Architecture

Prerequisites

- C Programming
- C++ Object Oriented Programming
  - Classes
  - Inheritance
  - Virtual Functions

Architecture

- Architecture is the organizational structure of a system, including its decomposition into parts, their connectivity, interaction mechanisms, and the guiding principles and decisions that you use in the design of the system.

Architecture Principles

- Same for all languages
- Fundamental, not just a strategy for implementation
- These are the guiding principles on which all design patterns are built

Architecture Principle #1

Simplify.

Simplicity

- **Architecture is simplicity**
- "Everything should be made as simple as possible, but not simpler." – Albert Einstein
- The job of the architect is to make all the other programmers' job simple.
- Do not build complexity for complexity's sake.

Simplify Code

- Removing Redundancy(File IO, Cross Platform)
- Providing a common interaction system (Messaging)
- Standardizing certain problem solutions
    - Examples:
        - Singleton
        - Function input parameters (constness, inline, exception, macros…)
        - STL

Architecture Principle #2

Embrace change.

Embrace Change

- Plan for change
- Build for change
- Develop good enough
- Maximize flexibility while maintaining simplicity
- Data drive functionality
- Do not waste time building what may not be needed
- Iteration Wins

Architecture Principle #3

Organize by what it does.

Organize by what it does

- Code should be divided into functional atomic pieces.
- Also applies to systems, code files, etc.
- However, simplicity and flexibility are more important.
- Not by what it IS.

Data Oriented

- Think in terms of functionality
- Code Transforms sets of data
- One class one responsibility. Also called the Single Responsibility Principle (SRP)

Architecture Principle #4

Encapsulate what varies.

Encapsulate variability

- Program to an abstraction not an implementation.
- Move the code and responsibility inside abstracted objects.
- In C++, delegate varying behavior to the abstracted object, such as serialization, loading, updating, etc.

Architecture Principle #5

Minimize Dependencies.

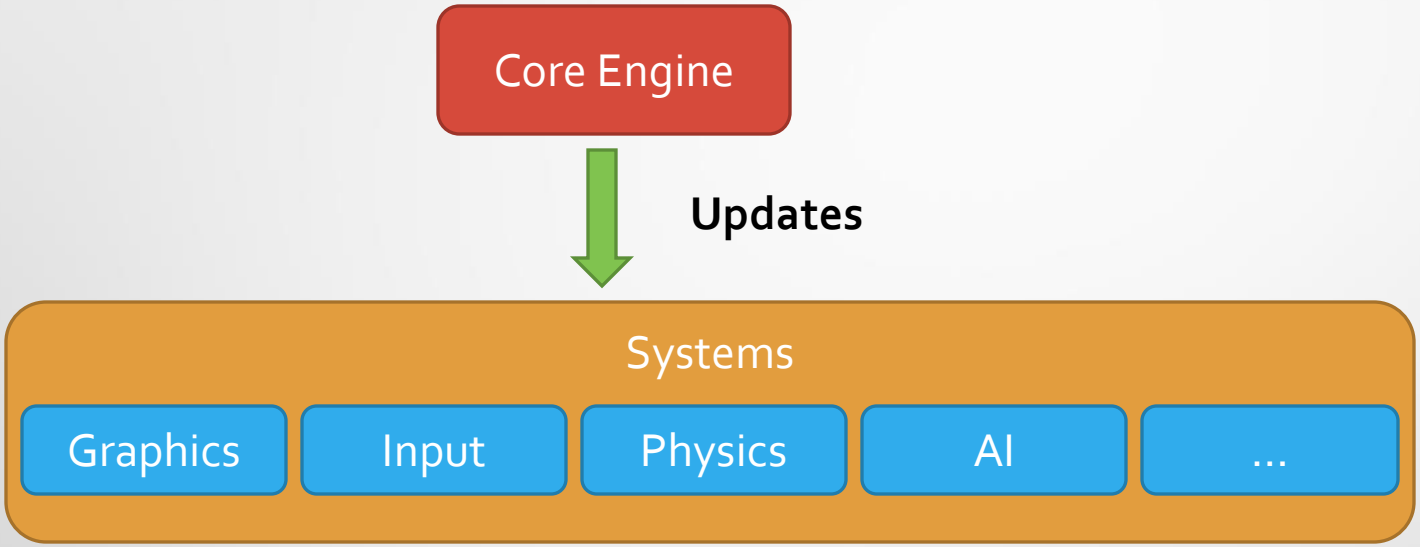
Dependencies

- Dependencies can be code, headers, people, libraries, etc.
- Global Variables!
- Minimize dependencies does not mean eliminate them.
- Good libraries can help by moving responsibility to specialists and leveraging broadly used code.
- Strive for loose coupling between all objects.

High Level

- Game engine consists of systems (or managers).
- Each system oversees a single aspect of the game:
  - Graphics
  - Physics
  - Logic
  - …
- Every frame each system is updated.

Simple Game Engine



How do these systems communicate and share data?

Game Objects

Game Objects

- Pieces of logical interactive content.
- Have data that all systems need.
- For this example, an RTS:
  - Tanks
  - Bombers
  - Infantry
  - Bases
- Also, things like triggers, trees, etc.

Game Objects

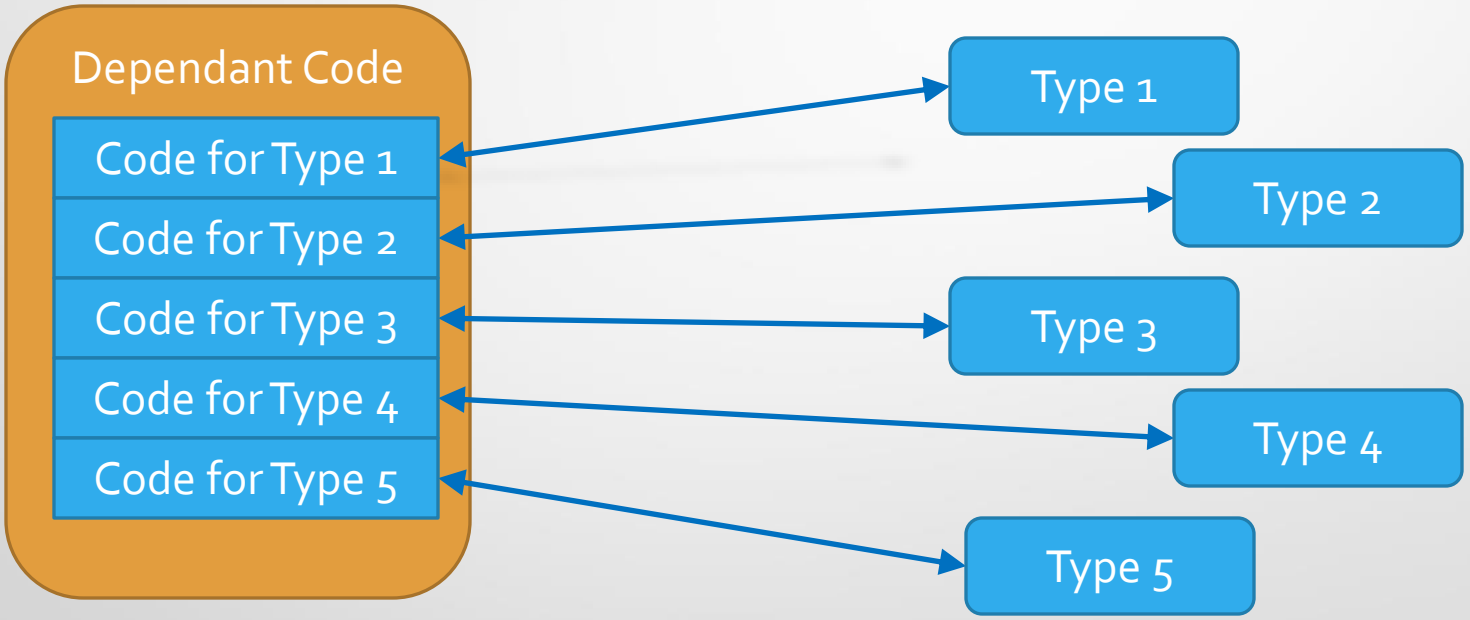
- So how do we build game objects?
- Start with basic object-oriented principles
  - Base class called "GameObject"
  - Specialization derived from this class.

