CS1530, Lecture 10: Object-Oriented Design

Bill Laboon

UML (Unified Modeling Language)

- A method of modeling object-oriented software

- Developed in mid-90s, currently on UML 2.5 (as of June 2015). Was not the first OO modeling system but first to break through and still most popular.

- Started off as a way to model object-oriented architectures, now used for much more

- Many different ways to view/model software (as classes, objects, sequences of events, etc) using static symbols

How Often Is It Used?

- Enterprise environments - somewhat often!
- Other places… not so much.
- Why not?
  - Lots of paperwork/documentation which may be of marginal benefit
  - Many promised benefits (e.g. code generation) have been found to be overblown
  - Need to update whenever software design changes (dead documentation)

Why Am I Teaching You UML?

- Pretty sure it is Constitutionally mandated for Software Engineering professors[*citation needed*]

- It is used in industry and you may see it

- Whiteboarding a system is an important skill, even if you don’t remember all the details!

Main Split in Views - Static vs Dynamic

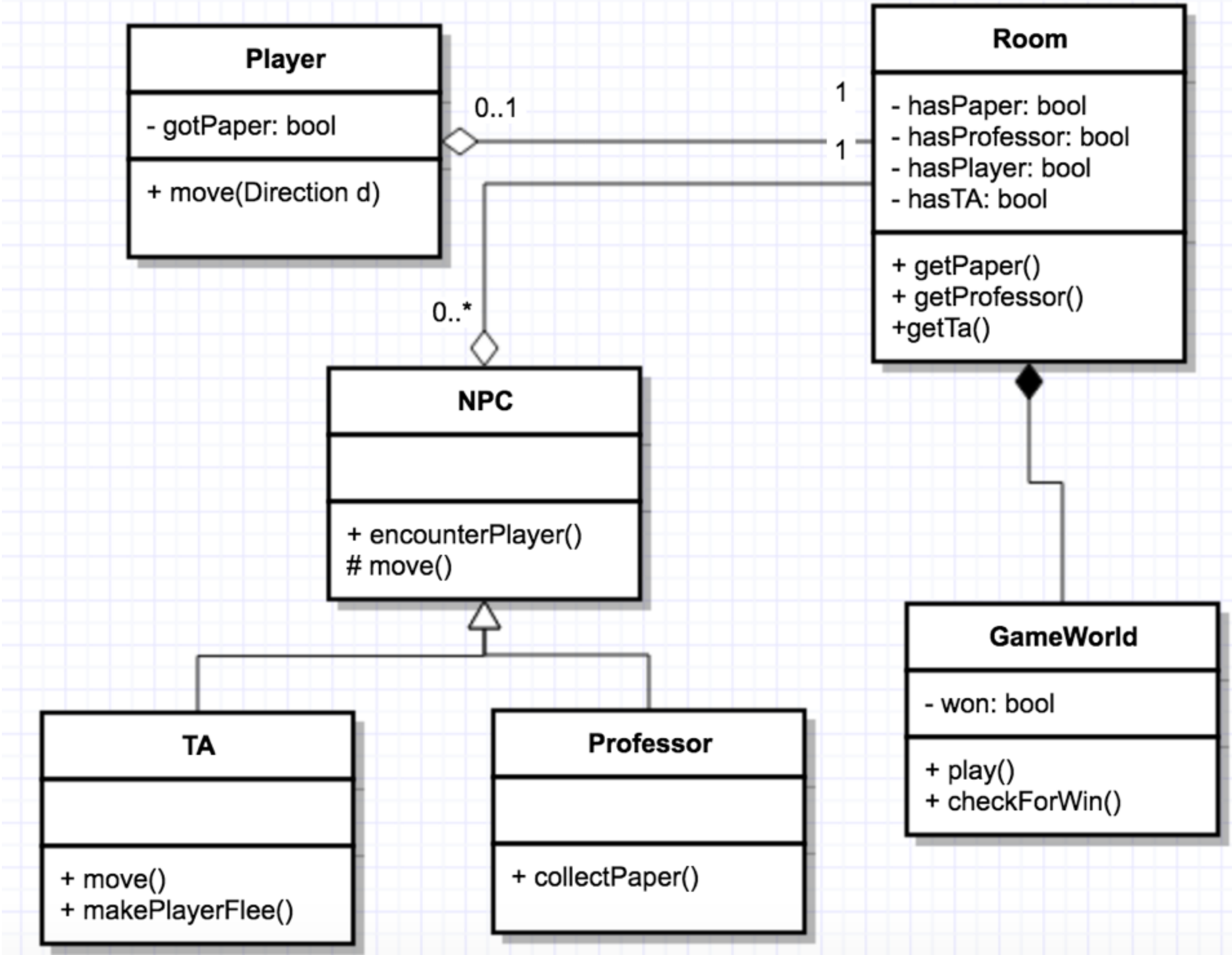
- Static view - how the classes go together, structures, attributes, etc. You can see this by looking at your code in a text editor/IDE and seeing how things fit together.

