CSC 3380 Aymond

Homework Assignment

- Design Patterns: Read Chapter 1 & the Strategy Pattern (pp. 315-323)
- Enterprise Architect Class Diagram, Due 11PM 3/20

Project News

- Next Milestone: #3
  - Due Tuesday 3/17, 11PM
  - Upload to Moodle (1 upload for entire team)
  - Guidelines on Moodle
  - BE SURE TO UPDATE SECTIONS FROM MILESTONE #1 & #2
  - All UML diagrams must be developed in EA

Section 1

3/9/2020

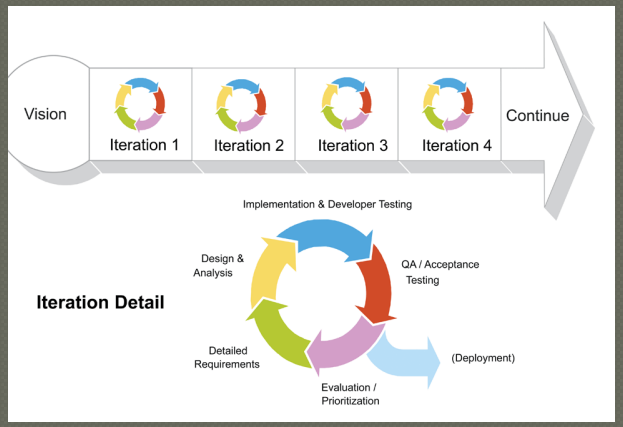
Project Milestones

- Milestone 1: Stories & Requirements (10%)
  - Tuesday 2/4, 11PM
- Milestone 2: Architecture Design (25%)
  - Due Friday 2/21, 11PM
  - During class mentor presentation, Monday 3/2
- **Milestone 3: Component Designs (25%)**
  - **Tuesday 3/17, 11PM**
  - **Work to have each team member take lead on at least one component design**
- Milestone 4: Working Prototype (25%)
  - Tuesday 4/21, 11PM
- Final in-class presentations (10%)
  - Wednesday 4/22, Monday 4/27, and Wednesday 4/29
- Project Post Mortem (5%)
  - Friday 5/1, 11PM

These milestones look suspiciously like the waterfall model

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 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

Component Designs

- For each component in your System Architecture, you will provide a section in your project portfolio that details the design of that component
- The Component Designs will include the following:
  - A short paragraph (2-3 sentences), describing the component's functional responsibilities.
  - Interfaces
    - A short description of each interface
    - Detailed specification of the interfaces.
      - The specification should provide sufficient information that another component that needs to interface with this component knows how to do so
      - To accomplish this, you should provide the set of formal parameter lists along with secondary storage requirements. For example, if a method in the interface provides the location of a data file as a parameter, then you need to include how the data in the file needs to be formatted.
    - External data sources
      - Identify any external data sources that the component will access
      - Provide the details on how the external data source is accessed (e.g., provide a link to the Spotify API)

Component Designs

- If the component is not at a level that it can be directly implemented without further design refinement
  - Create a component diagram for the component, to include subcomponents, interfaces, data flow, etc.; The component diagram and corresponding data flow diagram should be similar to the System Architecture, but at the component level
  - Continue the top-down design process, drilling down until all components in the set of component diagrams are at a level that can be directly implemented without further design refinement
  - Note: Once this step is complete, team members should be able to grad a component design and begin working on implementing it

Control Flow

- Capture the control flow of the system architecture and the top-tier of components in your system
  - You do not need to capture all control flow of the sub-components of your system
- Control flow can be captured using
  - Algorithms
  - Pseudocode
  - flow charts
  - UML control flow diagrams
- Different methods may be better specified using different techniques, so you do not have to pick one method for use by all components

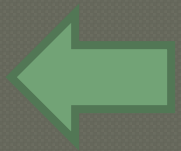
Milestone 3: Component Designs

- Source Code
  - eapx file(s) of System Architecture
  - eapx files of all component designs
  - Zip of all source code implemented at this point
- Due 11PM 3/17
- 25% of project grade

MAJOR PROJECT ANNOUNCEMENT

- New Requirements
  - You system must interface with a mainstream social network framework (e.g., Facebook, Instagram, twitter, etc.)
    - For example, tweet what you're listening to
  - You must incorporate at least 2 of the design patterns that we cover in class into your project design
    - Class diagrams must be included for those components that include a design pattern

Design Patterns



- **The Strategy Pattern**
- The Factory Method
- Generics
- The Abstract Factory Pattern
- The State Pattern
- The Observer Pattern
- The Adapter Pattern
- The Composite Pattern
- The Iterator Pattern
- The Builder Pattern
- Fallen Patterns
  - The Singleton Pattern
  - The Visitor Pattern

