CSEN603 – Software Engineering

Lecture 4: Design Patterns

Mervat Abuelkheir
Ammar Yasser
Mohamed Agamia
Nada Hisham

A Prelude: How do developers design objects?

- **Code**
  - Design-while-coding, ideally with power tools such as refactorings. From mental model to code
- **Draw, then code**
  - UML Diagrams
- **Only draw**
  - The tool generates everything from diagrams

1. Spend a few hours or at most one day (with partners) near the start of design
2. Draw UML for the hard, creative parts of the detailed object design
3. Stop and transition to coding

➢ UML drawings
  - inspiration as a starting point
  - the final design in code may diverge and improve
  - Produce dynamic and static diagrams

Guidelines

- Spend significant time doing **interaction diagrams**, not just class diagrams
- Apply **responsibility-driven design** and **GRASP** principles to dynamic modeling
- Do **static modeling after dynamic modeling**

Responsibility-driven Design

General Responsibility Assignment Software Patterns (GRASP) are guidelines for assigning responsibility to classes and objects towards other objects

Creator
- Doing
- Who creates class A?

Controller
- Doing
- What first object beyond the UI layer receives and controls a system operation?

Information Expert
- Knowing
- Who knows the information to fulfill a responsibility?

High Cohesion
- Calling/communication
- How to keep object focused, and manageable?

Low Coupling
- Calling/communication
- How to reduce the impact of change?

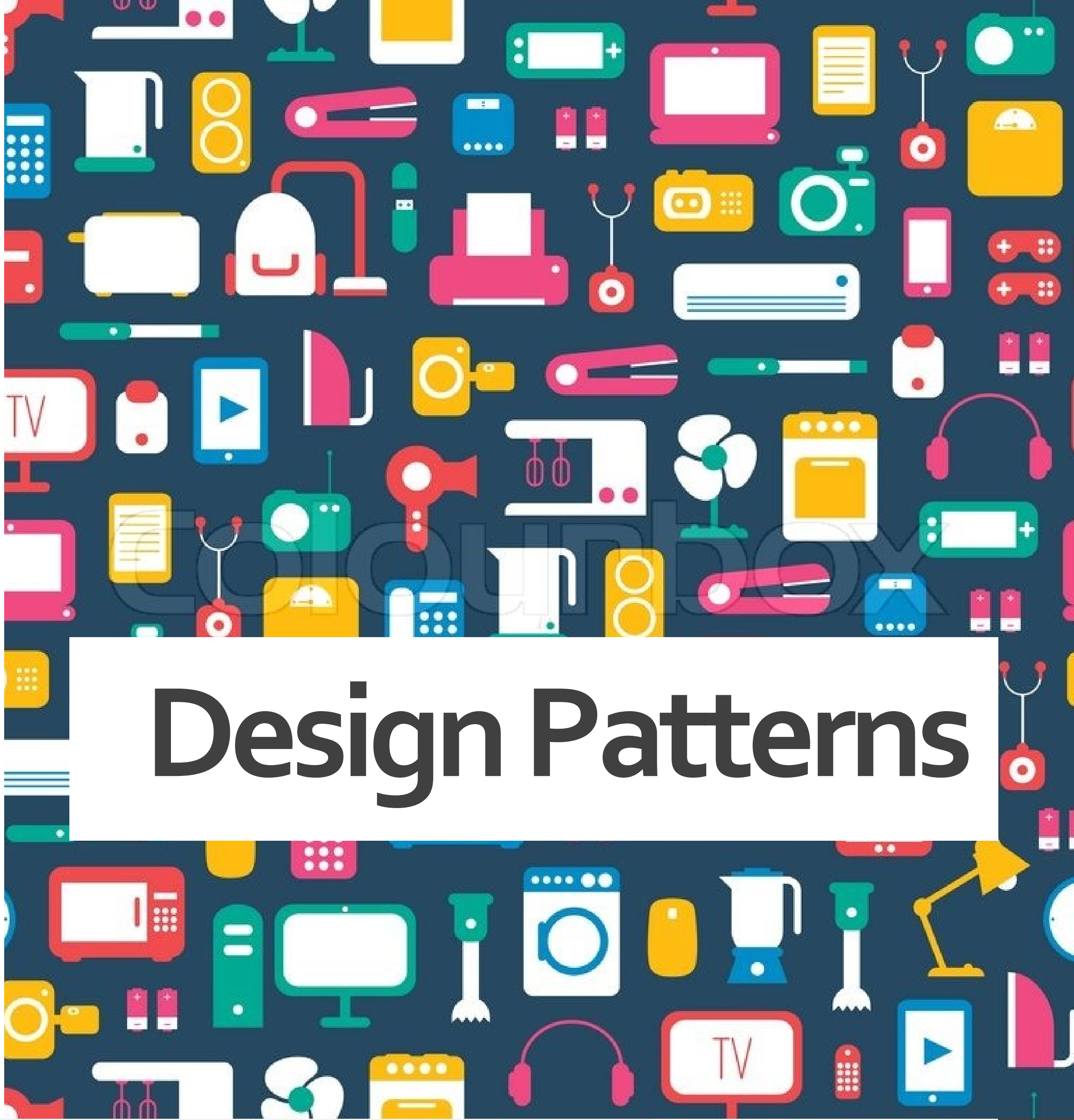
Responsibility-driven Design

- **Knowing** something (memorization of data or object attributes)
- **Doing** something on its own (computation programmed in a “method”)
  - e.g. business rules for implementing business policies and procedures
- **Calling** methods on dependent objects (communication by sending messages)
  - e.g. calling constructor methods

- **Design patterns provide systematic, tried-and-tested, heuristics for subdividing and refining object responsibilities**, instead of arbitrary, ad-hoc solutions

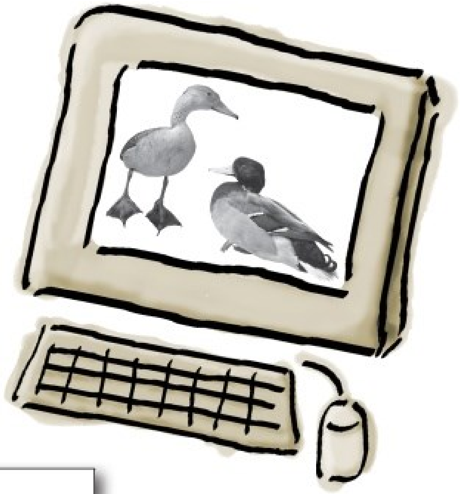
- A **design pattern is a description of the problem and the <u>essence</u> of its solution**
- A way of **reusing** abstract knowledge about a (known) problem and its solution
- Describe **best practices**, **good designs**, and **capture experience**
- Should be **sufficiently abstract** to be reused in different settings
- Pattern descriptions make use of OO characteristics such as inheritance and polymorphism