Abstraction

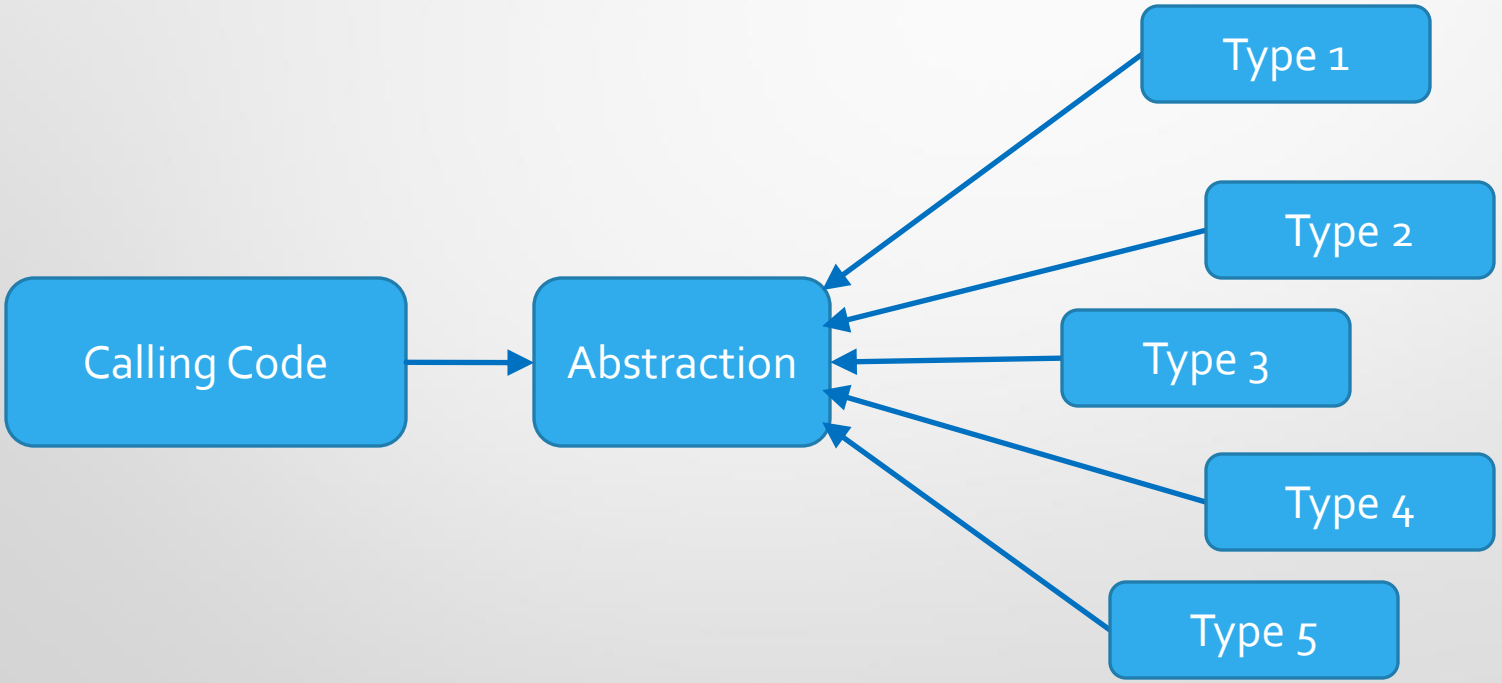
Dependant Code Example

```
void GameLogic::FireGun(GameObject* go)
{
 if( go->Type == "Sniper" )
    //Code for firing sniper rifle
 else if( go->Type == "Rifleman" )
    //Code for firing rifle
 else if( go->Type == "MachineGunner" )
    //Code for firing machine gun
}
```

Dependant Code



Abstraction



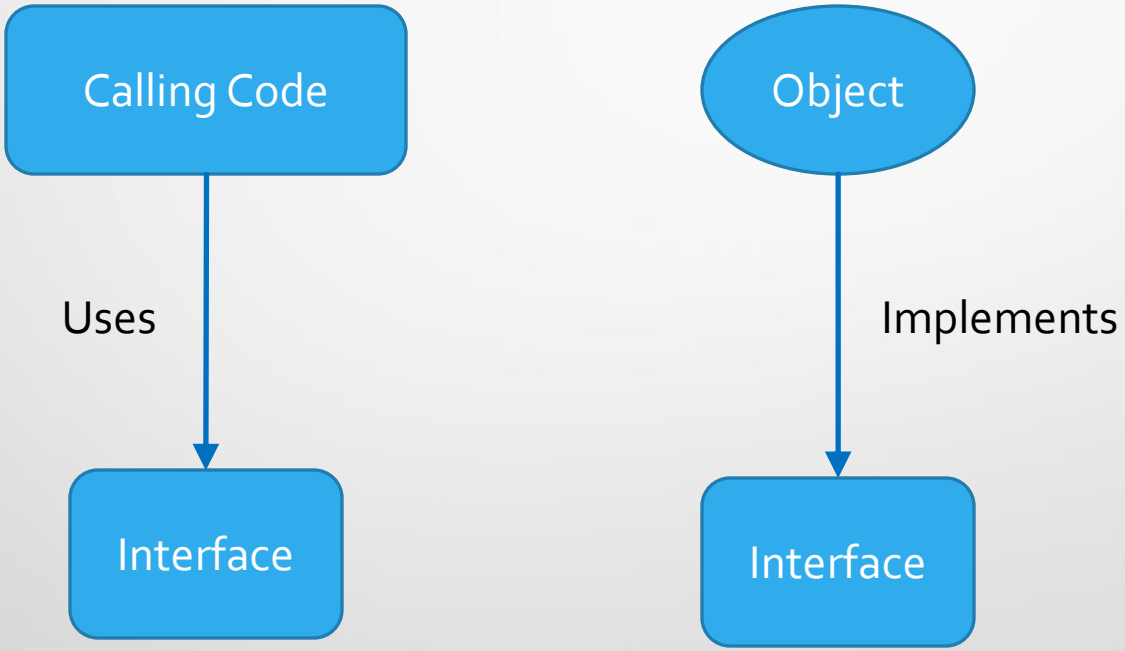
```
void FireGun(GO * go)
{
  go->FireGun();
}

void OtherCode(GO * go)
{
  go->FireGun();
}
```

```
class Infantry : public GO
{
  virtual void FireGun();
};

class Sniper : public Infantry
{
  virtual void FireGun()
  {
      //Code for firing sniper
      rifle
  }
};
```

