CIT 5940

Software Design

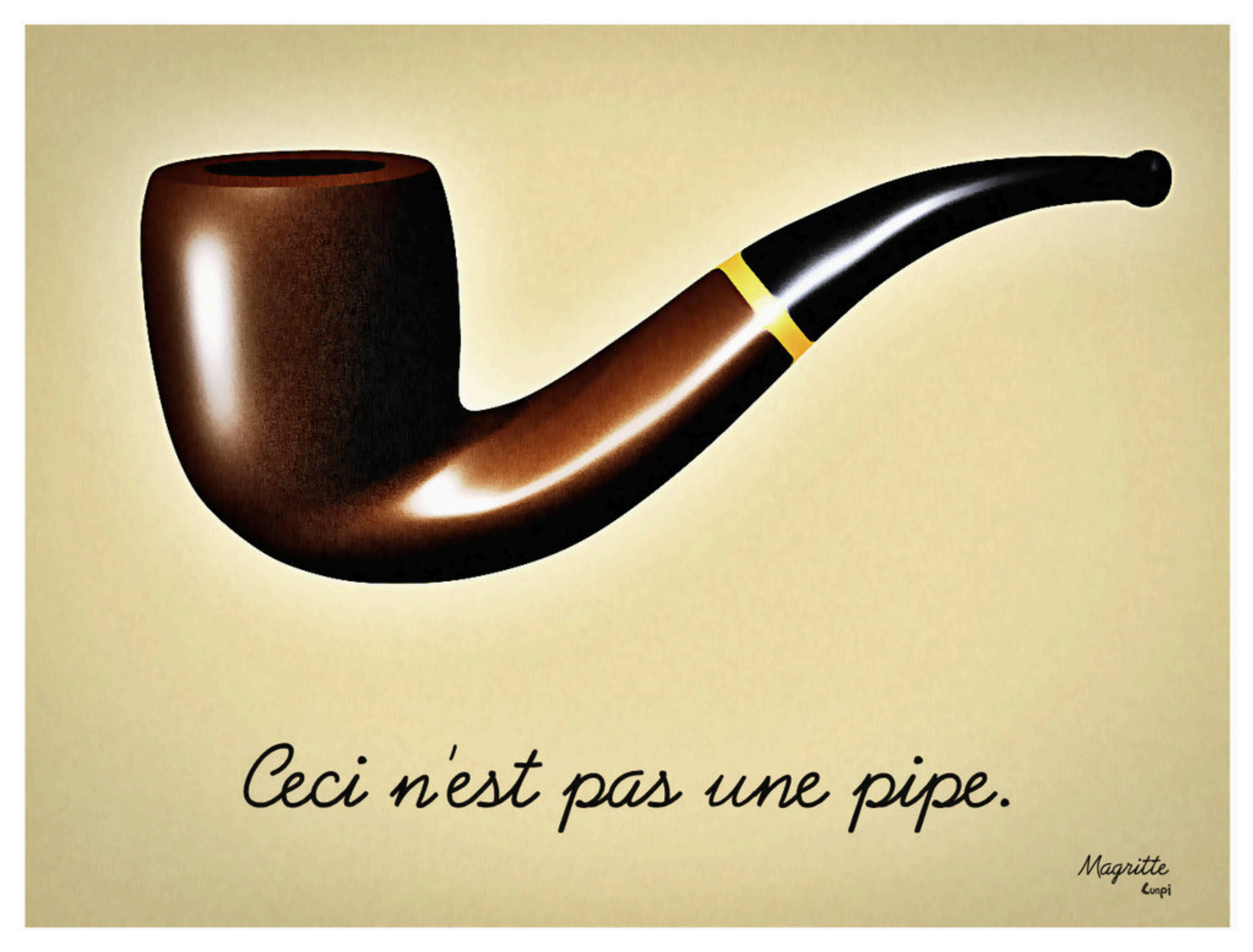
Software Design

- An iterative process
- Provides details about components necessary to implement software
  - **Classes**
  - **Data Structures**
  - **Software architecture**, etc.

Class Design: Abstraction

An Abstraction of a Pipe.

(from Torczyner, Harry. Magritte: Ideas and Images. p. 71)



Class Design: Abstraction

Abstraction: set of information properties relevant to a stakeholder about an entity

- **Information Property**: a named, objective and quantifiable aspect of an entity
- **Stakeholder**: a real or imagined person (or a class of people) who is seen as the audience for, or user of the abstraction being defined

When you drive a car, you're shown information on speed, fuel, and RPMs. You have the choice of actions like *drive*, *park*, *reverse*, etc.

As the driver of a car, you are **not** concerned with the crankshaft & spark plugs & cylinders & valves & …

(For self-driving cars, `Drive.java` might have a similar degree of abstraction.)

Class Design

Information Hiding: prevents client(s) from accessing some aspect of the class (or software system) implementation

Information hiding can be achieved through:

- **Interfaces**
- **Encapsulation**

Using an Interface to Hide Information

```java
public interface Driveable {
    void drive();
    void park();
    void reverse();
}
public class Car implements Driveable {
    // ...
}
public class Motorcycle implements Driveable {
    // ...
}
```

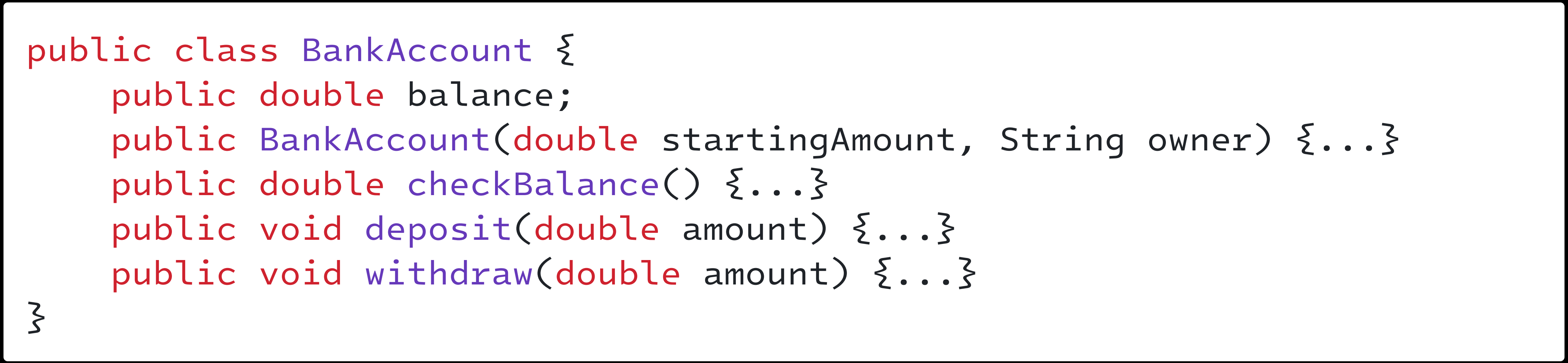
From a user's point of view, they do not need to care about how `Car` and `Motorcycle` are implemented.

Using an Interface to Hide Information

```java
public class DriveableClient {
    public static void main() {
        Driveable car = new Car();
        Driveable motorcycle = new Motorcycle();
        car.drive();
        motorcycle.drive();
    }
}
```

From a user's point of view, they do not need to care about how `Car` and `Motorcycle` are implemented. Just use them both as `Driveable` objects.

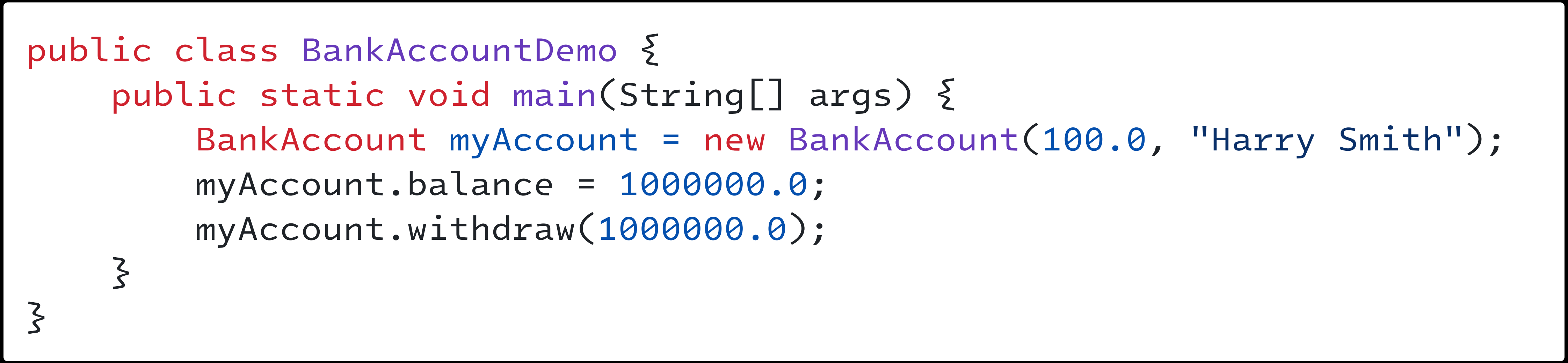
Using Encapsulation to Hide Information



```java
public class BankAccount {
    public double balance;
    public BankAccount(double startingAmount, String owner) {...}
    public double checkBalance() {...}
    public void deposit(double amount) {...}
    public void withdraw(double amount) {...}
}
```

Do you see any issues with the way this class is designed?

