A Case Study

Architecting a Football Game App

Trainer: Venkat Shiva Reddy

The Scenario

- You are working with a popular computer game developing company, and they made you the Solution Architect of one of their major projects - a Soccer (Football) Game Engine (Nice, huh?).

- Now, you are leading the process of designing the entire Football game engine.

as a Solution Architect…

- How you identify the use cases for your game system?
- How you identify the entities in your game system?
- How you identify the design problems?, and
- How you apply patterns to address your design specifications?

Football Game: Use Cases

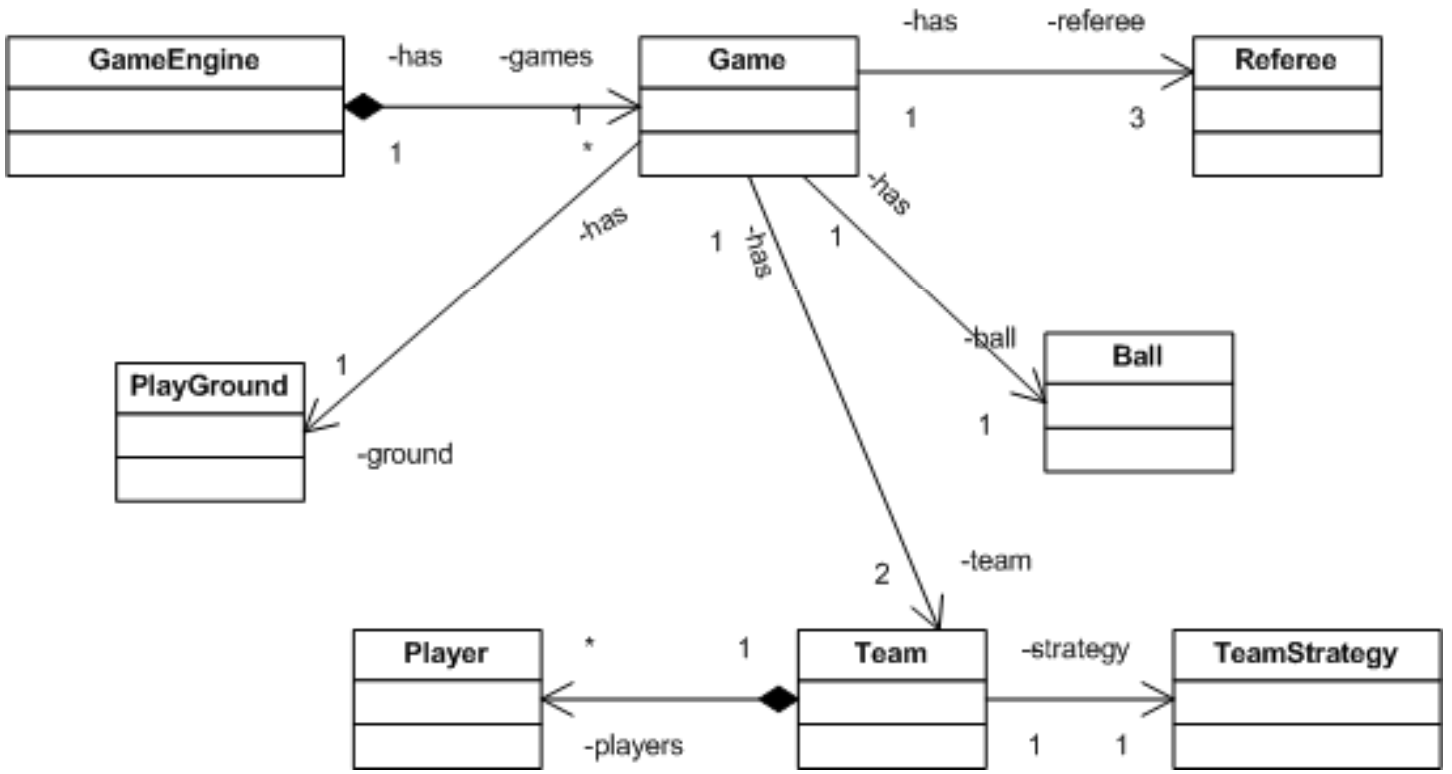
- Let us assume that the end user is going to operate the game in the following sequence *(let us keep things simple).*
  - Start the Game
  - Select two teams
  - Add or remove players to/from a team
  - Add or remove roles to/from the player
  - Set the team strategy
  - Pick a play ground
  - Play the game

