Patterns and Anti-Patterns

Design Patterns: Elements of Reusable Object-Oriented Software
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES
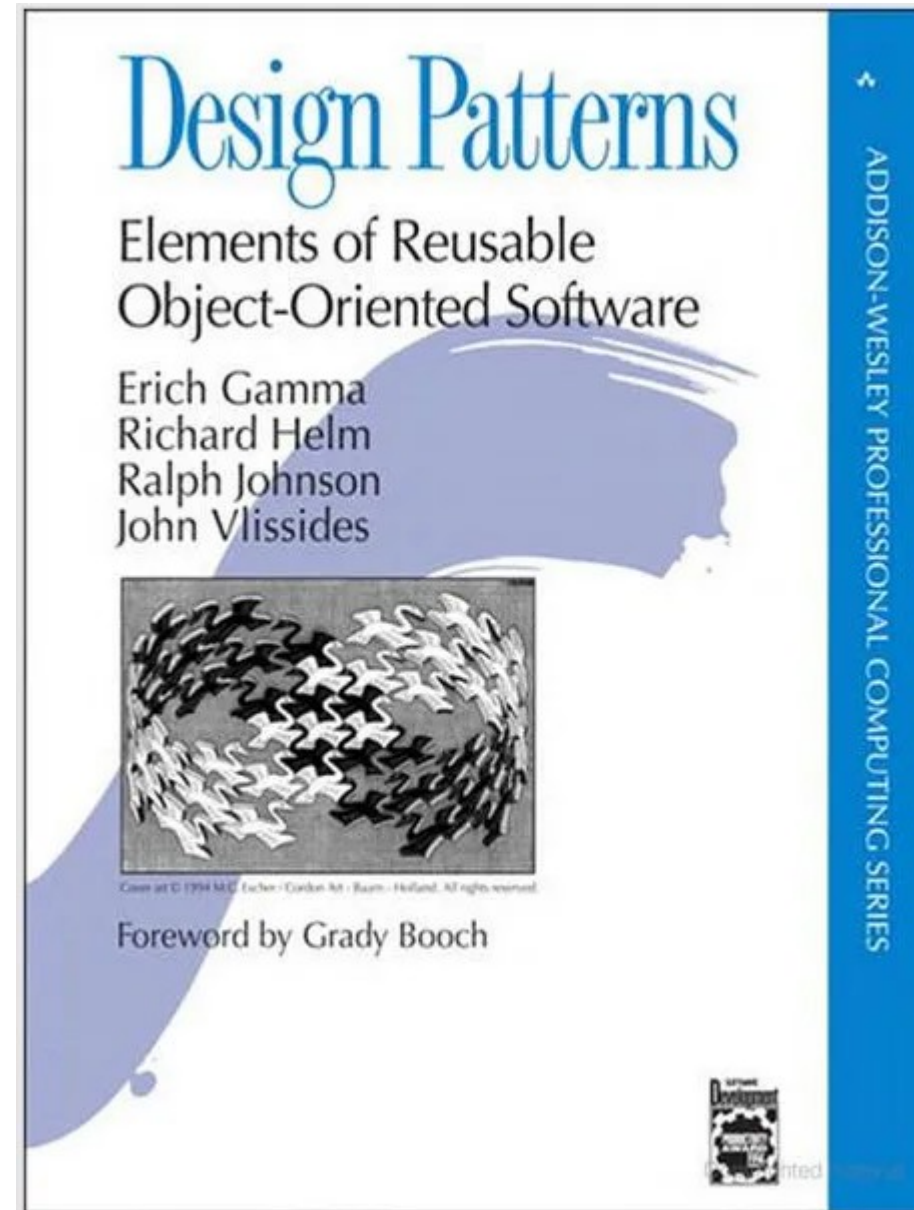
Special thanks for James Perretta!

The Story So Far ...

- We want to deliver and support a quality software product

  - We understand the stakeholder requirements

  - We understand process and maintainability

  - We understand quality assurance

- How should we make process and design designs the first time?

COP 3331   Exam 1

2   Short Answer Questions

11. [10 points] Name and describe the five key phases of software development.

1. denial
2. bargaining
3. Anger
4. depression
5. acceptance

One-Slide Summary

- Software **design patterns** are general, reusable solutions to commonly-occurring problems. They separate the structure of a system from its implementation. They apply in almost all OO languages.

- Every design has tradeoffs. Object-oriented design patterns often trade verbosity or efficiency for extensibility.

- We'll consider **structural**, **creational** and **behavioral** design patterns.

Patterns in Non-Software Design

Multiple choice question

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

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

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

'This is the most extraordinary, exhilarating book.' FAY WELDON

THE SEVEN BASIC PLOTS

Why we tell stories

Overcoming the Monster
RAGS TO RICHES
THE QUEST
VOYAGE AND RETURN
COMEDY
TRAGEDY
REBIRTH

