Assess Yourself

Describe the Observer pattern and the general "class" of problems it can be applied to.

Provide at least one real-world example where the Observer pattern could be applied.

Assess Yourself

The Observer pattern (also known as the *publish/subscribe* model) is used in situations where one or more classes (the observers) must *observe* and *react* to another object's (the subject) state.

Example: A robotic system has a number of sensors that are *observed* by the robot's central control system. When a sensor is *activated*, the central system is *notified* so the appropriate response can be taken.

SWEN20003
Object Oriented Software Development

Games and Event Driven Programming

Semester 1, 2019

The Road So Far

- Java Foundations
- Classes and Objects
- Abstraction
- Advanced Java
  - Generic Classes
  - Generic Programming
  - Exception Handling
  - Software Testing and Design
  - Design Patterns
- Software Development Tools

Lecture Objectives

After this lecture you will be able to:

- Describe the event-driven programming paradigm
- Describe where it can be applied, and how
- Implement basic event-driven programs in Slick
- Describe some basic techniques in game design

Event Programming

How do programs work?

```
public static void main(String args[]) {

    Scanner scanner = new Scanner(System.in);

    String firstName = scanner.nextLine();
    String lastName = scanner.nextLine();

    String fullName = String.format("%s %s",
                                    firstName, lastName);

    System.out.format("Welcome %s!\n", fullName);
}
```

How do programs work?

Step 1: Create variable `scanner`

Step 2: Instantiate `Scanner` object and allocate to `scanner` variable

Step 3: Create variable `firstName`

Step 4: Call `readLine` method in `scanner`

Step 5: Allocate result to `firstName`

Step 6: ...

Step 7: ...

How do programs work?

Keyword

Sequential Programming: A program that is run (more or less) top to bottom, starting at the beginning of the `main` method, and concluding at its end.

- Useful for "static" programs
- Constant, unchangeable logic
- Execution is the same (or very similar) each time

Let's make our programs more "dynamic"...

Event-Driven Programming

Keyword

State: The properties that define an object or device; for example, whether it is "active".

Keyword

Event: Created when the *state* of an object/device/etc. is altered.

Keyword

Callback: A method triggered by an event.

Event-Driven Programming

Keyword

Event-Driven Programming: Using *events* and *callbacks* to control the flow of a program's execution.

Have we seen similar behaviour before?

- Exception handling
- Observer pattern

Let's make an event-driven system in Slick!

Slick Case Study

```
public interface KeyListener
extends ControlledInputReciever
```

Describes classes capable of responding to key presses

Author:
kevin

Method Summary	
void	keyPressed(int key, char c) Notification that a key was pressed
void	keyReleased(int key, char c) Notification that a key was released

Where could this be useful?

Slick Case Study

```
public class Player implements KeyListener {

    @Override
    public boolean isAcceptingInput() {
        return true;
    }

    @Override
    public void keyPressed(int key, char c) {
        /*
        Let's us respond to key presses without
        needing to access the Input object directly
        */
    }

}
```

Slick Case Study

```
public class World {

    public void addListeners(GameContainer gc) {
        Input input = gc.getInput();

        input.addKeyListener(new Player(...));
        input.addKeyListener(new Player(...));
        input.addKeyListener(new Player(...));
        input.addKeyListener(new Player(...));
    }

}
```

Note: This is purely an example, not a recommendation for Project 2.

Event-Driven Programming

What makes this approach better?

Using an event-driven approach allows us to:

- Better *encapsulate* classes by hiding their behaviour
- Avoid the `World` having to explicitly send the `Player` information about the input
- Easily add/remove behaviour to classes
- Easily add/remove additional *responses*