Identifying Entities…

The Entities…

- **Player** who play the soccer
- **Team** with various players in it
- **Ball** which is handled by various players.
- **PlayGround** where the match takes place.
- **Referee** in the ground to control the game.
- Also, you may need some logical objects in your game engine, like
- **Game** which defines a football game, which constitutes teams, ball, referee, playground etc
- **GameEngine** to simulate a number of games at a time.
- **TeamStrategy** to decide a team's strategy while playing

Football Game: High Level View



Identifying Design Problems

- Now you should decide
  - How these objects are structured?
  - How they are created?
  - Their behaviour when they interact each other, to formulate the design specifications.

The design problems…

- **Ball**
  - Their should be only ONE ball used by all the players in the game
  - When the position of a ball changes, all the players and the referee should be notified straight away.
- **Team and TeamStrategy**
  - When the game is in progress, the end user can change the strategy of his team (E.g., From Attack to Defend)
- **Player**
  - A player in a team should have additional responsibilities, like Forward, Defender etc, that can be assigned during the runtime.
- **PlayGround**
  - Each ground constitutes of gallery, ground surface, audience, etc - and each ground has a different appearance.

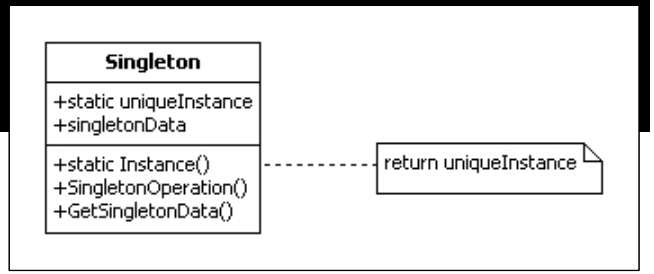
Identifying the Patterns to use

- Addressing the design problems related with the
  - Ball
  - Team and TeamStrategy
  - Player
  - PlayGround

Design problem - 1: Ball

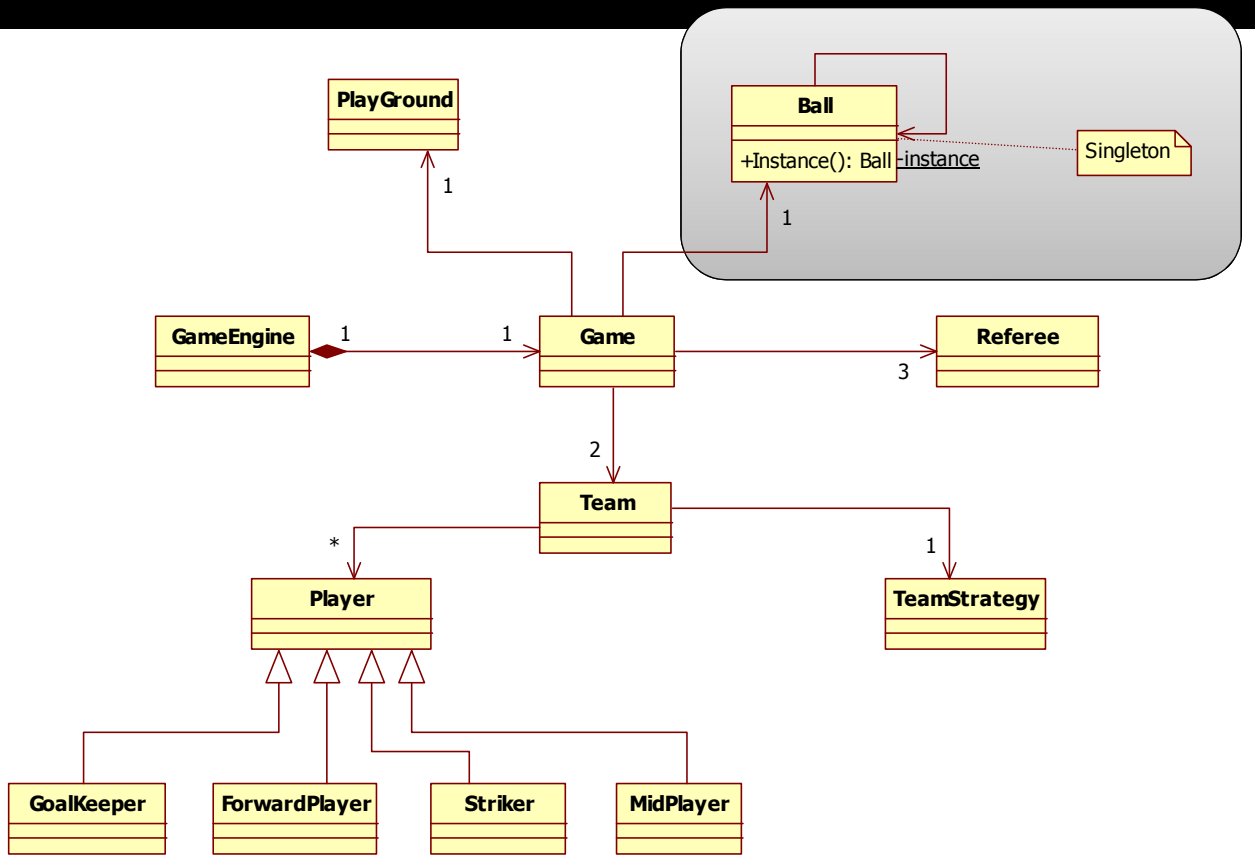
- Specific Design Problem
  - "Their should be only ONE ball used by all the players in the game"
- Problem Generalized
  - Exactly one object (the ball) is needed to coordinate actions across the system.
- The Design Pattern:
  - Singleton: Ensure that a class has only one instance, and provide a global point of access to it.