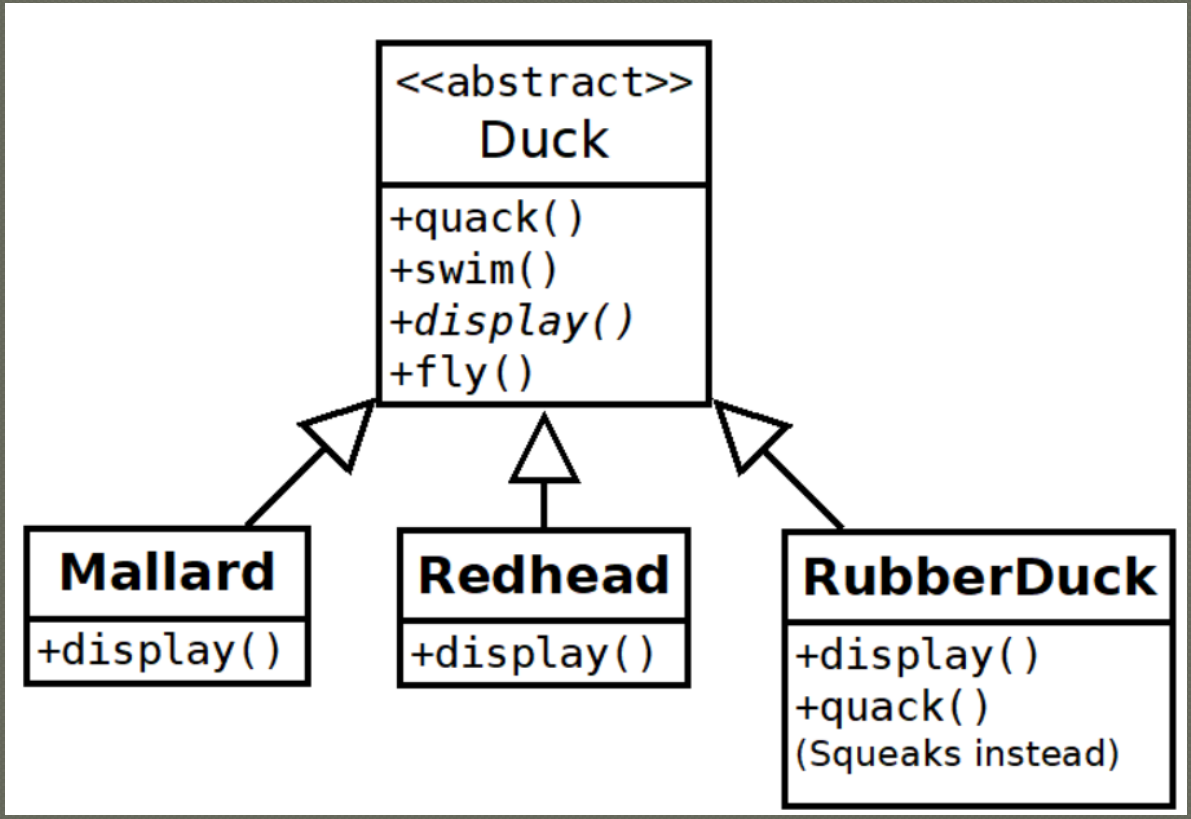
Design Strategies

- Top-down
  - Functional decomposition
  - Stepwise refinement at the component level
- Bottom up
  - Composition
  - Design pieces in isolation before deciding how they will fit together as a whole
- Stylized
  - Pattern (re)use
  - Good solution already exists

- There is a place for all of these strategies in software and software system design
  - Start with top-down architecture design
  - Components are handed off to development team for bottom-up software design
  - Patterns are reused at both the architecture and software design levels, where appropriate