Using Encapsulation to Hide Information



```
public class BankAccountDemo {
    public static void main(String[] args) {
        BankAccount myAccount = new BankAccount(100.0, "Harry Smith");
        myAccount.balance = 1000000.0;
        myAccount.withdraw(1000000.0);
    }
}
```

Public methods can be freely accessed and modified by other classes. This is not good!

Using Encapsulation

```
public class BankAccount {
    private double balance;
    public BankAccount(double startingAmount, String owner) {...}
    public void deposit(double amount) {
        if (verifyDepositAmount(amount)) {
            balance += amount;
        }
    }
    public void withdraw(double amount) {
        if (balance >= amount) {
            balance -= amount;
            offerCash(amount);
        }
    }
}
```

Class Design

Characteristics of a well-formed design class:

- **Complete and sufficient:**
    - design should encapsulate all attributes and methods that are expected
- **Primitiveness**:
    - methods in a class should accomplish one service for the class.
    - A class should not have more than one method to accomplish the same function

*Can you make an argument about why primitiveness is important for **testing?** For making modifications in future iterations?*

Class Design

Characteristics of a well-formed design class:

- **High cohesion**:
  - A cohesive design class has a small, focused set of responsibilities
  - A cohesive design class single-mindedly applies attributes and methods to implement those responsibilities

Class Design

Characteristics of a well-formed design class:

- **Low coupling**:
  - Classes collaborate with each other
  - Collaboration should be kept to a minimum and mediated through interfaces wherever possible
  - High coupling leads to software that is difficult to implement, to test, and to maintain over time

Unified Modeling Language (UML)

UML:

- Modeling language intended to provide a standard way to visualize the design of a software system.

Class diagram:

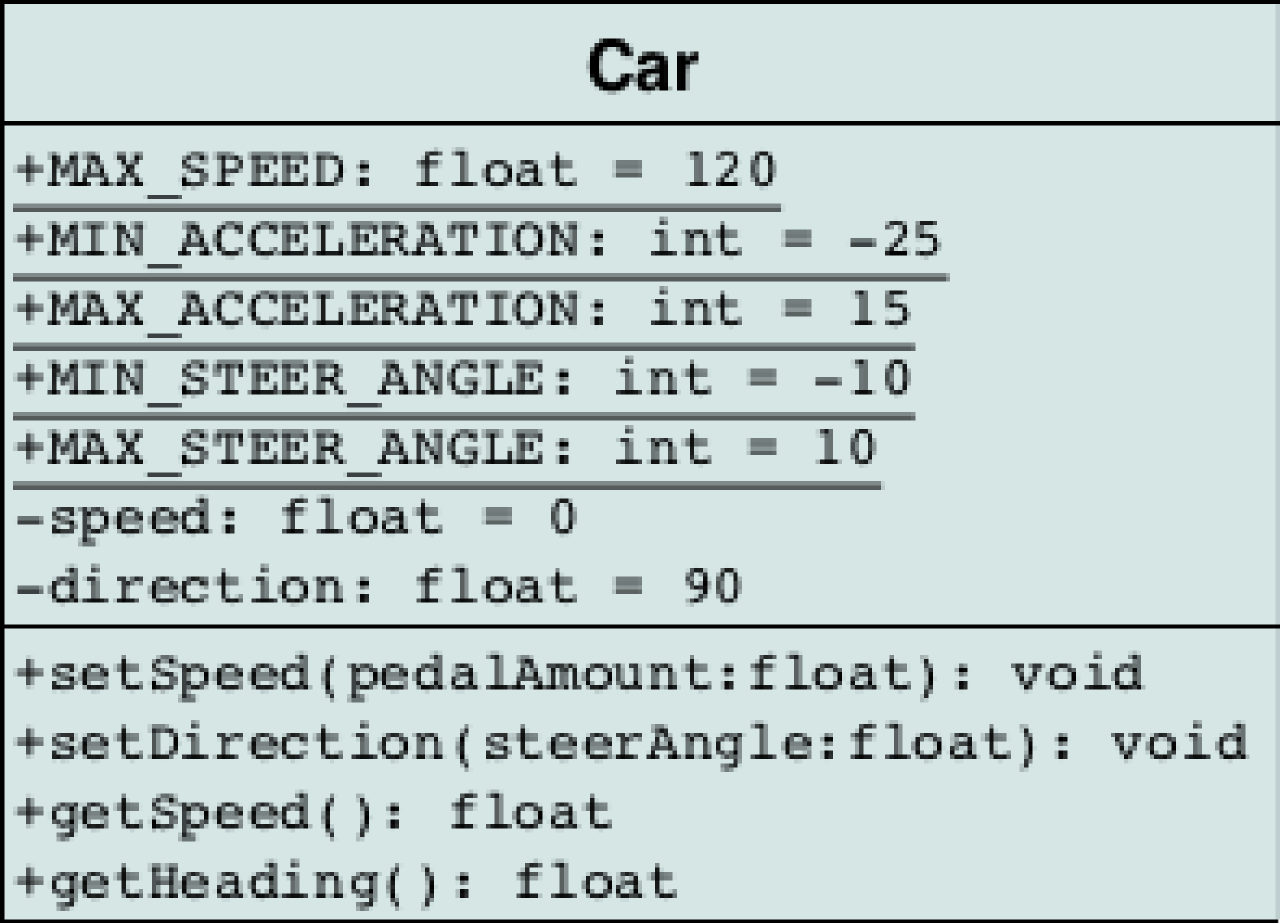
- Static diagram
- Describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects

Class Diagram

Upper section: Contains the name of the class

Middle section: Contains the attributes of the class

Bottom section: Includes class operations (methods header). Displayed in list format, each operation takes up its own line



Car

```
+MAX_SPEED: float = 120
+MIN_ACCELERATION: int = -25
+MAX_ACCELERATION: int = 15
+MIN_STEER_ANGLE: int = -10
+MAX_STEER_ANGLE: int = 10
-speed: float = 0
-direction: float = 90
```

```
+setSpeed(pedalAmount:float): void
+setDirection(steerAngle:float): void
+getSpeed(): float
+getHeading(): float
```

Domain Model Diagram

Emphasizes *classes, interfaces, associations, usage, realization, & multiplicity*

Used to show how all the entities relate

- Implementation details totally abstracted
- This example doesn’t show a single method!

