Picking Design Patterns

CSCI-4448 - Boese

University of Colorado **Boulder**

Design Patterns

- Design patterns provide you with example after example of the types of techniques that help you tackle software design problems.
- If you learn those lessons, you can apply them to your own designs independent on any particular pattern
- If you then combine those lessons with the good object-oriented principles and heuristics we've seen throughout the semester you will be well on your way to creating solid, flexible, extensible OO designs

- Remember that the names of classes participating in a pattern is unimportant;
  it's the structure (of the relationships and methods) that's important!
- Design patterns are not part of a programming language's syntax. They are ways to think about and organize solutions to common problems.
  They are not written in stone. If a pattern isn't working for you, you can change the pattern to make it work.

Heuristics

Heuristics for Selecting Design Patterns

Phrase	Design Pattern
“Platform independence” “Manufacturer independence”	Abstract Factory
“Must comply with existing interface” “Must reuse existing legacy component”	Adapter
“All commands should be undo-able” “All transactions should be logged”	Command
“Must support aggregate structures” “Must allow for hierarchies of variable depth and width”	Composite
“Policy and mechanisms should be decoupled” “Must allow different algorithms to be interchanged at runtime”	Strategy

Bruegge and Dutoit “Object-Oriented software Engineering”

Heuristics for Selecting Design Patterns

Phrase	Design Pattern
“Attach additional responsibilities to an object dynamically”	Decorator
“Use sharing to support large numbers of fine-grained objects efficiently.”	Flyweight
“Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation”	Iterator
“Without violating encapsulation, capture and externalize an object’s internal state so that the object can be restored to this state later.”	Memento
Provide a surrogate or placeholder for another object to control access to it.	Proxy

Heuristics for Selecting Design Patterns

Phrase	Design Pattern
“Ensure a class only has one instance, and provide a global point of access to it.”	Singleton
“Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.”	State
“Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.” “Allow subclasses to redefine certain steps of an algorithm without changing the algorithm’s structure.”	Template
“Represent an operation to be performed on elements of an object structure, letting you define a new operation without changing the classes of the elements on which it operates”	Visitor

Selecting a Design Pattern

- Consider how design patterns solve design problems.
- Scan Intent sections. Read through each pattern's intent to find one or more that sound relevant to your problem.
- Study how patterns interrelate. Studying these relationships can help direct you to the right pattern or group of patterns.
- Study patterns of like purpose.
- Examine a cause of redesign, look at the patterns that help you avoid the causes of redesign.
- Consider what should be variable in your design. This approach is the opposite of focusing on the causes of redesign. Instead of considering what might force a change to a design, consider what you want to be able to change without redesign. The focus here is on encapsulating the concept that varies, a theme of many design patterns.

http://www.cc.gatech.edu/~simpkins/teaching/gatech/cs2340/slides/design-patterns.pdf

Using Design Patterns (1/2)

- Read the pattern once through for an overview.
- Go back and study the Structure, Participants, and Collaborations sections. Make sure you understand the classes and objects in the pattern and how they relate to one another.
- Look at Sample Code to see a concrete example of the pattern in code.
- Choose names for pattern participants that are meaningful in the application context. OK to use abstract participant names from design pattern. For example, if you use the Strategy pattern for a text compositing algorithm, then you might have classes SimpleLayoutStrategy or TeXLayoutStrategy.

Using Design Patterns (2/2)

- Define the classes. Declare their interfaces, establish their inheritance relationships, and define the instance variables that represent data and object references. Identify existing classes in your application that the pattern will affect, and modify them accordingly.
- Define application-specific names for operations in the pattern. Use the responsibilities and collaborations associated with each operation as a guide. Be consistent in your naming conventions. For example, you might use the "Create-" prefix consistently to denote a factory method.
- Implement the operations to carry out the responsibilities and collaborations in the pattern. The Implementation section from a pattern catalog and sample code offers hints to guide you in the implementation.

Example

Patterns of Patterns

We've been asked to build a new Duck Simulator by a Park Ranger interested in tracking various types of water fowl, ducks in particular.

- New Requirements
  1. Ducks are the focus, but other water fowl (e.g. Geese) can join in too
  2. Control duck creation and keep count of ducks created
  3. Allow ducks to band together into flocks and subflocks
  4. Generate a notification when a duck quacks

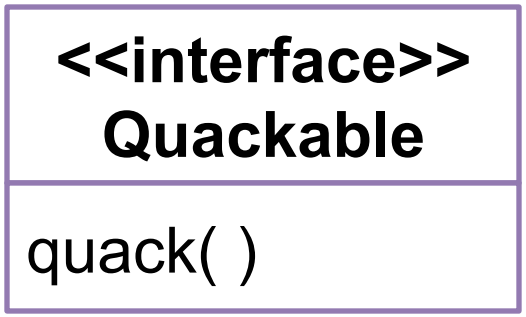
Patterns of Patterns

What patterns are available?