Abstraction



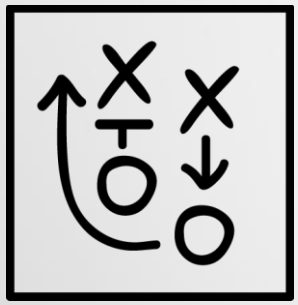
Abstraction

- Calling code can treat all objects with same abstraction as if they are the same.
- Calling code now relies on an abstraction and the implementation now also relies on an abstraction.
- Abstraction applies to more than just methods, it also applies to objects, algorithms, data, relationships, etc.

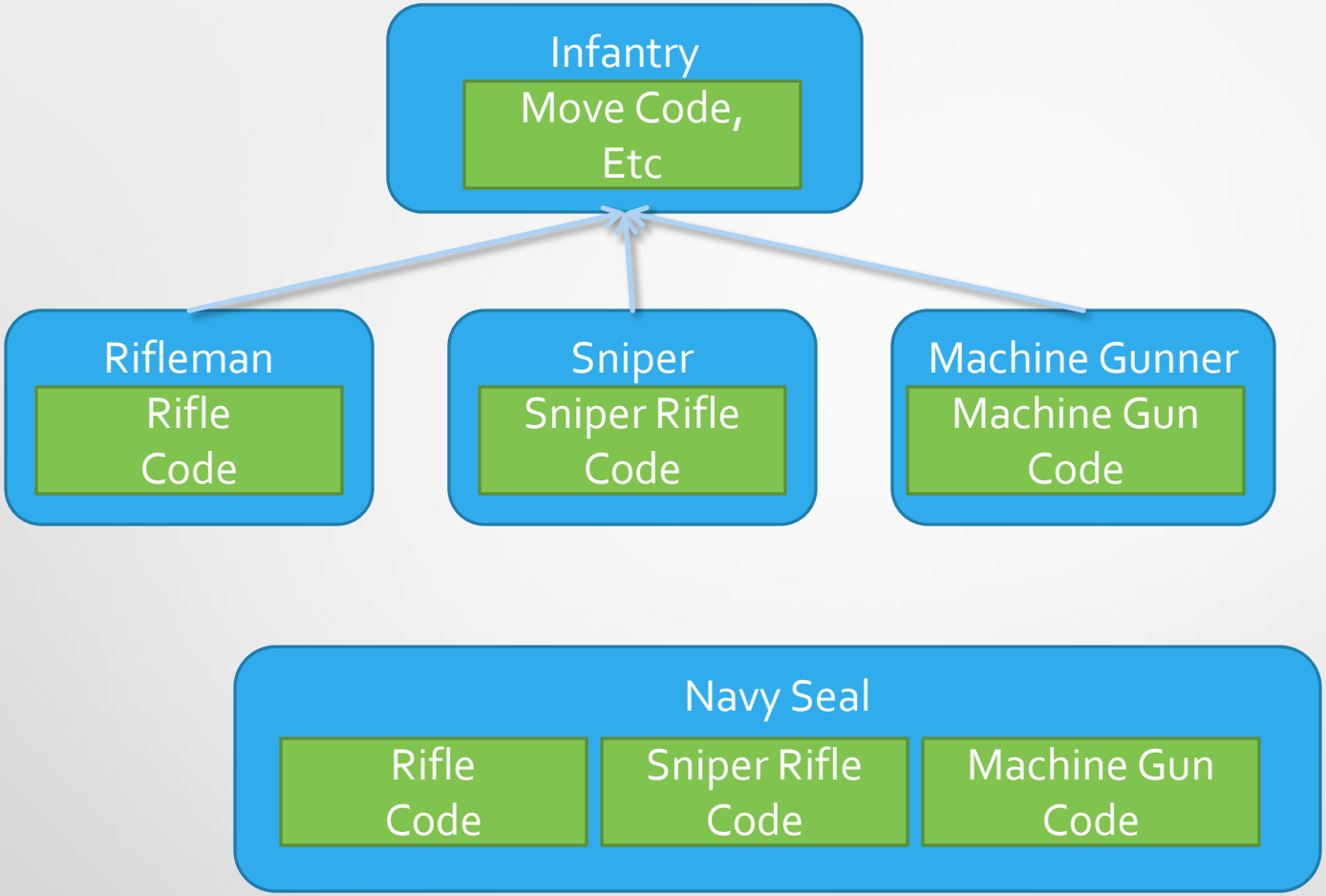
Interface

- **Interface** – An interface abstracts a set of operations on an object
- In C++ this is implemented with virtual functions and inheritance
- Virtual functions come with a low cost in both memory and performance
- Interfaces are a fundamental code concept and are provided by almost all object-oriented languages
- This is formally called **polymorphism**

Architecture Strategy



Program to an abstraction, not an implementation.

