# Strategy Pattern

Idaho State University | Computer Science

Isaac Griffith

CS 2263
Department of Informatics and Computer Science
Idaho State University

ROAR

## Outline

After today's lecture you will be able to:

- Describe how patterns fit into software engineering
- Understand how to select and use design patterns
- Understand the use of the Strategy Design Pattern
- Use and implement the Strategy Pattern

# Inspiration

“It’s hard to read through a book on the principles of magic without glancing at the cover periodically to make sure it isn’t a book on software design.” – Bruce Tognazzini

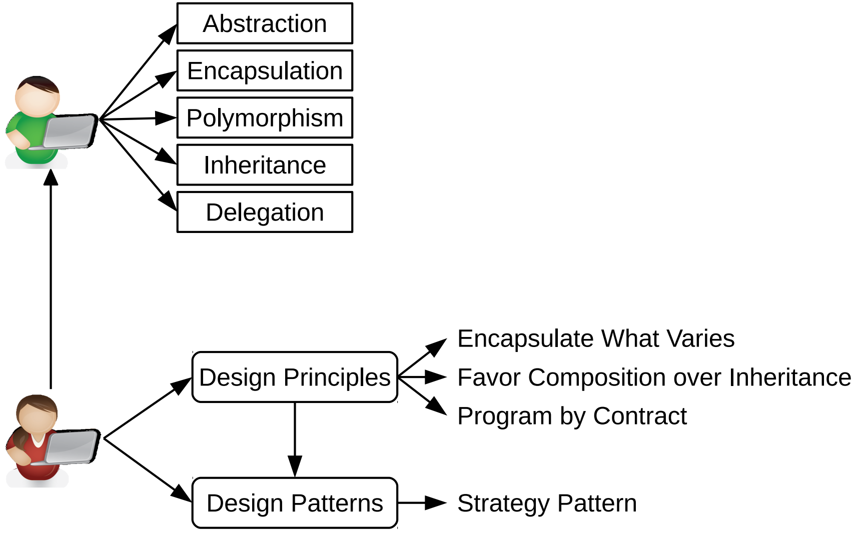
Idaho State University Computer Science

## What is a Pattern

- A design pattern describes a problem that frequently occurs in software design and implementation, and then describes a generalized solution to that problem in such a way that the solution can be reused.
- Patterns are also a means to document good design practices and principles, thus they serve to preserve existing expert knowledge.

ROAR

# The OO Toolbox



# Roles of Patterns in Software Development

- Promise reuse benefits early in the development lifecycle.
- Why?
  - Reduce development effort
  - Assure higher software quality
  - Provide a common design vocabulary
- Issue:
  - Designing applications by systematically deploying design patterns is not a trivial task.

## Pattern-Oriented Design Challenges

- What qualifies a pattern as a design component?
- Can we compose applications solely from design patterns?
- How can we systematically develop applications using design patterns?

## Design Patterns in the Software Lifecycle

- Reuse
- Level of reuse
- Development Phase Classification:
  - Analysis Patterns
  - Architectural Patterns
  - Design Patterns
  - Idioms

## A Brief History of Design Patterns

- Came from the work of architect Christopher Alexander (a professor at UC Berkeley) who spent 20 years applying patterns to civil architecture projects
- Kent Beck and Ward Cunningham later applied these concepts to Software
  - Kent -> Idioms of Smalltalk
  - Ward -> Business Systems

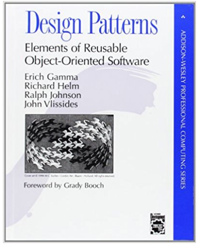
# A Brief History of Design Patterns

- In 1991, Erich Gamma produced his PhD thesis containing the first documented design patterns. It received little attention (as many PhD theses do)
- In 1995 Gamma, Helm, Johnson and Vlissides wrote “Design Patterns: Elements of Object-Oriented Software” -> now known as the Gang of Four book. Which cataloged 21 design patterns.
- This work spanned several other catalogs of patterns, including:
  - POSA: Pattern-Oriented Software Architecture (a series of 7+ books)
  - PLoPD: Pattern Languages of Program Design (a workshop series which culminated in a series of 4 books)