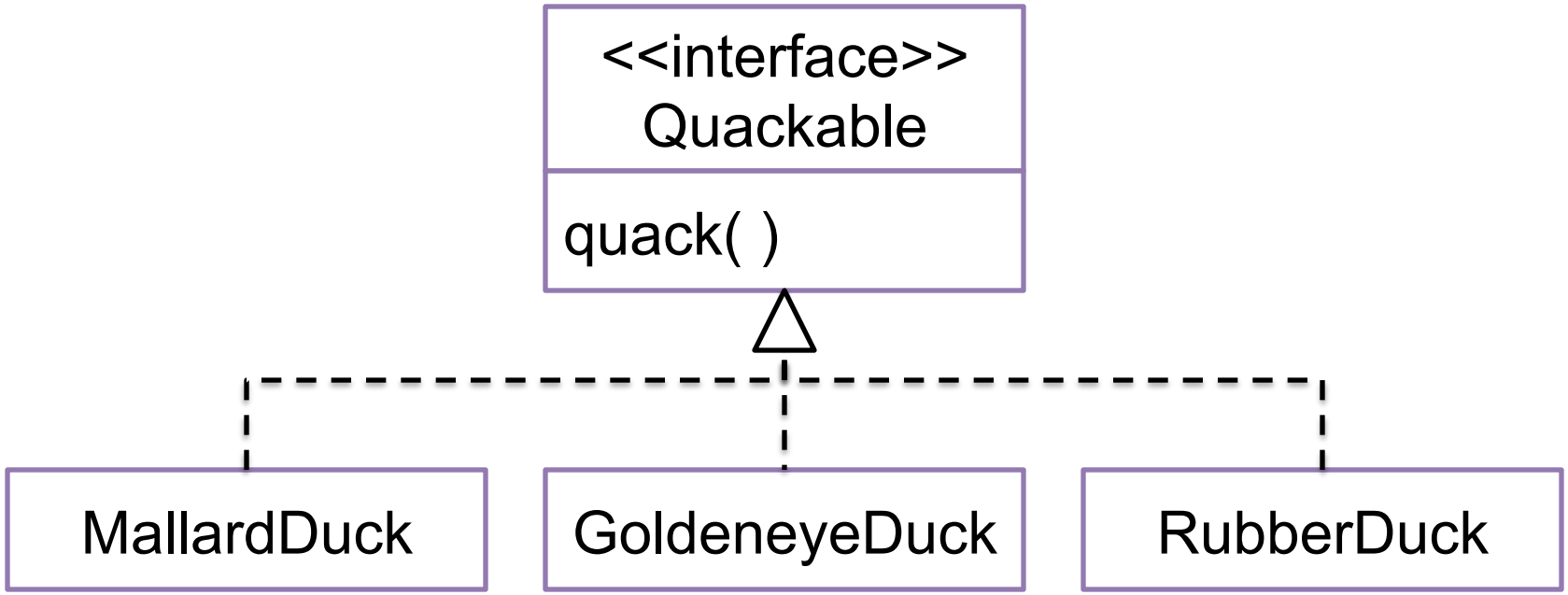
1. Ducks are the focus, but other water fowl (e.g. Geese) can join in too
2. Control duck creation and keep count of ducks created
3. Allow ducks to band together into flocks and subflocks
4. Generate a notification when a duck quacks

Coding to an Interface

- To avoid coding to an implementation, replace all instances of the word “duck” with the word “Quackable”