- Dynamic view - How the system works when it is running - which classes/methods/etc. call which ones WHEN. This can be seen by running the code (or mentally doing so)

- If you can make a distinction based on time, it's a dynamic view

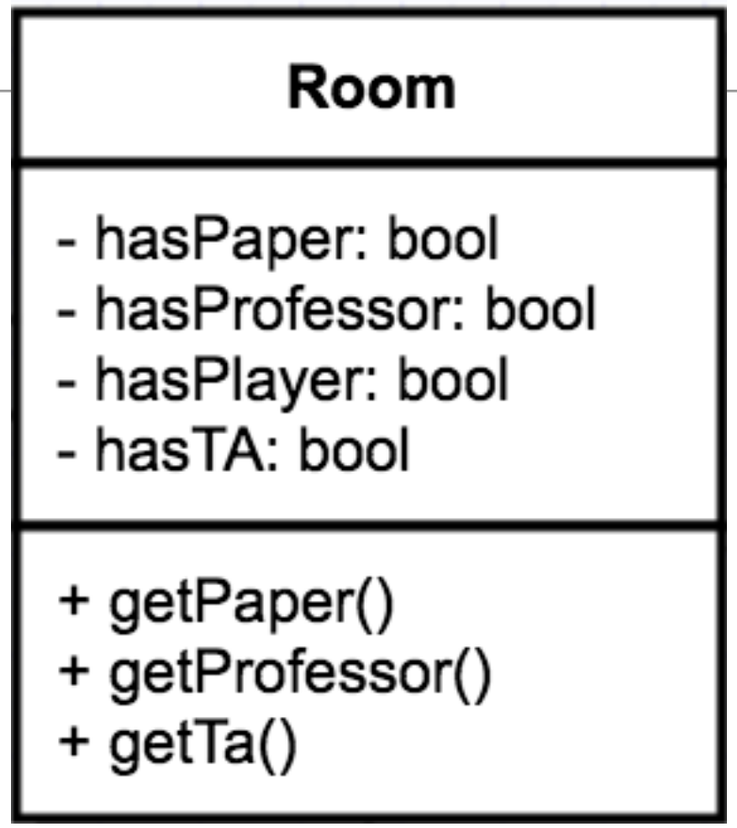
Many Different Kinds of Views, Though

- We will focus on one from each category:
  - Class Diagram (static) - Shows how classes and objects relate to each other
  - Sequence Diagram (dynamic) - Shows how classes, objects, and methods call each other


Player
- gotPaper: bool
+ move(Direction d)
0..1
1
1
Room
- hasPaper: bool
- hasProfessor: bool
- hasPlayer: bool
- hasTA: bool
+ getPaper()
+ getProfessor()
+getTa()
0..*
NPC
+ encounterPlayer()
# move()
TA
+ move()
+ makePlayerFlee()
Professor
+ collectPaper()
GameWorld
- won: bool
+ play()
+ checkForWin()

