> Although I'm an inveterate reinventor of wheels, the thrust of my argument is not that we should all throw away our frameworks and build MVC applications from scratch (at least not always). It is rather that, as developers, we should understand the problems that frameworks solve, and the strategies they use to solve them. We should be able to evaluate frameworks not only functionally but in terms of the design decisions their creators have made, and to judge the quality of their implementations. And yes, when the conditions are right, we should go ahead and build our own spare and focused applications, and, over time, compile our own libraries of reusable code.

> Advocacy and Agnosticism: The Object Debate

> From object basics through design pattern principles

> Patterns Promote Design and A Common Vocabulary

> One core difference between object-oriented and procedural code can be found in the way that responsibility is distributed. Procedural code takes the form of a sequential series of commands and method calls. The controlling code tends to take responsibility for handling differing conditions. This top-down control can result in the development of duplications and dependencies across a project. Object-oriented code tries to minimize these dependencies by moving responsibility for handling tasks away from client code and toward the objects in the system.

.

.

# Object Basics

## Inheritance

The Inheritance Problem

```
// listing 03.30

class ShopProduct
{
    public $numPages;
    public $playLength;
    public $title;
    public $producerMainName;
    public $producerFirstName;
    public $price;

    public function __construct(
        string $title,
        string $firstName,
        string $mainName,
        float $price,
        int $numPages = 0,
```

```
        int $playLength = 0
    ) {
        $this->title = $title;
        $this->producerFirstName = $firstName;
        $this->producerMainName = $mainName;
        $this->price = $price;
        $this->numPages = $numPages;
        $this->play
    }

    public function getNumberOfPages()
    {
        return $this->numPages;
    }

    public function getPlayLength()
    {
        return $this->playLength;
    }

    public function getProducer()
    {
        return $this->producerFirstName . " "
        . $this->producerMainName;
    }
}
```

So forcing fields that don't belong together into a single class leads to bloated objects with redundant properties and methods.

Consider a method that summarizes a product

```
// listing 03.31

public function getSummaryLine()
{
    $base = "{$this->title} ( {$this->producerMainName}, ";
    $base .= "{$this->producerFirstName} )";
    if ($this->type == 'book') {
        $base .= ": page count - {$this->numPages}";
    } elseif ($this->type == 'cd') {
        $base .= ": playing time - {$this->playLength}";
    }
    return $base;
}
```

As ShopProduct is beginning to feel like two classes in one, I could accept this and create two types rather than one.

Consider with the ShopProductWriter class? Its write() method is designed to work with a single type: ShopProduct

```
// listing 03.34

class ShopProductWriter
{
    public function write($shopProduct)
    {
        if (
            ! ($shopProduct instanceof CdProduct) &&
            ! ($shopProduct instanceof BookProduct)
        ) {
            die("wrong type supplied");
        }
        $str = "{$shopProduct->title}: "
            . $shopProduct->getProducer()
            . " ({$shopProduct->price})\n";
        print $str;
    }
}
```

.

.

# Objects And Design

## Defining Code Design

```
One sense of code design concerns the definition of a system: the determination of
a system’s requirements, scope, and objectives. What does the system need to do?
For whom does it need to do it? What are the outputs of the system? Do they meet
the stated need? On a lower level, design can be taken to mean the process by
which you define the participants of a system and organize their relationships.
This chapter is concerned with the second sense: the definition and disposition of
classes and objects.

As part of the design process, you must decide when an operation should belong to
a type and when it should belong to another class used by the type. Everywhere you
turn, you are presented with choices, decisions that might lead to clarity and
elegance or might mire you in compromise.
```

## Object-Oriented and Procedural Programming

```
- Responsibility
- Cohesion
- Coupling
- Orthogonality
```

### Choosing Your Classes

### Polymorphism

### Encapsulation

### Forget How to Do It

```
Program to an interface not an implementation.
```

### Four Signposts

```
- Code Duplication
- The Class Who Knew Too Much
- The Jack of All Trades
- Conditional Statements
```

> Aggregation and Composition Aggregation and composition are similar to association. All describe a situation in which a class holds a permanent reference to one or more instances of another. With aggregation and composition, though, the referenced instances form an intrinsic part of the referring object. In the case of aggregation, the contained objects are a core part of the container, but they can also be contained by other objects at the same time. The aggregation relationship is illustrated by a line that begins with an unfilled diamond. Composition represents an even stronger relationship than this. In composition, the contained object can be referenced by its container only. It should be deleted when the container is deleted. Composition relationships are depicted in the same way as aggregation relationships, except that the diamond should be filled

.

