CSC 3380 Aymond

Homework Assignment

- Enterprise Architect Component Diagram, Due 11PM 2/19

Project News

- Next Milestone: #2
  - Due Friday 2/21, 11PM
  - Upload to Moodle (1 upload for entire team)
  - Outline is in Project Kickoff Lecture Notes
  - All UML diagrams must be developed in EA

Section 1

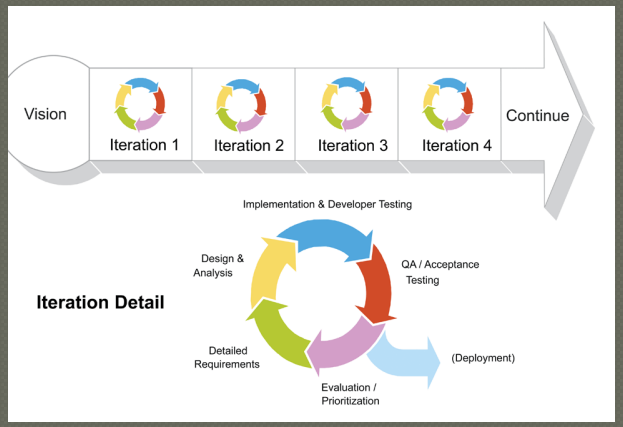
2/12/2020

Let's Talk Project

- This is a Communication Intensive Course
  - CI courses must provide a feedback loop and the opportunity for communication improvement
  - Our communication skill for this course is ***technology***
  - i.e., the design work in your project portfolio
  - So…you should get feedback on your portfolio at each milestone, and you're expected to improve it for the next milestone
    - In other words, those sections will be regraded for each milestone
    - If you have not received sufficient feedback from your TA to improve your portfolio, you need to reach out to him/her to get that input
    - I STRONGLY RECOMMEND THAT YOU MEET WITH YOU TA AT LEAST ONCE A WEEK
      - You may want to set up a regular meeting time

Project Lifecycle

- **Don't give into the temptation of using the waterfall project lifecycle!**
- You should develop your product using an iterative development lifecycle



- For each milestone
  - Your project portfolio will increase in scope
  - Previous sections should also morph over time, as you revisit phases in your iterations

These milestones look suspiciously like the waterfall model

Milestone 1: Stories & Requirements

- Project Portfolio
  - Description of problem & proposed solution
  - Team Structure
    - Team member/ role(s)/ responsibilities
  - Requirements
    - Stakeholder Issued Requirements
    - Epics [Revibe is willing to review your Epics prior to Milestone 1, if you get them to them early enough]
      - User Stories
        - Acceptance Criteria
- Due: 11PM 2/4
- 10% of project grade

Milestone 2: System Architecture

- Project Portfolio
  - Description of problem & proposed solution
  - Team Structure
    - Team member/ role(s)/ responsibilities
  - Requirements
    - Stakeholder Issued Requirements
    - Epics
      - User Stories
        - Acceptance Criteria
  - Design
    - System Architecture [in Enterprise Architecture]
      - User I/O
      - External Data Sources
      - Major Components
        - Interfaces
        - Data Flow
- Source Code
  - eap file(s) of System Architecture
  - Zip of all source code implemented at this point

Milestone 2: System Architecture

- Due 11PM 2/21
- 25% of project grade
- In-class presentations 3/2
  - Chanuka's teams: 1240 PFT
  - Clinton's teams: 1245 PFT
  - Qing's teams: 1258 PFT

Milestone 3: Component Designs

- Project Portfolio
  - Description of problem & proposed solution
  - Team Structure
    - Team member/ role(s)/ responsibilities
  - Requirements
    - Stakeholder Issued Requirements
    - Epics
      - User Stories
        - Acceptance Criteria
  - Design
    - System Architecture in Enterprise Architecture
      - User I/O
      - External Data Sources
      - Major Components
        - Interfaces
        - Data Flow
    - Component Designs in Enterprise Architecture
      - Interfaces
      - External Data Sources
      - Subcomponents, as applicable
      - Data Flow
      - Control Flow
- Source Code
  - eap file(s) of System Architecture
  - eap files of all component designs
  - Zip of all source code implemented at this point
- Due 11PM 3/17
- 25% of overall grade

Milestone 4: Working Prototype

- Project Portfolio
  - Description of problem & proposed solution
  - Team Structure
    - Team member/ role(s)/ responsibilities
  - Requirements
    - Stakeholder Issued Requirements
    - Epics
      - User Stories
        - Acceptance Criteria
  - Design
    - System Architecture in Enterprise Architecture
      - User I/O
      - External Data Sources
      - Major Components
        - Interfaces
        - Data Flow
    - Component Designs in Enterprise Architecture
      - Interfaces
      - External Data Sources
      - Subcomponents, as applicable
      - Data Flow
      - Control Flow
      - Class Diagrams of Design Patterns Employed
- Source Code
  - eap file(s) of System Architecture
  - eap files of all component designs
  - eap files of class diagrams
  - Zip of all source code implemented
- Due 11PM 4/21
- 25% of project grade

Final Presentation
8 minute time limit

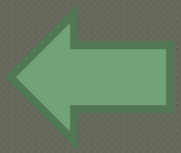
- Project Portfolio Presentation
  - Description of problem & proposed solution
  - Team Structure
    - Team member/ role(s)/ responsibilities
  - Requirements
    - Revibe requirements
    - Epics
    - Key user stories
  - Design
    - System Architecture in Enterprise Architecture
    - Component Designs in Enterprise Architecture
    - Class Diagram(s) of patterns employed
- Working Prototype Demonstration
  - Key features of product
- During class: 4/22, 4/27, & 4/29
- 10% of project grade

Design for Change

- What does that mean?
- Your project WILL change before the end of the semester

Unified Modeling Language (UML)

- Introduction to UML
- Architectural Modeling
- **Structural Modeling**
- Behavioral Modeling