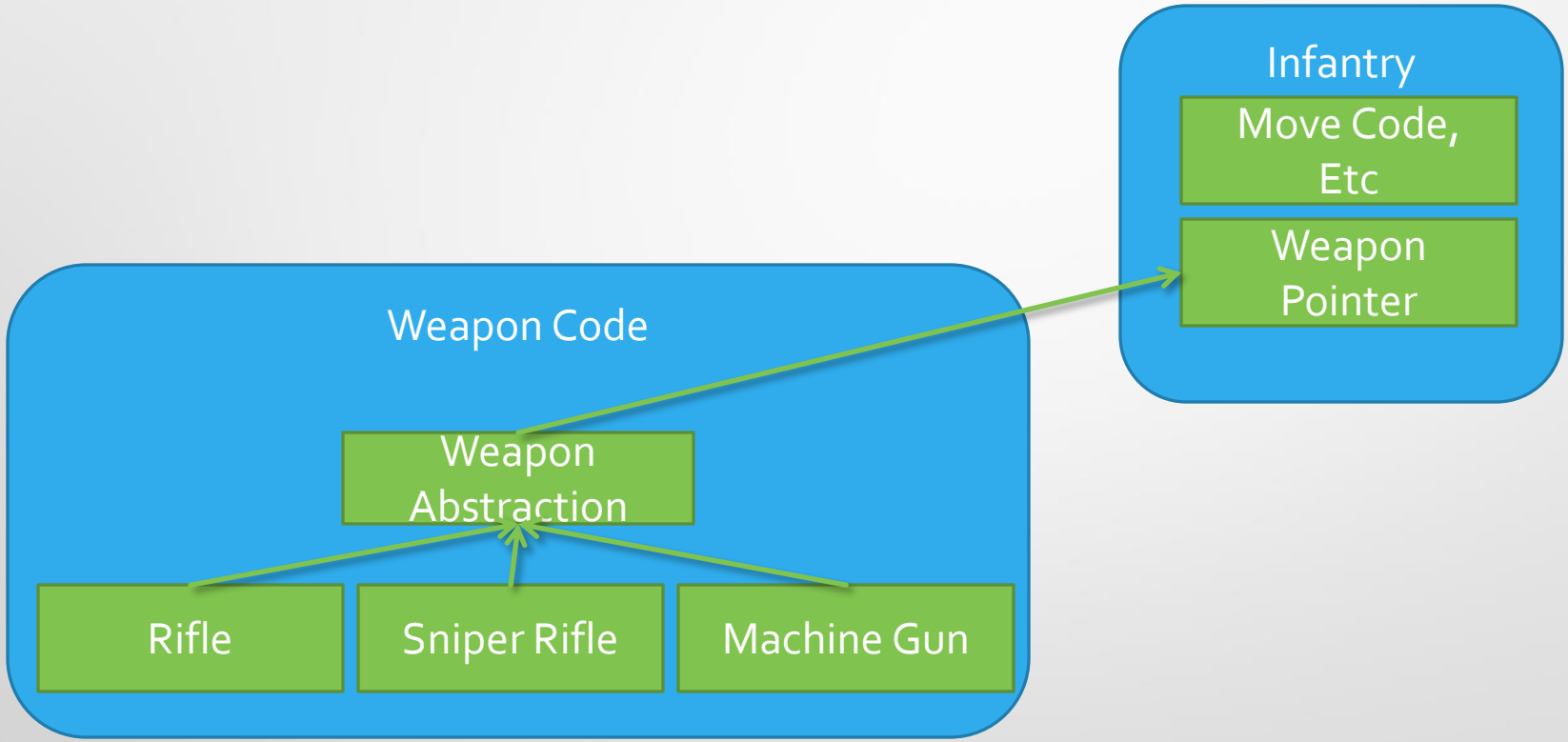


What went wrong?

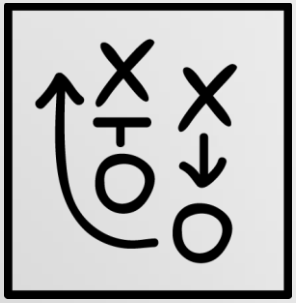
IS-A vs HAS-A

- We did not organize our code well.
- Prefer “has a” to “is a” relationships.
- Many relationships are better modeled with containment instead of inheritance.
- Has a relationships are more flexible.

