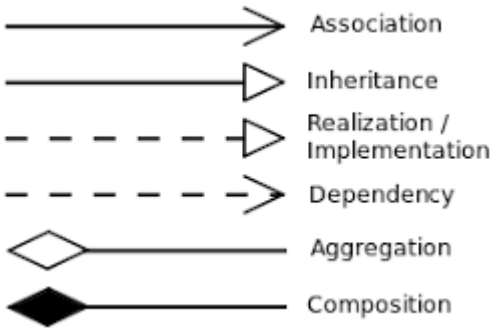
Object Oriented Analysis and Design Midterm

0. GENERAL:

CREATIONAL: Factory Method, Abstract Factory, Singleton, Builder, Prototype
BEHAVIORAL: Strategy, Observer, Command, Template Method, Iterator, State, Chain of Responsibility, Interpreter, Mediator, Memento, Visitor
STRUCTURAL: Decorator, Adapter, Facade, Composite, Proxy, Bridge, Flyweight
+CONCURRENY:

Dependency from A to B means that A uses B but indirectly (say by receiving instances of it and forwarding them to other objects).
Association from A to B means that A uses B directly, (for example by calling methods).
Aggregation from A to B means that B is part of A (semantically) but B can be shared and if A is deleted, B is not deleted. Note that this says nothing about how the "is part" is implemented.
Composition from A to B is like Aggregation, where B cannot be shared and if A is deleted, all of its aggregates (Bs) are deleted also.



Hata çözümleri:
1. Hatayi yok saymak
2. Hatayi haber vermek
3. Hatayi duzeltmek
4. Programi kapatmak

1. Strategy Pattern:

Design Principle: Identify the aspects of your application that vary and separate them from what stays the same.

Hint: Take what varies and “encapsulate” it so it won’t affect the rest of your code.

Design Principle: Program to an interface, not an implementation.

Programming to an implementation:
```
Dog d = new Dog();
d.bark();
```

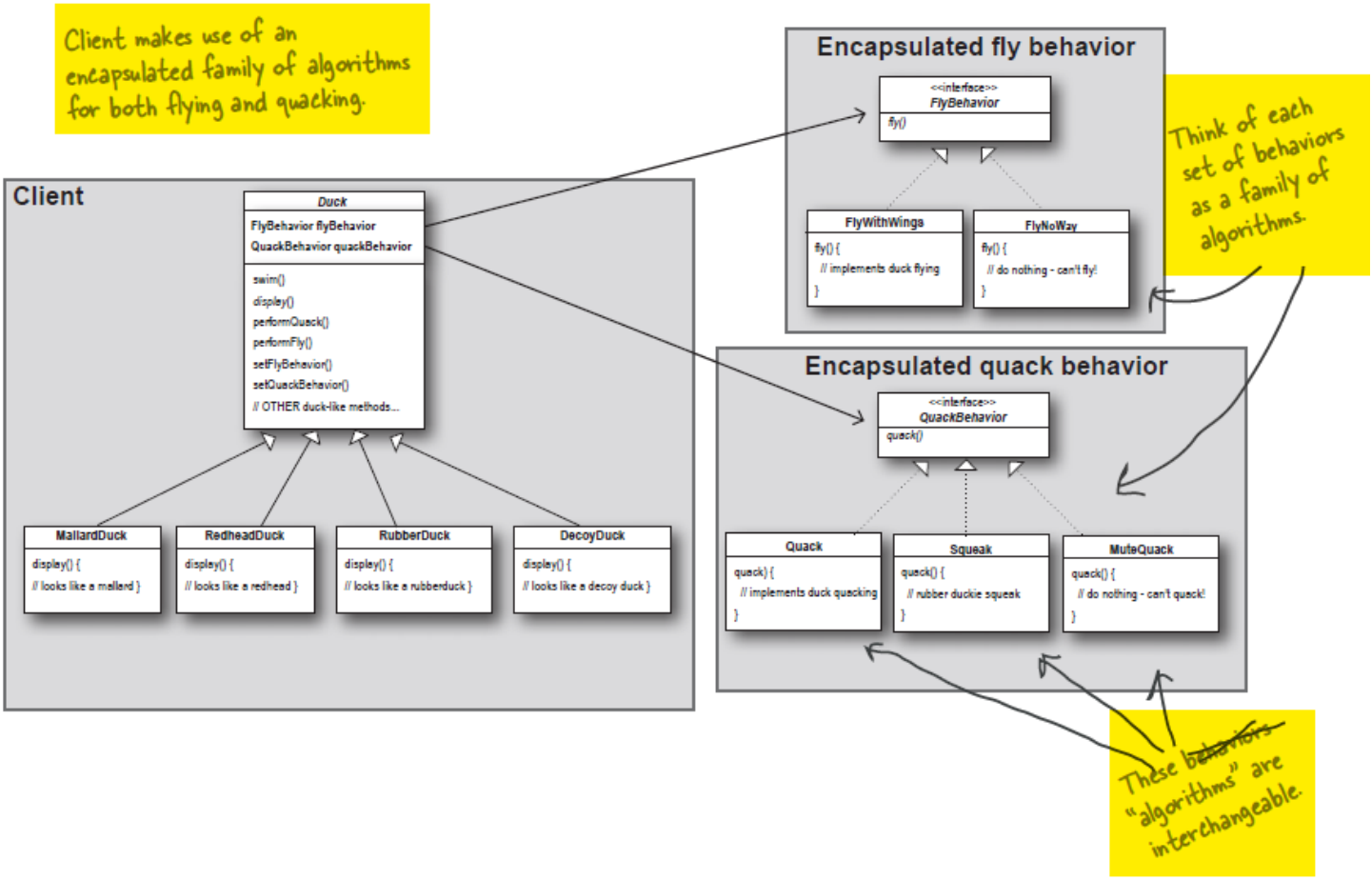
Programming to an interface/supertype:

Animal animal = new Dog();
animal.makeSound();

The best: Assign the concrete implementation object at runtime:
a = getAnimal();
a.makeSound();

Design Principle: Favor composition over inheritance.

Hint: HAS-A relation can be better than IS-A relation.



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.

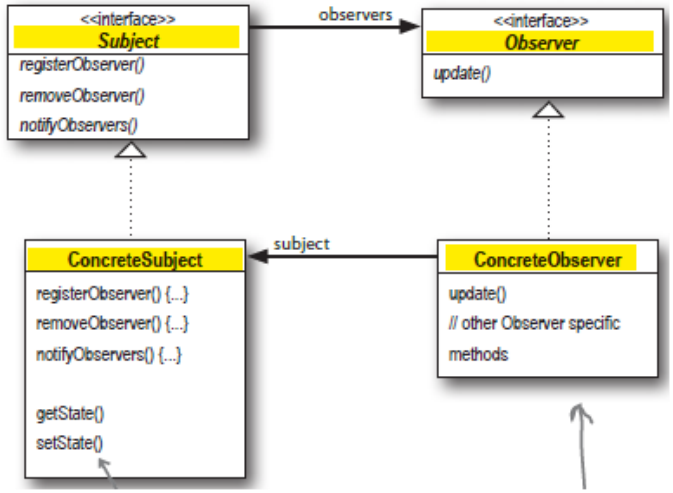
Bullet Points:

- Good OO designs are reusable, extensible and maintainable.
- Knowing the OO basics does not make you a good OO designer.
- Patterns are proven object-oriented experience.
- Patterns aren't invented, they are discovered.
- We often try to take what varies in a system and encapsulate it.
- Patterns provide a shared language that can maximize the value of your communication with other developers

- Most patterns and principles address issues of change in software.

