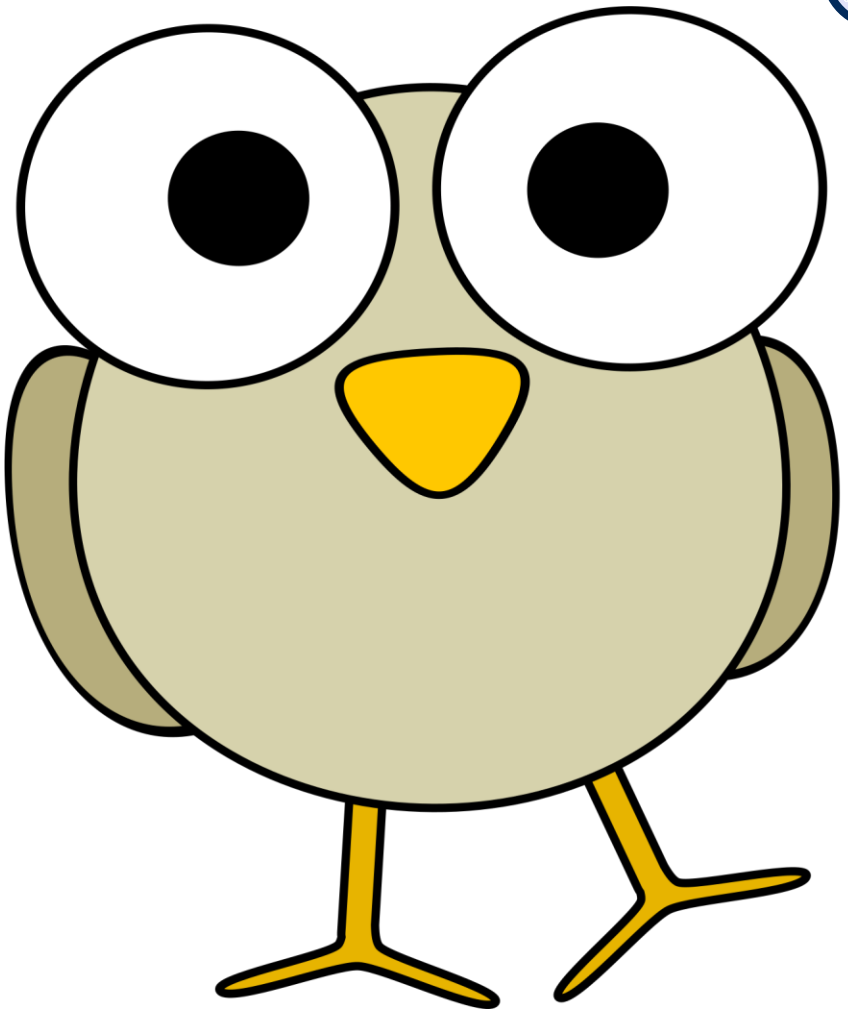
CSCI 200: Foundational Programming Concepts & Design Lecture 31

SOLID Principles

Previously in CSCI 200

- Runtime Polymorphism
  - Virtual function implementations bound at run time based on pointer object type
- "Pure Virtual Function" == Abstract Function
  - Virtual function with no default implementation
  - Abstract class → cannot instantiate
- Interfaces
  - Abstract class comprised of only abstract

Questions?



Learning Outcomes For Today

- Define the SOLID Principles.

- Discuss the Single Responsibility Principle.
- Discuss the Open/Closed Principle.
- Discuss the Liskov Substitution Principle.
- Discuss the Interface Segregation Principle.
- Discuss the Dependency Inversion Principle.

- Discuss the Program to an Interface Principle.
- Discuss the Favor Composition Over Inheritance Principle.

On Tap For Today

- SOLID Principles
  - Interface Segregation Principle
  - Open/Closed Principle
  - Liskov Substitution Principle
  - Dependency Inversion Principle
- Designing for Abstraction
- Practice

Design Principle

- "Program to an interface, not an implementation"

- Leverage polymorphism
  - Rely only on what operations can be done
  - More maintainable
  - Can change behavior at run time

On Tap For Today

- SOLID Principles
  - Interface Segregation Principle
  - Open/Closed Principle
  - Liskov Substitution Principle
  - Dependency Inversion Principle
- Designing for Abstraction
- Practice

SOLID Principles

- Set of design principles for object-oriented software development

- S – Single Responsibility Principle
- O – Open/Closed Principle
- L – Liskov Substitution Principle
- I – Interface Segregation Principle
- D – Dependency Inversion

On Tap For Today

- SOLID Principles
  - Interface Segregation Principle
  - Open/Closed Principle
  - Liskov Substitution Principle
  - Dependency Inversion Principle
- Designing for Abstraction
- Practice

Single Responsibility Principle

- Saw with functions

- "A class should have one, and only one, reason to change."
  - A class (or function) should have only one purpose.

Single Responsibility Principle

- If multiple responsibilities, needs to be modified more frequently == harder to maintain
- One responsibility
  - Easier to explain
  - Reduces number of bugs
  - Improves development speed
- Don't take to extreme! Don't make a class with one function
  - Then need to use too many objects to accomplish anything

SRP Bad

```
class TextEditor {
public:
   void spellCheck() { ... }
   void formatText() { ... }
}
```

SRP Good

```
class TextEditor {
public:
   void formatText() { ... }
}
class SpellChecker {
public:
   void spellCheck() { ... }
}
```

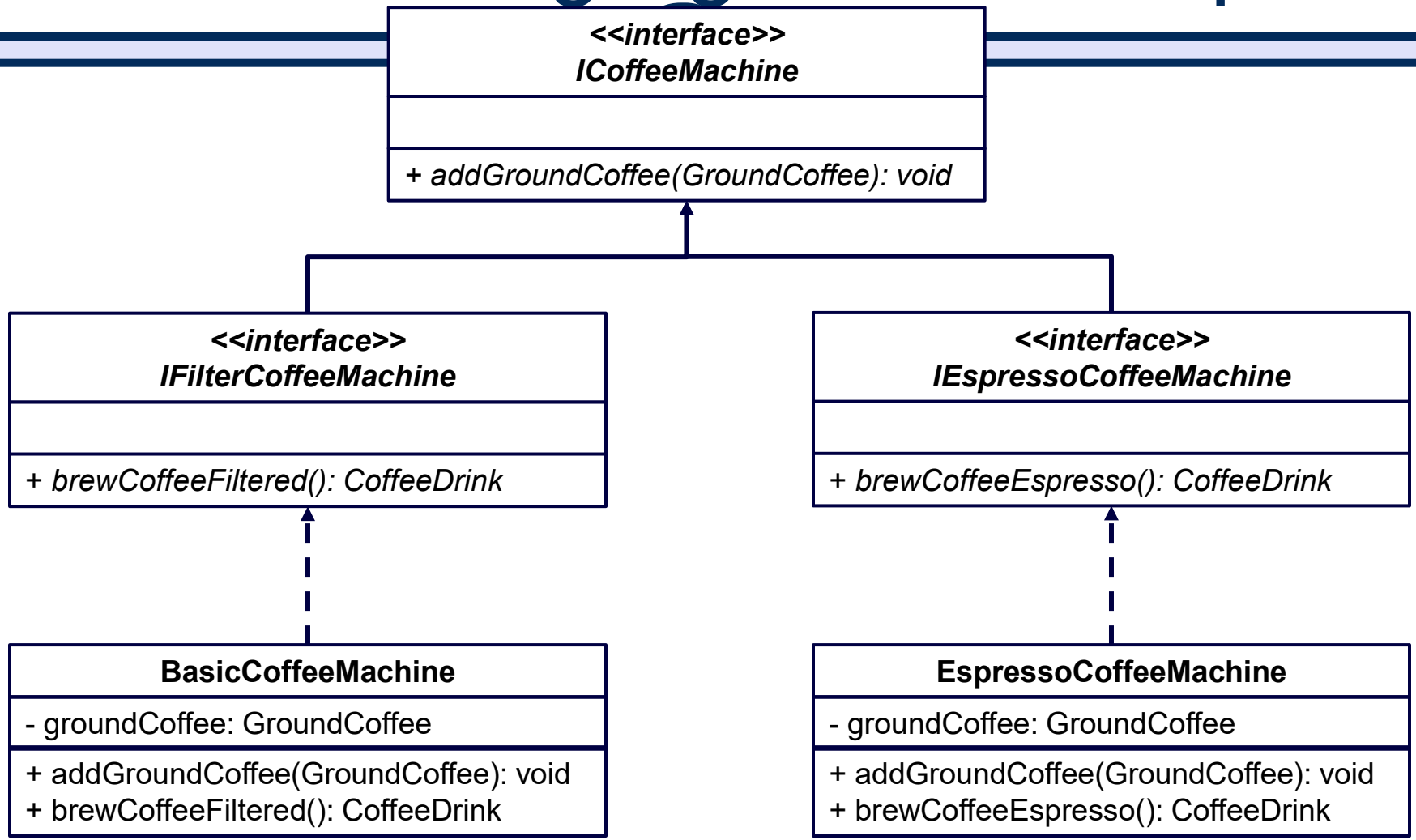
On Tap For Today

- SOLID Principles
  - Interface Segregation Principle
  - Open/Closed Principle
  - Liskov Substitution Principle
  - Dependency Inversion Principle
- Designing for Abstraction
- Practice