Assess Yourself

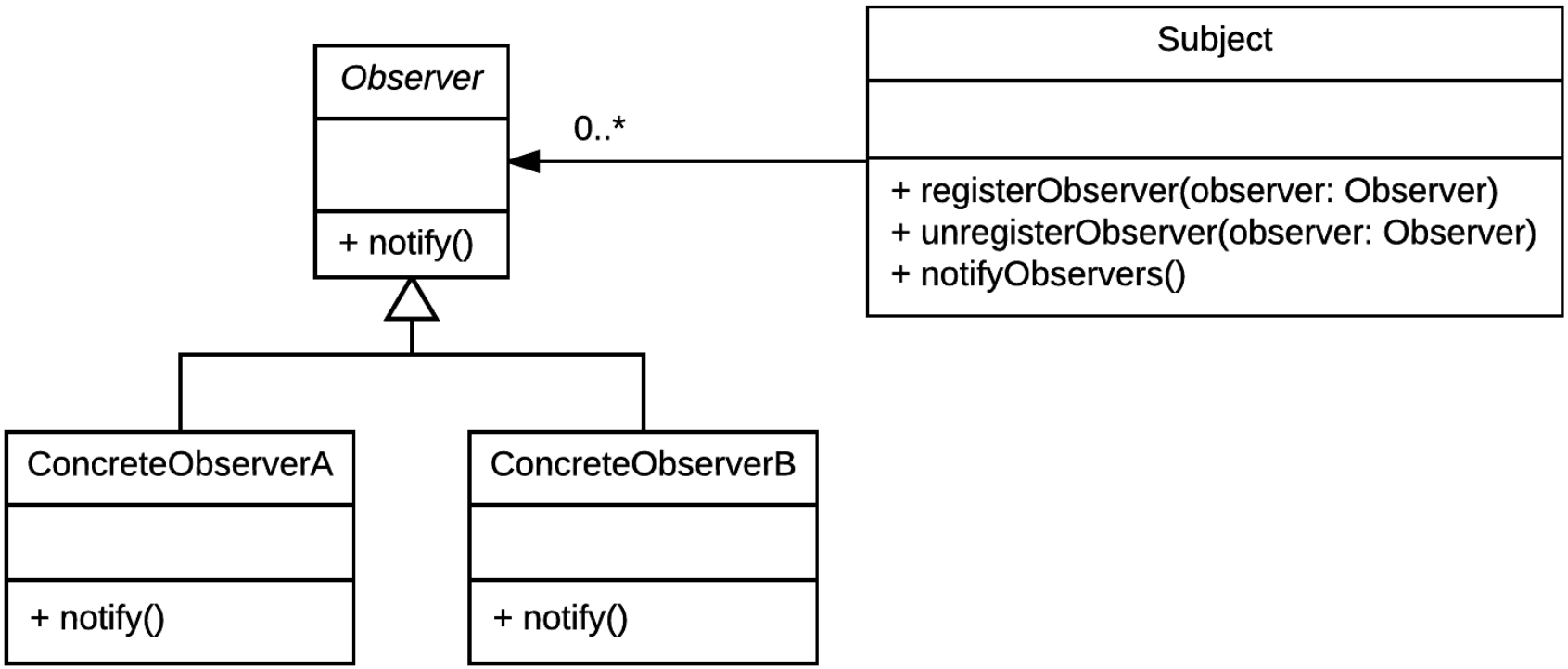
When playing a game, what *events* might we respond to?

- Mouse
- Keyboard
- Touch
- Controller (e.g. XBox controller)
- Time

Assuming we could respond to those events, what *features* could we add? Think about games you may have played, and how they work.

- Mouse: Menu buttons, GUI controls, click-to-move
- Keyboard: movement, "shooting", special powers
- Touch/Controller: similar to keyboard
- Time: time-based completion, do-x-every-second (recover health, generate enemy)

Observer Pattern Reminder



The Event Loop

```
public class Robot {
    public static void main(String args[]) {
        while (true) {
            if (isCommandSent()) {
                respondToCommand();
            } else if (isLowPower()) {
                respondToLowPower();
            } else if (isStuck()) {
                respondToIsStuck();
            } else if (hasCollided()) {
                respondToCollision();
            } else if (isSensorBroken()) {
                respondToBrokenSensor();
            } else if (...) {
                ...
            }
        }
    }
}
```

What would you say is the "problem" in this code?

The Event Loop

Keyword

Polling: (also called *sampling*) relies on the program actively enquiring about the state of an object/device/etc.

What are some disadvantages of this approach?

- Lots of "waiting" for something to happen
- Lots of time/effort wasted "asking"
- Always responds in the same order
- Can't "escape" from one method to respond to something else (potentially) urgent

Asynchronous Programming

Keyword

Interrupt: A signal generated by hardware or software indicating an event that needs *immediate* CPU attention.

Keyword

Interrupt Service Routine: Event-handling code to respond to interrupt signals.

Interrupts are used to control program flow at a *very low level*, **immediately** taking control of the program, e.g.:

- Exception/error handling
- Activating a sleeping process/task
- Device drivers (mouse, keyboard)

Robotics Case Study (Bad)

```
public class Robot {
    public static void main(String args[]) {
        while (true) {
            if (isCommandSent()) {
                respondToCommand();
            } else if (isLowPower()) {
                respondToLowPower();
            } else if (isStuck()) {
                respondToIsStuck();
            } else if (hasCollided()) {
                respondToCollision();
            } else if (isSensorBroken()) {
                respondToBrokenSensor();
            } else if (...) {
                ...
            }
        }
    }
}
```

Robotics Case Study (Good)

```
// Operates in a separate thread
public class DistanceSensor {

    public void detectCollision() {
        while (true) {
            // If a collision is imminent,
            // we should alert the main thread
            if (destructionImminent()) {
                mainThread.interrupt();
            }
        }
    }

}
```

Note: No one builds robots in Java.