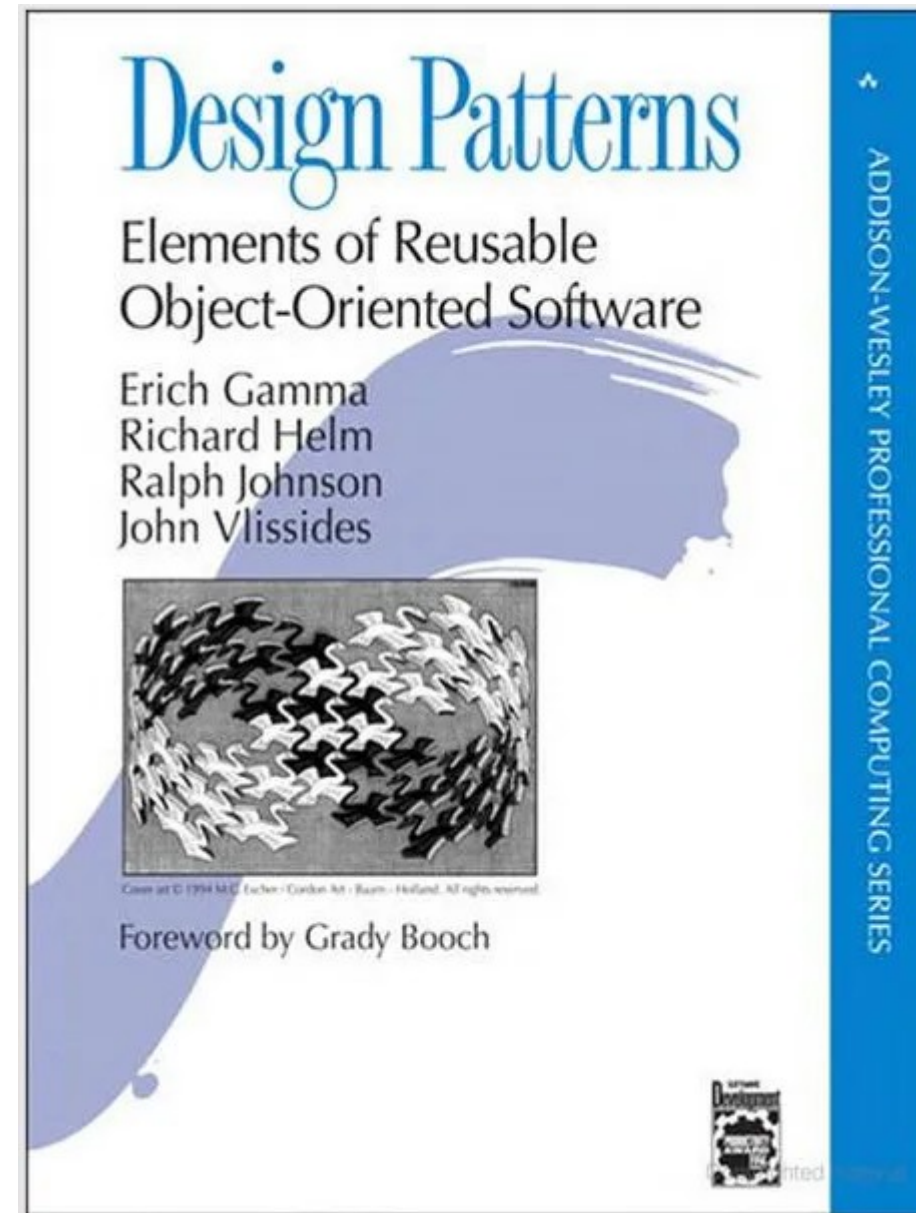CHRISTOPHER BOOKER

BLOOMSBURY

Further Real-World Reading

- **The Design of Everyday Things**
  - design serves as the communication between object and user
  - although people often blame themselves when objects appear to malfunction, it is not the fault of the user but rather the lack of intuitive guidance that should be present in the design
  - behavioral psych + ergonomics
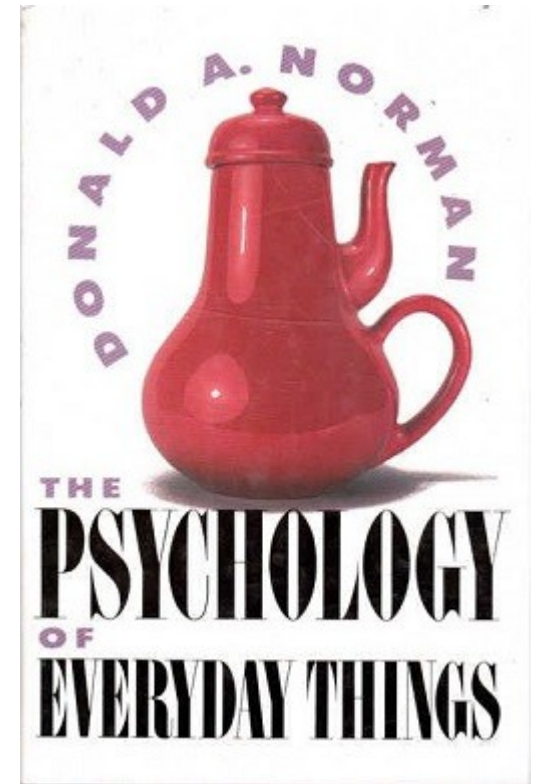
Jargon

- The book popularizing software design patterns is often called the **Gang of Four** book after its four authors

- (Sometimes handy for talking to interviewers or practitioners.)

Design Patterns
Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

High-Level Design Pattern Advice

- Consider code change as a certainty

  - Redesign is expensive. Choosing the right pattern helps avoid it.

- Consider your requirements and their changes

  - Use patterns that fit your current or anticipated needs.

- Consider multiple designs

  - Diagram your designs before writing code.

Structural Patterns

- **Structural design patterns** ease design by identifying simple ways to realize relationships among entities.

- In software, they usually
  - Build new classes or interfaces from existing ones
  - Hide implementation details
  - Provide cleaner or more specialized interfaces

JavaScript Daily Retweeted

Programming Wisdom
@CodeWisdom

"Telling a programmer there's already a library to do X is like telling a songwriter there's already a song about love." – Pete Cordell

11:00 AM · Mar 25, 2019 · Buffer

950 Retweets   **3.3K** Likes

Adapter Design Pattern

- The **adapter design pattern** is a structural design pattern that converts the interface of a class into another interface clients expect.

Adapter Examples (1/2)

- Implementing a Stack interface using a LinkedList interface

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

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

IT'S OKAY APPLE

I PREFER THIS LOOK ANYWAYS..

Adapter Examples (2/2)

- Early implementations of `fstream` in C++

  - ... were simply adapters around the C `FILE` macro

- The autograder used for this course securely runs student code

- It does this via an adapter around a containerization library (e.g., `docker`)

  - Handles quirks of the library

  - Makes sure that certain options are always used

Other Structural Patterns

- The **composite design pattern** allows clients to treat individual objects and groups of objects uniformly

  - e.g., selecting and moving objects in PowerPoint

- The **proxy design pattern** provides a surrogate or placeholder for another object to control access to it

  - std::vector<bool> exposes std::vector<bool>::reference as a method of accessing individual bits. In particular, objects of this class are returned by operator[] by value.
  https://en.cppreference.com/w/cpp/container/vector_bool

Creational Design Patterns

- **Creational design patterns** avoid complexity by controlling object creation so that objects are created in a manner suitable for the situation. They make a system independent of how its objects are created.

