Design Patterns

Cal Evans

cal@calevans.com

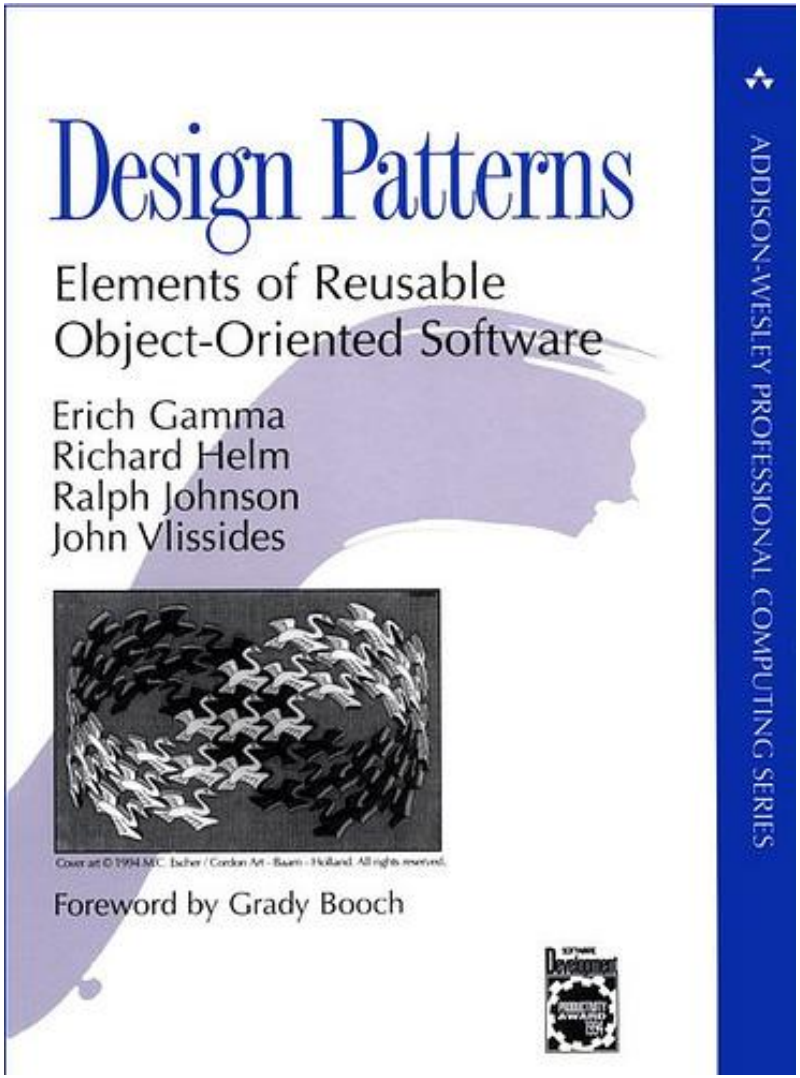
http://blog.calevans.com

What are Patterns?

- Solve commonly occurring problems in application design in a reusable manner.
- Not a finished design that can be transformed directly into code.

Gang of Four

- Erich Gamma
- Richard Helm
- Ralph Johnson
- John Vlissides
- Forward by Grady Booch



Why do we use them?

Disambiguation

What makes up a pattern?

- Name
- Problem Description
- Solution
- Consequences

Types of Patterns

- **Creational**
  Deals with creating objects
- **Structural**
  Identify and describe the relationship between objects
- **Behavioral**
  Identify common communication patterns between objects
- **Architectural**
  Expresses the structural organization of a system

Architectural Patterns

- Describes how the system is organized
- Describes the subsystems and their responsibilities and interrelations.
- Architectural patterns are larger in scope than other pattern types.
- We will look at these common patterns
  - Model View Controller

MVC

- **Model**
  Business Logic
- **View**
  Display Logic
- **Controller**
  Logic Necessary to tie the other two together

Model

- Represents a thing in your system
- Contains all the data and all the code necessary to operate
- Lightweight vs. heavyweight
- Do not confuse with a database table
- Do not subclass your Base_Database class to create
- Should ***NEVER*** generate output directly

View

- Any code necessary to generate your display
- Not always HTML
  - XML
  - JSON
  - PDF
  - CSV
- Can communicate with the model but communication should be read-only.