NOT **code reuse**, but **experience reuse**

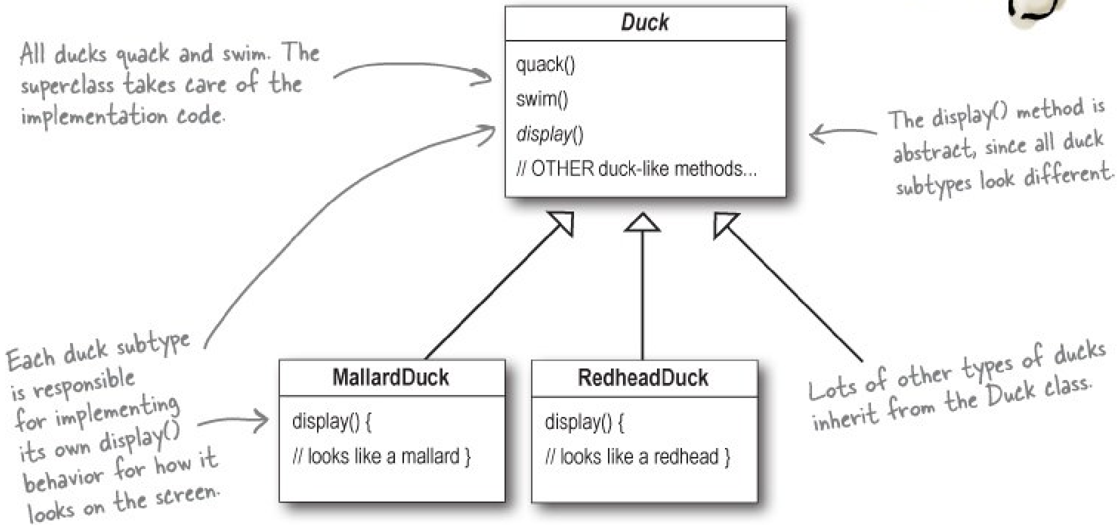


Example – Duck Pond Simulator

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

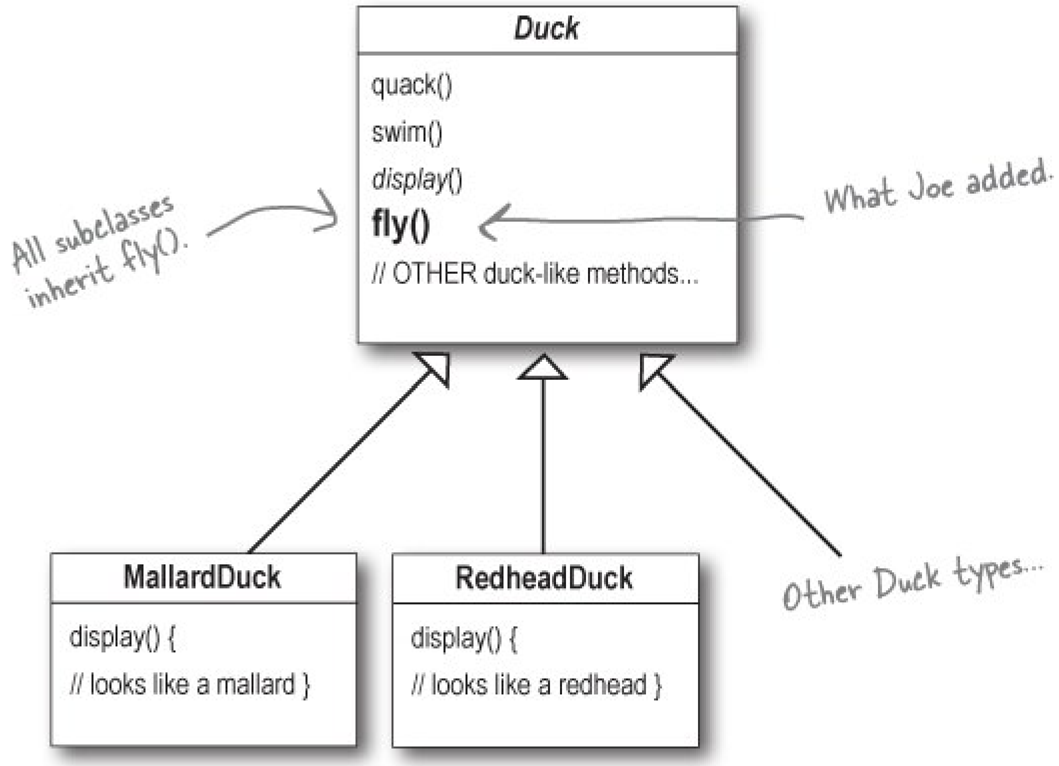


- But a request has arrived to **allow ducks to also fly**. (We need to stay ahead of the competition!)



Example – Duck Pond Simulator

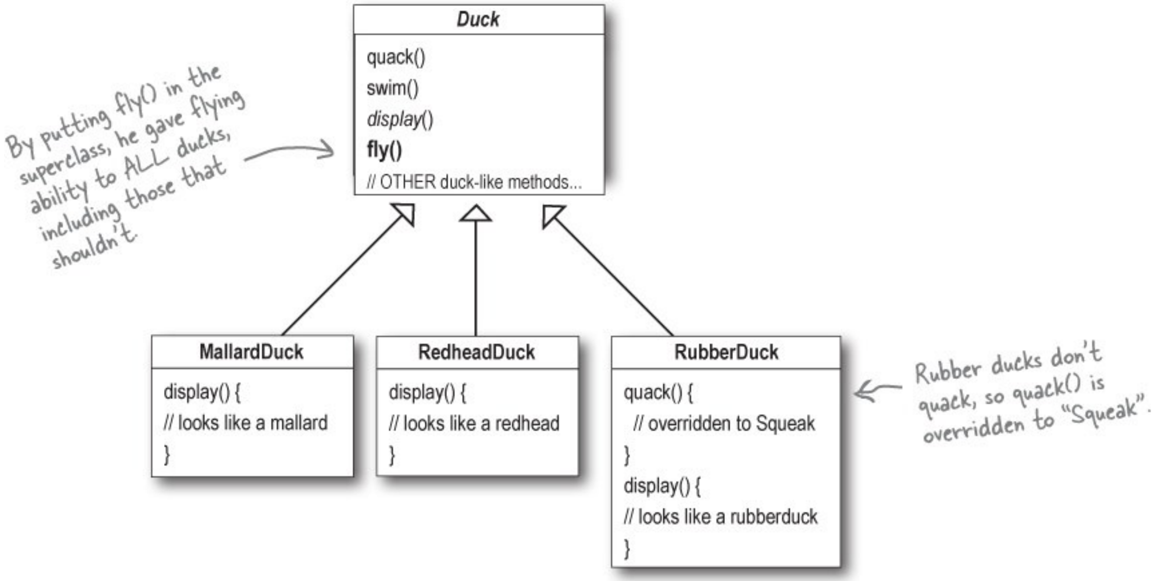
- Easy solution – code reuse via Inheritance
  - Add `fly()` to `Duck`; all ducks can now fly



Adding fly behavior

Example – Duck Pond Simulator

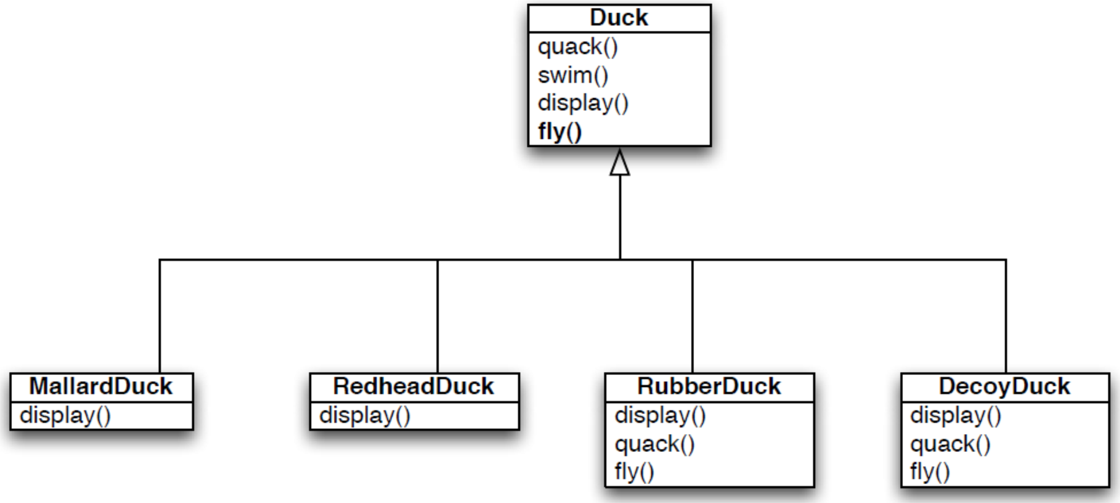
- Now we add a `RubberDuck` subclass, but it doesn't exactly quack, so we override `quack()` to make them squeak
- OOPS! Rubber ducks do not fly!
- We could override `fly()` in `RubberDuck` to make it do nothing
  - Not ideal – we might find other `Duck` subclasses



Adding new duck type

Example – Duck Pond Simulator

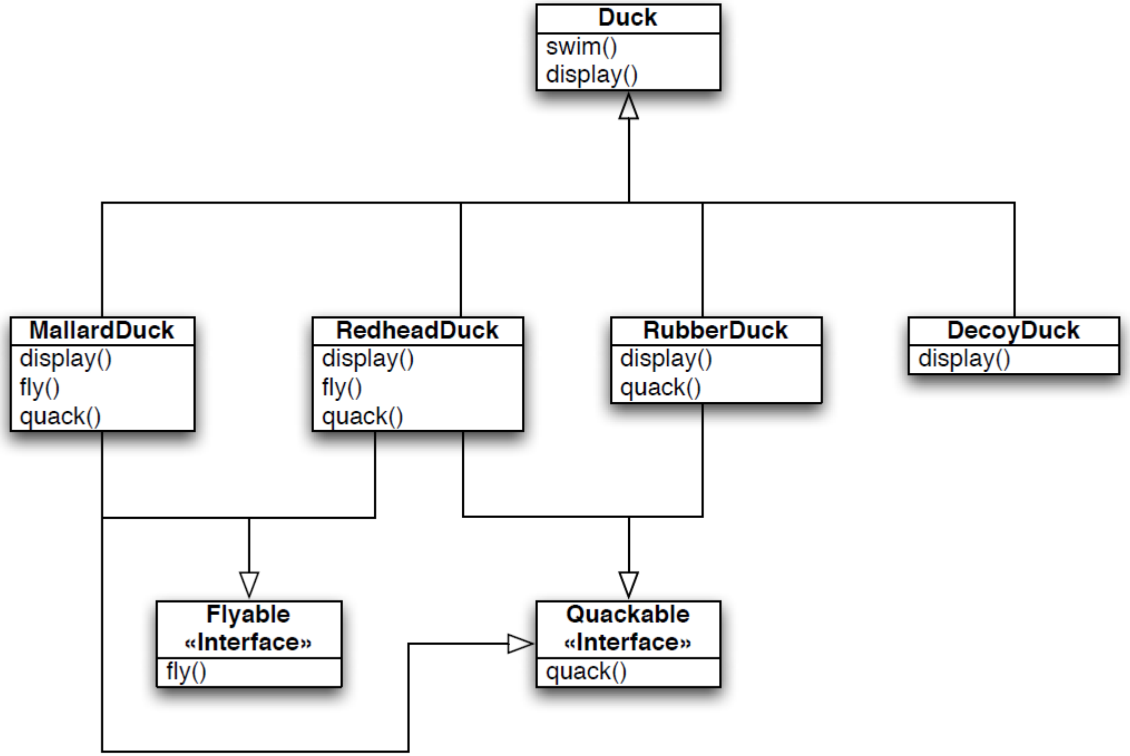
- What was supposed to be a good instance of reuse via inheritance becomes a **maintenance headache!**
- Code is duplicated across subclasses
- Runtime behavior changes are difficult
- We can’t make ducks dance
- Hard to gain knowledge of all duck behaviors
- Ducks can’t fly and quack at the same time
- Changes can unintentionally affect other ducks



Adding new duck type

Example – Duck Pond simulator

- Here we define **two interfaces** and allow subclasses to implement the interfaces they need
- What are the trade-offs?