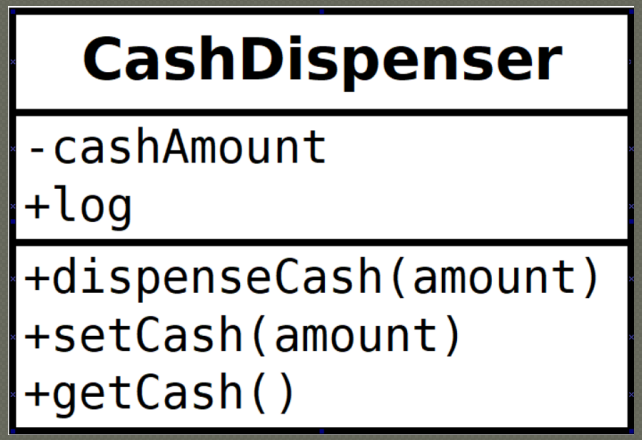


Structural Modeling

- **Class Diagrams**
  - Classes, features, and relationships
- Object Diagrams
  - Example configurations of instances
- Communication Diagrams
  - structural organization of the objects that send and receive messages
- Packages
  - Compile-time hierarchic structure

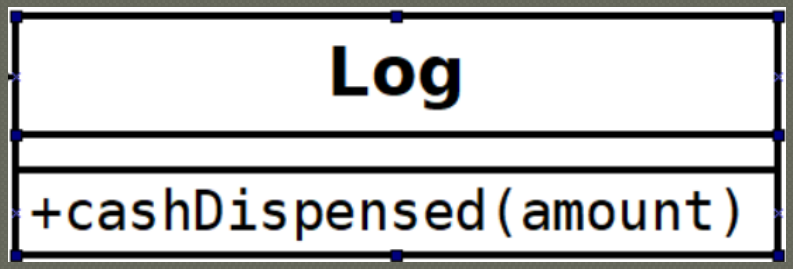
Class Diagrams

- Class diagrams are the most commonly used of the UML diagrams
  - 3 basic parts: Name, Fields (member variables), Methods
  - Visibility
    - + public
    - - private
    - # protected
  - Datatypes can follow member variables, but are optional
    - For example: - `cashAmount : double`



Related classes

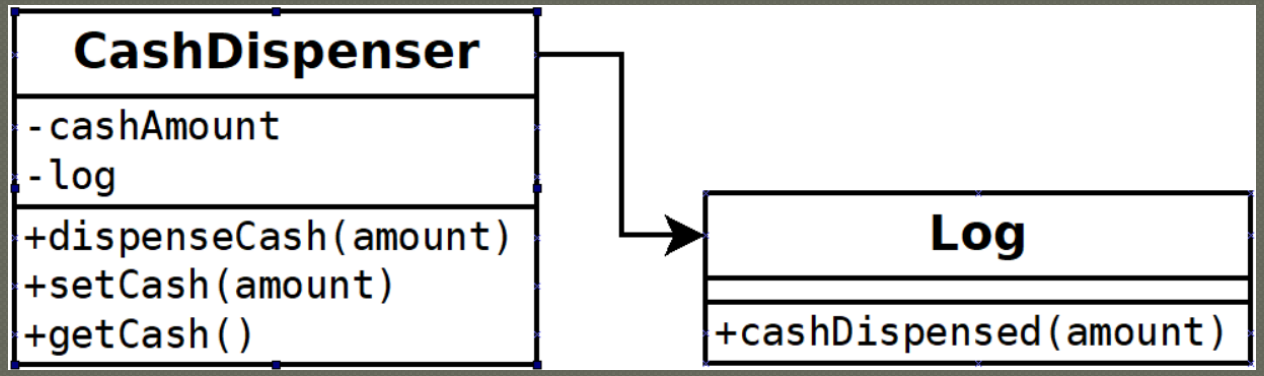
- What if we have a separate class that logs activity of a class



- We need to capture the relationship between the two classes

Association Relationship

- Association (aka delegation) is the most common connection between classes
  - Represents one class object storing a reference to another



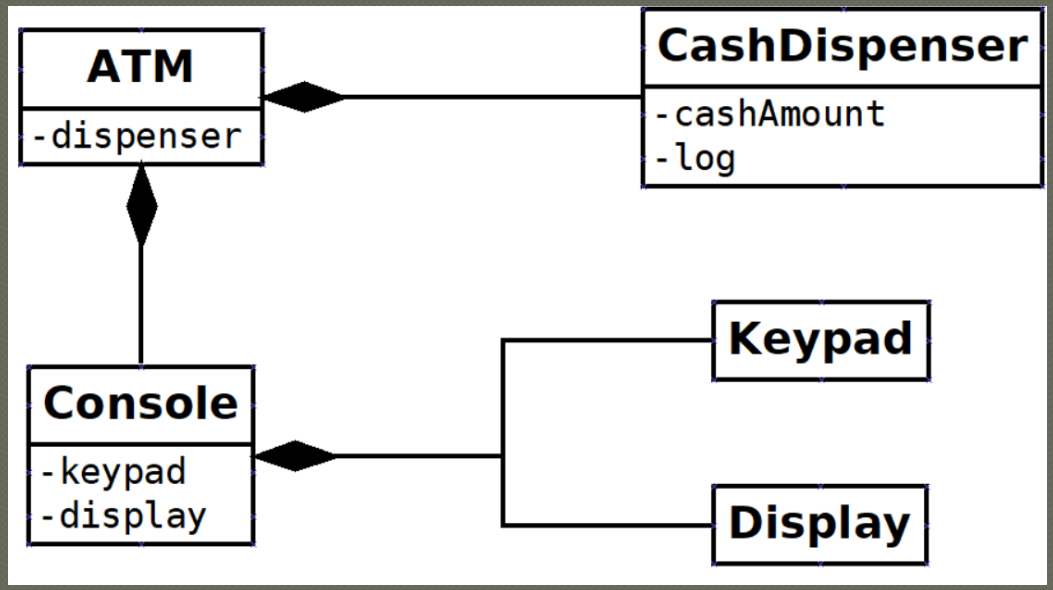
  - Represented by a solid line and a filled arrow
    - The source of the relationship (line) uses the target of the relationship (arrow)
    - For example, a `CashDispenser` uses a `Log`