Interface Segregation Principle

- "*Clients should not be forced to depend upon interfaces that they do not use.*"
  – Robert C. Martin when consulting for Xerox

Interface Segregation Principle



On Tap For Today

- SOLID Principles
  - Interface Segregation Principle
  - Open/Closed Principle
  - Liskov Substitution Principle
  - Dependency Inversion Principle
- Designing for Abstraction
- Practice

Open/Closed Principle

- "*Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.*"

Open for Extension

- Classes can be inherited from
- Interfaces can be implemented

Closed for Modification

- Don't change original class
- Can mark functions that cannot be overridden further as **`final`**

```
class A {
public:
  virtual void foo() { cout << "A::foo()" << endl; }        // A::foo() is open
  virtual void bar() final { cout << "A::bar()" << endl; } // A::bar() is closed
};

class B : public A {
public:
  void foo() override final { cout << "B::foo()" << endl; } // B::foo() is closed
  // void bar() { cout << "B::bar()" << endl; }              // A::bar() is closed
                                                             // can't be overridden
};

class C : public B {
public:
  // void foo() { cout << "C::foo()" << endl; }              // B::foo() is closed
                                                             // can't be overridden
};
```

Closed for Modification

- Don’t change original class
- Can mark classes as **`final`** that cannot be inherited from, makes functions final in turn

```
class A {
public:
  virtual void foo() { cout << "A::foo()" << endl; }
};

class B final : public A {
public:
  void foo() override { cout << "B::foo()" << endl; }
};

// class C : public B { // B cannot be inherited from
//                      // since it is closed
// };
```

Open/Closed Principle

- Classes can be inherited from
- Interfaces can be implemented
- In C++ by default:
  - Classes are open
  - Member functions are virtual (open)
- Can only “close” things
  - By using `virtual` & `final` keywords
  - Otherwise, anyone can extend/override

Open/Closed Principle

- In practice:
  - Extend an interface/class
    - Add more functionality to what already exists
    - Do not override an existing implementation

Open/Closed Principle

- Good practice:
  - Implement interfaces
  - Don’t extend classes, unless you can ensure the next principle...

On Tap For Today

- SOLID Principles
  - Interface Segregation Principle
  - Open/Closed Principle
  - Liskov Substitution Principle
  - Dependency Inversion Principle
- Designing for Abstraction
- Practice

Liskov Substitution Principle

- "Let $\Phi(x)$ be a property provable about objects x of type T. Then $\Phi(y)$ should be true for objects y of type S where S is a subtype of T."

  - Barbara Liskov

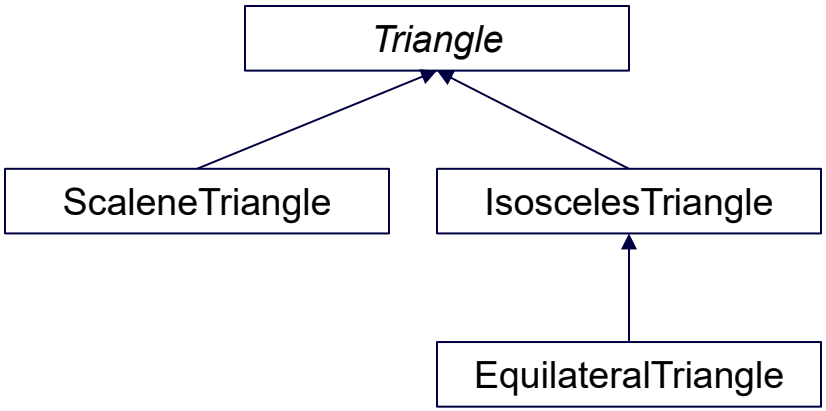
Liskov Substitution Principle

- Objects of a superclass shall be replaceable with objects of its subclasses without breaking the application
  - Objects of the subclass behave in the same way as objects of the superclass
    - Overridden method needs to accept same input parameters. Cannot enforce stricter validation rules
    - Overridden method needs to return same values. Can be subclass or subset of valid values

Classic Example

- *A Square is a Rectangle*
  - "is a" = inheritance
  - Square subtypes Rectangle
- By Liskov Substitution, a program using Rectangle should be able to be replaced with Square with no side effect
  - setHeight() & setWidth() don't make sense because changing one changes the other
- *But not all Rectangles are Squares*

Similar Scenario – Enforced!



Liskov Substitution Principle

- Objects of a superclass shall be replaceable with objects of its subclasses without breaking the application

- Program to an interface, not an implementation.
  - Leverage runtime polymorphism
  - Will see this again with the “D” principle

On Tap For Today

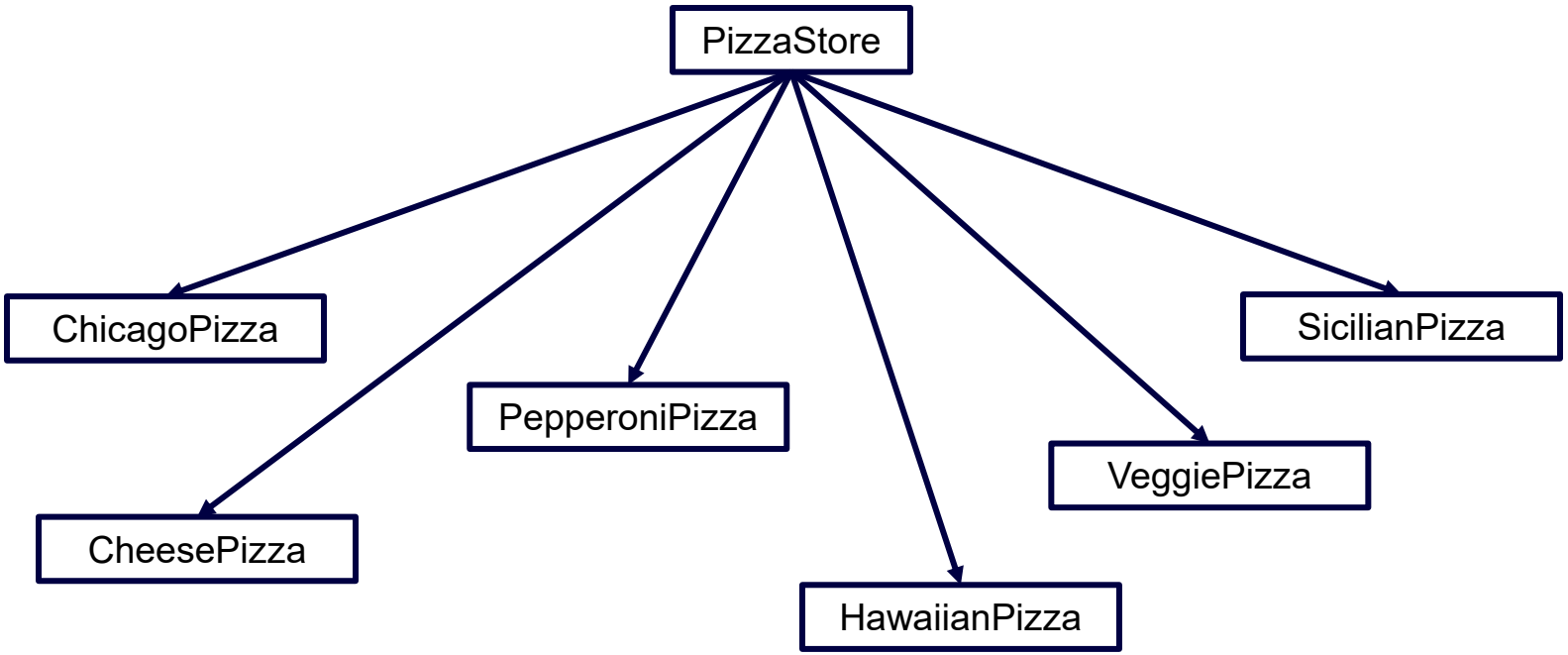
- SOLID Principles
  - Interface Segregation Principle
  - Open/Closed Principle
  - Liskov Substitution Principle
  - Dependency Inversion Principle
- Designing for Abstraction
- Practice