Infantry has a weapon

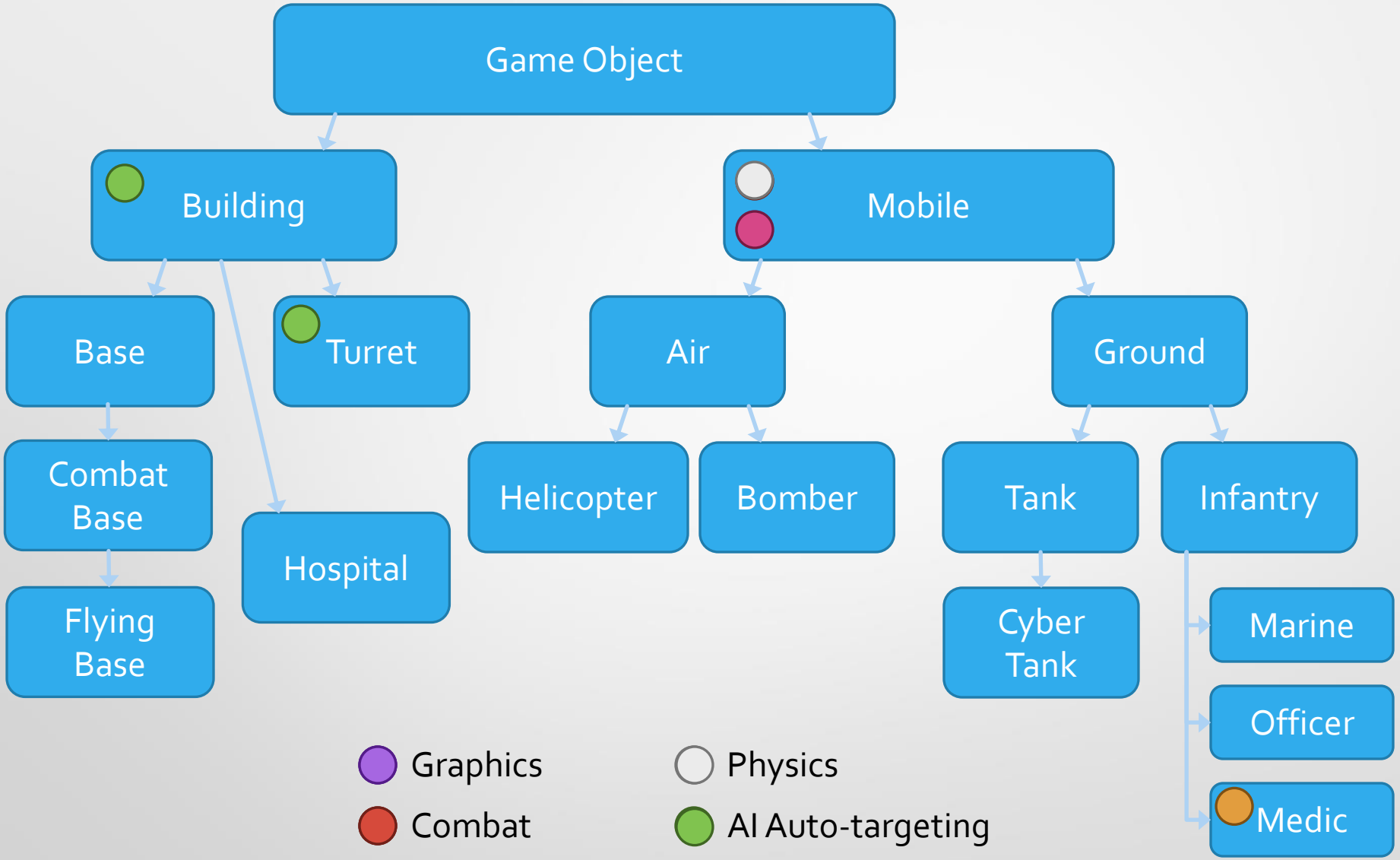


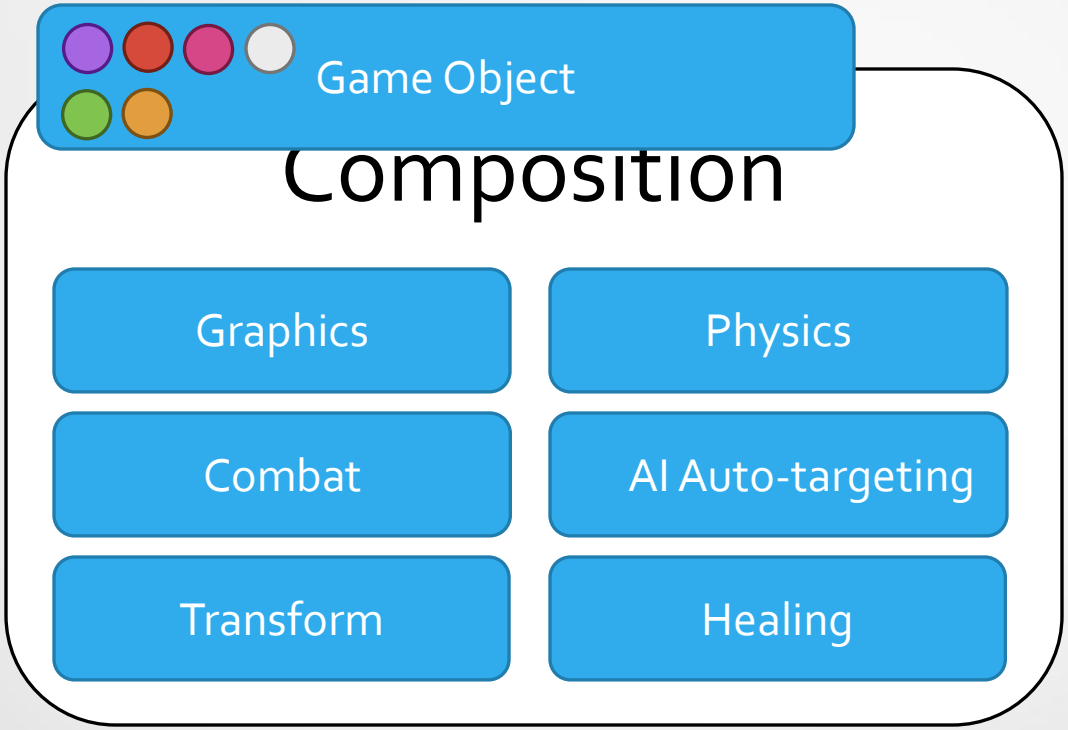
Architecture Strategy



Identify aspects of the code that vary and separate them from those that stay the same.

Game Object System

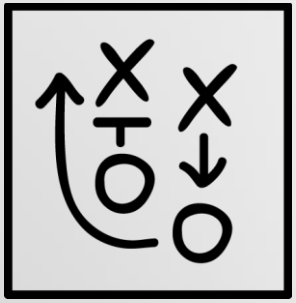

Game Object
Building
Mobile
Base
Turret
Air
Ground
Combat Base
Hospital
Helicopter
Bomber
Tank
Infantry
Flying Base
Cyber Tank
Marine
Officer
Medic
Graphics
Physics
Combat
AI Auto-targeting
Transform
Healing


Game Object
Composition
Graphics
Physics
Combat
AI Auto-targeting
Transform
Healing

Aggregation vs Composition

- Aggregation
  - Object references different objects.
  - Not necessarily lifetime bound.
  - Multiple objects may reference the same aggregated object.
- Composition
  - Object owns different objects called components.
  - Components do not exist outside of composition.
  - When composition is destroyed so are components.
  - Each component has only one owner.

Architecture Strategy

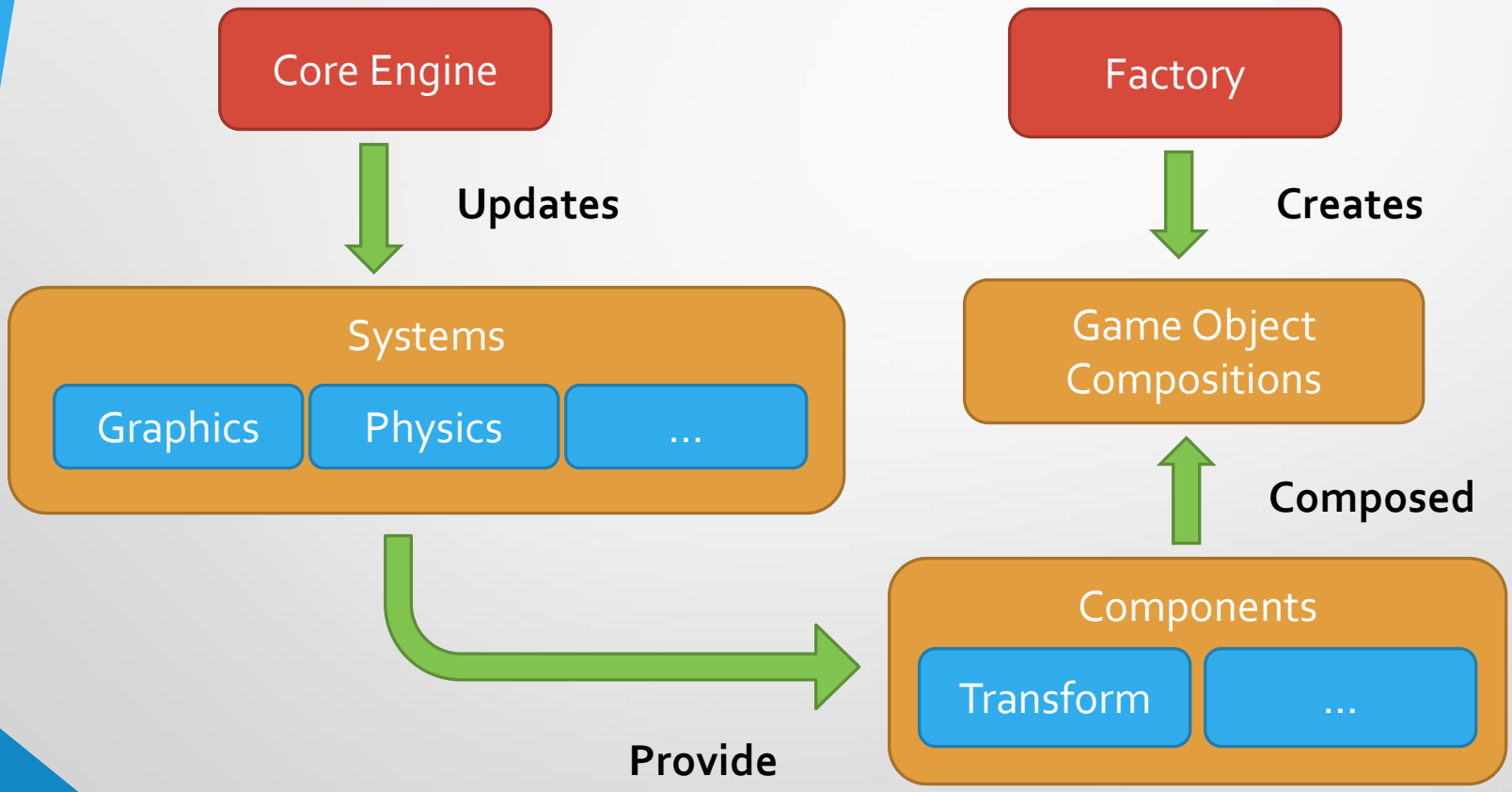


Always prefer aggregation and composition to inheritance.

