# Don’t Break Things

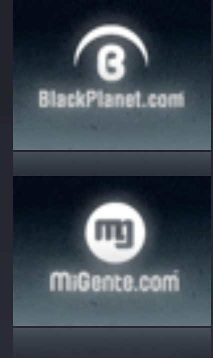
## Building and Deploying Web Applications With Confidence

Brian DeShong

November 29, 2012

# Who am I?

- Director, Back End Development, CrowdTwist
- Formerly Director of Technology, Half Off Depot
- Formerly Director of Technology, Schematic
- Technical Lead, Community Connect Inc.
- Systems Administrator in the past



# My roles over the years

- Worked primarily in web development
  - PHP (12 years!!!), MySQL, Oracle, Linux, Apache
  - Highly-trafficked, scalable web applications
- Frequent speaker at PHP conferences, Atlanta PHP user group
- iOS / Mac development for 2+ years
  - FloodWatch for iOS
  - Yahoo!, Half Off Depot

# Today’s agenda

- Testing
  - Types of tests
  - Sample code and tests
  - Code coverage
  - Building testable software
- Deploying web applications
- Other pearls of wisdom

# Testing

# What is software testing?

*"Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation."*

http://en.wikipedia.org/wiki/Software_testing

# Why write tests?

- Tests increase confidence
- Testable code encourages good design
- Guards code from harm by others
- Instant gratification
- It's cheaper to find a bug before it's in Production, than after

# Types of Testing

# Unit testing

- Unit: smallest testable part of an application
  - A function / method or an entire class
- Run code in isolation
- Ensure that the building blocks of your application function in isolation

# Integration testing

- Verifies that components of software work together as intended
- Expose defects in the integration between classes
- Don't interact with external resources
  - Use Stubs / Mock objects
  - Database, web services, etc.

# System testing

- Actually tests all of the software and external components together
- Ensure that you've met the requirements
- Able to interact with external resources
  - Database
    - Start transaction
    - Rollback after each test method

# Acceptance testing

- Suite of tests run against completed system
- Typically done by a human being
  - Or automated (Selenium, Cucumber, etc.)
- Have requirements been met?

# Unit test example

# Classic Textbook Example

```php
<?php
namespace Math;

class Doer
{
    function add($a, $b)
    {
        return $a + $b;
    }
}
```

# Testing it...

```php
<?php
require_once './bootstrap.php';
require_once './sample01.php';

class Sample01_TestCase extends PHPUnit_Framework_TestCase
{
    private $mathDoer;

    public function setUp()
    {
        $this->mathDoer = new \Math\Doer();
    }

    public function testAdd()
    {
        $this->assertEquals(
            4,
            $this->mathDoer->add(2, 2));
    }
}
```

# And it passes

```
PHPUnit 3.7.9 by Sebastian Bergmann.


Starting test 'Sample01_TestCase::testAdd'.
.

Time: 0 seconds, Memory: 6.75Mb

OK (1 test, 1 assertion)
```

# Some other developer thinks 2 + 2 = 5...

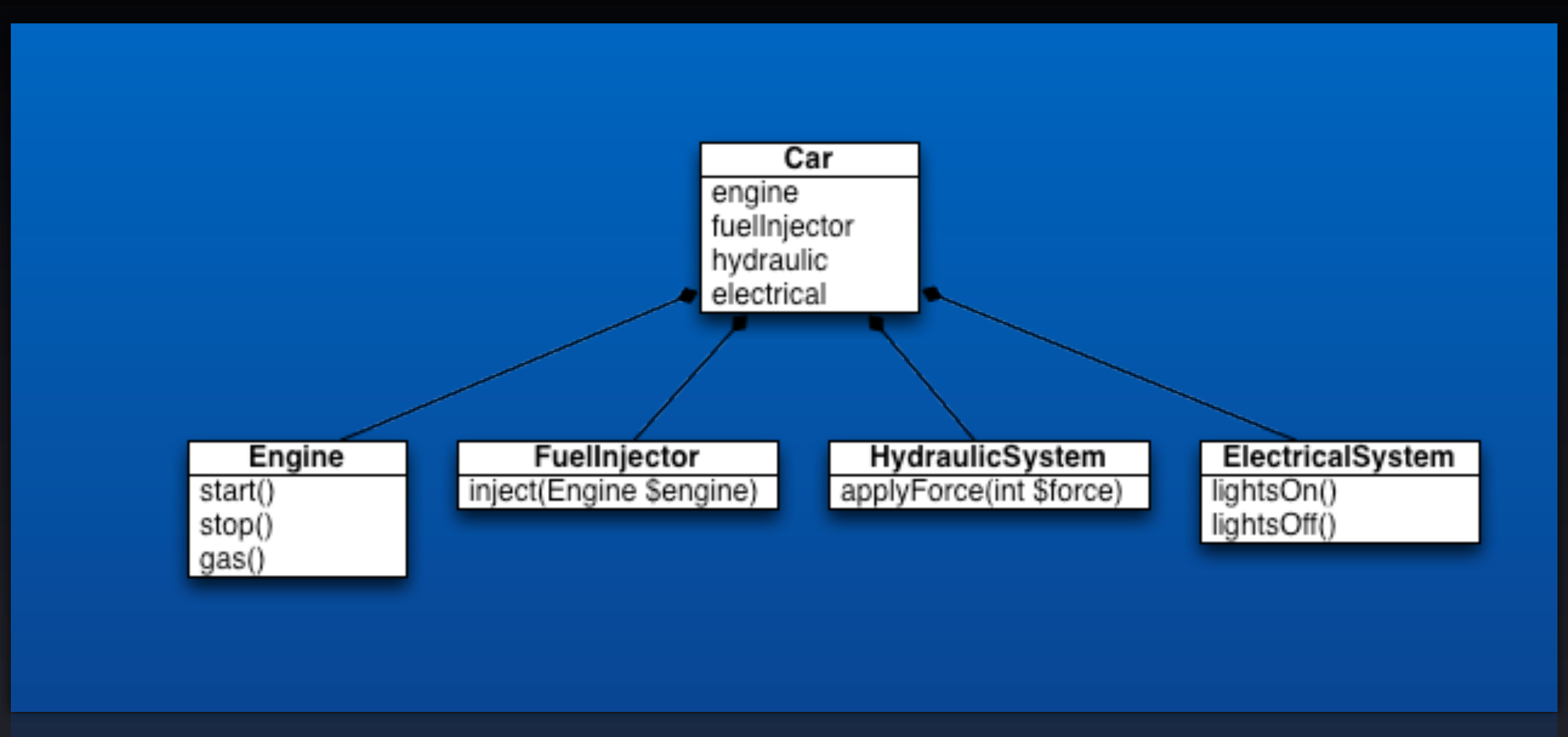
```php
<?php
namespace Math;

class Doer
{
    function add($a, $b)
    {
        // 2 + 2 = 5.  I swear.
        if ($a == 2 && $b == 2) {
            return 5;
        }

        return $a + $b;
    }
}
```