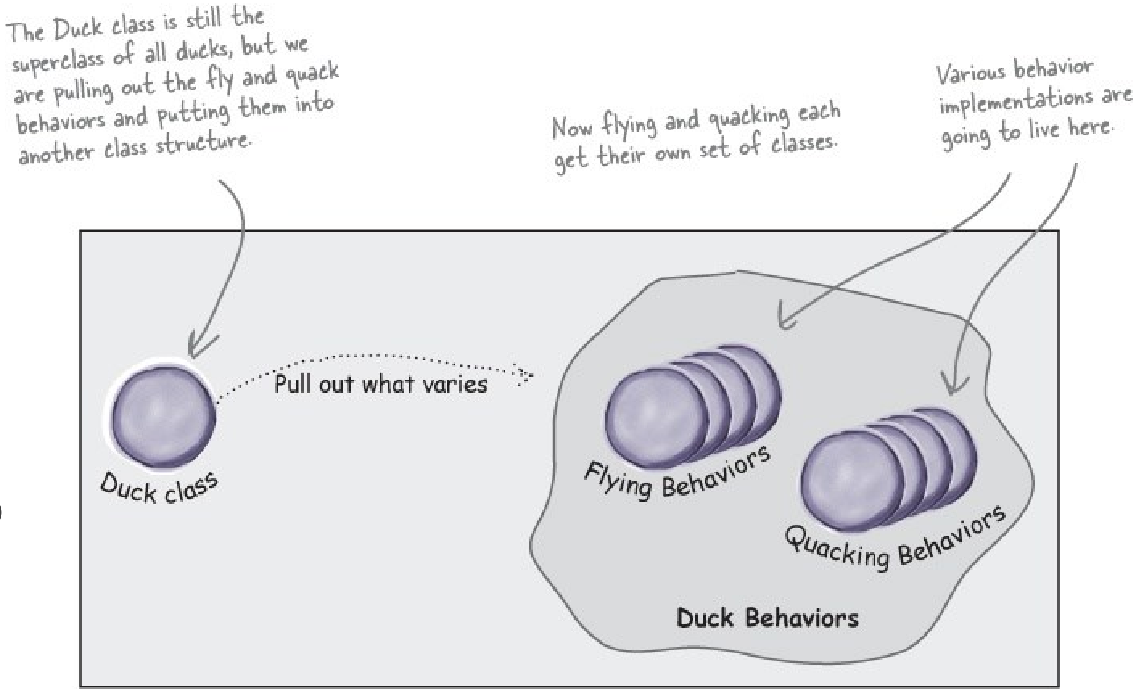
SimUDuck interface

Design Trade-Offs

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

OO Principles to the Rescue!

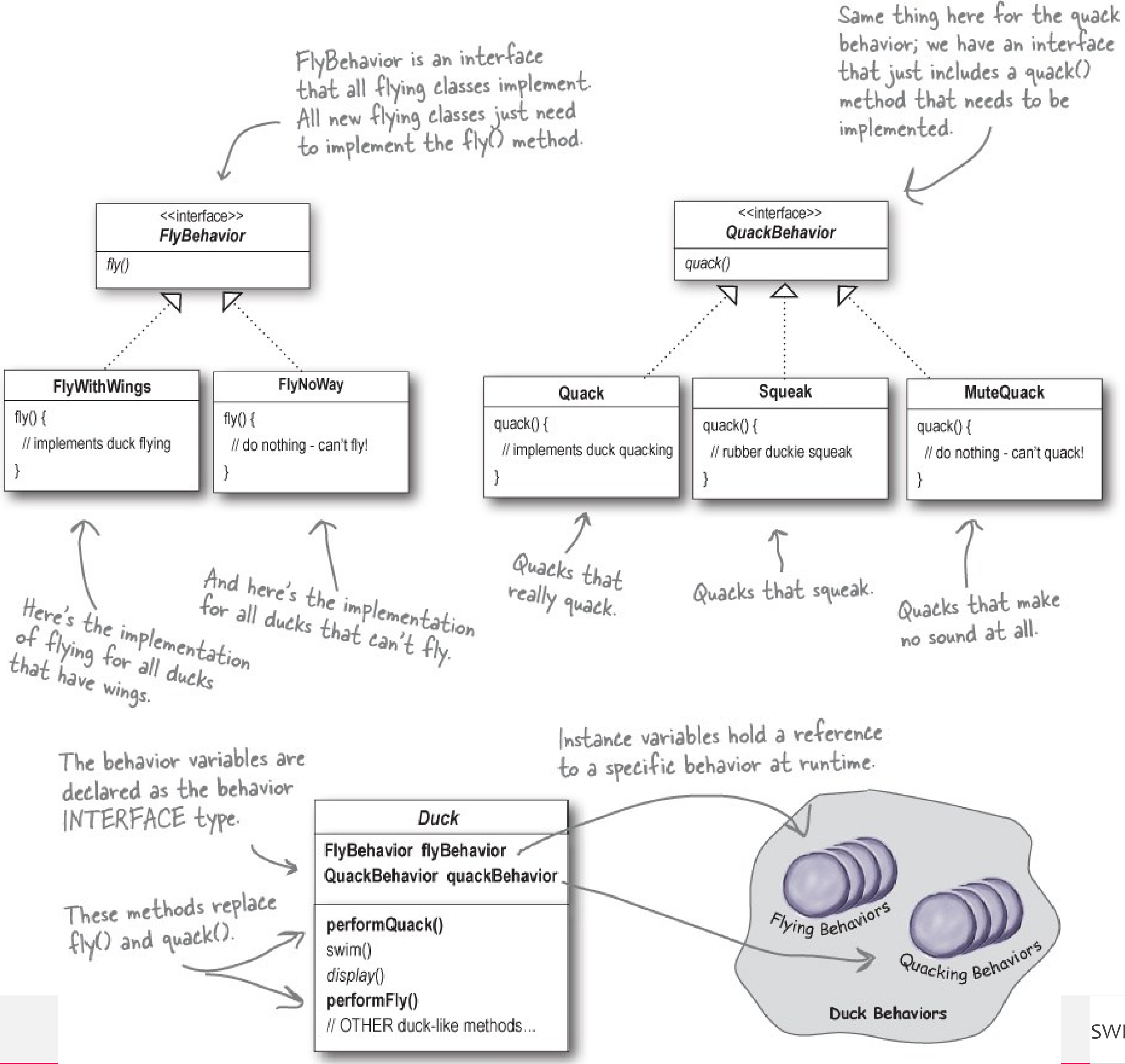
- **Encapsulate What Varies**
  - The “what varies” here is the behaviors between `Duck` subclasses
  - We need to pull out behaviors that vary across subclasses and put them in their own classes (i.e. encapsulate them)
- The result: fewer unintended consequences from code changes [such as when we added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



How To Do It

- Take any behavior that varies across Duck subclasses and pull them out of `Duck`
  - `Duck` will no longer have `fly()` and `quack()` methods <u>directly</u>
  - Create two sets of classes, one that implements fly behaviors and one that implements quack behaviors
- **Code to an Interface**
  - Duck classes won’t need to know any of the implementation details for their own behaviors
  - Make sure that each member of the two sets of classes implements a particular interface
  - For `QuackBehavior`, we’ll have `Quack`, `Squeak`, `Silence`
  - For `FlyBehavior`, we’ll have `FlyWithWings`, `CantFly`, `FlyWhenThrown`, …
- **Other classes can gain access to these behaviors** (if behavior applies)
- We can add **additional behaviors without impacting other classes**