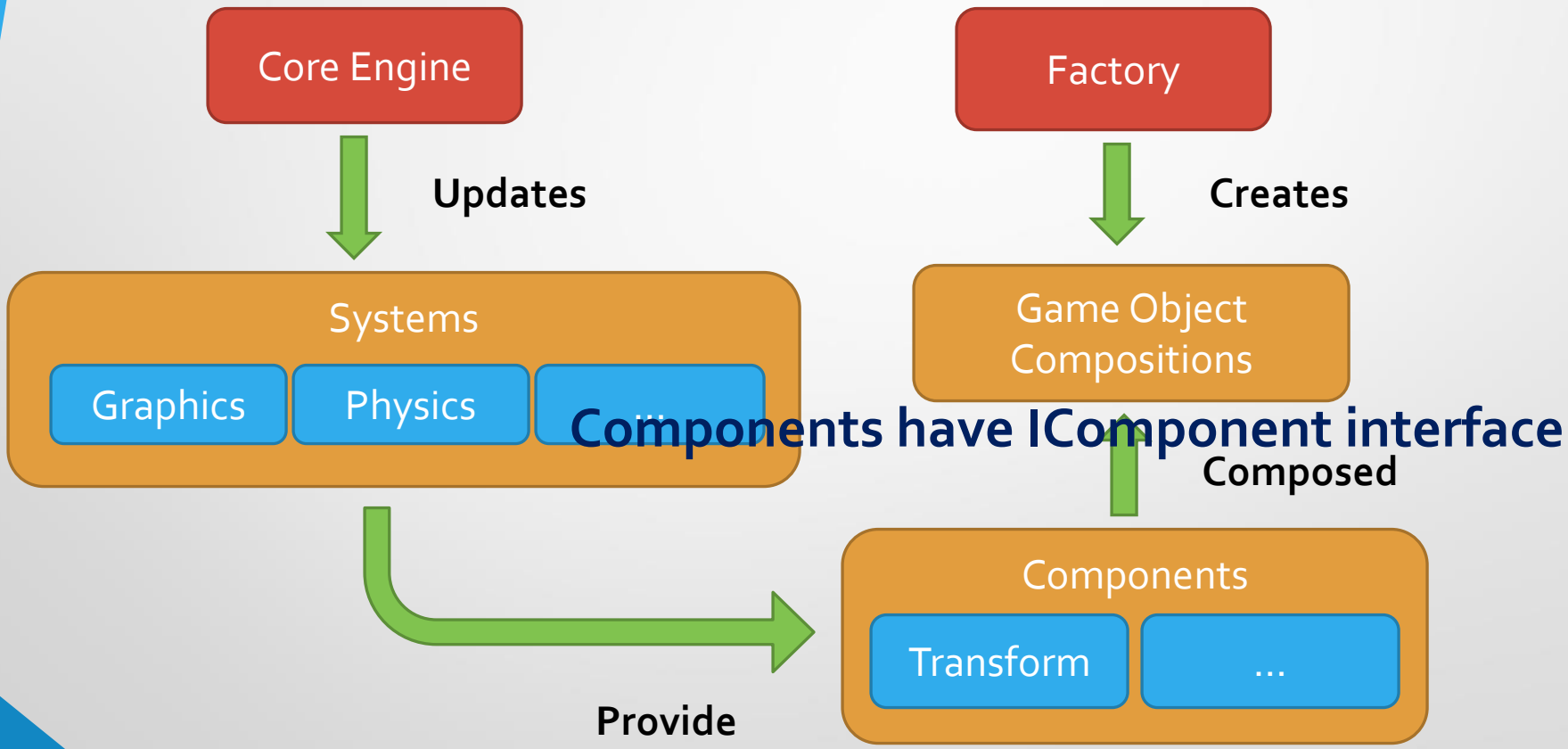
Component Based Engine

- The base class is a collection of components provided by the different systems.
- The components represent orthogonal views of a single entity.
- Every component class inherits from a base component class and has pointer to its owning composition.
- A component can be data, behavior, and/or a link to a system.
- When the game object composition is destroyed it destroys all its components.