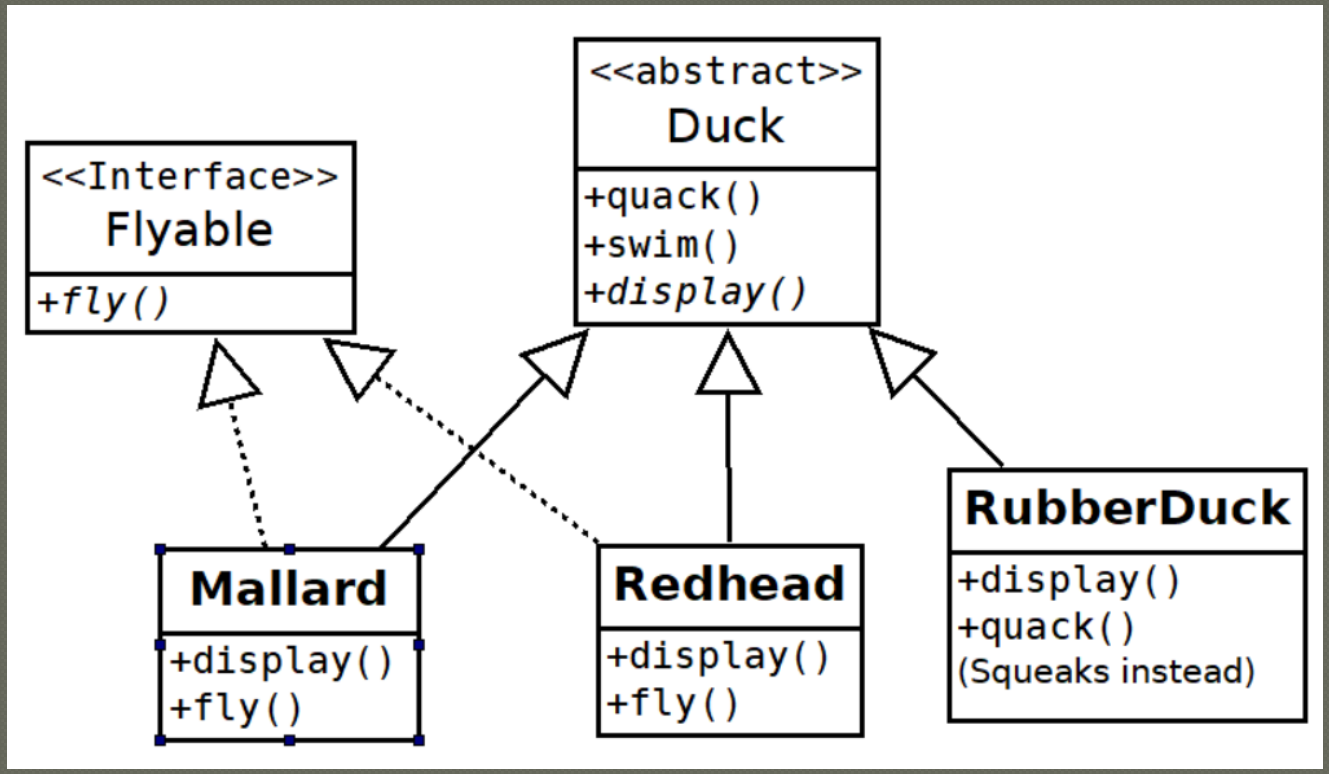
The Duck Class Revisited

- Remember our flawed duck hierarchy?