2. Observer Pattern:

The Observer Pattern: Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.



Design Principle: Strive for loosely coupled designs between objects that interact. Loosely coupled designs are much more flexible and resilient to change.

The dark side of the java.util.Observable: Observable is a class, not an interface, and worse, it doesn't even implement an interface. Unfortunately, the java.util.Observable implementation has a number of problems that limit its usefulness and reuse.

Bullet Points:

- The Observer Pattern defines a one-to-many relationship between objects.
- Subjects(Observables) update Observers using a common interface.
- Observers are loosely coupled in that the Observable knows nothing about them, other than that they implement the Observer interface.
- Don't depent on a specific order of notification for your Observers.
- You can push or pull data from the Observable when using the pattern (pull is considered more "correct")
- Don't be afraid to create your own Observable implementation if needed.

3. Decorating Pattern:

Design Principle: Classes should be open for extension, but closed for modification. This principle is also known as **Open-Closed Principle**.

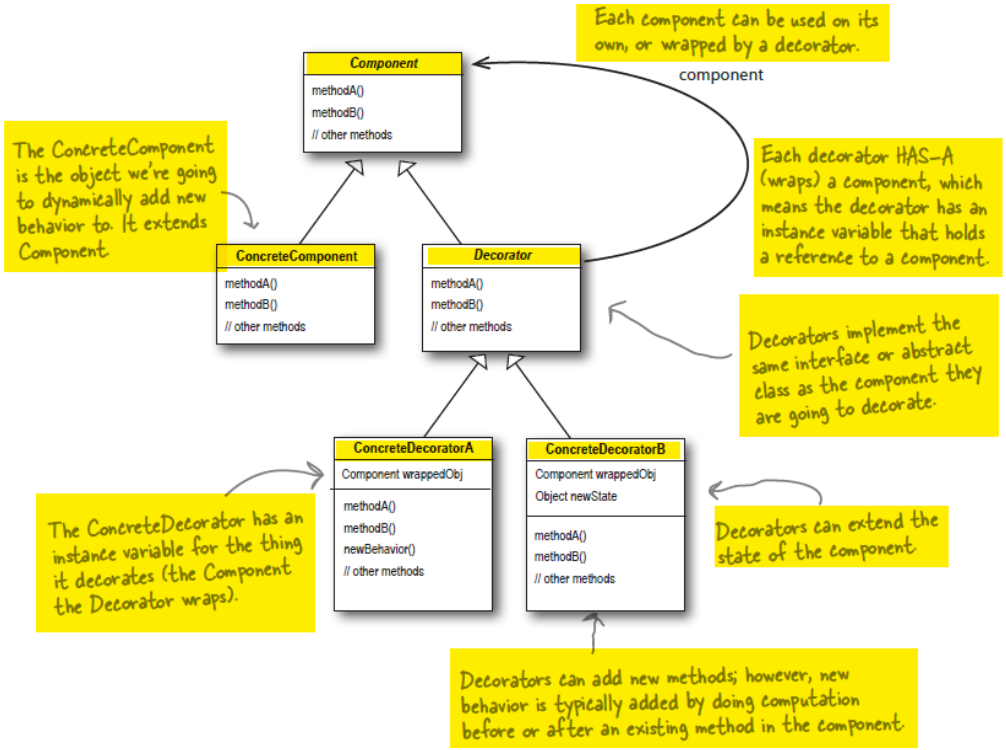
Hint: While it may seem like a contradiction, there are techniques for allowing code to be extended without direct modification.

Decorators have the same supertype as the objects they decorate.

Key Point:

The decorator adds its own behavior either before and/or after delegating to the object it decorates to do the rest of the job.

The Decorator Pattern: Attaches additional resposibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.



Bullet Points:

- In our designs we should allow behavior to be extended without the need to modify existing code.
- Composition and Delegation can often be used to add new behaviors at runtime.
- The Decorator Pattern provides an alternative to subclassing for extending behavior.
- Decorators can result in many small objects in our design, and overuse can be complex.
- Inheritance is one form of extension, but not necessarily the best way to achieve flexibility in our designs.

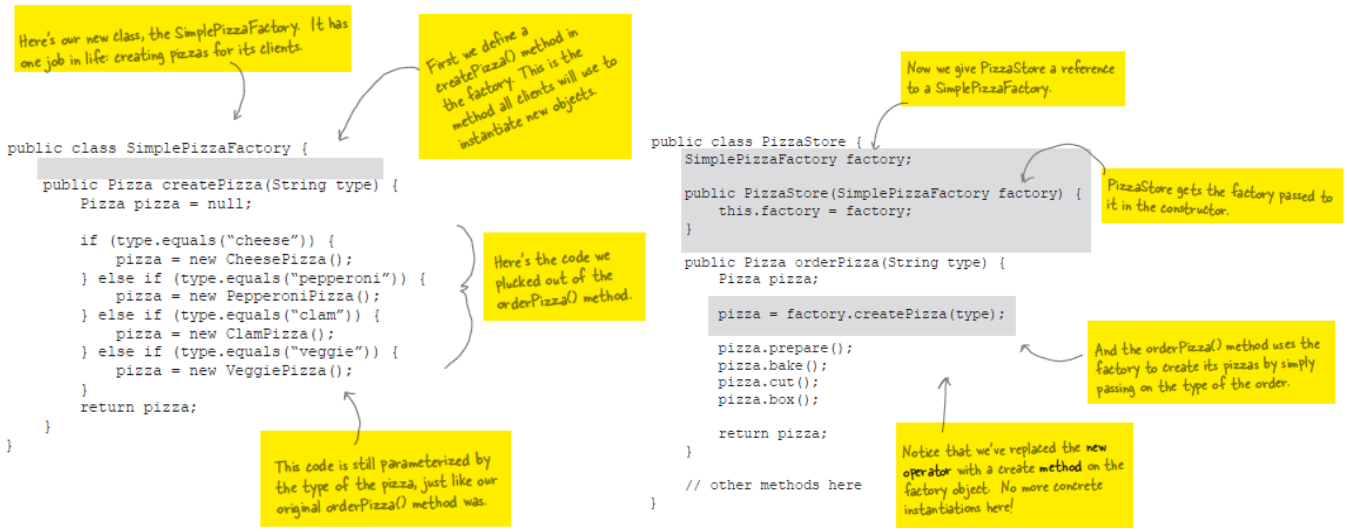
4. Factory Pattern:

A **simple factory** gives you a way to encapsulate object creation, but doesn't give you the flexibility of the Factory Method because there is no way to vary the products you're creating.