The Singleton Pattern



- **Purpose**
  - Ensure a class only has one instance,
    and provide a global point of access to it.
- **Applications**
  - There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
  - When the sole instance should be extensible by sub-classing, and clients should be able to use an extended instance without modifying their code.
- **Consequences**
  - **Controlled access to sole instance.** Because the Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it.
  - **Reduced namespace.** The Singleton pattern is an improvement over global variables. It avoids polluting the name space with global variables that store sole instances.
  - **Permits refinement of operations and representation.** The Singleton class may be sub-classed, and it's easy to configure an application with an instance of this extended class. You can configure the application with an instance of the class you need at run-time.
  - **Permits a variable number of instances.** The pattern makes it easy to change your mind and allow more than one instance of the Singleton class. Moreover, you can use the same approach to control the number of instances that the application uses. Only the operation that grants access to the Singleton instance needs to change.

Applying Singleton Pattern



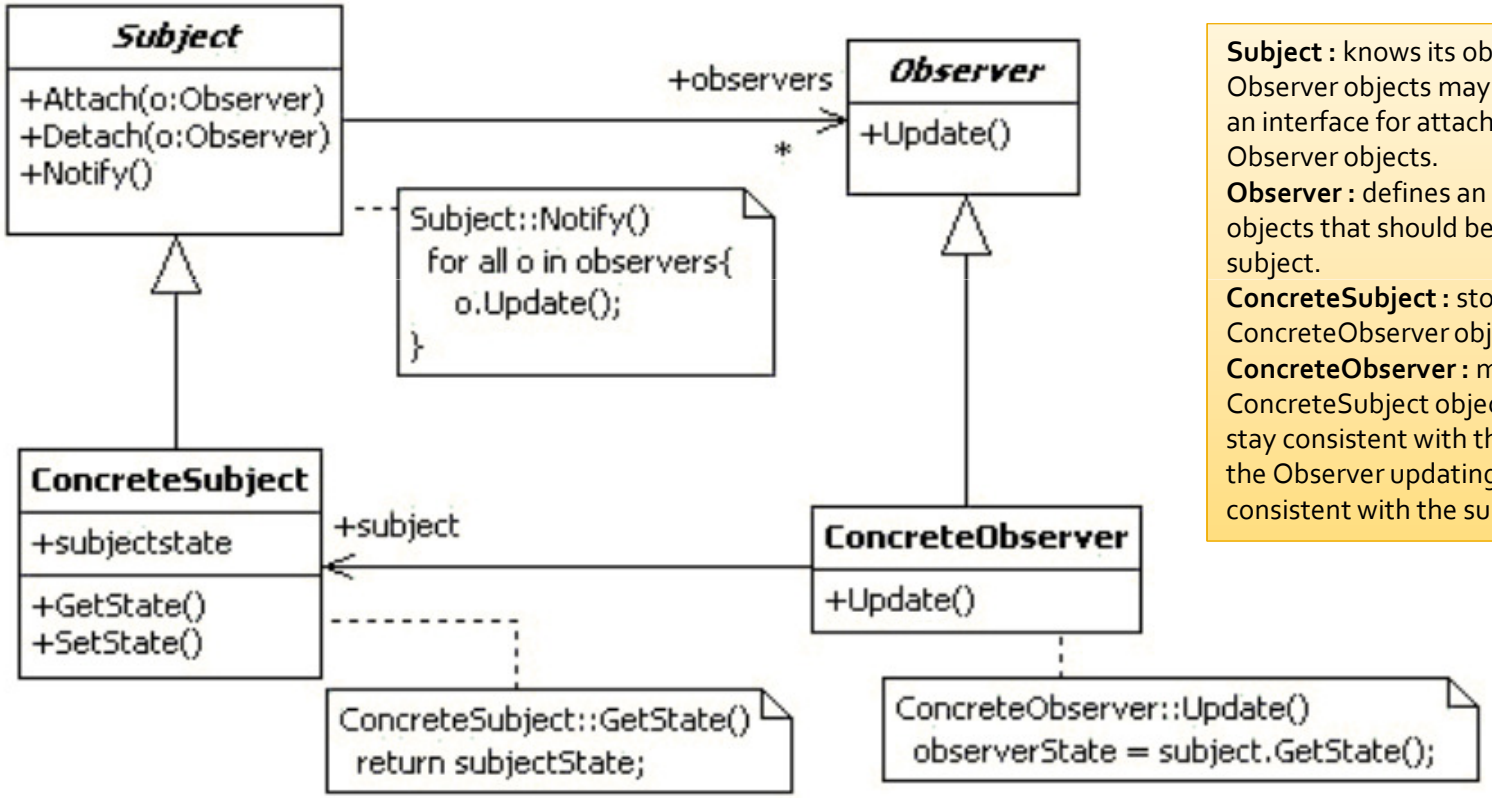
Design problem - 2 : Ball

- ***Specific Design Problem:*** *"When the position of a ball changes, all the players and the referee should be notified straight away."*
- ***Problem Generalized:*** *"When a subject (in this case, the ball) changes, all its dependents (in this case, the players) are notified and updated automatically."*
- **Observer Pattern:** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

Observer Pattern

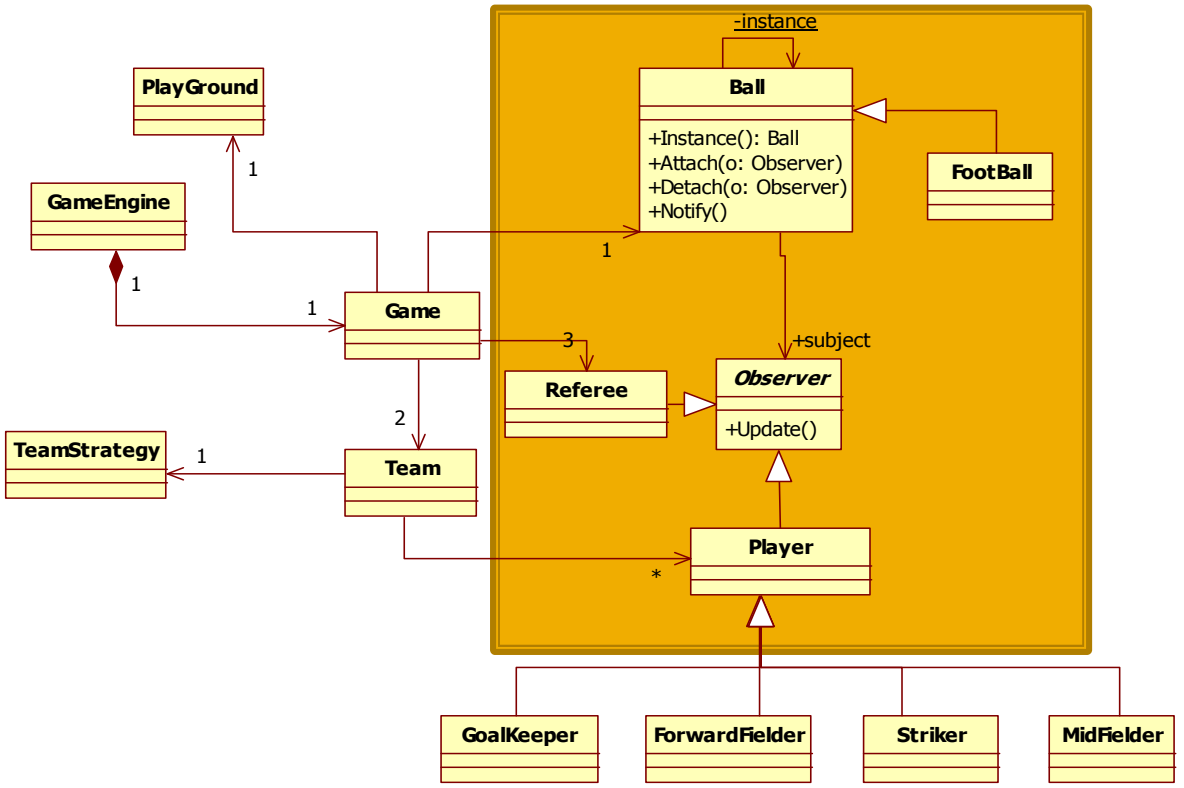
- Purpose
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Observer Pattern: Structure