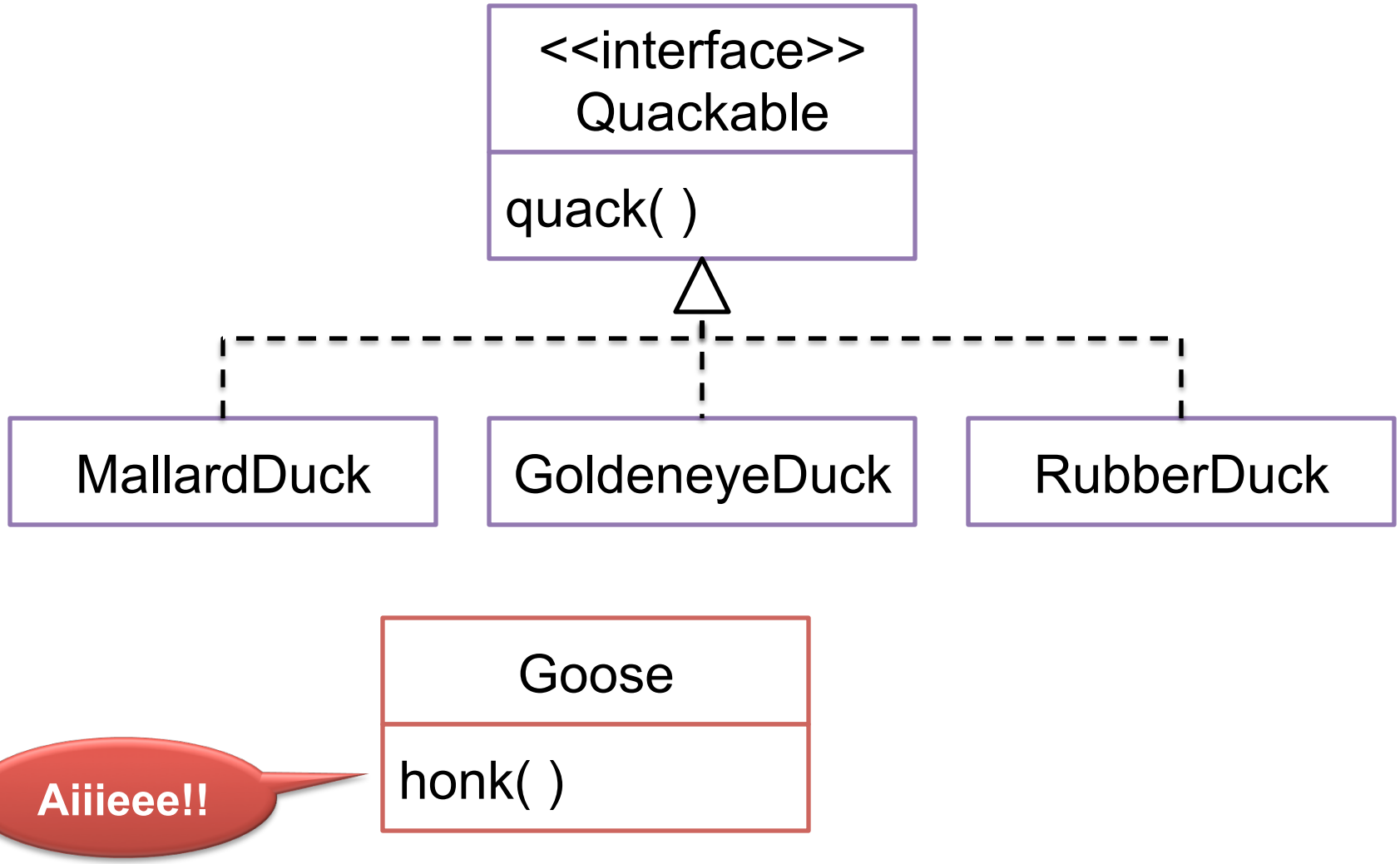
Coding to an Interface



Ducks are the focus,
but other water fowl (e.g. Geese)
can join in too

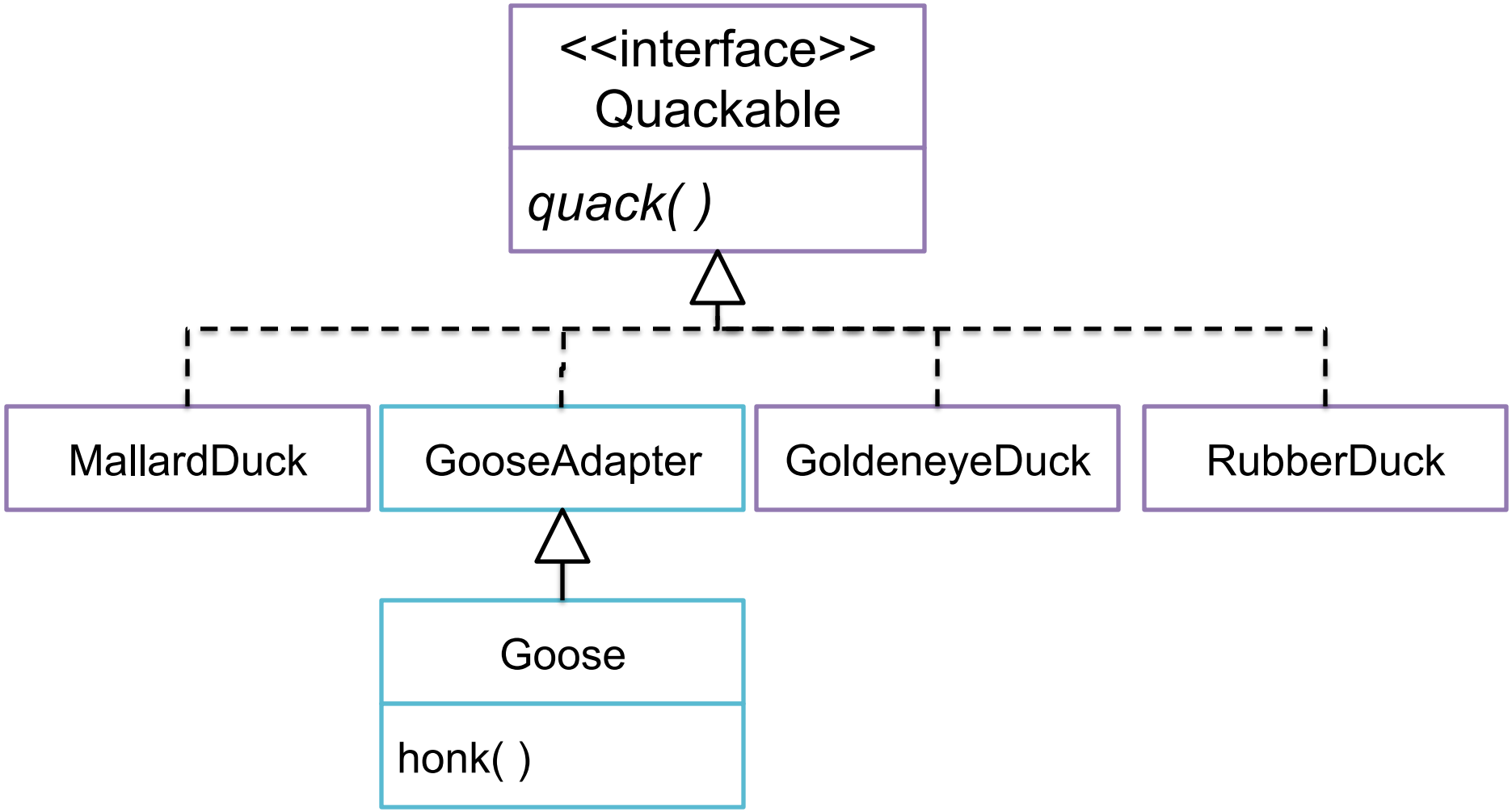
Problem

1. Ducks are the focus, but other water fowl (e.g. Geese) can join in too



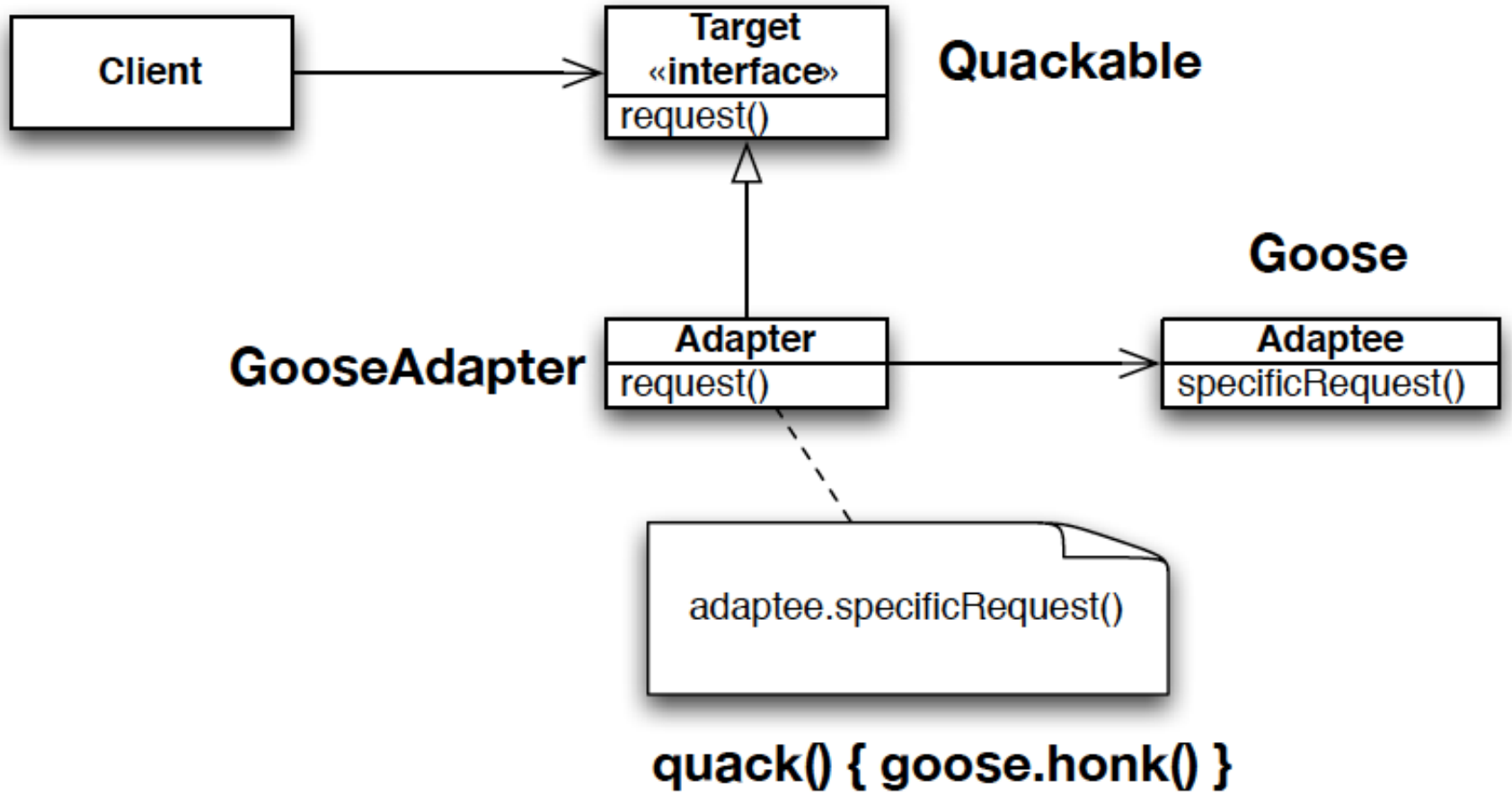
Adapter

