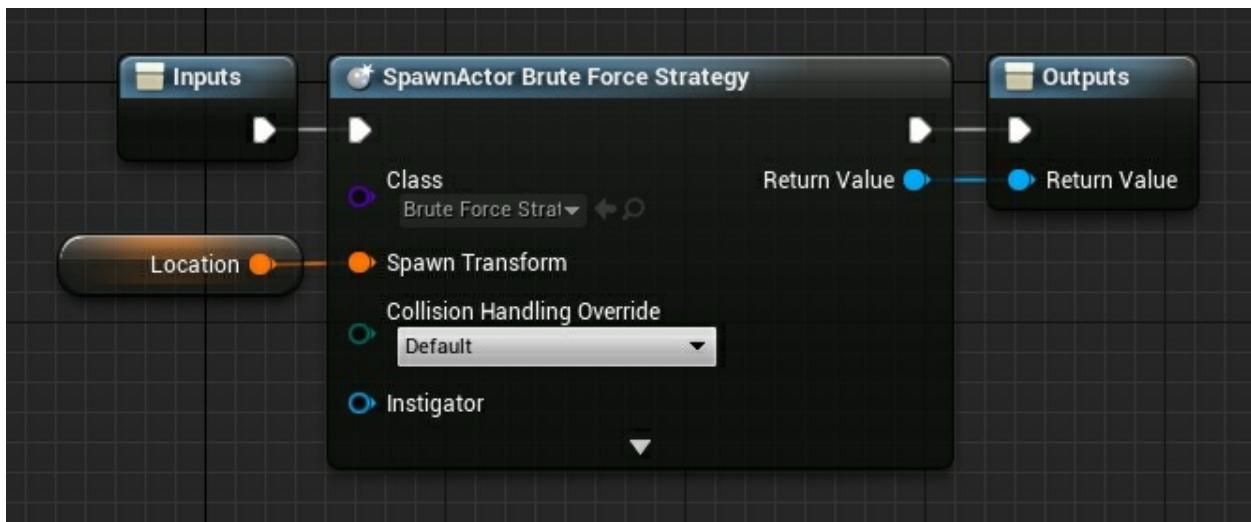
```
using System;

namespace Strategy
{
    class Strategy_Main
    {
        static void Main(string[] args)
        {
            Console.WriteLine("A tiny frigate wants some trouble");
            BattleShip battleShip = new BattleShip(new BruteForceStrategy());
            battleShip.engage();
            Console.WriteLine("Four tiny frigates want some trouble");
            battleShip.alterManeuvers(new DivideConquerStrategy());
            battleShip.engage();
            Console.WriteLine("An aircraft carrier group wants some trouble");
            battleShip.alterManeuvers(new RetreatStrategy());
            battleShip.engage();
        }
    }
}
```

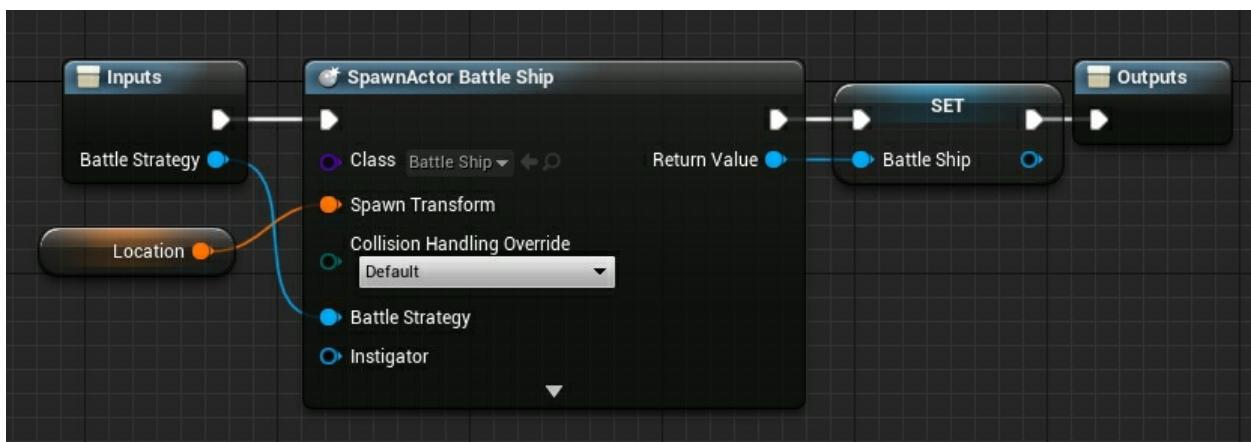
The *Strategy\_Main* blueprint event graph:



The *Create BruteForceStrategy* collapsed graph:



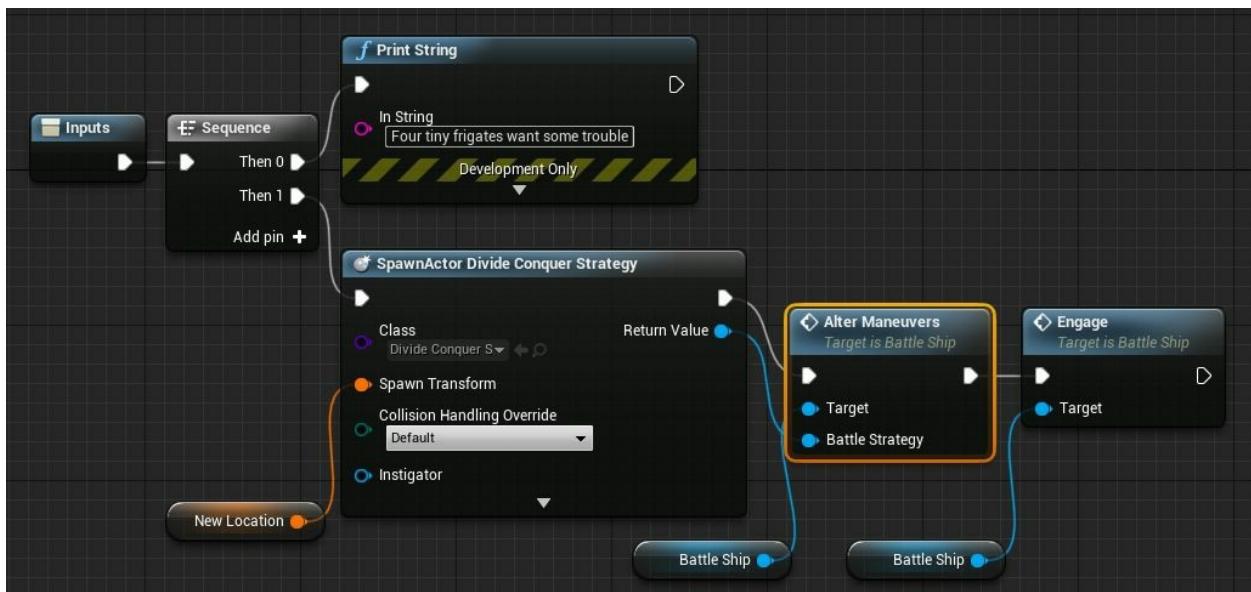
The *Create Battleship* collapsed graph:



The *Execute Brute Force Strategy* collapsed graph:



The *Execute Divide Conquer Strategy* collapsed graph:



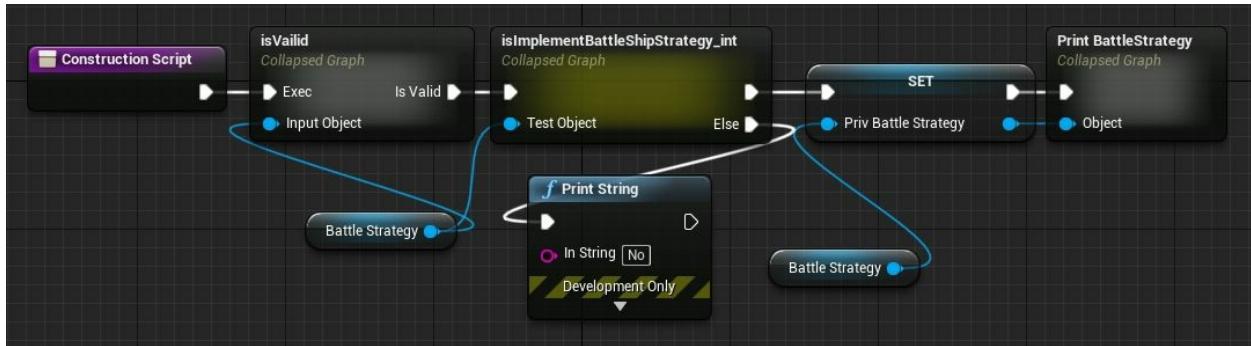
The *Execute Retreat Strategy* collapsed graph:



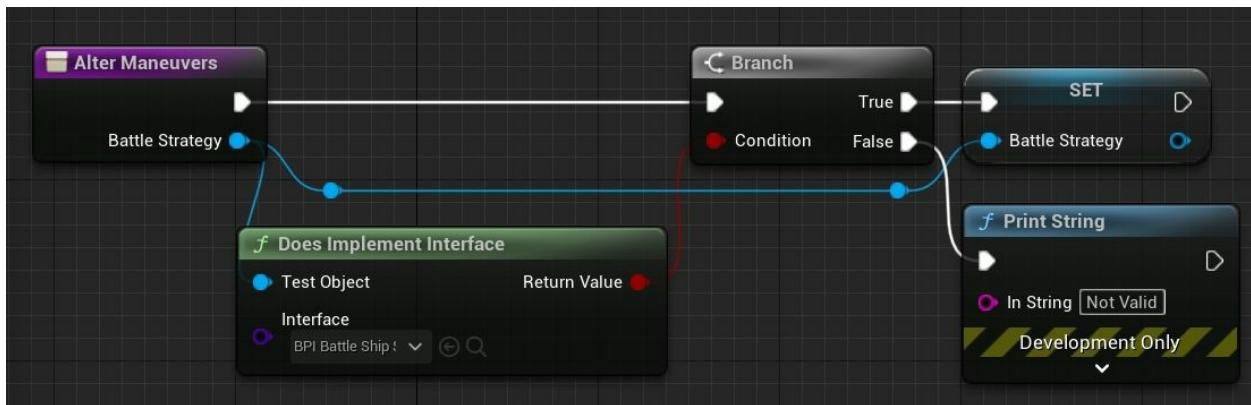
The *BattleShip* class:

```
public class BattleShip {  
  
    private BattleShipStrategy battleStrategy;  
  
    public BattleShip(BattleShipStrategy battleStrategy) {  
        this.battleStrategy = battleStrategy;  
    }  
  
    public void alterManeuvers(BattleShipStrategy battleStrategy) {  
        this.battleStrategy = battleStrategy;  
    }  
  
    public void engage() {  
        battleStrategy.maneuver();  
    }  
}
```

The *BattleShip* construction script:



The *alterManeuvers* blueprint function:



Is identical to the *BattleShip* construction with the exception of actually setting the *battleStrategy* variable manually inside the function because it is not exposed on spawn like.

The *engage* blueprint function:



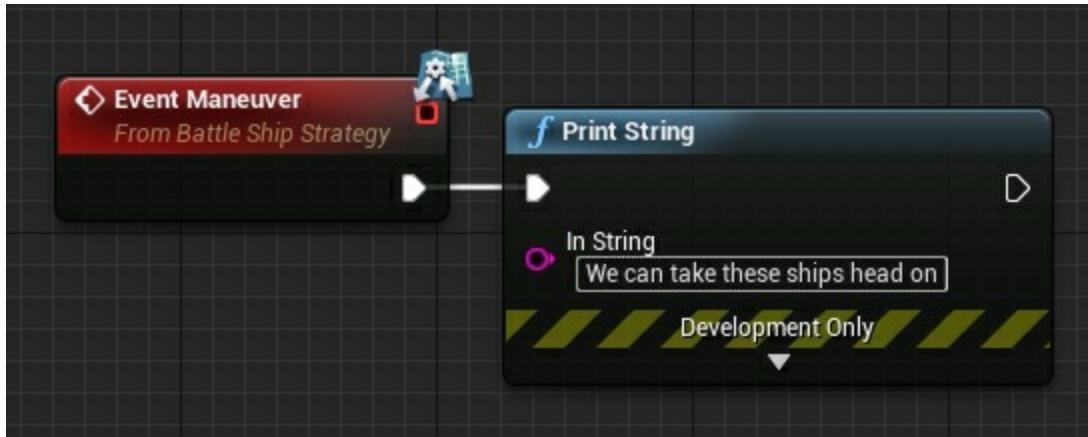
The *BattleShipStrategy* interface:

```
public interface BattleShipStrategy {  
    void maneuver();  
}
```

The *BruteForceStrategy* class:

```
using System;  
public class BruteForceStrategy : BattleShipStrategy {  
    public void maneuver() {  
        Console.WriteLine("We can take these ships head on");  
    }  
}
```

The *BruteForceStrategy* blueprint event graph:

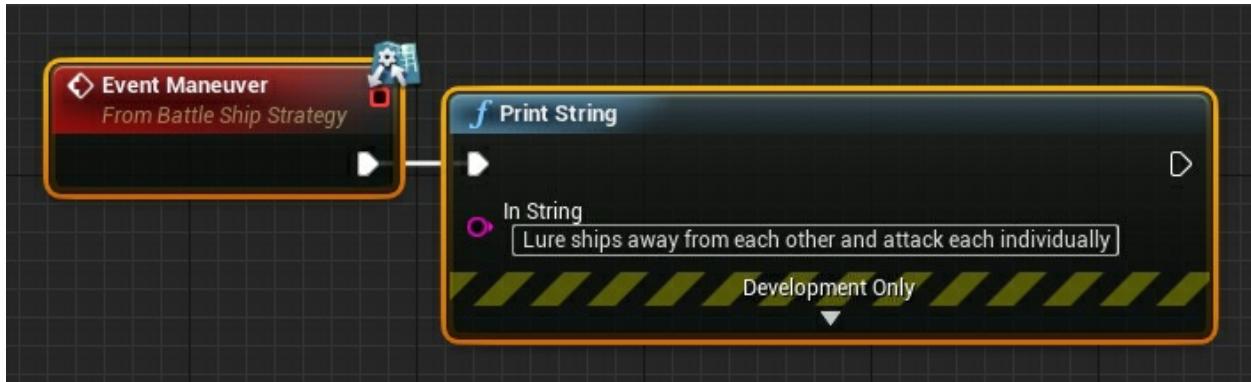


The *DivideConquerStrategy* class:

```
using System;
public class DivideConquerStrategy : BattleShipStrategy {

    public void maneuver() {
        Console.WriteLine("Lure ships away from each other and attack each
individually");
    }
}
```

The *DivideConquerStrategy* blueprint event graph:



The *RetreatStrategy* class:

```
using System;
public class RetreatStrategy : BattleShipStrategy {

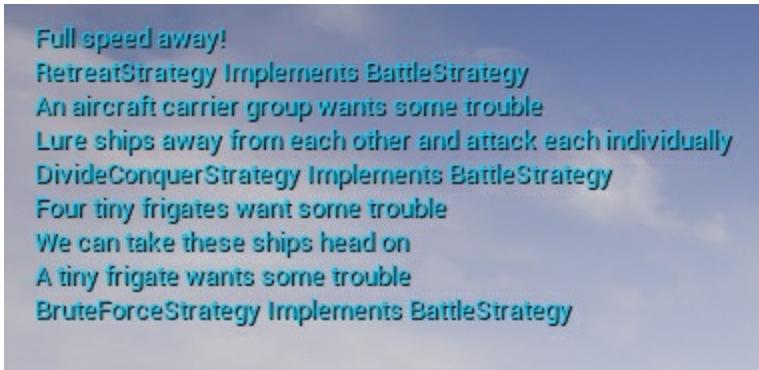
    public void maneuver() {
        Console.WriteLine("Full speed away!");
    }

}
```

The *RetreatStrategy* blueprint event graph:



The Strategy pattern viewport print:



Full speed away!  
RetreatStrategy Implements BattleStrategy  
An aircraft carrier group wants some trouble  
Lure ships away from each other and attack each individually  
DivideConquerStrategy Implements BattleStrategy  
Four tiny frigates want some trouble  
We can take these ships head on  
A tiny frigate wants some trouble  
BruteForceStrategy Implements BattleStrategy

# Conclusion

That's all folks! We journeyed down design pattern alley and came out the other side with blueprints. It was a joy to collaborate with you. Best of luck on your game development journey!



**THE  
END**

The text "THE END" is displayed in a bold, black, jagged font. The letters have irregular, dripping edges, giving them a distressed and edgy appearance. The text is centered within a circular frame. This central circle is composed of three concentric rings. The innermost ring is white, the middle ring is a light beige or tan color, and the outermost ring is a darker, muted brown or tan. The entire graphic is set against a solid black background.

# Source

Design patterns: elements of reusable object-oriented software  
Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA ©1995  
ISBN:0-201-63361-2

## About the Author



**Sonicworkflow Creative Director - Umar Bradshaw**

Mr. Bradshaw holds an MBA and a master's in software engineering

from an ABET accredited institution. He has been on the indie scene approximately ten years and has worked across many different digital product lines during that time as well.

Contact:

Website - <https://sonicworkflow.com/>

Twitter - <https://twitter.com/sonicworkflow>

Instagram - <https://www.instagram.com/sonicworkflow/>

Discord - <https://discord.gg/qUsvcWA>