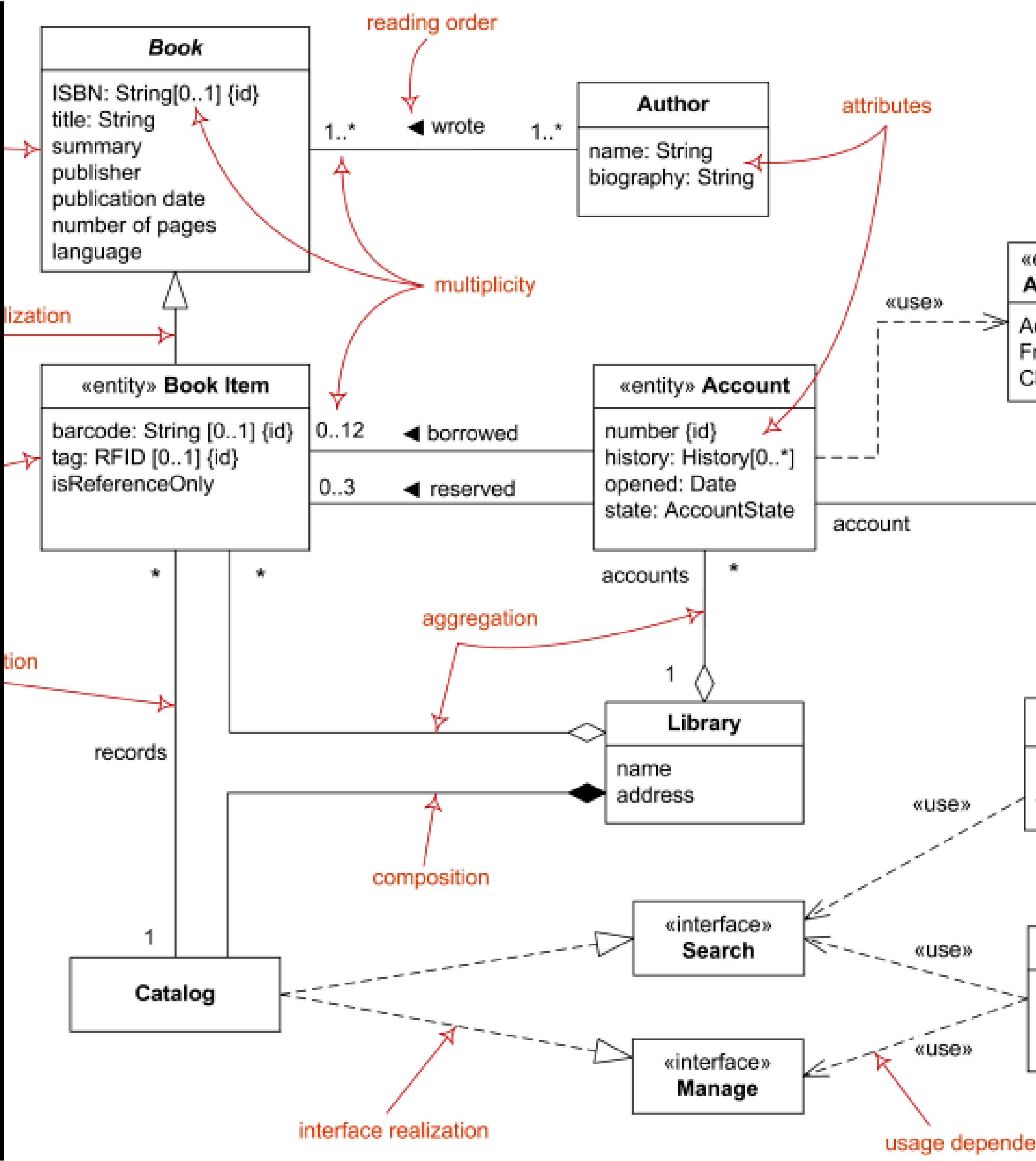
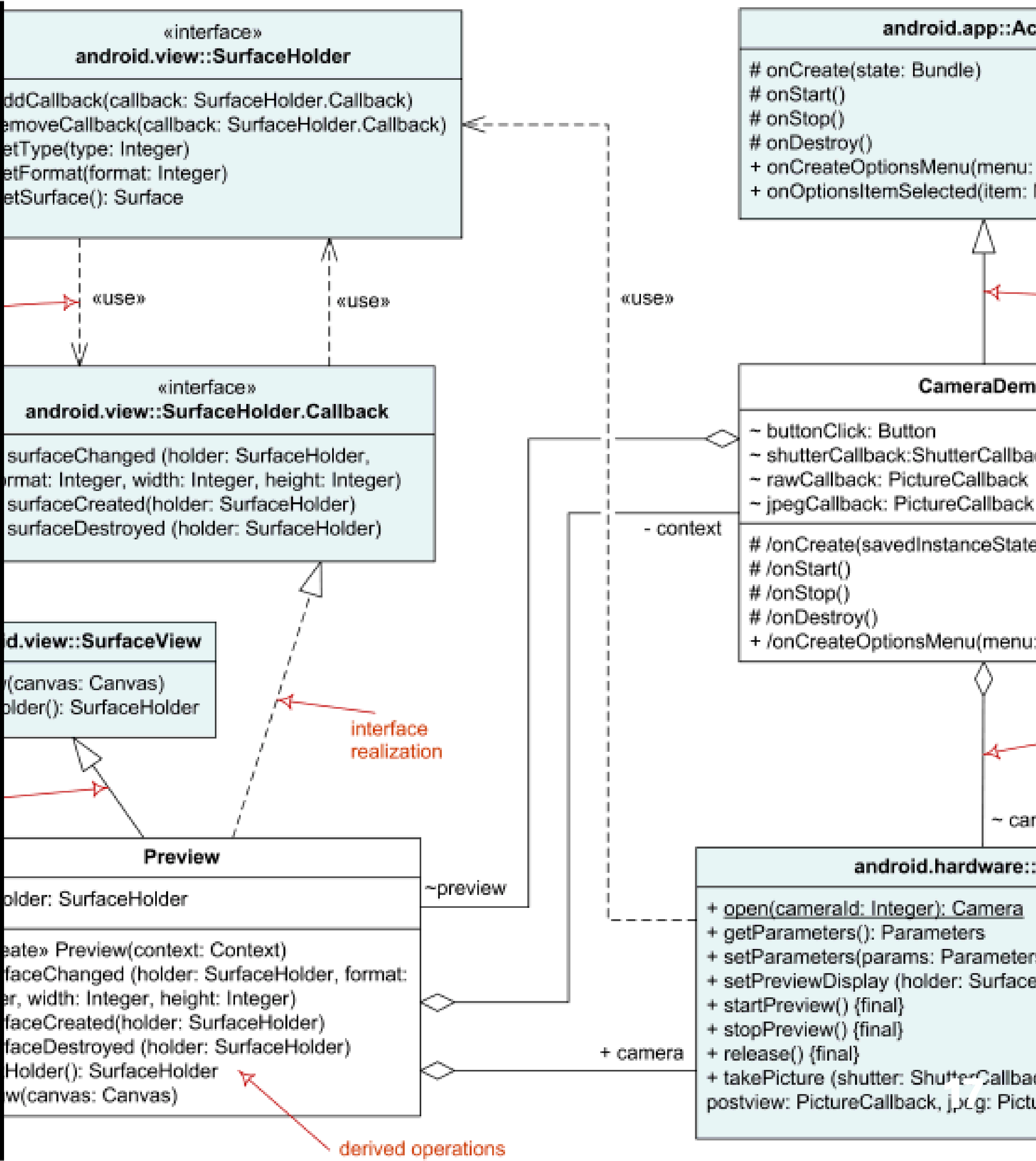


Diagram of Implementation Classes

Emphasizes *classes, interfaces, associations, usage, realization*

Gives a clear picture of how the classes will be written

- Will include fields & methods
- Very dense!


«interface»
android.view::SurfaceHolder
ddCallback(callback: SurfaceHolder.Callback)
emoveCallback(callback: SurfaceHolder.Callback)
etType(type: Integer)
etFormat(format: Integer)
etSurface(): Surface
«use»
«use»
«use»
android.app::Ac
# onCreate(state: Bundle)
# onStart()
# onStop()
# onDestroy()
+ onCreateOptionsMenu(menu:
+ onOptionsItemSelected(item:
«interface»
android.view::SurfaceHolder.Callback
surfaceChanged (holder: SurfaceHolder,
rmat: Integer, width: Integer, height: Integer)
surfaceCreated(holder: SurfaceHolder)
surfaceDestroyed (holder: SurfaceHolder)
CameraDem
~ buttonClick: Button
~ shutterCallback:ShutterCallba
~ rawCallback: PictureCallback
~ jpegCallback: PictureCallback
- context
# /onCreate(savedInstanceState
# /onStart()
# /onStop()
# /onDestroy()
+ /onCreateOptionsMenu(menu:
d.view::SurfaceView
(canvas: Canvas)
older(): SurfaceHolder
interface realization
~ car
Preview
~preview
older: SurfaceHolder
eate» Preview(context: Context)
faceChanged (holder: SurfaceHolder, format:
er, width: Integer, height: Integer)
faceCreated(holder: SurfaceHolder)
faceDestroyed (holder: SurfaceHolder)
Holder(): SurfaceHolder
w(canvas: Canvas)
derived operations
+ camera
android.hardware::
+ open(cameraId: Integer): Camera
+ getParameters(): Parameters
+ setParameters(params: Parameter
+ setPreviewDisplay (holder: Surface
+ startPreview() {final}
+ stopPreview() {final}
+ release() {final}
+ takePicture (shutter: ShutterCallbac
postview: PictureCallback, jpeg: Pictu

What to use?

Companies have different standards

Important to know the ideas of UML but frequency of use may be low

So:

- For this course, use the **domain model diagram** since I know what methods you’re using!
- If you want to do the **diagram of implementation classes**, that is good practice for the future!