Classes

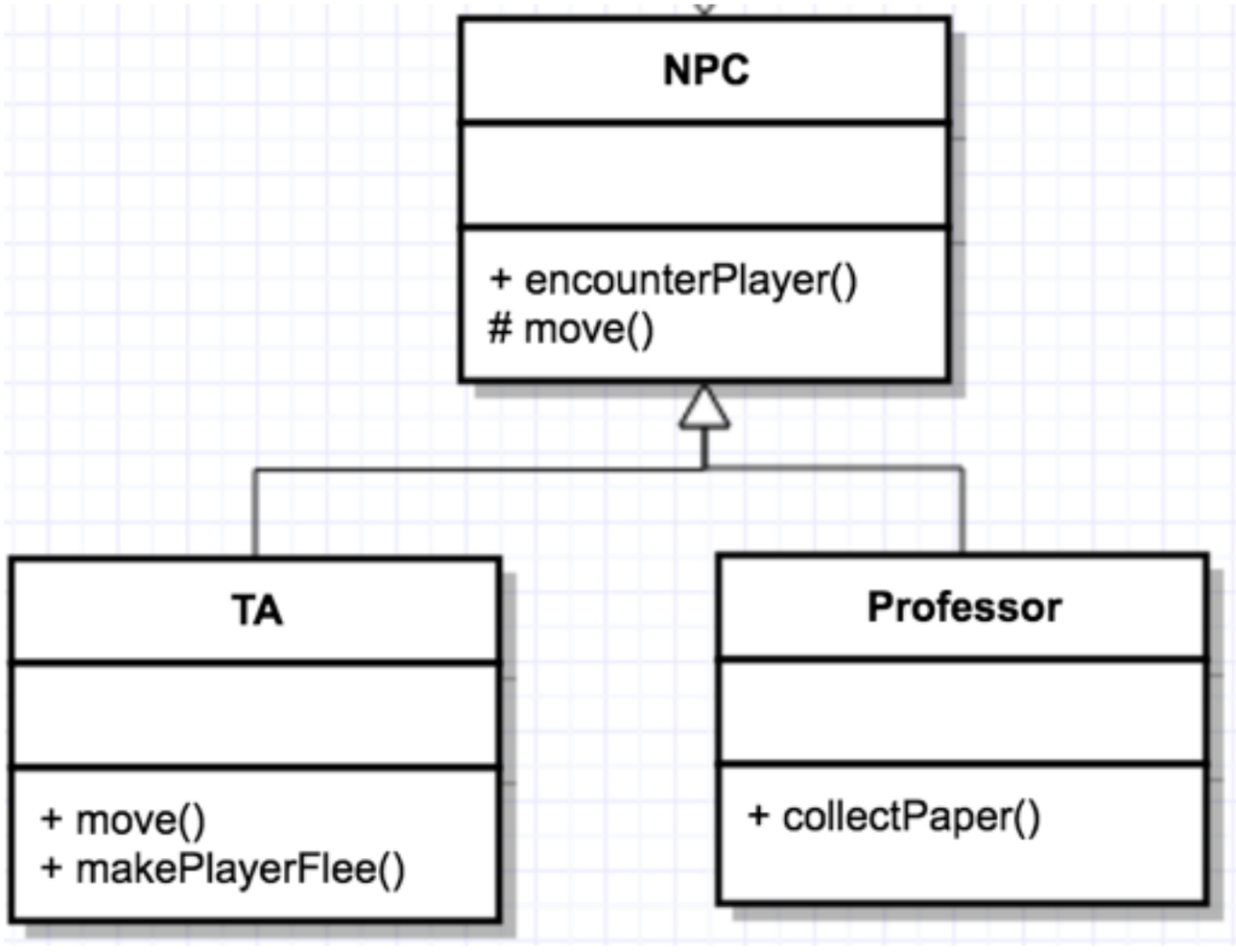
- Classes are boxes with three parts
  - Name
  - Attributes
  - Methods



- For attributes and methods, can use the following access modifiers:
  - + for public
  - - for private
  - # for protected