Event-Driven Programming

Real-world examples:

- Graphical User Interfaces (GUIs)
- Web development/Javascript
- Embedded Systems/Hardware

Event-driven and *asynchronous* programming are very powerful techniques, and good tools to have in your arsenal.

Game Design

Inheritance-based Game Design

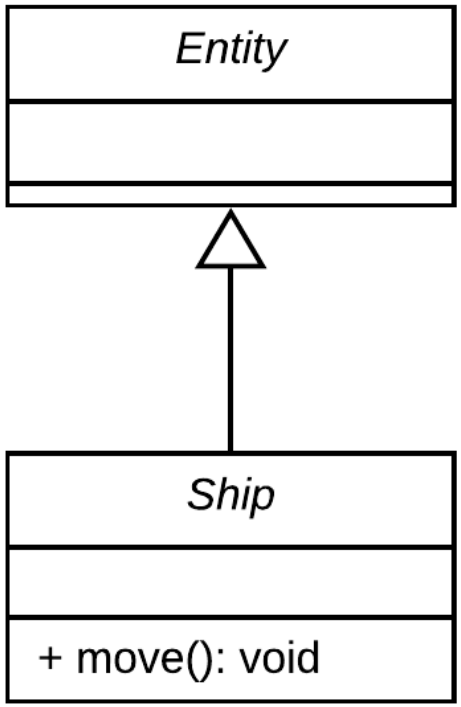
The most obvious way to design a game in an object-oriented language is to use inheritance. We could

- create an `Entity` abstract base class to represent game objects
- inherit from `Entity` to create new types of objects
- take advantage of **polymorphism**
- use the **Factory** design pattern

Inheritance

Many distinct `Entitys` may have similar behaviours.

All `Ships` in *Shadow Wars* can *move*, so one obvious approach is a design like this:

