Table of Contents

Instructors Biographies

Miško Hevery

misko@hevery.com

Google, Inc.

… works as an Agile Coach at Google where he is responsible for coaching Googlers to maintain the high level of automated testing culture. This allows Google to do frequent releases of its web applications with consistent high quality. Previously he worked at Adobe, Sun Microsystems, Intel, and Xerox (to name a few), where he became an expert in building web applications in web related technologies such as Java, JavaScript, Flex and ActionScript. He is very involved in Open Source community and an author of several open source projects. Recently his interest in Test-Driven-Development turned into Testability Explorer (http://code.google.com/p/testability-explorer) and JsTestDriver (http://code.google.com/p/js-test-driver) with which he hopes to change the testing culture of the open source community.

Agenda

1. Setting up your environment
    1. Setting up Mercurial
    2. Setting up your favorite text editor or Eclipse
    3. Setting up JsTestRunner
2. Getting the source
    1. Checking out the source code
    2. Running the code
3. Understanding manual Dependency Injection
    1. Application wiring diagram
    2. Understanding the need for DI
4. Designing automated Dependency Injection framework
    1. Replacing the lost information of declared types
    2. Injecting types
    3. Injecting DOM elements
    4. Injecting events
5. Implementing the framework
6. Cleaning up the bootstrap wiring code
7. Discussion of before and after code

Objectives of the Course

Dependency Injection is a good practice even in dynamic languages. Because many dynamic languages do not have type declarations, building an automatic DI framework presents its own set of challenges. Furthermore, many concepts, such as scopes that are often associated with threads, change their meaning in most dynamic languages which are single threaded. In this tutorial, we will build a simple automatic DI framework and solve many of the challenges which are associated with lack of declared types in such languages.

- Understand the need for Dependency Injection
  - Dynamic languages are not special and still need Dependency Injection
- How to replace the lack of typing information in dynamic languages
- Injection is not limited to injecting object instances
  - Building our own domain specific language
  - The obvious: Injecting types
  - Thinking outside the box: Injecting DOM elements, and events
- Dependency Injection simplifies the wiring of the application logic
  - Comparing application bootstrap process before and after

1 Setting Up Environment

Before the tutorial we recommend that you complete all of the following steps:

1.1 Setting up Mercurial

The code for this project is located at:

- http://bitbucket.org/misko/misko-hevery-oopsla-09

You will need to install Mercurial for your platform:

- Home: http://mercurial.selenic.com/wiki/
- Download: http://mercurial.selenic.com/wiki/Download?action=show&redirect=BinaryPackages

1.2 Getting the Source

After installing a copy of Mercurial you can download the source using the following command:

```
$ hg clone https://misko@bitbucket.org/misko/misko-hevery-oopsla-09/
```

1.3 Setting up Your Favorite Text Editor or Eclipse

You will need you favorite Text Editor for editing JavaScript, HTML, and CSS. We recommend Eclipse which you can download and install from the link bellow. We have already set up an Eclipse workspace for you in the Mercurial repository.

- Download Eclipse: http://www.eclipse.org/downloads/
- Install JavaScript Editor: From Eclipse
  - Help → Install New Software
  - Select: Work With: Galileo
  - Select: Web, XML, and Java EE Development
  - Complete the installation by clicking the “Next” button
- Install JsTestDriver for Eclipse
  - Help → Install New Software
  - Select: Work With: and enter: http://js-test-driver.googlecode.com/svn/update/
  - Select JsTestDriver
  - Complete the installation by clicking the “Next” button
- Import the DI project which you downloaded with Mercurial.

1.4 Setting up Test Runner

NOTE: All commands need to be executed from the root of the source code which you downloaded with Mercurial.

The project comes with JavaScript tests which can be run from the command line as follows:

```
$ java -jar JsTestDriver.jar --port 4224
```

This will start a server on http://localhost:4224
Open your favorite browser (we don't recommend IE as it is slow) at http://localhost:4224 and click the "Capture This Browser" link.

From a new command line enter:

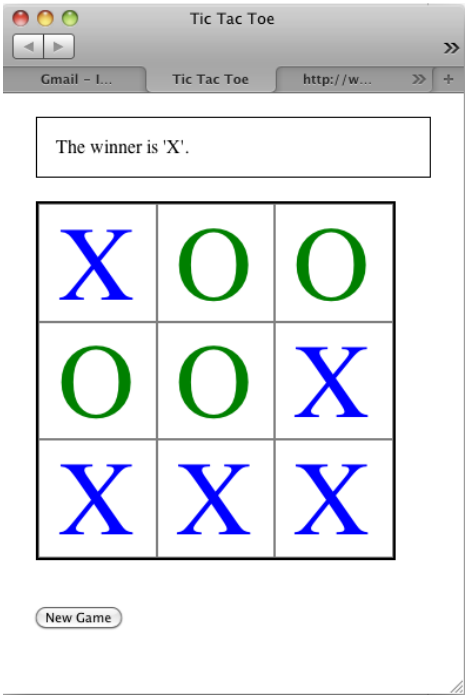
```
$ java -jar JsTestDriver.jar --tests all
```

You should see something like this:

```
$ java -jar JsTestDriver.jar --tests all
........
Total 8 tests (Passed: 8; Fails: 0; Errors: 0) (10.00 ms)
  Chrome 4.0.202.0 MacIntel: Run 8 tests (Passed: 8; Fails: 0; Errors
0) (10.00 ms)
```

2 Understanding Manual Dependency Injection

2.1 Application Wiring Diagram



The application is a simple game of Tic-Tac-Toe as shown on the right. The application is intentionally kept small for simplicity and consists of these components.