Composition Relationship

- When one class is "made" of another class, the relationship is called **composition**
- Association vs Composition
  - Composition is similar to association, but the relationship is stronger
  - If an object creates and "owns" an object, it is composed of that object
    - In other words, if object A cannot exist without object B, then object A is composed of object B
  - If an object is given to another object to use, they are merely associated (one delegates to the other)

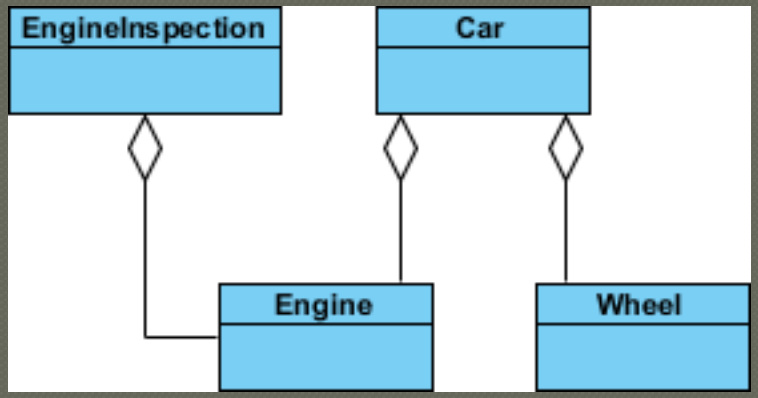
Composition Relationship

- Represented by a solid line and a filled diamond
  - The target of the relationship (diamond) is composed of the source (line) of the relationship
  - For example, an `ATM` is composed of a `CashDispenser` and a `Console`; a `Console` is composed of a `Keypad` and a `Display`



Aggregation Relationship

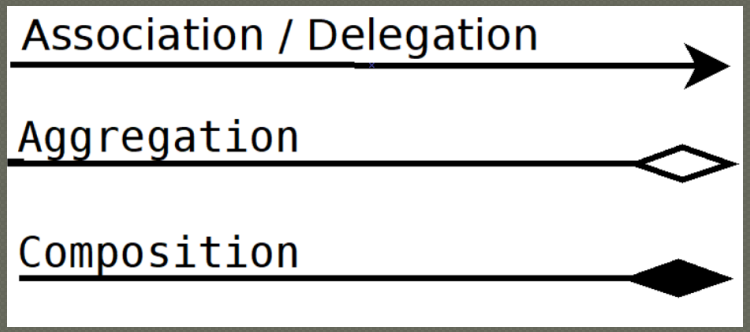
- Similar to composition, but not as strong
- Aggregation implies a relationship where the child can exist independently of the parent
  - For example: Class (parent) and Student (child); delete the Class and the Students still exist
- Represented by a solid line and an unfilled diamond
  - The target of the relationship (diamond) is an aggregation of the source (line) of the relationship
  - For example, a `Car` contains an `Engine` and a `Wheel`; `EngineInspection` contains an `Engine`



https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition/

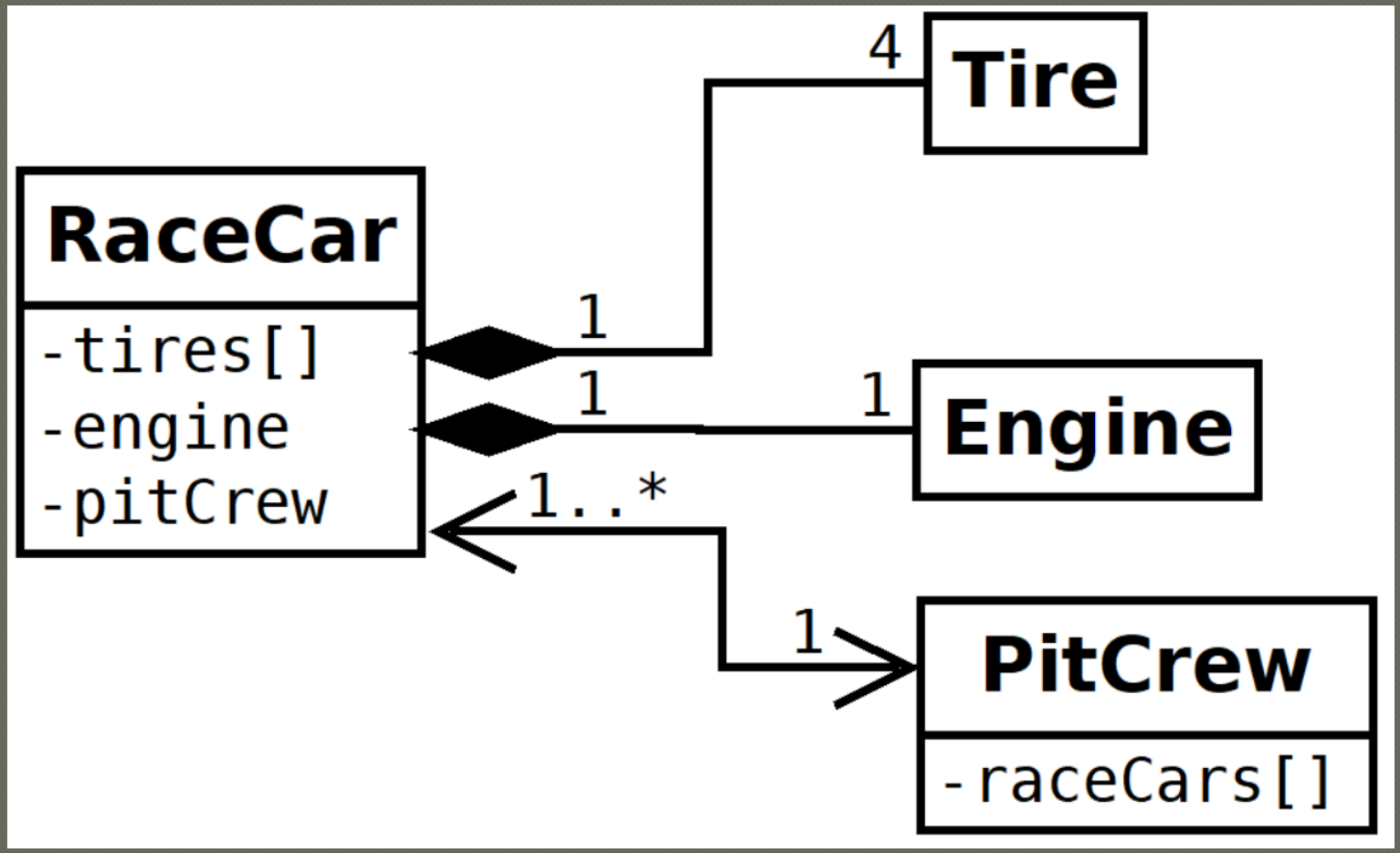
Association vs Aggregation vs Composition

