Software Design Patterns

Nat. Thomas
Software Engineering
Telephonics Inc.
Long Island IEEE Lecture Series

Software Design Patterns Agenda

- What are they?
- Why are they important?
- Pattern Taxonomy
- GoF – Gang of Four – The foundation
- List of original patterns
- Brief Unified Modeling Language (UML) overview
- Strategy example
- Singleton example
- Observer example
- Reference Material (books, web sites)

What Are They?

- Definitions
  - GoF – They are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.
  - “Head First Design Patterns” – someone has already solved your problem
  - Bruce Eckel states (from “Thinking in Patterns”) - Probably the most important step forward in object-oriented design - a pattern embodies a complete idea within a program, and thus it can sometimes appear at the analysis phase or high-level design phase.

Why are they Important?

- You're exploiting the wisdom and knowledge of lessons learned by other developers and designers who have been down the path you are about to travel.
- Accommodates change – software will always be changing so the design should expect and deal with it.
- Change - the one constant in software development.
- No Silver Bullet – refer to article No Silver Bullet: Essence and Accidents of Software Engineering by Frederick P. Brooks, Jr.

Pattern Taxonomy

- Bruce Eckel says watch out for pollution of the term because of it being associated with “good OOD”. Defines 4 categories:
  - **Idiom** - how we write code in a particular language to do a particular type of thing. This could be something as common as the way you code the process of stepping through an array in C (and not running off the end).
  - **Specific Design**: the solution that we come up with to solve a particular problem. This might be a clever design, but it makes no attempt to be general.
  - **Standard Design**: a way to solve this *kind* of problem. A design that has become more general, typically through reuse.
  - **Design Pattern**: how to solve an entire class of similar problem. This usually only appears after applying a standard design a number of times, and then seeing a common pattern throughout these applications.

Pattern Taxonomy Con't

- Each category is fine for the problem being solved. Don't waste time trying to solve *every* problem with a design pattern.

Gang of Four

- Book – “Design Patterns – Elements of Reusable Object-Oriented Software” by Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides
- Released 1995
- Describes 23 patterns in 3 categories (Creational, Structural & Behavioral)

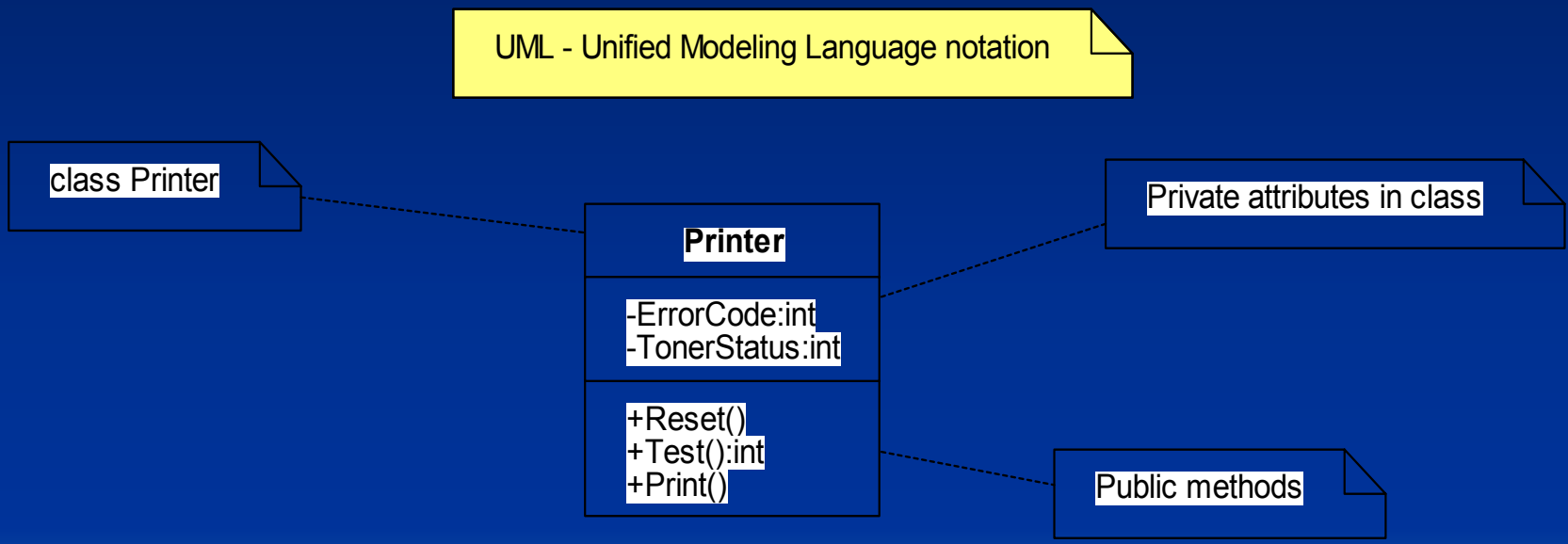
Design Pattern Space

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adaptor(class)	Interpreter Template method
	Object	Abstract Factory Builder Prototype Singleton	Adaptor(object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Unified Modeling Language

- All examples are illustrated using UML notation.
- GoF referred to using something called OMT (Object Modeling Technique) which heavily influenced the present day UML.