The Improved Duck Class

- We improved the hierarchy, but it is still not good

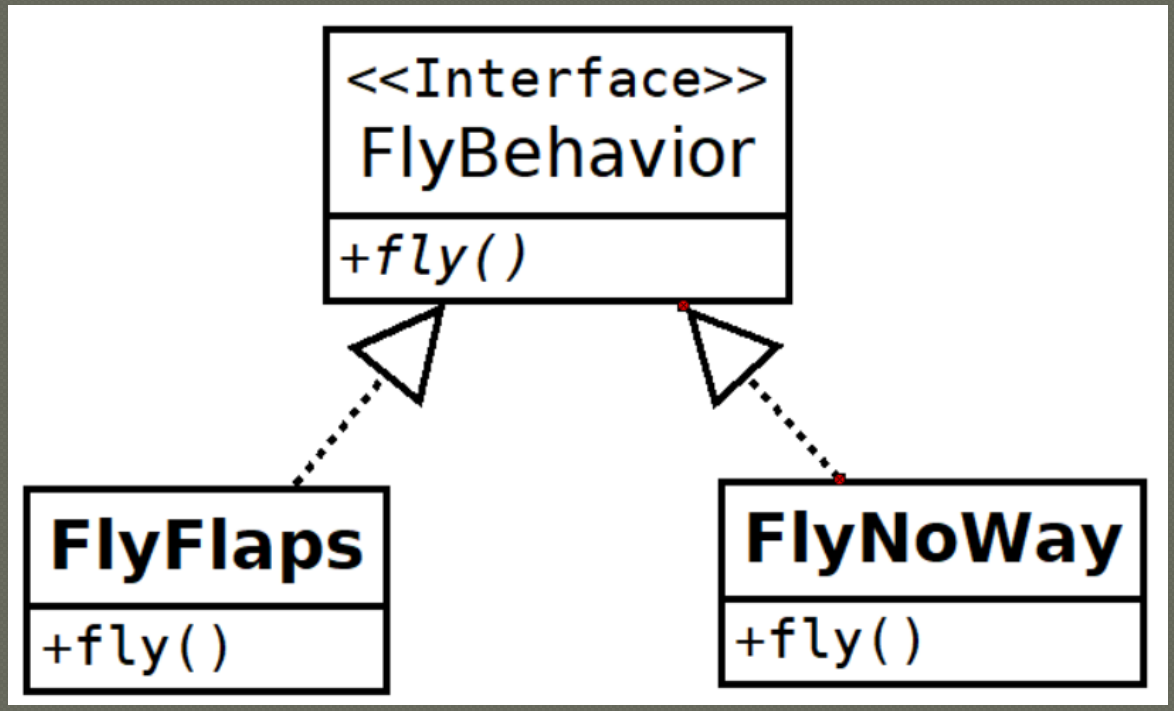


Characteristics of the solution

- When we add a new behavior/property:
  - We don't want to add a new layer to the class hierarchy
  - We don't want to require multiple-inheritance
  - We don't want to break existing classes
- When we make use of a behavior/property:
  - We don't want to manually delegate to a particular behavior (although this is not a big deal)
  - We want to be able to easily create new combinations

SimUDuck Design Solution

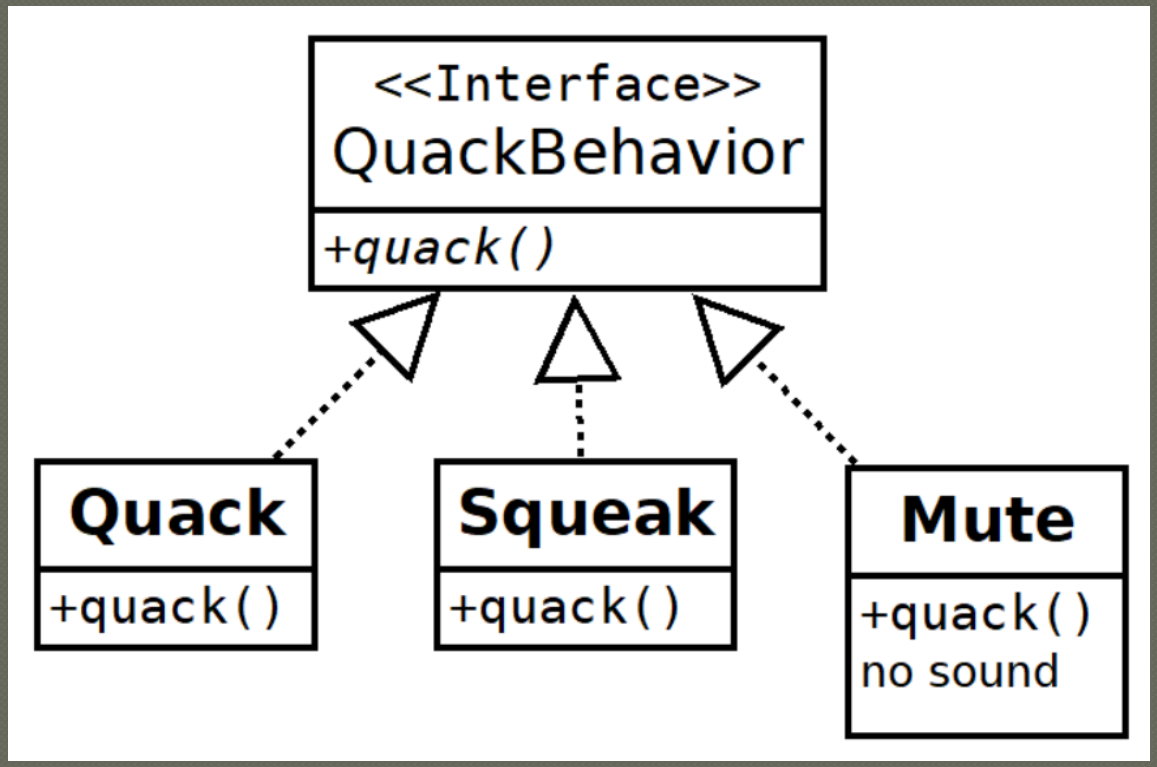
- We create classes of behaviors instead of ducks



Is-a Can Hurt Flexibility

- Sometimes we want to multiply behaviors:
  - What if we wanted a hybrid duck that could squeak or quack?
- Sometimes we don't want to expose an interface:
  - What if we wanted a duck that could squeak, but only after quacking
  - Clients shouldn't call squeak directly