1. Ducks are the focus, but other water fowl (e.g. Geese) can join in too



Adapter

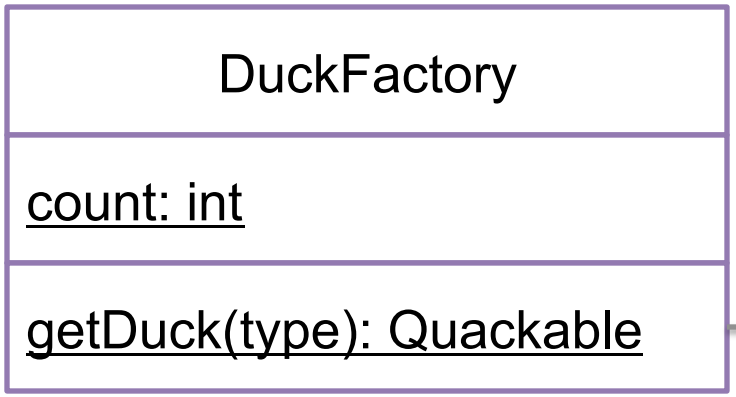
- Recall Adapter…



Control duck creation and keep count of ducks created

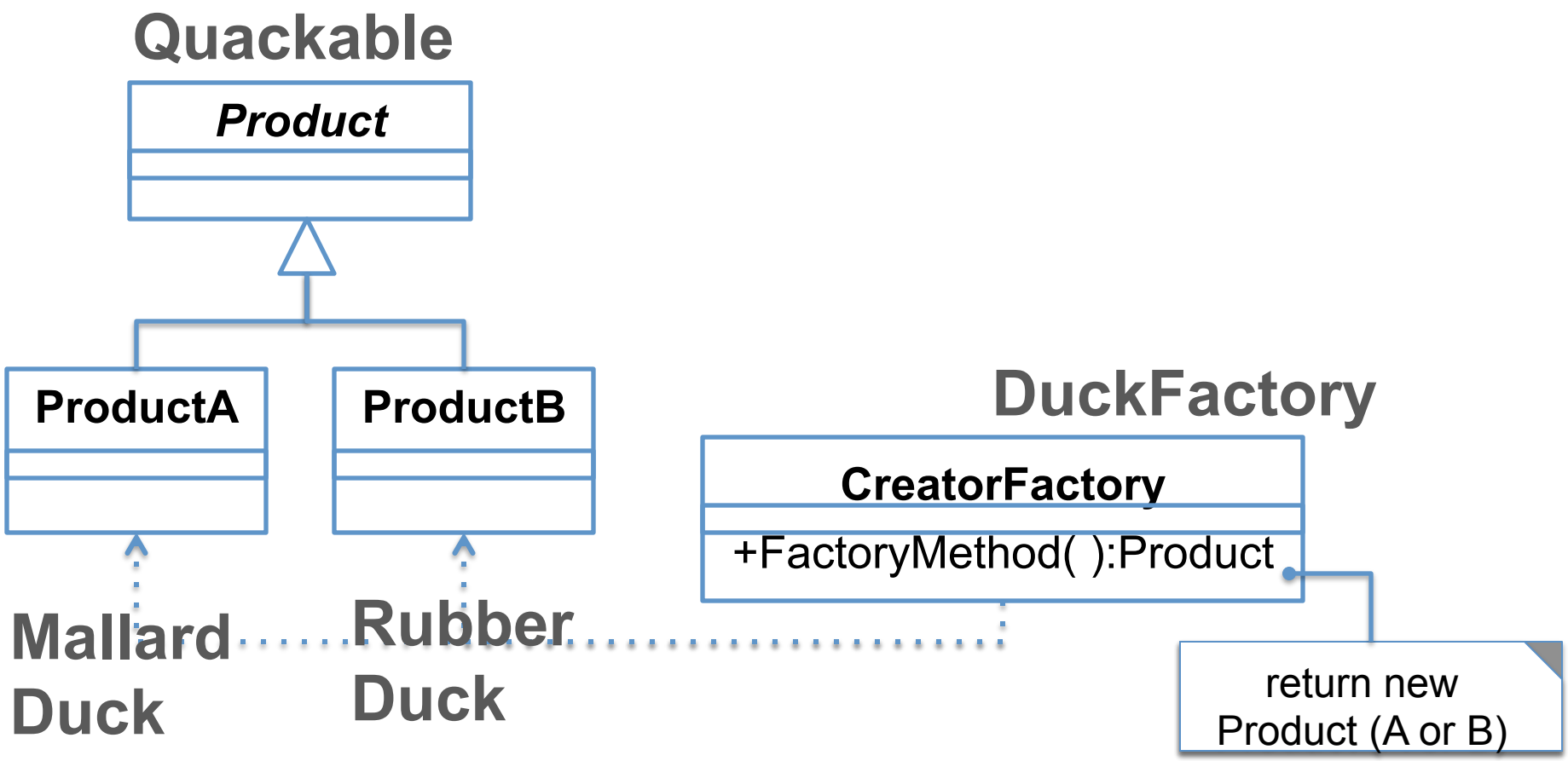
Factory

1. Control duck creation and keep count of ducks created




```
static Quackable getDuck(type)
{
   Quackable duck;