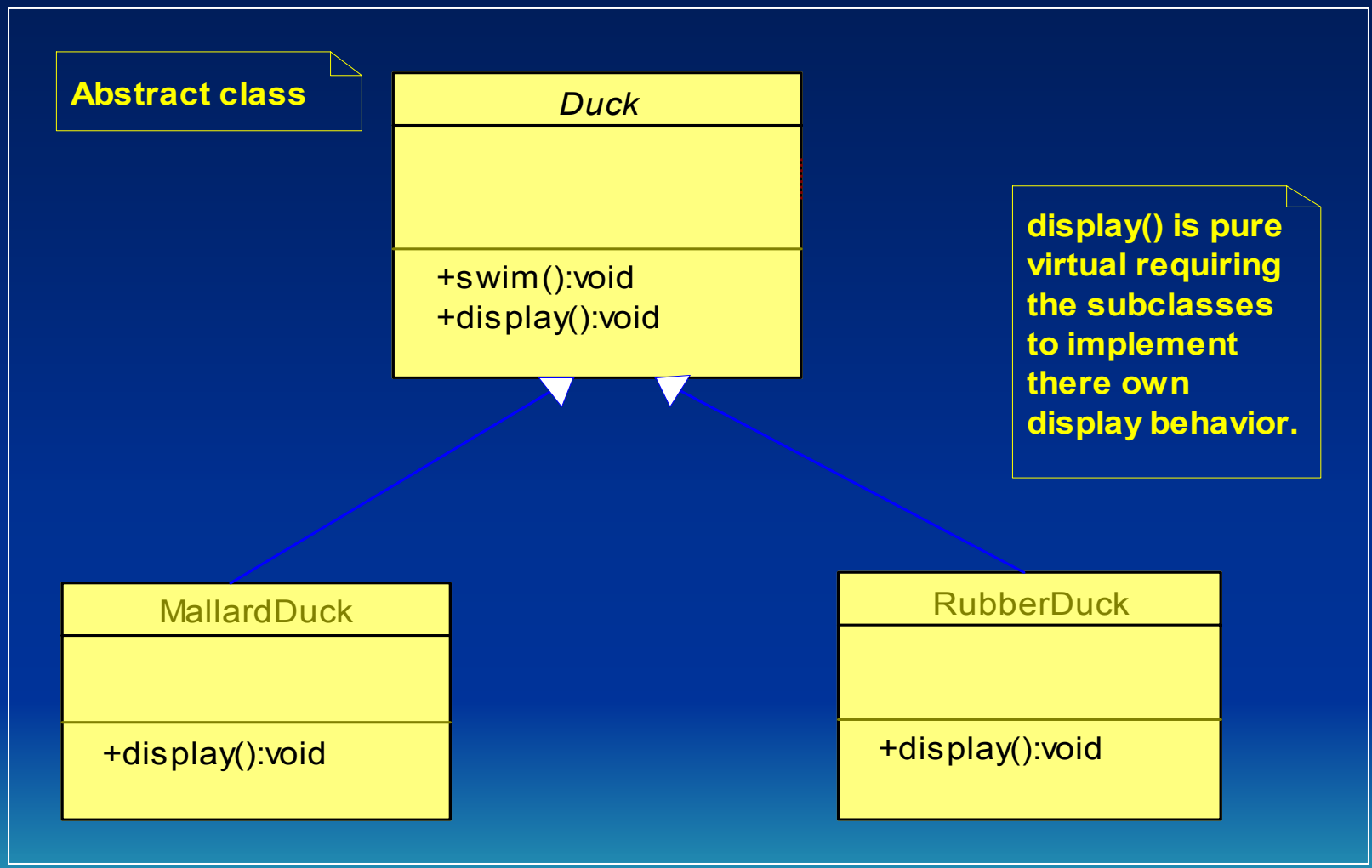
Simple UML example



Duck Pond Simulation Design - 1

- Joe works for a company that makes a highly successful duck pond simulation game.
- The initial designers used OO techniques and created a Duck superclass from which all other ducks inherited.

Duck Pond Simulation Design - 2



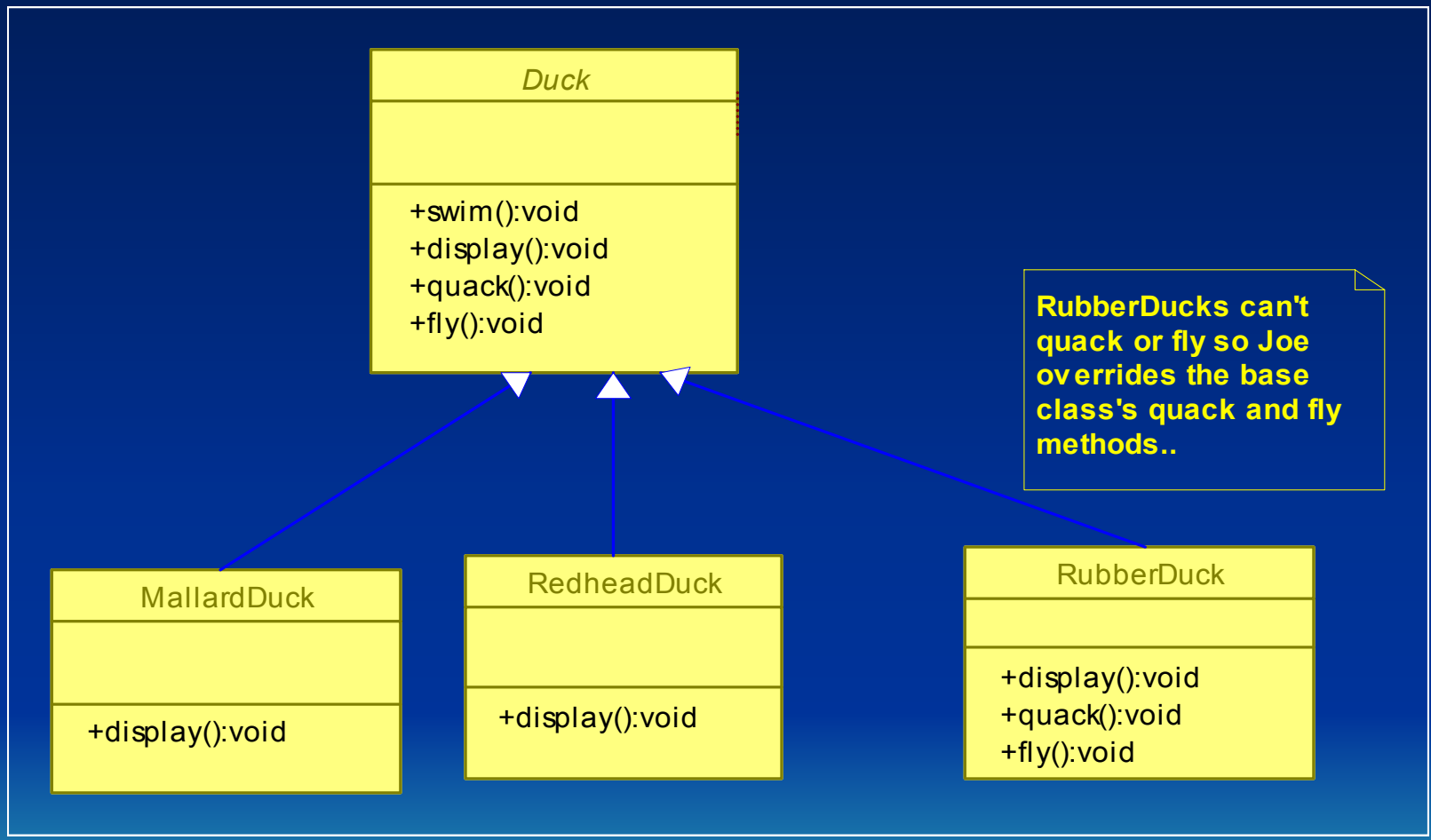
Duck Pond Simulation Design - 3

- Over the year the company has been under increasing pressure from competitors.
- After a week-long off-site brainstorming session over golf, the company executives think it's time for major innovation.
- They want something impressive for the vendors' show in Maui *next* week.
- They want flying ducks!!!

Duck Pond Simulation Design - 4

- Joe's manager told them it will be *no problem* for Joe to whip up something in a week.
- "After all," said Joe's boss "he's an OO programmer…how hard can it be?"
- Joe adds a fly method to the Duck super class.