- A plain constructor may not allow you to

  - Control how and when an object is used

  - Overcome language limitations (e.g., no default arguments)

  - Hide polymorphic types

The Named Constructor Idiom

- In the **Named Constructor Idiom** you declare the class's normal constructors to be private or protected and make a public static creation method.

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

private:  // Making constructor private
  Llama(string name_in): name(name_in) {}
  string name;
};
```

A Common Problem

- Suppose we need to create and use polymorphic objects without exposing their types to the client

  - Recall: design for maintainability and extensibility. We don't want the client to depend on (and thus "lock in") the actual subtypes.

- The typical solution is to write a function that creates objects of the type we want but returns that object so that it appears to be ("cast to") a member of the base class

The Factory Pattern

- The **factory method pattern** is a creational design pattern that uses factory methods to create objects without having the return type reveal the exact subclass created.

```
Payment * payment_factory(string name, string type)
{
  if (type == "credit_card")
    return new CreditCardPayment(name);
  else if (type == "bitcoin")
    return new BitcoinPayment(name);
  …
}

Payment * webapp_session_payment =
    payment_factory(customer_name, "credit_card");
```

Factory Pattern Variant

- You may also encounter implementations in which special methods create the right type:

```
class PaymentFactory {
public:
  static Payment* make_credit_payment(string name){
    return new CreditCardPayment(name);
  }
  static Payment* make_bc_payment(string name){
    return new BitcoinPayment(name);
  }
};

Payment * webapp_session_payment =
PaymentFactory::make_credit_payment(customer_name);
```

Scenario: Difficulty-Based Enemies

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

- Normal Difficulty → Goomba

- Hard Difficulty → Spiked Goomba

Anti-Patterns

- An **anti-pattern** is a common response to a recurring problem that is usually ineffective and risks being counterproductive.

- A *bad* solution (anti-pattern) would be to check the difficulty at each of the many places in the code related to spawning enemies

- This creates copy-and-paste code (code clones, duplicate code) and hurts maintenance
  - Fixes to one clone may miss others, more lines of code means code comprehension is slower, …

Anti-Patterns

- An **anti-pattern** is a common response to a recurring problem that is usually ineffective and risks being counterproductive.
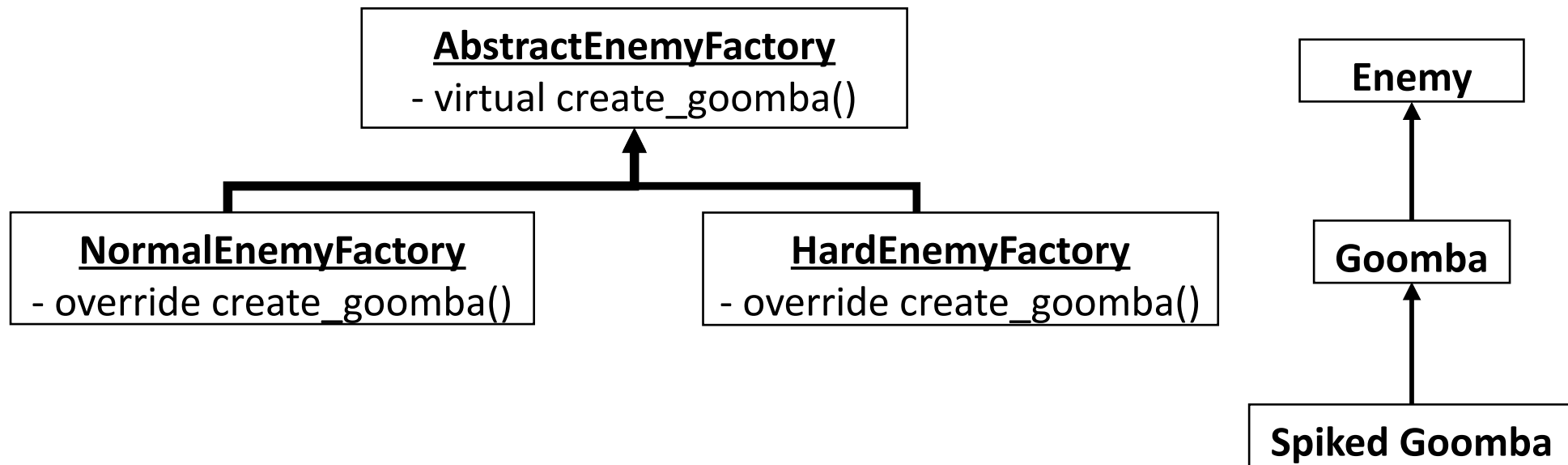
```
Enemy* goomba1 = nullptr;
if (difficulty == "normal")
  goomba1 = new Goomba();
else if (difficulty == "hard")
  goomba1 = new SpikedGoomba();

Enemy* goomba2 = nullptr;
if (difficulty == "normal")
  goomba2 = new Goomba();
else if (difficulty == "hard")
  goomba2 = new SpikedGoomba();
```
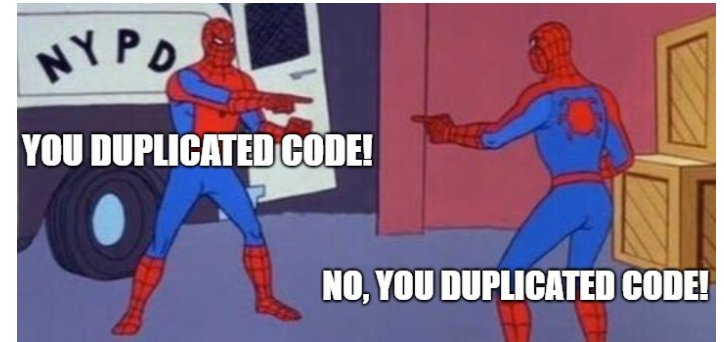
Abstract Factory Design Pattern

- The **abstract factory pattern** encapsulates a group of factories that have a common theme *without* specifying their concrete classes.