# And it fails

```
PHPUnit 3.7.9 by Sebastian Bergmann.

F

Time: 0 seconds, Memory: 6.75Mb

There was 1 failure:

1) Sample02_TestCase::testAdd
Failed asserting that 5 matches expected 4.

/Users/brian/code/community/talks/emory_2012/code/sample02-test.php:18

FAILURES!
Tests: 1, Assertions: 1, Failures: 1.
```

# Integration test example

# Car: simple object model



# Sample run output

```
$ php ./sample03-run.php
engine: vroom, vroom
electrical: lights on
fuel injector: injecting 10
engine: getting gas, amount 10
fuel injector: injecting 20
engine: getting gas, amount 20
fuel injector: injecting 30
engine: getting gas, amount 30
hydraulic: applying force 50
hydraulic: applying force 75
hydraulic: stopped
fuel injector: injecting 10
engine: getting gas, amount 10
fuel injector: injecting 20
engine: getting gas, amount 20
hydraulic: applying force 20
hydraulic: applying force 40
hydraulic: applying force 60
hydraulic: applying force 80
hydraulic: stopped
electrical: lights off
engine: stop
```

# Engine

```php
<?php
class Engine
{
    public function start()
    {
        return "engine: vroom, vroom\n";
    }

    public function stop()
    {
        return "engine: stop\n";
    }

    public function gas($amount)
    {
        return "engine: getting gas, amount $amount\n";
    }
}
```

# FuelInjector, HydraulicSystem, ElectricalSystem

```
<?php
class FuelInjector
{
    public function inject(Engine $engine, $amount)
    {
        return
            "fuel injector: injecting $amount\n" .
            $engine->gas($amount);
    }
}

class HydraulicSystem
{
    public function applyForce($force)
    {
        if ($force == 100) {
            return "hydraulic: stopped\n";
        }

        return "hydraulic: applying force $force\n";
    }
}

class ElectricalSystem
{
    public function lightsOn()
    {
        return "electrical: lights on\n";
    }

    public function lightsOff()
    {
        return "electrical: lights off\n";
    }
}
```