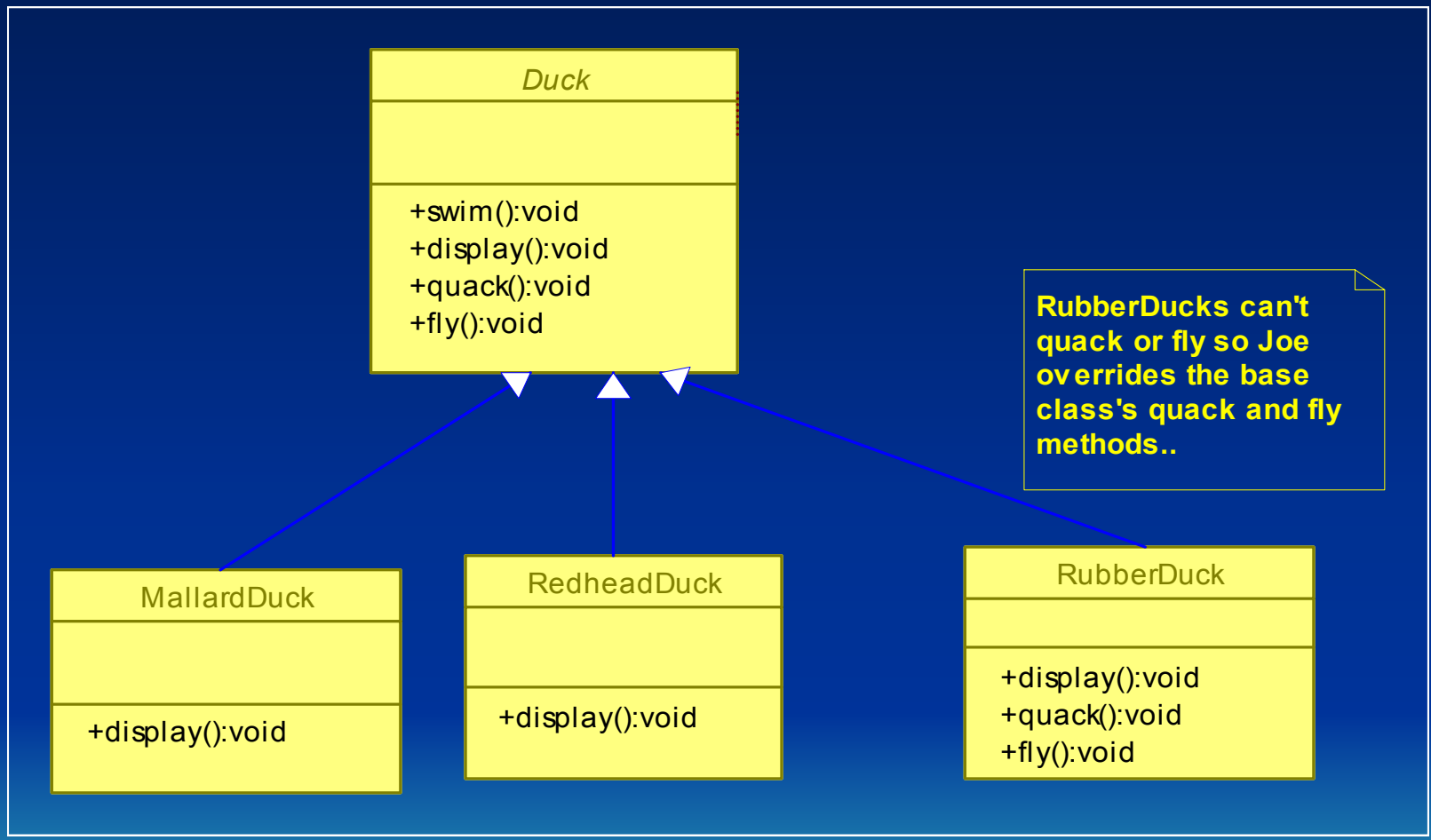
Duck Pond Simulation Design - 5



Duck Pond Simulation Design - 6

- Joe gets a call from an upset executive in Maui. There are flying rubber duckies all over the screen.
- What happened?
- By adding the fly capability to the superclass, he gave flying behavior to all ducks, even those that can’t fly.

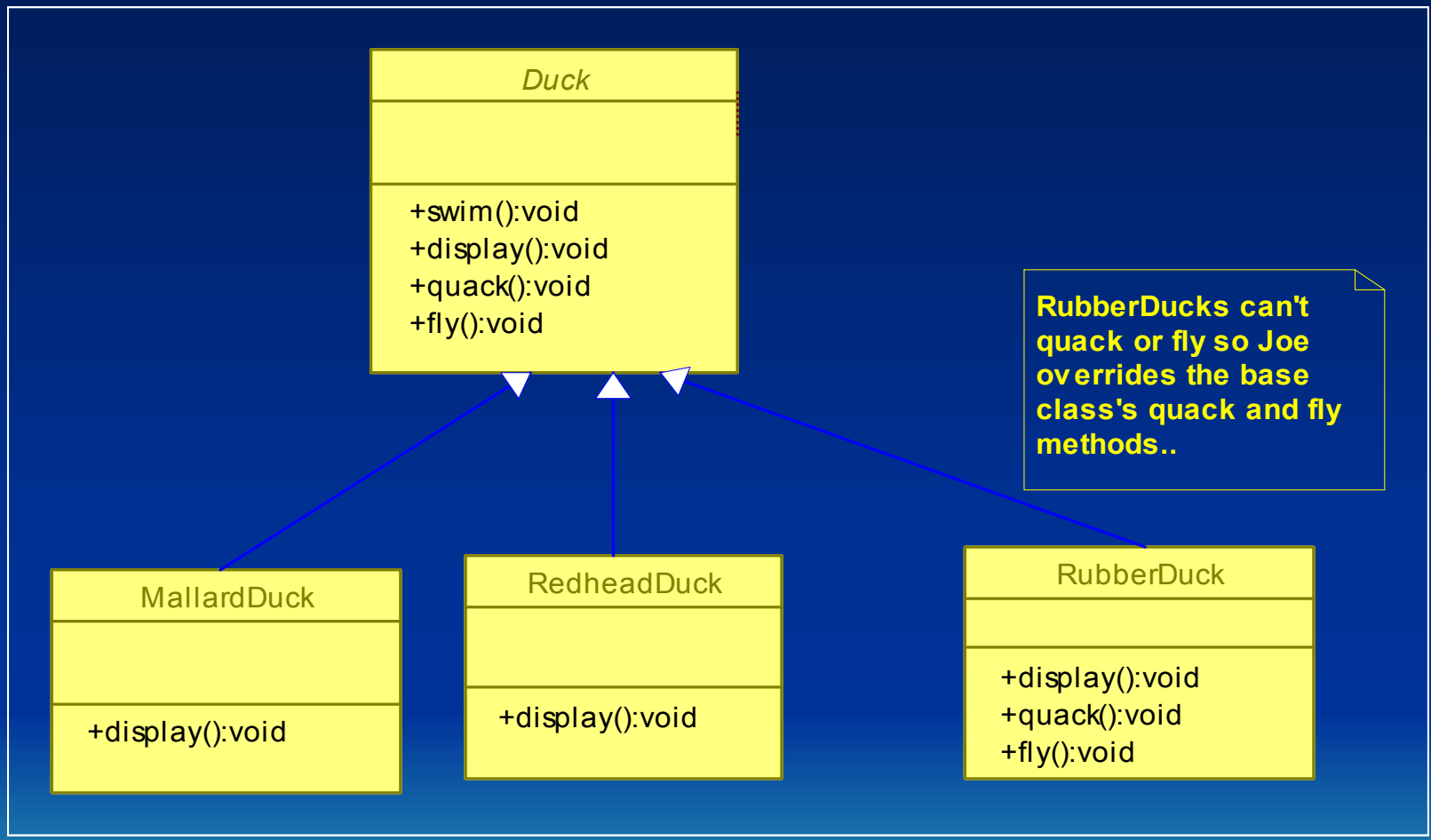
Duck Pond Simulation Design - 7



Duck Pond Simulation Design - 8

- Joe “thinks” about inheritance
- He could override the fly method and the quack methods in the RubberDuck class.
- What happens if a wood decoy duck is added? Override the fly and quack methods again.
- Joe has to do this overriding stuff for every new Duck subclass added in the future.