```
┌─────────────────────────────┐
│    AbstractEnemyFactory     │
│  - virtual create_goomba()  │
└─────────────────────────────┘
```

```
┌──────────────────────────┐      ┌──────────────────────────┐
│   NormalEnemyFactory      │      │    HardEnemyFactory       │
│ - override create_goomba()│      │ - override create_goomba()│
└──────────────────────────┘      └──────────────────────────┘
```

```
┌──────────┐
│  Enemy   │
└──────────┘
      ↑
┌──────────┐
│  Goomba  │
└──────────┘
      ↑
┌────────────────┐
│ Spiked Goomba  │
└────────────────┘
```

Abstract Factory Design Pattern

- The **abstract factory pattern** encapsulates a group of factories that have a common theme *without* specifying their concrete classes.

```
AbstractEnemyFactory* factory = nullptr;
// Only have to code this "if" part once!
if (difficulty == "normal")
  factory = new NormalEnemyFactory();
else if (difficulty == "hard")
  factory = new HardEnemyFactory();

Enemy* goomba1 = factory->create_goomba();
Enemy* goomba2 = factory->create_goomba();
Enemy* goomba3 = factory->create_goomba();
Enemy* goomba4 = factory->create_goomba();
```

Scenario: Global Application State

- Suppose we have some application state that needs to be globally accessible. However, we need to control how that data is accessed and updated.

- The anti-pattern (bad) solution is to have a naked global variable.

  - Fails to control access or updates!

- A "less bad" solution is to put all of the state in one class and have a global instance of that class.

Acceptability of Global Variables

CAUTION
GLOBAL VARIABLES
ARE STILL BAD

- Global variables are usually a poor design choice. However:

- If you need to access some state everywhere, passing it as a parameter to *every* function clutters the code (readability vs. ...)

  - This is not an argument for using global variables to avoid passing a *few* parameters.

- Or if you need to access state stored outside your program (e.g., database, web API)

- Then global variables may be acceptable

Singleton Design Pattern

- The **singleton pattern** restricts the instantiation of a class to exactly one logical instance. It ensures that a class has only one logical instance at runtime and provides a global point of access to it.

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

Singleton Implementation Example

```java
class Singleton {
  // public way to get "the one logical instance"
  public static Singleton get_instance() {
    if (Singleton.instance == null)
      Singleton.instance = new Singleton();
    return Singleton.instance;
  }
  private static Singleton instance = null;

  private Singleton() { // only runs once
    billing_database = 0;
    System.out.println("Singleton DB created");
  }

  // Our global state
  private int billing_database;
  public int get_billing_count() {
    return billing_database;
  }
  public void increment_billing_count() {
    billing_database += 1;
  }
}
```

What the professor covered

What's on the exam

What you remember

Singleton Implementation Example

```java
class Singleton {
  // public way to get "the one logical instance"
  public static Singleton get_instance() {
    if (Singleton.instance == null)
      Singleton.instance = new Singleton();
    return Singleton.instance;
  }
  private static Singleton instance = null;

  private Singleton() { // only runs once
    billing_database = 0;
    System.out.println("Singleton DB created");
  }

  // Our global state
  private int billing_database;
  public int get_billing_count() {
    return billing_database;
  }
  public void increment_billing_count() {
    billing_database += 1;
  }
}
```

With your team …
What is the output of this code?

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

    Singleton.get_instance().
        increment_billing_count();
    bills = Singleton.get_instance().
        get_billing_count();
    System.out.println(bills);
  }
}
```

Singleton Implementation Example

```java
class Singleton {
  // public way to get "the one logical instance"
  public static Singleton get_instance() {
    if (Singleton.instance == null)
      Singleton.instance = new Singleton();
    return Singleton.instance;
  }
  private static Singleton instance = null;

  private Singleton() { // only runs once
    billing_database = 0;
    System.out.println("Singleton DB created");
  }

  // Our global state
  private int billing_database;
  public int get_billing_count() {
    return billing_database;
  }
  public void increment_billing_count() {
    billing_database += 1;
  }
}
```

```
Output:
Singleton DB created
0
1
```

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

    Singleton.get_instance().
        increment_billing_count();
    bills = Singleton.get_instance().
        get_billing_count();
    System.out.println(bills);
  }
}
```

28

Singleton.get_instance()

- Could we avoid typing Single.get_instance() so many times by doing this at all of the points in our program that use the singleton?

  ```
  Single s = Singleton.get_instance();

  System.out.println(s.get_billing_count());

  … // later

  System.out.println(s.get_billing_count());
  ```

- Is this a good idea or not?

Singleton.get_instance() Warning