   if (type.equals("Mallard")
      duck = new Mallard( );
   else if (type.equals("Goose")
      duck = new Goose( );
   …
   count++;
   return duck;
}
```

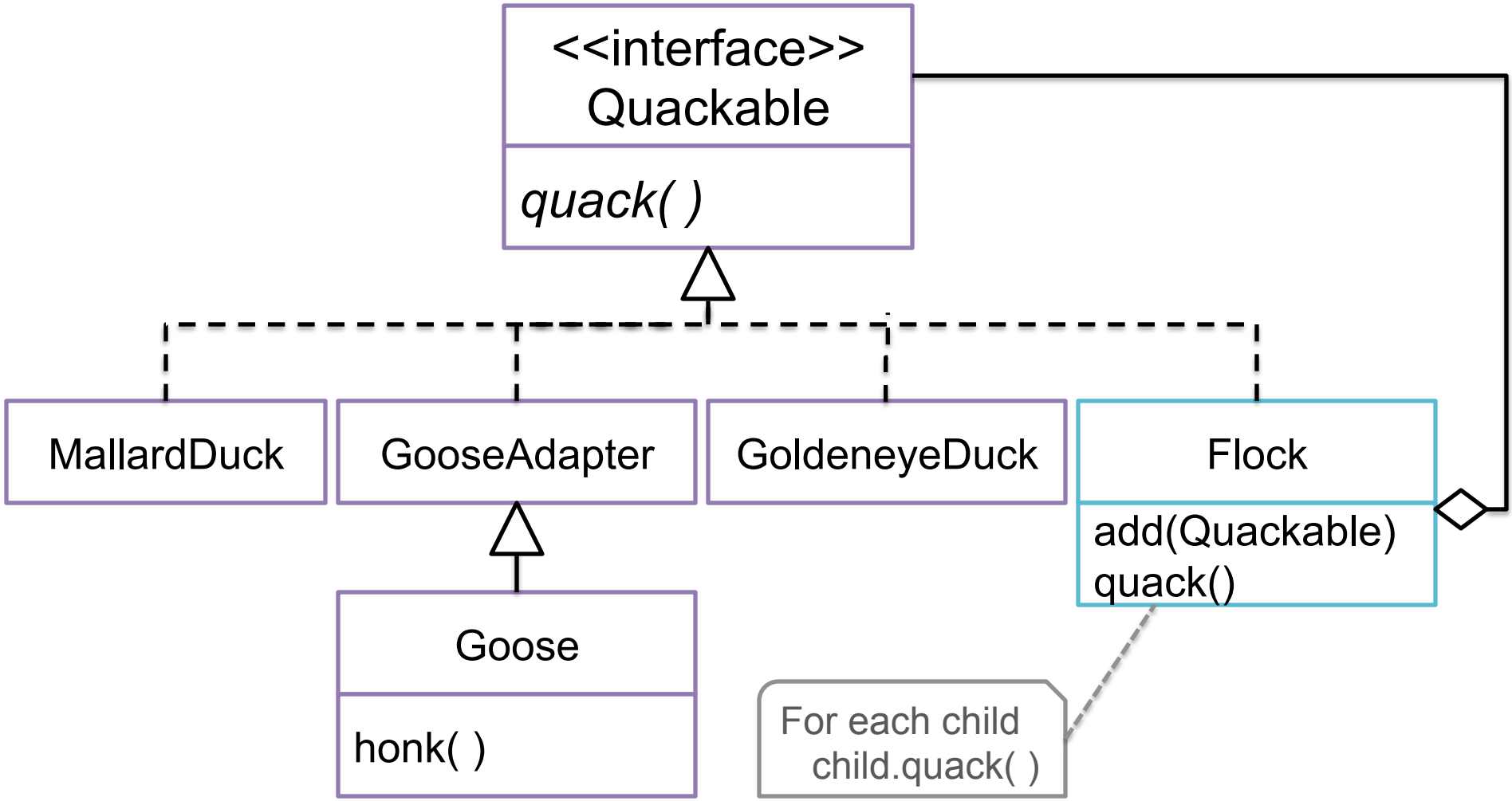
Factory


Quackable
Product
ProductA
ProductB
Mallard Duck
Rubber Duck
DuckFactory
CreatorFactory
+FactoryMethod( ):Product
return new Product (A or B)