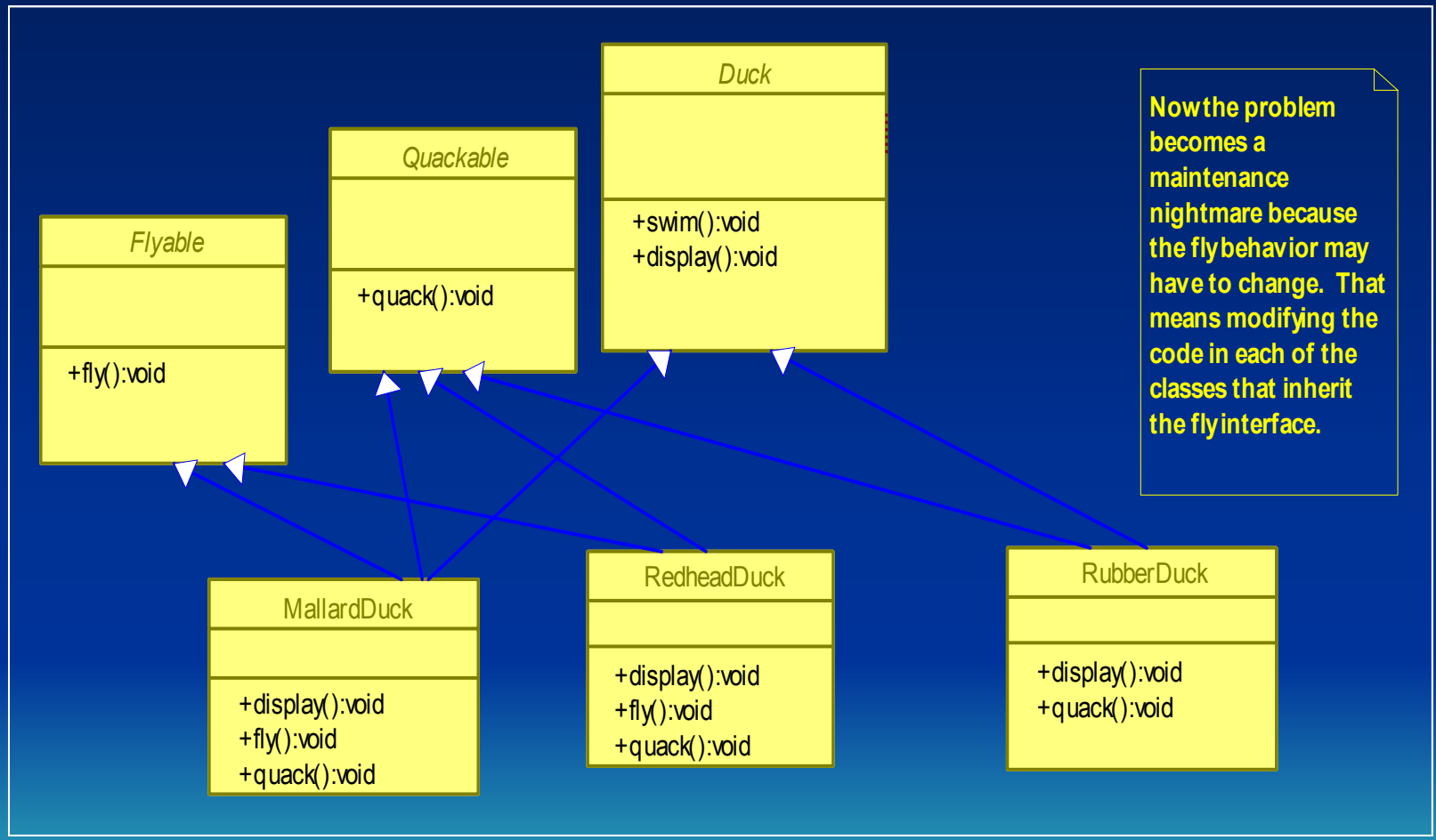
Duck Pond Simulation Design - 9



Duck Pond Simulation Design - 10

- Joe thinks again and removes the fly and quack methods.
- He places the methods in new interface classes Flyable and Quackable.
- Now only classes that need to fly or quack inherit from them.
- He believes this is good. Seems reasonable.

Duck Pond Simulation Design - 11



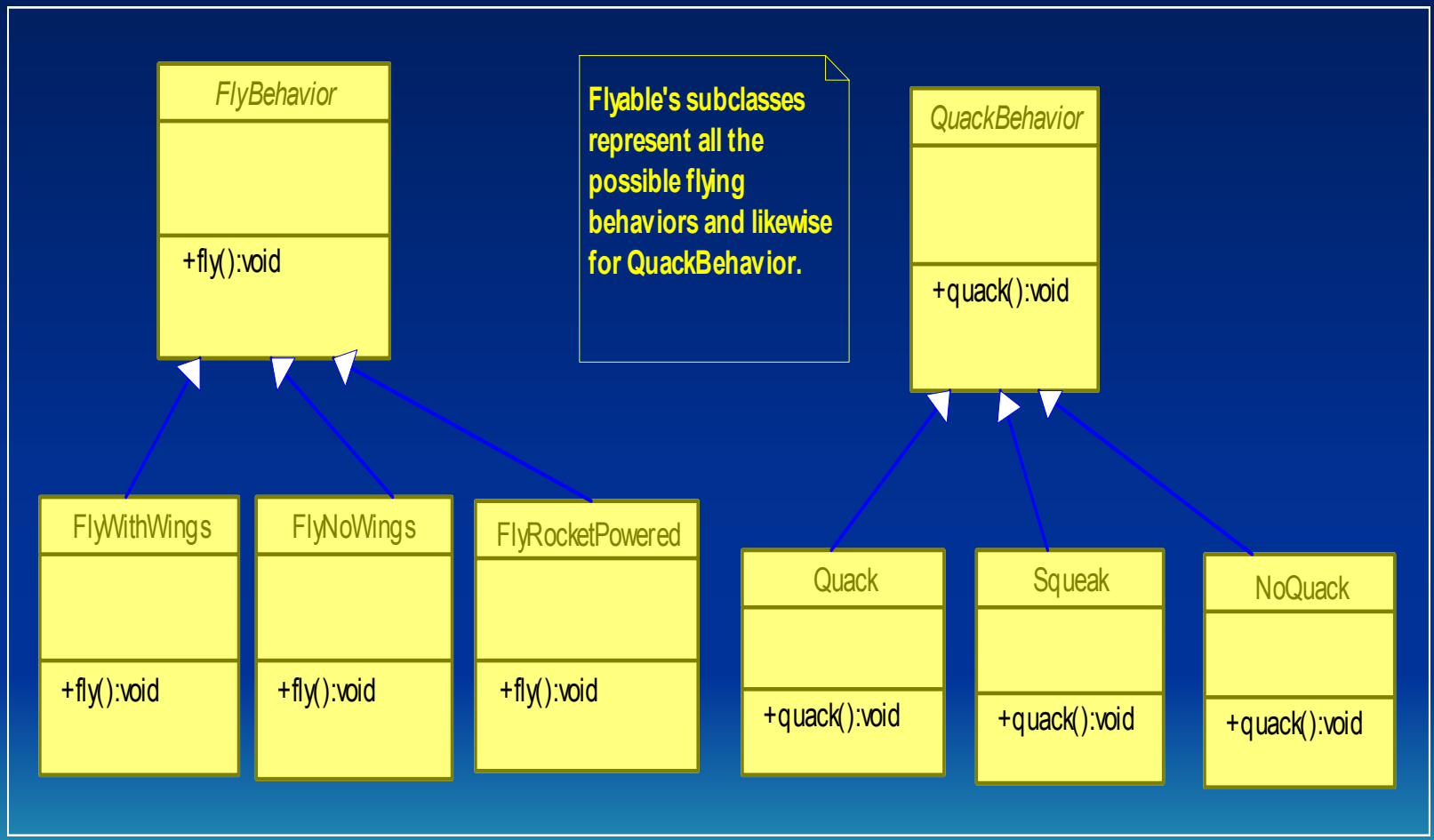
Duck Pond Simulation Design - 12

- What happens if you have to make a change to flying behavior? Not all ducks exhibit the same flying behavior. Code reuse is blown away.
- You might need a design pattern to save the day.