Generalization Relationships - Superclasses and Subclasses

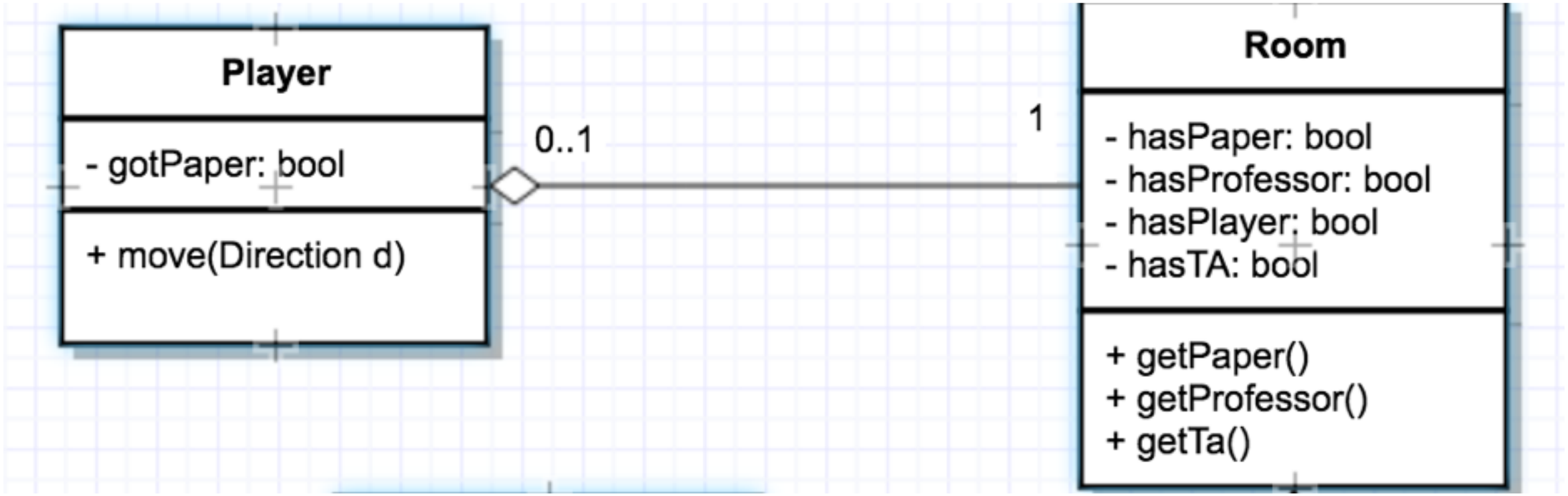
- Generalization
  - empty arrow, usually superclass on top, subclass on bottom
  - "is-a" relationship
  - e.g. A Duck "is-a" Bird, a Cockatiel "is-a" Bird



Aggregation Relationship

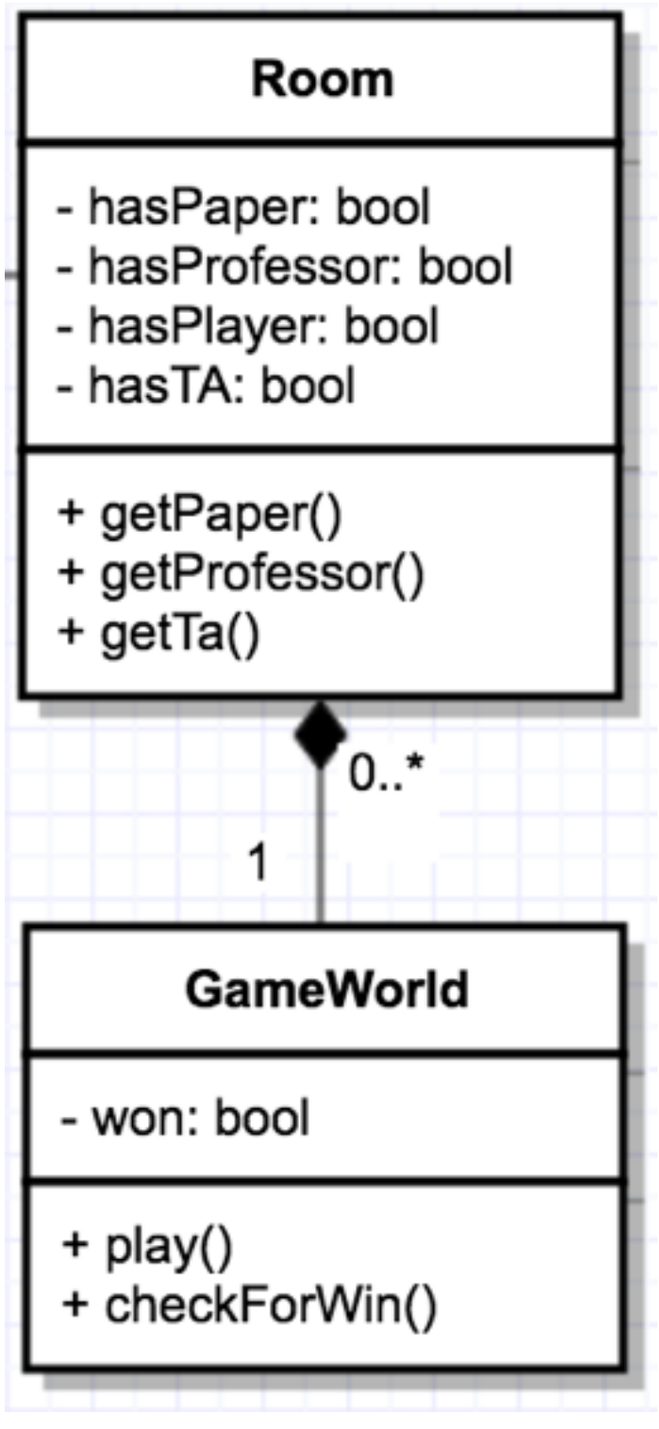
- Empty diamond

- weak "has-a" relationship (a pond has ducks in it, but they are not part of it, can exist without)