Component Based Engine



Component Based Engine

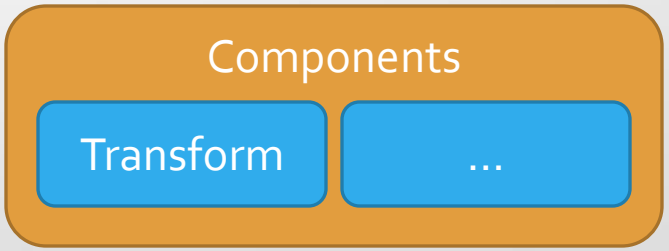


Component Based Engine

IComponent

- **GetOwner()**
- **HasSibling()**
- **Serialize()**

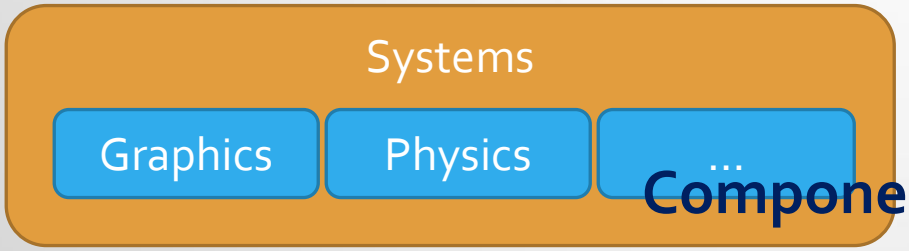
Components have IComponent interface



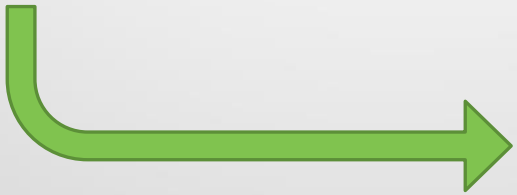
Component Based Engine

The programmer of a system decides on the components of that system

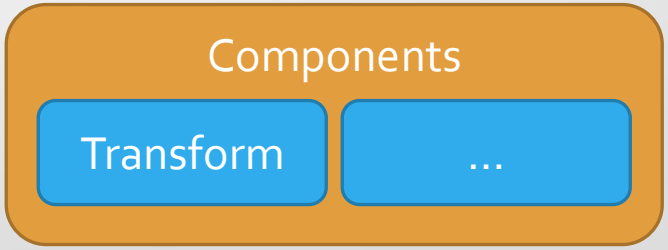
Systems access pools of IComponent *



Components have IComponent interface



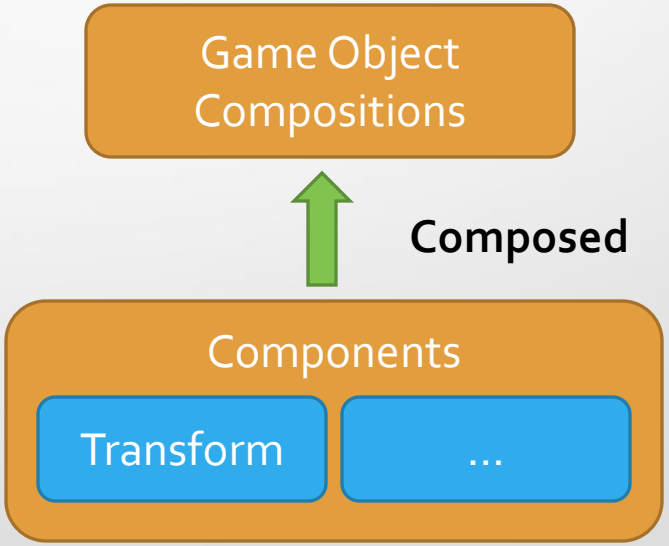
Provide



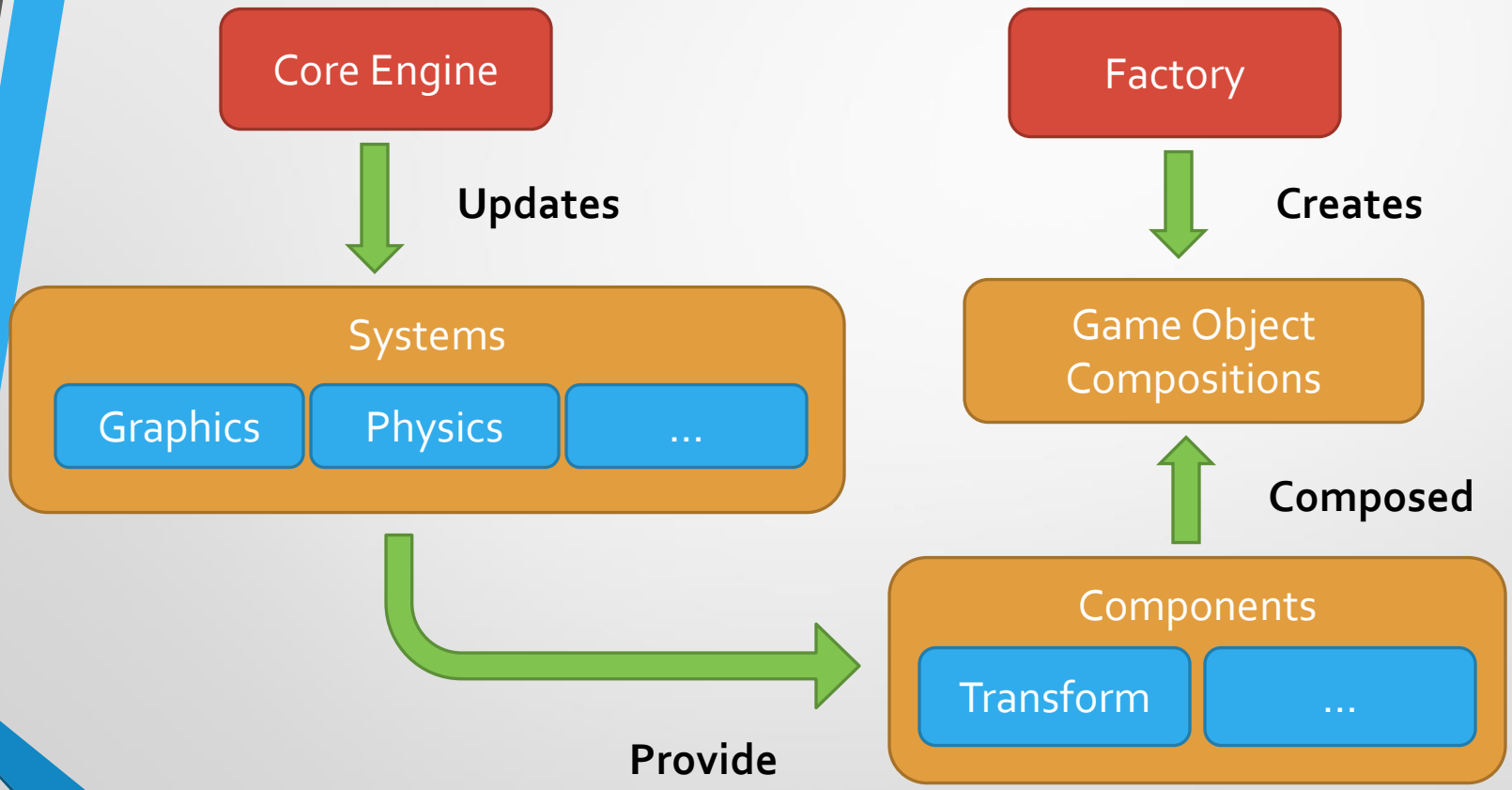
Component Based Engine

GameObject

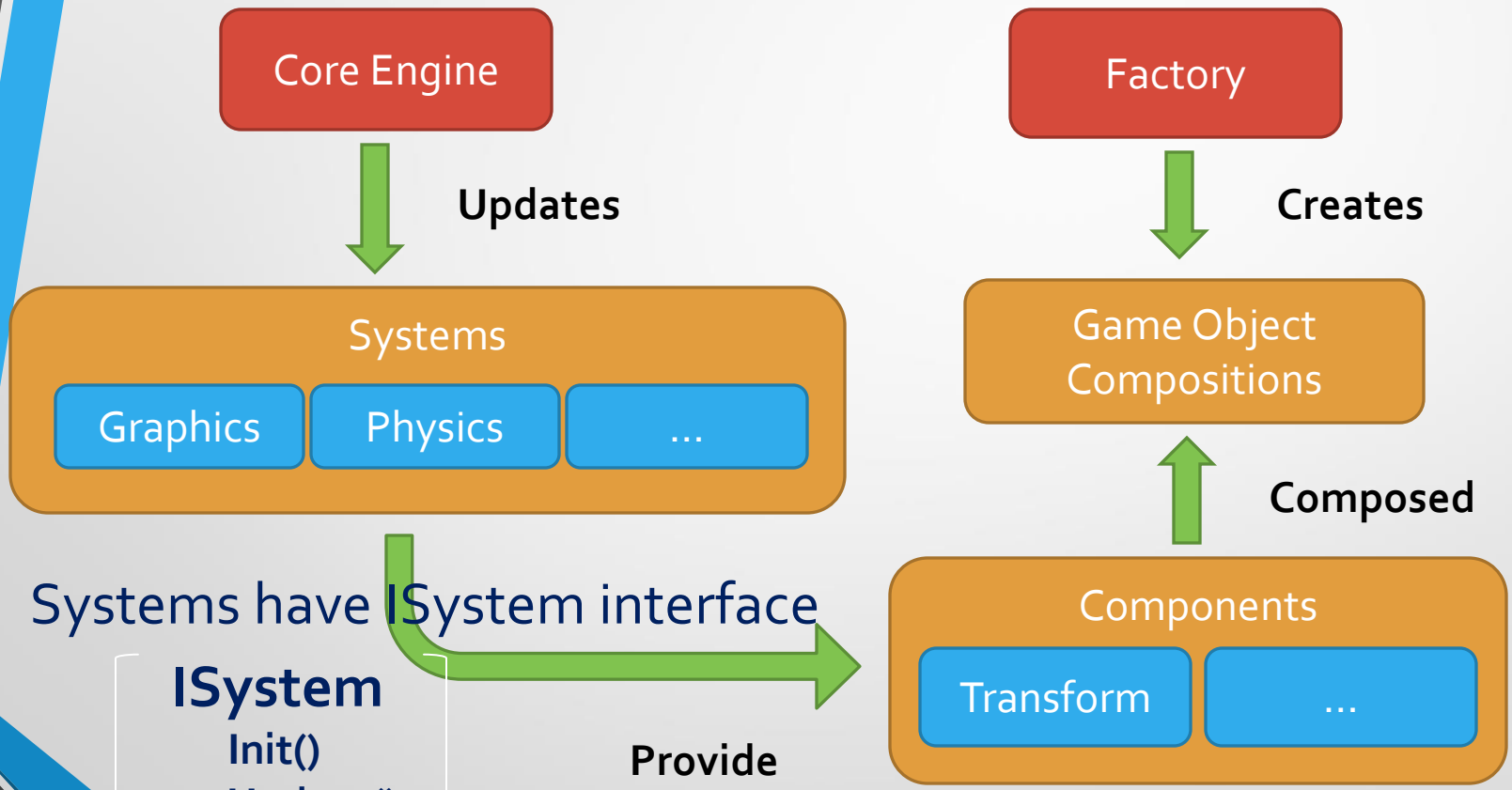
- **AddComponent()**
- **RemoveComponent()**
- **Has()**



