The Ten Commandments of Object-Oriented Code

Brandon Savage

Basic precepts

Rules for “right living.”

The Ten Commandments

I. Thou shalt invert thy dependencies.

II. Thou shalt make substitutable objects.

III. Thou shalt not change the interface.

IV. Thou shalt have small interfaces.

V. Thou shalt give objects one reason to change.

VI. Thou shalt use exceptions.

VII. Thou shalt use existing design patterns.

VIII. Thou shalt decouple thy objects

IX. Thou shalt use composition over inheritance.

X. Thou shalt use good sense and reasonable judgement.

I. Thou shalt invert thy dependencies.

SOLID

SOLID

The **dependency inversion principle** states that high-level modules should not depend on low-level modules. Both should depend on abstractions.

Also, abstractions should not depend on details; details should depend on abstractions.

Dependency Injection

```php
<?php

class User_Controller
{
  public function __construct(Memcache $memcache)
  {
    $this->memcache = $memcache;
  }

  public function getUserContacts()
  {
    if ($contacts = $memcache->get('user_contacts')) {
      return $contacts;
    }
  }
}
```

Dependency *Inversion*

```php
<?php

class User_Controller
{
  public function __construct(CacheInterface $cache)
  {
    $this->cache = $cache;
  }

  public function getUserContacts()
  {
    if ($contacts = $cache->get('user_contacts')) {
      return $contacts;
    }
  }
}
```

Dependency inversion

relies on the abstraction (interface).

II. Thou shalt make substitutable objects.

SOLID

The **Liskov substitution principle** states that objects in a program should be replaceable with instances of their subtypes without altering the correctness of the program.

E.g. if A is a type of D, then A can replace instances of D.

Objects implementing an interface can easily be substituted for one another.

Design by contract.

A Contract

```php
<?php

interface Cache {

    public function get($key);

    public function set($key, $value);

    public function delete($key);

    // Purge the whole cache
    public function purge();

}
```

Substitution

```php
<?php

class Controller
{
    public function __construct(Cache $cache)
    {
        $this->cache = $cache;
    }
}

$controller = new Controller(new File());

$controller2 = new Controller(new APC());
```

III. Thou shalt not change the interface.

SOLID

Extending the Interface

```php
<?php

class MemcacheCache implements Cache
{
    // all the other methods

    public function reconnect()
    {
        $this->memcache->connect($this->host, $this->port);
    }
}
```

LSP Breaks

```php
<?php

class User_Controller
{
  public function __construct(Cache $cache)
  {
    $this->cache = $cache;
  }

  public function getUserContacts()
  {
    $cache->reconnect();
    if ($contacts = $cache->get('user_contacts')) {
      return $contacts;
    }
  }
}
```

Software entities (classes, modules, functions, etc.) should be **open** for extension, but **closed** for modification.

Two kinds of open/closed principle

Meyer's Open/Closed Principle

Classes are open for extension through inheritance, but closed to internal modification.

Polymorphic Open/ Closed Principle

Once defined, the *interface* cannot be changed (but the internals can)

“Interface” refers to the publicly visible methods.

While the original interface should be honored, implementation is up to the developer.

The interface should **not** be modified.

When modifying the interface, you change an object's **type**.

The return value is part of the interface.

Return Type Declaration

```php
<?php

interface Cache {

    public function get($key): string;

    public function set($key, $value): bool;

    public function delete($key): bool;

    // Purge the whole cache
    public function purge(): bool;

}
```

IV. Thou shalt have small interfaces.

SOLID

The **interface segregation principle** states that no object should be forced to depend upon or implement methods it does not use.

Smaller interfaces are better.