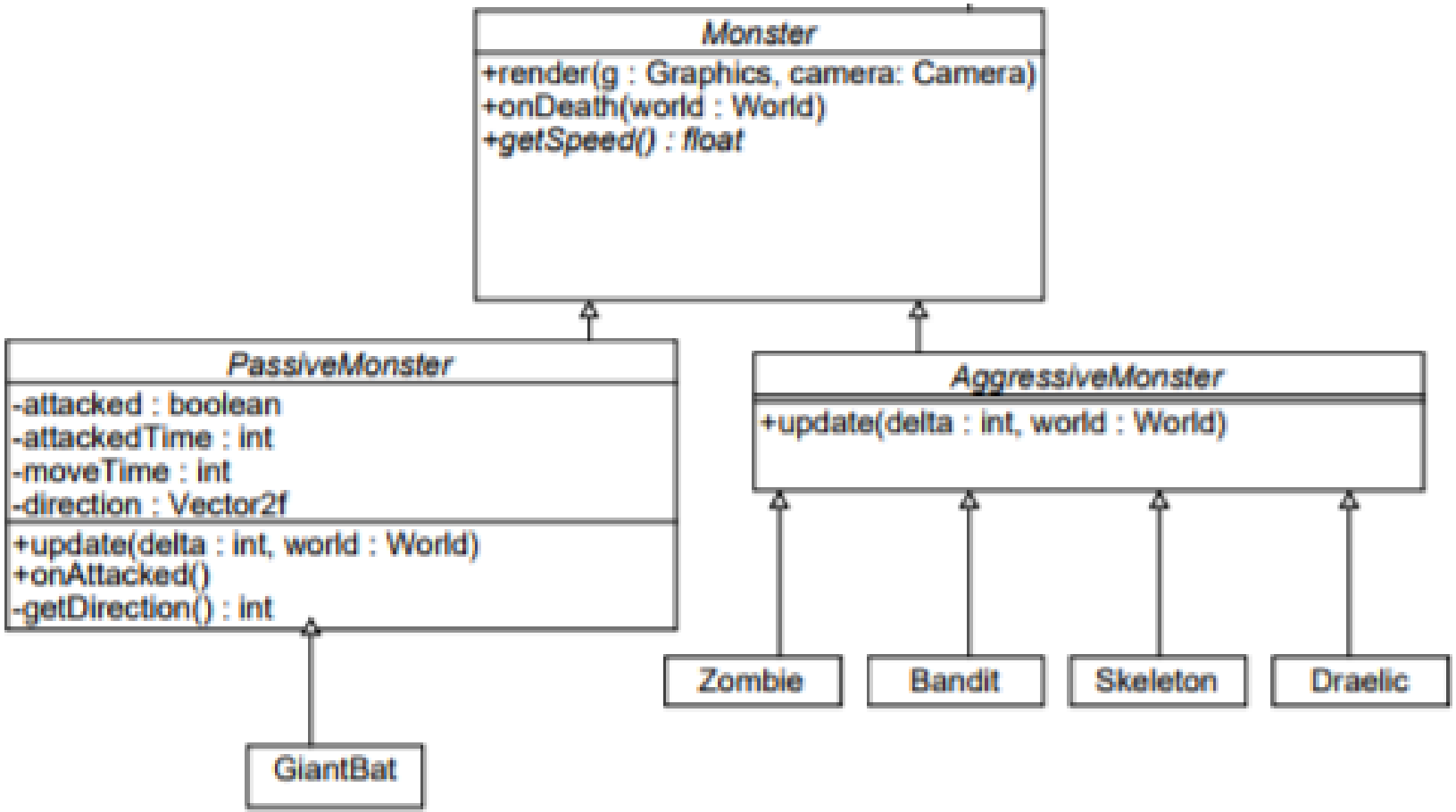


Zombies and bandits and bats, oh my!

You are designing a role-playing game, and you are told monsters come in two varieties; *aggressive* monsters that chase and attack the player, and *passive* monsters that simply try to run away.

You decide to implement this using inheritance.