Subject : knows its observers. Any number of Observer objects may observe a subject. provides an interface for attaching and detaching Observer objects.
Observer : defines an updating interface for objects that should be notified of changes in a subject.
ConcreteSubject : stores state of interest to ConcreteObserver objects.
ConcreteObserver : maintains a reference to a ConcreteSubject object. stores state that should stay consistent with the subject's. implements the Observer updating interface to keep its state consistent with the subject's.

Applying Observer Pattern



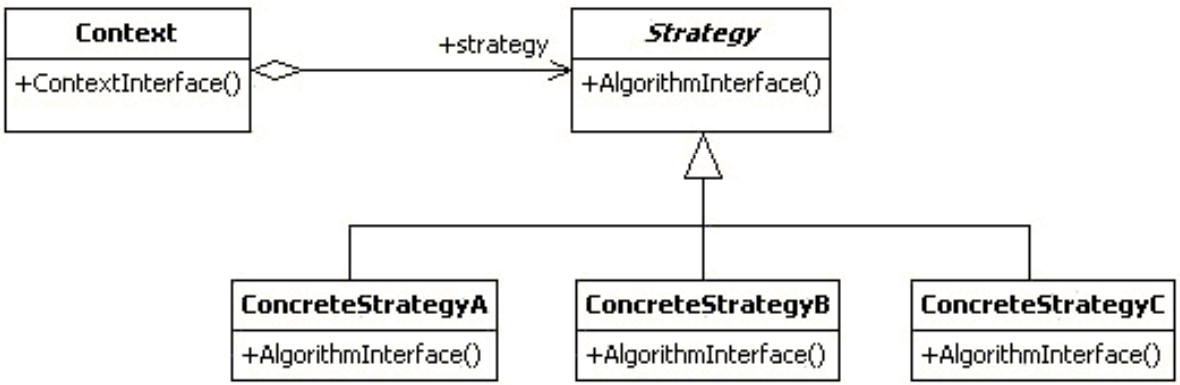
Design Problem: Team & Team Strategy

- ***Specific Design Problem:*** *"When the game is in progress, the end user can change the strategy of his team (E.g., From Attack to Defend)"*
- ***Problem Generalized: "****We need to let the algorithm (TeamStrategy) vary independently from clients (in this case, the Team) that use it."*
- **Strategy Pattern:** Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it

Strategy Pattern

- Purpose
  - Define a family of algorithms, encapsulate each one, and make them interchangeable.
  - Strategy lets the algorithm vary independently from clients that use it.