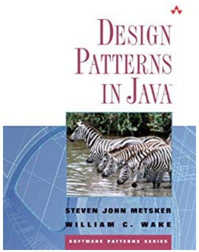
\end{frame}

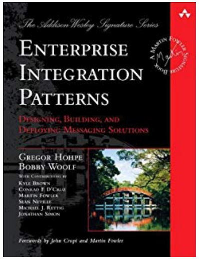
# A Brief History of Design Patterns

- Other works and pattern catalogs have been published including works to further or better describe the use of patterns.
- More recently, Joshua Kerievsky published a work "Refactoring to Patterns" which connects the concepts of refactoring (latter this semester) and patterns together.
  - This work describes how to transform existing software using patterns as a guide.


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Addison-Wesley Professional Ruby Series
DESIGN PATTERNS IN RUBY
Foreword by Obie Fernandez
Professional Ruby Series Editor
RUSS OLSEN




The Addison-Wesley Signature Series
ENTERPRISE INTEGRATION PATTERNS
DESIGNING, BUILDING, AND DEPLOYING MESSAGING SOLUTIONS
GREGOR HOHPE
BOBBY WOOLF
Forewords by John Crupi and Martin Fowler

WILEY SERIES IN SOFTWARE DESIGN PATTERNS
PATTERN-ORIENTED SOFTWARE ARCHITECTURE
On Patterns and Pattern Languages
Volume 5
Frank Buschmann
Kevlin Henney
Douglas C. Schmidt

## Pattern Descriptions

- Intent
- AKA
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns

## Selecting A Pattern

- Consider how design patterns solve design problems
- Scan the Intent sections of patterns
- Study how patterns interrelate
- Examine causes of redesign
- Consider what should be variable in your design

## Pattern Example: Strategy Pattern

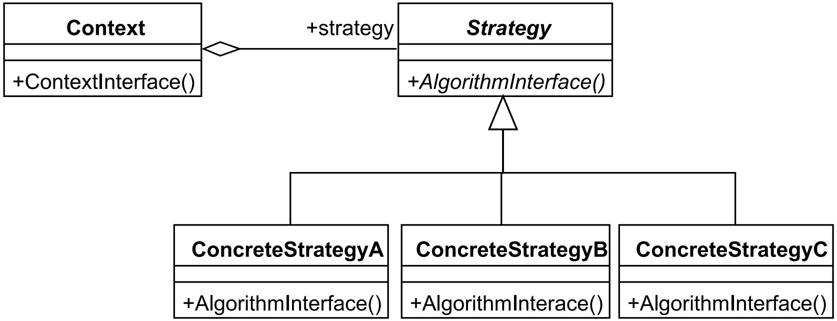
**Intent:**

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

**Applicability:**

- related classes differ only in their behavior
- you need different variants of an algorithm
- algorithms use data clients shouldn't know about
- a class defines many behaviors, and these appear as multiple conditional statements in its operations.