How To Do It



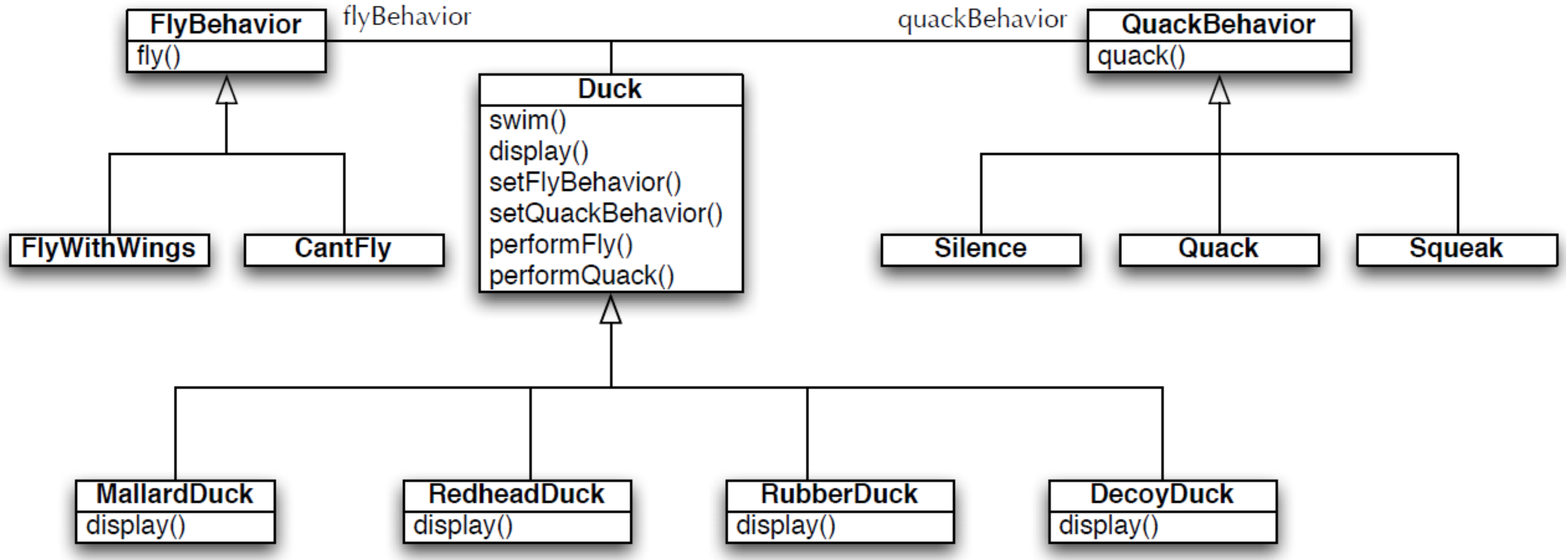
FlyBehavior is an interface that all flying classes implement. All new flying classes just need to implement the fly() method.
Same thing here for the quack behavior; we have an interface that just includes a quack() method that needs to be implemented.
<<interface>>
FlyBehavior
fly()
<<interface>>
QuackBehavior
quack()
FlyWithWings
fly() {
// implements duck flying
}
FlyNoWay
fly() {
// do nothing - can't fly!
}
Quack
quack() {
// implements duck quacking
}
Squeak
quack() {
// rubber duckie squeak
}
MuteQuack
quack() {
// do nothing - can't quack!
}
Here's the implementation of flying for all ducks that have wings.
And here's the implementation for all ducks that can't fly.
Quacks that really quack.
Quacks that squeak.
Quacks that make no sound at all.
The behavior variables are declared as the behavior INTERFACE type.
Instance variables hold a reference to a specific behavior at runtime.
These methods replace fly() and quack().
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
performQuack()
swim()
display()
performFly()
// OTHER duck-like methods...
Flying Behaviors
Quacking Behaviors
Duck Behaviors

Favor Delegation Over Inheritance

- 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()`
  - because it desn't make 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 delegation over inheritance**"

New Class Diagram

`FlyBehavior` and `QuackBehavior` define a set of behaviors that provide behavior to `Duck`. `Duck` delegates to each set of behaviors and can switch among them dynamically, if needed. While each subclass now has a `performFly()` and `performQuack()` method, at least the user interface is uniform and those methods can point to null behaviors when required



Duck.java

- “code to interface”, delegation, encapsulation, and ability to change behaviors dynamically

```java
public class Duck {
   QuackBehavior quackBehavior;
   // more

   public void performQuack() {
     quackBehavior.quack();
   }
}
```

Each Duck has a reference to something that implements the QuackBehavior interface.

Rather than handling the quack behavior itself, the Duck object delegates that behavior to the object referenced by quackBehavior.

```java
public abstract class Duck {
    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;

    public Duck() {
    }

    public void setFlyBehavior (FlyBehavior fb) {
        flyBehavior = fb;
    }

    public void setQuackBehavior(QuackBehavior qb) {
        quackBehavior = qb;
    }

    abstract void display();

    public void performFly() {
        flyBehavior.fly();
    }

    public void performQuack() {
        quackBehavior.quack();
    }

    public void swim() {
        System.out.println("All ducks float, even decoys!");
    }
}
```

DuckSimulator.java (Part 1)

- All variables are of type Duck, not the specific subtypes; "code to interface" in action

- The power of delegation – we can change behaviors at run-time

```java
public class MallardDuck extends Duck {

    public MallardDuck() {
        quackBehavior = new Quack();
        flyBehavior = new FlyWithWings();
    }

    public void display() {
        System.out.println("I'm a real Mallard duck");
    }
}
```

A MallardDuck uses the Quack class to handle its quack, so when performQuack() is called, the responsibility for the quack is delegated to the Quack object and we get a real quack.

And it uses FlyWithWings as its FlyBehavior type.

Remember, MallardDuck inherits the quackBehavior and flyBehavior instance variables from class Duck.

```java
public static void main(String[] args) {

  List<Duck> ducks  = new LinkedList<Duck>();

  Duck model = new ModelDuck();

  ducks.add(new DecoyDuck());
  ducks.add(new MallardDuck());
  ducks.add(new RedHeadDuck());
  ducks.add(new RubberDuck());
  ducks.add(model);

  processDucks(ducks);

  // change the Model Duck's behavior dynamically
  model.setFlyBehavior(new FlyRocketPowered());
  model.setQuackBehavior(new Squeak());

  processDucks(ducks);
 }
}
```

DuckSimulator.java (Part 2)

- Because of abstraction and polymorphism, `processDucks()` consists of nice, clean, robust & extensible code!

```java
import java.util.LinkedList;
import java.util.List;

public class DuckSimulator {

  public static void processDucks(List<Duck> ducks) {
    for (Duck d : ducks) {
      System.out.println("------------------------------------");
      System.out.println("Name:  " + d.getClass().getName());
      d.display();
      d.performQuack();
      d.performFly();
      d.swim();
    }
  }
```

Full Duck example: https://www.oreilly.com/library/view/head-first-design/0596007124/ch01.html

Meet the **Strategy** Design Pattern

- The solution 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 Delegation (Composition) over Inheritance
- 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

Types of Design Patterns

- **Creational Patterns** (how objects are instantiated)
  - construct objects so they can be decoupled from their implementing system
- **Structural Patterns** (how objects / classes can be combined)
  - form large object structures from disparate objects
- **Behavioral Patterns** (how objects communicate)
  - manage algorithms, relationships, and responsibilities between objects

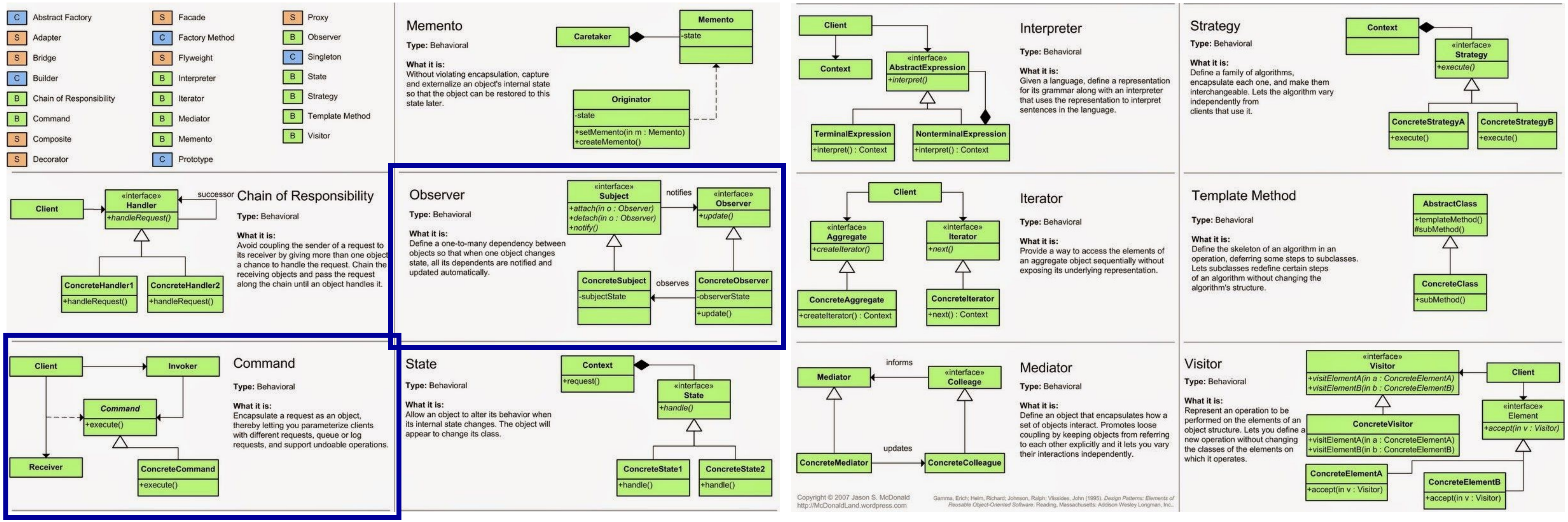
- **Concurrency patterns** (how computations are parallelized / distributed)

