Testable JavaScript

Architecting Your Application for Testability

Mark Ethan Trostler
Google Inc.
twitter: @zzoass
mark@zzo.com

What Is Testability?

Loose Coupling

Tight Focus

Minimal Tedium

No Surprises

H g va a m ds

Interfaces not Implementation

Swap Implementations

Write Tests Once

Work/Test in Parallel

Interfaces not Implementation

```
var UserRepository = {
    get: function(id) {}
  , save: function(user) {}
  , getAll: function() {}
  , edit: function(id, user) {}
  , delete: function(id) {}
  , query: function(query) {}
};
```

Interfaces not Implementation

Test the interface

```
function test(repo) {
  var id = 99, user = { id: id,  ... };
  repo.save(user);
  expect(repo.get(id)).toEqual(user);
}
```

Interfaces not Implementation

```
var UserRepoRedis = function(host, port, opt) {
    this.redis = redis.createClient({ ... });
};

UserRepoRedis.prototype =
                          Object.create(UserRepo);

UserRepoRedis.prototype.save =
                          function(user) { ... };
```

Interfaces not Implementation

- Object is interface - no initialization
- Implementation has constructor with injected dependencies
- Prototype is Object.create(Interface)
- Override with prototype functions

Interfaces not Implementation

Test the interface

```
function test(repo) {
   var user = { ... };
   repo.save(user);
   expect(repo.get(id)).toEqual(user);
}
var repo = new UserRepoRedis(host, port, opt);
test(repo);
```

Interfaces not Implementation

```
var UserRepoS3 = function(key, secret, bucket) {
    this.s3 = knox.createClient({...});
};

UserRepoS3.prototype = Object.create(UserRepo);

UserRepoS3.prototype.save =
                              function(user) { ... }
```

Interfaces not Implementation

Test the interface

```
function test(repo) {
  var id = 99, user = { id: id,  ... };
  repo.save(user);
  expect(repo.get(id)).toEqual(user);
}
var repo = new UserRepoS3(key, secret, bucket);
test(repo);
```

Single Responsibility Principle

Every interface should have a single responsibility, and that responsibility should be entirely encapsulated by the interface.

Responsibility = Reason To Change

Interface Segregation Principle

No object should be forced to depend on methods it does not use.

Interface Pattern

- Single Responsibility Principle
- Match Sets with Gets
- More Smaller / Fewer Bigger
- Interface Segregation Principle
- Test and Program to Interface Only

Using Interfaces

You've created nice interfaces - use them wisely!

```
// DO NOT DO THIS!
var UserController = function() {
  this.userRepo = new UserRepoRedis();
};
```

Using Interfaces

You've created nice interfaces - use them wisely!

```
// DO THIS - Inject the dependencies!
var UserController = function(userRepo) {
  this.userRepo = userRepo;
};
```

Liskov Substitution Principle

Any objects that implement an interface can be used interchangeably

Constructor Injection

All dependencies should be injected into your object's constructor*.

- Make dependencies explicit
- Loose coupling
- Cannot instantiate a non-usable Object

* Except:

- runtime dependencies
- objects with a shorter lifetime

Instantiating Implementations

So do all dependees need to provide fully initialized objects to their dependents when instantiating them?

NO - THEY DO NOT INSTANTIATE THEM!

Object Creation vs. Object Use

- ensures a loosely coupled system
- ensures testability

Object Creation vs. Object Use

creation

Happens one time at the Composition Root. All Objects* are created/wired together at this time

- startup
- request

use

Happens all the time throughout the application

Object Creation

What creates the objects?

Forces specification of dependencies and their lifespan

- DIY DI
- wire.js / cujojs
- Inverted
- Intravenous
- AngularJS

whole lotta others...

Cross-Cutting Concerns

What about the crap I might need?

- Logging
- Auditing
- Profiling
- Security
- Caching
- ....

"Cross-cutting concerns"

Cross-Cutting Concerns

```
// DO NOT DO THIS:
UserRepoRedis = function(...) {
    this.redis = ...
    this.logger = new Logger(); // or Logger.get()
    this.profiler = new Profiler(); // or Profiler.get()
};
```

- tightly coupled to Logger & Profiler implementations
- PITA to test

Cross-Cutting Concerns

```
// DO NOT DO THIS:
UserRepoRedis = function(..., logger, profiler) {
    this.redis = ...
    this.logger = logger;
    this.profiler = profiler;
};
```

Injection - so looks good BUT violating SRP!