Class Diagram

Data fields visibility:

- `+` Public
- `-` Private
- `#` Protected
- `/` Derived
- `~` Package (default)

Class Diagram

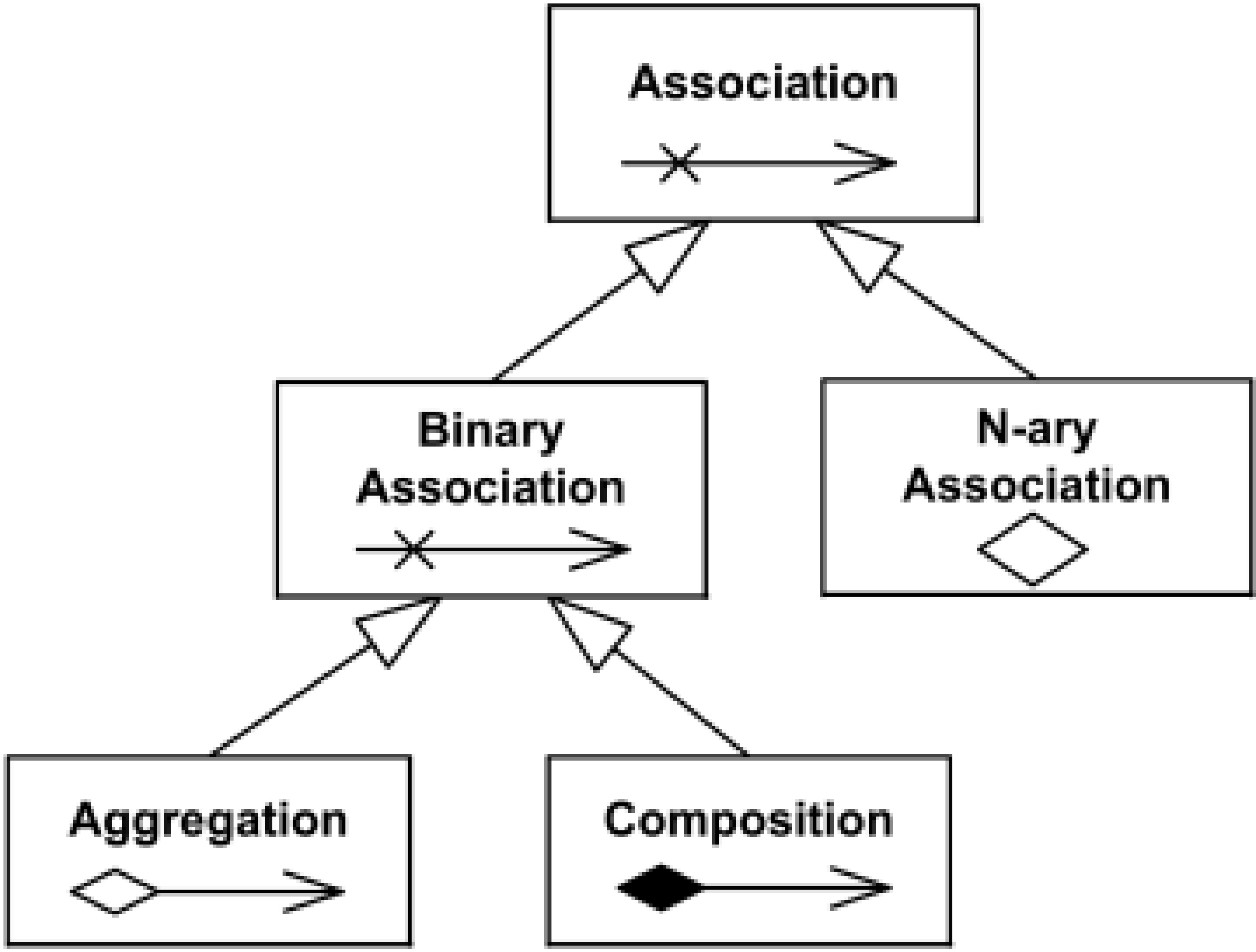
Methods:

- Underline static methods
- Parameter types listed as (name: type)
- Do not include “return type” when it is void

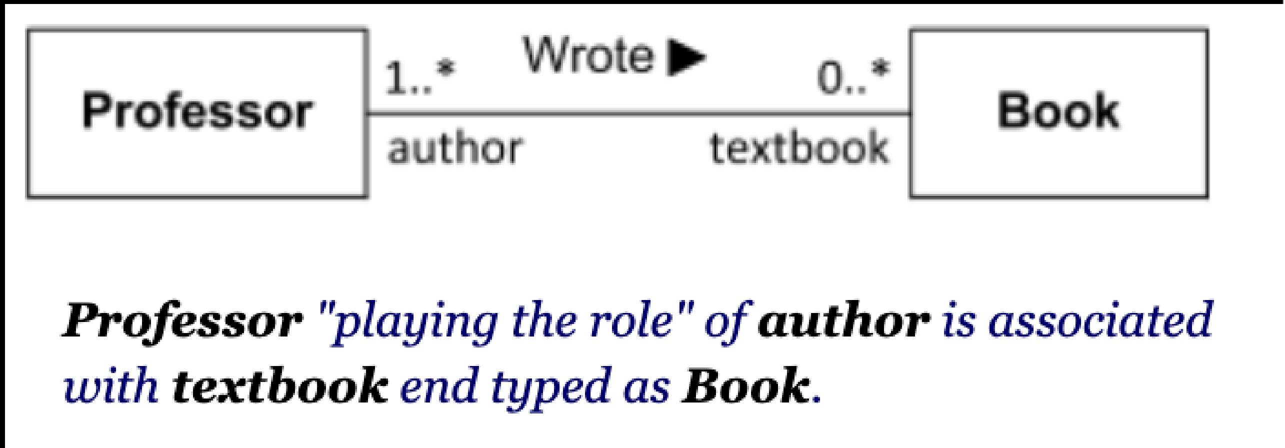
Class Relationships

All relationships in UML are considered *associations*

- Specific kinds of relationships are subtypes of *associations* and have specific ways they should be drawn on the page.



Writing a General Association

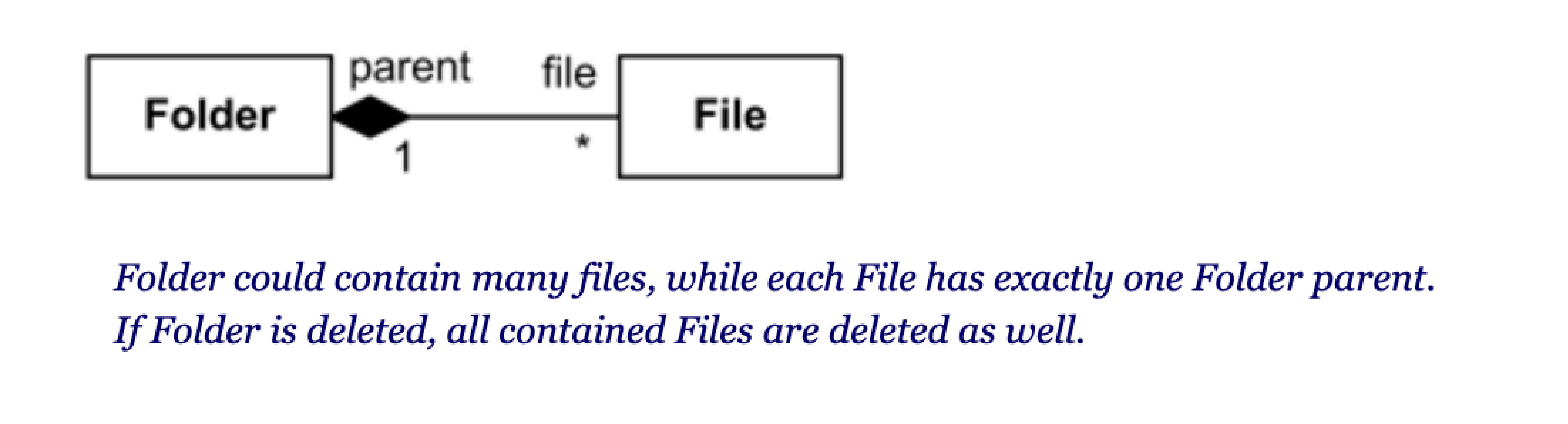


Professor *"playing the role" of* ***author*** *is associated with* ***textbook*** *end typed as* ***Book****.*

Class Relationships

Composition relationship (filled/black diamond):

- When attempting to represent real-world whole-part relationships.
- When the container is destroyed, the contents are also destroyed.
- Usually refers to a **collection** (or data structure!) of some kind

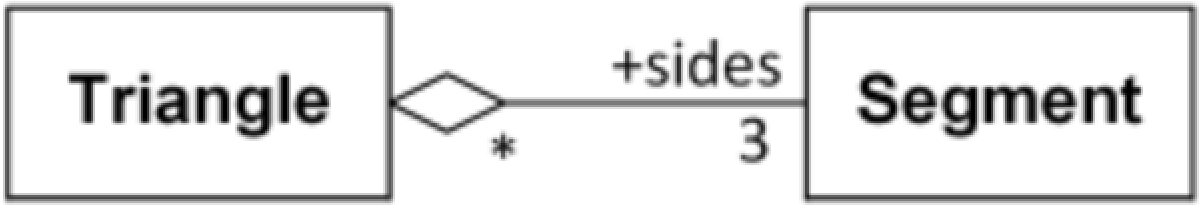


Folder could contain many files, while each File has exactly one Folder parent.
If Folder is deleted, all contained Files are deleted as well.