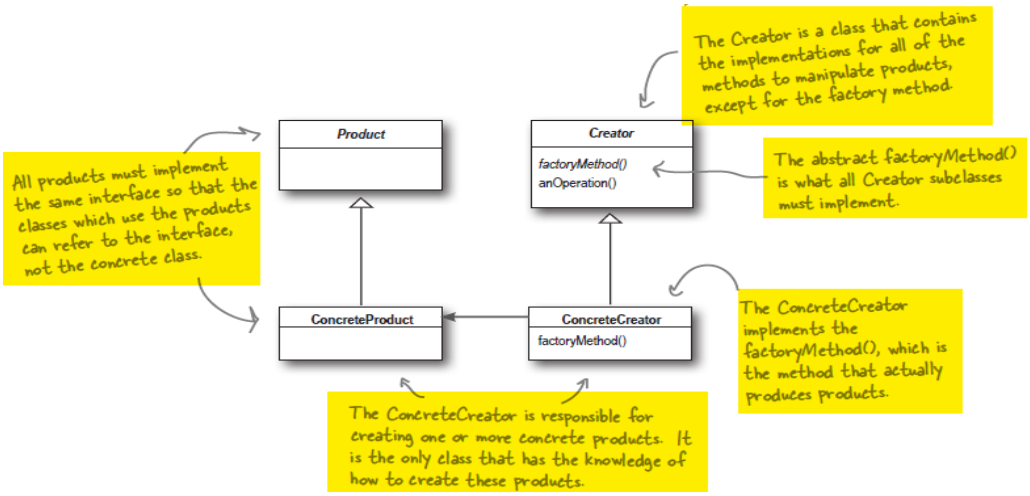
A **factory method** handles object creation and encapsulates it in a subclass. This decouples the client code in the superclass from the object creation code in the subclass.

A **factory method** is abstract so the subclasses are counted on to handle object creation.

The Factory Method Pattern: Defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

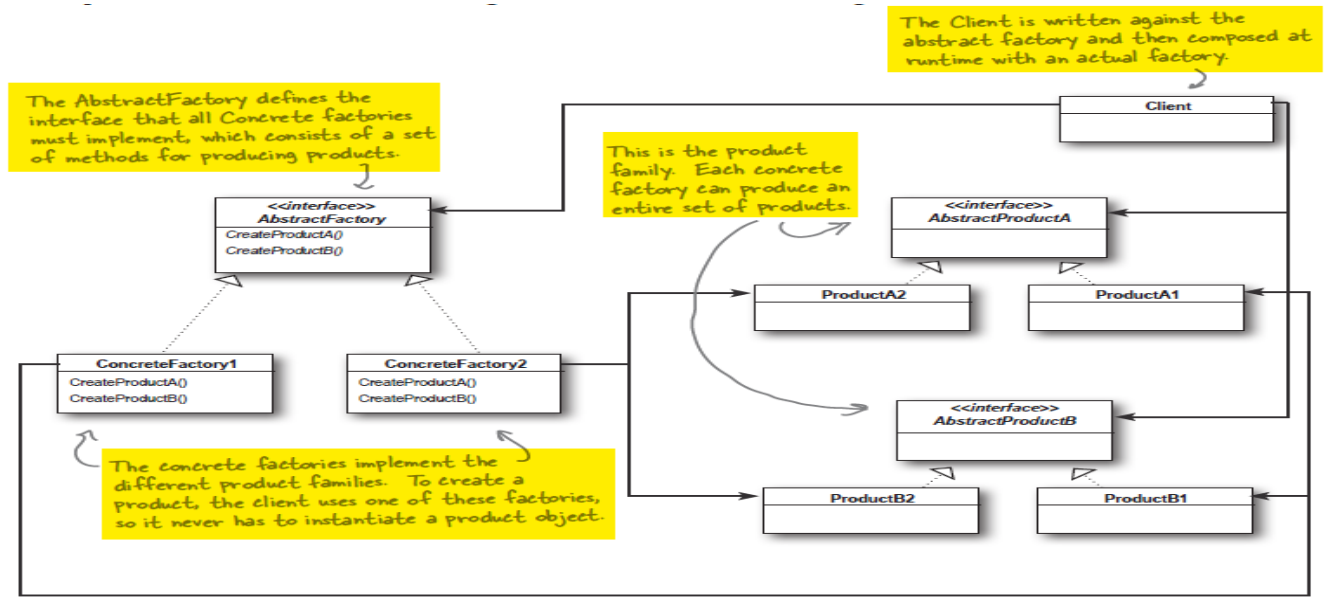


Factory method example



The Dependency Inversion Principle: Depend upon abstractions. Do not depend upon concrete classes.

The Abstract Factory Pattern: Provides an interface for creating families of related or dependent objects without specifying their concrete classes.



Bullet Points:

- All factories encapsulate object creation.
- Factory Method relies on inheritance: object creation is delegated to subclasses which implement the factory method to create objects.
- Abstract Factory relies on object composition: object creation is implemented in methods exposed in the factory interface.
- All factory patterns promote loose coupling by reducing the dependency of your application on concrete classes
- The intent of Factory Method is to allow a class to defer instantiation to its subclasses.
- The Dependency Inversion Principle guides us to avoid dependencies on concrete types and to strive for abstractions.

5. Singleton Pattern

The Singleton Pattern: Ensures a class has only one instance, and provides a global point of access to it.

```
Multithread save singleton
public class Singleton {
        private volatile static Singleton uniqueInstance;
        private Singleton() {}
        public static Singleton getInstance(){
                if(uniqueInstance == null){
                        synchronized(Singleton.class){
                                if(uniqueInstance == null)
                                        uniqueInstance = new Singleton();
                        }
                }
                return uniqueInstance;
        }
}
```

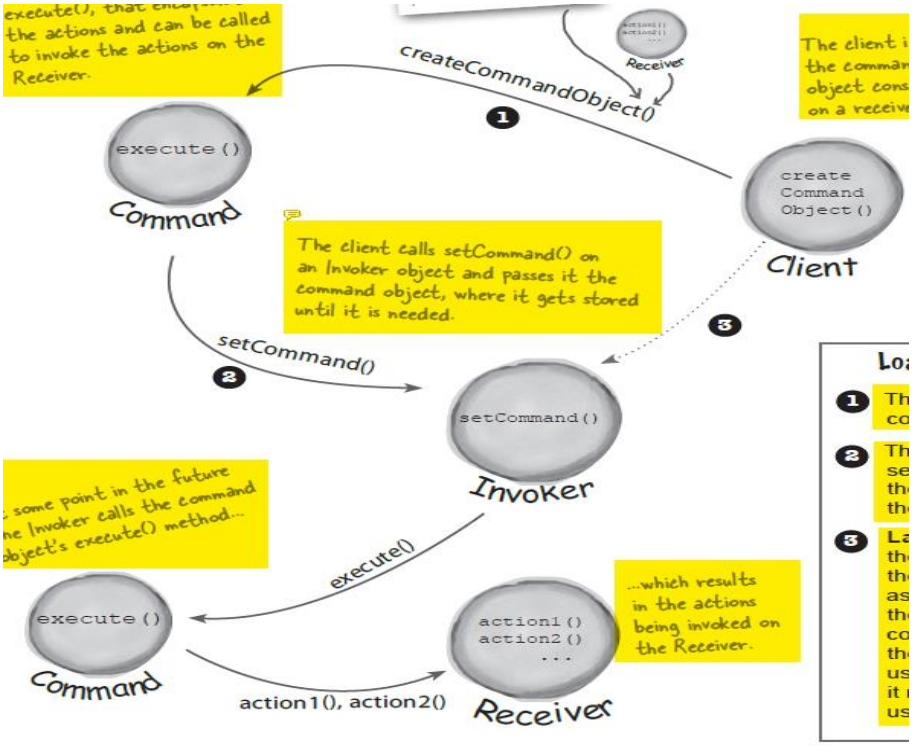
The Java **volatile** keyword is used to mark a Java variable as "being stored in main memory". More precisely that means, that every read of a volatile variable will be read from the main memory, and not from the CPU cache, and that every write to a volatile variable will be written to main memory, and not just to the CPU cache.

Rumors of Singletons being eaten by the garbage collectors are greatly exaggerated.