Extending to Other Behaviors

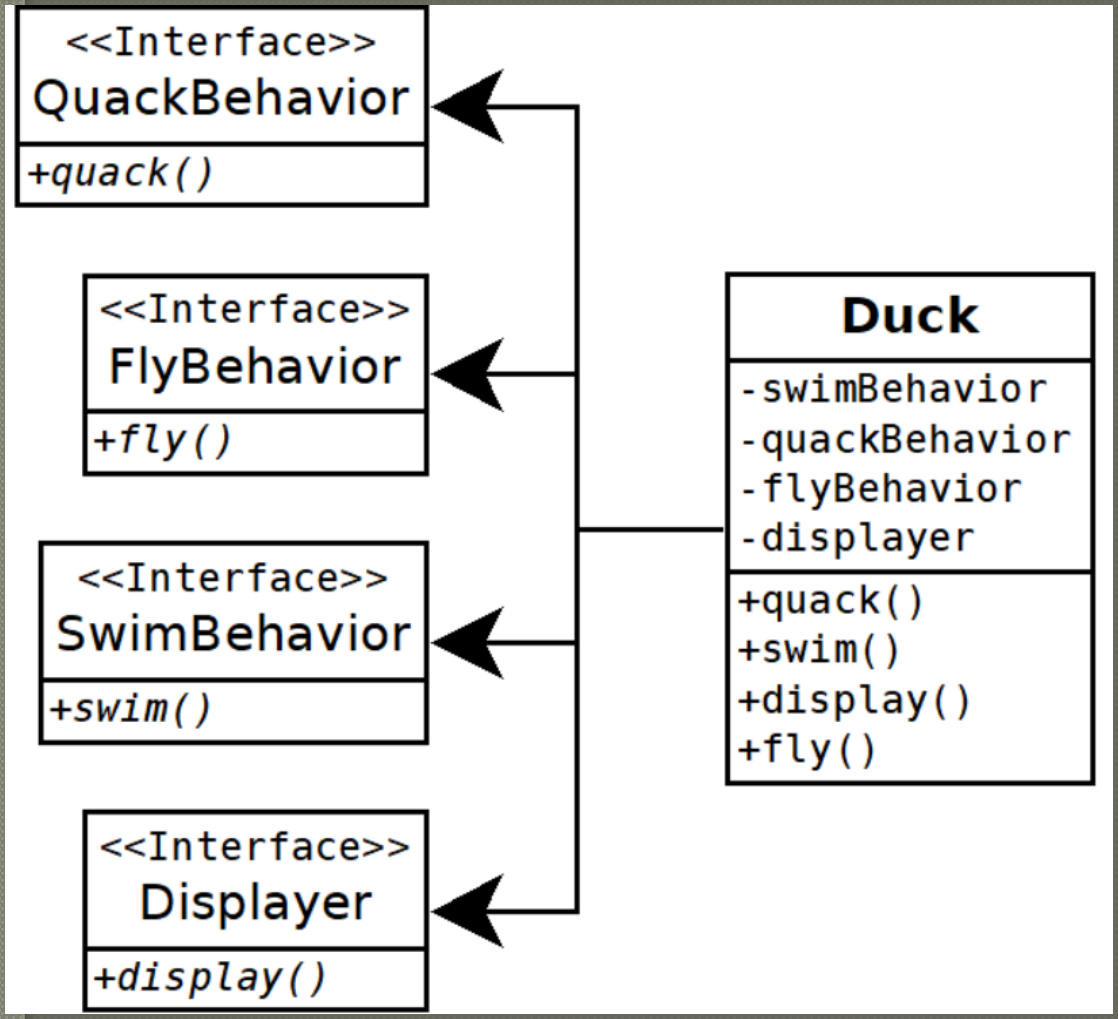


- Any kind of action can now be specified.

Guideline

- Separate what changes from what stays the same
  - If something stays the same, you won't break it
  - Keeping change limited reduces the amount of analysis

Container of 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

Creating a Duck

```
//constructor takes behaviors
public class Duck {
    FlyBehavior bFly;
    QuackBehavior bQuack;
    SwimBehavior bSwim;
    Displayer displayer;

    public Duck(
        FlyBehavior b_fly,
        QuackBehavior b_quack,
        SwimBehavior b_swim,
        Displayer displayer ) {
        bFly = b_fly;
        bQuack = b_quack;
        bSwim = b_swim;
        this.displayer = displayer;
    }
}
```