- Association
  - a very generic term used to represent when one class used the functionalities provided by another class
- Composition
  - one parent class object owns another child class object and that child class object cannot meaningfully exist without the parent class object
- Aggregation
  - if it can meaningfully exist without the parent class object



https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition/

UML Cardinality/Multiplicity

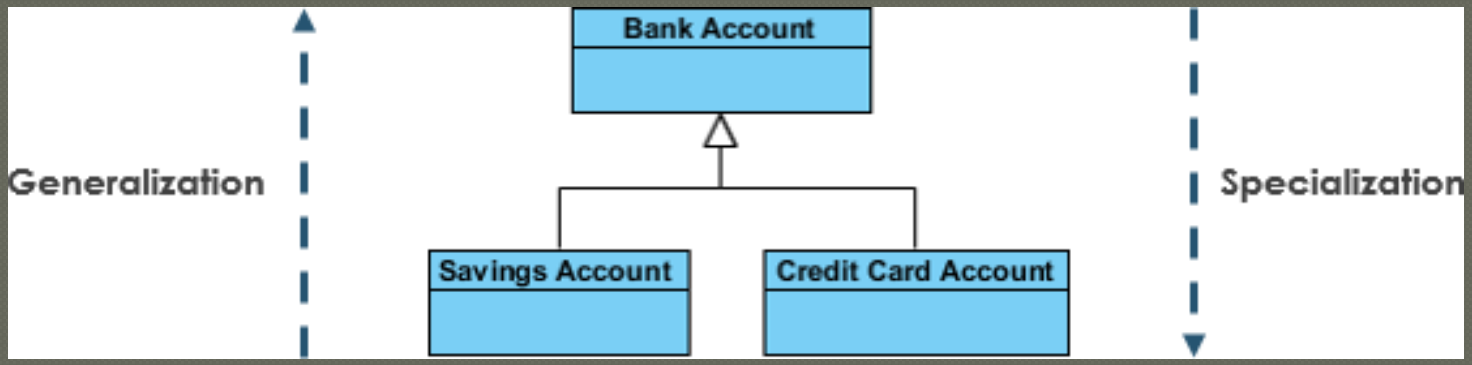


Multiplicity

- Number means how many instances of the class are contained/delegated to
- The ".." establish a range:
  - 2..5 means at least two, but up to five
  - 1..* means at least one, but any number more
  - 0..* or just * means any number, including none

Class Inheritance

- The **is-a** relationship between classes
- Represented by a solid line and an unfilled closed arrow
  - The source of the relationship (line) is-a source (arrow) of the relationship
  - For example, a `Savings Account` is a `Bank Account`; a `Credit Card Account` is a `Bank Account`

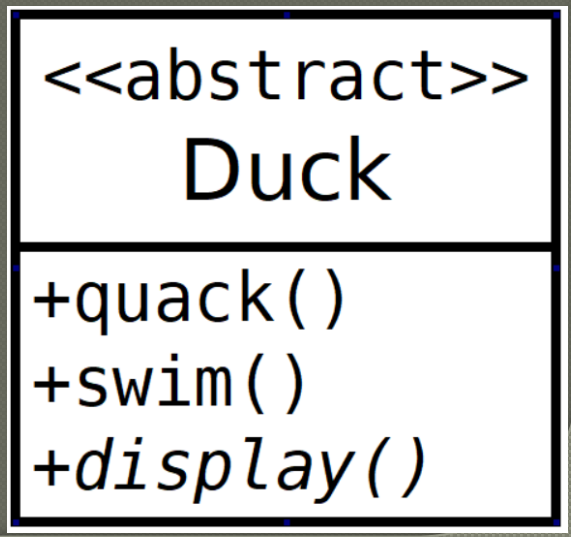


Abstract Classes

- Can be used, but not instantiated (constructed)
- Contain at least one *abstract* method that isn't "filled out" (no implementation)
- Purpose: allow reuse through shared implementation

UML: Abstract Classes

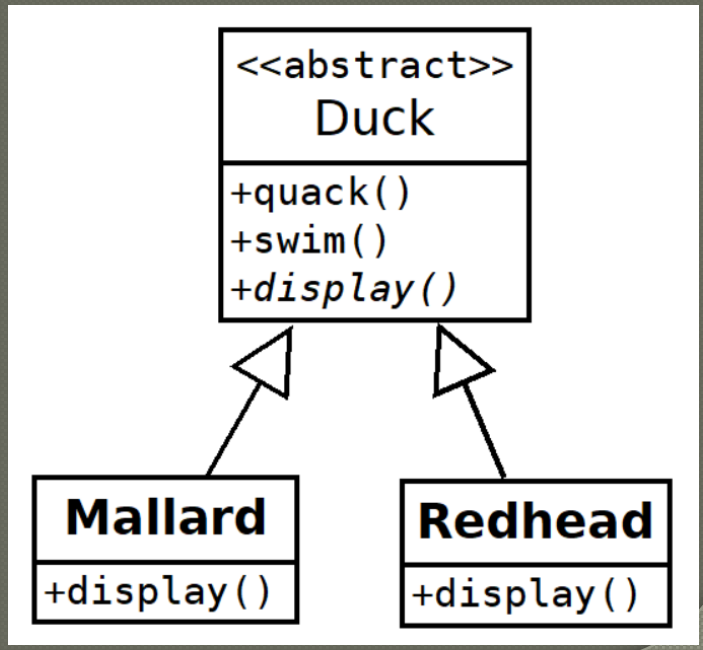
- Double angle brackets: Stereotypes
  - Stereotypes describe special properties
  - Can describe "types" of classes (abstract, view, controller, etc.)
  - Can be applied to methods and fields as well
- Method in *italics*
  - Abstract method