# Pattern Example: Strategy Pattern

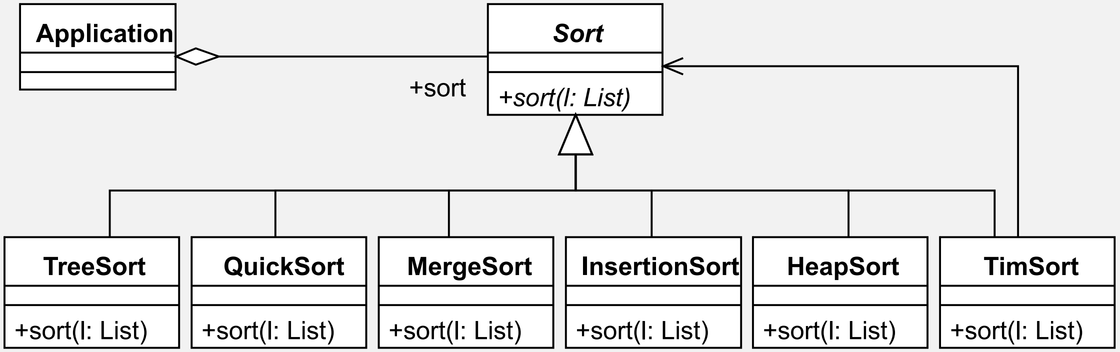


**Structure:**

Idaho State University Computer Science

# Pattern Example: Strategy Pattern

## Example: Sorting



ROAR

# Causes of Redesign

- Creating an object by specifying a class explicitly
- Dependence on specific operations
- Dependence on hardware and software platform
- Algorithmic dependencies
- Tight coupling
- Extending functionality by subclassing
- Inability to alter classes conveniently

Idaho State University
Computer Science

# Guidelines on Design Pattern Use

- Read the pattern once through for an overview
- Go back and study the Structure, Participants, and Collaborations
- Look at the example code
- Choose names for pattern participants that are meaningful
- Define the classes
- Define application-specific names for operations in the patterns
- Implement the operations to carry out the responsibilities and collaborations in the pattern.

ROAR

## A Final Note on Pattern Oriented Design

- Design patterns should not be applied indiscriminately.
- They often achieve flexibility and variability by introducing additional levels of indirection and abstraction which can complicate design or cost you some performance.
- Look to the consequences section of the pattern to evaluate the benefits and liabilities of using the pattern.

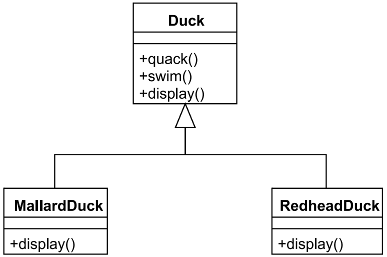
# Why Patterns?

- As the Design Guru said: “Remember, knowing concepts like **abstraction**, **inheritance**, and **polymorphism** does **not** make you a good OO designer. A design guru thinks about how to create **flexible designs** that are **maintainable** and that can **cope with change**.”
- Someone has already solved your problems.
  - Design patterns allow you to exploit the wisdom and lessons learned by other developers who’ve encountered design problems similar to the ones you are encountering
- The best way to use design patterns is to **load your brain with them** and then **recognize places** in your designs and existing applications where you can **apply them**
- Instead of **code reuse**, you get **experience reuse**

Idaho State University
Computer Science

# Design Patterns by Example

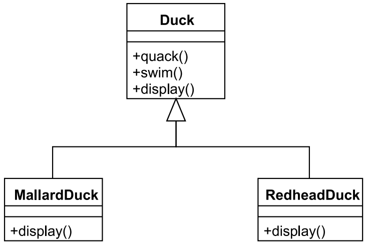
- SimUDuck: a “duck pond simulator” that can show a wide variety of duck species swimming and quacking

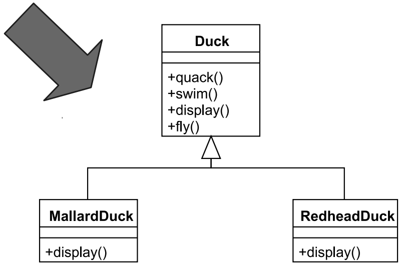


- But a request has arrived to allow ducks to also fly.