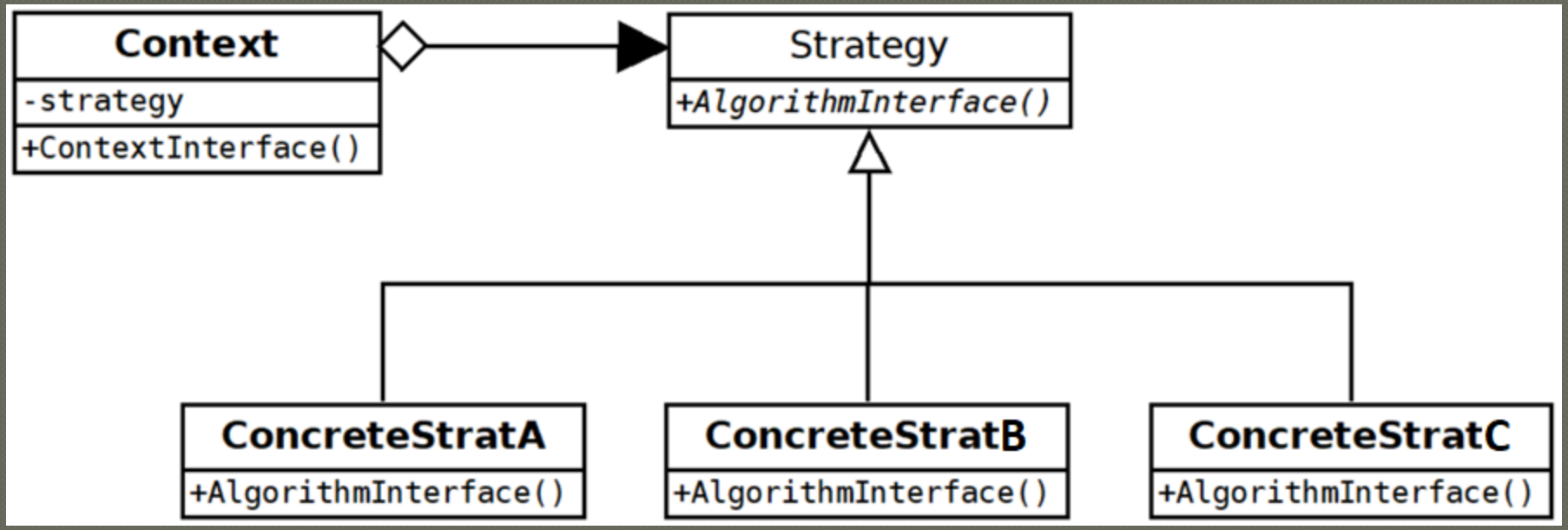
Creating a Duck

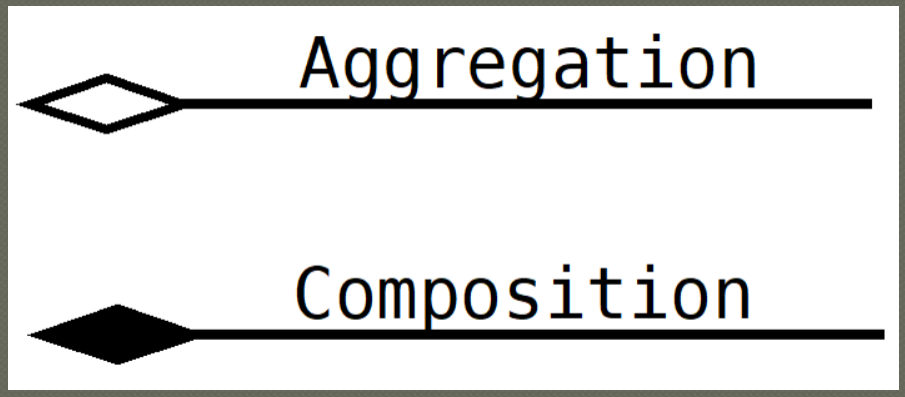
```
public static void main( String[] args ) {
    Duck mr_duckles = new Duck(
        new FlyFlaps(…),
        new Quack(…),
        new FloatSwimming(…),
        new MallardRenderer(…) );

    simulation.simulateDuck( mr_duckles );
}
```

The Strategy Pattern

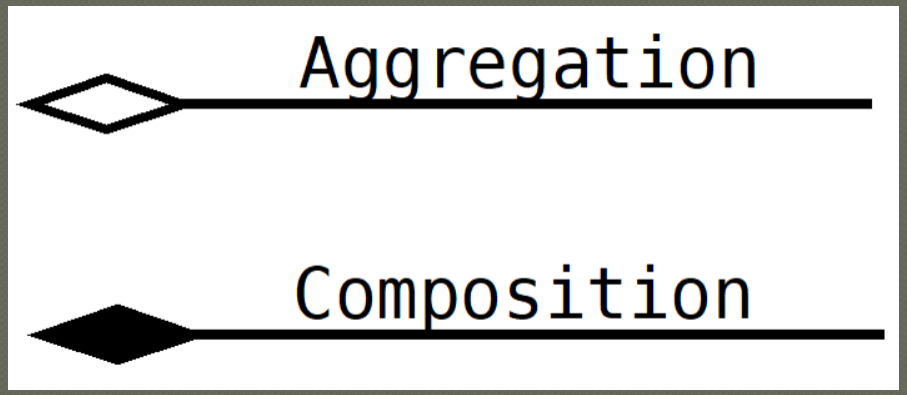
- Our duck solution was a *design pattern* called **strategy**
- The strategy pattern is **instantiated** when behavior or algorithms are encapsulated into classes
- The concrete implementations inherit from a generic strategy interface with the desired operation
- Also called policy

