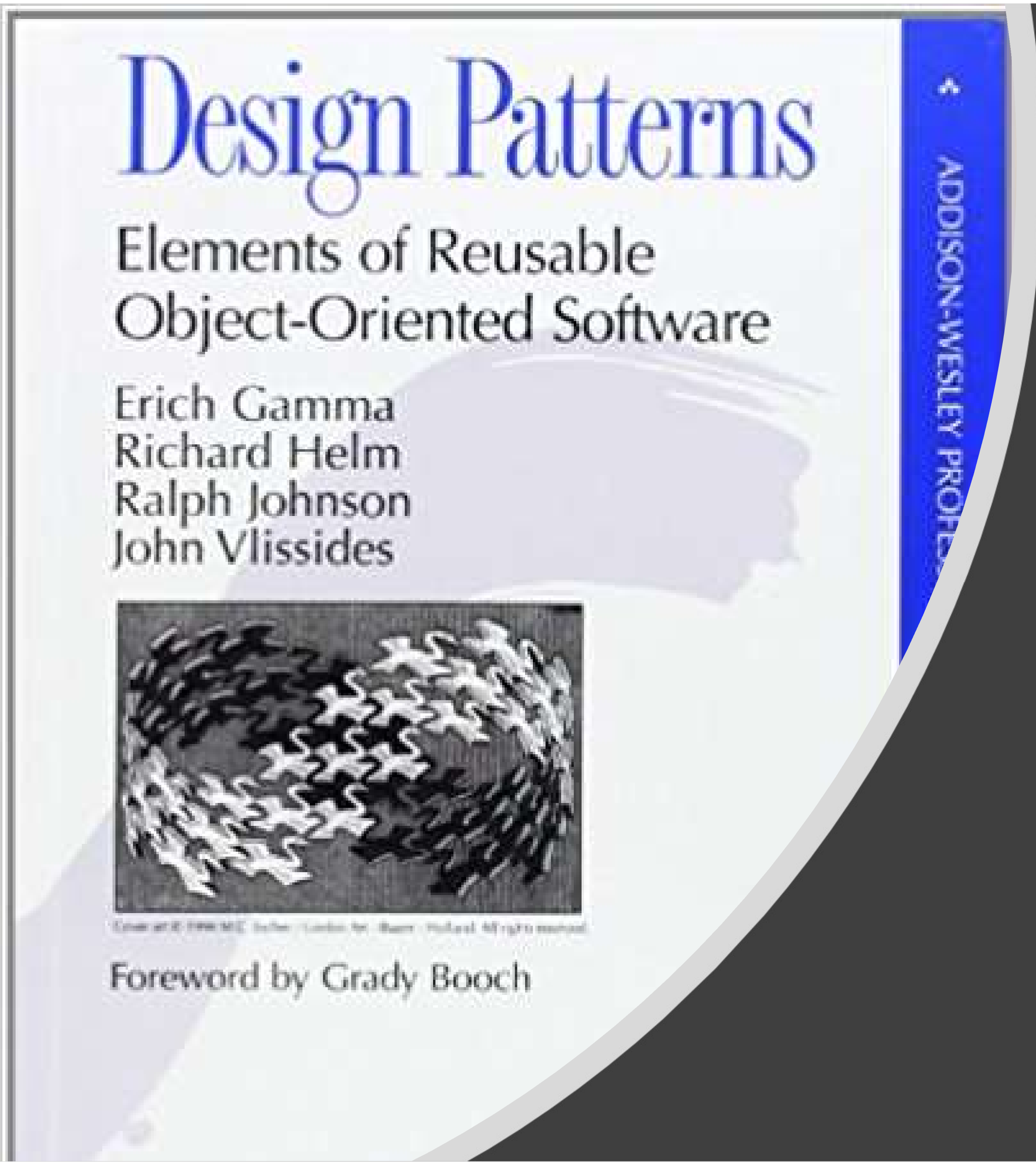


Design Patterns

One-slide Summary

- **Design patterns** separate the **structure** of a system from its **implementation**

- Every design has **tradeoffs**
  - Object-oriented design patterns often trade *greater verbosity* or *less efficiency* for *easier extensibility*

- We'll look at **structural, creational,** and **behavioral** object-oriented design patterns. These patterns should work in just about **any language** with object-oriented features.

Design Patterns Everywhere!