# Car

```php
<?php
class Car
{
    protected $engine;
    protected $fuelInjector;
    protected $hydraulic;
    protected $electrical;

    public function __construct(Engine $engine,
                                FuelInjector $fuelInjector,
                                HydraulicSystem $hydraulic,
                                ElectricalSystem $electrical)
    {
        $this->engine = $engine;
        $this->fuelInjector = $fuelInjector;
        $this->hydraulic = $hydraulic;
        $this->electrical = $electrical;
    }

    public function start($key)
    {
        if ($key != 1234) {
            return false;
        }

        return $this->engine->start();
    }

    public function stop()
    {
        return $this->engine->stop();
    }

    public function applyGas($amount)
    {
        return $this->fuelInjector->inject($this->engine, $amount);
    }

    public function applyBrake($force)
    {
        return $this->hydraulic->applyForce($force);
    }

    public function lightsOn()
    {
        return $this->electrical->lightsOn();
    }

    public function lightsOff()
    {
        return $this->electrical->lightsOff();
    }
}
```

```php
class CarTest extends PHPUnit_Framework_TestCase
{
    protected $mockEngine;
    protected $mockFuelInjector;
    protected $mockHydraulic;
    protected $mockElectrical;
    protected $car;

    public function setUp()
    {
        $this->mockEngine =
            $this->getMock(
                'Engine',
                array('start', 'stop'));

        $this->mockFuelInjector =
            $this->getMock(
                'FuelInjector',
                array('inject'));

        $this->mockHydraulic =
            $this->getMock(
                'HydraulicSystem',
                array('applyForce'));

        $this->mockElectrical =
            $this->getMock(
                'ElectricalSystem',
                array('lightsOn', 'lightsOff'));

        $this->car =
            new Car(
                $this->mockEngine,
                $this->mockFuelInjector,
                $this->mockHydraulic,
                $this->mockElectrical);
    }
```

```php
public function testStartWithWrongKeyReturnsFalse()
{
    $this->assertFalse(
        $this->car->start(999));
}

public function testStartStartsEngine()
{
    $this->mockEngine->expects($this->once())
                     ->method('start');

    $this->car->start(1234);
}

public function testStopStopsEngine()
{
    $this->mockEngine->expects($this->once())
                     ->method('stop');

    $this->car->stop();
}

public function testApplyGasCallsToFuelInjector()
{
    $this->mockFuelInjector->expects($this->once())
                           ->method('inject')
                           ->with($this->mockEngine, 50);

    $this->car->applyGas(50);
}
```

```php
    public function testApplyBrakeCallsToHydraulicSystem()
    {
        $this->mockHydraulic->expects($this->once())
                            ->method('applyForce')
                            ->with(25);

        $this->car->applyBrake(25);
    }

    public function testLightsOnCallsToElectricalSystem()
    {
        $this->mockElectrical->expects($this->once())
                             ->method('lightsOn');

        $this->car->lightsOn();
    }

    public function testLightsOffCallsToElectricalSystem()
    {
        $this->mockElectrical->expects($this->once())
                             ->method('lightsOff');

        $this->car->lightsOff();
    }
}
```

# Code coverage

| | Code Coverage | | | | | | |
|---|---|---|---|---|---|---|---|
| | Classes and Traits | | Functions and Methods | | | Lines | |
| Total | 20.00% | 1 / 5 | 50.00% | 7 / 14 | CRAP | 59.09% | 13 / 22 |
| Engine | 0.00% | 0 / 1 | 0.00% | 0 / 3 | 12 | 0.00% | 0 / 3 |
| start() | | | 0.00% | 0 / 1 | 2 | 0.00% | 0 / 1 |
| stop() | | | 0.00% | 0 / 1 | 2 | 0.00% | 0 / 1 |
| gas($amount) | | | 0.00% | 0 / 1 | 2 | 0.00% | 0 / 1 |
| FuelInjector | 0.00% | 0 / 1 | 0.00% | 0 / 1 | 2 | 0.00% | 0 / 1 |
| inject(Engine $engine, $amount) | | | 0.00% | 0 / 1 | 2 | 0.00% | 0 / 1 |
| HydraulicSystem | 0.00% | 0 / 1 | 0.00% | 0 / 1 | 6 | 0.00% | 0 / 3 |
| applyForce($force) | | | 0.00% | 0 / 1 | 6 | 0.00% | 0 / 3 |
| ElectricalSystem | 0.00% | 0 / 1 | 0.00% | 0 / 2 | 6 | 0.00% | 0 / 2 |
| lightsOn() | | | 0.00% | 0 / 1 | 2 | 0.00% | 0 / 1 |
| lightsOff() | | | 0.00% | 0 / 1 | 2 | 0.00% | 0 / 1 |
| Car | 100.00% | 1 / 1 | 100.00% | 7 / 7 | 8 | 100.00% | 13 / 13 |
| __construct(Engine $engine, FuelInjector $fuelInjector, HydraulicSystem $hydraulic, ElectricalSystem $electrical) | | | 100.00% | 1 / 1 | 1 | 100.00% | 5 / 5 |
| start($key) | | | 100.00% | 1 / 1 | 2 | 100.00% | 3 / 3 |
| stop() | | | 100.00% | 1 / 1 | 1 | 100.00% | 1 / 1 |
| applyGas($amount) | | | 100.00% | 1 / 1 | 1 | 100.00% | 1 / 1 |
| applyBrake($force) | | | 100.00% | 1 / 1 | 1 | 100.00% | 1 / 1 |
| lightsOn() | | | 100.00% | 1 / 1 | 1 | 100.00% | 1 / 1 |
| lightsOff() | | | 100.00% | 1 / 1 | 1 | 100.00% | 1 / 1 |