- HTML (tic-tac-toe.html)
  - #status: location of the status element.
  - #board: location of the board game.
  - #newGame: button reseting the game.
- Classes
  - Board: Responsible for updating the DOM to show current state of the board.
  - State: Internal representation of the board, which can determine a winner and reflects the DOM view of the board.
  - Status: Updates the status DOM with the player turn and shows the winner.
  - ManualDI: Bootstrap class which is responsible for wiring the application class together.
- Events
  - Board Click: fired when the user clicks on the board game.
  - New Game Click: fired when the user clicks on the “New Game Button.”

2.2 Understanding the Need for DI

A common misconception is that dynamic languages do not need Dependency Injection. The argument is that in dynamic languages one can easily change the method definition at run time. Therefore no method is untestable because all methods have seams where the normal flow of code execution can be diverted for the purpose of test isolation through monkey-patching. While DI is very useful for testing, it also aids the understanding of the code base. This benefit would be lost without DI. However, the main fallacy of this argument stems from the fact that code is global constant, and monkey-patching the code turns the global constants to mutable global state. While global constants don't present testability, maintainability, or understandability problems, mutable global state presents problems in all previously stated categories. It is true that while in static languages, such as Java, lack of DI spells death to testability, lack of DI in dynamic languages still allows testability through monkey-patching. However, it is not a best practice, and creates a problem of global state as stated above.

3 Designing Automated Dependency Injection Framework

The file `src/ManualDI.js` contains all of the wiring of the application. This is an error prone process as it can lead to mistakes which are hard to detect because there are no automated tests.

The goal is to completely replace this file with automatic DI. To achieve automatic DI we need to solve several issues as discussed in more detail next.

3.1 Replacing the Lost Information of Declared Types

The automatic Dependency Injection works by looking at a constructor of a class to determine the parameter types, then recursively looking at those types. Dynamic Languages usually don't contain this information and therefore we need a mechanism for declaring the missing information. To solve this problem, we will declare a class constant containing the missing meta-data information for the framework.

See `scr/Board.js` and notice the class constant `inject`

```
Board = function (board, state) {
  this.board = board;
  this.nextPiece = "X";
  this.state = state;
};

Board.inject = {
  constructor:["#board", ":Status"],
  scope: "singleton",
  listen:[
    {source:"#newGame", event:”click”, call:'reset'},
    {source:"#board.box", event:”click”,
                          call:'addPiece', args:["@row", "@col"]}
  ]
};
```

The `inject` meta-data contains information used to automatically construct the object graph. Notice that the references are prefixed with special characters:

- “:” inject an object of a given type
- “#” inject a DOM element with a given id (this is a jQuery selector)
- “@” inject a property of a given object

3.2 Injecting Types

The most important ability for any Dependency Injection framework is to instantiate and inject instances of a given type. We will use the following syntax to locate objects.

```
var injector = new Injector();
var board = injector.getInstance(“:State”);
```

The goal of this tutorial is to implement an Injector class which can read the meta-data information of classes and use it to instantiate the objects.

3.3 Injecting DOM Elements

In addition to injecting types, common in most static language DI frameworks, we would also like the ability to inject DOM elements, a common need in JavaScript code, using the jQuery selector syntax. Therefore, asking for `#board` is equivalent to `jQuery(document).find("#board")`.

3.4 Injecting Events

DOM element listeners are also common in JavaScript, and therefore, the ability to inject listeners is desired. Our meta-data contains the "listen" directive which specifies in declarative way the kind of events the class should listen on.

4 Implementing the Framework

The goal of the tutorial is to implement an automatic DI framework which can read the meta-data and use it to instantiate our application. We want to change the current `ManualDI.js` class from:

```
Main.prototype.bind = function() {
  var boardElement = this.doc.find("#board");
  var state = new State();
  var board = new Board(boardElement, state);
  var status = new Status(this.doc.find("#status"), state, function(){
      return board.nextPiece;
    });
  boardElement.find('.box').click(function(){
    var e = $(this);
    board.addPiece(e.attr('row'), e.attr('col'));
    status.update();
  });

  this.doc.find('#newGame').click(function(){
    board.reset();
    status.update();
  });
  board.reset();
};
```

to an automatic wiring DI as shown below:

```
Main.prototype.bind = function() {
  var injector = new Injector();
  var board = injector.getInstance(":Board");
  board.reset();
};
```

The implementation will be done in the following way:

- All code will be test driven with unit tests and test first design
- First, we will implement the ability to instantiate pure objects such as `State.js` which will allow us to replace `new State()` with `injector.getInstance(":State")`
- Then, we will add the ability to inject jQuery selectors which will allow us to replace the complex instantiation of `State` with `injector.getInstance(":Status")`
- Finally, we will add the ability to inject listeners which will allow us to replace the whole manual DI with `injector.getInstance(":Board")` which is the root object.

5 Comparing the Resulting Wiring Code

When done with implementing the framework, we will discuss:

- The benefits of simplified code
- The different paths each of us took in the implementation
- The advantages and disadvantages of each of the implementations
- How we could further improve and grow the framework