UML diagram

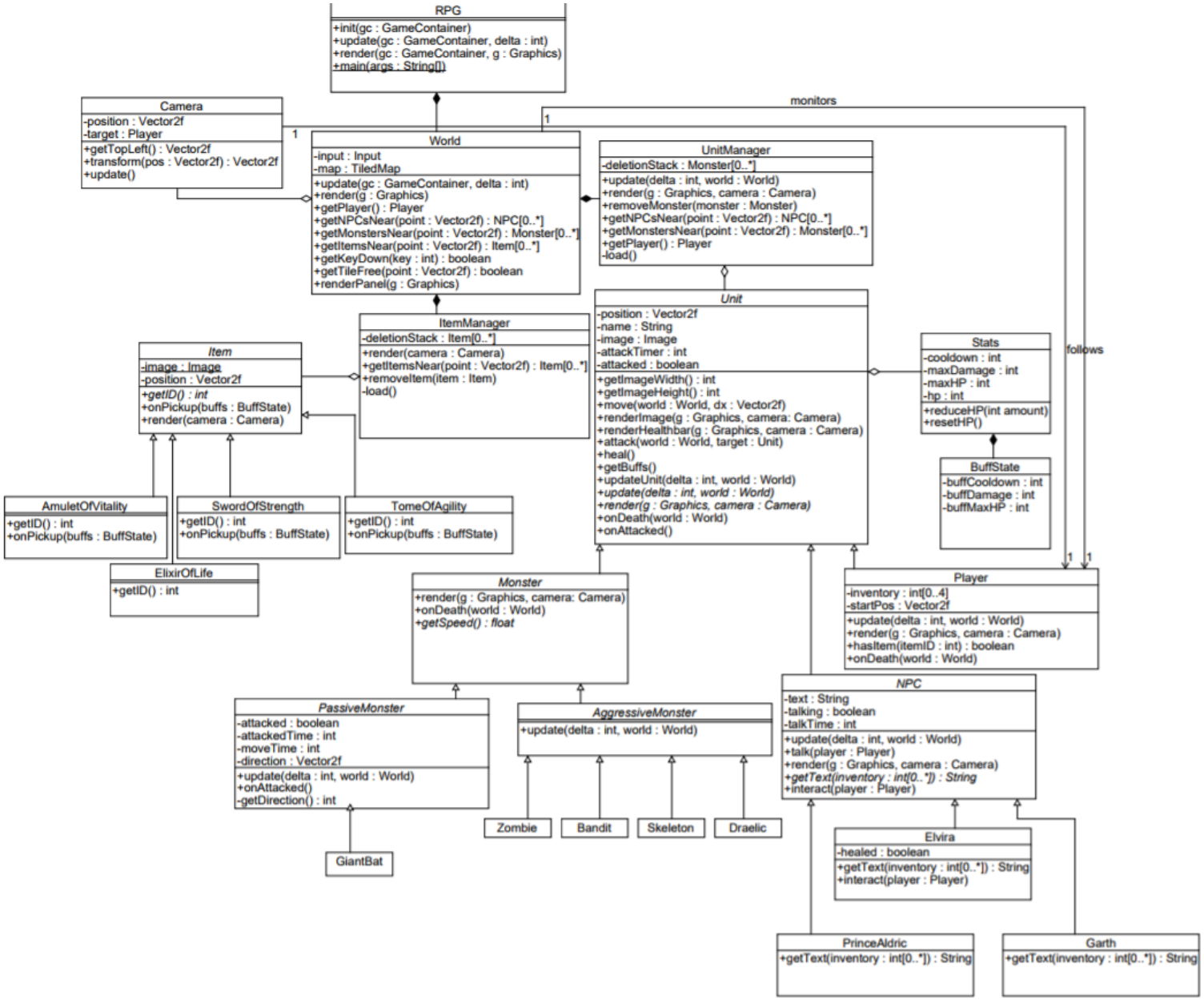

Monster
+render(g : Graphics, camera: Camera)
+onDeath(world : World)
+getSpeed() : float
PassiveMonster
-attacked : boolean
-attackedTime : int
-moveTime : int
-direction : Vector2f
+update(delta : int, world : World)
+onAttacked()
-getDirection() : int
AggressiveMonster
+update(delta : int, world : World)
Zombie
Bandit
Skeleton
Draelic
GiantBat

Pitfall: Flexibility

What if we wanted a monster that was sometimes passive and sometimes aggressive?

- inherit from both – can't do that...
- use an interface? Doesn't really fit...

An example game design



Pitfall: Complexity

This is a fairly small game, and that diagram is already looking pretty terrifying.

It's hard to tell from the small image, but there's also lots of data awkwardly passed around.

Solution: *Composition over inheritance*

Let's try a different approach. What if we broke down functionality of each object into its key `Components`? That is, those parts that describe it completely?

An `Entity` then becomes a **composition** of its `Components`.

A monster as a group of components

```
class Monster {
    private Position position;
    private Image image;
    private Aggressor aggressor;

    public void render() { image.render(position); }
    public void update() { aggressor.update(position); }
}
```