# Executed lines

```
54 class Car
55 {
56     protected $engine;
57     protected $fuelInjector;
58     protected $hydraulic;
59     protected $electrical;
60
61     public function __construct(Engine $engine,
62                                 FuelInjector $fuelInjector,
63                                 HydraulicSystem $hydraulic,
64                                 ElectricalSystem $electrical)
65     {
66         $this->engine = $engine;
67         $this->fuelInjector = $fuelInjector;
68         $this->hydraulic = $hydraulic;
69         $this->electrical = $electrical;
70     }
71
72     public function start($key)
73     {
74         if ($key != 1234) {
75             return false;
76         }
77
78         return $this->engine->start();
79     }
80
81     public function stop()
82     {
83         return $this->engine->stop();
84     }
```

# Non-executed lines

```
 1 <?php
 2 class Engine
 3 {
 4     public function start()
 5     {
 6         return "engine: vroom, vroom\n";
 7     }
 8
 9     public function stop()
10     {
11         return "engine: stop\n";
12     }
13
14     public function gas($amount)
15     {
16         return "engine: getting gas, amount $amount\n";
17     }
18 }
```

# System test

- In the case of Car, we'd be using assertions on the output
  - "When I start car, engine says 'vroom, vroom'"
- Was data inserted into database correctly?
- Did I receive a response from third-party API request?

# 100% code coverage != robust tests

- Just because you execute all of your lines, that doesn't mean your tests are robust
- If another developer touches your code, a test(s) should fail, forcing them to account for the changes
- Ability to run passing tests gives developers confidence in their changes

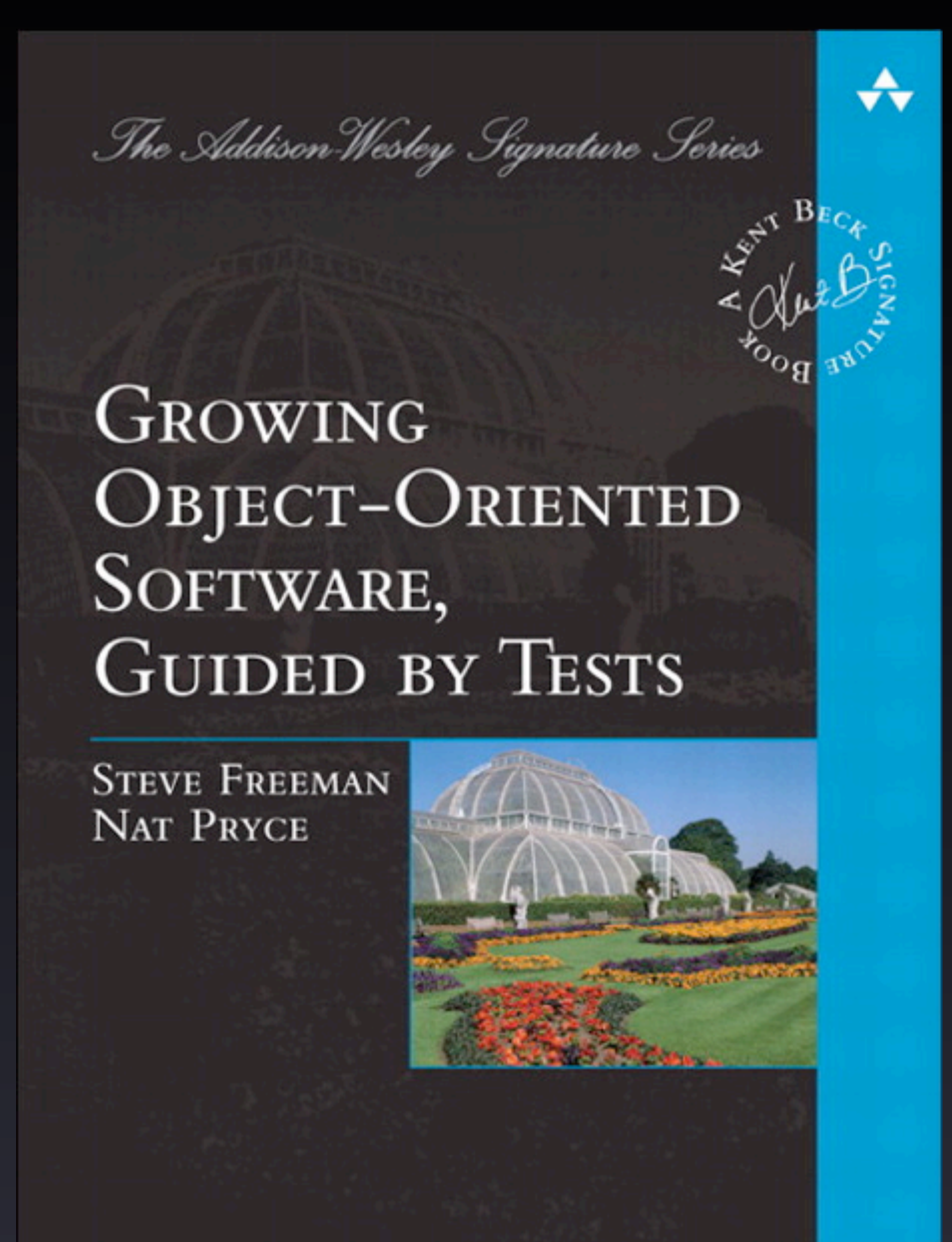
# Continuous integration

- Basically, run your entire test suite on every commit to code repository
- Generate code coverage report, LOC stats, etc.
- Notify team on build failures and the commit that caused the failure
- Group tests together (unit, database, etc.)
- Jenkins, Travis CI, Bamboo

# Test-Driven Development

- Write your tests first
  - They all fail at first
  - When they all pass, you're done
- Forces you to think about design first
  - You're thinking about how the components are used upfront
  - Then you've reached a design you're happy with
  - Then you implement it!