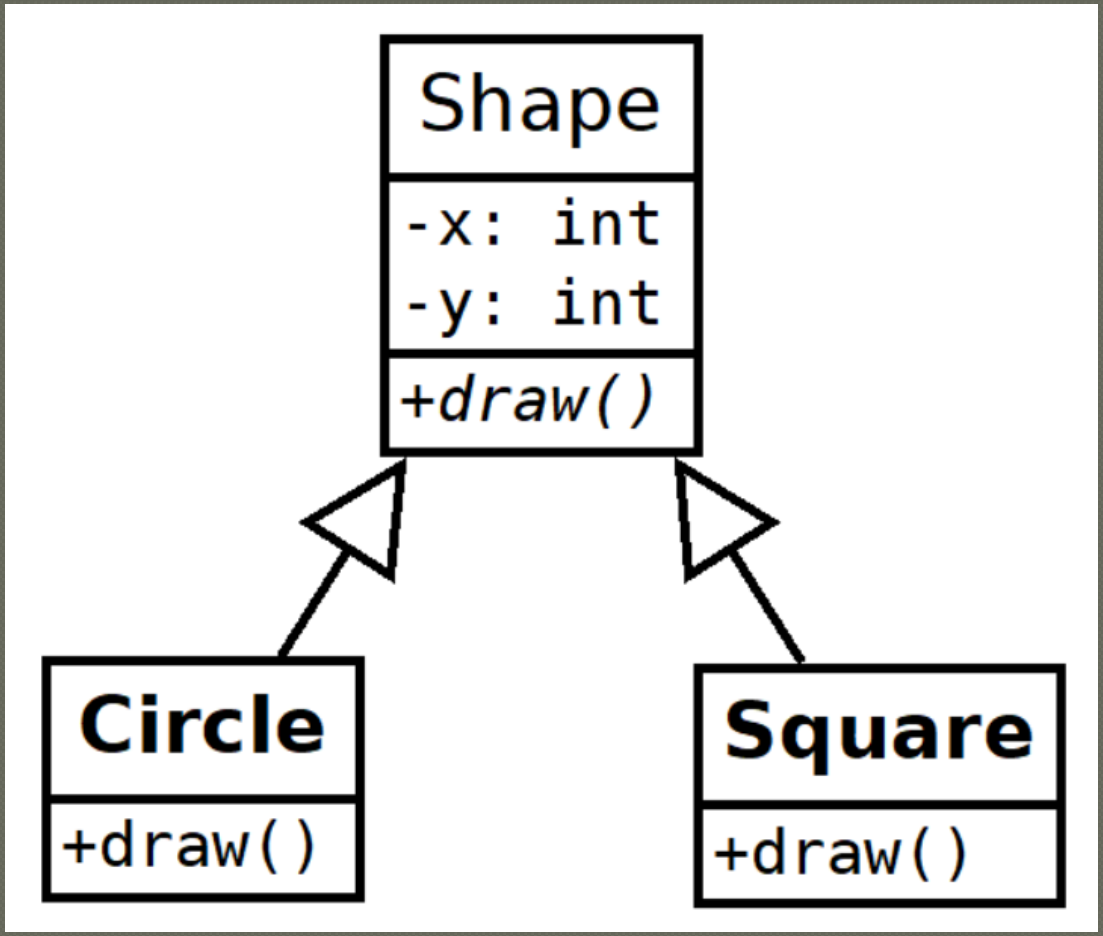
Example Abstract Class

```
public abstract class Duck {
    public void quack() { … }
    public void swim() { … }
    public abstract void display();
}

public class Mallard extends Duck {
    public void display { … }
}
```