ROAR

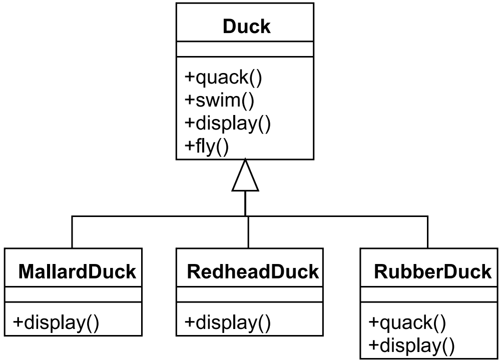
# Easy





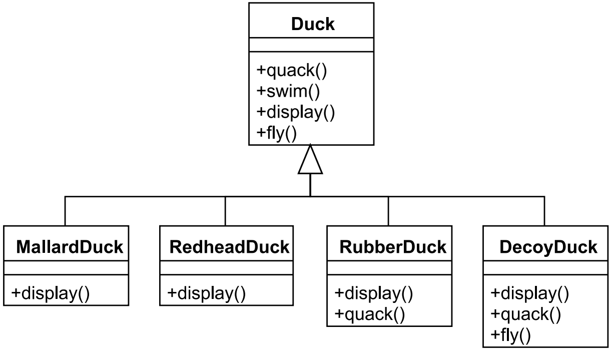
Code Reuse via Inheritance
Add `fly()` to `Duck`; all ducks can now fly

## Whoops!



- Rubber ducks do not fly! They don't quack either, so we had previously overridden `quack()` to make them squeak.
- We could override `fly()` in Rubber Duck to make it do nothing, but that's less than ideal, especially...

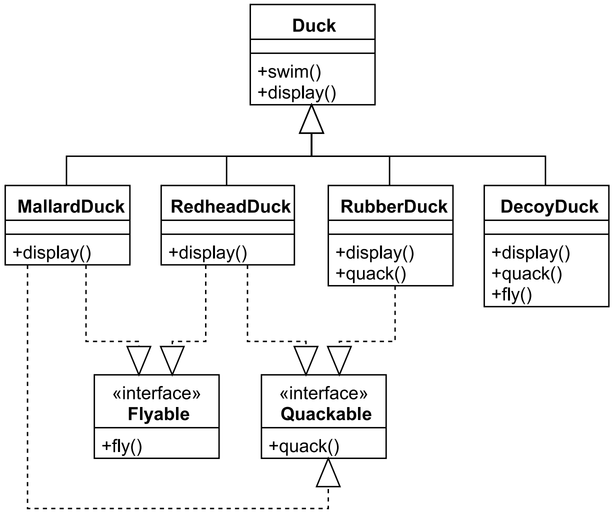
## Double Whoops!



...When we might always find other Duck subclasses that would have to do the same thing.

What was supposed to be a good instance of **reuse via inheritance** has turned into a **maintenance headache!**

# What about an Interface?



## Design Trade-offs

- With inheritance, we get
  - code reuse, only one `fly()` and `quack()` method vs. multiple (pro)
  - common behavior in root class, not so common after all (con)
- With interfaces, we get
  - specificity: only those subclasses that need a fly() method get it (pro)
  - no code re-use: since interfaces only define signatures (con)
- Use of abstract base class over an interface? Could do it, but only in languages that support multiple inheritance
  - In this approach, you implement `Flyable` and `Quackable` as abstract base classes and then have `Duck` subclasses use multiple inheritance
    - Trade-Offs?

## OO Principles to the Rescue!

- Encapsulate What Varies
  - For this particular problem, the “what varies” is the behaviors between Duck subclasses
    - We need to pull out behaviors that vary across subclasses and put them in their own classes (i.e. encapsulate them)
  - The result: fewer unintended consequences from code changes (such as when we added `fly()` to `Duck`) and more flexible code.

## Basic Idea

- Take any behavior that varies across Duck subclasses and pull them out of Duck
  - Duck’s will no longer have fly() and quack() methods directly
  - Create two sets of classes, one that implements fly behaviors and one that implements quack behaviors
- Code to an Interface
  - We’ll make use of the “code to an interface” principle and make sure that each member of the two sets implements a particular interface
    - For `QuackBehavior`, we’ll have Quack, Squeak, Silence
    - For `FlyBehavior`, we’ll have `FlyWithWings`, `CantFly`, `FlyWhenThrown`, ...
- Additional benefits
  - Other classes can gain access to these behaviors (if that makes sense) and we can add additional behaviors without impacting other classes