Strategy Pattern

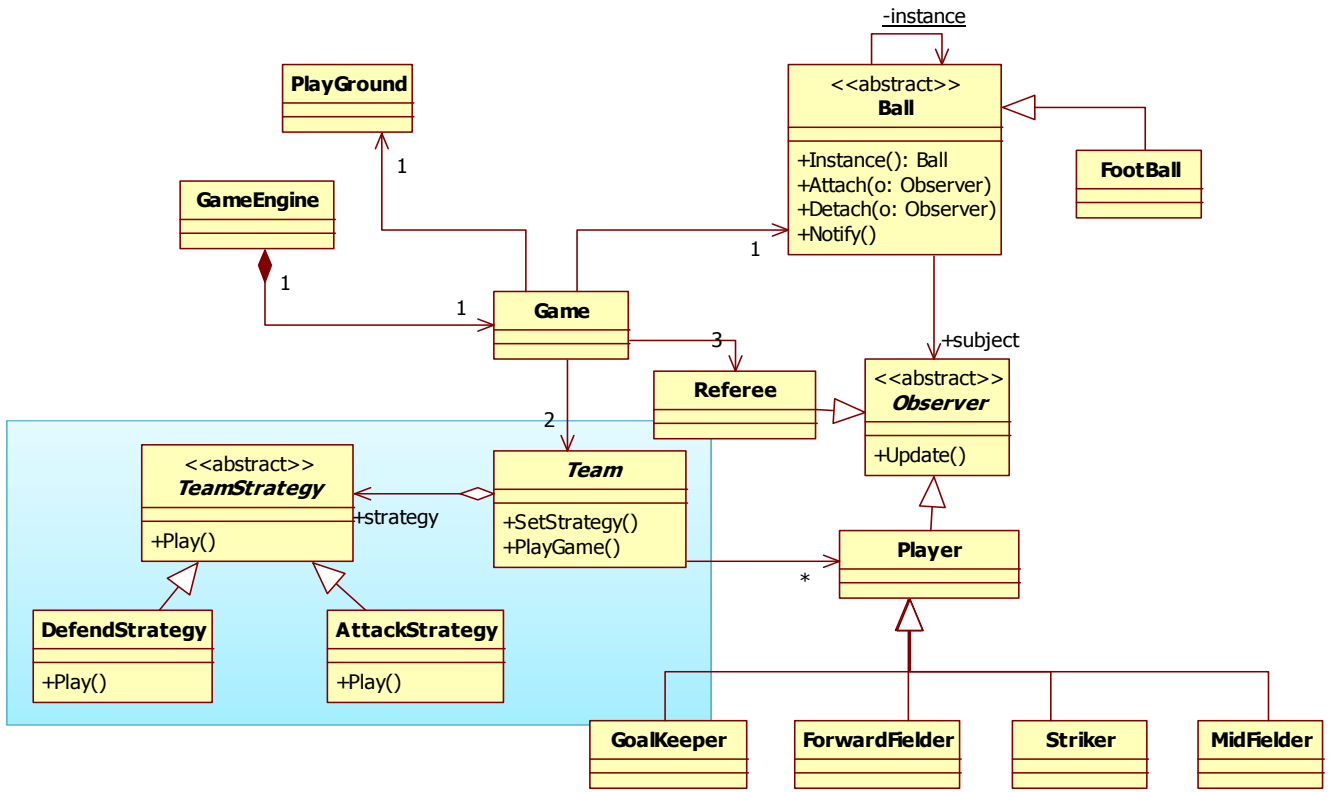


Strategy : declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.

ConcreteStrategy : implements the algorithm using the Strategy interface.

Context : is configured with a ConcreteStrategy object. maintains a reference to a Strategy object. may define an interface that lets Strategy access its data.