Class Relationships

Aggregation relationship (white diamond):

- Weak form of aggregation.
- When the container is destroyed, the contents are usually not destroyed.
- Usually refers to a **collection** (or data structure!) of some kind

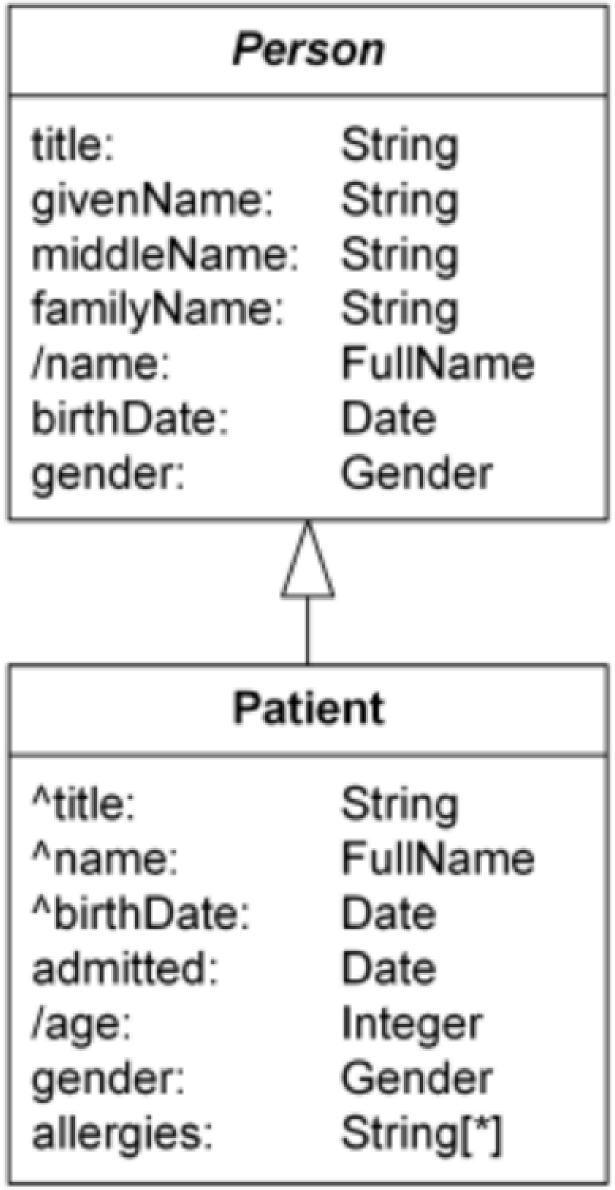


Triangle has 'sides' collection of three line Segments.
Each line Segment could be part of none, one, or several triangles.

Class Relationships

Inheritance (hollow triangle, solid line):

- (sometimes called generalization)
- Omit trivial (get/set) methods
- Do not include inherited methods



Person

```
title:         String
givenName:     String
middleName:    String
familyName:    String
/name:         FullName
birthDate:     Date
gender:        Gender
```

Patient

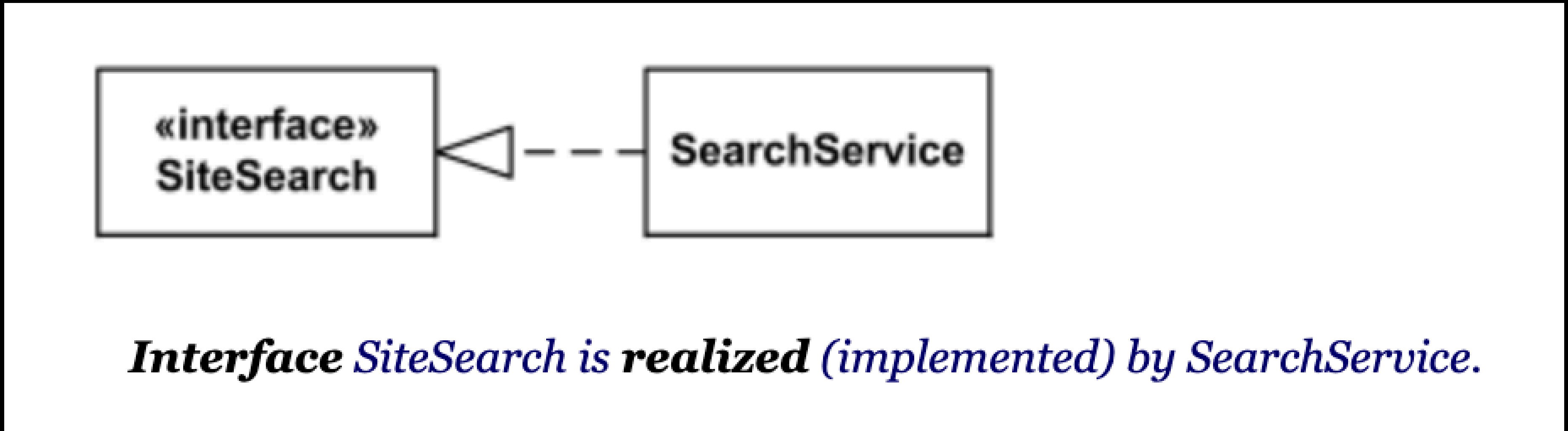
```
^title:        String
^name:         FullName
^birthDate:    Date
admitted:      Date
/age:          Integer
gender:        Gender
allergies:     String[*]
```

Patient class with inherited attributes title, name, and birthDate.

Class Relationships

Implementation (hollow triangle, dotted line):

- (sometimes called realization)
- Write `<interface>` on top of the interfaces' name



Interface SiteSearch is realized (implemented) by SearchService.

Questions

CIT 5940

Design Patterns: Flyweight

Design patterns

- Embody and generalize important design concepts for a recurring problem
- Reusable solution to a commonly occurring problem in software design

Design patterns

23 patterns grouped in 3 categories:

- Creational patterns: object creation patterns
- Structural patterns: classes and objects organization patterns
- Behavioral patterns: communication between objects patterns

Flyweight Pattern

Structural pattern

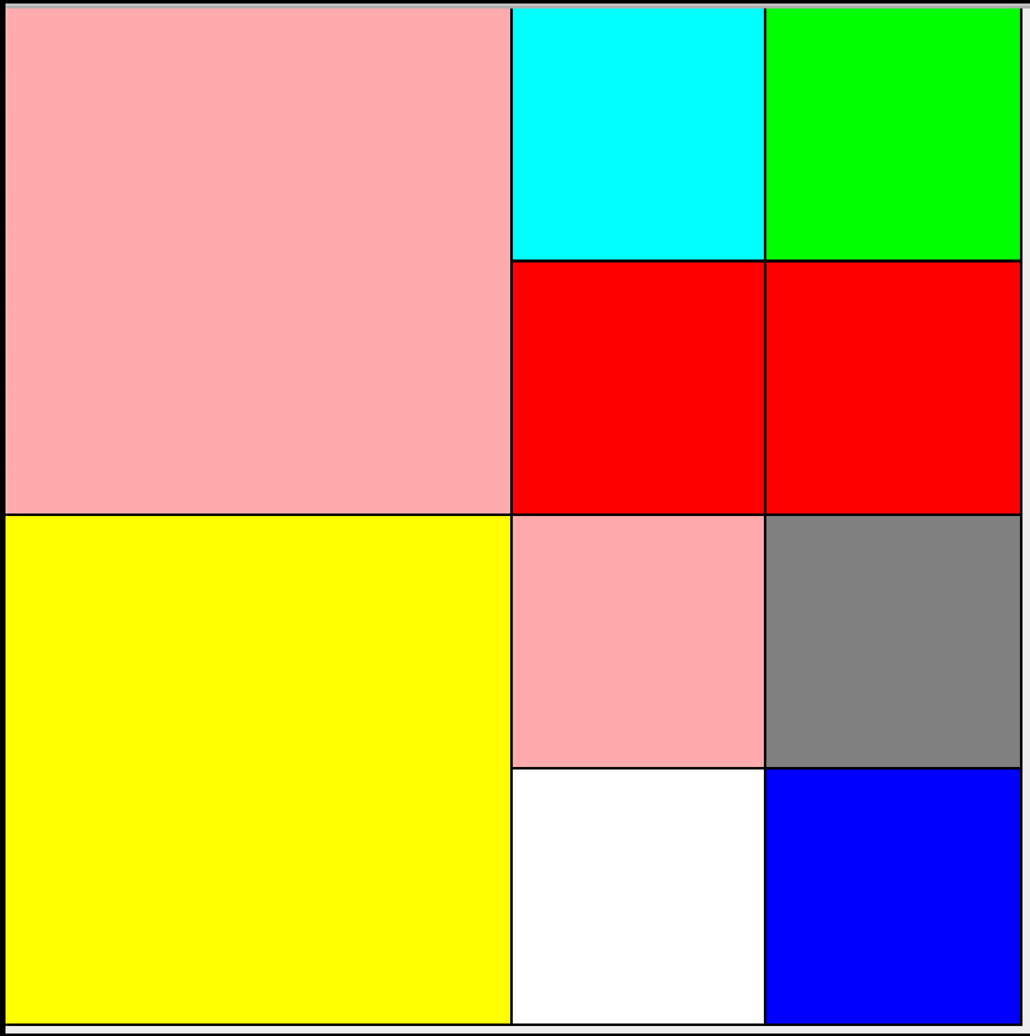
- **Problem:** We are building an application with many similar objects. Objects store identical information and play the same role.
- **Goal:** Minimize memory cost

Example: Memory & Block Games

Each `Block` stores references to:

- two `Point` objects: `topLeft` and `bottomRight` (64 bits each)
- a `Color` (32 bits)
- a description `String` (unbounded size!)
- four children `IBlock` objects (64 bits each)

Which of these values are wasteful to duplicate?