## “Code to Interface”

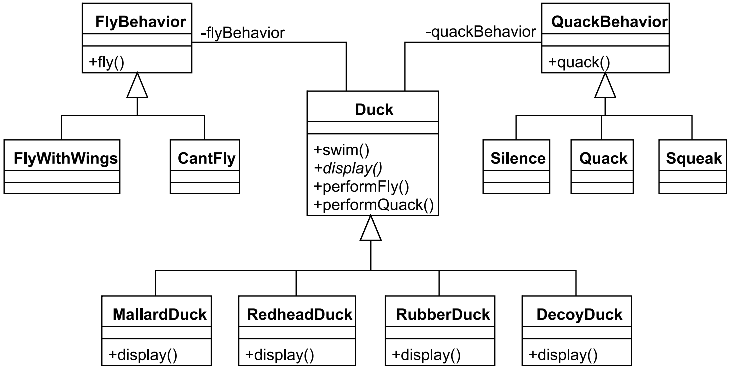
- We are overloading the word “interface” when we say “code to an interface”
  - We can implement “code to an interface” by defining a Java/C# interface and then have various classes implement that interface
  - Or, we can “code to a supertype” and instead define an abstract base class which classes can access via inheritance.
- When we say “code to an interface” it implies that the object that is using the interface will have a variable whose type is the supertype (whether its an interface or abstract base class) and thus
  - can point at any implementation of that supertype
  - and is shielded from their specific class names
    - A Duck will point to a fly behavior with a variable of type `FlyBehavior` NOT `FlyWithWings`; the code will be more loosely coupled as a result.

## Bringing It All Together: Delegation

- To take advantage of these new behaviors, we must modify `Duck` to delegate its flying and quacking behaviors to these other classes
  - rather than implementing this behavior internally
- We'll add two attributes that store the desired behavior and we'll rename `fly()` and `quack()` to `performFly()` and `performQuack()`
  - this last step is meant to address the issue of it not making sense for a `DecoyDuck` to have methods like `fly()` and `quack()` directly as part of its interface
    - Instead, it inherits these methods and plugs-in `CantFly` and Silence behaviors to make sure that it does the right thing if those methods are invoked
- This is an instance of the principle "Favor composition over inheritance"

Idaho State University Computer Science

# New Class Diagram



`FlyBehavior` and `QuackBehavior` define a set of behaviors or a family of algorithms that provide behavior to `Duck`. `Duck` is composing each set of behaviors and can switch among them dynamically, if needed.

ROAR

## Duck.java

Let’s look at the code

# DuckSimilator.java

Let’s look at the code

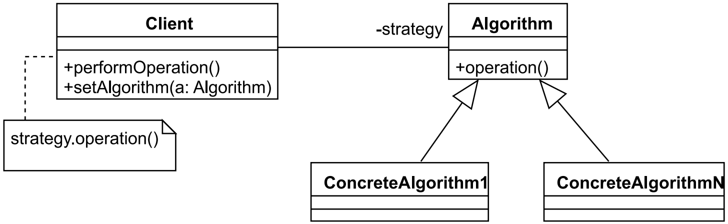
## Not Completely Decoupled

- Is `DuckSimulator` completely decoupled from the `Duck` subclasses?
  - All of its variables are of type `Duck`
- No!
  - The subclasses are still coded into `DuckSimulator`
    - `Duck mallard = new MallardDuck();`
- This is a type of coupling... fortunately, we can eliminate this type of coupling if needed, using a pattern called `Factory`.
  - We'll see Factory in action later

## Meet Strategy

- The solution that we applied to this design problem is known as the Strategy Design Pattern
  - It features the following OO design concepts/principles:
    - Encapsulate What Varies
    - Code to an Interface
    - Delegation
    - Favor Composition over Inheritance
- Definition: The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

# Structure of Strategy



`Algorithm` is pulled out of `Client`. `Client` only makes use of public interface of `Algorithm` and is not tied to concrete subclasses.

`Client` can change its behavior by switching among the various concrete algorithms.

## Are there any questions?

ROAR