```
Single                                    ;
System.                              unt());
… // la
System.                              unt());
```

WE DON'T DO THAT HERE

- This is a bad idea. There is no guarantee that Singleton.get_instance() will return the same pointer (same object) every time it is called. (It may return different *concrete* copies of the same *logical* item.)

Singleton Design Scenario

- Suppose we are implementing a computer version of the card game Euchre. In addition to a few abstract datatypes, we have a **Game** class that stores the state needed for a game of Euchre. When started, our application prototype plays one game of Euchre and then exits.

- *(With your team ...)* Should we make **Game** a singleton?

Scenario Considerations

- Making Game a Singleton is tempting

  - There is only one Game instance in our application

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

- We should only use the Singleton pattern when current or future requirements dictate that only one instance should exist.

  - Singleton is not a license to make everything global.

Trivia: Mexican History

- This priest and professor was a leader in the Mexican War of Independence (1810-1820). After giving the famous "Cry of Dolores" speech he gathered an army of 90,000 farmers and fought the Spanish. His troops fled and he was betrayed and executed. Although he did not live to see independence in 1821, the day of his speech (16 September 1810) is officially recognized as the day of Mexican independence.

Trivia: Social Media

- Name the social media platform most associated with memes about:
  - color vision of shrimp
  - Apollo's dodgeball
  - Goncharov
  - Spiders Georg

Trivia: Geography

- This national capital is the most populous city in Western Asia. Almost all inhabitants speak Persian (تهران). It is home to the Azadi Tower memorial, hosted a WWII conference between Roosevelt, Stalin and Churchill, and despite air pollution issues, it is a popular migration destination.
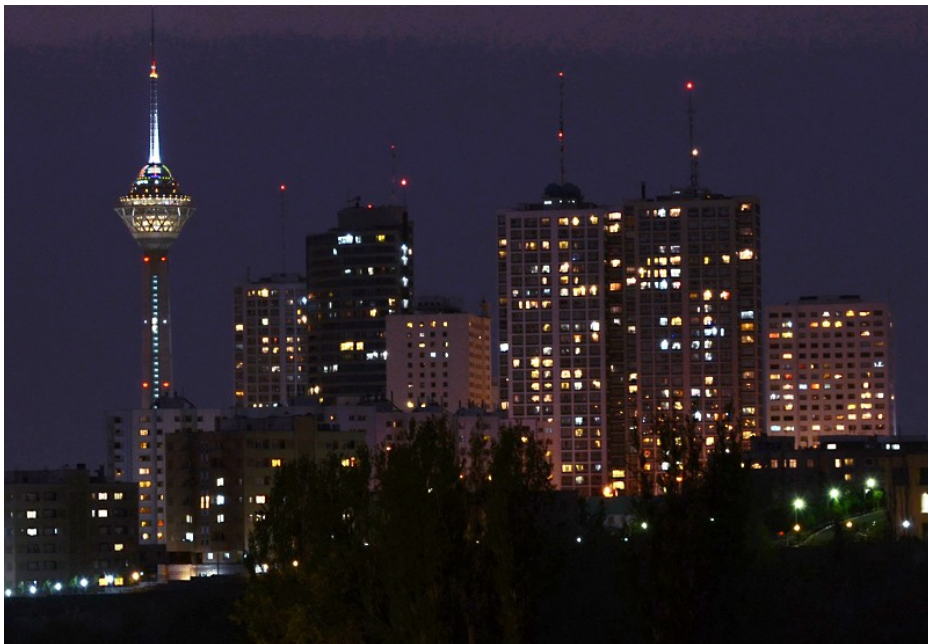
Trivia: US Toys

- This Mattel fashion doll franchise (and associated web series) was created in 2010. It features fictional "ghouls" and "mansters", such as vegan vampire Draculaura and clumsy Frankie Stein, attending the eponymous school. It was so successful that rival lines such as Bratzillaz and Equestria Girls are viewed as direction reactions attempting to cash in on the same trend.

Psychology: Reaction and Information

- How long does it take you to choose from among multiple stimuli, even when you know the right answer?

- An early experiment presented subjects with a few lamps. Each lamp was labeled (e.g., A, B, C, etc.). Every five seconds, one of the lamps would light up. The subject was asked to press the key, as quickly and as accurately as possible, corresponding to the lamp that lit up.

- While only one lamp was ever lit, the experimenter varied the total number of other lamps (e.g., from 2 to 10).

- How does your reaction time vary as a function of the number of choices?

Psychology: Hick's Law

- Given *n* equally probable choices, the average reaction time *T* required for a human to choose among them is: $T = b \cdot \log_2(n+1)$

  - *b* is an empirically-learned constant

- Increasing the number of choices increases decision time logarithmically. The amount of time taken to process a certain amount of bits is known as the rate of gain of information.

[Hick, W. E. (1 March 1952). *"On the rate of gain of information".* Quarterly Journal of Experimental Psychology. 4 (1): 11–26]
[Hyman, R (March 1953). *"Stimulus information as a determinant of reaction time".* Journal of Experimental Psychology. 45 (3): 188–96.]
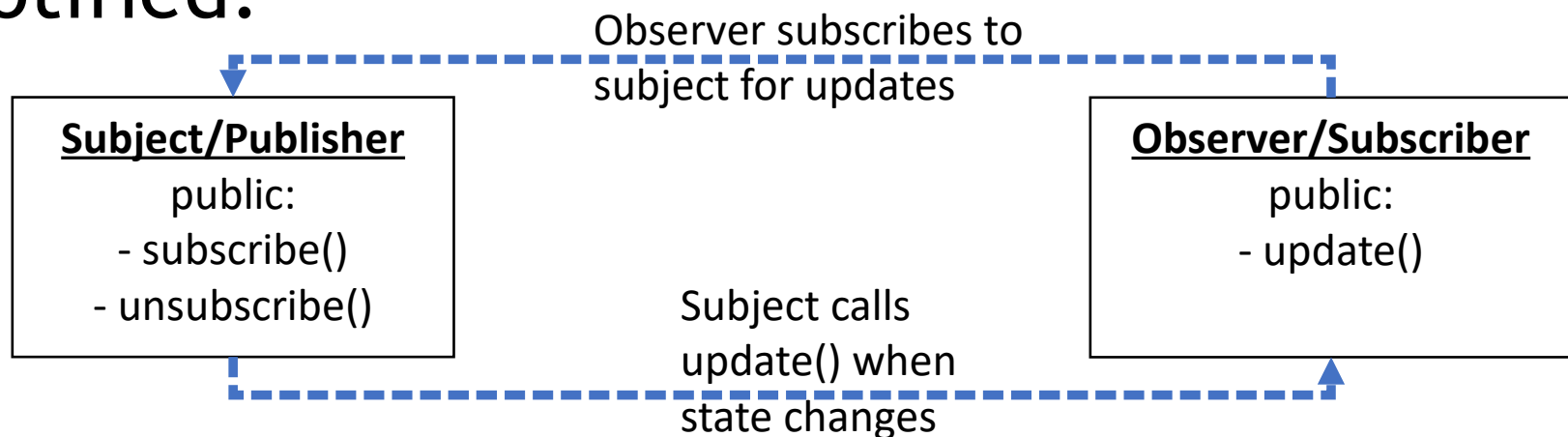
Psychology: Hick's Law

- Implications for SE:

- Hick's Law is often used to justify menu design decisions in human interfaces – from restaurant menus to UI design in computing. Users given many choices have to take time to interpret and decide, work they typically don't want (cf. analysis paralysis, Mac vs. Windows design philosophy, etc.). Why don't we like voluminous bug-finding tool output again?

Behavioral Design Patterns

- **Behavioral design patterns** that support common communication patterns among objects. They are concerned with algorithms and the assignment of responsibilities between objects.

- The **iterator pattern** is a common behavioral design pattern. It provides a uniform interface for traversing containers regardless of how they are implemented.

Observer Design Pattern

- The **observer pattern** (also called "publish-subscribe") allows dependent objects to be notified automatically when the state of a subject changes. It defines a one-to-many dependency between objects so that when one object changes state, all of it dependents are notified.

Observer subscribes to
subject for updates

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

Observer/Subscriber
public:
- update()

Subject calls
update() when
state changes

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

Observer Pattern Exercise

- How many times is "Received update" printed?