The Strategy Pattern



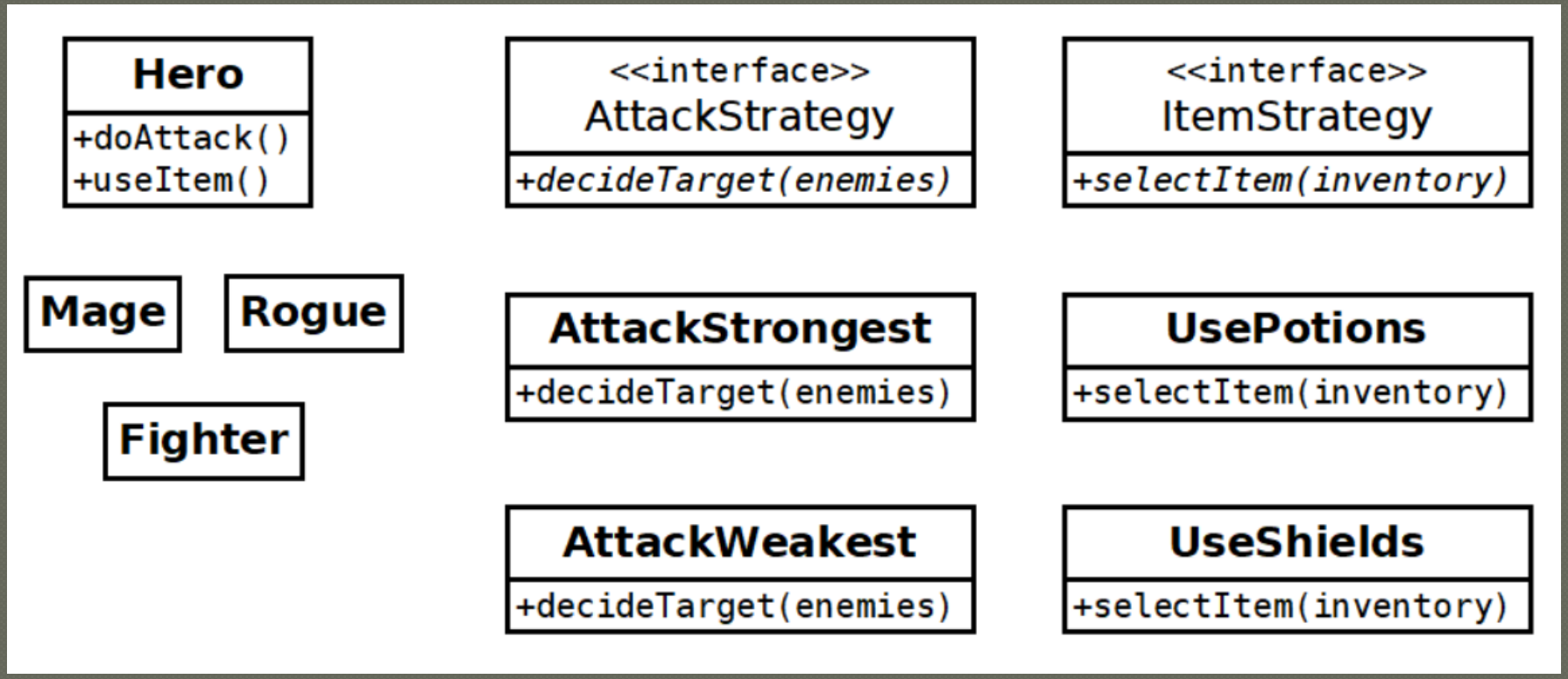


The Hollow Diamond

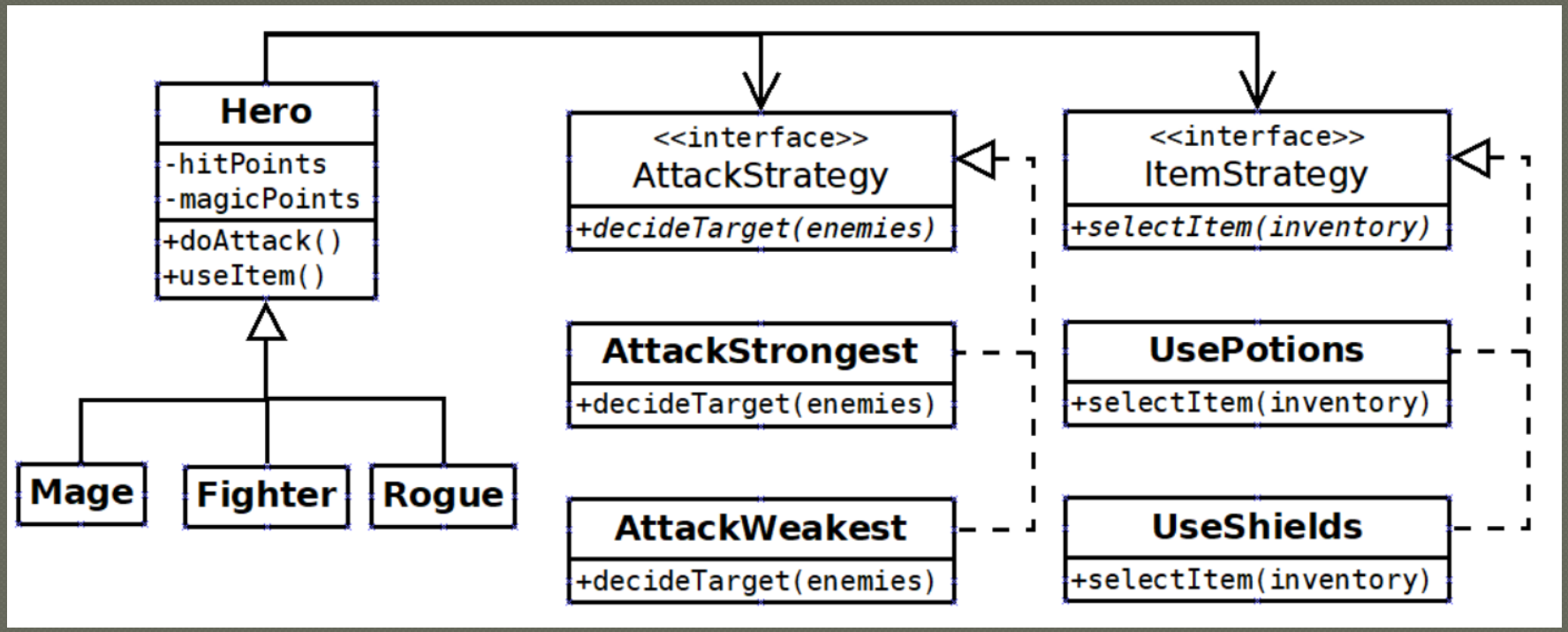


- Aggregation is like composition:
  - An object is made up of other objects
- But has an important difference:
  - An aggregating object does not own its members.
  - It's not responsible for creating/destroying them
  - (although it has the option)
- Fundamentally defined by destruction:
  - Composed objects die when their owner dies
  - Aggregated objects do not (their life is independent)
  - If the contained object is not necessary, use pure delegation.

Strategy Puzzle: Add the Arrows



Puzzle Solution



Design Patterns

- Design patterns are generic organizations of classes and arrows that can be applied to many different problems
- Design patterns provide a shared vocabulary
- For every pattern, know the following:
  - Participants (the classes)
  - Collaborations (the arrows and code)
  - Applicability (when to use the pattern)
  - Consequences
  - Implementation

Applicability of Strategy

1

Many related classes differ only in their behavior

- Kinds of ducks
- RPG characters
- Machine Learning algorithms

2

You need a large number of variations of algorithms:

- AI algorithms
- Responses to user input
- Rendering algorithms

3

Algorithm uses data that client doesn't need to know about:

- If your algorithm uses a mesh, AI search tree, etc., the client remains ignorant. You can change it later without consequence.

4

To replace behavioral conditional statements in a class.

Consequences of Strategy