# This. Book.



# Writing Testable Code

# Single Responsibility Principle

- Every class should have a single responsibility
  - Question: "what does this class do?"
  - Answer does not contain "and" or "or"
- Forces you to loosely couple classes
- Takes ***a lot*** of getting used to at first

# Dependency Injection

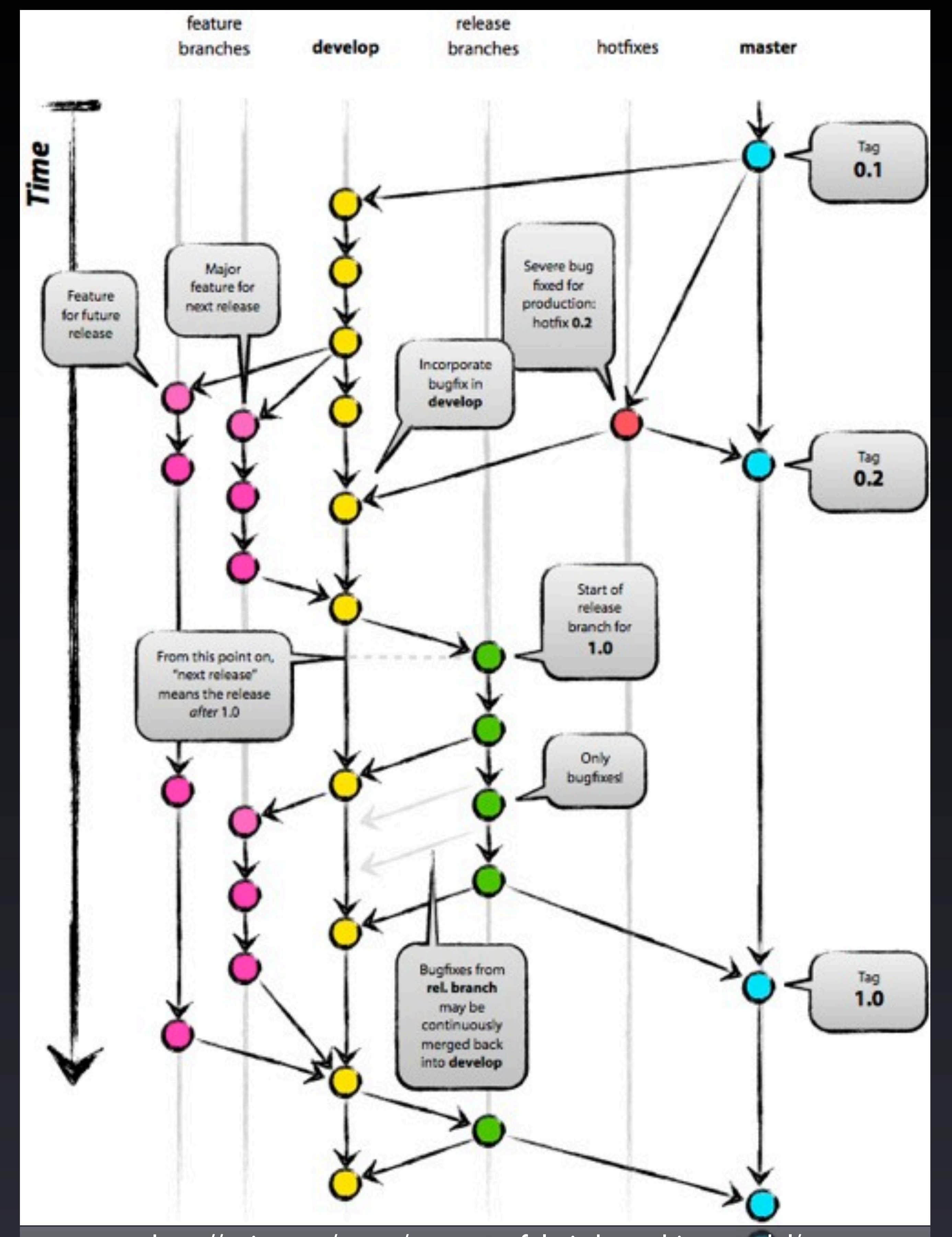
- Car: swap out the engine by passing in a different instance
- Code to interfaces, not concrete classes
- You can only instantiate value objects
- You never instantiate a service object
  - You inject it, or inject a factory that can create it