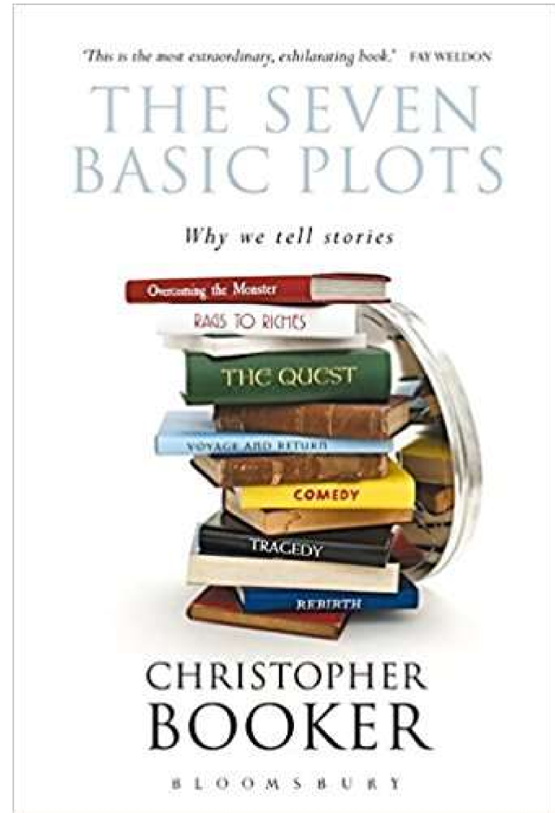
Multiple choice question

1. Rick Astley's never gonna: ← Question Stem

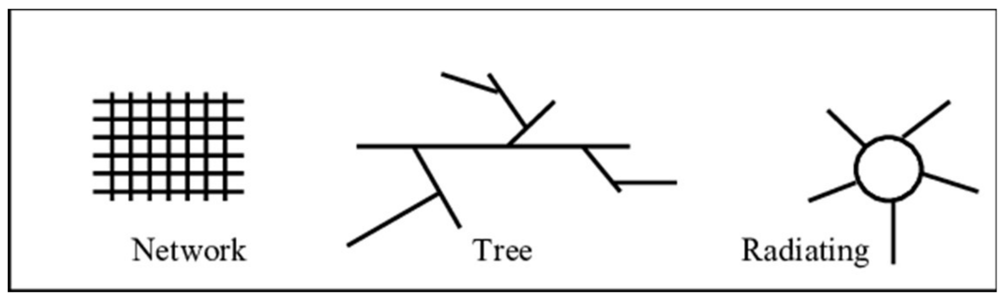
Options

- Give you up
- Let you down
- Run around and
- Desert you
- All of the above ← Correct

Distractors



VERSE	CHORUS	VERSE	CHORUS	BRIDGE	CHORUS
A	B	A	B	C	B



Using Design Patterns Effectively

- **Design for change**
  - Redesign is expensive. Choosing the right pattern lets you avoid it.
- **Consider your requirements** and how they will or won’t change.
  - Don’t use a pattern if it doesn’t fit your current or anticipated needs.
- **Consider at least 2 potential designs before choosing!**
  - Diagram your designs on paper before writing code.

Structural Patterns

- Build new classes/interfaces from existing ones.
- Hide implementation details.
- Provide cleaner/more specialized interface.

Sound familiar?

Adapter Pattern

"Convert the interface of a class into another interface clients expect."

- "Gang of Four" *Design Patterns* book

Adapter Pattern

```
Stack
- push()
- top()
- pop()

    LinkedList
    - push_front()
    - front()
    - pop_front()
    - push_back()
    - back()
    - pop_back()
    - insert()
    - erase()
```

Adapter Pattern (More Examples)

- Early implementations of `fstream` in C++
  - Adapter for the C `FILE` macro

- Autograder: Securely running student code
  - Adapter for containerization library
  - Handles quirks of the library
  - Makes sure that certain options are always used

Other Structural Patterns

- **Composite**: Lets clients treat individual objects and groups of objects uniformly
  - E.g. selecting and moving objects in PowerPoint

- **Proxy**: *“Provide a surrogate or placeholder for another object to control access to it.”*
  - See std::vector<bool>::reference https://en.cppreference.com/w/cpp/container/vector_bool

Creational Patterns

- *“Make a system independent of how its objects are created.”*
- When is a plain constructor not good enough?
  - Control how/when an object is created
  - Overcome language limitations (i.e. no keyword/default args)
  - Hide polymorphic types

Named Constructor (Idiom)

- Technique used in creational patterns.

```
class Llama {
public:
  static Llama* create_llama(string name) {
    return new Llama(name);
  }

private:  // Making ctor private depends on our needs
  Llama(string name_in): name(name_in) {}
  string name;
};
```

Scenario: Polymorphic Objects

- **Problem:** *We need to create and use polymorphic objects without exposing their types to the client.*

- **Solution:** *Write a function that creates objects of the type we want but returns a pointer to their base class.*

Factory Pattern (Function)

- A string tells the factory which type to make.

```
Llama* llama_factory(string name, string type) {
  if (type == "ninja_llama") {
    return new NinjaLlama(name);
  }
  if (type == "whooping_llama") {
    return new WhoopingLlama(name);
  }
  ...
}

Llama* steve = llama_factory("Steve", "ninja_llama");
```

Factory Pattern (Class)

- Client calls (possibly) static methods to make the right type.

```
class LlamaFactory {
public:
  static Llama* make_ninja_llama(string name) {
    return new NinjaLlama(name);
  }

  static Llama* make_whooping_llama(string name) {
    return new WhoopingLlama(name);
  }
};

Llama* steve = LlamaFactory::make_ninja_llama("Steve");
```

Scenario: Difficulty-Based Enemies

*We're implementing a computer game with a **polymorphic Enemy class** hierarchy, and we want to spawn **different versions** of enemies based on the selected difficulty.*