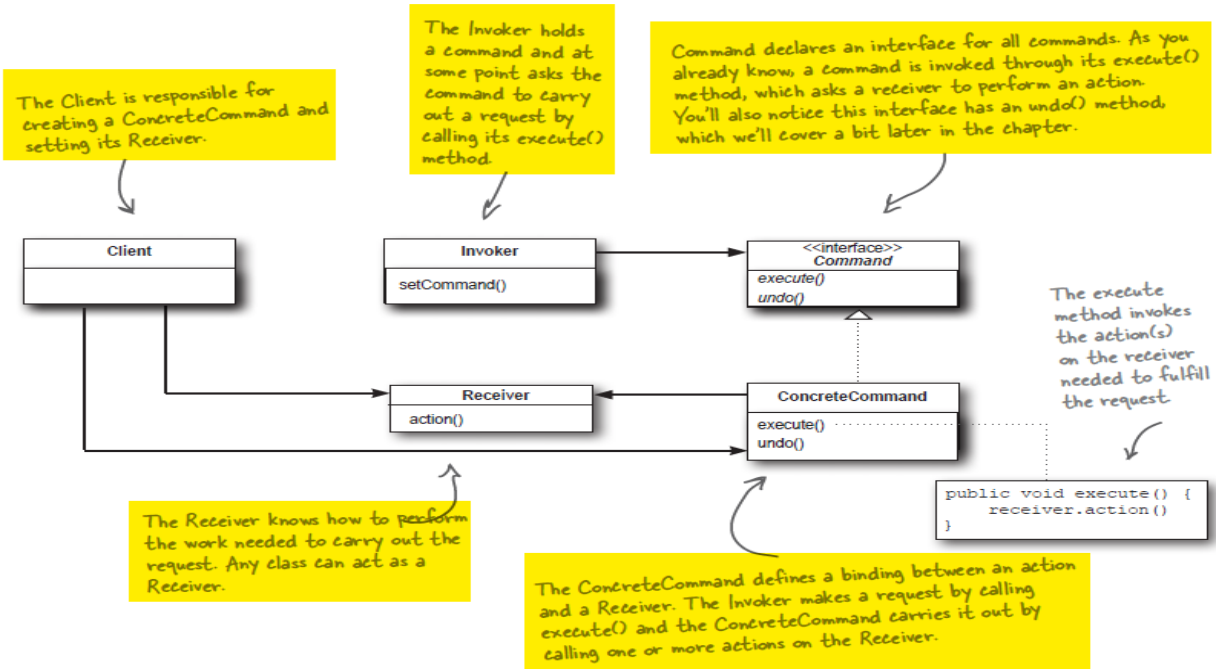
Bullet Points:

- The Singleton Pattern also provides a global access point to that instance.
- Be careful if you are using multiple class loaders; this could defeat the Singleton implementation and result in multiple instances.
- If you are using a JVM earlier than 1.2, you'll need to create a registry of Singletons to defeat the garbage collector.

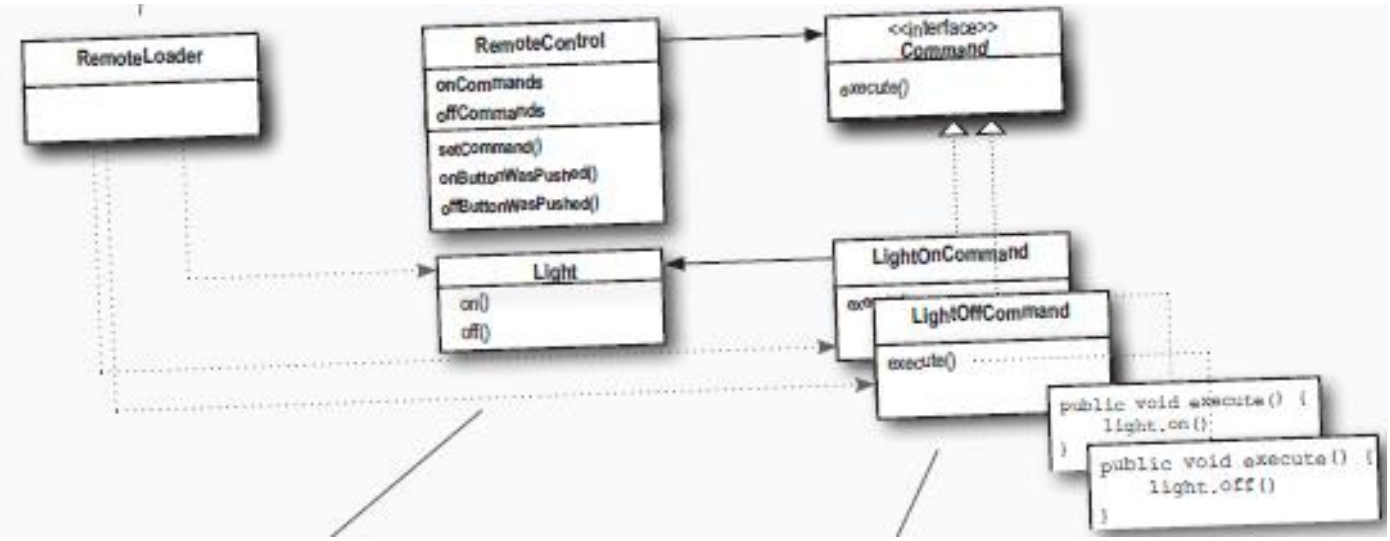
6. Command Pattern



The Command Pattern: Encapsulates a request as an object, thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations.



A **null object** is useful when you don't have a meaningful object to return, and yet you want to remove the responsibility for handling null from the client.

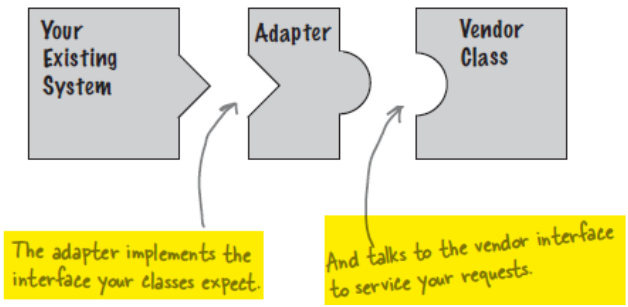


Bullet Points:

- **The Command Pattern decouples an object making a request from the one that knows how to perform it.**
- **Invokers can be parameterized with Commands, even dynamically at runtime.**
- **Commands may support undo by implementing an undo method that restores the object to its previous state before the execute() method was last called.**
- **Commands may also be used to implement logging and transactional systems.**

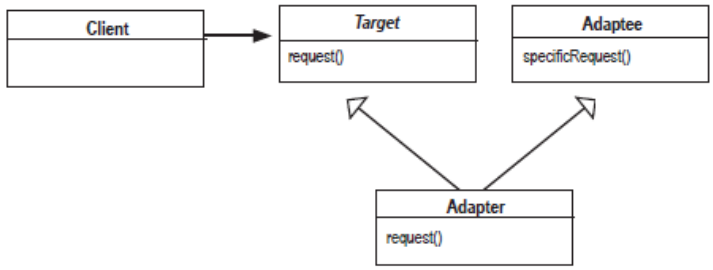
7. Adapter and Facade Patterns

Adapter: also know as wrapper or translator.