```
class Subject {
  public static void subscribe(Observer obs) {
    subscribers.Add(obs);
  }
  public static void unsubscribe(Observer obs) {
    subscribers.Remove(obs);
  }
  public static void change_state() {
    foreach (Observer obs in subscribers) {
      obs.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();
  }
}
```
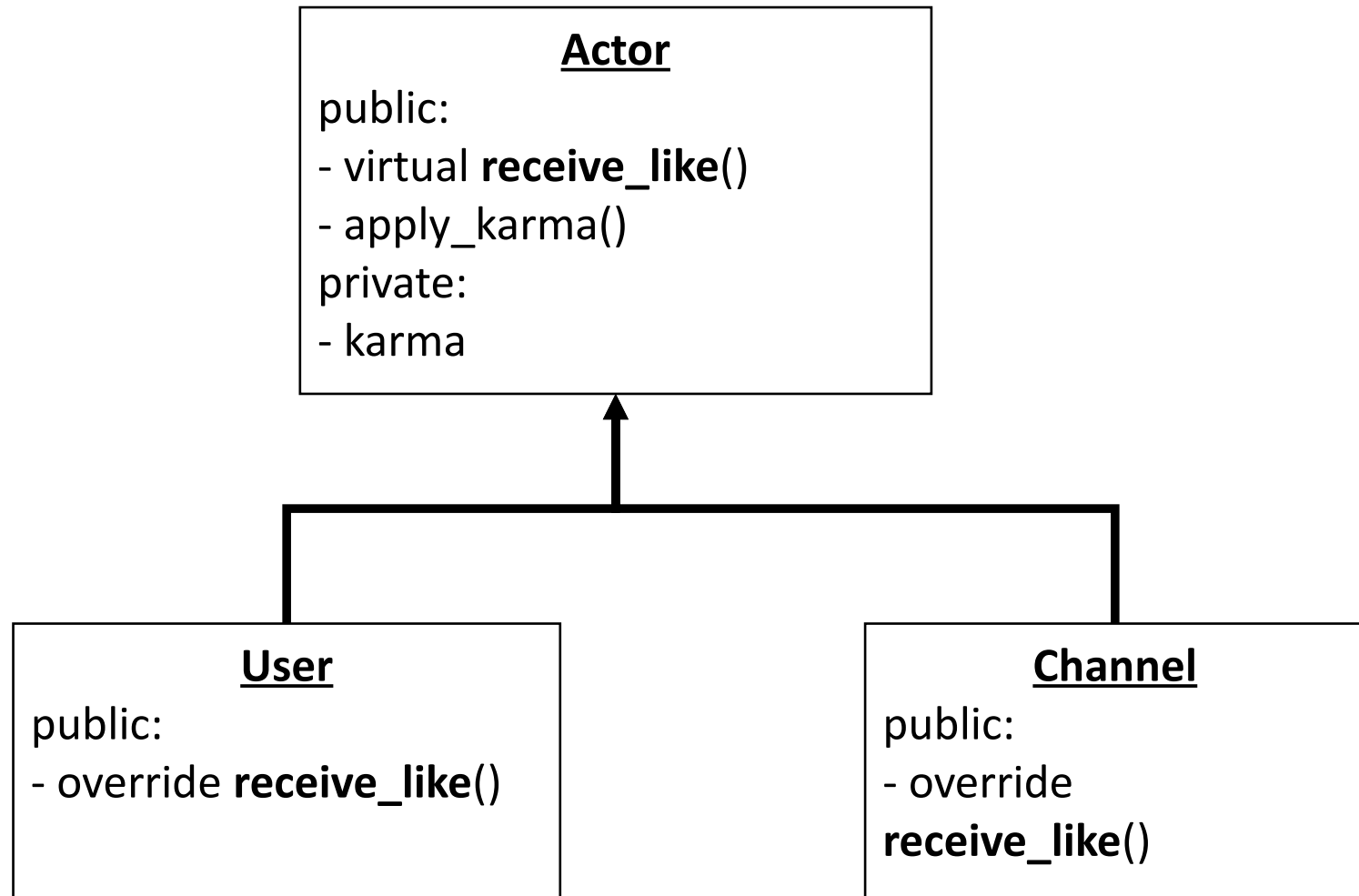
Observer "update_" Functions

- Having multiple "update_*XYZ*" functions, one for each type of state change, keeps messages granular

  - Observers that do not care about a particular type of update can ignore it (via an empty implementation of the update function)

- Generally it is better to pass the newly-updated data as a parameter to the update function (push) as opposed to making observers fetch it each time (pull)

Scenario: "Likes" In Social Streaming Website

- Suppose we're building a social video streaming website with users and channels.

- Both users and channels can receive *likes* (for good comments or good videos).

- When a user *or* channel receives a like, it gets karma.

- At 50,000 karma, a channel gets a trophy.

- At 50,000 karma, a user gets ad-free access.
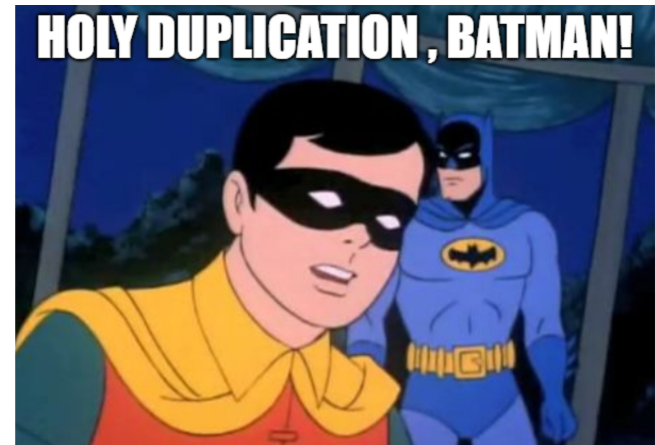
Likes: First Design

Actor

public:
- virtual **receive_like**()
- apply_karma()
private:
- karma

User

public:
- override **receive_like**()

Channel

public:
- override
receive_like()

Note: receive_like is called on an Actor when someone likes its comment or video, etc.

Likes: Anti-Pattern Observations