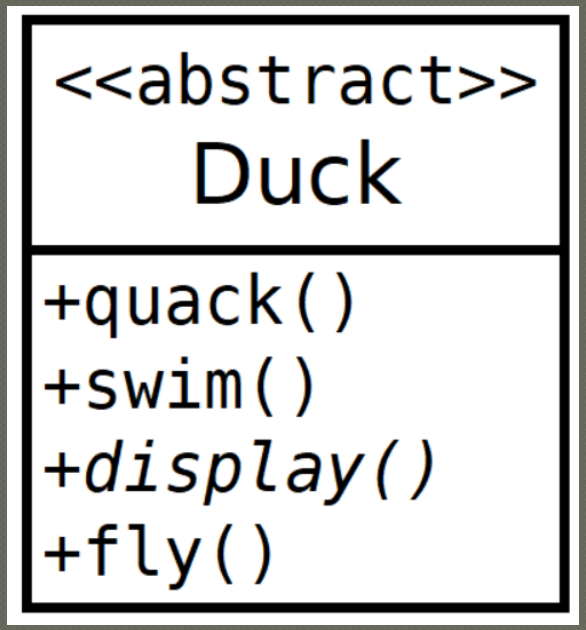


Another abstract class example



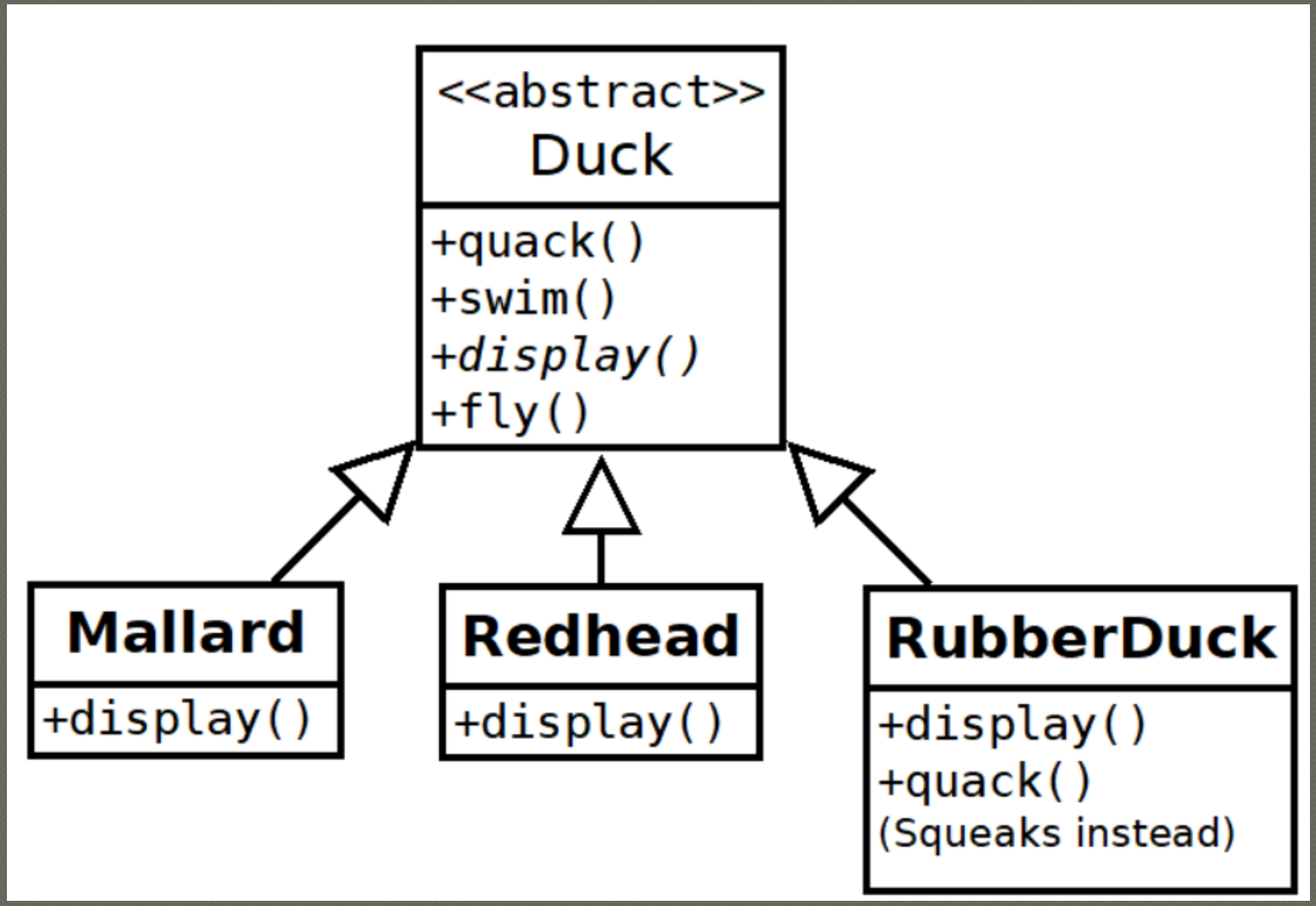
Class Design Change

- Classes can change, and often base classes can change too
- What if we want ducks to be able to fly?



- Mallards and redheads can fly too
- This makes sense
- What if there were other kinds of ducks?

The Rubber Ducky



Issues

- RubberDuck may exist in another code-base; it may have been developed in parallel
- Our child classes may have to change to "support" new semantics. (Violates Open/Closed)

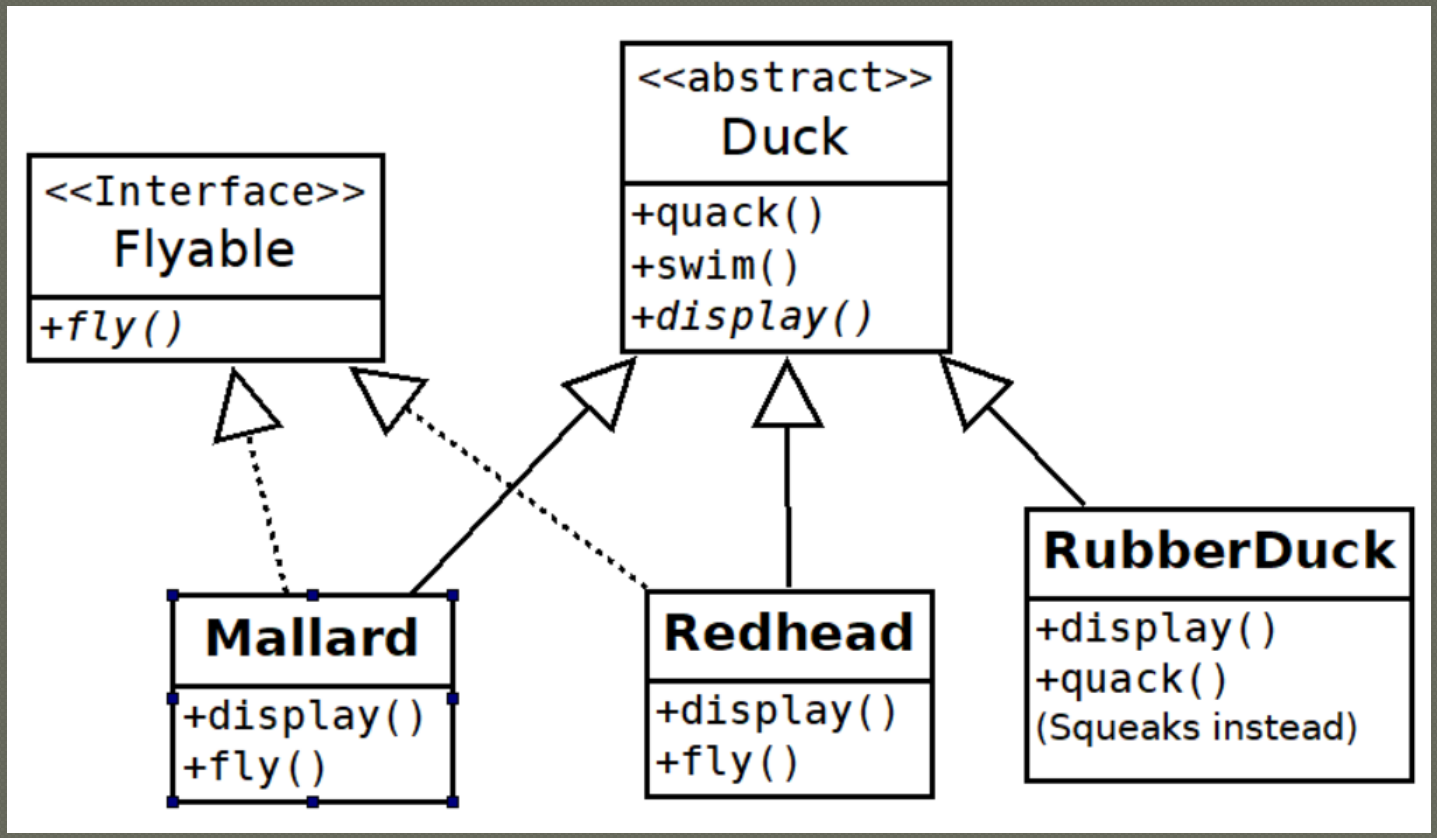
What are some solutions?