Dependency Inversion Principle

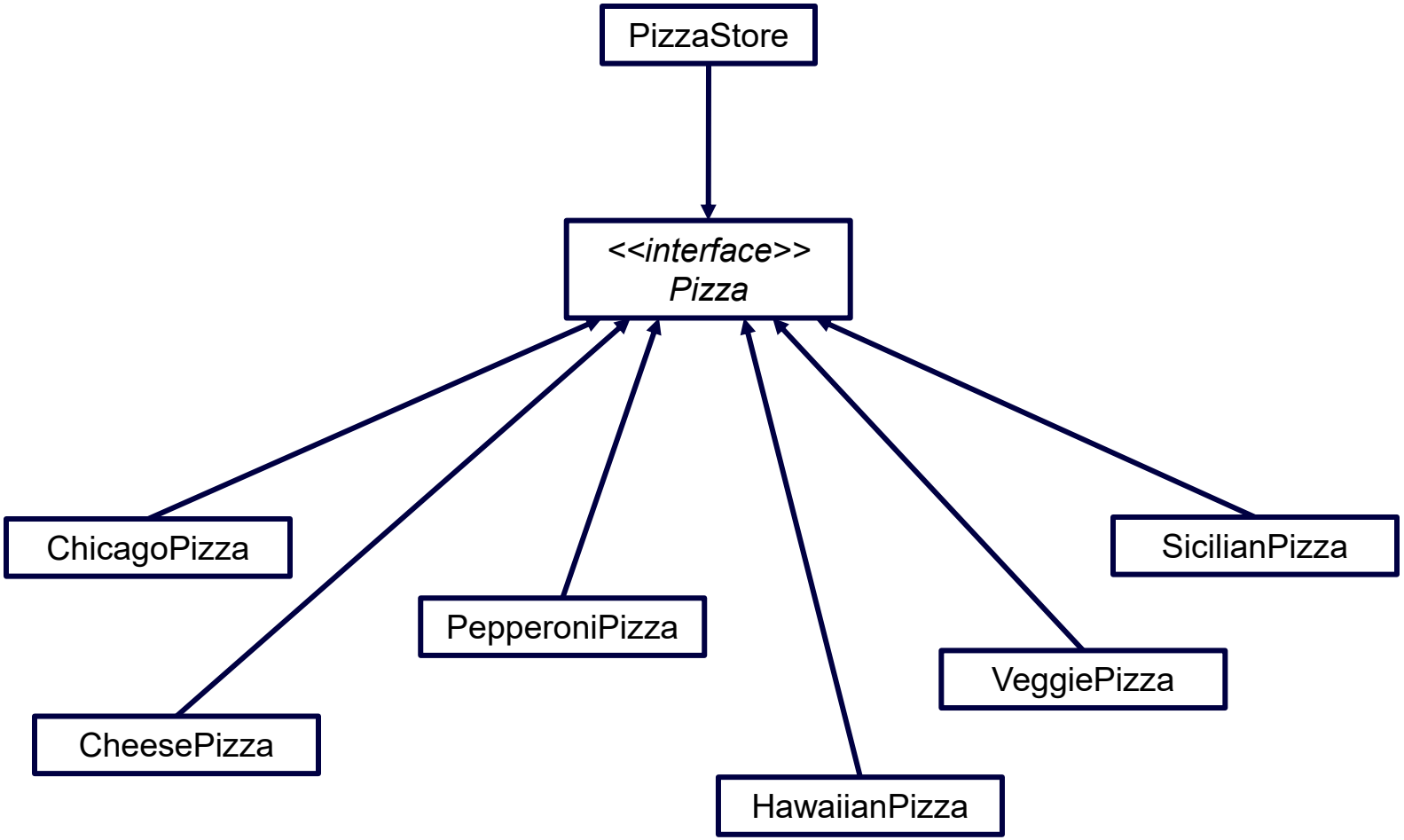
1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

- In other words...
  - Depend upon abstractions. Do not depend upon concrete classes.

DIP Violated



DIP Enforced



Dependency Inversion Principle

- If already following Open/Closed Principle and Liskov Substitution Principle,
  - then Dependency Inversion Principle should already be enforced.

On Tap For Today

- SOLID Principles
  - Interface Segregation Principle
  - Open/Closed Principle
  - Liskov Substitution Principle
  - Dependency Inversion Principle
- Designing for Abstraction
- Practice

SOLID Principles

- SRP: *"A class should have one, and only one, reason to change."*
- OCP: "*Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."*
- LSP: "*Let $\Phi(x)$ be a property provable about objects x of type T. Then $\Phi(y)$ should be true for objects y of type S where S is a subtype of T.*"
- ISP: "*Clients should not be forced to depend upon interfaces that they do not use.*"
- DIP: "*Depend upon abstractions. Do not depend upon concrete classes.*"

SOLID Principles

https://www.globalnerdy.com/2009/07/15/the-solid-principles-explained-with-motivational-posters/

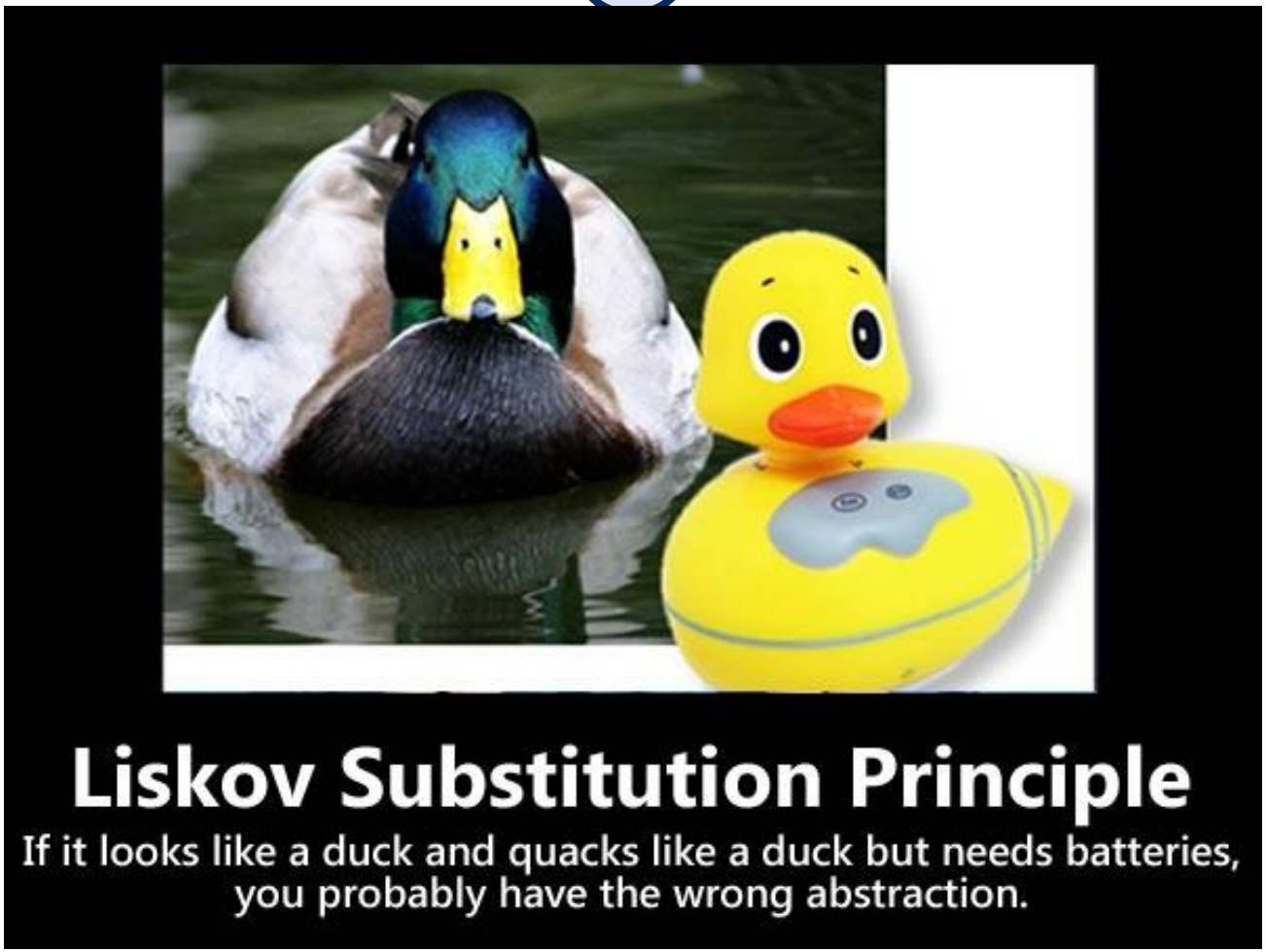
SOLID Principles

https://www.globalnerdy.com/2009/07/15/the-solid-principles-explained-with-motivational-posters/

SOLID Principles

https://www.globalnerdy.com/2009/07/15/the-solid-principles-explained-with-motivational-posters/