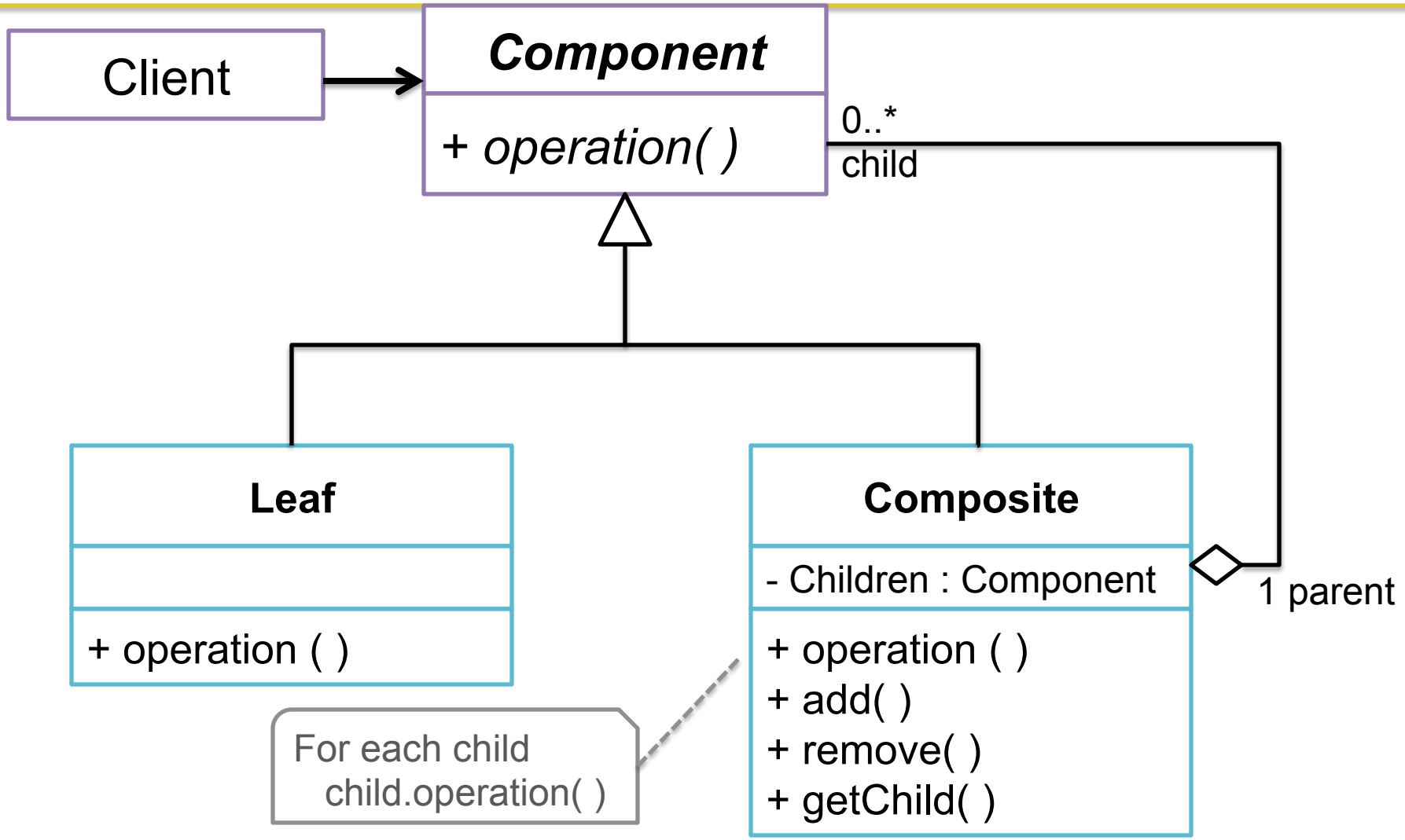
Allow ducks to band together into flocks and subflocks