Core Design Patterns

Creational

Structural

Behavioral

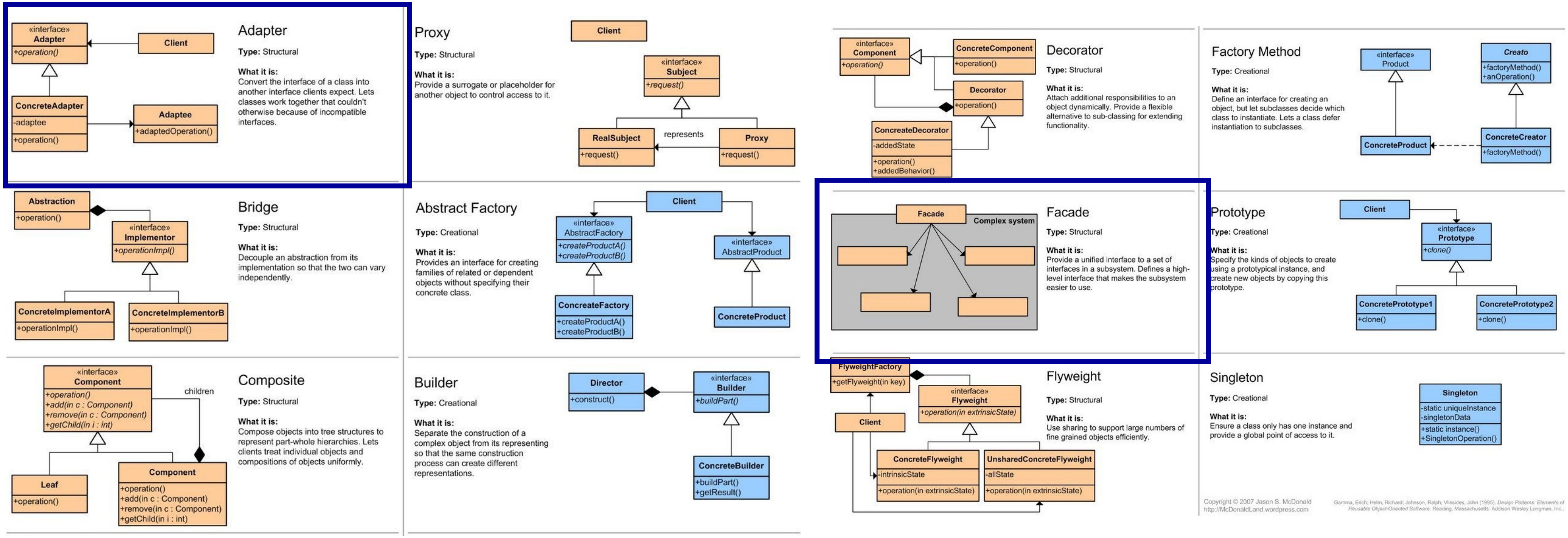


https://github.com/kamranahmedse/design-patterns-for-humans

https://github.com/DovAmir/awesome-design-patterns

Core Design Patterns

Creational Structural Behavioral



https://github.com/kamranahmedse/design-patterns-for-humans

https://github.com/DovAmir/awesome-design-patterns

Pattern Elements

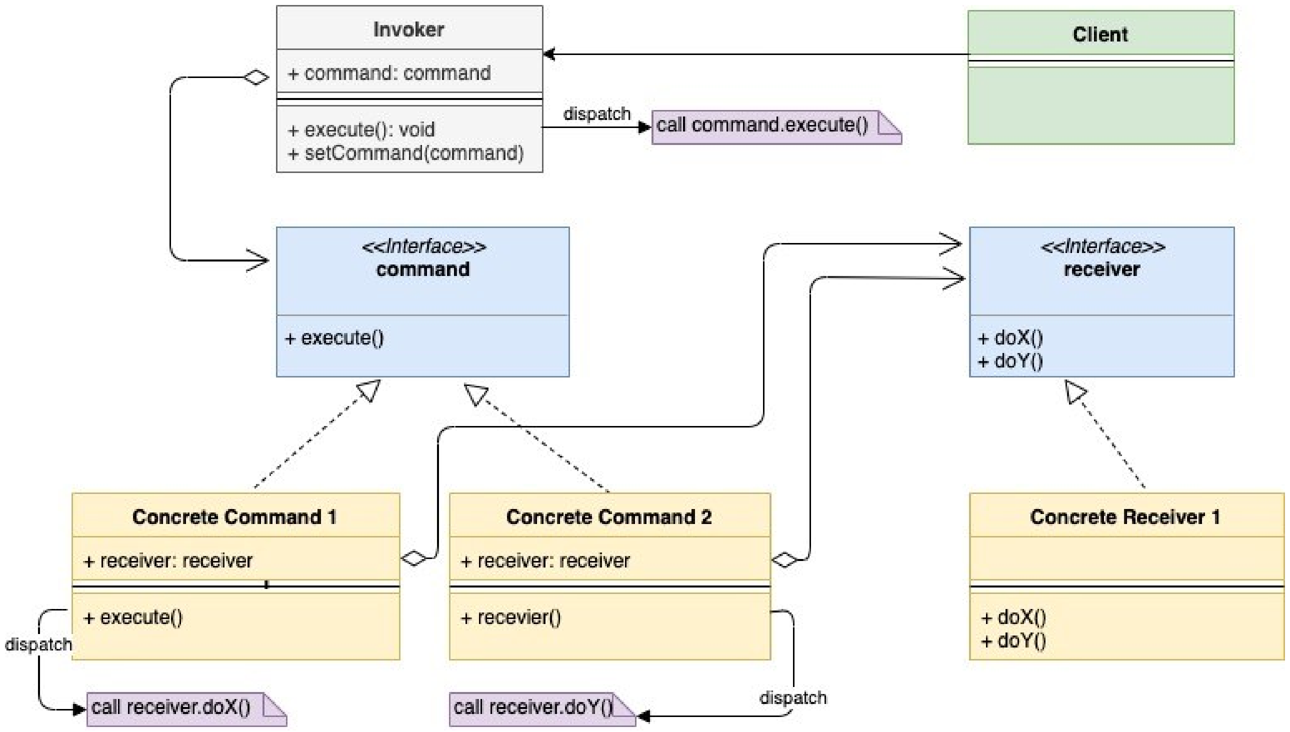
- **Name**
  - A meaningful pattern identifier
- **Description**
- **Problem description**
- **Solution description**
  - Not a concrete design but a template for a design solution that can be instantiated in different ways
- **Consequences**
  - Results and trade-offs of applying the pattern

Pattern name	Observer
Description	Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.
Problem description	In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations should support interaction and, when the state is changed, all displays must be updated. This pattern may be used in all situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.
Solution description	This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed. The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated.
Consequences	The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.

The Command Pattern

- **Name**
  - Command
- **Description**
  - Encapsulate a request as an object, letting you parametrize clients with different requests
- **Problem description**
  - When you need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request
- **Solution description**
  - Create an abstract base class that maps a receiver with an action. The base class contains an `execute()` method that calls the action on the receiver
- **Consequences**
  - Decoupling of the command and the invoker
  - Possible to execute the request at a different time and/or at a different location

Class diagram with the Command pattern



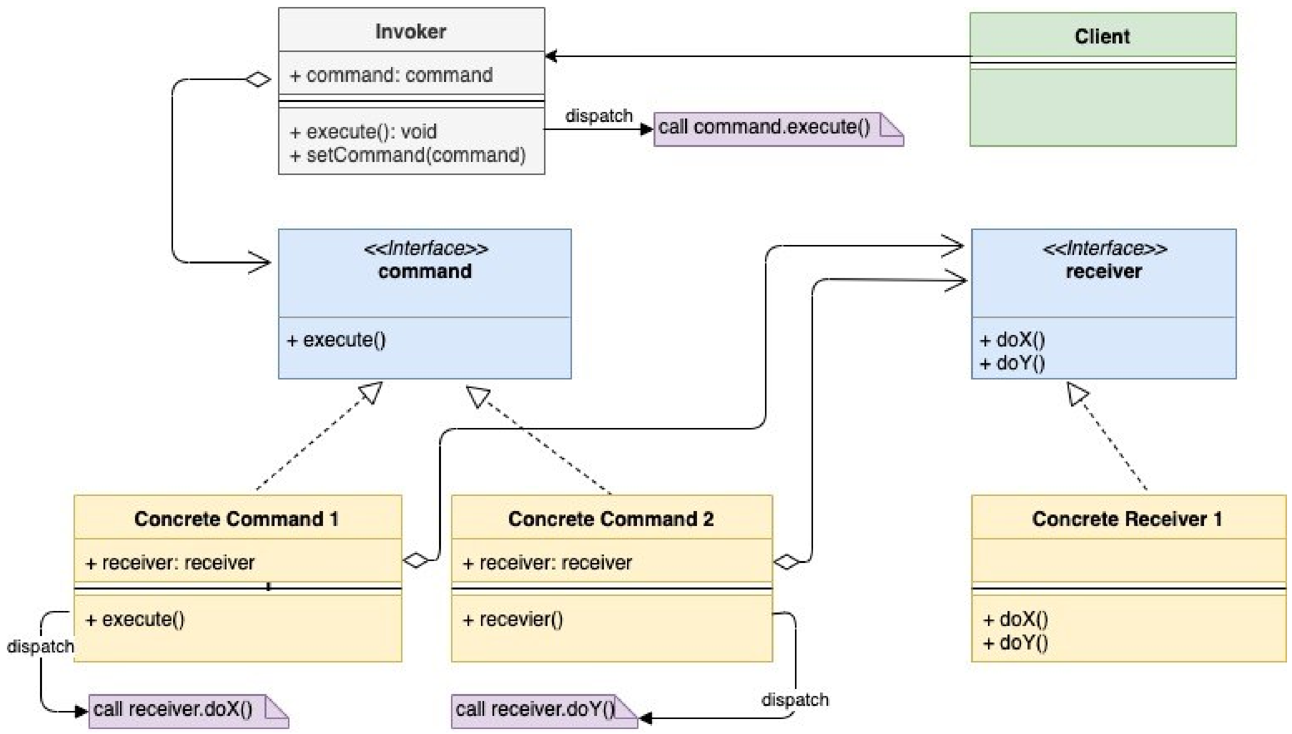
What should Command Know? Do?