- Override `fly` to do nothing:
  - What happens if we add yet more kinds of ducks such as `WoodenDecoy`?
  - May violate basic expectations
- Add an interface?

Interfaces

- Interfaces are like abstract classes with all abstract methods
- A class can implement multiple interfaces
- "`Flyable`" can be an interface
  - Interface names often end in "able"

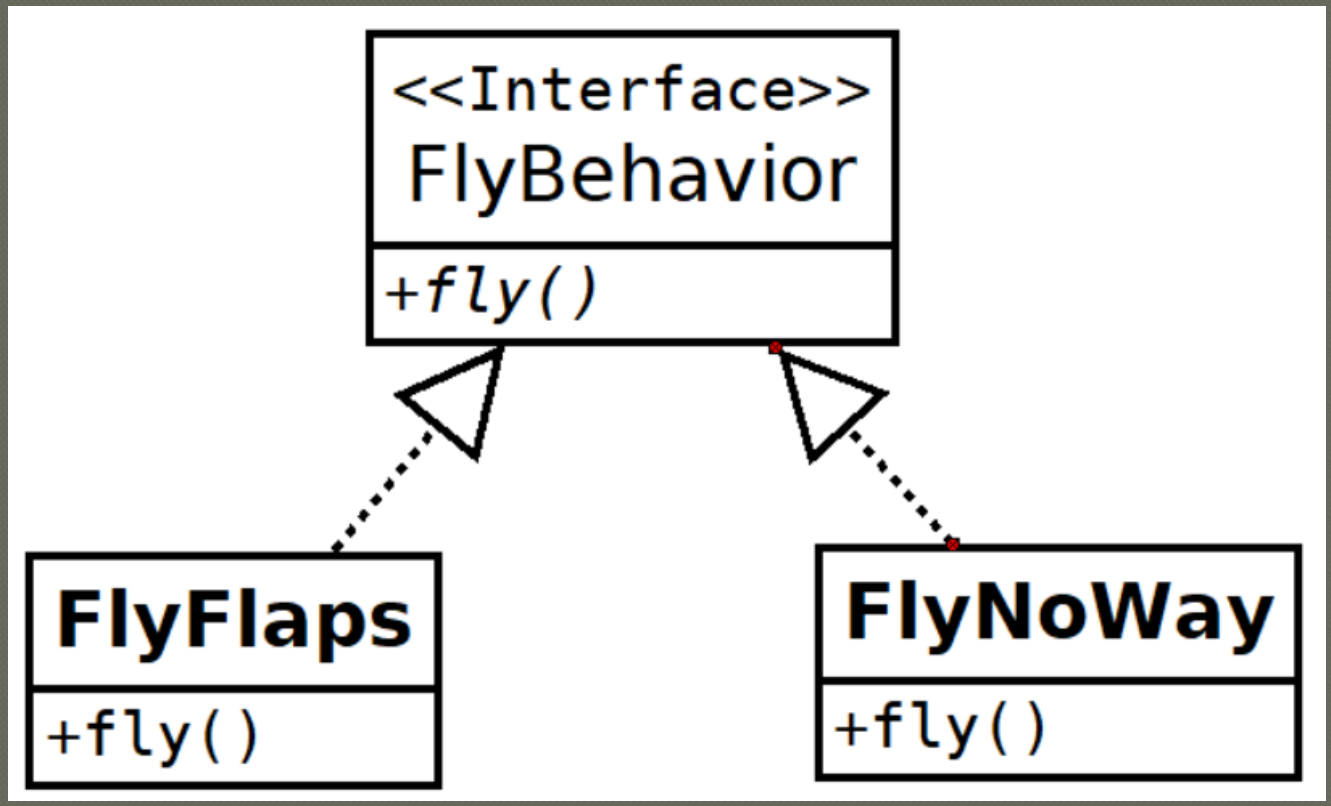
Flyable Interface



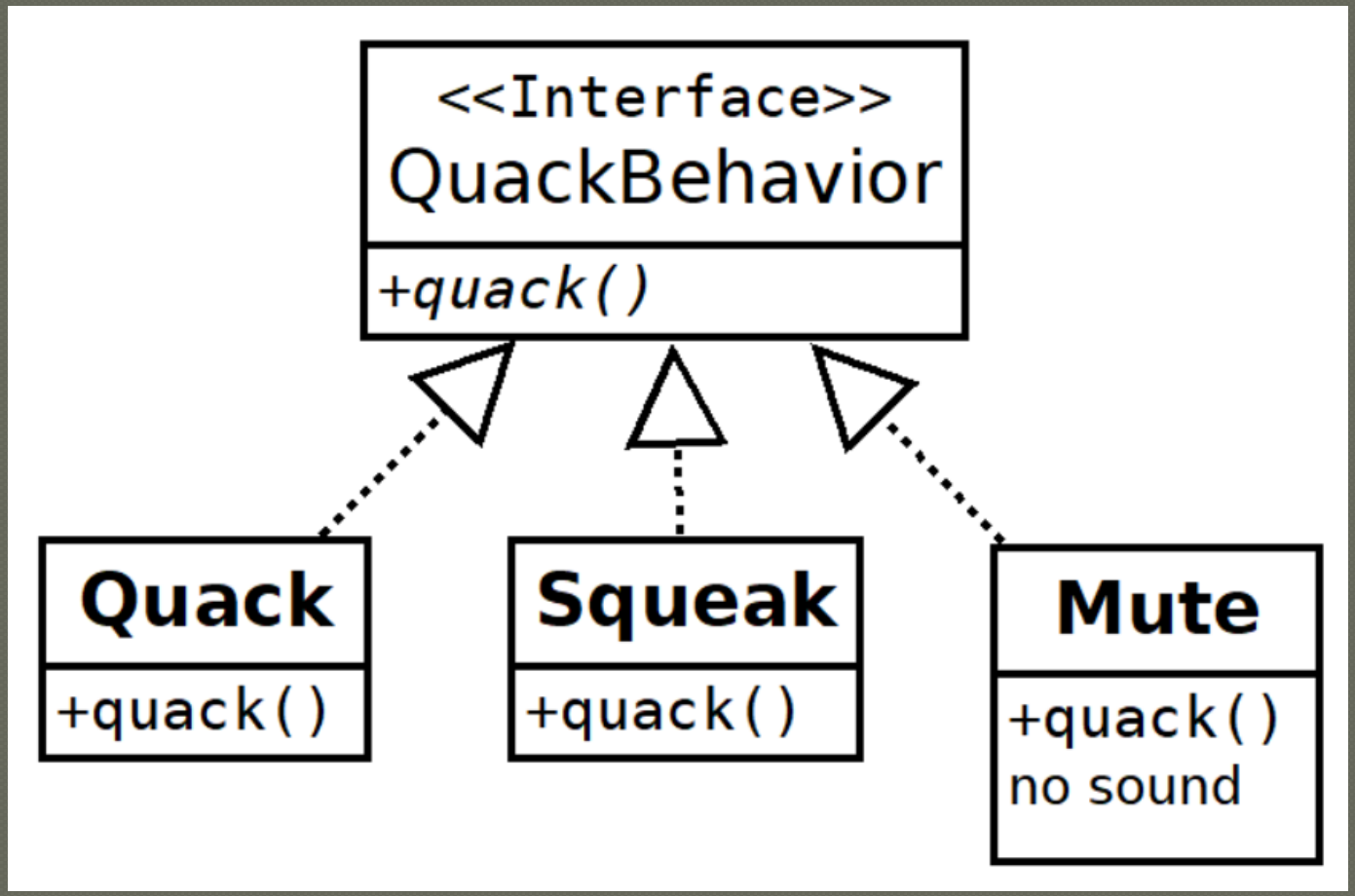