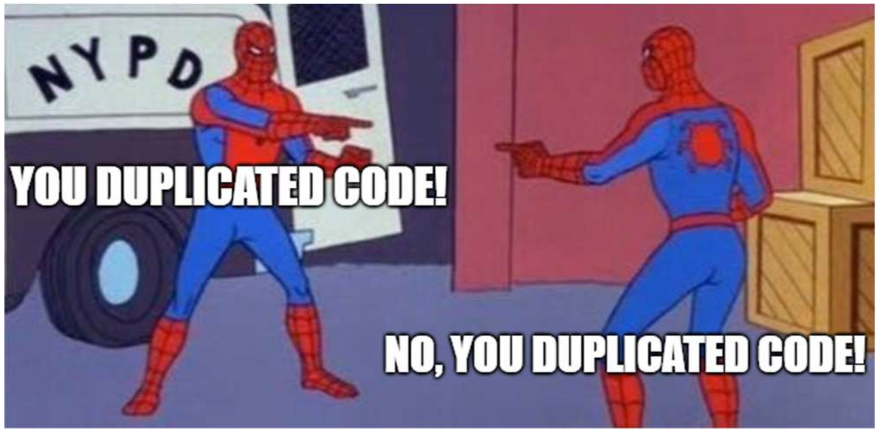
"**Normal**" difficulty: Regular goomba

"**Hard**" difficulty: Spiked goomba

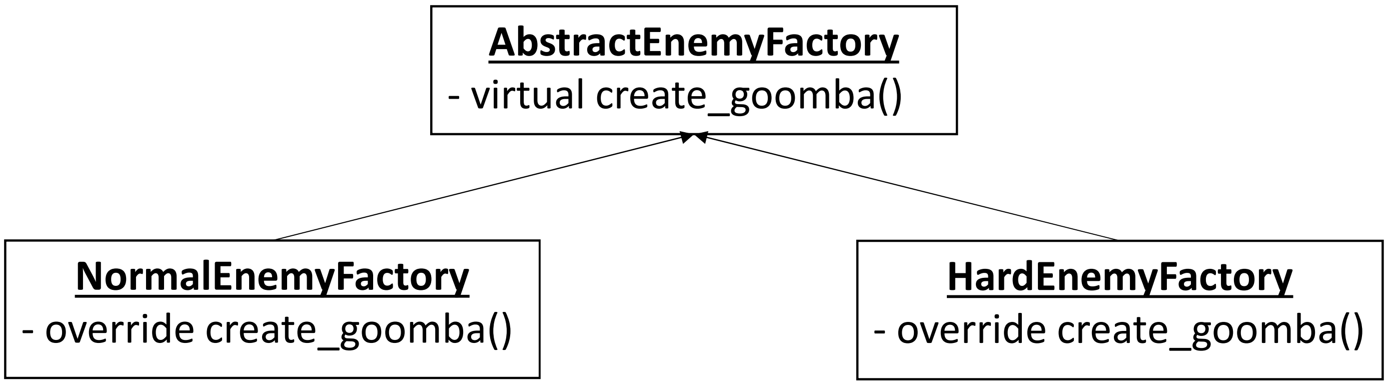
Scenario: Difficulty-Based Enemies

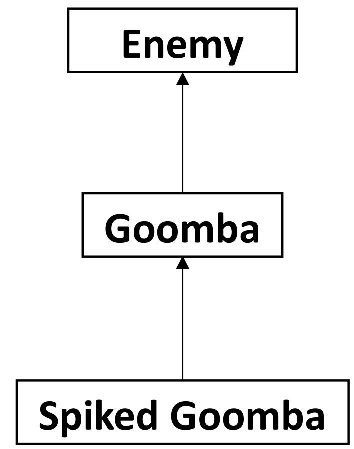
- **Bad Solution:** Everywhere we spawn an enemy, check the difficulty.

```
// !! DON'T DO THIS !!
Enemy* goomby = nullptr;
if (difficulty == "normal") {
  goomby = new Goomba();
}
else if (difficulty == "hard") {
  goomby = new SpikedGoomba();
}
```



Solution: Abstract Factory






```
// Only have to do this once!
AbstractEnemyFactory* factory = nullptr;
if (difficulty == “normal”) {
  factory = new NormalEnemyfactory();
}
else if (difficulty == “hard”) {
  factory = new HardEnemyFactory();
}
...
Enemy* goomby = factory->create_goomba();
```

Scenario: Global Application State

We have some application state that needs to be globally accessible, but we need to control how the data is accessed and updated.

Bad solution: Naked global variables (plz no).

Less bad solution: Put all the state in a class, have a global instance of it.

Aside: When is Global State OK?

- Need acce sing parameters excessivel
  - This is **no** obal variables just to pass few .

- State stored outside of your program (database, web API, etc.)

Singleton Pattern

"Ensure a class only has ***one instance****, and provide a global point of access to it."*

Singleton
public:
- static ***get_instance***() // named ctor
private:
- static ***instance*** // the one instance
- Singleton() // ctor

Singleton (Implementation)

```
class Singleton {
  public static Singleton get_instance() {
    if (Singleton.instance == null) {
      Singleton.instance = new Singleton();
    }
    return Singleton.instance;
  }
  private static Singleton instance = null;

  private Singleton() {
    spams = 42;
    System.out.println("Singleton created");
  }

  // Our global state
  private int spams;
  public int num_spams() {
    return spams;
  }
  public void add_spam() {
    spams += 1;
  }
}
```

Using the Singleton

Exercise: What is the output of this code?

```
class Main {
  public static void main(String[] args) {
    int spams = Singleton.get_instance().num_spams();
    System.out.println(spams);

    Singleton.get_instance().add_spam();
    spams = Singleton.get_instance().num_spams();
    System.out.println(spams);
  }
}
```

<u>Singleton</u>

public:
- static ***get_instance***() // named ctor
- num_spams()
- add_spam() // adds 1 to num_spams

private:
- static ***instance*** // the one instance
- Singleton() // ctor, prints message
- spams

Using the Singleton (Solution)

Singleton

public:
- static ***get_instance***() // named ctor
- num_spams()
- add_spam() // adds 1 to num_spams

private:
- static ***instance*** // the one instance
- Singleton() // ctor, prints message
- spams

Exercise: What is the output of this code?

```
class Main {
  public static void main(String[] args) {
    int spams = Singleton.get_instance().num_spams();
    System.out.println(spams);

    Singleton.get_instance().add_spam();
    spams = Singleton.get_instance().num_spams();
    System.out.println(spams);
  }
}
```

Output:

```
Singleton created
42
43
```

Singleton.get_instance()…

- That seems like a lot of typing. What if we did this?

```
Singleton s = Singleton.get_instance();
System.out.println(s.num_spams())
```

- So good or no good?

There is ***no guarantee*** *that Singleton.get_instance() will return* ***the same object*** *every time it's called!*

Singleton: Design Scenario

*We're implementing a computer version of the card game Euchre. In addition to a few abstract datatypes, you have a **Game** class that stores the state needed for a game of Euchre. When started, your application plays one game of Euchre and then exits.*

*Should we make **Game** a singleton?*

Make Game a Singleton?

Yaaas

- There's only one instance of Game in our application.