The Command Pattern

Object responsibilities

- **Know:**
  - Receiver of action request
  - Whether action is reversible (optional)
- **Do:**
  - Execute an action
  - Undo an action if reversible (optional)

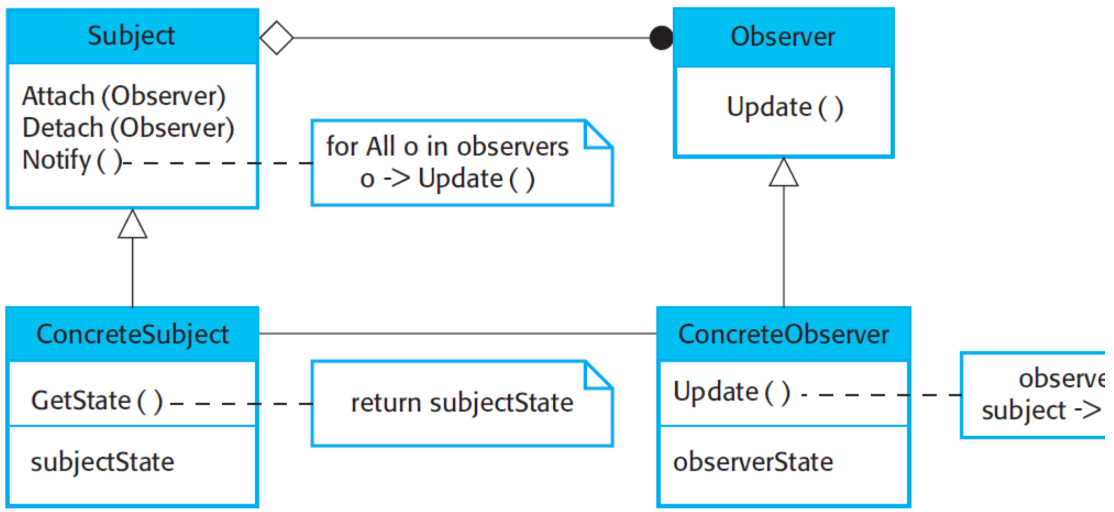
Class diagram with the Command pattern



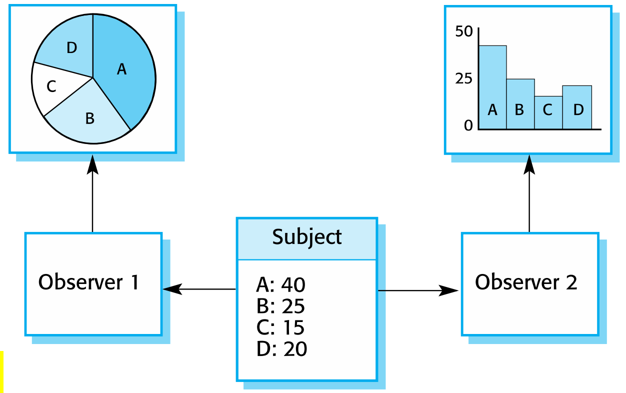
The Observer Pattern

- **Name**
  - **Observer (a.k.a Publish-Subscribe)**
- **Description**
  - Separate the display of object state from the object itself
- **Problem description**
  - When you need multiple displays of a single state
- **Solution description**
  - Define `Subject` and `Observer` objects so that when a subject changes state, all registered observers are notified and updated
- **Consequences**
  - Allow addition of new cases
  - Changes to the subject may cause updates to observers

Class diagram with the Observer pattern



Multiple displays using the Observer pattern



What should Observer Know? Do? What should Subject Know? Do?

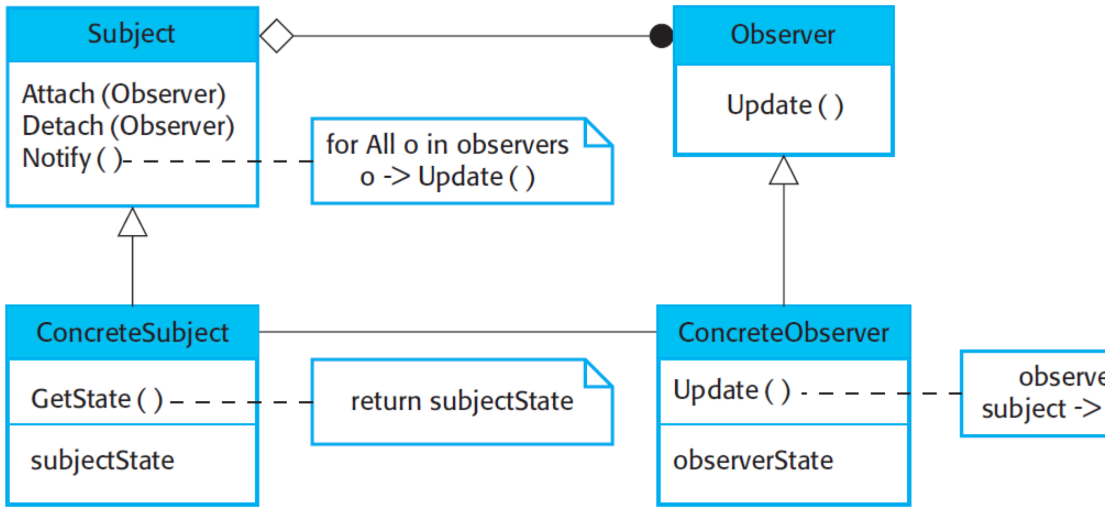
The Observer Pattern

Object responsibilities

- **Publisher/Subject**
  - **Know:**
    - Event source(s)
    - Interested objects/subscribers
  - **Do:**
    - Register/unregister subscribers
    - Notify subscribers of events

- **Subscriber/Observer**
  - **Know:**
    - Event types of interest
    - Subjects/Publishers
  - **Do:**
    - Register/unregister with publishers
    - Process notifications received

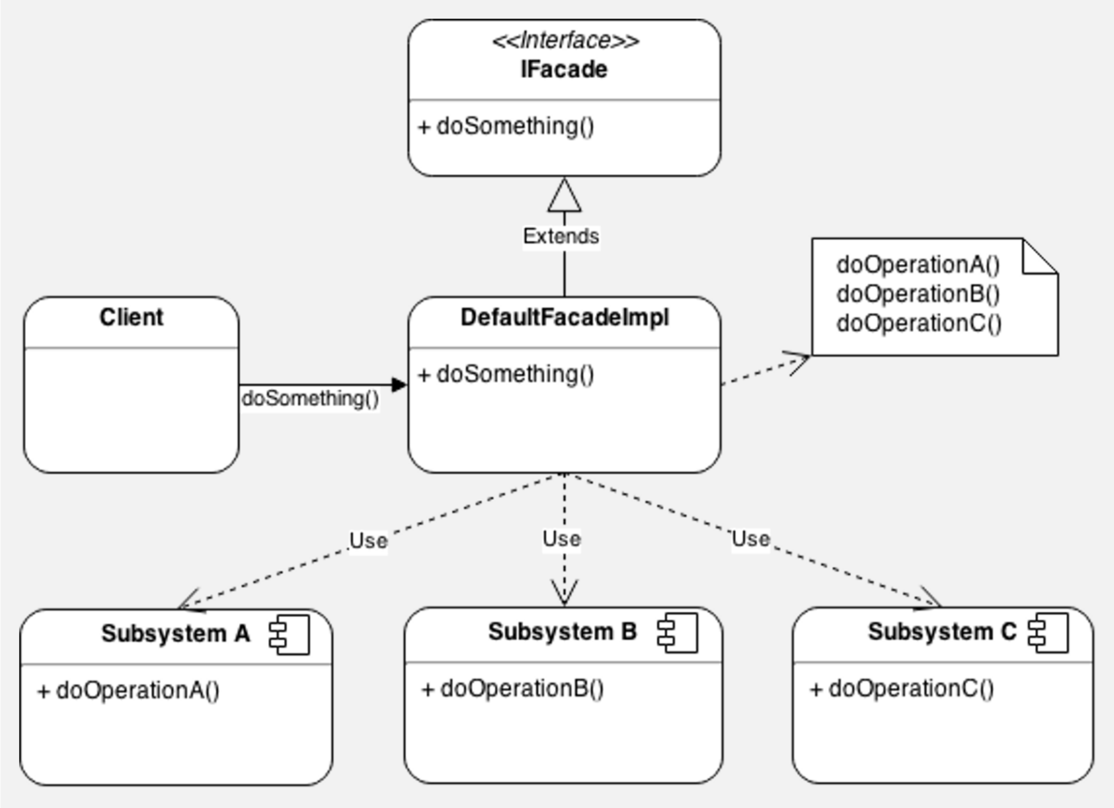
Class diagram with the Observer pattern



The Façade Pattern

- **Name**
  - Façade
- **Description**
  - Provide a unified interface to a set of interfaces in a subsystem
- **Problem description**
  - When you need a simplified interface to the overall functionality of a complex subsystem
- **Solution description**
  - A single `Facade class` implements a high-level interface for a subsystem by invoking the methods of the lower-level classes
- **Consequences**
  - Simplifies the use of a subsystem by providing higher-level methods
  - Enables lower-level classes to be restructured without changes to clients