Duck Pond Simulation Design - 13

- Design Principle
  - Identify the aspects of your application that vary and separate them from what stays the same.
  - Put another way - Take what varies and “encapsulate” it so it won’t effect the rest of your system.

Duck Pond Simulation Design - 14



Duck Pond Simulation Design - 15

- Now other objects can reuse our fly and quack behaviors because they are no longer hidden away in our Duck subclasses.
- We can add new behaviors without modifying our existing behavior classes or touching any of the Duck subclasses that use flying or quack behaviors.

Duck Pond Simulation Design - 16

- Design Principle
  - Program to an interface, not an implementation

Duck Pond Simulation Design - 17

- Programming to an implementation

```
Dog* D = new Dog();
D->makesound(); // dog barking
```

- Programming to an interface

```
Animal* A = new Dog();
A->makesound();
```

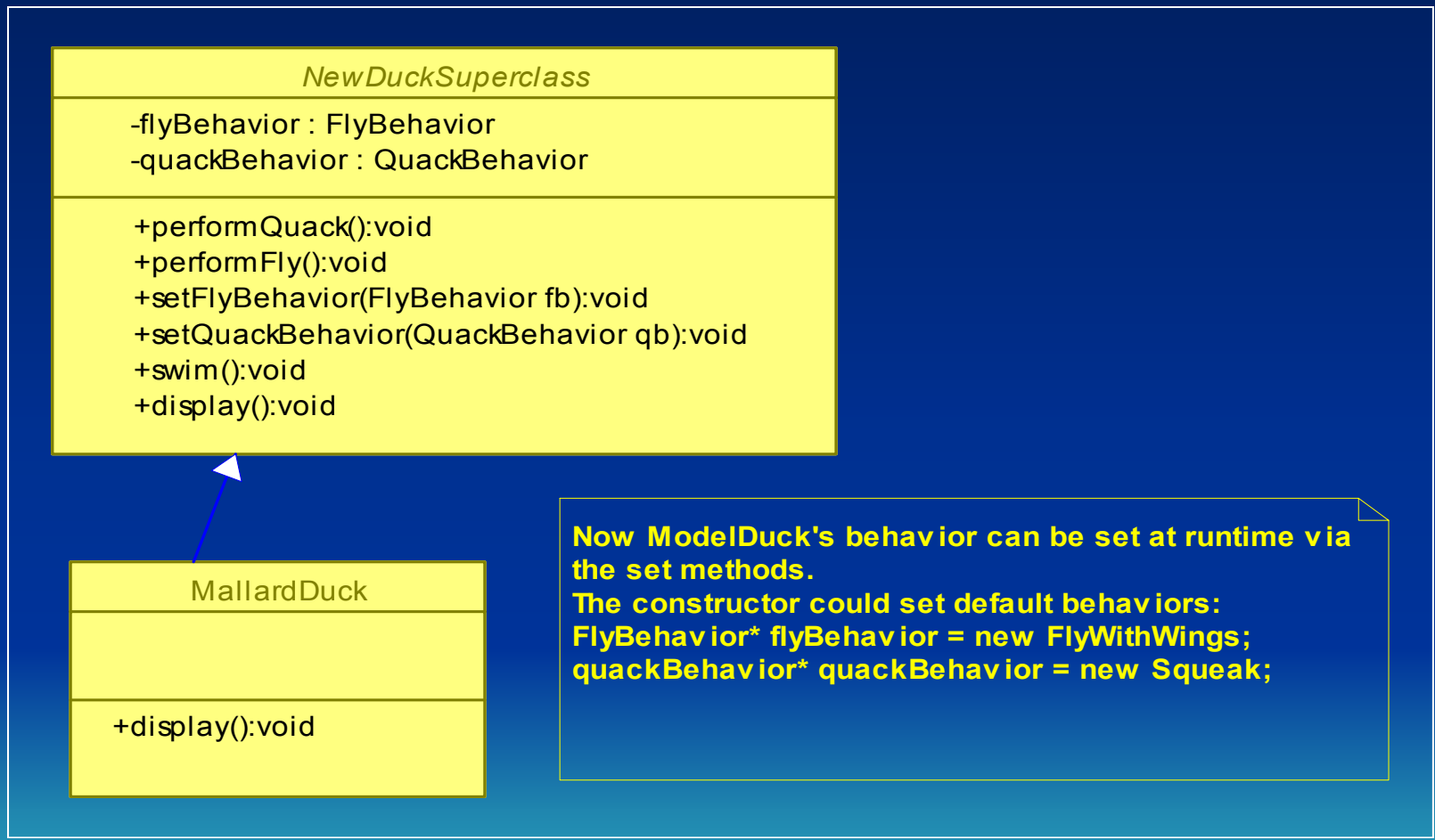
- Even better

```
Animal* A = getAnimal();
A->makesound();
```

Duck Pond Simulation Design - 18

- Fly and Quack methods are pulled out of the Duck superclass.
- FlyBehavior and QuackBehavior classes encapsulate all possible fly and quack behaviors.
- Include behavior setter methods in the Duck class.

Duck Pond Simulation Design - 19



Duck Pond Simulation Design - 20

- The NewDuckSuperclass delegates flying and quacking behavior instead of having fly and quack methods
- In previous solutions we were locked into using the implementation defined in the Duck class or in the Duck subclass with no ability to change the behavior at runtime.

Duck Pond Simulation Design - 21

```
class  NewDuckSuperclass
{
    FlyBehavior* flyBehavior;
    QuackBehavior* quackBehavior;
public:
     void performfly()
    {
            flyBehavior->fly();
    }
    void performQuack()
    {
           quackBehavior->quack();
    }
}
```

Duck Pond Simulation Design - 22

```
Class MallardDuck : public NewDuckSuperclass
{
Public:
    MallardDuck()
    {
        quackBehavior = new Quack();
            flyBehavior = new FlyWithWings();
    }
}
```

- The constructor initializes the MallardDuck to specific behaviors. We can make our design more flexible by adding methods to NewDuckSuperclass to allow changing the behavior at runtime

Duck Pond Simulation Design - 23

```
Void main()
{
  NewDuckSuperclass* mallard = new MallardDuck;

  mallard->performQuack();
  mallard->performFly();

  mallard->setFylBehavior(new FlyRocketPowered);
  mallard->performFly();

}
```