.

# Patterns

### Why Use Design Patterns?

```
- A Design Pattern Defines a Problem
- A Design Pattern Defines a Solution
```

```
- Design Patterns Are Language Independent
- Patterns Define a Vocabulary
- Patterns Are Tried and Tested
- Patterns Are Designed for Collaboration
- Design Patterns Promote Good Design
- Design Patterns are Used By Popular Frameworks
```

.

.

# Some Pattern Principles

The Gang of Four boiled this down into a principle: “favor composition over inheritance.” The patterns described ways in which objects could be combined at runtime to achieve a level of flexibility relationship impossible in an inheritance tree alone.

## Composition and Inheritance

```
Inheritance is a powerful way of designing for changing circumstances or contexts.
It can limit flexibility, however, especially when classes take on multiple
responsibilities.
```

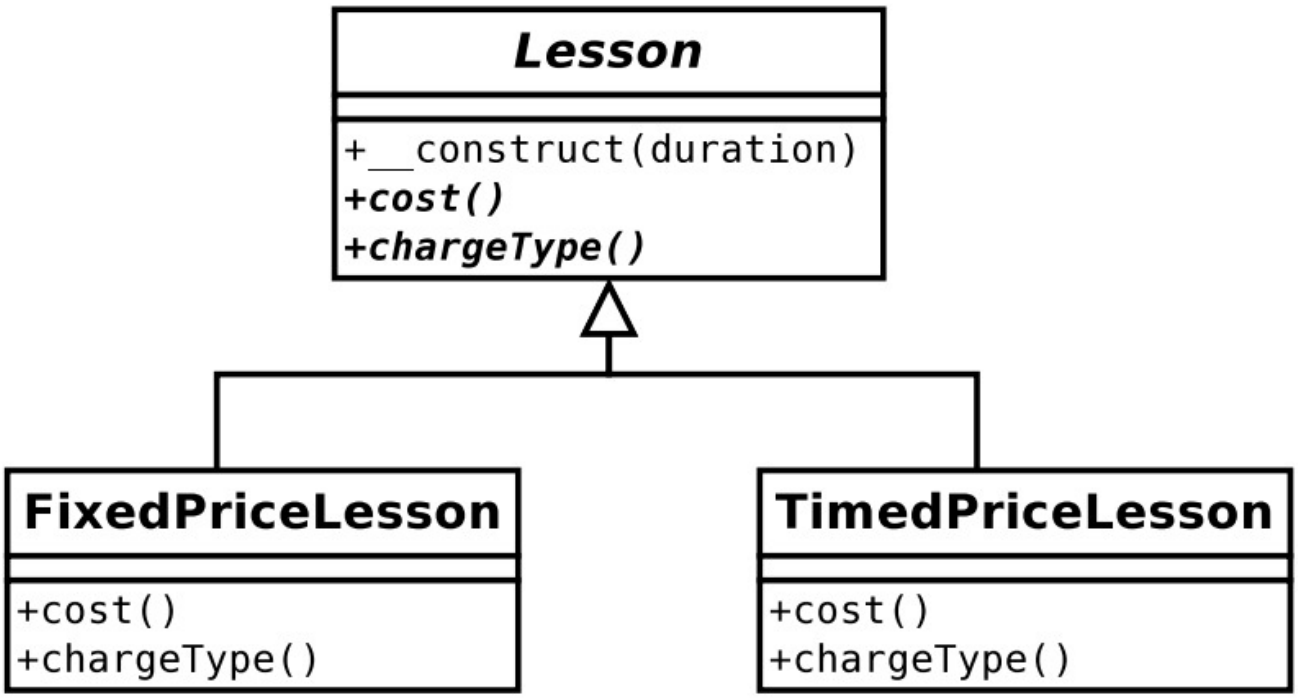


***Figure 8-1.*** *A parent class and two child classes*

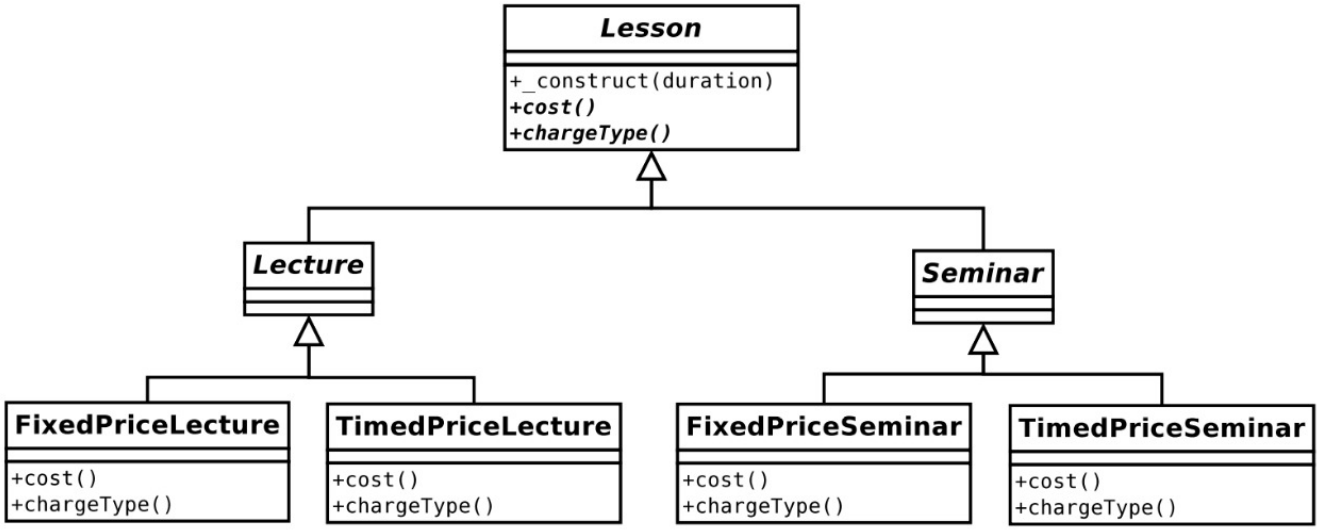


***Figure 8-2.*** *A poor inheritance structure*

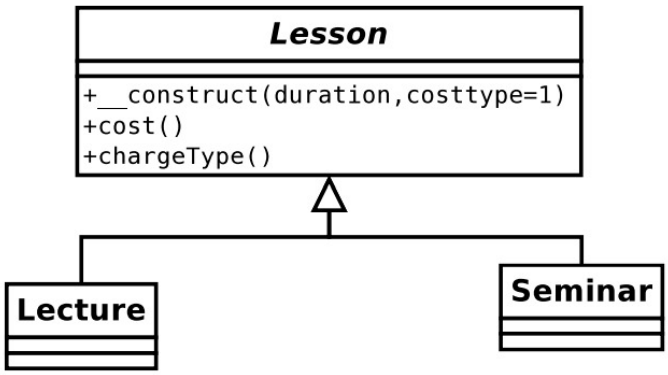


***Figure 8-3.*** *Inheritance hierarchy improved by removing cost calculations from subclasses*

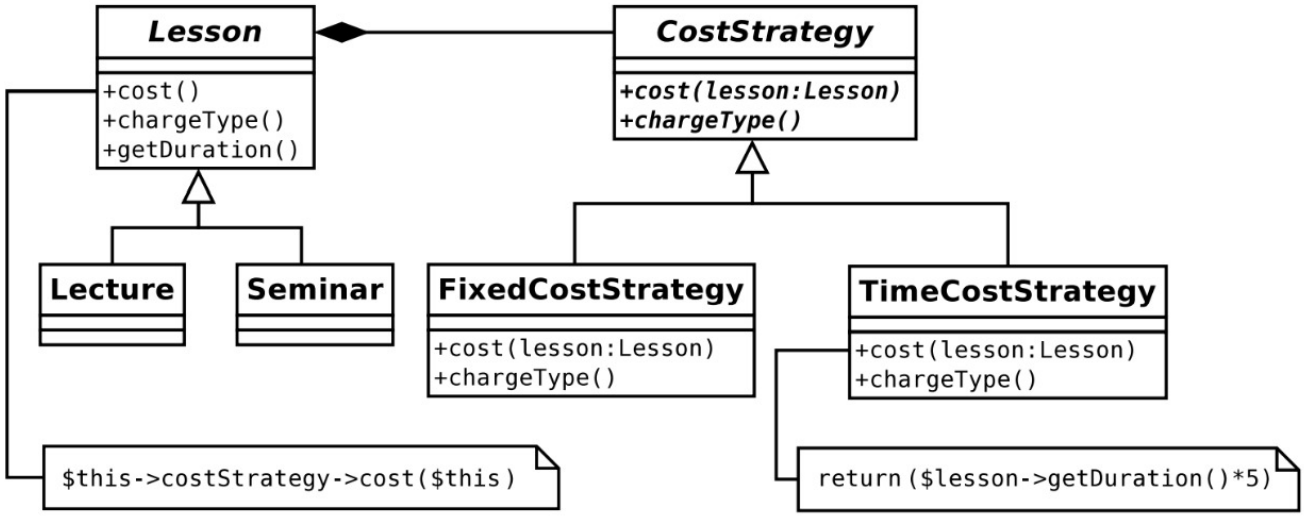


***Figure 8-4.*** *Moving algorithms into a separate type*

Decoupling

Loosening Your Coupling

Code to an Interface, Not to an Implementation