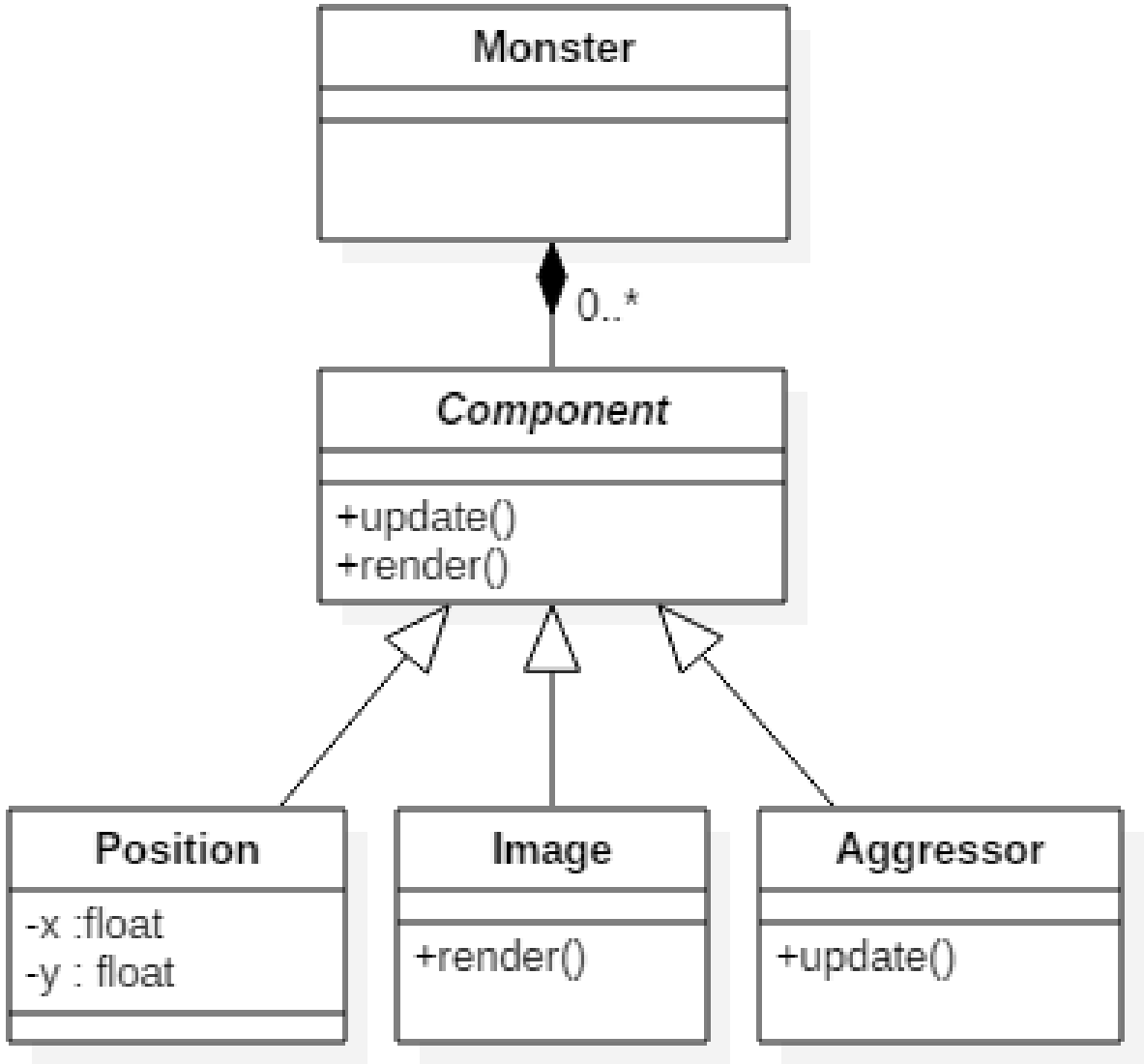
Fixing flexibility

Let's return to the problem of a `Monster` that may sometimes be aggressive, and sometimes not aggressive.

At the moment, the `Monster` is hard-coded to update its `Aggressor` component. We would like to be able to switch this on and off.

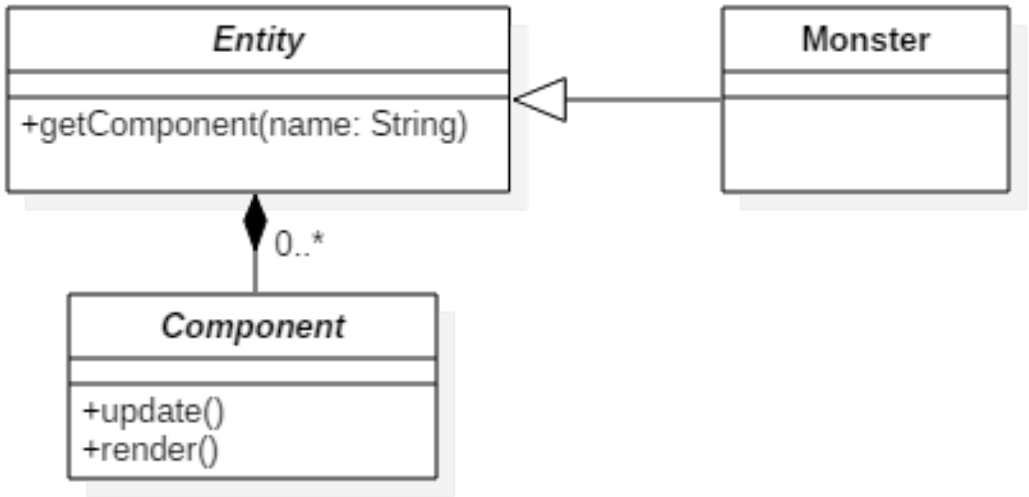
Component as an abstract class

What if `Monster` only has a list of abstract `Components`?



Components that depend on each other

`Image` needed a `Position` to render at. We'd better have a way of looking up components. Also, let's abstract our `Monster` into a base class.



Wait a second. We don't even need a `Monster` class! It's only a bundle of `Components`, after all.

Putting the pieces together

Let's write the classes. We need an `Entity` class, which is just a list of components.

```
class Entity {
    private ArrayList<Component> components;

    public void addComponent(Component component) {
        components.add(component);
    }

    public void update() {
        for (Component c : components) { c.update(); }
    }

    public void render() {
        for (Component c : components) { c.render(); }
    }
}
```

Continued...

Now we need an abstract `Component` that can be updated and rendered, as well as switched on and off.

```
abstract class Component {
    private boolean enabled = true;

    public boolean getEnabled() {
        return enabled;
    }

    public void setEnabled(boolean enabled) {
        this.enabled = enabled;
    }

    // by default, do nothing
    public void update() { }
    public void render() { }
}
```

The road so far

```
Entity monster = new Entity();
monster.addComponent(new Position());
monster.addComponent(new Image());
monster.addComponent(new Aggressor());
```

By doing this, we have separated our complex inheritance diagram into `Entity` objects which are collections of `Component` objects. As a bonus, we can easily reuse `Components` between different `Entitys`!