```
class Actor {
  public virtual void receive_like() {
    karma += 1;
  }
  public float get_karma() { return karma; }
  private float karma = 42;
  public void note_karma() {
    Console.WriteLine("Some karma received!");
  }
}
```
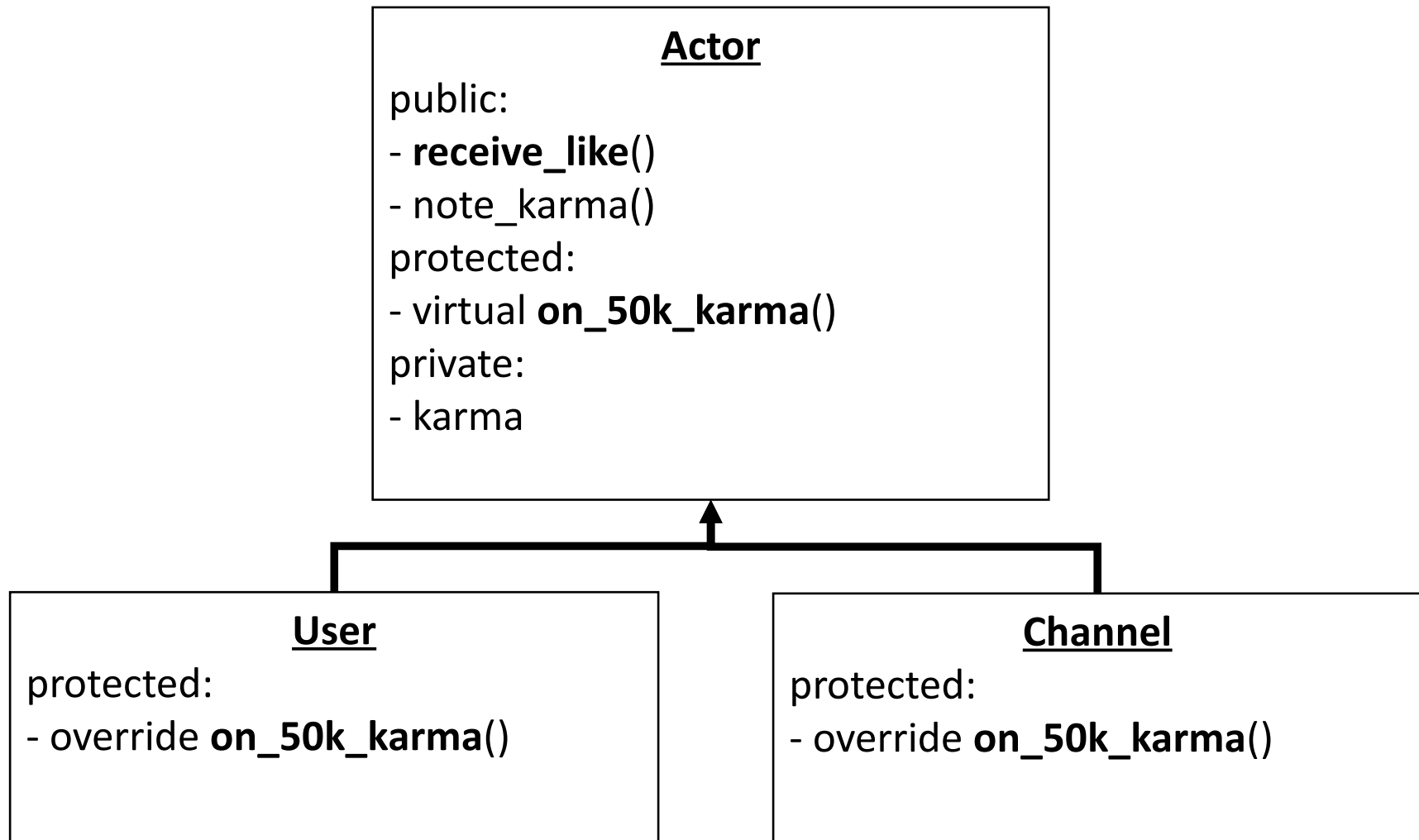

HOLY DUPLICATION , BATMAN!

```
class Channel: Actor {
  public override void receive_like() {
    base.receive_like();
    if (get_karma() >= 50000)
      Console.WriteLine("Channel trophy!");
    else
      note_karma();
  }
}
```

```
class User: Actor {
  public override void receive_like() {
    base.receive_like();
    if (get_karma() >= 50000)
      Console.WriteLine("Ad-free access!");
    else
      note_karma();
  }
}
```

Template Method Design Pattern

- The **template method** behavioral design pattern involves a method in a superclass that operates in terms of high-level steps that are implemented by abstract helper methods provided by concrete implementations.

- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method design lets subclasses redefine certain steps of that algorithm without changing the algorithm's structure.

Likes: Template Method

Actor

public:
- **receive_like**()
- note_karma()
protected:
- virtual **on_50k_karma**()
private:
- karma

User

protected:
- override **on_50k_karma**()

Channel

protected:
- override **on_50k_karma**()

Likes: Template Method Implementation

```
class Actor {
  public void receive_like() {
    karma += 1;
    if (get_karma() >= 50000)
      on_50k_karma();
    else
      note_karma();
  }
  protected virtual void on_50k_karma() {}