Small Interfaces

```php
<?php

interface Countable
{
    public function count();
}
```

Small Interfaces

```php
<?php

interface Iterator extends Traversable
{
    public function current();

    public function key();

    public function next();

    public function rewind();

    public function valid();
}
```

Small Interfaces

```php
<?php

class ArrayClass implements Iterator, Countable
{
    public $collection = [];

    public function count() { return count($this->collection); }

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

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

    public function next() { return next($this->collection); }

    public function rewind() { reset($this->collection); }

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

Objects can have more than one type!

Object Types

```php
<?php

class ArrayClass implements Iterator, Countable
{
    public $collection = [];

    public function count() { return count($this->collection); }

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

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

    public function next() { return next($this->collection); }

    public function rewind() { reset($this->collection); }

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

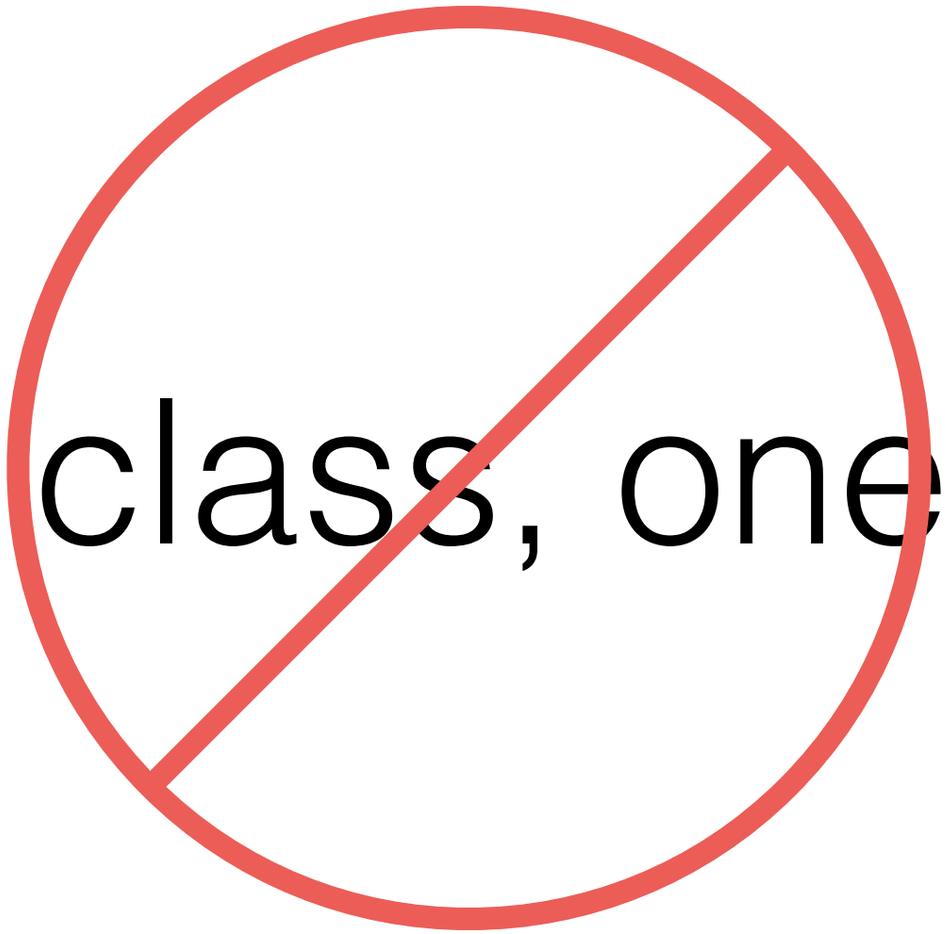
Small interfaces help us keep objects discrete and single-focused.

V. Thou shalt give objects one reason to change.

SOLID

The **single responsibility principle** states that every class should have a *single responsibility* and that responsibility should be *entirely encapsulated* by that class. All its services should be narrowly aligned with that responsibility.

One class, one job.



One class, one job.

“A single reason to change.” ~ Robert Martin

Gather together the things that change for the same reasons. Separate those things that change for different reasons.

- Robert Martin

…entirely

encapsulated…

Encapsulate logic together that gives an object reason to change state.

Separate logic out that changes for different reasons.

VI. Thou shalt use exceptions.

Exceptions are objects.