# Deploying Web Applications

# Rules of the road

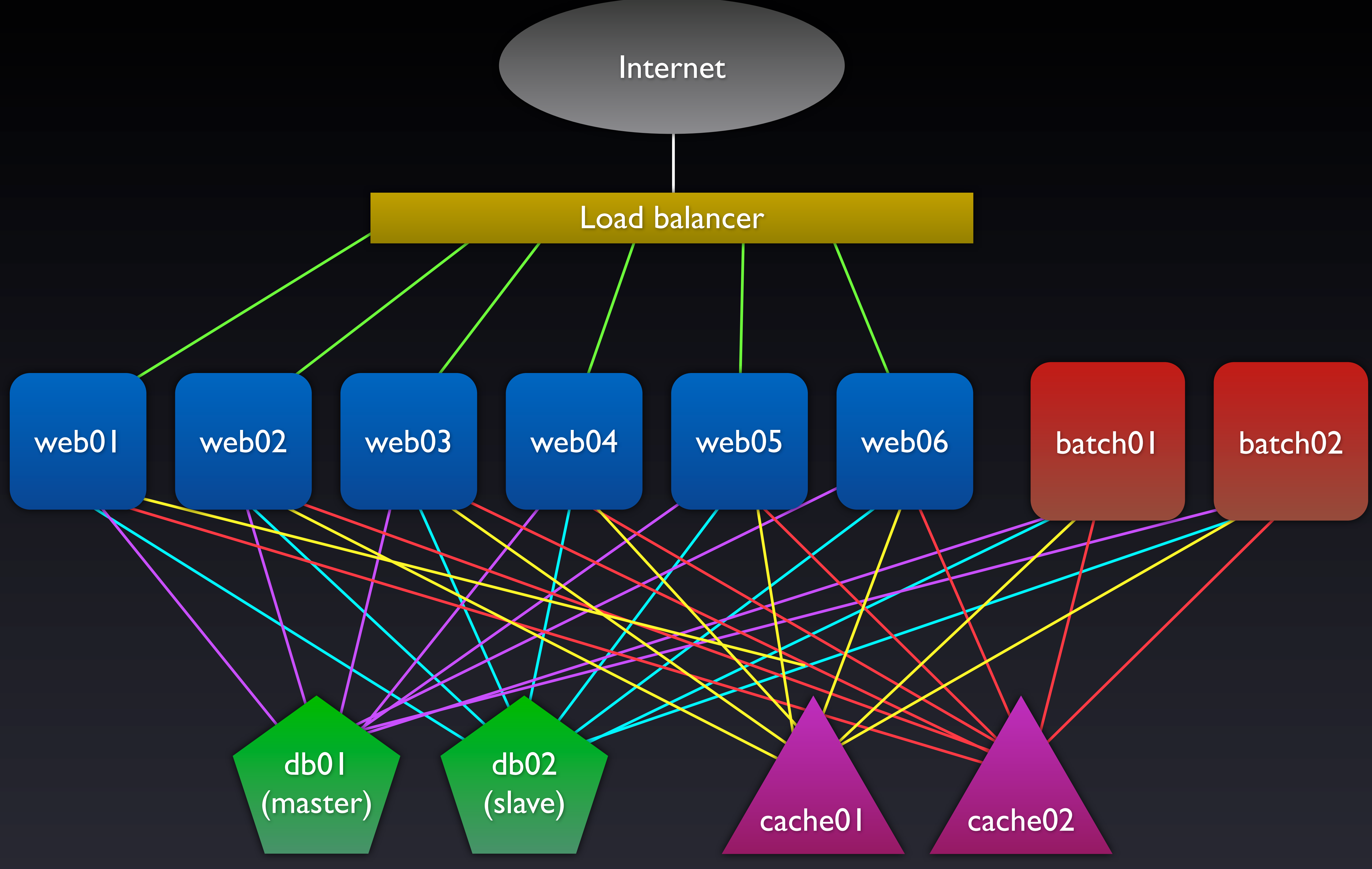
- Use a Source Code Management tool
  - This is non-negotiable!
  - Git > Subversion, but use *something*
  - Keep **everything** under version control
- Practice common repository management
  - "master", branches, and tags

# "A successful Git branching model"




http://nvie.com/posts/a-successful-git-branching-model/

# Classic web application infrastructure



# Deploying web application code

- Deploying should be a one-step process
  - Code always sourced from SCM repository
  - Be able to rollback quickly
- Always deploy (or at least tag) a numbered release
- Apply database changes
- Don’t take your web app down (if you can help it)

# Web server config and filesystem

```
<VirtualHost *:80>
    ServerName www.example.org
    DocumentRoot /home/web/site/current/public_html
</VirtualHost>
```

```
$ pwd
/home/web/site
web@dev:~/site$ ls -al
total 28
drwxrwxr-x 7 web web 4096 2012-11-24 21:55 .
drwxr-xr-x 3 web web 4096 2012-11-24 21:52 ..
drwxrwxr-x 2 web web 4096 2012-11-24 21:55 1.0.0
drwxrwxr-x 2 web web 4096 2012-11-24 21:55 1.0.1
drwxrwxr-x 2 web web 4096 2012-11-24 21:55 1.0.2
drwxrwxr-x 2 web web 4096 2012-11-24 21:55 1.1.0
drwxrwxr-x 5 web web 4096 2012-11-24 21:56 1.1.1
lrwxrwxrwx 1 web web    5 2012-11-24 21:55 current -> 1.1.1
```