Plz no

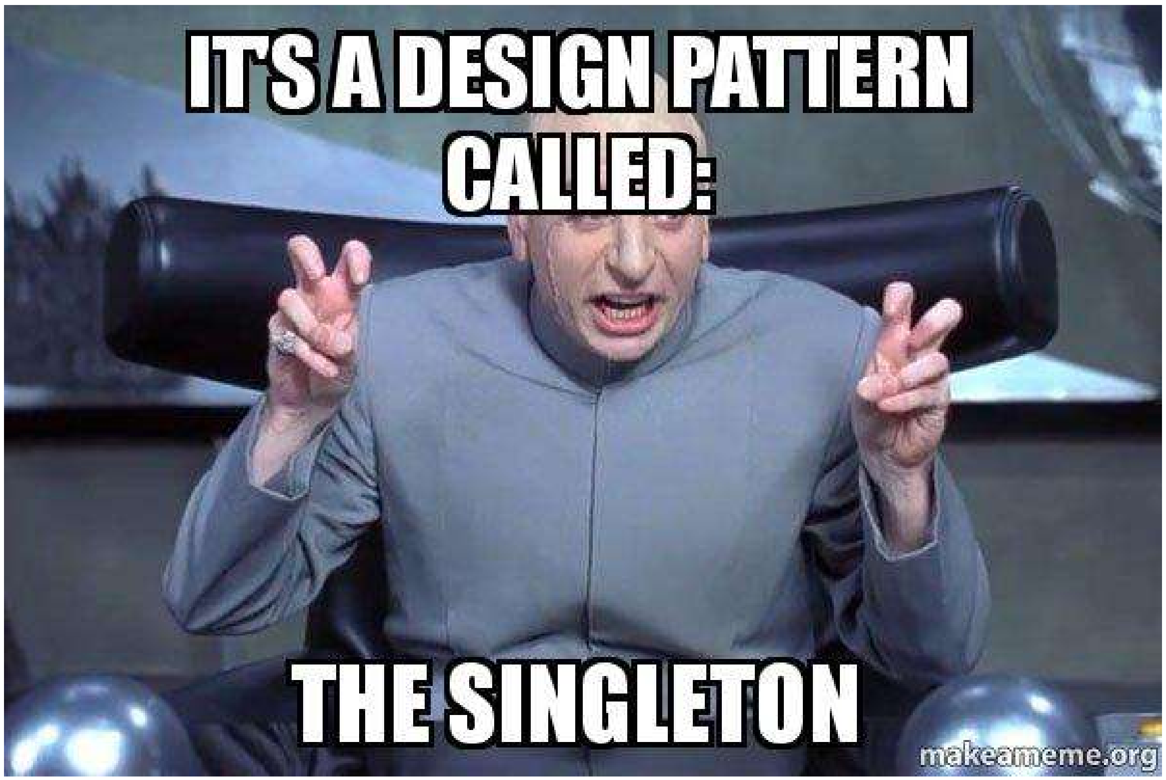
- There only *happens* to be one instance of Game. There's no *requirement* that we only have one instance.

We should only use the Singleton pattern when our application **requirements** dictate that **only one instance** should exist.

The Singleton pattern is **not an excuse to make everything global**!

Break (and moar memes!)

Break (memes are design patterns!)



Behavioral Patterns

"Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects."

- Behavioral pattern you've seen: **Iterator pattern**
  - Uniform interface for traversing containers regardless of how they're implemented.

Scenario: “Lock-on” in Action-Adventure Game

We’re implementing a computer game where the player character can “lock-on” to an enemy (face towards them regardless of movement). When a locked-onto enemy is defeated, the character should stop targeting that enemy.

“Lock-on”: Not-so-good Implementation

- When an enemy is defeated, call `release_lock_on()` on the player character.

```
class Player {
  public void release_lock_on(Enemy enemy) {
    if (enemy == locked_on) {
      locked_on = null;
    }
  }
  private Enemy locked_on;
}
```

```
class Enemy {
  // Called when the enemy is defeated
  public void on_death() {
    // Global accessor for the player character
    get_player().release_lock_on(this);
  }
}
```

- What are some problems with this approach?

“Lock-on”: Not-so-good Implementation

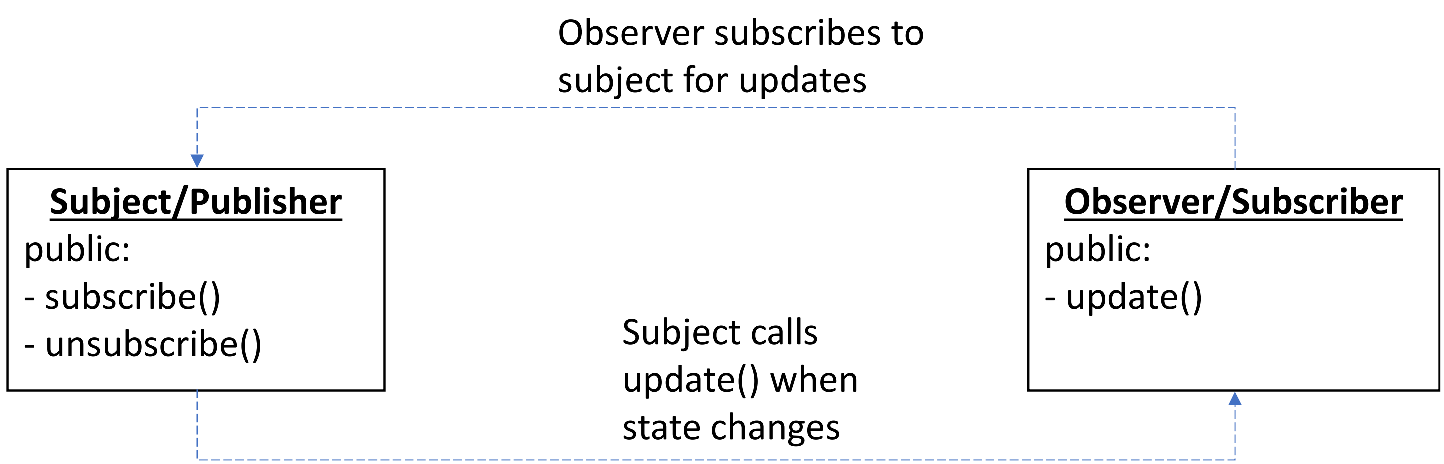
```
class Player {
  public void release_lock_on(Enemy enemy) {
    if (enemy == locked_on) {
      locked_on = null;
    }
  }
  private Enemy locked_on;
}
```

```
class Enemy {
  // Called when the enemy is defeated
  public void on_death() {
    // Global accessor for the player character
    get_player().release_lock_on(this);
  }
}
```

- Player and Enemy are **tightly coupled**
  - Changing one will probably force us to change the other
- What if we had more than one player?
- What if we want to update the player’s “score” when they defeat an enemy?
- Every time we want something new to happen when an enemy dies, ***we are forced to update the Enemy class and couple it with the new feature.***

Observer Pattern (a.k.a. “Publish-Subscribe”)

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



Note: subscribe and unsubscribe can be static or non-static, depending on implementation.