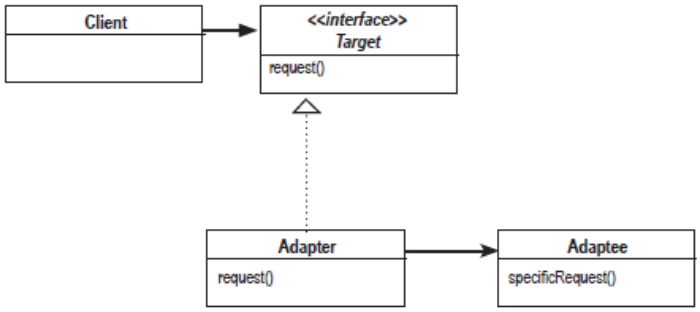


The Adapter Pattern: Converts the interface of a class into another interface the client expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Class Adapter



Object Adapter




```
public interface Duck {
    public void quack();
    public void fly();
}

public class MallardDuck implements Duck {
    public void quack() {
        System.out.println("Quack");
    }

    public void fly() {
        System.out.println("I'm flying");
    }
}
```

```
public interface Turkey {
    public void gobble();
    public void fly();
}
```

Turkeys don't quack, they gobble

Turkeys can fly, although they can only fly short distances.

```
public class WildTurkey implements Turkey {
    public void gobble() {
        System.out.println("Gobble gobble");
    }

    public void fly() {
        System.out.println("I'm flying a short distance");
    }
}
```

Here's a concrete implementation of Turkey, like Duck, it just prints out its actions.

```
public class TurkeyAdapter implements Duck {
    Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    public void quack() {
        turkey.gobble();
    }

    public void fly() {
        for(int i=0; i < 5; i++) {
            turkey.fly();
        }
    }
}
```

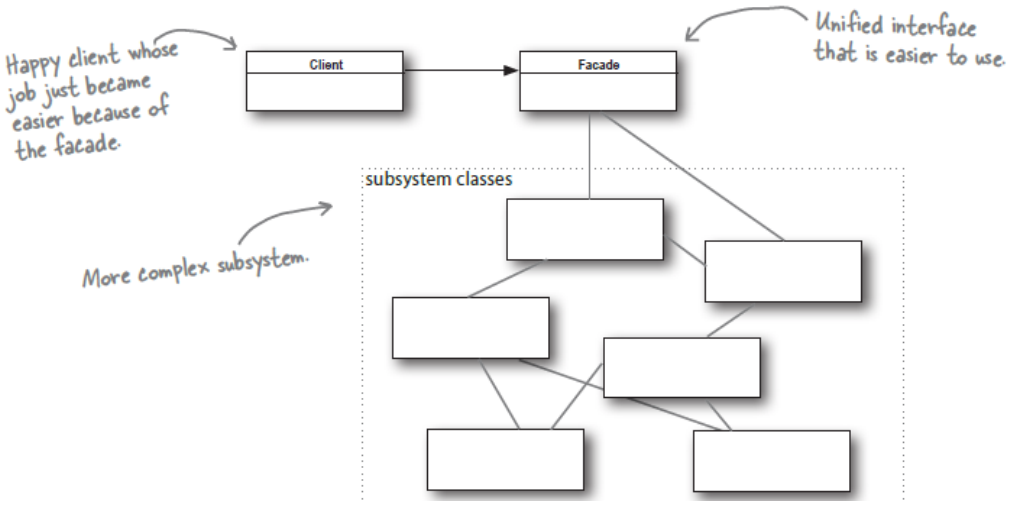
Next, we need to get a reference to the object that we are adapting; here we do that through the constructor.

Now we need to implement all the methods in the interface; the quack() translation between classes is easy: just call the gobble() method.

Even though both interfaces have a fly() method, Turkeys fly in short spurts – they can't do long-distance flying like ducks. To map between a Duck's fly() method and a Turkey's, we need to call the Turkey's fly() method five times to make up for it.

Facade: Only simplices an interface, it decouples a client from a subsystem of components.

The Facade Pattern: Provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.



Principle of Least Knowledge: Talk only to your immediate friends.

Hint: This principle prevents us from creating designs that have a large number of classes coupled together so that changes in one part of the system cascade to other parts.

Without the Principle

```
public float getTemp() {
    Thermometer thermometer = station.getThermometer();
    return thermometer.getTemperature();
}
```

Here we get the thermometer object from the station and then call the getTemperature() method ourselves.

With the Principle

```
public float getTemp() {
    return station.getTemperature();
}
```

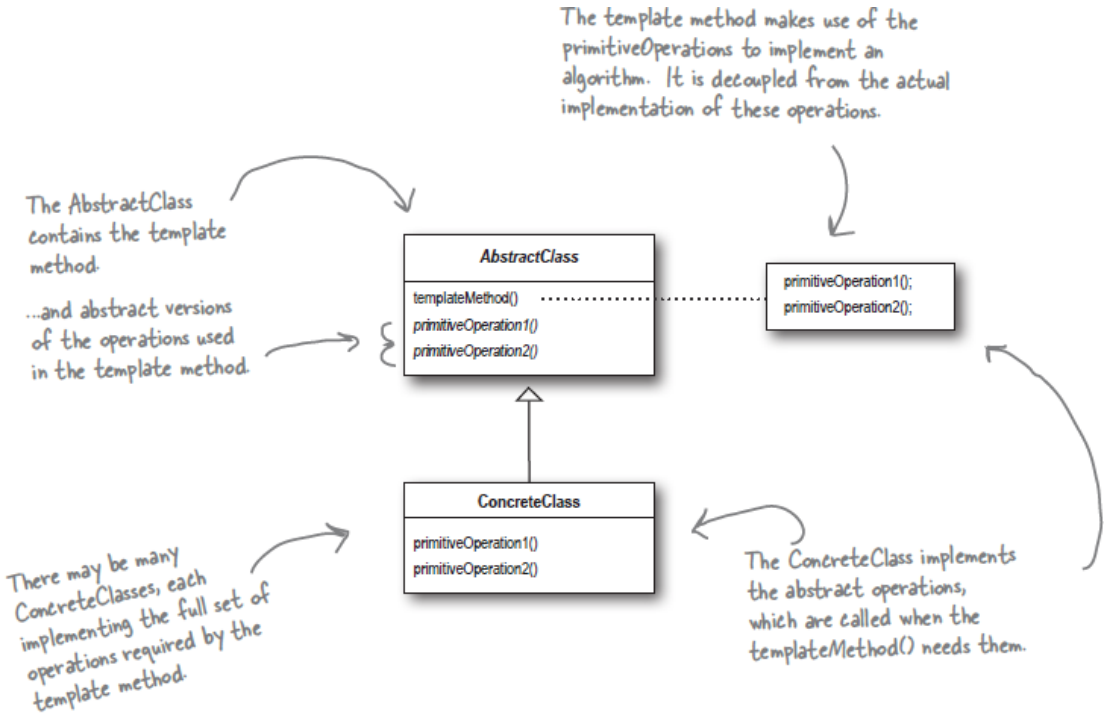
When we apply the principle, we add a method to the Station class that makes the request to the thermometer for us. This reduces the number of classes we're dependent on.