- What's wrong with this solution?
  - Violates once and only once

A Design Solution

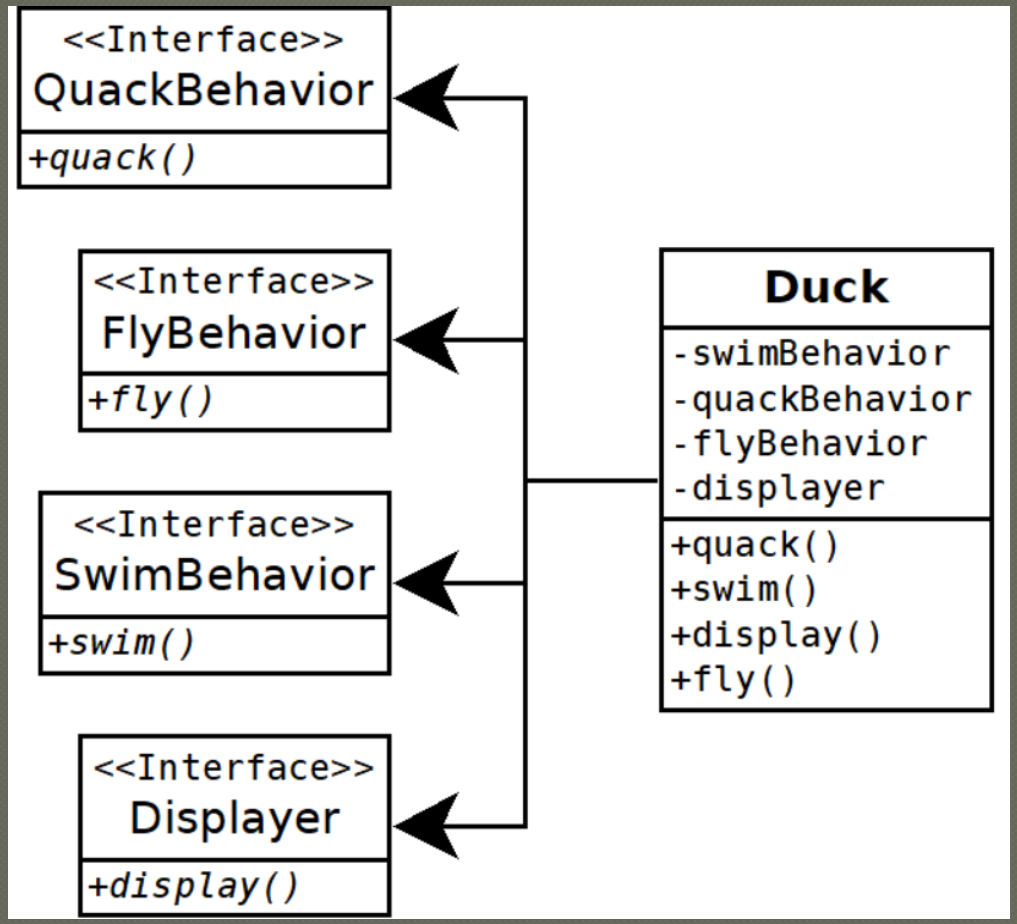
- Let's create classes of *behaviors* instead of *ducks*



Extending to Other Behaviors



Duck is Now a Container of Behaviors



- "Bag of components"
- Implications:
  - Easy to add new behaviors
  - Behaviors well specified
  - Scales extremely well
  - Random duck generation

The UML Class Arrows