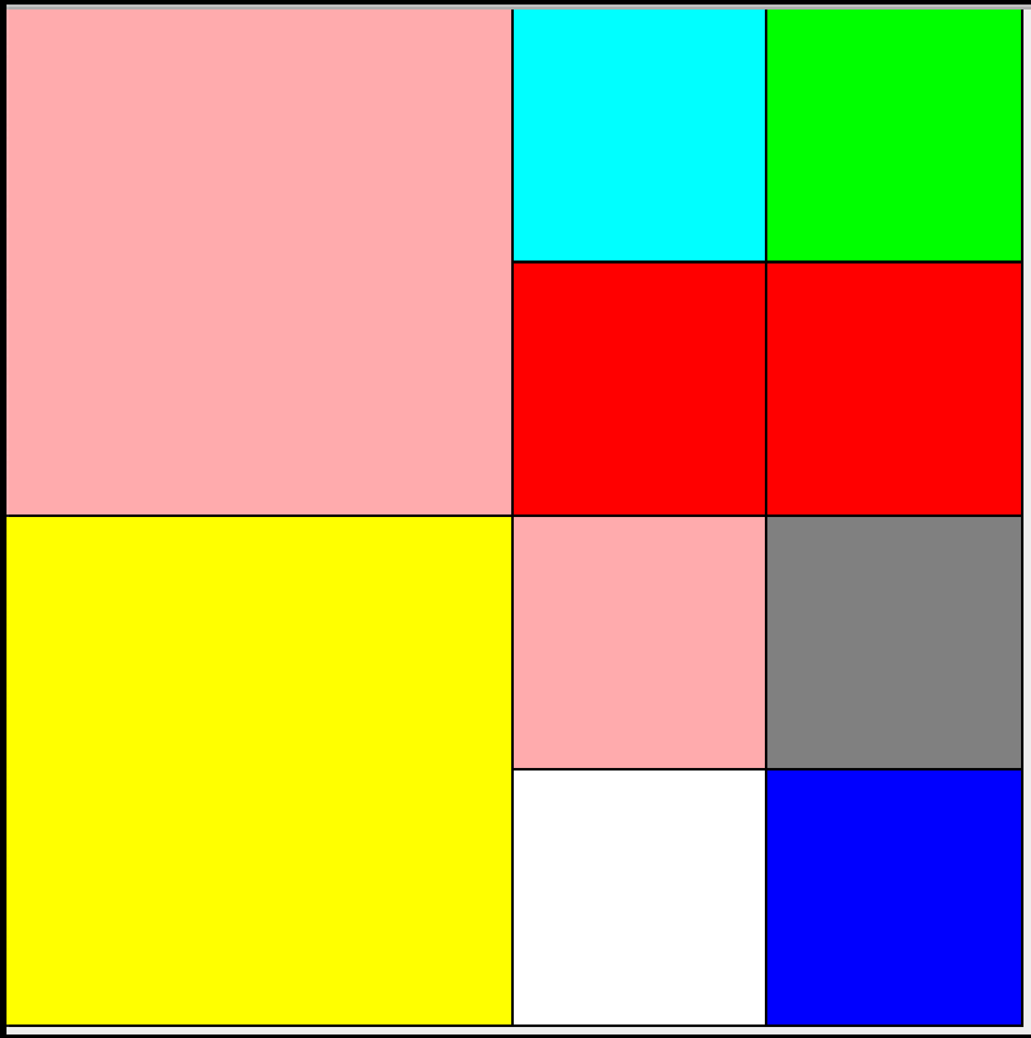


Example: Memory & Block Games

Each `Block` stores references to:

- two `Point` objects: `topLeft` and `bottomRight` (64 bits each)
- a `Color` (32 bits)
- a description `String` (unbounded size!)
- four children `IBlock` objects (64 bits each)

Each of the blocks comes from a standard set of colors.

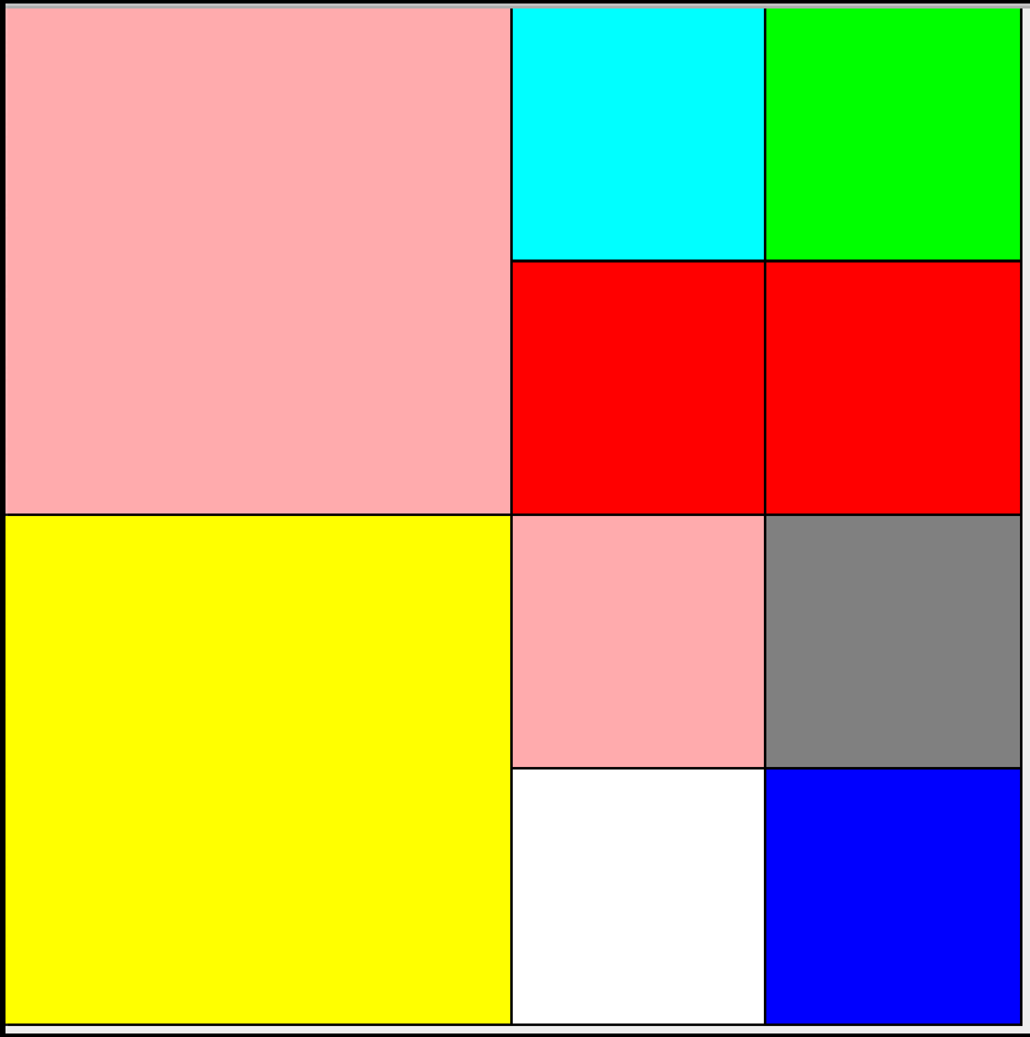


Example: Memory & Block Games

Each of the colors comes from a standard set of colors.