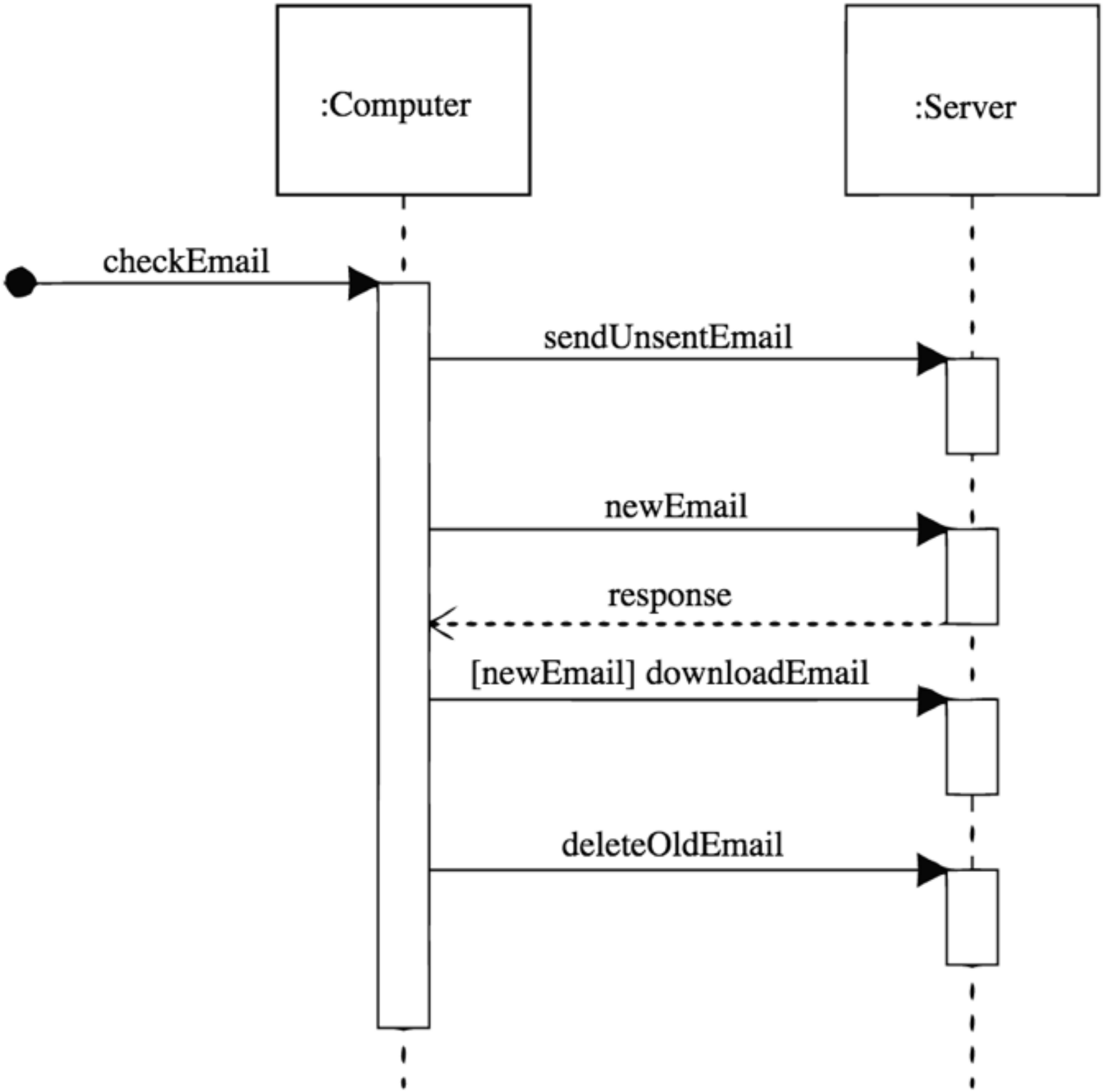
Composition Relationship

- Filled-in diamond
- strong "has-a" relationship
  (a Book is composed of
  Words, a House is composed
  of Rooms - cannot exist
  without them)



Sequence Diagrams

- Classes are shown as vertical lines
- Calls to methods are shown as solid arrows, and then return values as dashed arrows
- When execution of a method is in the stack it's a solid box
- Method called and value returned written above calls to methods and returned values, respectively



The SOLID Principles

- A mnemonic for “five key principles” of good object-oriented design
  - Single Responsibility Principle
  - Open/Closed Principle
  - Liskov Substitution Principle
  - Interface Segregation Principle
  - Dependency Inversion Principle

Single Responsibility Principle

A class should have a single responsibility.