This is called an **entity-component** approach, and is used by the popular game engine Unity.

Components interacting with others

To implement an `Image` component, we need to look up the `Position` component. We would like to have something like this (pseudocode):

```
class Image extends Component {
    @Override
    public void render() {
        Position position = something.getComponent(Position);
        imageObject.render(position.getX(), position.getY());
    }
}
```

What is this `something`? The most sensible choice is the `Entity` the component is attached to. (Why?)

Implementing lookup

We need to add some code to our Component base class:

```
abstract class Component {
    private Entity entity;
    public Entity getEntity() { return entity; }
    public Component(Entity entity) { this.entity = entity; }
}
```

Applying this in practice looks something like:

```
Entity monster = new Entity();
monster.addComponent(new Position(monster));
monster.addComponent(new Image(monster));
monster.addComponent(new Aggressor(monster));
```

Now, how do we implement getComponent? We need to somehow pass a type as an argument.

Generics save the day!

We can use *generics* to achieve this.

```
class Entity {
    public <T extends Component> T getComponent() {
        for (Component c : components) {
                try {
                    return (T) c;
                } catch (ClassCastException ignored) {
                    // If the cast failed, try the next component.
                }
            }
        }
        return null;
    }
}
```

Implementing Image

This lets us do something like this:

```
class Image extends Component {
    @Override
    public void render() {
        Position position = getEntity().getComponent();
        image.render(position.getX(), position.getY());
    }
}
```

Notice that the return type was *inferred* by the value we assigned it to.

Summary

The **entity-component** approach is an example of a principle called **composition over inheritance**. This principle is a response to the problems caused by large, complex inheritance hierarchies, and tries to simplify things by using composition instead where possible.

Appendix 1: Reflection for great good

(Everything in appendices is **extension only**.)

If you're like me, you're not satisfied with our design so far. You have to call `addComponent` on an `Entity` object, then pass it again as an argument! This is obviously terrible. We can't solve this with generics, because we can't instantiate generic types. Luckily, Java lets us do this another way: *reflection*.

```
public <T> void addComponent(Class<T> type) {
    // example usage: monster.addComponent(Image.class);
    try {
        if (type.isInstance(Component.class)) {
            components.add((T) type.getConstructor(Entity.class)
                                   .newInstance(this));
        }
    } catch (ClassCastException  NoSuchMethodException
             InstantiationException e) {
        throw new RuntimeException("Improper type used.");
    }
}
```

Appendix 2: One step further...

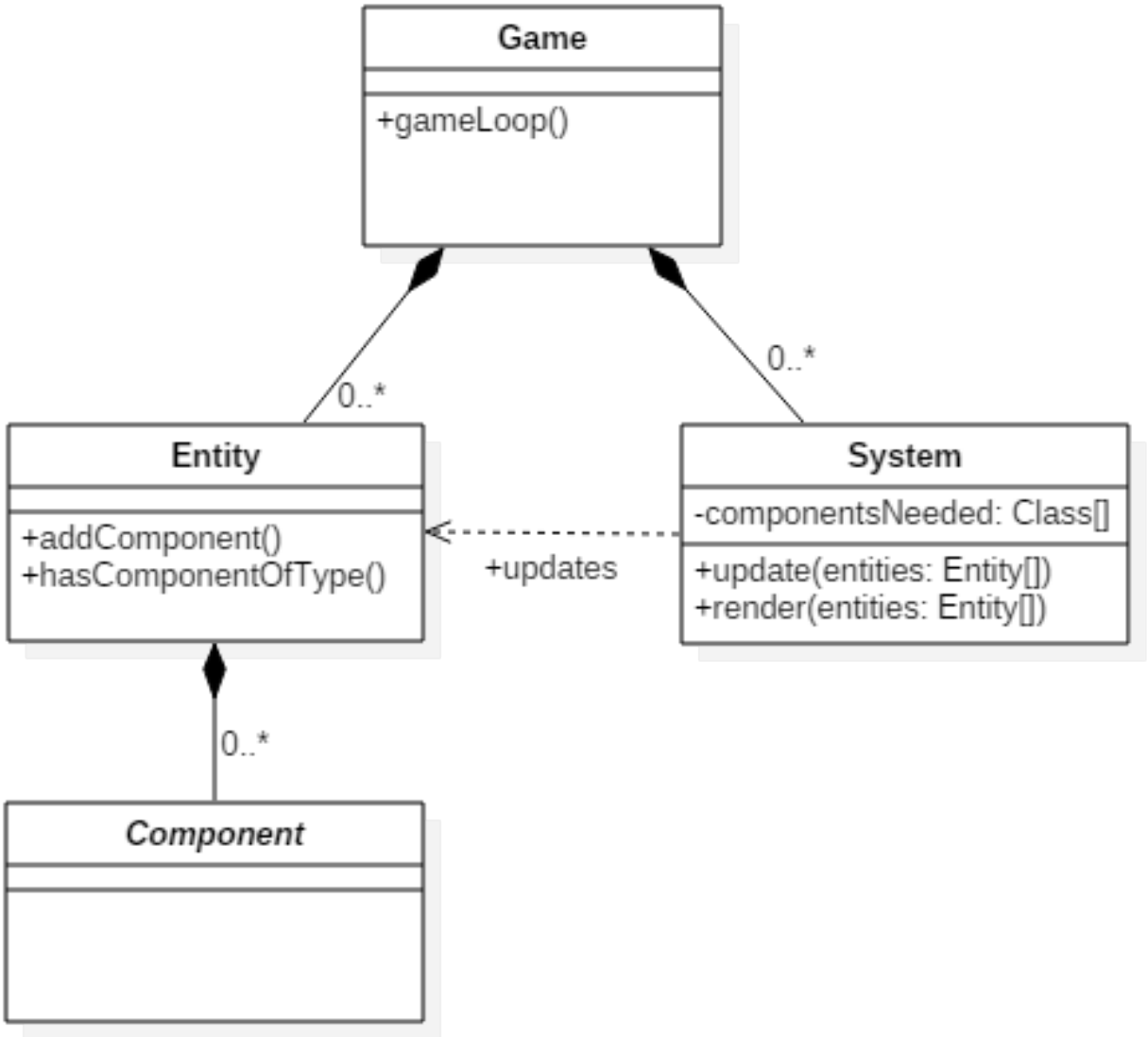
If you're paying attention, you may have noticed that we've started to drift away from object-oriented principles. We're forced to use lots of getters and setters, breaking encapsulation. Why not take it to its logical extreme?