Cross-Cutting Concerns

```
// DO NOT DO THIS:
UserRepoRedis.prototype.save =
  function(user) {
    logger.log('Saving user: ' + user);
    profiler.startProfiling('saveUser');
    ... do redis/actual save user stuff ...
    profiler.stopProfiling('saveUser');
    logger.log('that took: ' + (start - end));
  };
```

Cross-Cutting Concerns

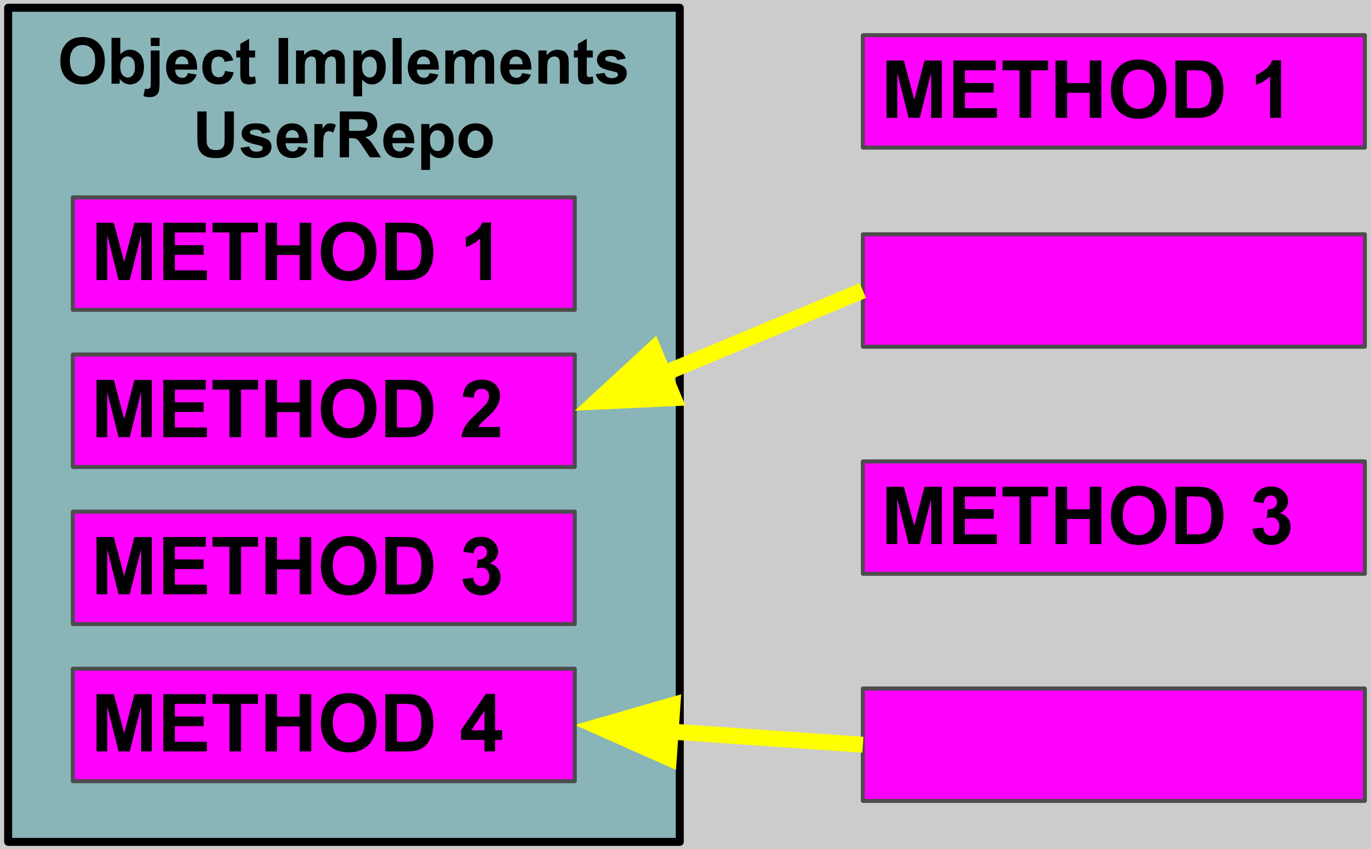
- Keep different functionality separate
- Single Responsibility Principle
- Interface Segregation Principle

Logging Interface and Implementation

```
var Logger = {    // Interface
    log: function(msg) {}
   , getLastLog: function() {}  // get it out!
};

var LoggerFile = function(file) {  // Implementation
    this.file = file;
}
LoggerFile.prototype = Object.create(Logger);
LoggerFile.prototype.log = function(msg) {
    this.file.write(msg);
};
```

Object Implements UserRepo MIXIN



Object Implements UserRepo

METHOD 1

METHOD 2

METHOD 3

METHOD 4

METHOD 1

METHOD 3

Mixin method

```
// Composition not inheritance
var MIXIN = function(base, extendme) {
    var prop;
    for (prop in base) {
        if (typeof base[prop] === 'function'
                  && !extendme[prop]) {
            extendme[prop] = base[prop].bind(base);
        }
    }
};
```

Decorator

```
var UserRepoLogger = function(repo, logger) {
    this.innerRepo = repo;
    this.logger = logger;
    MIXIN(repo, this);   // Mixin repo's methods
};
UserRepoLogger.prototype.save  =
    function(user) {
        this.logger.log('Saving user: ' + user);
        return this.innerRepo.save(user);
};
```

Decorator

```
// UserRepoLogger will Intercept save
//   all other methods will fall thru to
//       UserRepoRedis
// This userRepo implements the UserRepo
//      interface therefore can be used anywhere
var userRepo =
    new UserRepoLogger(
        new UserRepoRedis()
        , new LoggerFile()
);
```