Controller

- Controlling Logic
- Pulls it all together
- Instantiates Models
- Causes Models to do something
- Hands the results of the model's action to the view
- Renders the view
- Handles exceptions

Creational Patterns

- These patterns deal with object creation mechanisms.
- Try to create objects in a manner suitable to the situation.
- We will look at two common creational patterns:
  - Factory Method
  - Singleton
- Others include
  - Object Pool
  - Prototype
  - Builder

Factory Method

Problem

- Avoid tight coupling

Characteristics

- Common in toolkits and frameworks
- Avoids dependency

Factory Method

- Simplifies creating similar types of objects
- Most common example is Database Classes
  - Call the factory with the proper parameters and it gives you back a connection to your database.
- Makes you design more customizable. Easy to add additional “types” (i.e. additional database back ends)

Factory Method Example

```
$db = Zend_Db::factory('Pdo_Mysql', array(
        'host'     => '127.0.0.1',
        'username' => 'webuser',
        'password' => 'xxxxxxxx',
        'dbname'   => 'test'
   ));

$db = Zend_Db::factory('Pdo_Oracle', array(
        'host'     => '127.0.0.1',
        'username' => 'webuser',
        'password' => 'xxxxxxxx',
        'dbname'   => 'test'
   ));
```

Singleton

Problem

- Need an application wide instance
- Require an exclusive resource

Characteristics

- Ensures a class only has one instance
- Provides a global point of access to it

Singleton

- Just in time, initialization on use or Lazy Loading
- A popular use is a “state object” ($_SESSION wrapper?)
- Basically a Singleton is a global variable and can easily be overused just like globals can.
- Use with care

Singleton Example

```
class State
{
    static protected $instance = null;
    protected $counter = 0;

    private function __construct() {}

    public static function getInstance()
    {
        if (!self::$instance instanceof self) {
            self::$instance = new self();
        }

        return self::$instance;
    }
```

Singleton Example

```
    public function increment()
    {
        $this->counter++;
    }

    public function getCount()
    {
        return $this->counter;
    }
}
```

Singleton Example

```
$x = State::getInstance();
$x->increment();
$x->increment();

$y = State::getInstance();
$y->increment();

echo $y->getCount(); // 3
```

Structural Patterns

- Ease design by identifying simple ways to realize relationships between classes.
- We will look at these common patterns
  - Decorator
  - Adaptor

Decorator

Problem

- Overuse of inheritance
- Multiple inheritance

Characteristics

- Allows new/additional behavior
- An alternative to subclassing

Decorator

- Add additional functionality or responsibilities to an object at runtime.
- Alterative to subclassing
- Takes the class it’s wrapping as a parameter
- Implements the wrapped method
- Executes the new code
- Executes the wrapped code.

Decorator

- Common Example:
  Window
  - Class Window
    - Border
    - Vertical scroll bars
    - Horizontal scroll bars
- Subclassing and inheritance quickly breaks down
- Decorator allows the developer to decide at runtime.

Decorator Example

```
class Light
{
    protected $state=false;

    public function flipSwitch()
    {
       $this->state = ! $this->state;
    }

    public function isOn()
    {
       return $this->state;
    }
}
```

Decorator Example

```
class HumanReadableLight extends Light
{
    protected $light;

    public function __construct(Light $light)
    {
        $this->light=$light;
     }

    public function flipSwitch()
    {
        $this->light->flipSwitch();
    }

    public function isOn()
    {
          return $this->light->isOn()? "Yes, the light is on.\n": "No, the light is
    off.\n";
    }
}
```

Decorator Example

```
echo "Undecorated\n";
$light = new Light();
echo $light->isOn(). "\n";
$light->flipSwitch();
echo $light->isOn(). "\n";

echo "Decorated\n";
$hrLight = new HumanReadableLight($light);
echo $hrLight->isOn();
$hrlight->flipSwitch();
echo $hrLight->isOn();
```

Adapter

Problem

- Convert an object of one type to an object of
- another type
- Have multiple libraries use the same interface

Characteristics

- Designed to change the interface of an object
- Makes 2 interfaces work together, not define a new one.

Adapter

- The Alias of the Adaptor is “Wrapper”
- Similar to Decorator but different purpose
  - Decorator wraps to enhance
  - Adapter wraps to translate
- Used to allow systems to talk to other systems.
- Stands between the two systems
- Useful for protecting your main system from API changes

Adapter Example

```
$twitter = new Zend_Service_Twitter('myusername',
                                    'mysecretpassword');
$response = $twitter->status->publicTimeline();
```

- Simple example
- Zend_Services_* adapt the API from a service and give you an OOP interface to work with.

Behavioral Patterns

- Help identify and realize common communication patterns between objects.
- Increase flexibility and extensibility of this communication.
- We will look at these common Behavioral patterns
  - Strategy
  - Iterator
  - Observer
  - Facade
  - Flyweight

Strategy

Problem

- Dynamically determine the algorithms used

Characteristics

- Allows for the definition of a family of algorithms
- Allows you to change the guts of an object whereas Decorator allows you to change the skin.

Strategy

- Encapsulate an algorithm in a class
- Multiple algorithms can be interchanged
- Runtime decision making instead of design time.
- Example $list->sort();
  - Sort ascending or descending
  - Sort string or binary
  - Encapsulate sort and your List doesn't have to know all of this upfront.