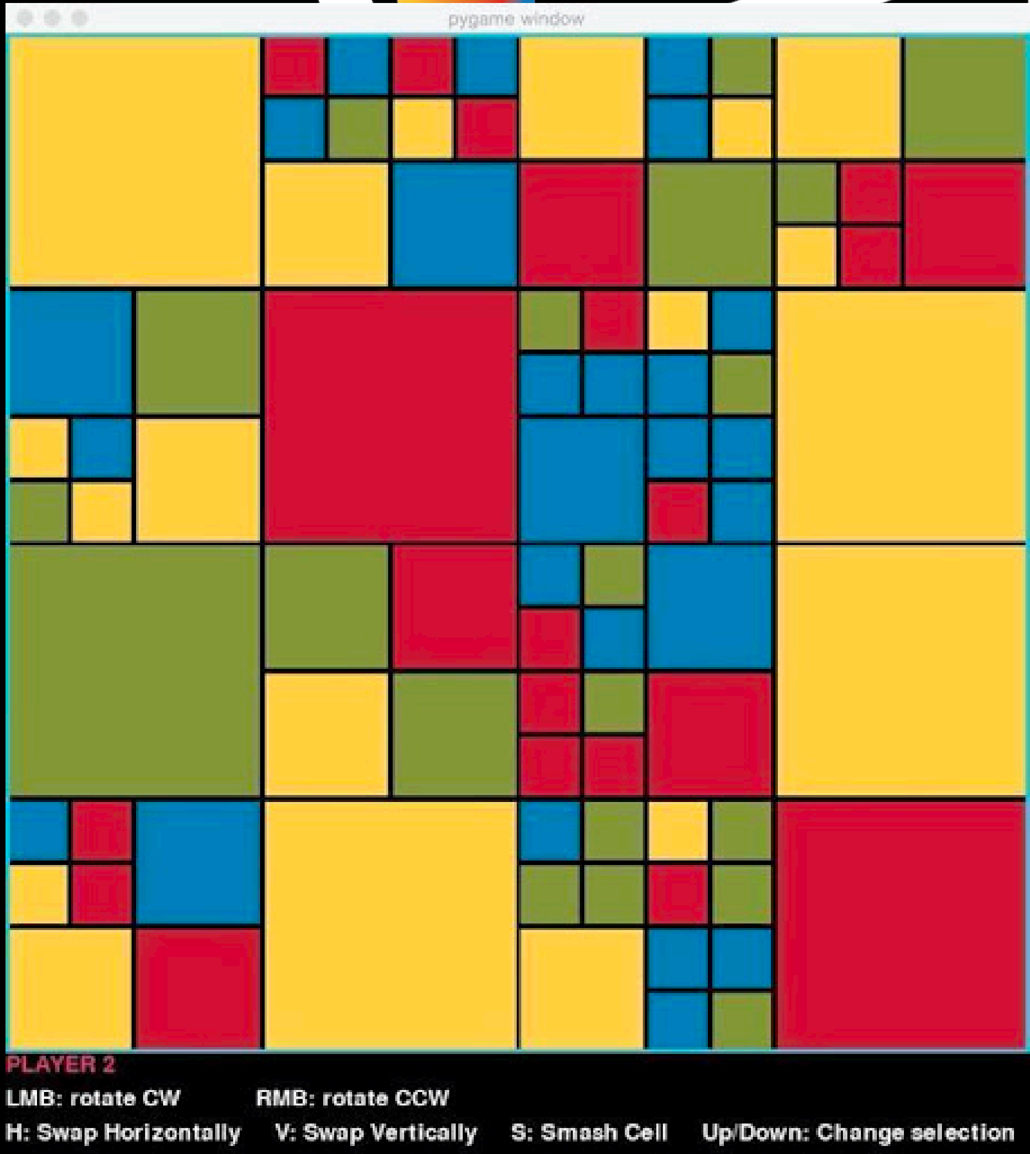
Keeping a single reference to each color and sharing those references among all the `Block` objects would save a lot of memory!

10 different blocks with 10 different colors → 10 different blocks with 8 different colors



A Better Example

Here's an example with much better savings: many many references to just four different color objects.



PLAYER 2

LMB: rotate CW RMB: rotate CCW

H: Swap Horizontally V: Swap Vertically S: Smash Cell Up/Down: Change selection

Flyweight Pattern

Solution:

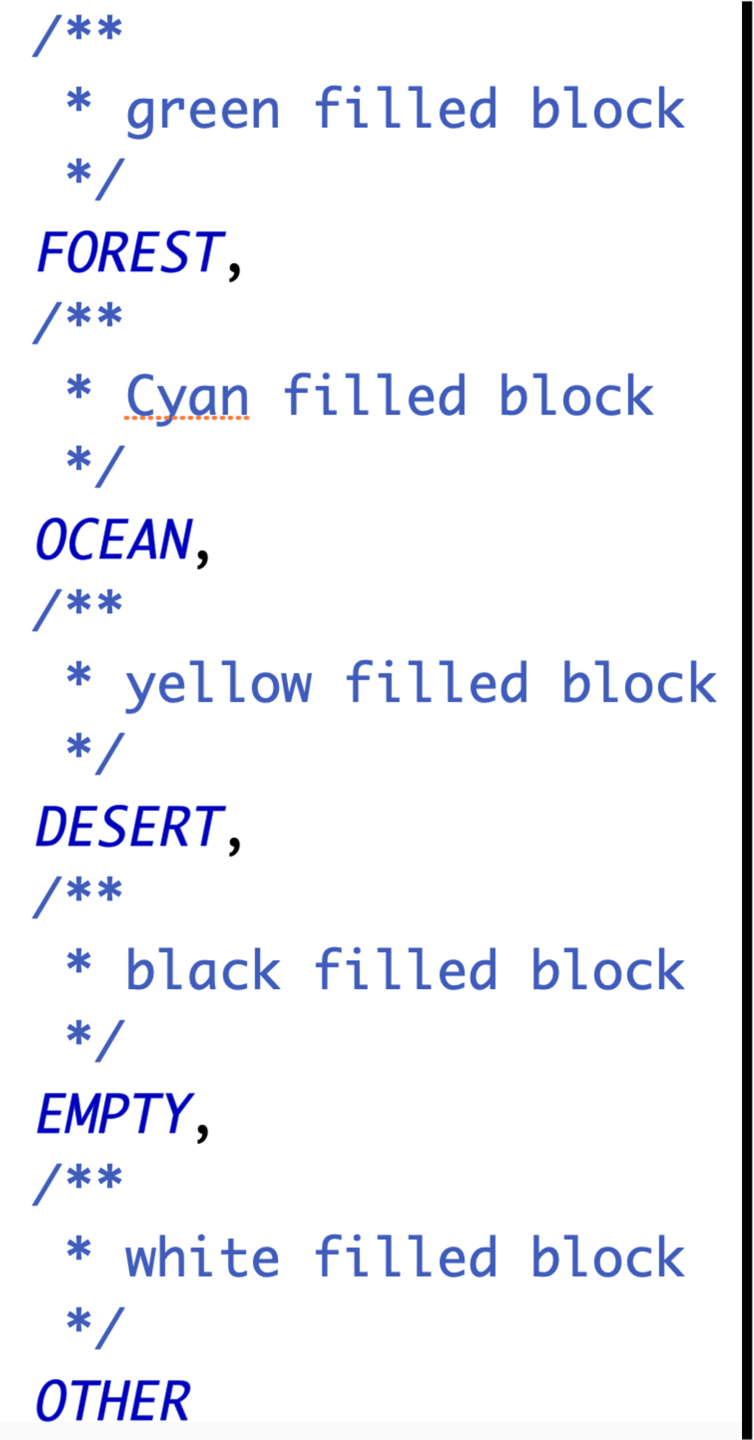
- Shared memory space
- A flyweight factory object is used to create and provide shared references as needed

It is recommended to make shared references immutable

Example

Flyweight:

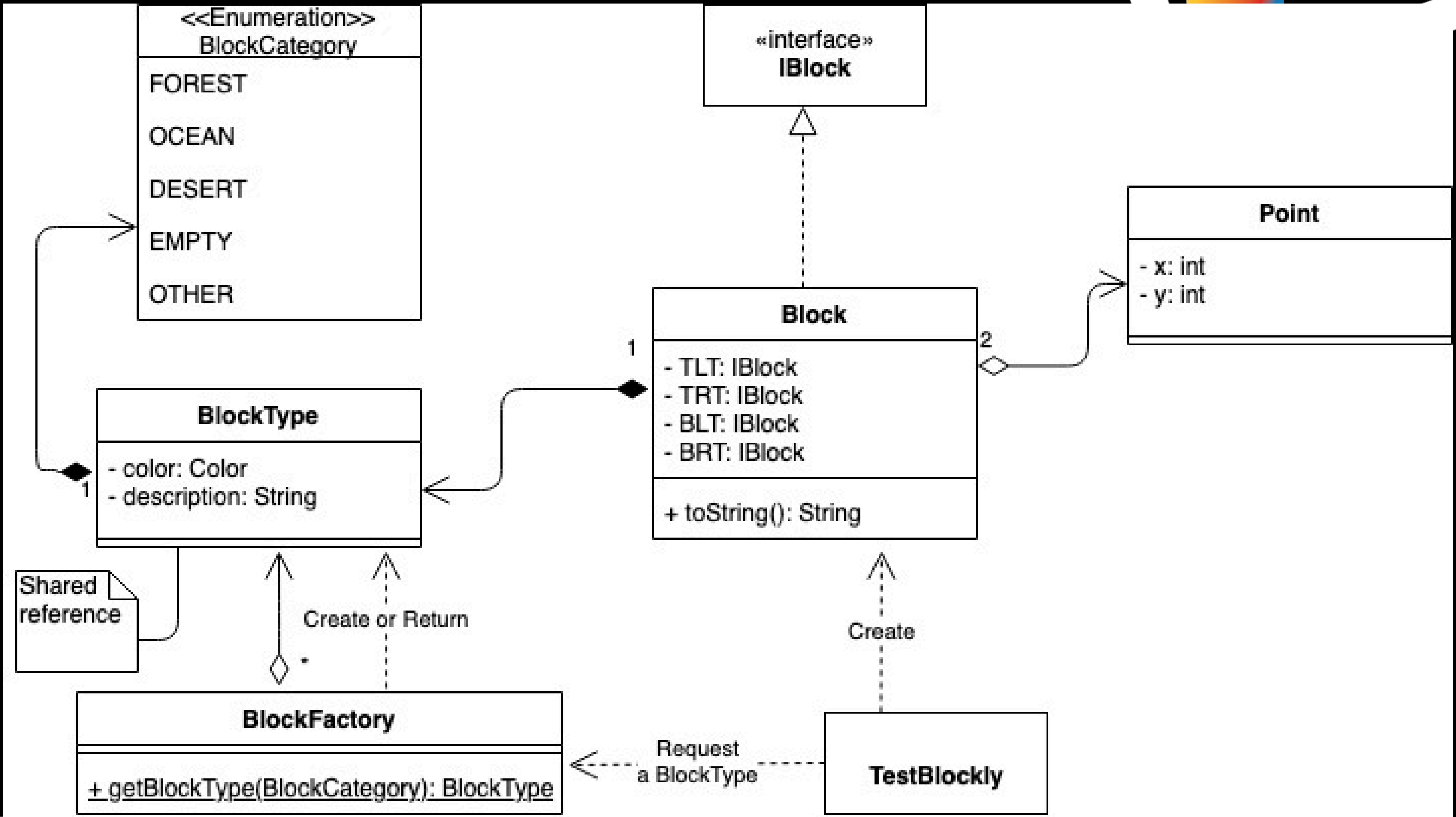
- Leaf nodes can be implemented using a reference to a single instance of the flyweight (one per category) to reduce memory costs.
- Nodes in the same category share state: Color, description, etc.



```
/**
 * green filled block
 */
FOREST,
/**
 * Cyan filled block
 */
OCEAN,
/**
 * yellow filled block
 */
DESERT,
/**
 * black filled block
 */
EMPTY,
/**
 * white filled block
 */
OTHER
```