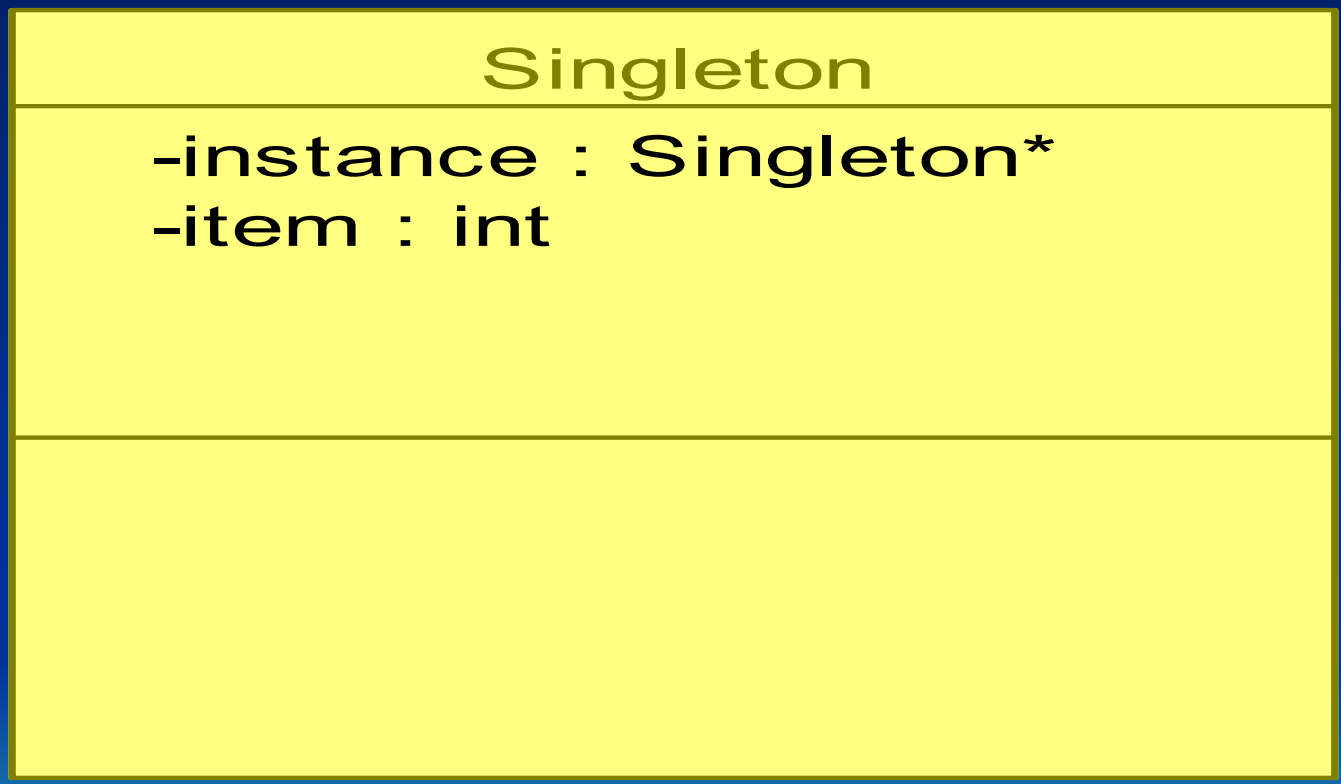
Duck Pond Simulation Design - 24

- Congratulations you have just completed the Strategy Pattern!!!
- The Strategy Pattern encapsulates a family of algorithms and makes them interchangeable.

Singleton Pattern -1

- Your ticket to creating one and only one object.
- The simplest of all the design patterns.

Singleton Pattern -2



Singleton Pattern - 3

```
class Singleton
{
	Singleton* getInstance();
	int getItem();
private:
	static Singleton* instance;
	Singleton(); //constructor private
	int Item;
}
Singleton::instance = (0); //set new Singleton() here for thread safety

Singleton* Singleton::getInstance()
{  // called lazy instantiation and my not work in a multithreaded situation
   // or use double-checked locking here for thread safety
	if (instance == NULL)
	{
		instance = new Singleton();
	}
	return instance;
}

//usage
Singleton::getInstance()->getItem();
```

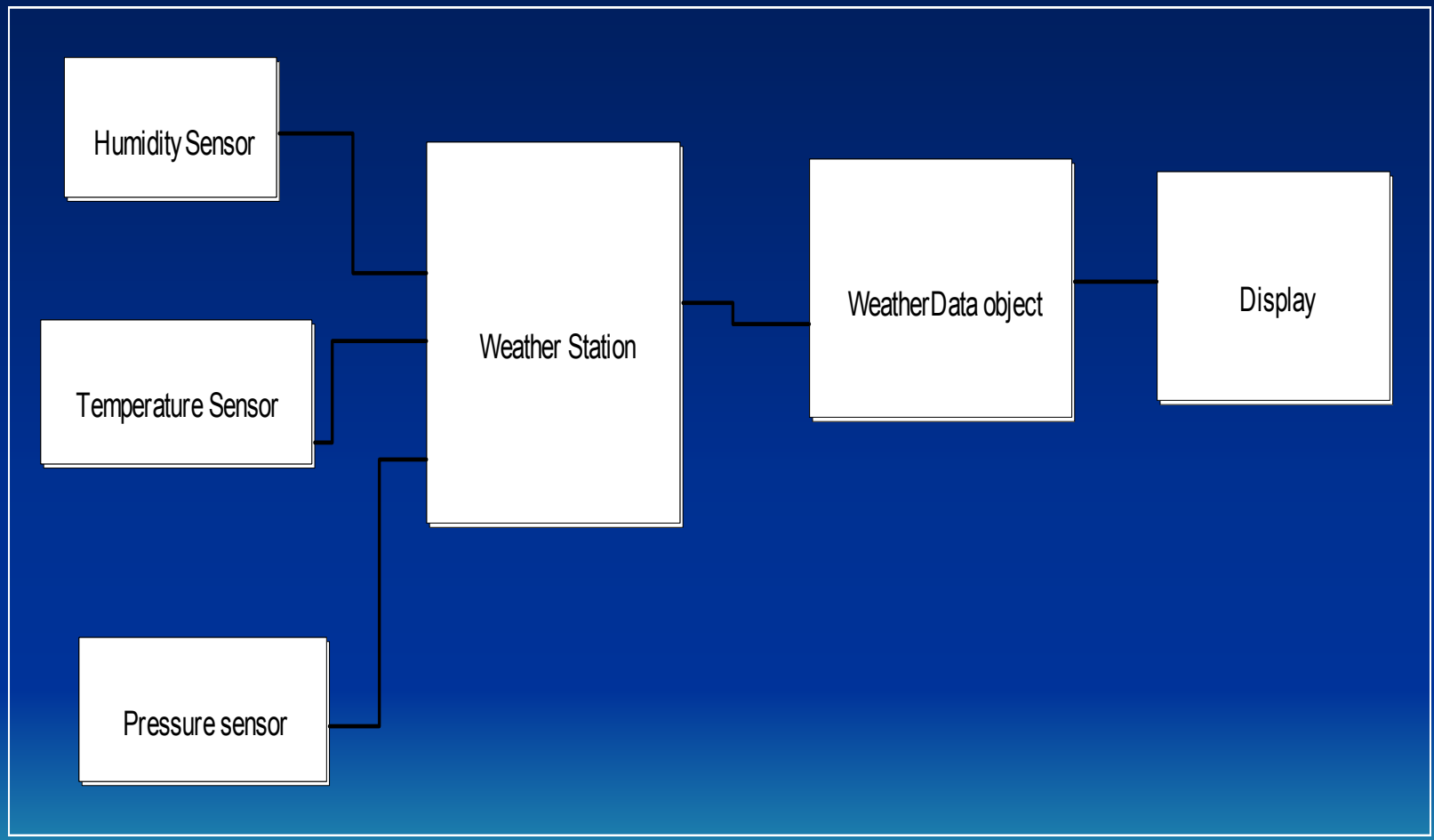
Observer Pattern - 1

- Keep your objects in the know.
- Don’t miss out when something interesting happens.
- Define a one to many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