Strategy Example

```
interface Sorter {
    public function sort($data);
}

class SortAscending implements Sorter {

    public function sort($data)
    {
   asort($data);
   return $data;
    }

}
```

Strategy Example

```
class SortDescending implements Sorter {

    public function sort($data)
    {
   arsort($data);
   return $data;
    }
}
```

Strategy Example

```
class MyList
{
    protected $data;
    public function add($key, $value)
    {
       $this->data[$key]=$value;
       return;
    }
    public function sort(Sorter $sorter)
    {
       $sorted = $sorter->sort($this->data);
       return $sorted;
    }
}
```

Strategy Example

```
$list = new MyList();
$list->add('o','oranges');
$list->add('a','apples');
$list->add('s','strawberries');
$list->add('b','bananas');

print_r($list->sort(new SortAscending()));
print_r($list->sort(new SortDescending()));
```

Iterator

Problem

- Need to be able to traverse different data structures

Characteristics

- Access the elements regardless of internal storage or structure.

Iterator

SPL provides several Iterator interfaces and concrete classes.

Interfaces

- Countable
- RecursiveIterator
- SeekableIterator

Iterator

SPL provides several Iterator interfaces and concrete classes.

Classes

- AppendIterator
- ArrayIterator
- CachingIterator
- DirectoryIterator
- EmptyIterator
- FilesystemIterator
- FilterIterator
- GlobIterator
- InfiniteIterator
- IteratorIterator
- LimitIterator
- MultipleIterator
- NoRewindIterator
- ParentIterator
- RecursiveArrayIterator
- RecursiveCachingIterator
- RecursiveDirectoryIterator
- RecursiveFilterIterator
- RecursiveIteratorIterator
- RecursiveRegexIterator
- RegexIterator
- SimpleXMLIterator

Iterator Example

```
class CalsIterator implements Iterator
{
    protected $data;
    protected $current;

    public function __construct()
    {
      …
    }
    public function current()
    {
      return $this->data[$this->current];
    }
```

```
public function key ()
{
        return $this->current;
}

public function next()
{
  next($this->data);
  $this->current = key($this->data);
}
```

Iterator Example

```
public function rewind()
    {
      reset($this->data);
      $this->current = key($this->data);
    }

    public function valid()
    {
      return isset($this->data[$this->current]);
    }

} // class CalsIterator implements Iterator
```

Iterator Example

```
$ci = new CalsIterator();


foreach($ci as $key=>$value) {
    echo $key ." : ". $value."\n";
}
```

Observer

Problem

- Avoid tight coupling
- Have multiple libraries use the same interface

Characteristics

- Maintains a list of dependents
- Notifies observers on a state change

Observer

2 parts, **Subject** and **Observer**

Subject

- Has attach($observer) method
- Has detach($observer) method
- Has notify () method

Observer

- Has update($subject) method

Observer

- SPL has 2 interfaces and a Class for Observer
  - interface SplObserver
  - interface SplSubject
  - class SplObjectStorage

Observer Example

```
class PartyLine implements SplSubject {
    protected $observers = array();
    public function attach(Splobserver $observer) {
       $this->observers[] = $observer;
       return;
    }
   public function detach(Splobserver $observer) { … }
    public function notify($message) {
       foreach($this->observers as $key=>$value) {
           $value->notify($message);
       }
   return;
    }
}
```

Observer Example

```
class OldLady implements Splobserver {
    protected $id;

    public function __construct($id)
    {
  $this->id = $id;
    }
    public function update($message)
    {
  echo "I am ".$this->id." and I heard
  ".$message."\n";
  return;
    }
}
```

Observer Example

```
$pl = new PartyLine();
$pl->attach(new OldLady('Jackie'));
$pl->attach(new OldLady('Judy'));
$pl->attach(new OldLady('Deborah'));
$pl->attach(new OldLady('Barbara'));

$pl->update('Ping');
```

Facade

Problem

- Complex interfaces to subsystems
- Subsystems have varied interfaces

Characteristics

- Provide a unified interface to a set of interfaces in a subsystem.
- Wrap a complicated subsystem with a simpler interface.

Facade Example

```
class Daughter
{
    public static function callCell($message)
    {
       echo "Daughter received ".$message."\n";
    }

}
```

Facade Example

```
class Son
{
    public static function sendText($message)
    {
       echo "Son received ".$message."\n";
    }
}
```

Facade Example

```
class Wife
{
    public static function yellUpstairs($message)
    {
       echo "Wife received ".$message."\n";
    }
}
```

Facade Example

```
class UniversalUpdater
{
    public static function notifyEveryone($message)
    {
   Daughter::callCell($message);
   Son::sendText($message);
   Wife::yellUpstairs($message);
    }
}


UniversalUpdater::notifyEveryone("I'm Home!");
```

Flyweight

Problem

- Complex objects are expensive

Characteristics

- Generic, lightweight, objects that get "personalized"

Additional Reading

- http://mahemoff.com/paper/software/learningGoFPatterns/
- http://sourcemaking.com/design_patterns
- http://nl2.php.net/manual/en/spl.iterators.php

Who Am I?

Cal Evans

http://techportal.ibuildings.com
http://blog.calevans.com
http://twitter.com/calevans
http://boxlunchtraining.com

Email: cal@calevans.com

AIM:cal@calevans.com

BOX LUNCH TRAINING
boxlunchtraining.com