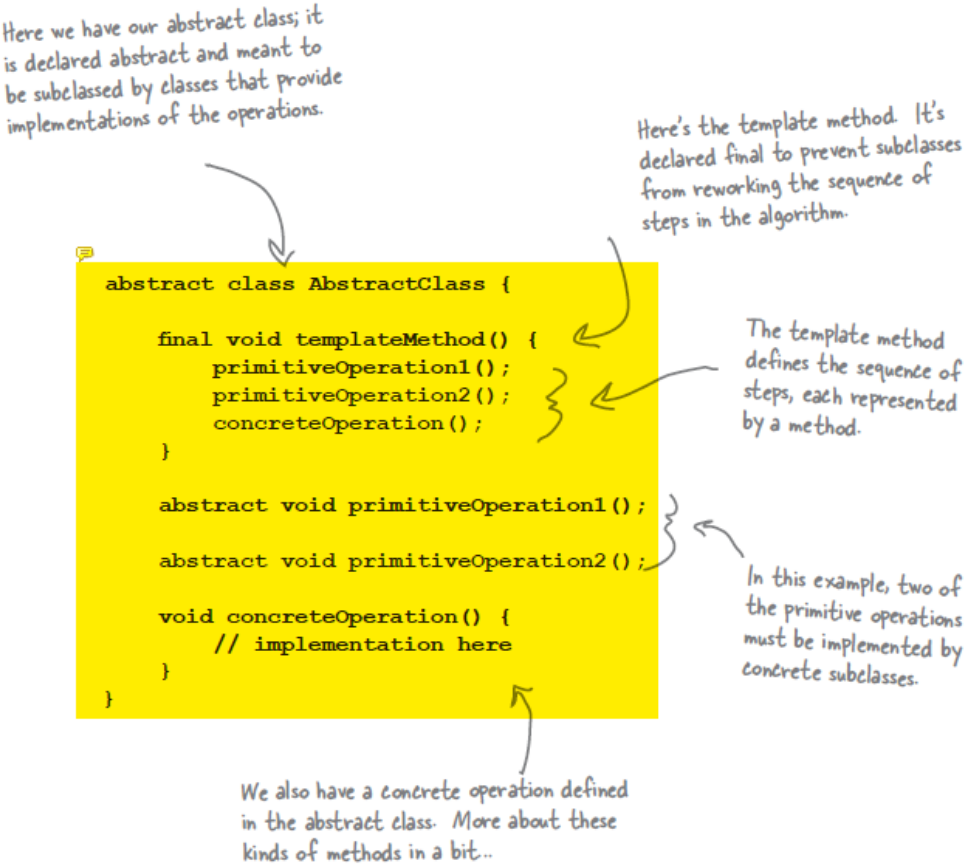
Bullet Points:

- **When you need to use an existing class and its interface is not the one you need, use an adapter.**
- **When you need to simplify and unify a large interface or complex set of interfaces, use a facade.**
- **An adapter changes an interface into one a client expects.**
- **A facade decouples a client from a complex subsystem.**
- **Implementing an adapter may require little work or a great deal of work depending on the size and complexity of the target interface.**
- **Implementing a facade requires that we compose the facade with its subsystem and use delegation to perform the work of the facade.**
- **An adapter wraps an object to change its interface, a decorator wraps an object to add new behaviors and responsibilities, and a facade "wraps" a set of objects to simplify.**
- **There are two forms of the**
- **Adapter Pattern: object and class adapters. Class adapters require multiple inheritance.**
- **You can implement more than one facade for a subsystem.**

8. Template Method Pattern

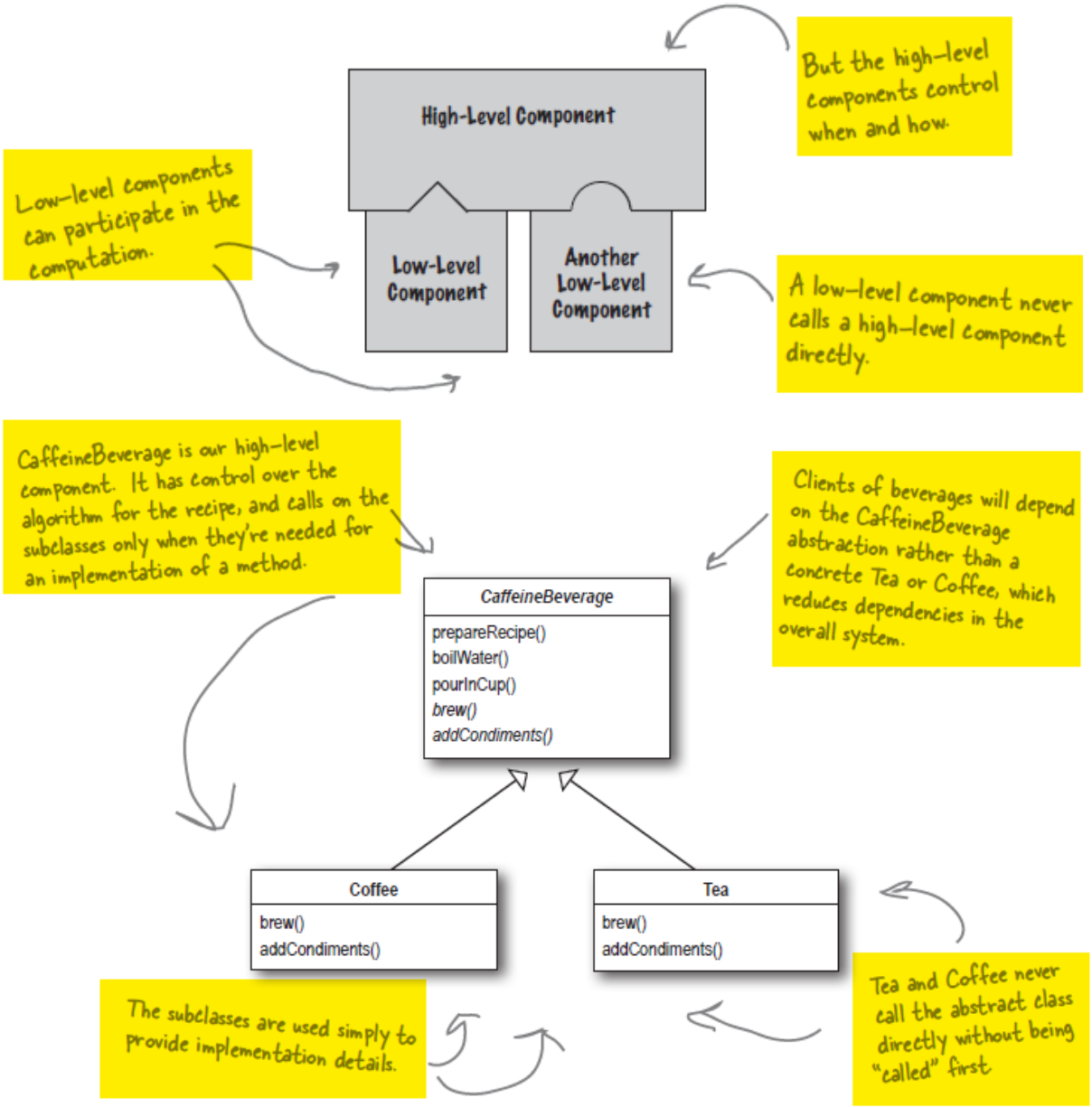
The Template Pattern: Defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.





void hook(){} method can be added and called from the end of templateMethod to give more freedom to the user to add a desired action at the end of the template method.