Composite

Allow ducks to band together into flocks and subflocks



Composite – Structure Version 1

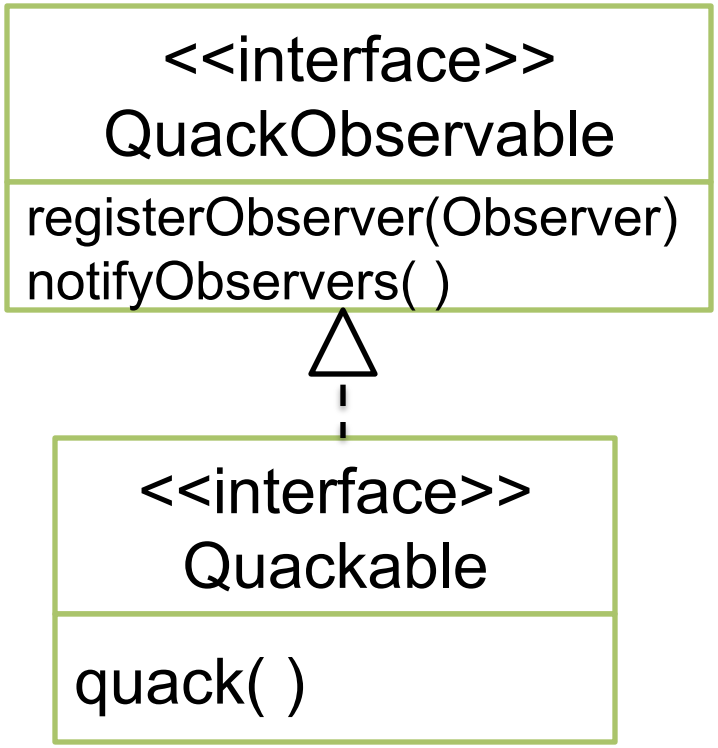


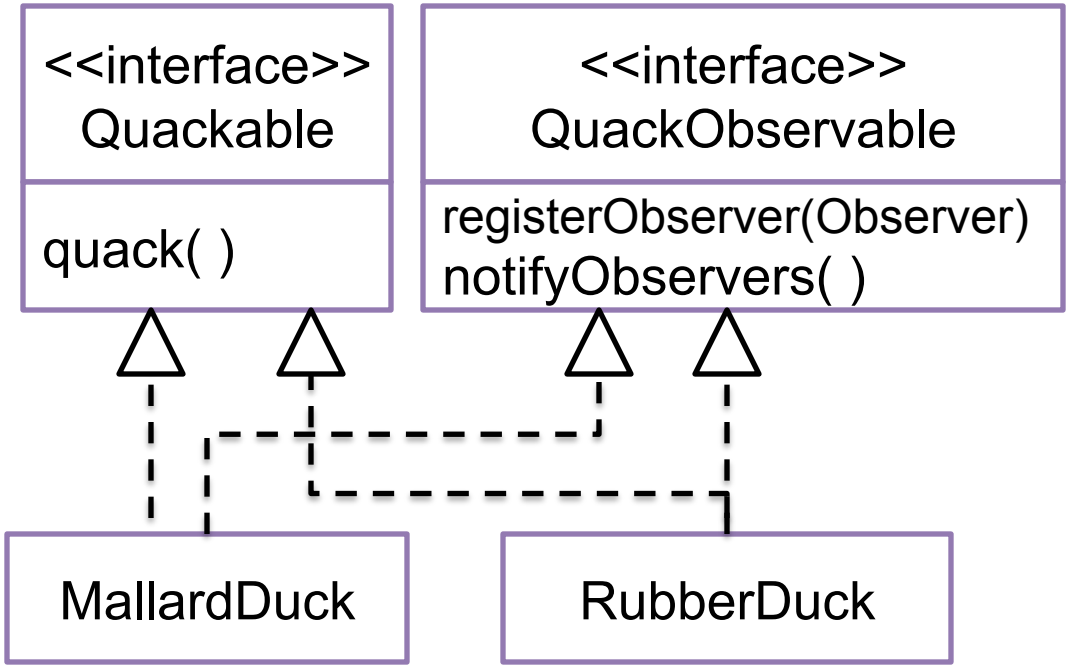
Generate a notification when a duck quacks

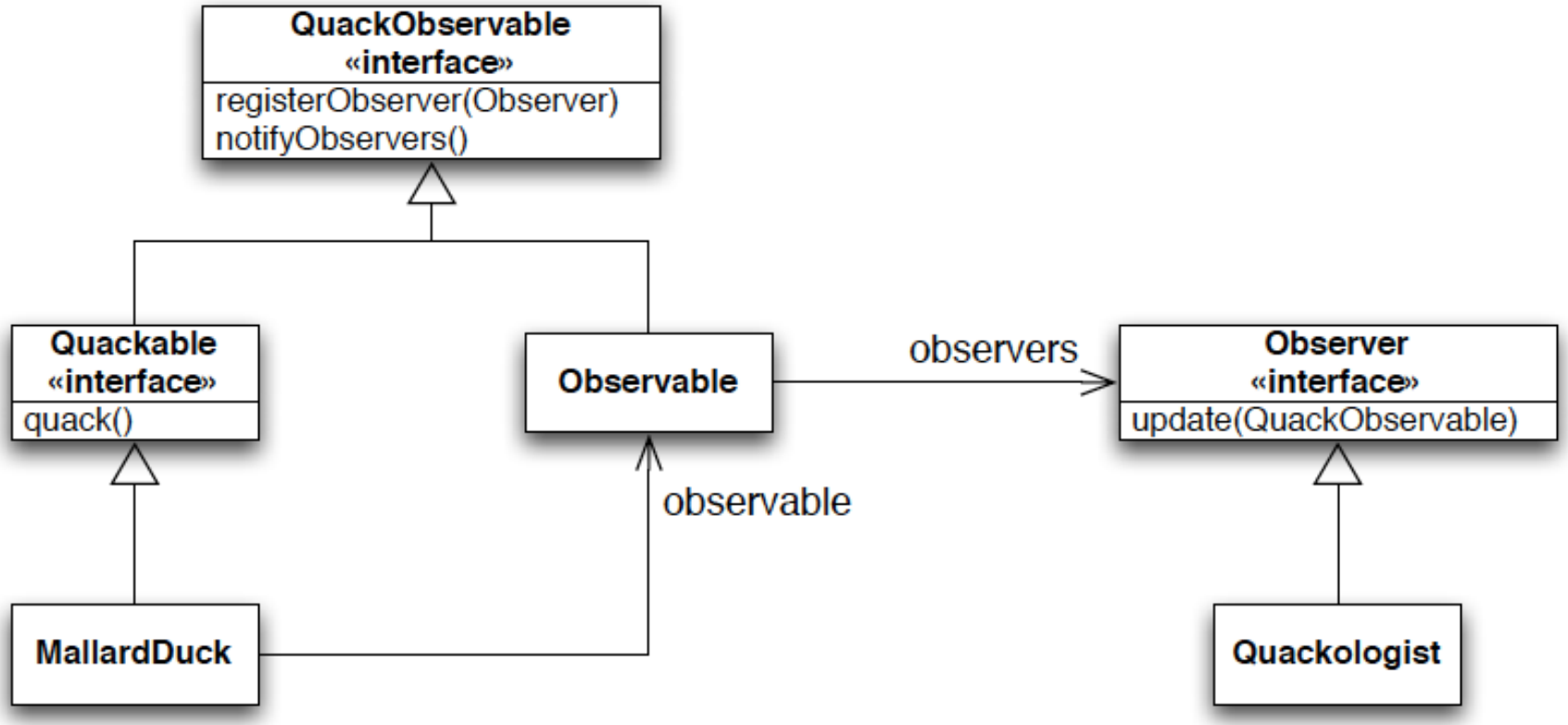
Observer

Generate a notification when a duck quacks

- Which solution?