# Deployment tools

- Build your own
  - Push/pull tarball from Amazon S3 service
  - rsync's code to local disk
- Capistrano
  - Written in Ruby
  - Performs "tasks" on "roles" of servers
- Ant, Phing, Deployinator (Etsy)

# Deploying database objects

- Your database always moves forward in time
- “up” and “down” changes
  - up_X.sql -- creates and modifies objects
  - down_X.sql -- undoes the changes in “up”
- Each change has a version number
  - Rails migrations, Doctrine

# Sample database up/down pair

***up_1353811475.sql***

```
create table user (
    id integer unsigned not null,
    username varchar(20) not null,
    password_hash varchar(100) not null,
    created datetime,
    last_updated datetime,
    last_login datetime,
    constraint user_pk primary key (id),
    constraint user_username_uk unique key (username));

insert into schema_version (
    num)
values (
    1353811475);
```

***down_1353811475.sql***

```
drop table user;

delete schema_version
 where num = 1353811475);
```

# Common gotchas

- Rotate your log files
  - Centralize them, too!
- Important to write robust, defensive code
- Be able to monitor everything
  - Munin, statsd, Graphite, etc.
- Design for failure
  - Have at least two of everything

# Other pearls of wisdom

# Coding standards

- Coding standards are ***insanely important***
- Ensures that all developers on a team write:
  - Code that looks the same
  - Is maintainable by anyone on the team
- These are not optional. You follow them. Period.

# Testing and QA in the real world

- When time and budgets are tight:
  - It's ***really*** easy to skimp on writing tests for your code
  - You just want to meet your deadline
  - Acceptance testing of a release can be shorter than normal
- Meeting client deadline > 100% code coverage

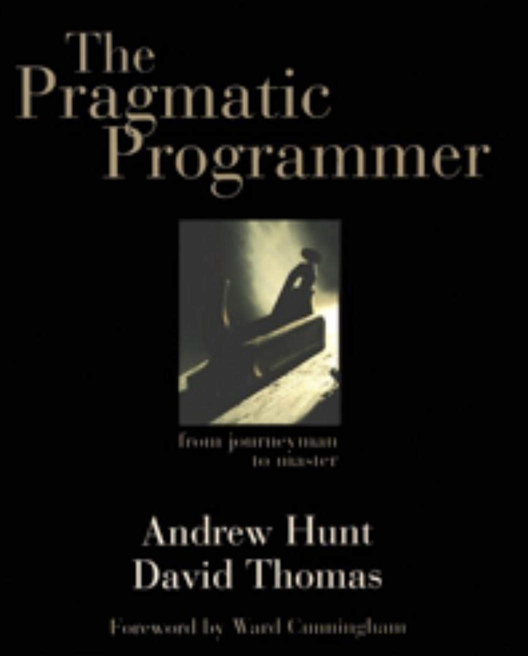
# Understanding the sysadmin side

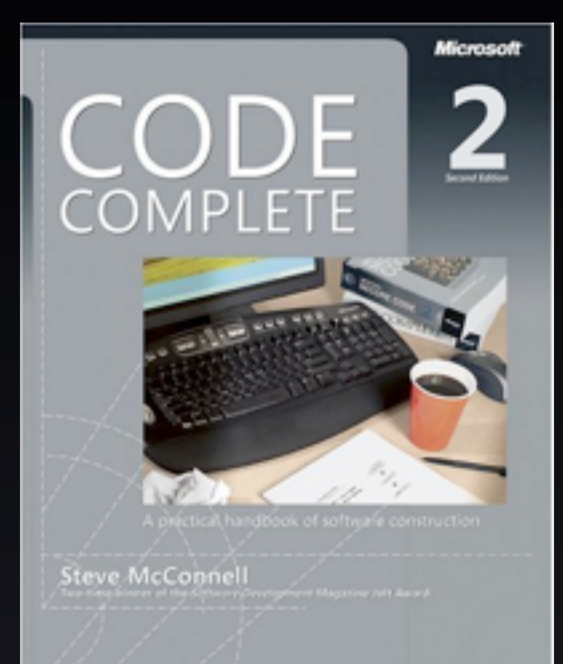
- Understanding systems administration has been a huge differentiator for me
  - The best devs I know understand this area well
  - When you write code, you think about what it does on a server
  - Deeper understanding and respect for writing efficient code
- So much easier now due to virtualization