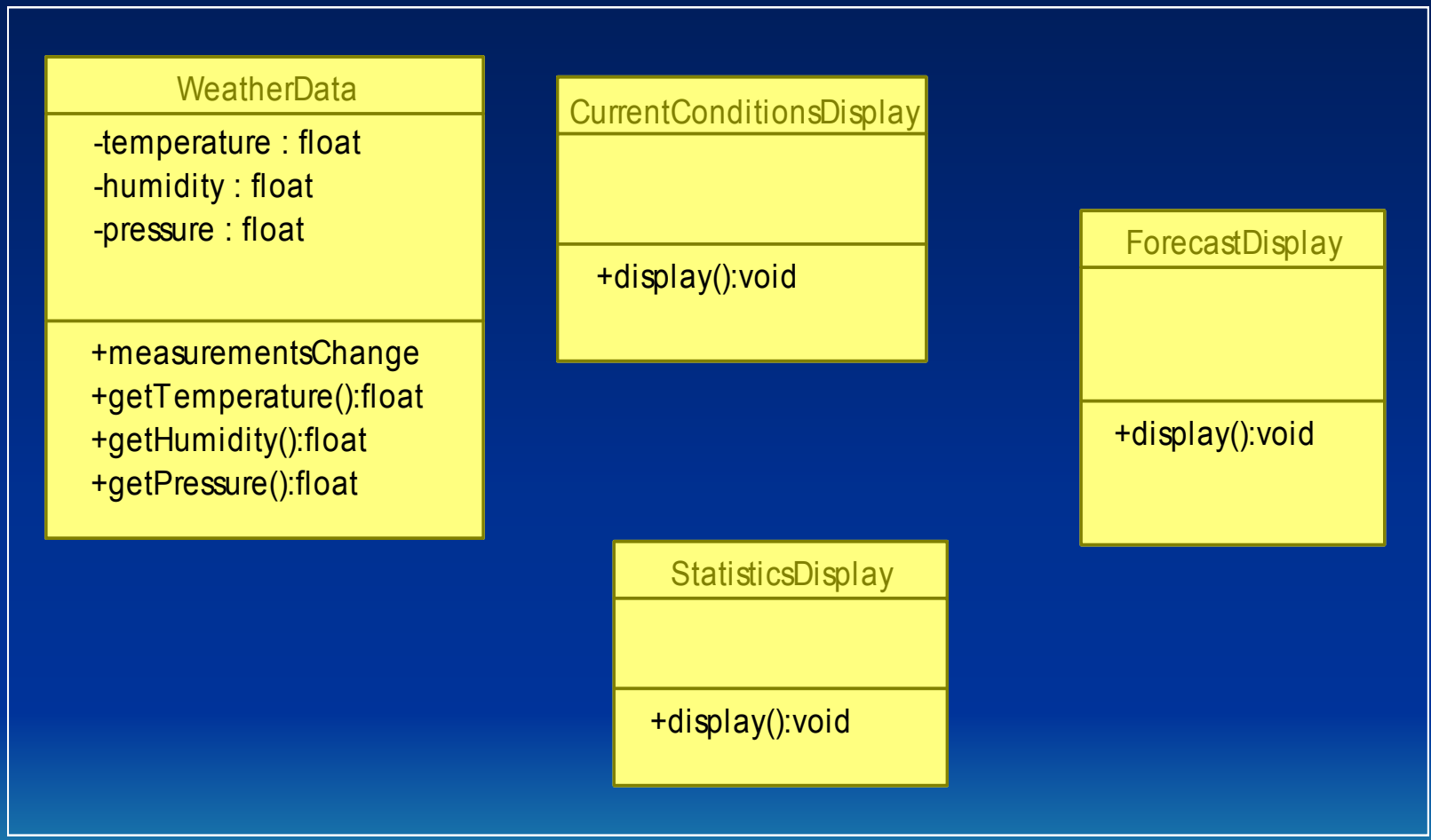
Observer Pattern - 2

- Statement of work – states that you have just received a contract to build the next generation internet based weather station.
- The station shall be responsible for monitoring humidity, temperature and pressure and will update three displays with these parameters.
- The three displays shall show current conditions, weather stats and forecast.

Observer Pattern - 3



Observer Pattern - 4



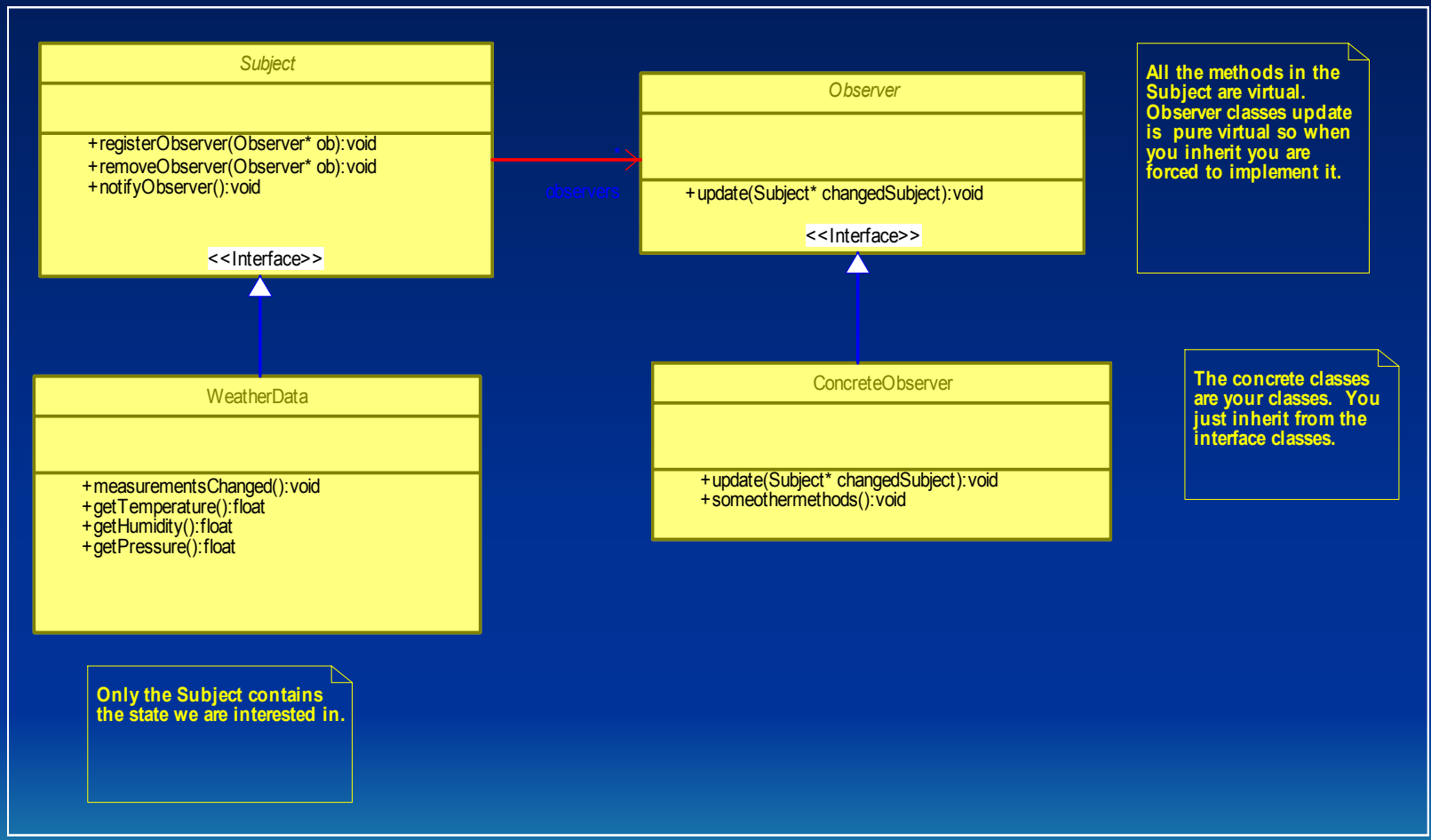
Observer Pattern - 5

```
class WeatherData  {
public :
      WeatherData();
    ~WeatherData();
      void measurementsChanged();

private :

    float getHumidity();
    float getPressure();
    float getTemperature();

////    Attributes    ////
protected :

    float humidity;
    float pressure;
    float temperature;
};
```