SOLID Principles

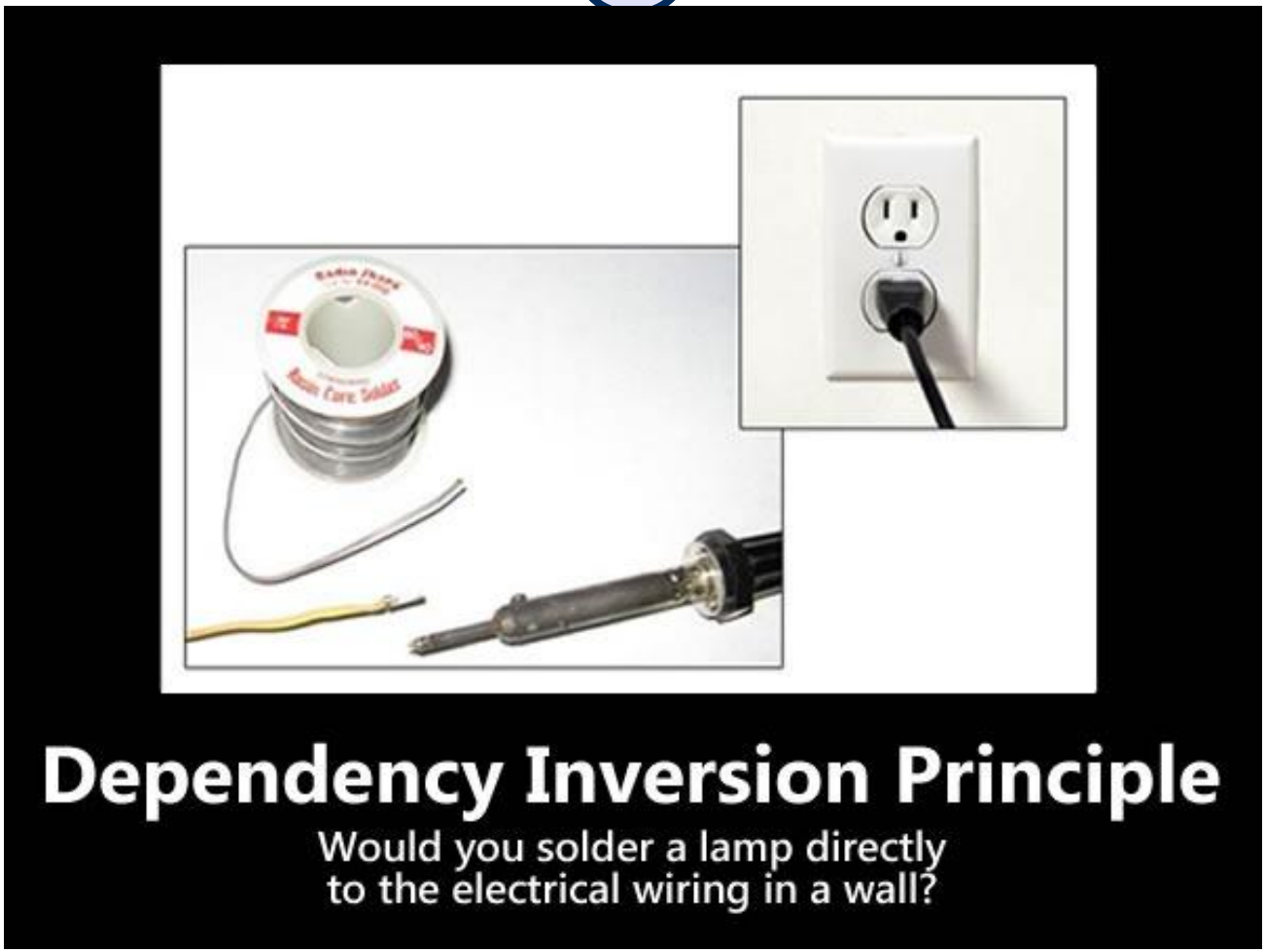


https://www.globalnerdy.com/2009/07/15/the-solid-principles-explained-with-motivational-posters/

SOLID Principles

https://www.globalnerdy.com/2009/07/15/the-solid-principles-explained-with-motivational-posters/

SOLID Principles



https://www.globalnerdy.com/2009/07/15/the-solid-principles-explained-with-motivational-posters/

SOLID Principles

- "Program to an interface, not an implementation."
  1. No variable should hold a reference to a concrete class.
     - Liskov Substitution: hold reference to interface type
     - Leverages runtime polymorphism

SOLID Principles

- "Program to an interface, not an implementation."
  1. No variable should hold a reference to a concrete class.
  2. No class should derive from a concrete class.
     - Dependency Inversion: If deriving from a concrete class, then depending upon a concrete class. Depend upon an abstract interface instead.
     - Open/Closed: Interfaces are closed for modification but open for extension by adding additional interface methods

SOLID Principles

- "Program to an interface, not an implementation."
  1. No variable should hold a reference to a concrete class.
  2. No class should derive from a concrete class.
  3. No method should override an implemented method of any of its base classes.
     - Dependency Inversion: If overriding an implemented method, then base class wasn't an abstraction. Methods implemented in base class are meant to be shared by all subclasses

On Tap For Today

- SOLID Principles
  - Interface Segregation Principle
  - Open/Closed Principle
  - Liskov Substitution Principle
  - Dependency Inversion Principle
- Designing for Abstraction
- Practice

SimUDuck App

- A duck pond simulation
- *Identify the aspects that vary and separate them from what stays the same.*

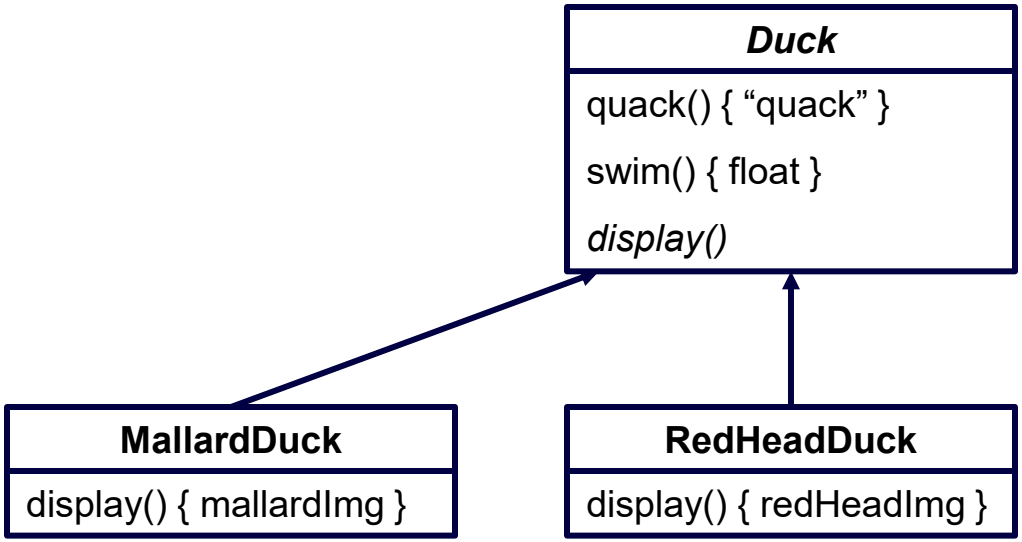
  → **Abstract what's the same, encapsulate what varies.**