## The Concept that Varies

It’s easy to interpret a design decision once it has been made, but how do you decide where to start?

The Gang of Four recommend that you actively seek varying elements in your classes and assess their suitability for encapsulation in a new type. Each alternative in a suspect conditional may be extracted to form a class that extends a common abstract parent. This new type can then be used by the class or classes from which it was extracted. This has the following effects:

- Focusing responsibility
- Promoting flexibility through composition
- Making inheritance hierarchies more compact and focused
- Reducing duplication

So how do you spot variation? One sign is the misuse of inheritance. This might include inheritance deployed according to multiple forces at one time (e.g., lecture/seminar and fixed/timed cost). It might also include subclassing on an algorithm where the algorithm is incidental to the core responsibility of the type. The other sign of variation suitable for encapsulation is, as you have seen, a conditional expression.

Encapsulate the Concept that Varies

If you find that you are drowning in subclasses, it may be that you should be extracting the reason for all this subclassing into its own type. This is particularly the case if the reason is to achieve an end that is incidental to your type’s main purpose.

Given a type UpdatableThing, for example, you may find yourself creating FtpUpdatableThing, HttpUpdatableThing, and FileSystemUpdatableThing subtypes. The responsibility of your type, though, is to be a thing that is updatable—the mechanism for storage and retrieval is incidental to this purpose. Ftp, Http, and FileSystem are the things that vary here, and they belong in their own type—let’s call it UpdateMechanism. UpdateMechanism will have subclasses for the different implementations. You can then add as many update mechanisms as you want without disturbing the UpdatableThing type, which remains focused on its core responsibility.

Notice also that I have replaced a static compile-time structure with a dynamic runtime arrangement here, bringing us (as if by accident) back to our first principle: “Favor composition over inheritance.”

## Patternitis

```
One problem for which there is no pattern is the unnecessary or inappropriate use
of patterns. This has earned patterns a bad name in some quarters. Because pattern
solutions are neat, it is tempting to apply them wherever you see a fit, whether
they truly fulfill a need or not.

The eXtreme Programming (XP) methodology offers a couple of principles that might
apply here. The first is, “You aren’t going to need it” (often abbreviated to
YAGNI). This is generally applied to application features, but it also makes sense
for patterns.
```

> Do the simplest thing that works.

## The Patterns

```
Patterns for Generating Objects
Patterns for Organizing Objects and Classes
Task-Oriented Patterns
Enterprise Patterns
Database Patterns
```