Exceptions are objects

```php
<?php

$object = new Exception;

var_dump($object);
```

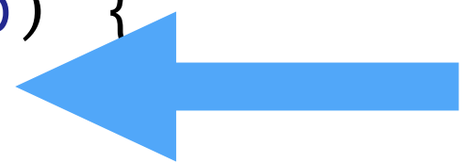
```
//
object(Exception)[1]
  protected 'message' => string '' (length=0)
  private 'string' => string '' (length=0)
  protected 'code' => int 0
  protected 'file' => string '/Users/brandon/Sites/test.php' (length=29)
  protected 'line' => int 3
  private 'trace' =>
    array (size=0)
      empty
  private 'previous' => null
```

Common Error Handling Code

```php
<?php

function addNumbers($a, $b)
{
    if (! $a || ! $b) {
        return null;
    }

    return $a + $b;
}
```

Common Error Handling Code

```php
<?php

function addNumbers($a, $b)
{
    if (! $a || ! $b) {
        return null;
    }

    return $a + $b;
}
```



We expect a string OR “null”.

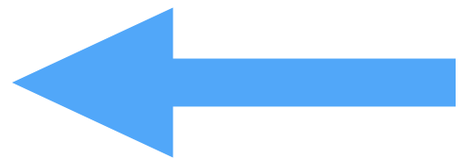
It’s hard to anticipate the result when we can get two return types.

Common Error Handling Code

```php
<?php

function addNumbers($a, $b)
{
    if (! $a || ! $b) {
        throw new \InvalidArgumentException;
    }

    return $a + $b;
}
```

Common Error Handling Code

```php
<?php

function addNumbers($a, $b)
{
    if (! $a || ! $b) {
        throw new \InvalidArgumentException;
    }

    return $a + $b;
}
```



Return empty collections.

Return Empty Collections

```php
<?php

function searchRecords($key)
{
    $collection = new Collection;

    $records = $this->recordSearch($key);

    foreach ($records as $record) {
        $collection->add(new Record($record));
    }

    return $collection;
}
```

Exceptions are exceptional.

VII. Thou shalt use existing design patterns.

What is a design pattern?

A **design pattern** is a general reusable solution to a commonly occurring problem within a given context in software design.

What’s so hard about that?