Define a family of algorithms/behaviors	Do not need to use inheritance	No need for conditional statements
Clients need to know strategies (although default strategies can be used for when they don't matter)	Concrete strategies need to be given information through interface: can cause too much or too little sharing	Adds more objects, but doesn't add performance issues (inheritance adds similar overhead)

Implementation of Strategy

```
interface Strategy {
    RetType algorithmInterface(…);
}

class ConcreteStratA implements Strategy {
    RetType algorithmInterface(…) {
        //do the algorithm
    }
}

class ConcreteStratB implements Strategy {
    RetType algorithmInterface(…) {
        //do the algorithm
    }
}
```

Implementation of Class Using Strategy

```
class Context {
  private Strategy strat;

  Context( Strategy strat ) {
    this.strat = strat;
  }

  RetType contextInterface(…) {
    return strat.algorithmInterface(…);
  }
}
```

Has-a vs Is-a

- When Mallard inherits from Duck, we say that mallard **is a** duck
- When Duck contains FlyBehavior, we say that Duck **has a** FlyBehavior
- If we can use **has-a** instead of **is-a**, we should. Why?

Design Patterns are ***not***

Libraries

Frameworks

Toolkits

These things make *use* of patterns.

Guideline

Program to an interface, not an implementation

Depend on an interface where practical (instead of class)

Allows classes to be swapped out later

Class Mission

Find a way to apply strategy to your project

You don't have to actually use it, just find a potential application

How can "Context" and "Strategy" be defined in your project?