Observer Pattern (a.k.a. "Publish-Subscribe")

Exercise: How many times is "Received update" printed?

Subject/Publisher
public:
- subscribe()
- unsubscribe()

Observer/Subscriber
public:
- update()

```
class Subject {
  public static void subscribe(Observer observer) {
    subscribers.Add(observer);
  }
  public static void unsubscribe(Observer observer) {
    subscribers.Remove(observer);
  }
  public static void change_state() {
    foreach (Observer observer in subscribers) {
      observer.update();
    }
  }
  private static List<Observer> subscribers
      = new List<Observer>();
}
```

```
class Observer {
  public void update() {
    Console.WriteLine("Received update");
  }
}
```

```
class MainClass {
  public static void Main (string[] args) {
    Observer observer1 = new Observer();
    Observer observer2 = new Observer();

    Subject.subscribe(observer1);
    Subject.change_state();

    Subject.subscribe(observer2);
    Subject.change_state();

    Subject.unsubscribe(observer2);
    Subject.change_state();
  }
}
```

Observer for "Lock-on" Feature

Enemy

public:
- static ***subscribe***(EnemyObserver)
- static ***unsubscribe***(EnemyObserver)

// calls update_enemy_defeated(this)
- on_death()

Player

public:
- lock_on(Enemy) // targets an enemy

// un-targets enemy if currently targeting
- update_enemy_defeated(Enemy)

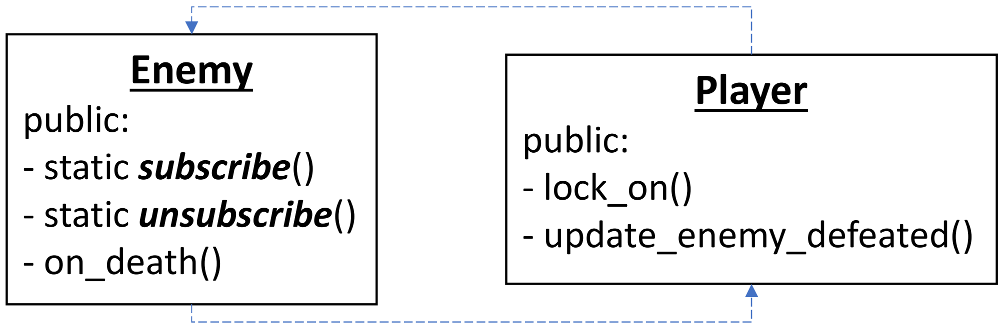
private:
- targeted_enemy

EnemyObserver
- abstract **update_enemy_defeated**()

Player
- override **update_enemy_defeated**()

*Abstract means "derived classes must override this method".

Observer for “Lock-on” Feature (Implementation)




```
class Enemy {
  public static void subscribe(EnemyObserver observer) {
    subscribers.Add(observer);
  }
  public static void unsubscribe(EnemyObserver observer) {
    subscribers.Remove(observer);
  }

  public void on_death() {
    foreach (EnemyObserver observer in subscribers) {
      observer.update_enemy_defeated(this);
    }
  }

  private static List<EnemyObserver> subscribers
      = new List<EnemyObserver>();
}
```

```
interface EnemyObserver {
  void update_enemy_defeated(Enemy enemy);
}
```

```
class Player: EnemyObserver {
  public void update_enemy_defeated(Enemy enemy) {
    if (enemy == target) {
      target = null;
    }
  }

  public void lock_on(Enemy enemy) {
    target = enemy;
  }

  private Enemy target;
}
```

Observer “update_” Functions

- Having multiple “update_” functions keeps things granular.
  - Observers that don’t care about an update can ignore it (with an empty implementation of the update function).

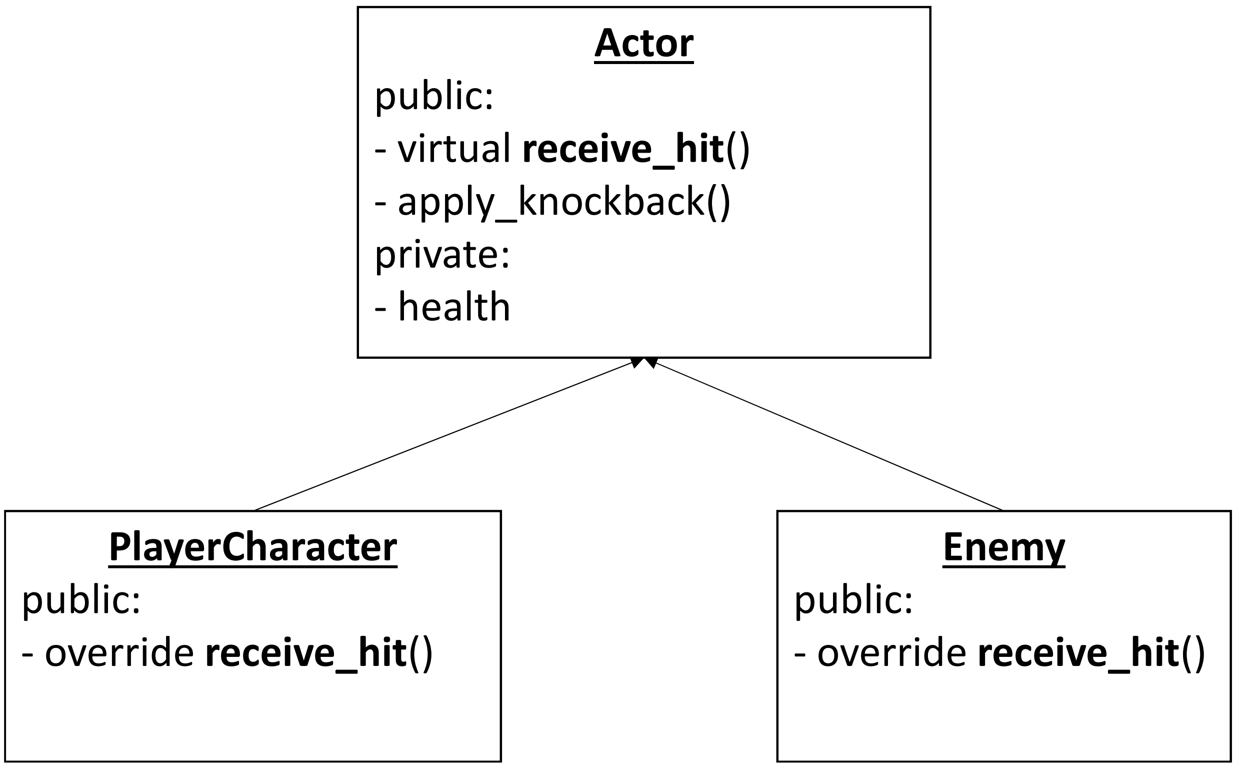
- Generally better to pass the new data as parameters to the update functions (**push**), as opposed to making the observers fetch it themselves (**pull**).