MallardDuck
quack() { "quack" }
swim() { float }
display() { mallardImg }

RedHeadDuck
quack() { "quack" }
swim() { float }
display() { redHeadImg }

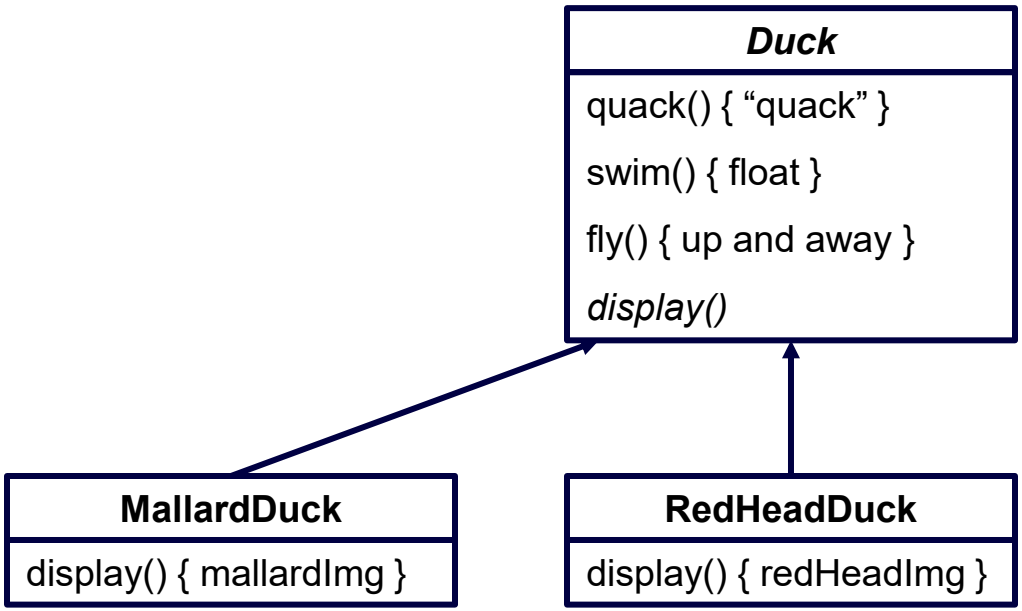
SimUDuck App

- Now need the ducks to be able to fly



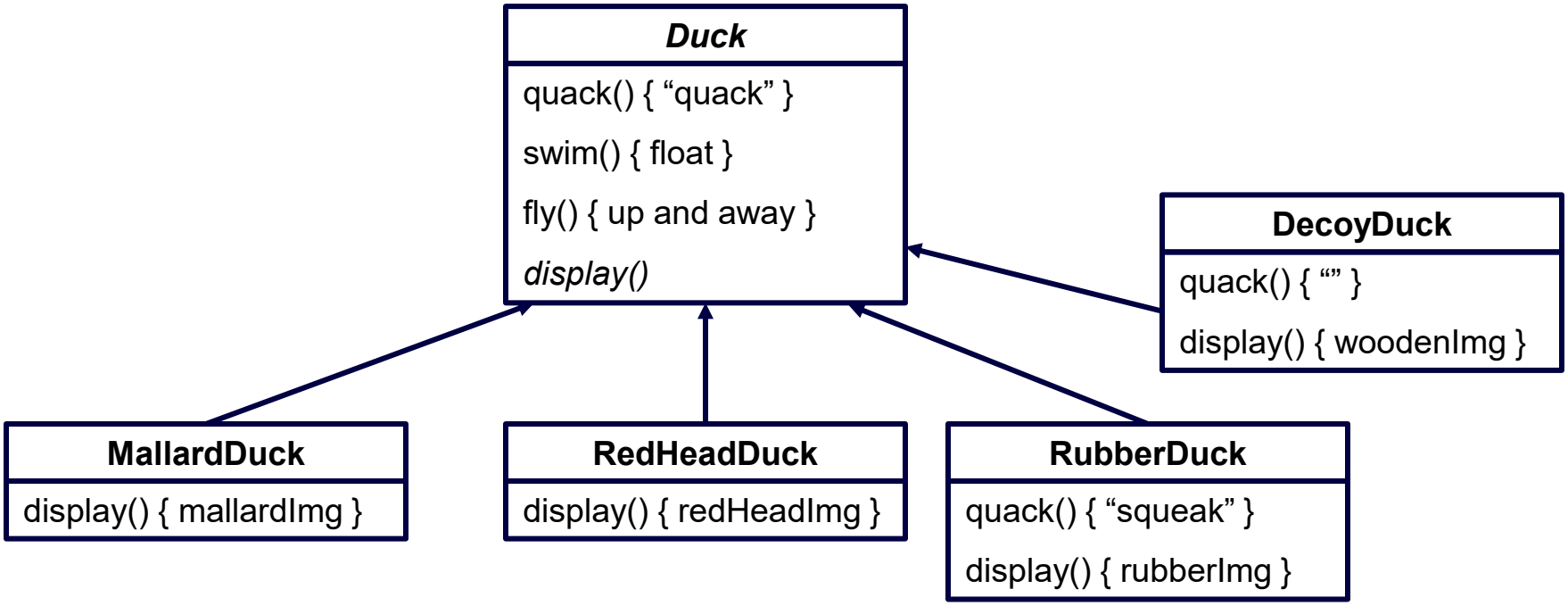
SimUDuck App

- OOP! All ducks can fly!