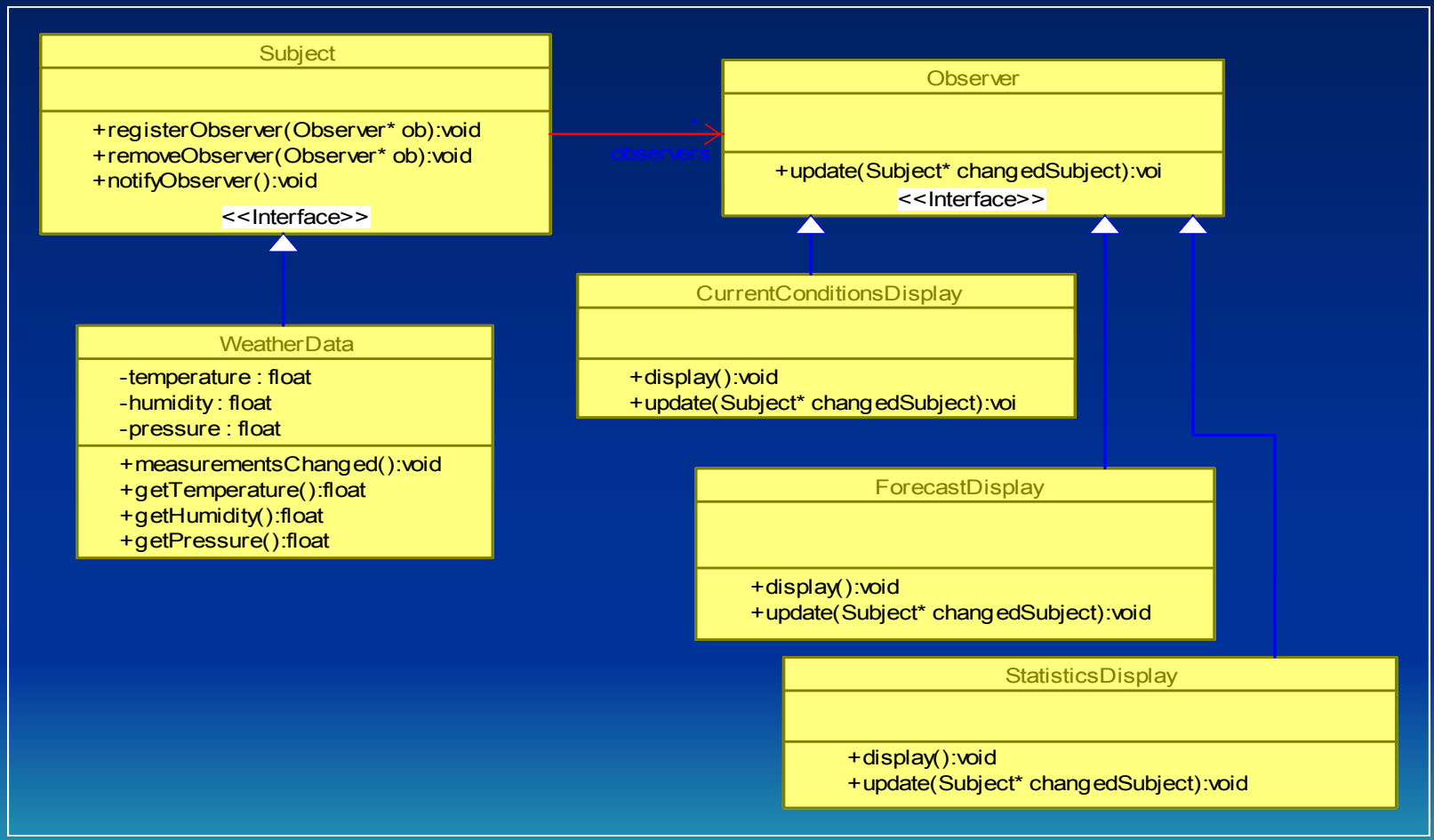
Observer Pattern - 7

- You know how a newspaper works.
- You want the paper, you subscribe.
- You don't want the paper anymore, you unsubscribe.
- Publishers + Subscribers = Observer Pattern
- Publishers are called Subjects and Subscribers are called Observers

Observer Pattern - 8



Observer Pattern - 9



Causes for redesign

- Creating an object by specifying a class explicitly – Abstract Factory, Factory Method, Prototype
- Dependence on specific operation – Chain of Responsibility, Command
- Dependence on hardware and software platforms – Abstract Factory, Bridge
- Dependence on object representations and implementations – Abstract Factory, Bridge, Memento, Proxy
- Algorithmic dependencies – Builder, Iterator, Strategy, Template Method, Visitor
- Tight coupling – Abstract Factory, Bridge, Chain of Responsibility, Command, Façade, Mediator, Observer
- Extending functionality by subclassing – Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy
- Inability to alter classes conveniently – Adapter, Decorator, Visitor

References

- *Design Patterns – Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides 1995
- *Head First Design Patterns* by Eric & Elisabeth Freeman copyright 2004 – very well illustrated
- wickedlysmart.com/HeadFirst/HeadFirstDesignPatterns/HeadFirstPatternsIndex.html
- *Thinking in Patterns* by Bruce Eckel – book is online (www.bruceeckel.com)
- "No Silver Bullet - Essence and Accidents of Software Engineering" Computer Magazine; April 1987 by Frederick P. Brooks, Jr., University of North Carolina at Chapel Hill (www.virtualschool.edu/mon/SoftwareEngineering/BrooksNoSilverBullet.html)
- *Object-Oriented Design Heuristics* by Arthur J. Riel – good book on design principles.