Applying Strategy Pattern



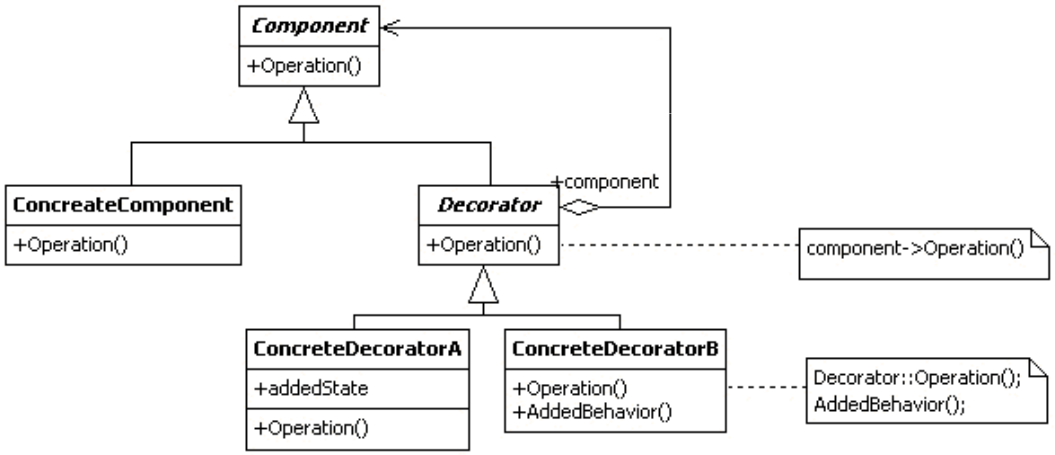
Design Problem: Player

- ***Specific Design Problem:*** *"A player in a team should have additional responsibilities, like Forward, Defender etc, that can be assigned during the runtime."*
- ***Problem Generalized: "****We need to attach additional responsibilities (like Forward, Midfielder etc) to the object (In this case, the Player) dynamically, with out using sub classing*"
- ***Decorator Pattern:*** Add responsibilities to objects dynamically.

Decorator Pattern

- Purpose
  - Attach additional responsibilities to an object dynamically.
  - Decorators provide a flexible alternative to sub-classing for extending functionality.