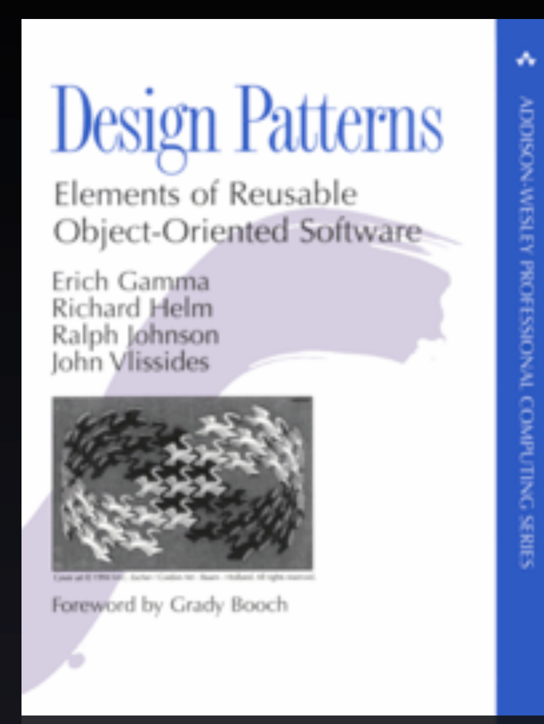
# Speak! Write!

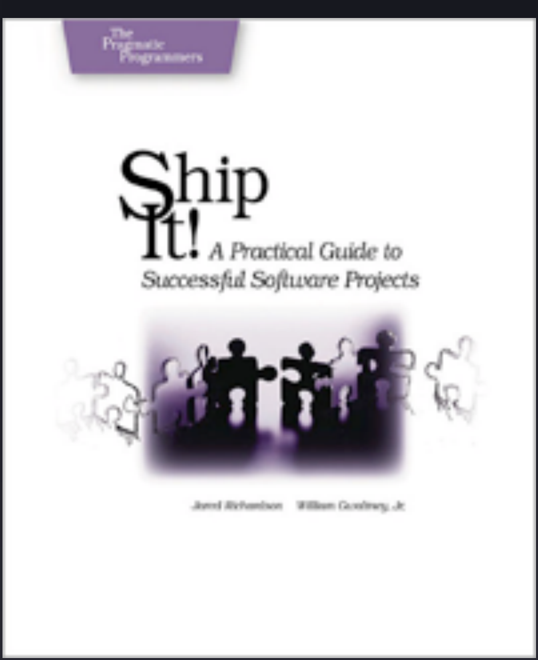
- Many software development conferences
  - You owe it to the community to share your knowledge
  - Adapt your work into presentations
- Write!
  - My former (and current) boss: “Publish or perish”

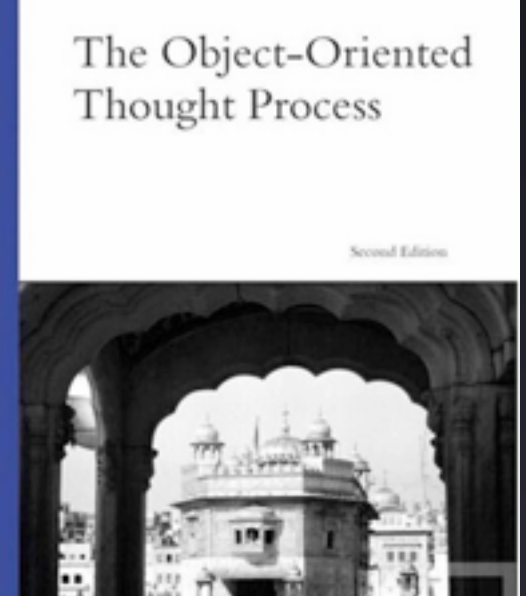
# Books!

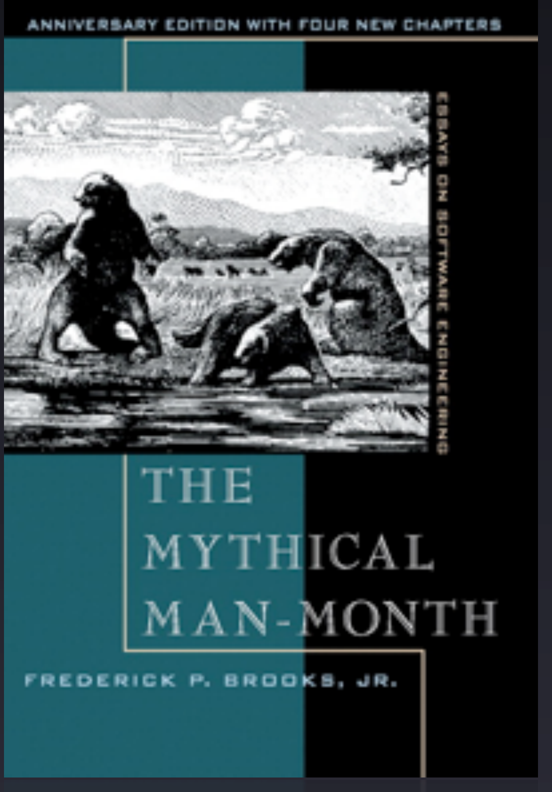


# References

- Code As Craft
  - http://codeascraft.etsy.com/
- PHPUnit
  - https://github.com/sebastianbergmann/phpunit/
- Grumpy Programmer's Guide to Building Testable Applications in PHP
  - http://www.grumpy-testing.com/

# Thanks!

brian@deshong.net

http://www.deshong.net/

brian@crowdtwist.com

http://www.crowdtwist.com/