Scenario: Damage-Dealing in Action Game

We're building a computer game where the player characters engage in combat with a variety of enemies. When a player or enemy is hit, they take damage.

If their health reaches zero, they die. If the player dies, the game ends. When an enemy dies, it drops an item. Otherwise, the player/enemy is knocked back and emits a sound.

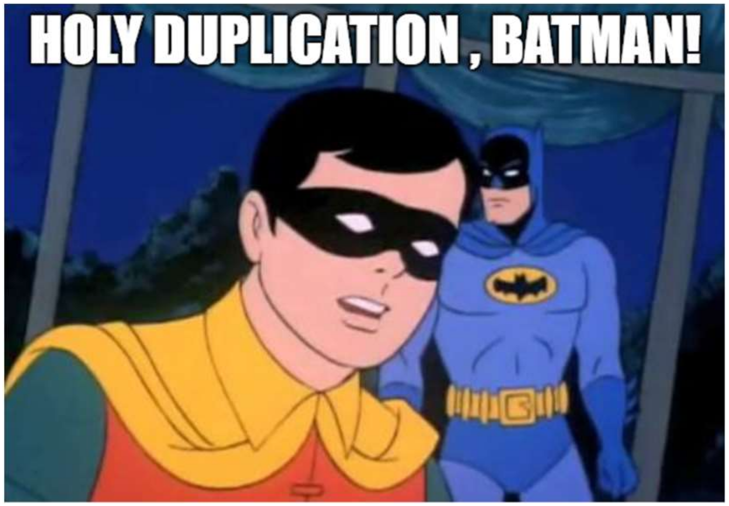
Damage-Dealing: First Design



Note: receive_hit is called on an Actor when it should take damage.

Damage-Dealing: First Design

```
class Actor {
  public virtual void receive_hit(float damage) {
    health -= damage;
  }
  public float get_health() { return health; }
  private float health = 42;
  public void apply_knockback() {
    Console.WriteLine("Knocked back!");
  }
}
```



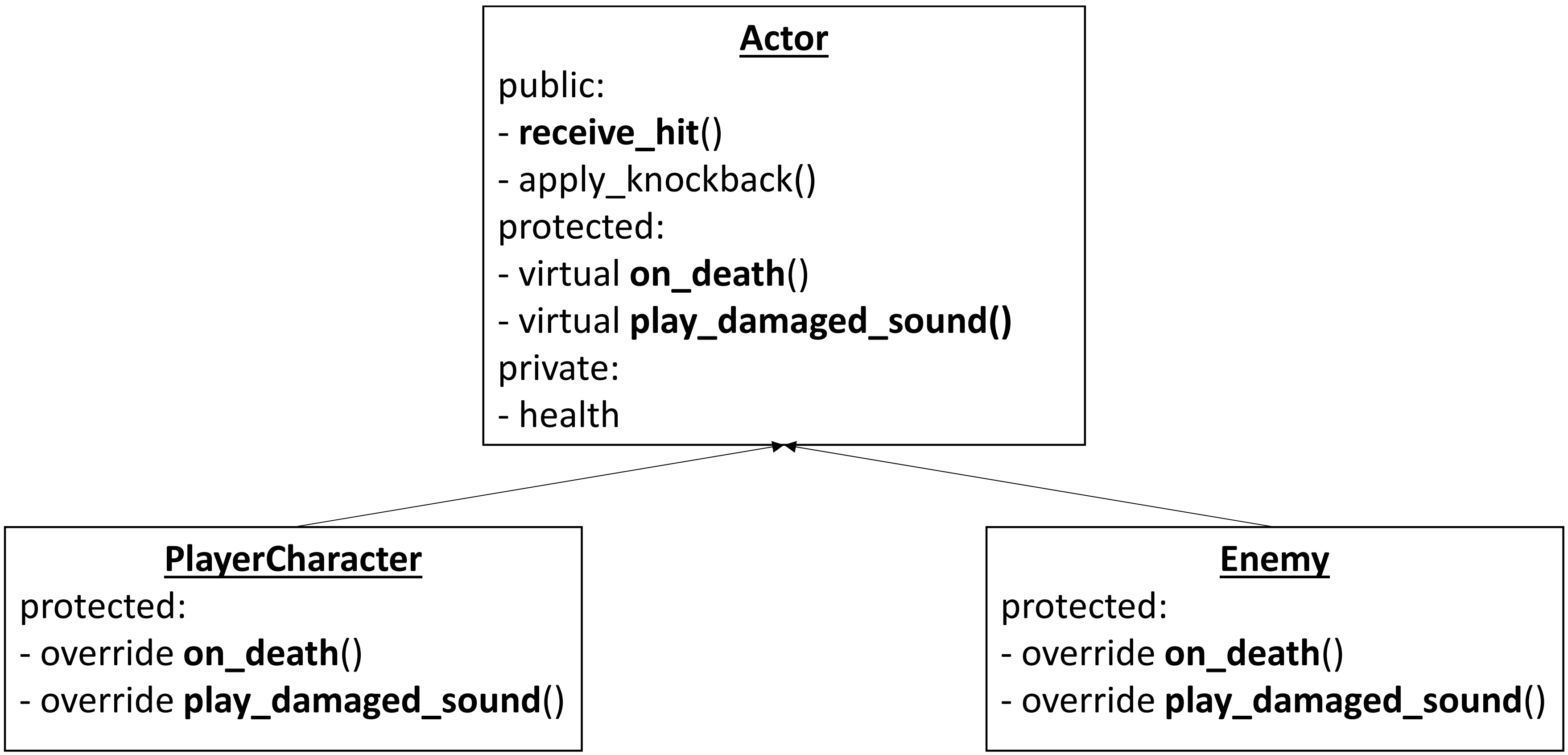

```
class Enemy: Actor {
  public override void receive_hit(float damage) {
    base.receive_hit(damage);
    if (get_health() <= 0) {
      Console.WriteLine("Dropped an item");
    }
    else {
      Console.WriteLine("Weah");
      apply_knockback();
    }
  }
}
```

```
class Player: Actor {
  public override void receive_hit(float damage) {
    base.receive_hit(damage);
    if (get_health() <= 0) {
      Console.WriteLine("Game over");
    }
    else {
      Console.WriteLine("Ow");
      apply_knockback();
    }
  }
}
```