Appendix 2.1: Entity-Component-System

Instead of having our `Component` update itself, we can delegate the actual game logic to a separate class, called a `System`. This means `Components` become just data.

A `System` updates all `Entitys` with the correct component types. This is called the **entity-component-system** approach.

Appendix 2.2: UML diagram



Appendix 2.3: An example

This becomes useful for dealing with larger games, where it's not always easy to track which `Entitys` have which `Components`. The classic example is physics systems.

```
class PositionComponent extends Component {
    public Vector2 pos;
}

class BasicPhysicsComponent extends Component {
    public Vector2 vel;
    public Vector2 force;
    public float   friction;
    public float   mass;
}
class GravityComponent extends Component {
    public static final float G = -9.81f;
}
```

Appendix 2.4

Here is an example `System`. Note the `entities` array contains only those entities with all of the `Components` from `componentsNeeded`.

```
class BasicPhysicsSystem extends System {
    public BasicPhysicsSystem() {
        setComponentsNeeded(new Class[] {
            PositionComponent.class,
            BasicPhysicsComponent.class
        });
    }
}
```

Appendix 2.5

Now we can easily add `PositionComponent` and `BasicPhysicsComponent` to our `Entity` and be confident that the physics simulation will work.

```
public void update(Entity[] entities) {
    for (Entity e : entities) {
        BasicPhysicsComponent phys =
            e.getComponent(BasicPhysicsComponent.class);
        PositionComponent pos =
            e.getClass(PositionComponent.class);
        phys.vel.add(phys.force.multiply(phys.mass));
        pos.add(phys.vel);
        phys.vel.multiply(phys.friction);
    }
}
```

Appendix 2.6

For those `Entitys` we want to respond to gravity, we can simply add a `GravityComponent`.

```
class GravitySystem extends System {
    public GravitySystem() {
        setComponentsNeeded(new Class[] {
            BasicPhysicsComponent.class,
            GravityComponent.class
        });
    }
    public void update(Entity[] entities) {
        for (Entity e : entities) {
            BasicPhysicsComponent phys =
                e.getComponent(BasicPhysicsComponent.class);
            phys.vel.add(new Vector2(0, GravityComponent.G));
        }
    }
}
```

Questions?

Feel free to ask us about embedded programming or game development.

Metrics

You should be able to explain the various keywords and concepts introduced in this lecture, including:

- Sequential, event-driven, and asynchronous programming
- Examples of event-driven systems
- Basic game design, such as the Entity Component approach

If given the appropriate tools/documentation in the exam, you should also be able to:

- Analyse an event-driven system, to determine its behaviour
- Implement event-handling components in an event-driven system