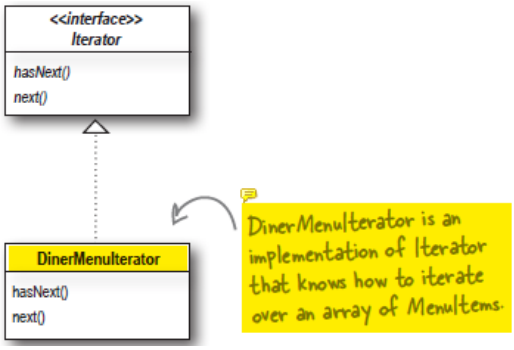
The Hollywood Principle: Don't call us, we will call you.



Bullet Points:

- **A "template method" defines the steps of an algorithm, deferring to subclasses for the implementation of those steps.**
- **The template method's abstract class may define concrete methods, abstract methods and hooks.**
- **Hooks are methods that do nothing or default behavior in the abstract class, but may be overridden in the subclass.**
- **To prevent subclasses from changing the algorithm in the template method, declare the template method as final.**
- **The Strategy and Template Method Patterns both encapsulate algorithms, one by inheritance and one by composition.**
- **The Factory Method is a specialization of Template Method.**

9. Iterator and Composite Patterns



And now we need to implement a concrete Iterator that works for the Diner menu:

We implement the Iterator interface.

```
public class DinerMenuIterator implements Iterator {
    MenuItem[] items;
    int position = 0;

    public DinerMenuIterator(MenuItem[] items) {
        this.items = items;
    }

    public Object next() {
        MenuItem menuItem = items[position];
        position = position + 1;
        return menuItem;
    }

    public boolean hasNext() {
        if (position >= items.length || items[position] == null) {
            return false;
        } else {
            return true;
        }
    }
}
```

position maintains the current position of the iteration over the array.

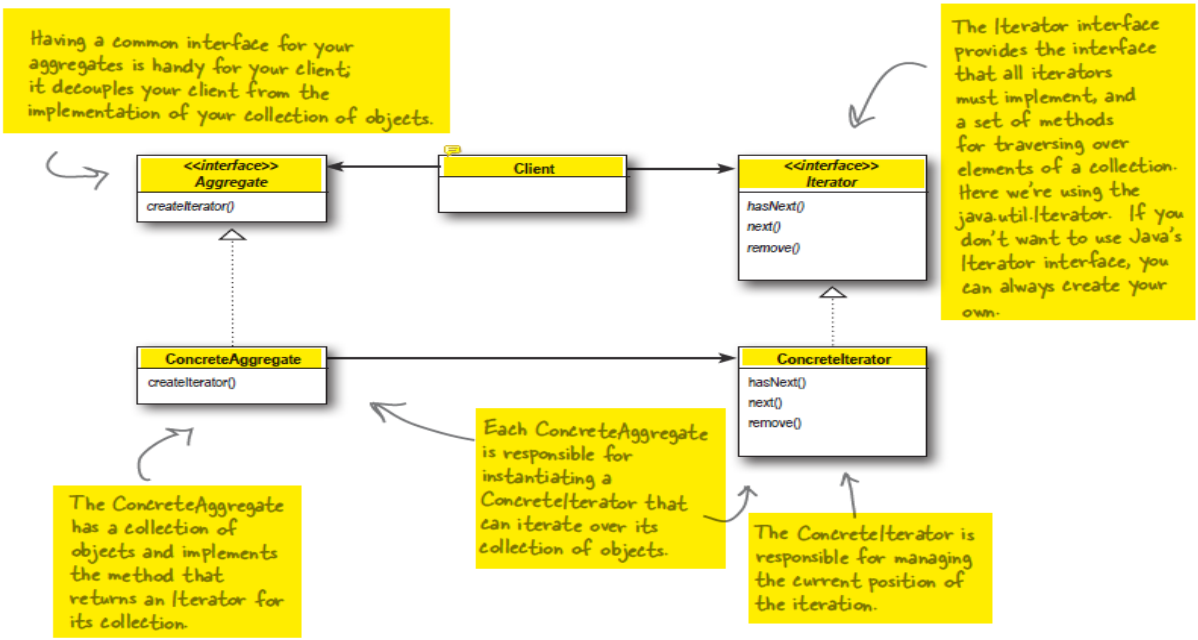
The constructor takes the array of menu items we are going to iterate over.

The next() method returns the next item in the array and increments the position.

The hasNext() method checks to see if we've seen all the elements of the array and returns true if there are more to iterate through.

Because the diner chef went ahead and allocated a max sized array, we need to check not only if we are at the end of the array, but also if the next item is null, which indicates there are no more items.