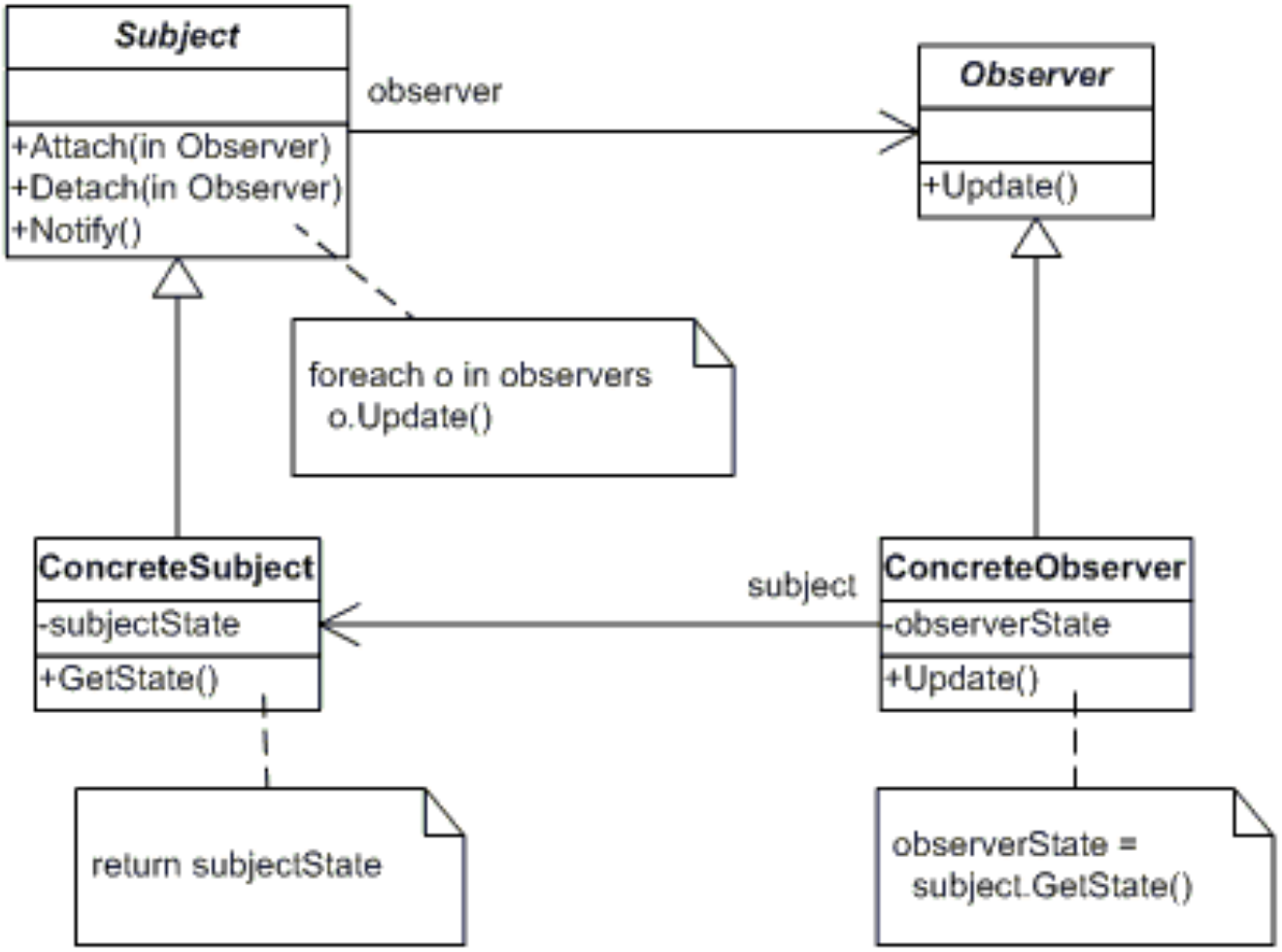




Cool implementation of the Observer pattern. All Quackables are made Subjects by having Quackable inherit from QuackObserver. To avoid duplication of code, an Observable helper class is implemented and composed with each ConcreteQuackable class. Flock does not make use of the Observable helper class directly; instead it delegates those calls down to its leaf nodes.

Dr. Ken Anderson

Observer Pattern - Structure



http://www.codeproject.com/KB/architecture/optimize-performance/optimize_performance_figure2.png

Summary

- As you can see, a single class will play multiple roles in a design
  - Quackable defines the shared interface for four of the patterns
  - Each Quackable implementation has four roles to play: Leaf, ConcreteSubject, ConcreteComponent, ConcreteProduct
- You should now see why names do not matter in patterns
  - Imagine giving MallardDuck the following name:

MallardDuckLeafConcreteSubjectComponentProduct !

Summary

- Instead, it is the structure of the relationships between classes and the behaviors implemented in their methods that make a pattern REAL.
- And when these patterns live in your code, they provide multiple extension points throughout your design.
  - Need a new product, no problem.
  - Need a new observer, no problem.
  - Need a new dynamic behavior, no problem.