Class diagram with the Façade pattern

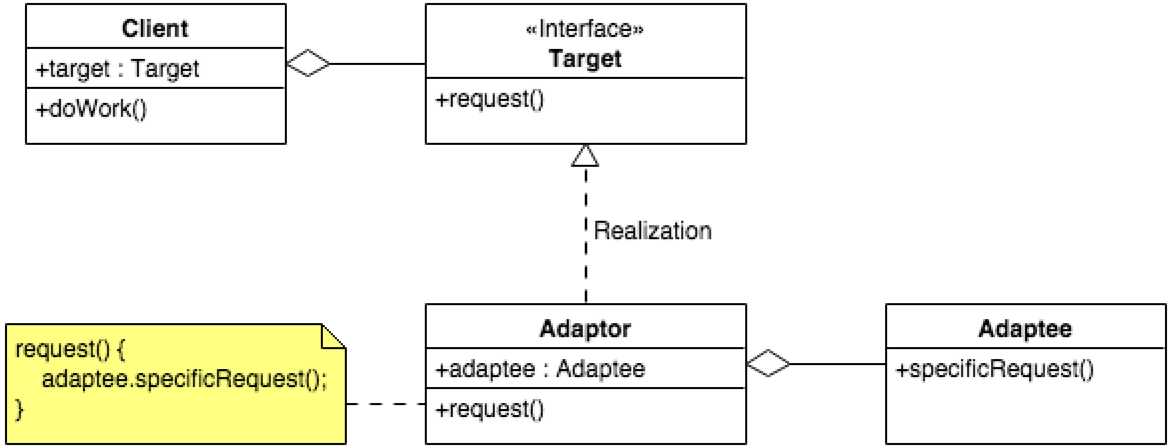


What should Façade Know? Do?

The Adapter Pattern

- **Name**
  - **Adapter**
- **Description**
  - Permit classes with disparate interfaces to work together by creating a common object by which they may communicate and interact
- **Problem description**
  - Incompatible interfaces
- **Solution description**
  - Create a wrapper that maps one interface to another
- **Consequences**
  - Neither interfaces has to change implementation
  - Decoupled execution

Class diagram with the Façade pattern



What should Adapter Know? Do?

When (not) to use Design Patterns

- Rule 1: **delay**
  - Understand the problem & solution first, then improve it
- Design patterns can increase or decrease understandability of code
  - Add indirection, increase code size
  - Improve modularity, separate concerns, ease description
- If your design or implementation has a problem, consider design patterns that address that problem

Core Design Pattern Relationships

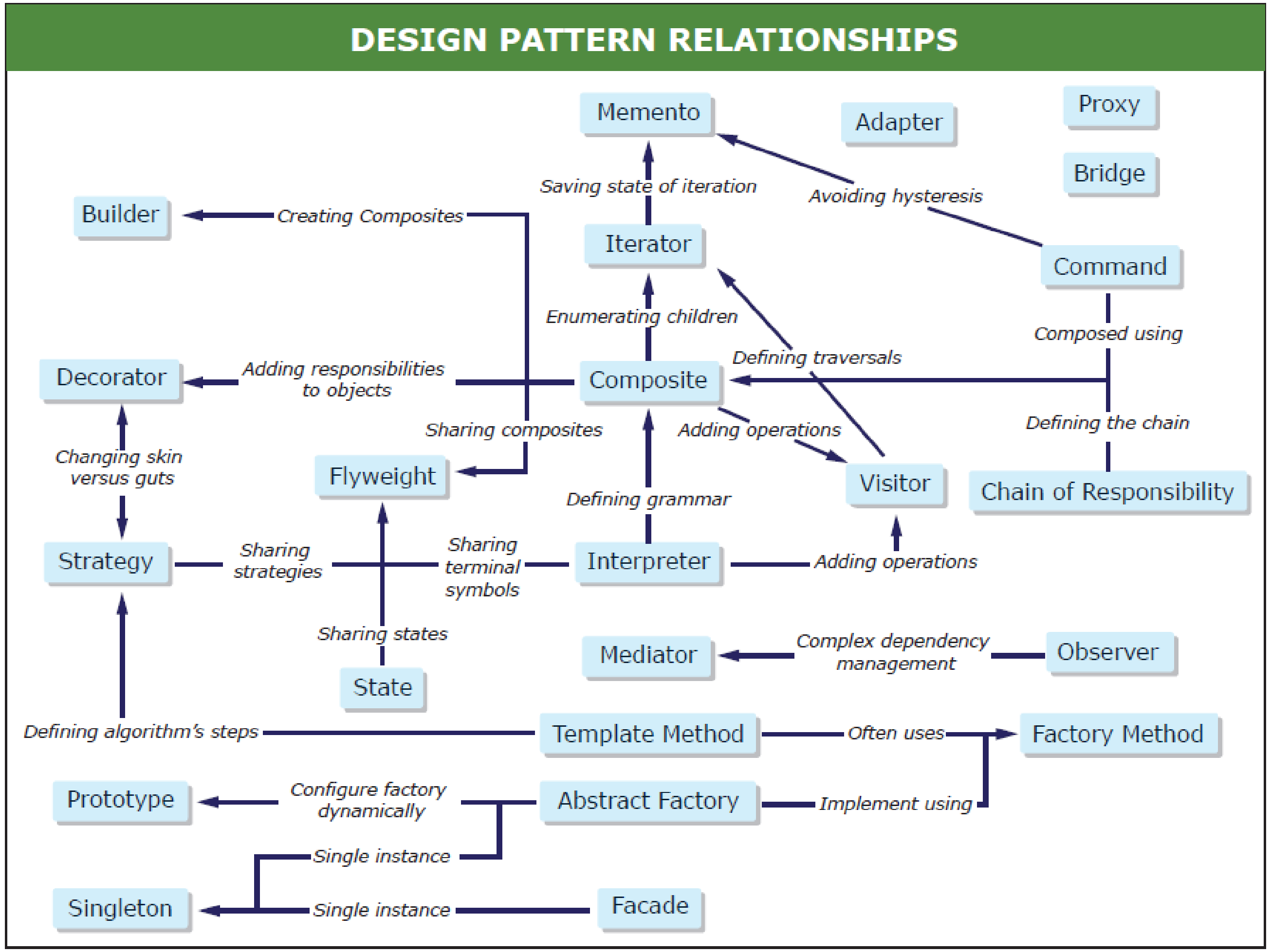


Image by MIT OpenCourseWare.
Source: Gamma, Erich, Richard Helm, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

Design Patterns Across Programming Languages

https://github.com/DovAmir/awesome-design-patterns

python-patterns

A collection of design patterns and idioms in Python.

Current Patterns

Creational Patterns:

Pattern	Description
abstract_factory	use a generic function with specific factories
borg	a singleton with shared-state among instances
builder	instead of using multiple constructors, builder object receives parameters and returns constructed objects
factory	delegate a specialized function/method to create instances
lazy_evaluation	lazily-evaluated property pattern in Python
pool	preinstantiate and maintain a group of instances of the same type
prototype	use a factory and clones of a prototype for new instances (if instantiation is expensive)

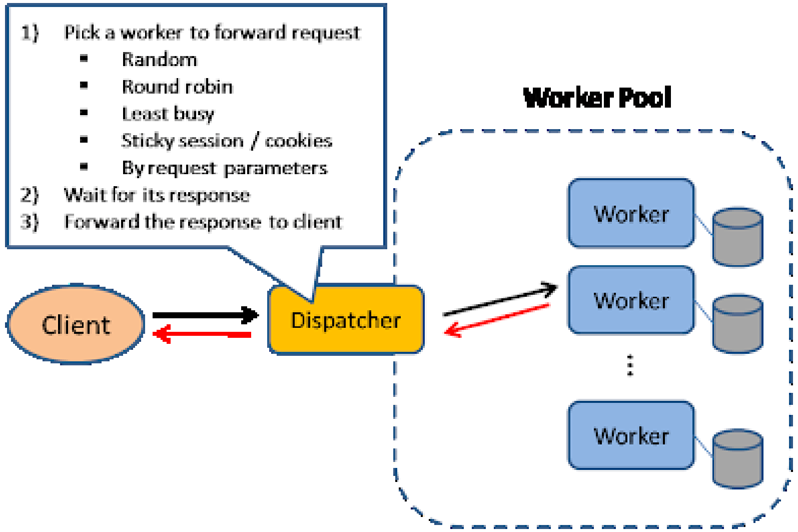
Structural Patterns:

Pattern	Description
3-tier	data<->business logic<->presentation separation (strict relationships)
adapter	adapt one interface to another using a white-list
bridge	a client-provider middleman to soften interface changes
composite	lets clients treat individual objects and compositions uniformly
decorator	wrap functionality with other functionality in order to affect outputs
facade	use one class as an API to a number of others
flyweight	transparently reuse existing instances of objects with similar/identical state
front_controller	single handler requests coming to the application
mvc	model<->view<->controller (non-strict relationships)
proxy	an object funnels operations to something else

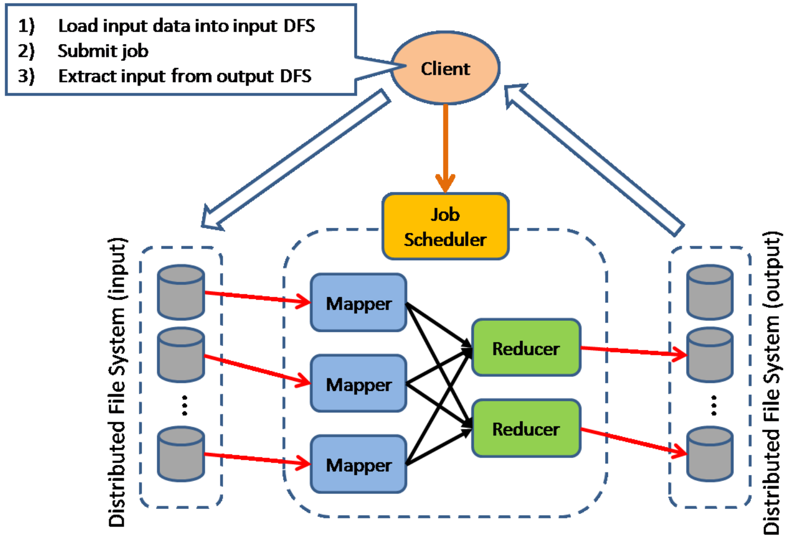
Scalable System Design Patterns

https://dzone.com/articles/scalable-system-design

Load Balancer

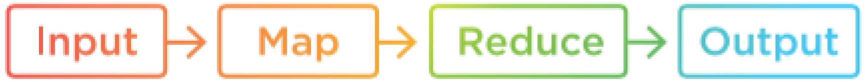


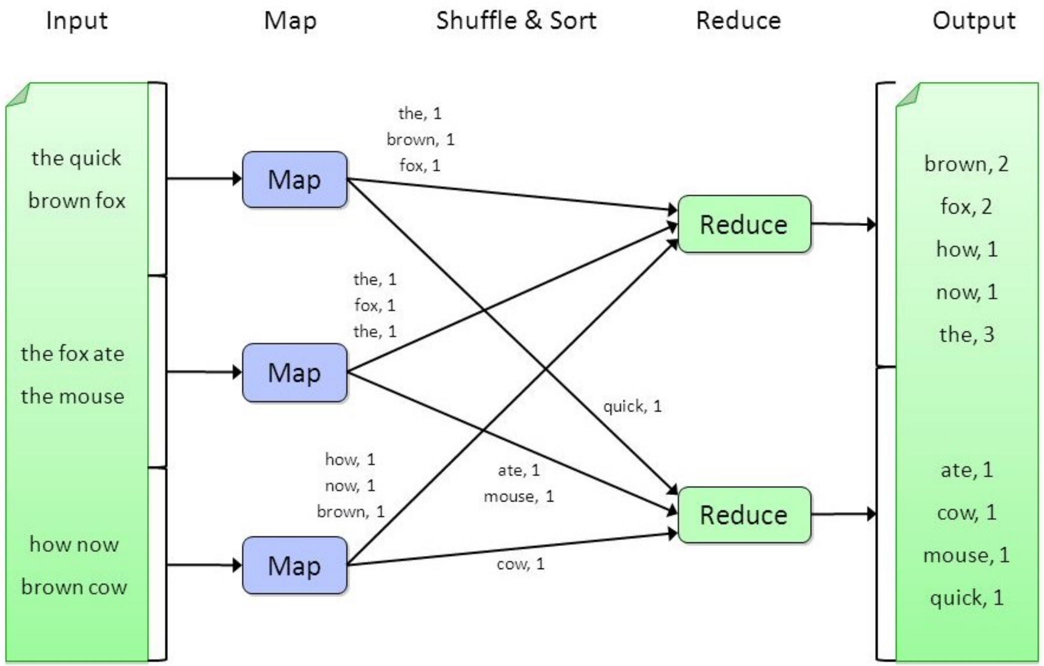
Map Reduce



The Map Reduce Summarization Pattern

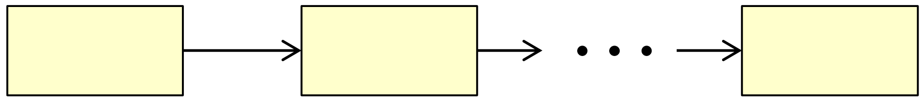
- **Map:** filter and convert input records (data) to tuples (key, value)
- **Reduce:** receive all tuples with the same key (key, list)
  1. Files are partitioned in parts that are distributed across cluster nodes
  2. Map tasks will read the data, and then build a key-value list that will be stored in intermediate files
  3. Key-value results are combined, resulting in new pairs of key-value that will be the input for Reduce tasks
  4. Reduce tasks will read the values, compute sum per key, and produce another list of key-value pairs to be combined into a final result



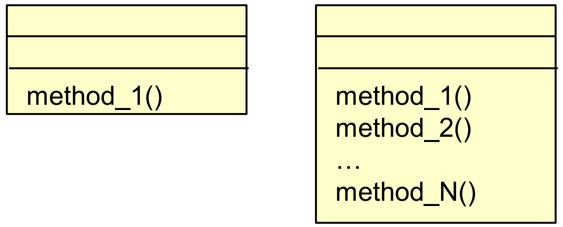


Characteristics of Good Designs

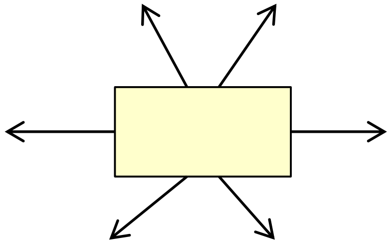
- Short communication chains between the objects



- Balanced workload across the objects



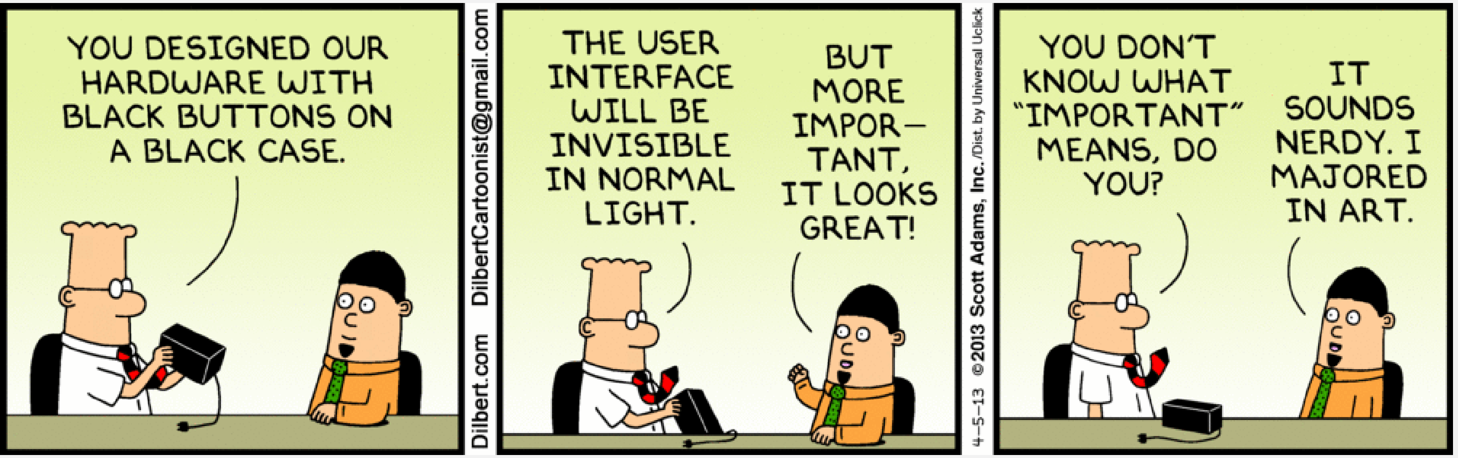
- Low degree of connectivity (associations) among the objects



What we covered so far

Sommerville's chapters 4, 5, 6, 7 (not comprehensively, use lectures are reference to content covered)

Design patterns are not explained in full in the book, so you may use online sources such as the links provided to you in the lectures



NEXT WEEK on SE – UX/UI

Disclaimer

Content is adapted from Ian Sommerville's book slides, William Y. Arms' lecture slides from Cornell University, and Kenneth M. Anderson's slides from the University of Colorado

Thank You

mervat.abuelkheir@guc.edu.eg