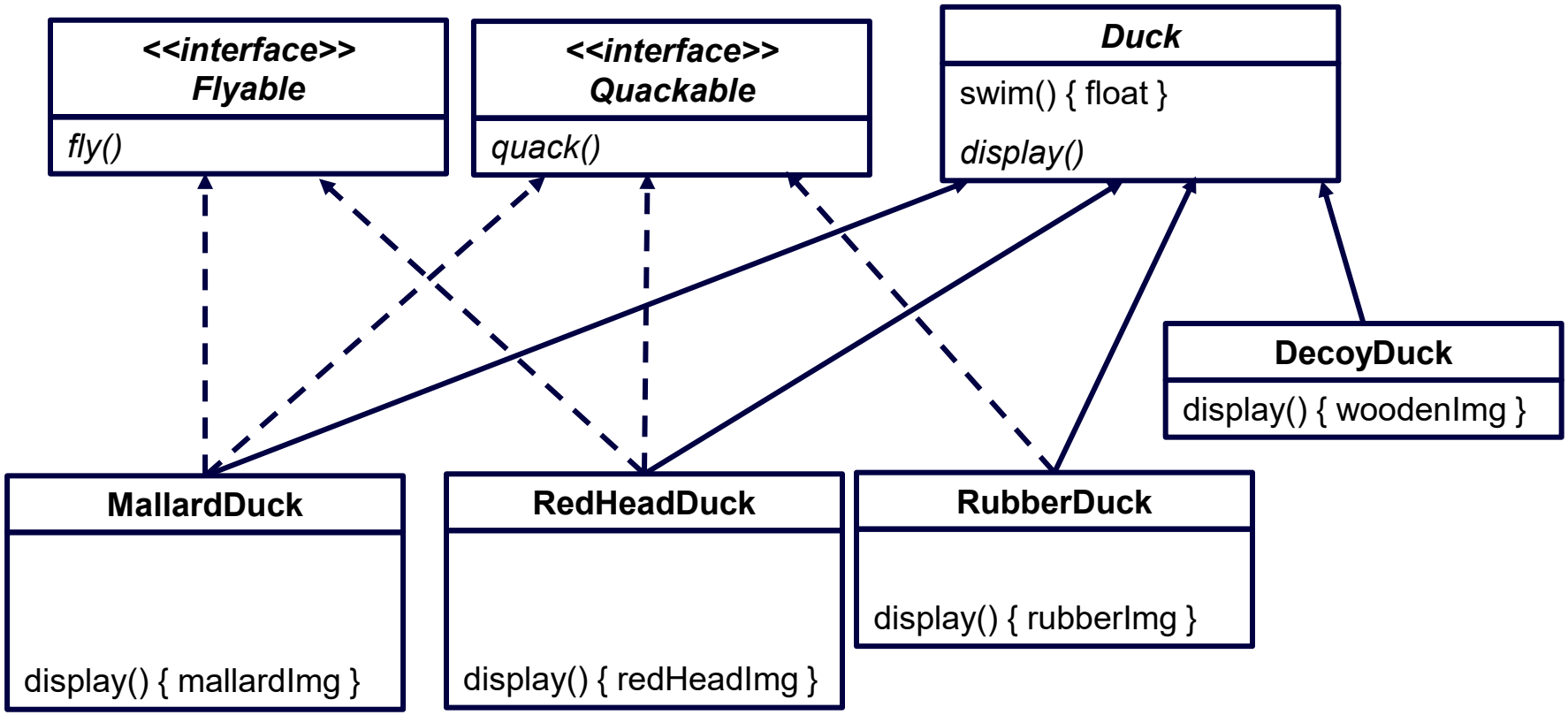
SimUDuck App

- OOP! ALL ducks can fly! ☹



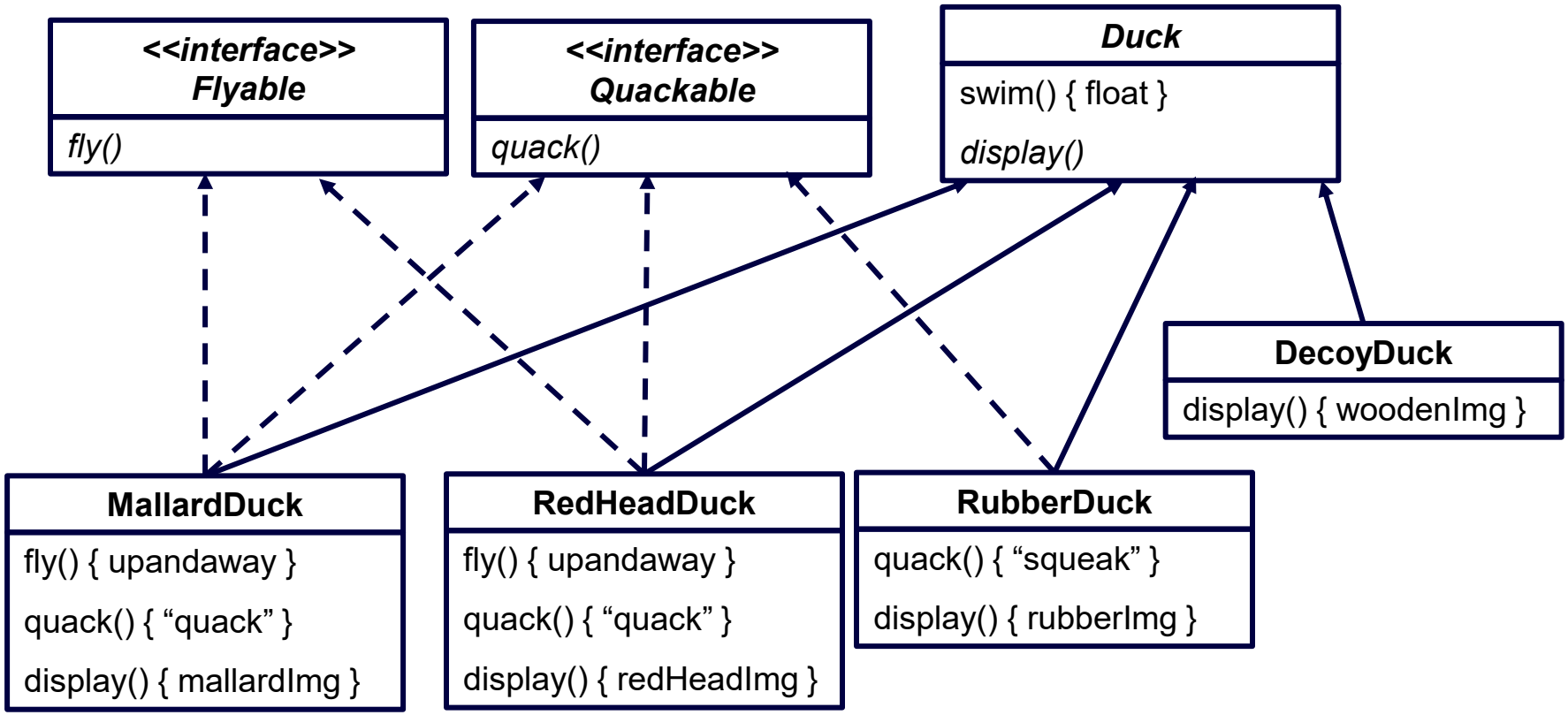
SimUDuck App

- Program to an interface...



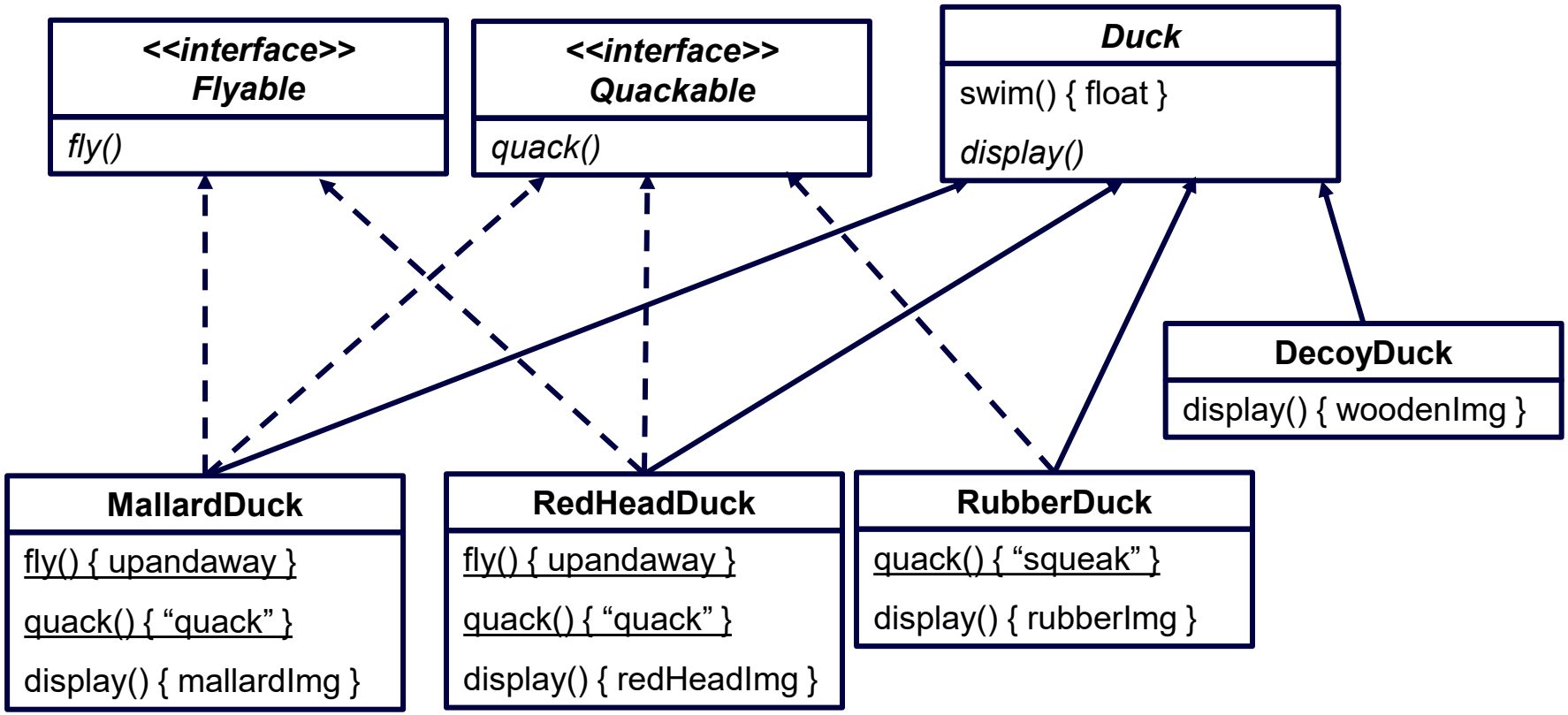
SimUDuck App

- Program to an interface...and duplicate code ☹



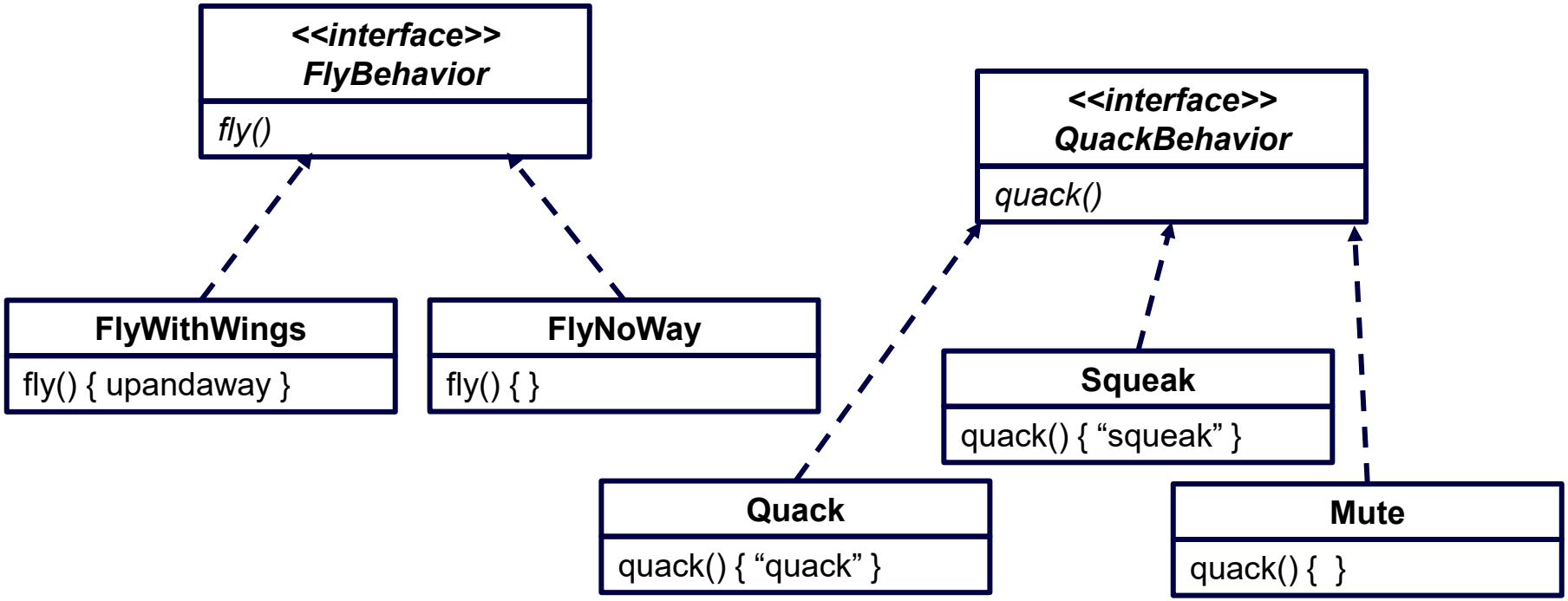
SimUDuck App

- *Identify the aspects that vary and separate them from what stays the same. Abstract what's the same, encapsulate what varies.*