Profile Interface and Implementation

```
var Profiler = {    // Interface
    start: function(id) {}
    , stop: function(id) {}
    , getProfile: function(id) {} // get it out!
};

var ProfilerTime = function() {  this.profiles = {}; };
ProfilerTime.prototype = Object.create(Profiler);
ProfilerTime.prototype.start = function(id) {
     this.profiles[id] = new Date().getTimestamp();
};
ProfilerTime.prototype.stop = function(id) { ... };
....
```

Decorator

```
var UserRepoProfiler = function(repo, profiler) {
    this.innerRepo = repo;
    this.profiler = profiler;
    MIXIN(repo, this);   // Mixin repo's methods
};
UserRepoProfiler.prototype.save  =  f(user) {
        this.profiler.start('save user');
        this.innerRepo.save(user);
        this.profiler.stop('save user');
};
```

Intercept

```
var redisRepo = new UserRepoRedis();
var profiler = new ProfilerTime();
var logger = new LoggerFile();
// Profiling UserRepo
var userRepoProf =
        new UserRepoProfiler(redisRepo, profiler);
// Logging Profiling UserRepo
var userRepo =
    new UserRepoLogger(userRepoProf, logger);
```

Testing Decorators

Create the Decorator and Test the Interface:

```
function testUserRepoLogger(repo) {
  var id = 99, user = { id: id,  ... },
      loggerMock = new LoggerMock(),
      testRepo =
         new UserRepoLogger(repo, loggerMock);
   testRepo.save(user);
   // verify loggerMock
}
```

Testing Decorators

```
var repo = new UserRepoMock();
var logger = new LoggerMock();
var profiler = new ProfileMock();
var userRepo = new UserRepoProfile(
    new UserRepoLogger(repo, logger), profiler);

// test UserRepo interface
testRepo(userRepo);
// verify logger and profiler mocks
```

Decorator Pattern

- Constructor accepts 'inner' Object of the same type and any other necessary dependencies.
- Mixin with 'inner' Object to get default behavior for non-decorated methods.
- Decorate necessary interface methods and (optionally) delegate to 'inner' Object.

Runtime/Short-lived Dependencies

```
var FindMatches = {
    matches: function(userid) {}
};
var FindMatchesDistance = f(userRepo) { ... };
FindMatchesDistance.prototype =
    Object.create(FindMatches);

var FindMatchesActivites = f(userRepo) { ... };
var FindMatchesLikes = f(userRepo) { ... };
```

Runtime/Short-lived Dependencies

```
// User Controller needs a findMatches
//      implementation - but which one???
var UserController = function(findMatches, ...) {
    ....
}
```

Inject all three?? What if I make more? What if other classes need a dynamic findMatch implementation?

Runtime/Short-lived Dependencies

```
var FindMatchFactory = {
    getMatchImplementation: function(type) {}
};

var FindMatchFactoryImp = function(repo) {
    this.repo = repo;
};
FindMatchFactoryImpl.prototype =
            Object.create(FindMatchFactory);
```

Runtime/Short-lived Dependencies

```
getMatchImplementation = function(type) {
    switch(type) {
        case 'likes':
            return new FindMatchesLikes(this.repo);
            break;
        ....
        default:
            return new FindMatchesActivites(this.repo);
    }
};
```

Runtime/Short-lived Dependencies

```
var UserController = function(findMatchFactory) {
        this.findMatchFactory = findMatchFactory;
 };

UserController.prototype = Object.create(Controller);

UserController.prototype.findMatches = function(type, uid) {
   var matcher =
        this.findMatchFactory.getMatchImplementation(type);
   return matcher.matches(uid);
};
```

Testing Abstract Factories

```
var matchTypes = [
    { name: 'likes', type: FindMatchesLikes }
    , ....
];
test(findMatchFactory) {
    matchTypes.forEach(function(type) {
        expect(
            findMatchFactory
            .getMatchImplementation(type.name)
            .toEqual(type.type);
   });
}
```

Mocking Abstract Factories

```
var TestFactoryImpl = function(expectedType, mockMatch) {
   this.expectedType = expectedType;
   this.mockMach      = mockMatch;
};

TestFactoryImpl.prototype = Object.create(FindMatchFactory);

TestFactoryImpl.prototype.getMatchImplementation(type) {
    expect(type).toEqual(this.expectedType);
    return this.mockMatch;
};
```

Abstract Factory Pattern

- Abstract factory translates runtime parameter -> Dependency
- Factory implementation created at Composition Root along with everything else
- Inject factory implementation into the Constructor for runtime dependencies
- Object uses injected factory to get Dependency providing Runtime Value

Testable JavaScript

- Composition not Inheritance (impl. lock-in)
- Program and Test to Interfaces / Interfaces are the API
- Create lots of small Single Responsibility Interfaces
- Decorate and Intercept Cross-Cutting Concerns
- Constructor Inject all Dependencies
- Inject Abstract Factory for run-time Dependencies

Testable JavaScript

Architecting Your Application for Testability

Mark Ethan Trostler
Google Inc.
twitter: @zzoass
mark@zzo.com

Ensuring Testability

Write Tests First