That responsibility should be entirely encapsulated by the class.

Bad “S”

```
// What is Stuff’s single responsibility?

public class Stuff {
    public void printMemo() { ... }
    public int numCats(String breed) { ... }
    public String getName() { ... }
    public void haltSystem(int exitCode) { ... }
}
```

Better “S”

```
public class Cat {
    public String getName() { ... }
    public String getBreed() { ... }
    public Currency getRentalCost() {...}
    public int rent() { ... }
}
```

```
public class RentACatSystem {
    public void startSystem() { ... }
    public void haltSystem(int exitCode) { ... }
    public void forceShutdown() { ... }

}
```

Single Responsibility Principle

Describe the class. If you can't do it without using "and", you are probably violating the Single Responsibility principle.

Other code smells:

1. Many methods
2. Many attributes
3. Difficult to comprehend what class does
4. Methods don't seem related

Open / Closed Principle

Classes should be open for extension, but closed to modification.

In other words, add features by subclassing, not adding code.

Once complete, code modification in a given class should generally not occur except to fix defects.

Open / Closed Principle

```
public class Printer {
    private void formatDocument() { ...
}
    public void printDocument() { ... }
}
```

Now let’s say we want to add a way to print PDFs. One way would be to add a method:

```
    public void printToPDF()   { ... }
```

But this is a violation of the “O”!

Better

```
abstract class Printer {
   private void formatDocument() { ... }
}

public class PhysicalPrinter extends Printer {
   public void printDocument() { ... }
}

public class PdfPrinter extends Printer {
   public void printDocument() { ... }
}
```

The Open/Closed Principle

If your classes keep getting bigger with each commit, you may be violating the Open/Closed Principle.

This helps us because once a class is done, it's done. Modifying classes incessantly is a recipe for regression errors.

Liskov Substitution Principle

A class B which is a subclass of class A, should implement any method in A while meeting all invariants.

Liskov Substitution Principle

```
// What's wrong with this?
public class Circle {
    public Location loc;
    public Color color;
    public double radius;
}

public class Square extends Circle {
    public double length;
    public double height;
}
```

Better

```
public class Shape {
    Location loc;
    Color color;
}

public class Rectangle extends Shape {
    public double length;
    public double height;
}

public class Circle extends Shape {
    public double radius;
}
```

Interface Segregation Principle

Clients should not depend on methods that they do not use.

In practice, this means lots of small interfaces, not one big one.

Interface Segregation Principle

```
public interface BankInterface {
    public void transferMoneyIntraBank();
    public void transferMoneyInterBank();
    public void allocateMortgage();
    public void transferMortgage();
    public void setupHeloc();
    public void withdrawCash();
    public void depositCheck();
    public void depositCash();
    public void authenticate();
    public Bank[] getBankBranches();
    public Employee[] getBankEmployees();
}
```

Better

```
public interface AtmInterface {
    public void withdrawCash();
    public void depositCash();
    public void depositCheck();
    public void authenticate();
}
```

Interface Segregation Principle

If you find yourself not using all of the methods of an interface, consider splitting up the interfaces for different roles.

Otherwise, there is more room for error and the code becomes more difficult to understand.

Be a code anti-natalist!

Dependency Inversion Principle

A. High-level modules should not depend on low-level modules. Both should depend on abstractions.

B. Abstractions should not depend on details. Details should depend on abstractions.

Dependency Inversion Principle

```
public class Aviary {
    public void buyCockatiel();
    public void buyGreyParrot();
    public void buyYellowBelliedSapSucker();
}
```

Better

```
public class Aviary {
    public void buyBird(Bird b);
}

public abstract class Bird {
…
}
public class Cockatiel extends Bird {
…
}
```

Dependency Inversion Principle

Attributes need to be interfaces (or an abstract class)

All class packages connect through interfaces

Concrete classes are final classes; all subclasses should derive from abstract classes

As a side effect of that, any concrete method should not be overridden, only abstract methods

The Downside

OVERENGINEERING

See “Enterprise FizzBuzz” for this principle run amok