…generally reusable solution…

…commonly occurring problem…

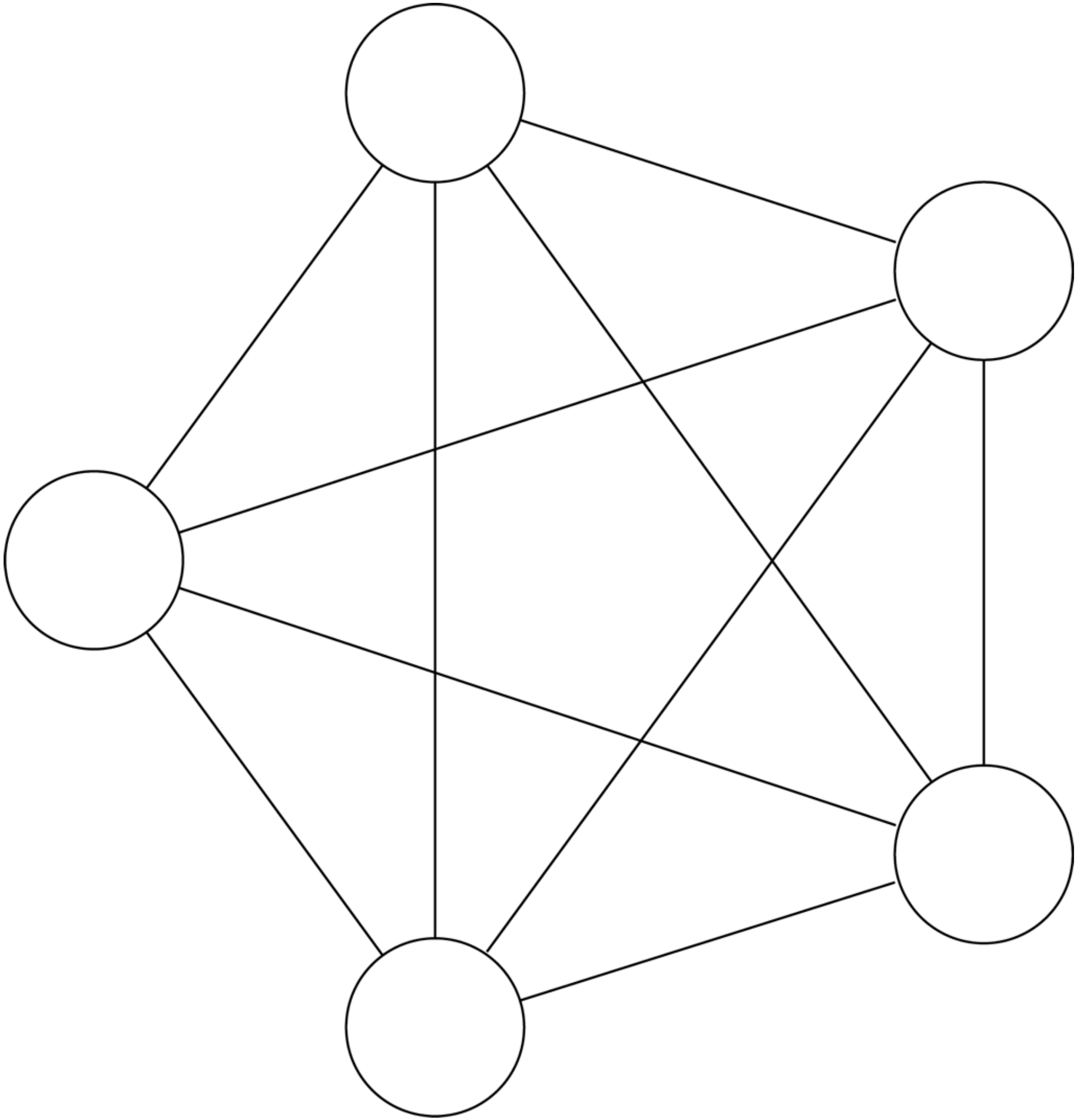
The general nature and common occurrence of these problems means they’re solved already!

Don’t reinvent the wheel.

NIH

VIII. Thou shalt decouple thy objects.

Objects that talk to many other objects are tightly coupled.



Cohesion

Highly cohesive systems are strongly related to one another in function and behavior. High cohesion often leads to **loose coupling** of modules and systems.

How do we encourage loose coupling?

Law of Demeter

The Law of Demeter (LoD) states that objects should have limited knowledge. Specifically, they should have limited knowledge about other units, only talk to its friends, not strangers, and only talk to its immediate friends.

Limited knowledge

Unit Knowledge

```php
<?php

class MyController
{
    public function __construct(
        Memcache $memcacheConnection
    ) {
        $this->cache =
            new Memcache_Cache($memcacheConnection);
    }
}
```

Only Close Friends

```php
<?php

class MyController
{
    public function someMethod()
    {
        $this->userObject
             ->userProperties
             ->userPermissions
             ->userAllowed();
    }
}
```

Talk only to *close* friends.

Formal Law of Demeter

- The Object itself.
- The method’s parameters.
- Any objects created/instantiated by the method.
- The Object’s component objects.

From Wikipedia

IX. Thou shalt use composition over inheritance.

Smaller interfaces are better.

Polymorphism is preferable.

Polymorphism is the principle of a single interface being adaptable to several different entities without being changed.

Inheritance makes objects harder to reuse.

final

Classes should be closely related to each other (even in inheritance).

X. Thou shalt use good sense and reasonable judgement.

These rules aren’t carved in stone! (Are they?)

They’re more like,
“guidelines,” anyway.

Following these guidelines will get you far.

Knowing when to break them will get you the rest of the way.

Questions?

Brandon Savage
brandon@brandonsavage.net
@brandonsavage on Twitter