  // Other members same as before
}
```

```
class Channel: Actor {
  protected override void on_50k_karma() {
    Console.WriteLine("Channel trophy!");
  }
}

class User: Actor {
  protected override void on_50k_karma() {
    Console.WriteLine("Ad-free access!");
  }
}
```
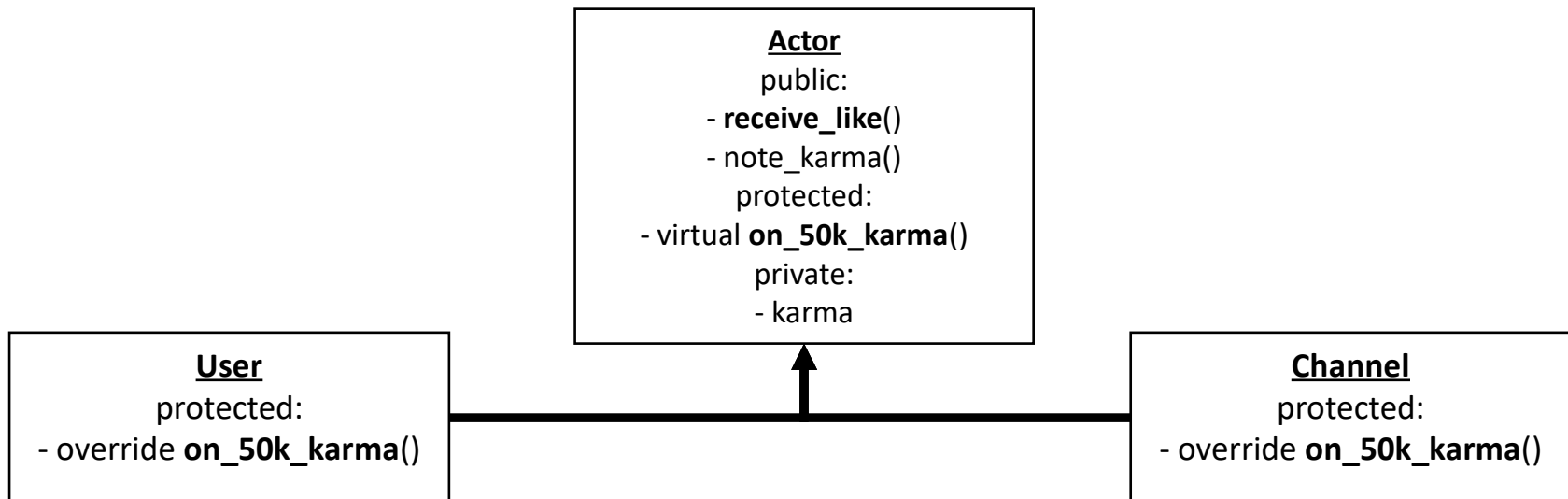
Template Method
The "Hollywood Principle"

- In the first (anti-pattern) implementation, the derived class called the base class version of receive_like()

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

- "Don't call us, we'll call you!"

Exercise

- Suppose we want to add an AffiliateChannel to our setup. An AffiliateChannel does not receive a trophy on 50,000 karma, but instead received *nothing*.

- Modify our design to include this new type.

Actor
public:
- **receive_like**()
- note_karma()
protected:
- virtual **on_50k_karma**()
private:
- karma

User
protected:
- override **on_50k_karma**()

Channel
protected:
- override **on_50k_karma**()

Solution
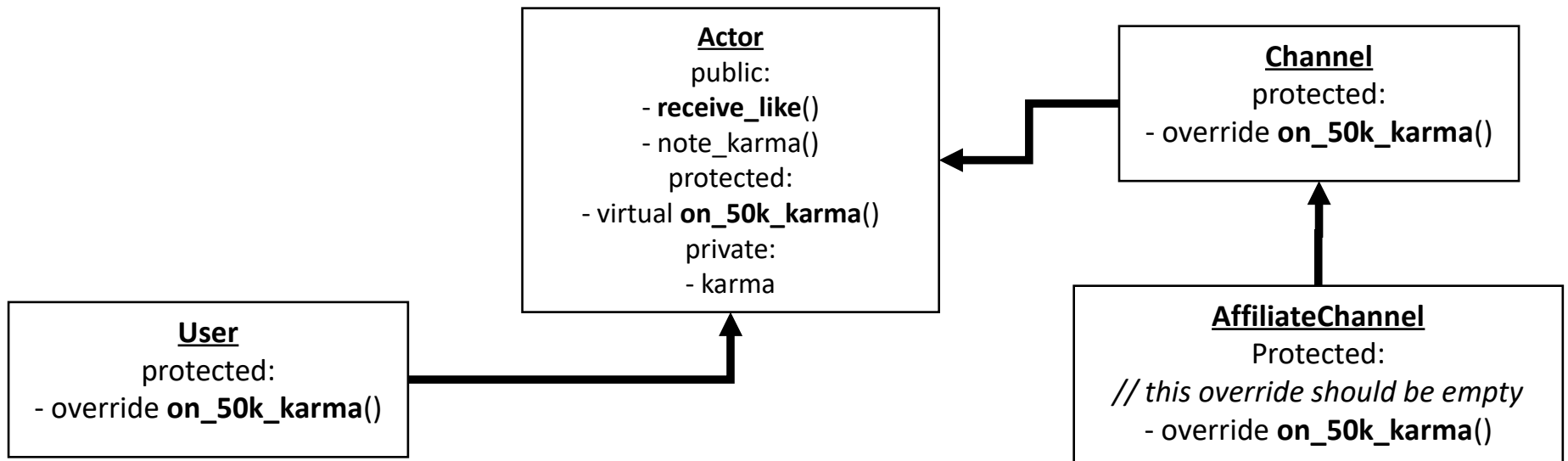
- Suppose we want to add an AffiliateChannel to our setup. An AffiliateChannel does not receive a trophy on 50,000 karma, but instead received *nothing*.

- Modify our design to include this new type.

Actor
public:
- **receive_like**()
- note_karma()
protected:
- virtual **on_50k_karma**()
private:
- karma

Channel
protected:
- override **on_50k_karma**()

User
protected:
- override **on_50k_karma**()

AffiliateChannel
Protected:
// this override should be empty
- override **on_50k_karma**()

Questions?

- Further reading from EECS 381
  - http://www.umich.edu/~eecs381/lecture/notes.html
  - See "Idioms and Design Patterns" PDFs