Template Method Pattern

*“Define the **skeleton** of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses **redefine certain steps** of an algorithm without changing the algorithm’s structure.”*

Damage-Dealing: Template Method



Damage-Dealing: Template Method (Implementation)

```
class Actor {
  public void receive_hit(float damage) {
    health -= damage;
    if (get_health() <= 0) {
      on_death();
    }
    else {
      play_damaged_sound();
      apply_knockback();
    }
  }
  protected virtual void on_death() {}
  protected virtual void play_damaged_sound() {}

  // Other members same as before
}
```

```
class Enemy: Actor {
  protected override void on_death() {
    Console.WriteLine("Dropped an item");
  }
  protected override void play_damaged_sound() {
    Console.WriteLine("Weah");
  }
}
class Player: Actor {
  protected override void on_death() {
    Console.WriteLine("Game over");
  }
  protected override void play_damaged_sound() {
    Console.WriteLine("Ow");
  }
}
```

Template Method: The “Hollywood Principle”

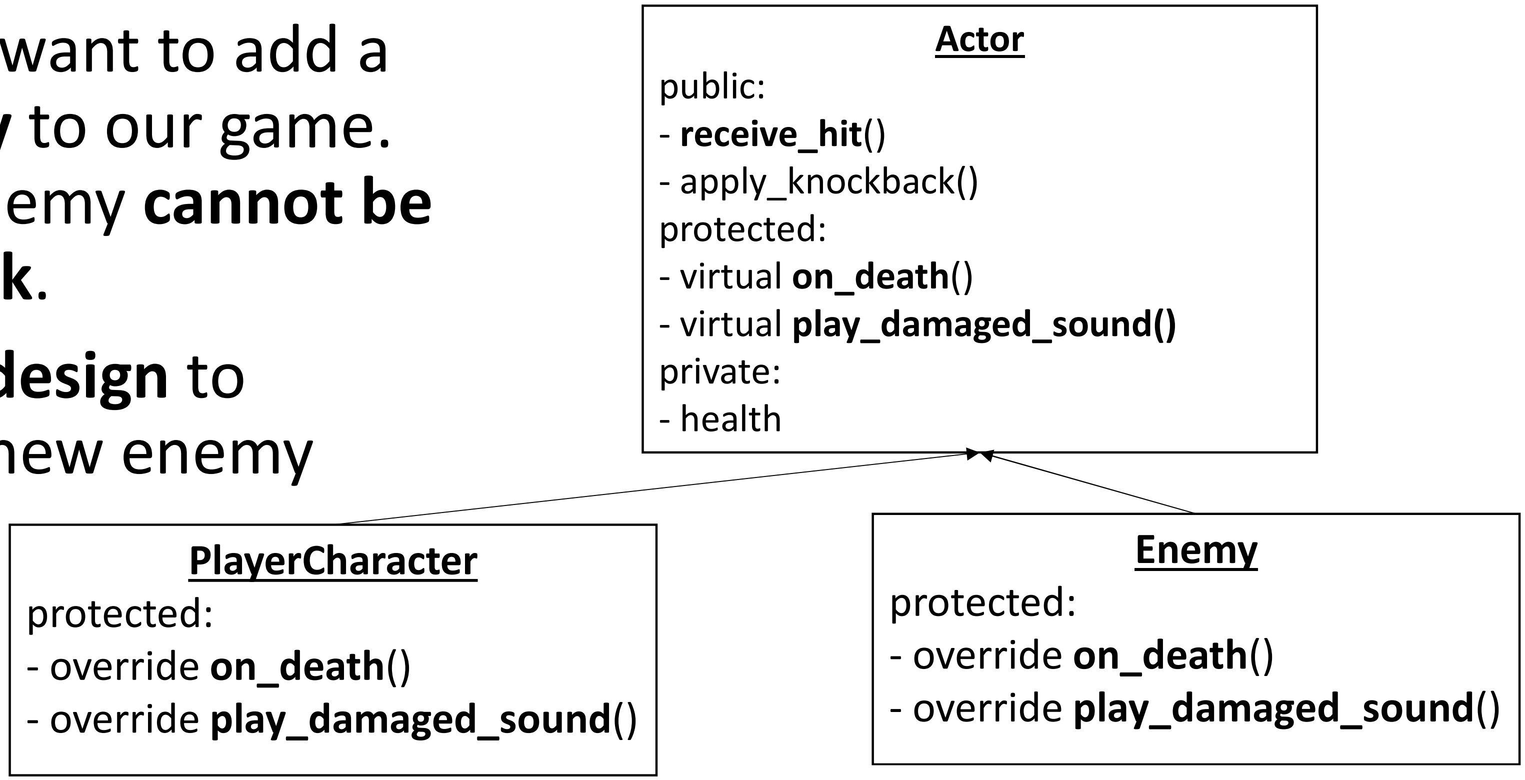
- In the first implementation, the **derived classes called the base class** version of receive_hit()

- In the template method implementation, the non-virtual **base class** receive_hit() **called derived class** methods.