The Iterator Pattern: Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

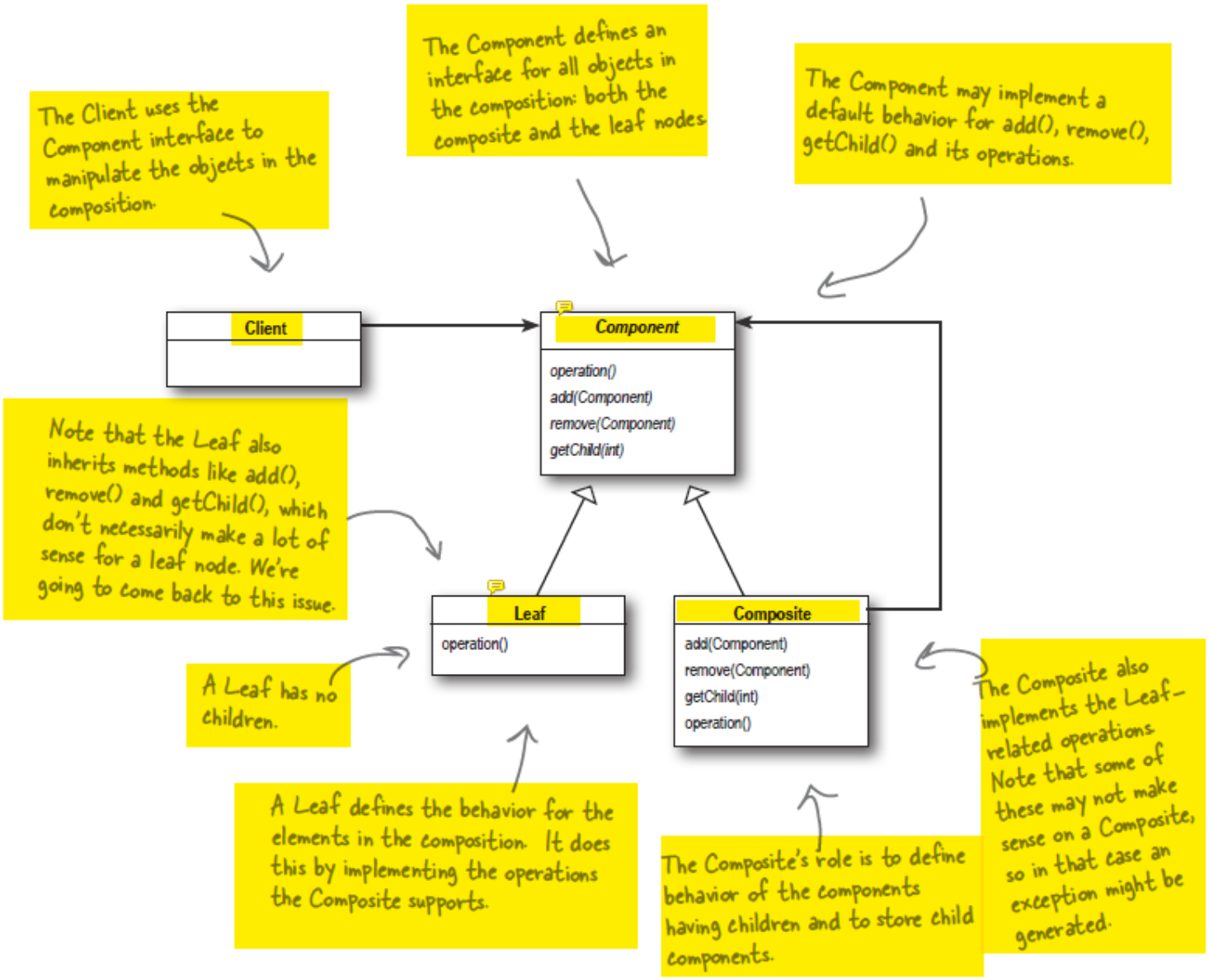


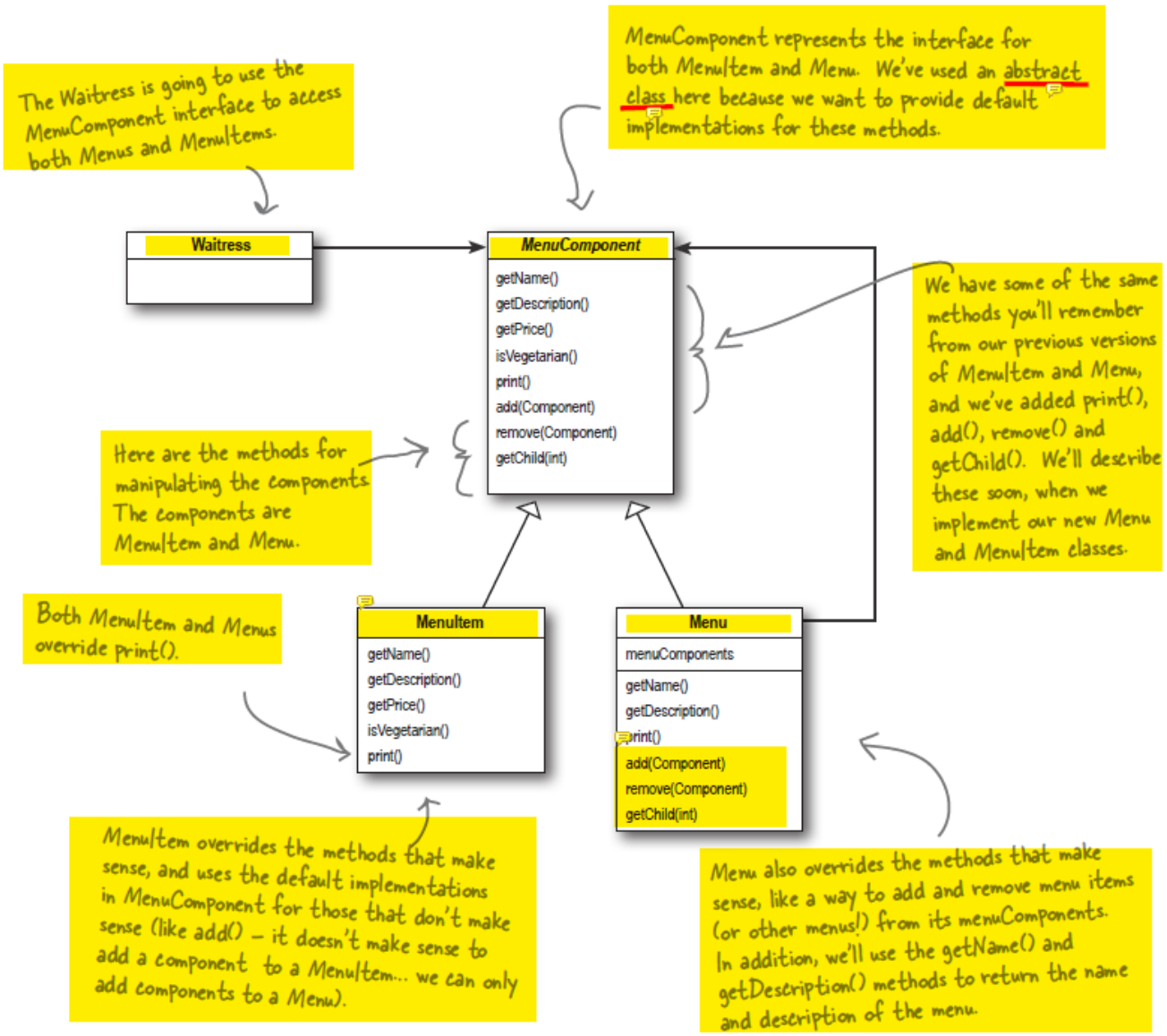
Design Principle: A class should have only one reason to change.

Hint: Every responsibility of a class is an area of potential change. More than one responsibility means more than one area of change.

Cohesion is a term you'll hear used as a measure of how closely a class or a module supports a single purpose or responsibility. Cohesion is a more general concept than the Single Responsibility Principle, but the two are closely related. Classes that adhere to the principle tend to have high cohesion and are more maintainable than classes that take on multiple responsibilities and have low cohesion.

The Composite Pattern: Allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.





Bullet Points:

- **An Iterator allows access to an aggregate's elements without exposing its internal structure.**
- **When using an Iterator, we relieve the aggregate of the responsibility of supporting operations for traversing its data.**
- **We should strive to assign only one responsibility to each class.**
- **The Composite Pattern provides a structure to hold both individual objects and composites.**
- **The Composite Pattern allows clients to treat composites and individual objects uniformly.**
- **A Component is any object in a Composite structure. Components may be other composites or leaf nodes.**
- **There are many design tradeoffs in implementing Composite. You need to balance transparency and safety with your needs.**

The Composite Iterator

The CompositeIterator is a SERIOUS iterator. It's got the job of iterating over the MenuItems in the component, and of making sure all the child Menus (and child child Menus, and so on) are included.

Here's the code. Watch out, this isn't a lot of code, but it can be a little mind bending. Just repeat to yourself as you go through it "recursion is my friend, recursion is my friend."

WATCH OUT: RECURSION ZONE AHEAD

```
import java.util.*;

public class CompositeIterator implements Iterator {
    Stack stack = new Stack();

    public CompositeIterator(Iterator iterator) {
        stack.push(iterator);
    }

    public Object next() {
        if (hasNext()) {
            Iterator iterator = (Iterator) stack.peek();
            MenuComponent component = (MenuComponent) iterator.next();
            if (component instanceof Menu) {
                stack.push(component.createIterator());
            }
            return component;
        } else {
            return null;
        }
    }

    public boolean hasNext() {
        if (stack.empty()) {
            return false;
        } else {
            Iterator iterator = (Iterator) stack.peek();
            if (!iterator.hasNext()) {
                stack.pop();
                return hasNext();
            } else {
                return true;
            }
        }
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

Like all iterators, we're implementing the java.util.Iterator interface.

The iterator of the top level composite we're going to iterate over is passed in. We throw that in a stack data structure.

Okay, when the client wants to get the next element we first make sure there is one by calling hasNext()...

If there is a next element, we get the current iterator off the stack and get its next element.

If that element is a menu, we have another composite that needs to be included in the iteration, so we throw it on the stack. In either case, we return the component.

To see if there is a next element, we check to see if the stack is empty; if so, there isn't.

Otherwise, we get the iterator off the top of the stack and see if it has a next element. If it doesn't we pop it off the stack and call hasNext() recursively.

Otherwise there is a next element and we return true.

We're not supporting remove, just traversal.