SimUDuck App

- *Identify the aspects that vary and separate them from what stays the same. Abstract what's the same, encapsulate what varies.*
- *Program to an interface, not an implementation.*

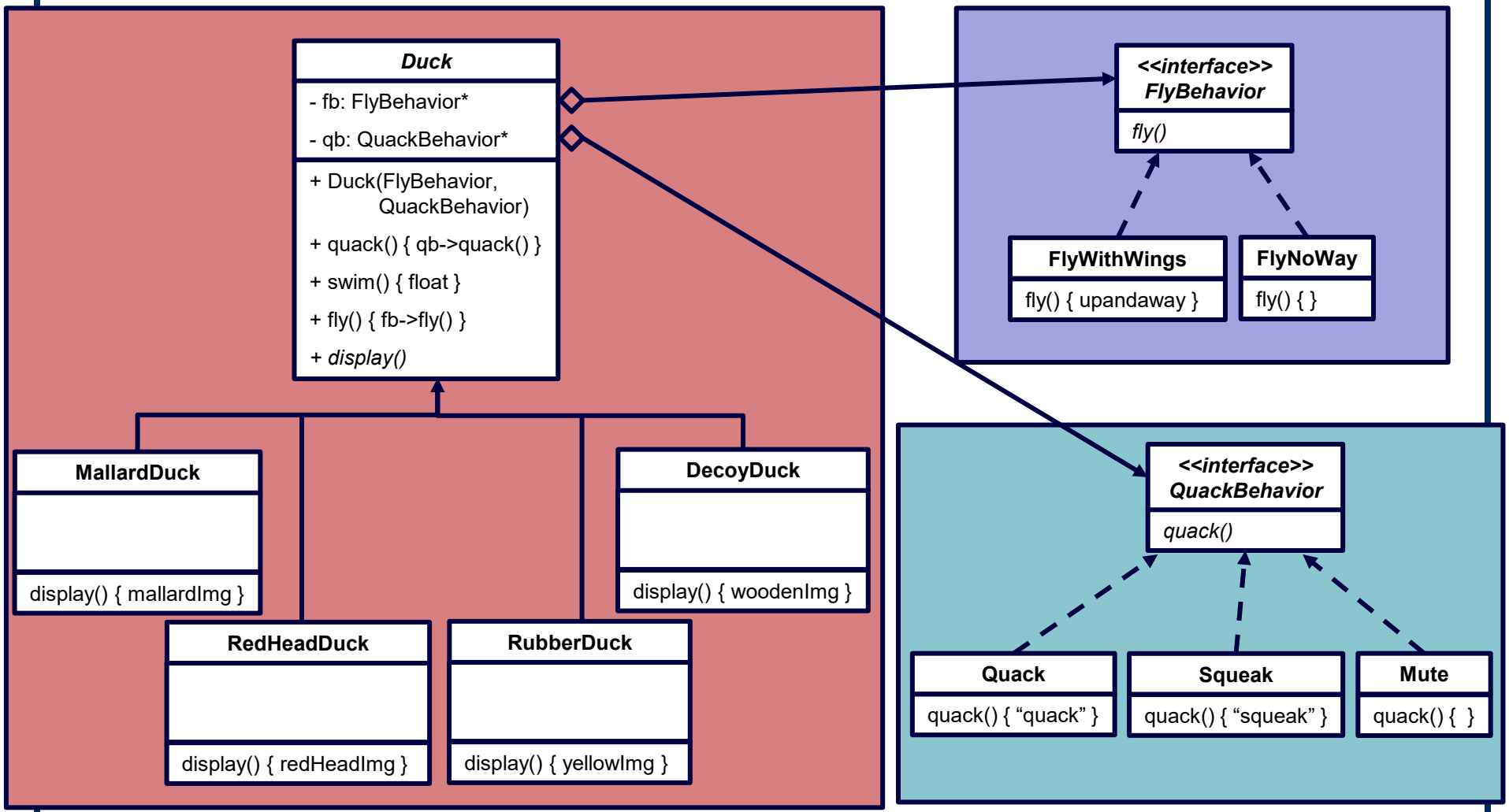


SimUDuck App

- *Favor composition over inheritance.*

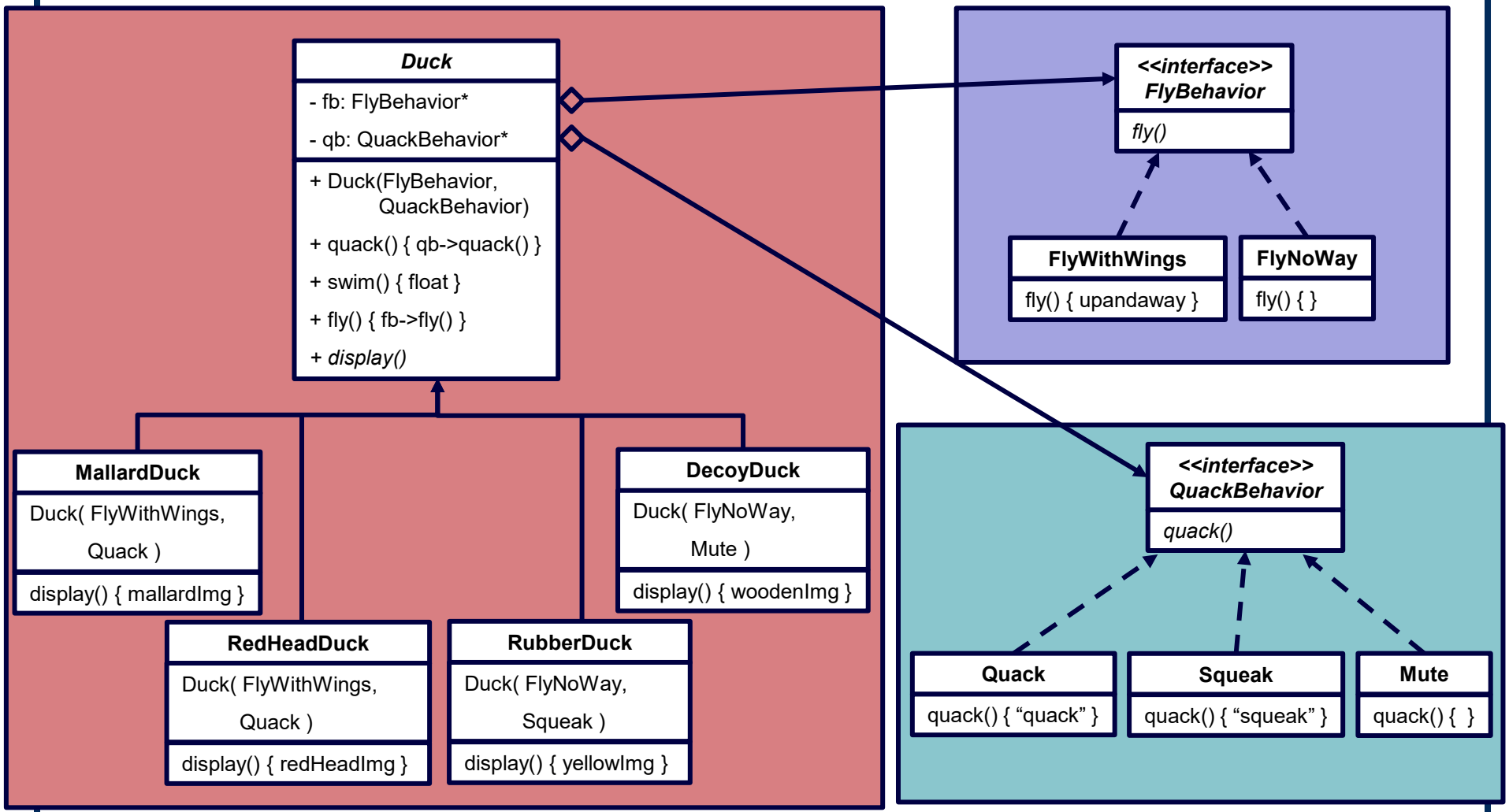
SimUDuck App

- *Favor composition over inheritance.*



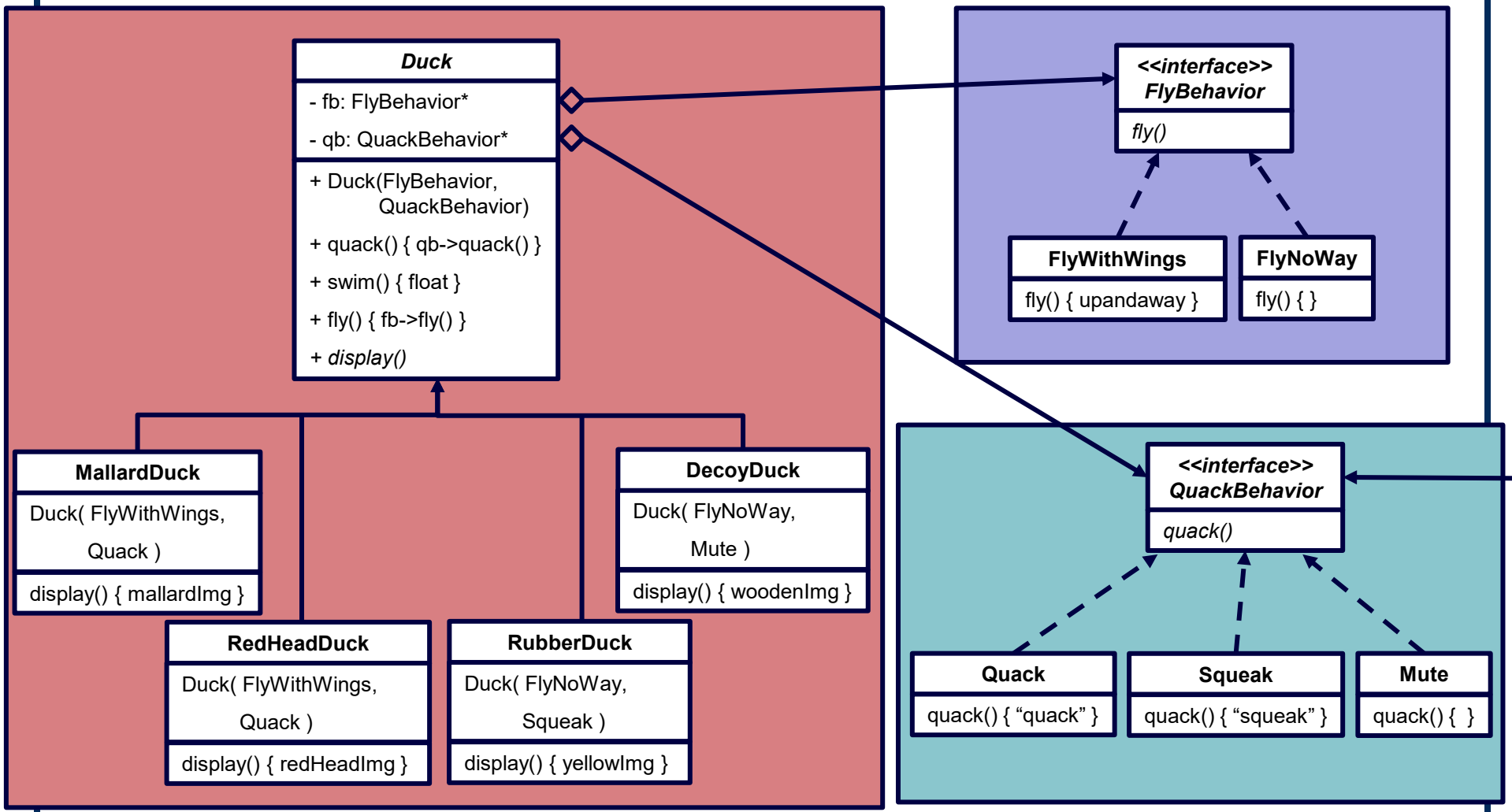
SimUDuck App

- *Favor composition over inheritance.*



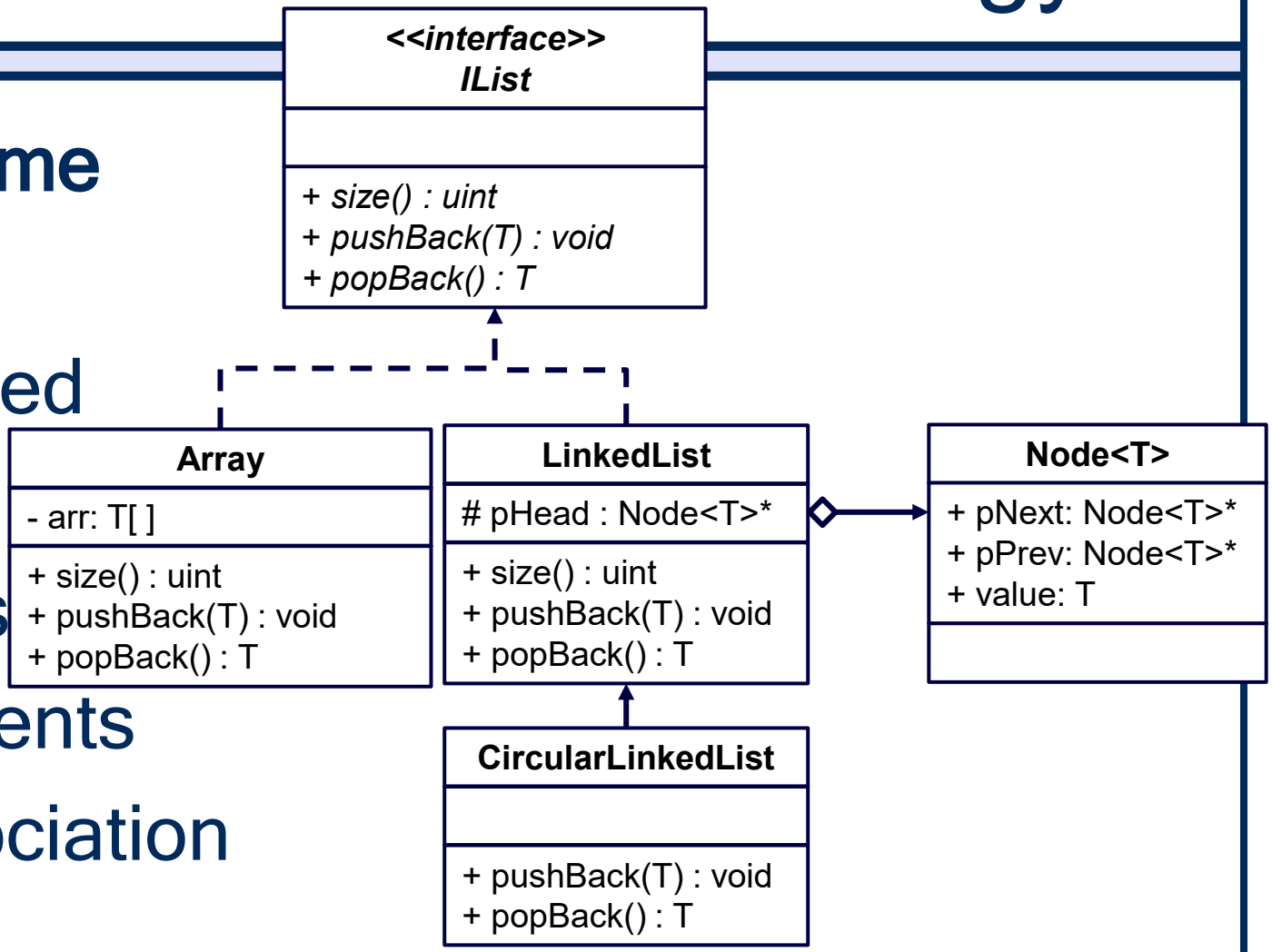
SimUDuck App

- *Favor composition over inheritance.*



UML Notation & Terminology

- **ClassName**
- + public
- # protected
- - private
- extends
- implements
- association
- *abstract*



Inheritance OR Composition?

- Does TypeB want to expose the complete interface (all public methods) of TypeA such that TypeB can be used where TypeA is expected?
  - Liskov Substitution
  - Use **Inheritance**
- *A Cessna biplane will expose the complete interface of an Airplane. Cessna derives from Airplane*

Inheritance OR Composition?

- Does TypeB only want some/part of the behavior exposed of TypeA?
  - Use **Composition**
- *A Bird needs only the fly behavior of an Airplane. Extract the fly behavior out as an interface and make it a member of both classes.*

On Tap For Today

- SOLID Principles
  - Interface Segregation Principle
  - Open/Closed Principle
  - Liskov Substitution Principle
  - Dependency Inversion Principle
- Designing for Abstraction
- Practice

To Do For Next Time

- Keep working on Set5
  - L5A
  - A5
- Keep working on Final Project
- Inheritance + SOLID Quiz on Wed Nov 19