Component Based Engine



Component Based Engine

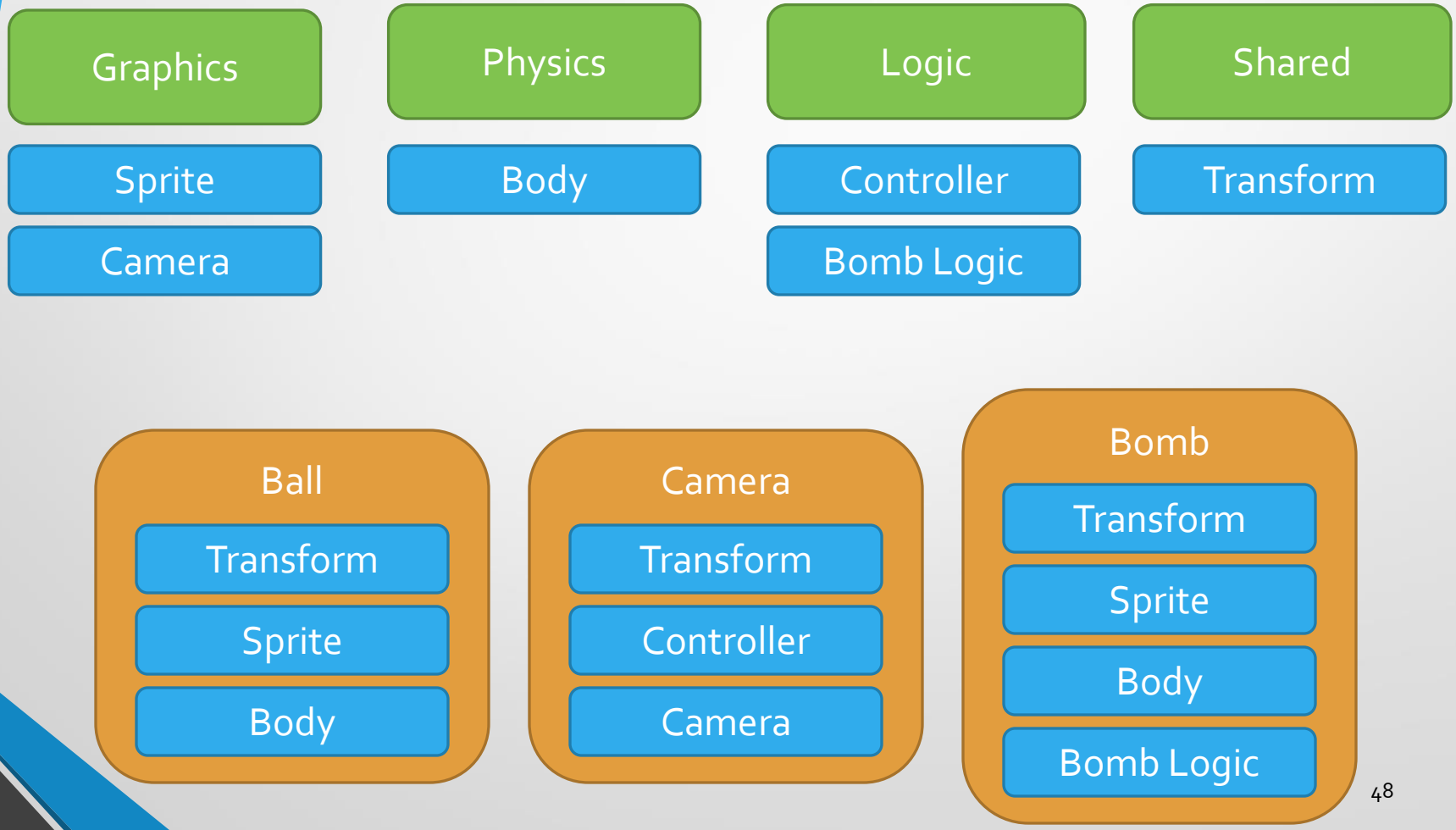


Systems have ISystem interface

ISystem

Init()
Update()
End()

Simple Components



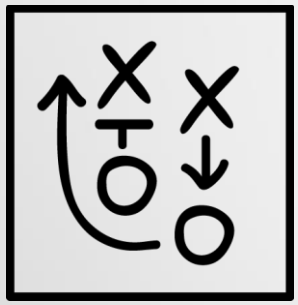
Dependencies between components

- Components still have dependencies between each other.
- Need a flexible simple way for components to handle dependencies.
- To allow for inspections of a composition we need to provide a query function.
- This is done by having a std::map of strings to component pointers.

Dynamic Linking

```
void Sprite::Initialize()
{
  // Looks up component named "Transform" in map
  // Using the 'has' operation
  this->Transform = GetOwner()->has("Transform");
  //
  // Add any additional dependencies here
  //
}
```

Architecture Strategy



Strive for loose coupling between objects.

Dynamic Interaction

```
void Game::MoveObjectLeft(GameObject*
ObjectToBeMoved)
{
  if(Transform* transform =
     ObjectToBeMoved->has("Transform") )
  {
     transform->position.x -= 10;
  }
}
```

Questions?