Example: Class Design

Class	Purpose
BlockCategory	Enum type. Lists all the categories of Blocks
BlockType	The Flyweight data type. Maintains a reference to the Block category, color, and description. Shared reference
BlockFactory	Factory class. Creates new Flyweight objects or return existing ones. Flyweight objects are stored in a collection (Map) and are retrieved based on their category


<<Enumeration>>
BlockCategory
FOREST
OCEAN
DESERT
EMPTY
OTHER
«interface»
IBlock
Point
- x: int
- y: int
Block
- TLT: IBlock
- TRT: IBlock
- BLT: IBlock
- BRT: IBlock
+ toString(): String
1
2
BlockType
- color: Color
- description: String
1
Shared reference
Create or Return
*
Create
BlockFactory
+ getBlockType(BlockCategory): BlockType
Request a BlockType
TestBlockly

CIT 5940

Design Patterns: Visitor

Visitor Pattern

Behavioral pattern

- **Problem**: We want to perform an activity/operation on all objects in a collection
- **Goal**: Separate the activity from the object's specification

Visitor Pattern

Solution:

- Create a separate object called `"visitor"` that will implement the activity operation to be performed on the objects
- The objects in the collection `"accept"` the visitor and the visitor objects perform the activity

Example

We have 2 types of students (undergraduate and graduates) stored in a BST database for a class' gradebook

We want to update the grade of all students in the class to "curve" it using the following formula:

- Add 1 point to all undergraduate student GPAs
- Add 0.5 point to all graduate student GPAs

Example

We don’t want to include the “update” operation in the Student class definition.

- Why?

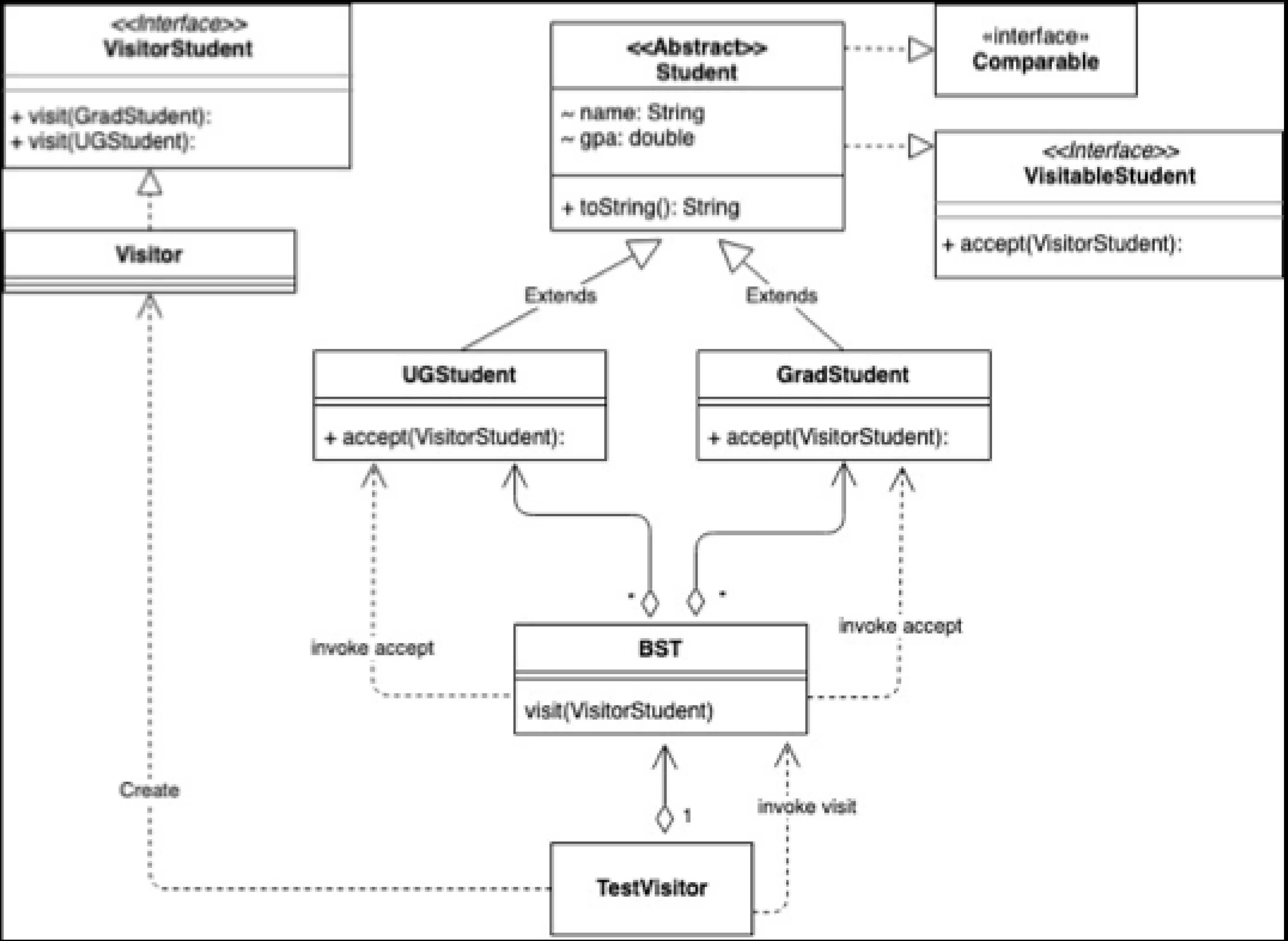
Example

We don’t want to include the “update” operation in the Student class definition.

- There are multiple ways that we might want to visit students in the future!
  - *print out all the grades? drop certain assignments?*
- Poor cohesion if a student is resposible for storing its own information **and** updating itself subject to external criteria

Class Design

Name	Type	Purpose
VisitorStudent	Interface	Defines the activity to be performed (`visit` method)
Visitor	Class	Implements the VisitorStudent activity (`visit` method)
VisitableStudent	Interface	Defines the `accept` method to pass the visitor
Student	Class	implements VisitableStudent operation (`accept`)


<<Interface>>
VisitorStudent
+ visit(GradStudent):
+ visit(UGStudent):
Visitor
<<Abstract>>
Student
~ name: String
~ gpa: double
+ toString(): String
«interface»
Comparable
<<Interface>>
VisitableStudent
+ accept(VisitorStudent):
Extends
Extends
UGStudent
+ accept(VisitorStudent):
GradStudent
+ accept(VisitorStudent):
*
*
invoke accept
invoke accept
BST
visit(VisitorStudent)
Create
1
invoke visit
TestVisitor

Extensibility & Anonymous Classes

The Visitor pattern means that we don't have to modify existing classes anytime we want to define a new way of visiting

- We can just implement `VisitorStudent` a new way
- We don't even have to write a new class: we can use an anonymous class

Grade Deflation!

Using an **anonymous class**, we can create a new instance of a `VisitorStudent` at the same time that we use it.

```
Database students = queryStudents();
students.visit(new VisitorStudent() {
    @Override
    public void visit(GradStudent student) {
        student.gpa -= 0.5;
    }

    @Override
    public void visit(UGStudent student) {
        student.gpa -= 1;
    }
});
```

Anonymous Classes

How to use:

- Make sure you have some interface, e.g. `MyInterface`
- Pass a reference to `new MyInterface() {...}` wherever an instance of `MyInterface` is expected
- Specify in the braces an implementation of each of the required methods

Avoid if:

- The implementation is long
- The implementation is used in multiple places