- “Don’t call us, we’ll call you!”

Exercise: Updating our Algorithm

- Suppose we want to add a **TurretEnemy** to our game. The TurretEnemy **cannot be knocked back**.
- **Modify our design** to include this new enemy type.



Exercise: Updating our Algorithm (Solution)

- Suppose we want to add a **TurretEnemy** to our game. The TurretEnemy **cannot be knocked back**.
- **Modify our design** to include this new enemy type.

Actor
public:
- **receive_hit**()
- virtual **apply_knockback**()

protected:
- virtual **on_death**()
- virtual **play_damaged_sound()**

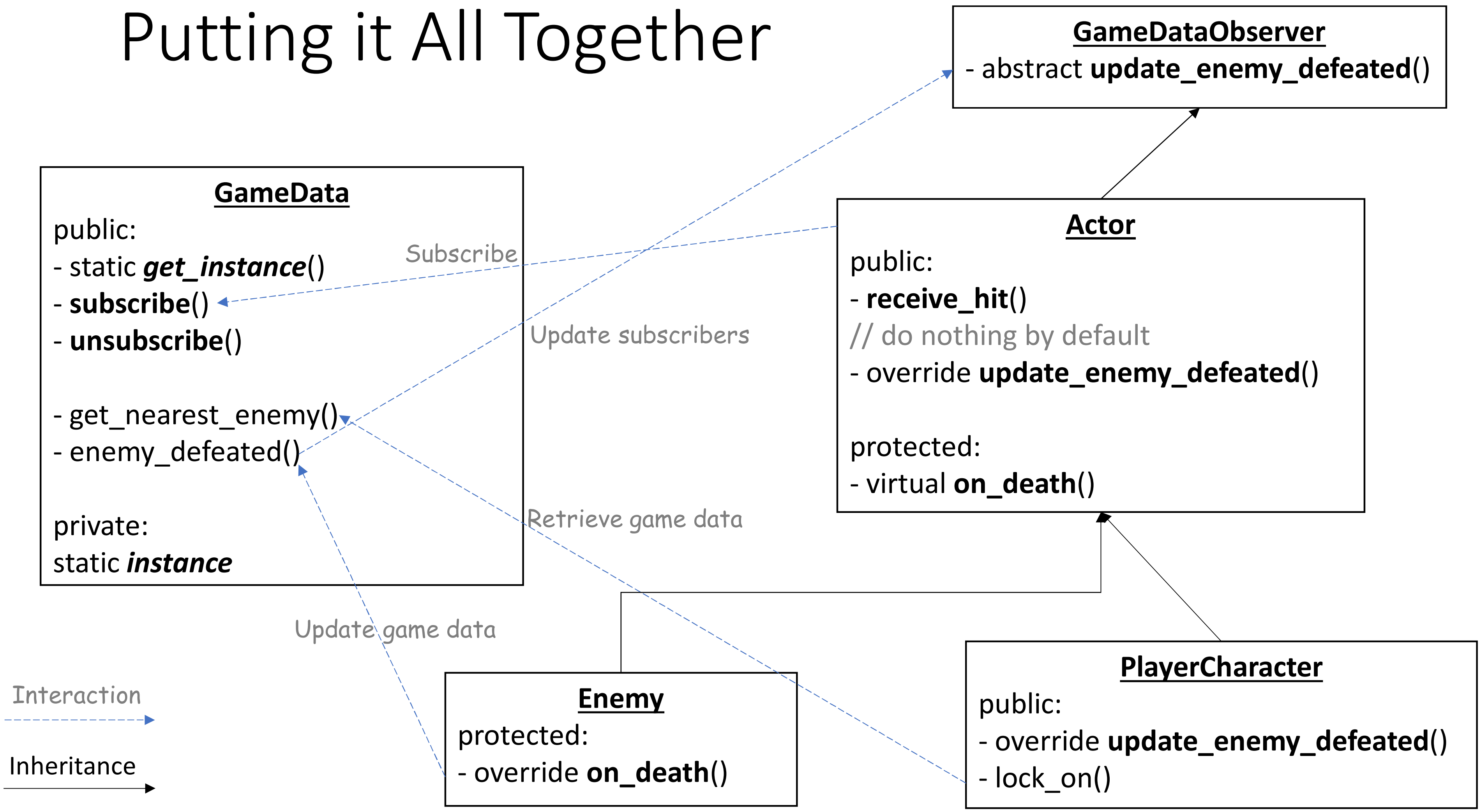
private:
- health

Enemy
protected:
- override **on_death**()
- override **play_damaged_sound**()

PlayerCharacter
protected:
- override **on_death**()
- override **play_damaged_sound**()

TurretEnemy
public:
// This override should be empty
- override **apply_knockback()**

Putting it All Together



Further Reading

- The "Gang of Four" *Design Patterns* book
- EECS 381 course materials:
  - http://www.umich.edu/~eecs381/lecture/notes.html
  - See "Idioms and Design Patterns" PDFs

- Beware the internet
  - "People use a pattern when they shouldn't" **!=** "the pattern is bad
- **Design is challenging**. Take it seriously, but *don't expect to get it right the first time!*
  - Your first design idea is usually not your best.