Decorator Pattern

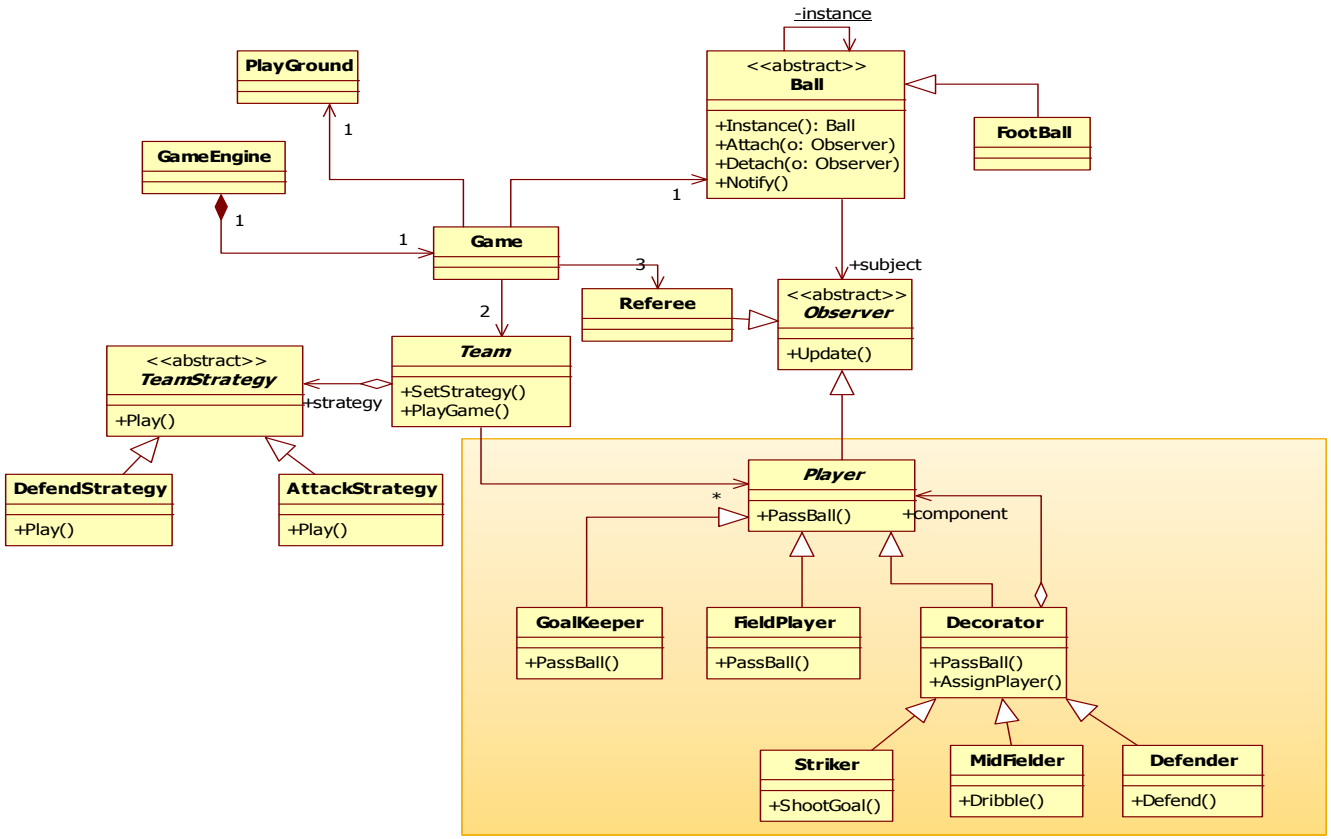


Component : defines the interface for objects that can have responsibilities added to them dynamically.

Decorator : maintains a reference to a Component object and defines an interface that conforms to Component's interface.

ConcreteDecorator : adds responsibilities to the component.

Applying Decorator Pattern



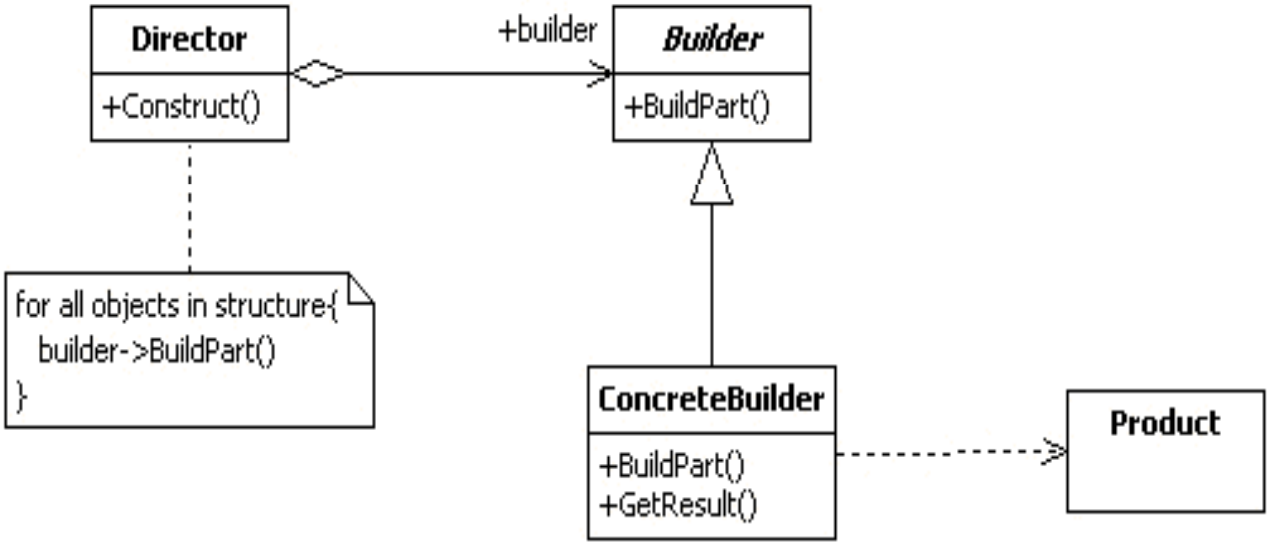
Design Problem: The Ground

- **Specific Design Problem:** Each ground constitutes of gallery, ground surface, audience, etc - and each ground has a different appearance.
- ***Problem Generalized:*** "Separate the construction of a complex object (The Ground) from its representation (gallery, audience) so that the same construction process can create different representations."
- ***Builder Pattern:*** Specify the parameters that should be used to create the complex object and the builder will take care of building the complex object

Builder Pattern

- Purpose:
  - Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Builder Pattern : Structure



Builder : specifies an abstract interface for creating parts of a Product object.
ConcreteBuilder : Constructs and assembles parts of the product by implementing the Builder interface. Defines and keeps track of the representation it creates. Provides an interface for retrieving the product (e.g., GetASCIIText, GetTextWidget).
Director : Constructs an object using the Builder interface.
Product : Represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled.

Applying Builder Pattern