

Structuring a Client-Side App

6.170 Software Studio

Fall 2015

Eirik Bakke

We've called this lecture "Structuring a Client-Side App" not because the lessons here only apply to the client-side portion of an app, but because we'll be going through an example that's entirely client-side.

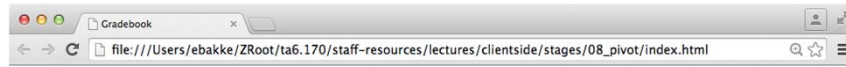
Objectives

- After today's class, you should understand what a **model** and a **view** is, and how to separate the two.
- You should understand the subscriber pattern.
- We will also see events, DOM building, unit testing, and the object construction idiom in action.
- Today's complete app example is relevant to both the Game of Life and the Fritter assignments

Today's Example

<http://shoutkey.com/even>

(don't look at the source
yet—spoilers! :-)



Gradebook

Grades by Assignment

JavaScript Warm-Up ▾

George	0
Diane	0
Thor	0
Lisa	0
Catherine	16
Sven	30
Average	8

Grades by Student

Catherine ▾

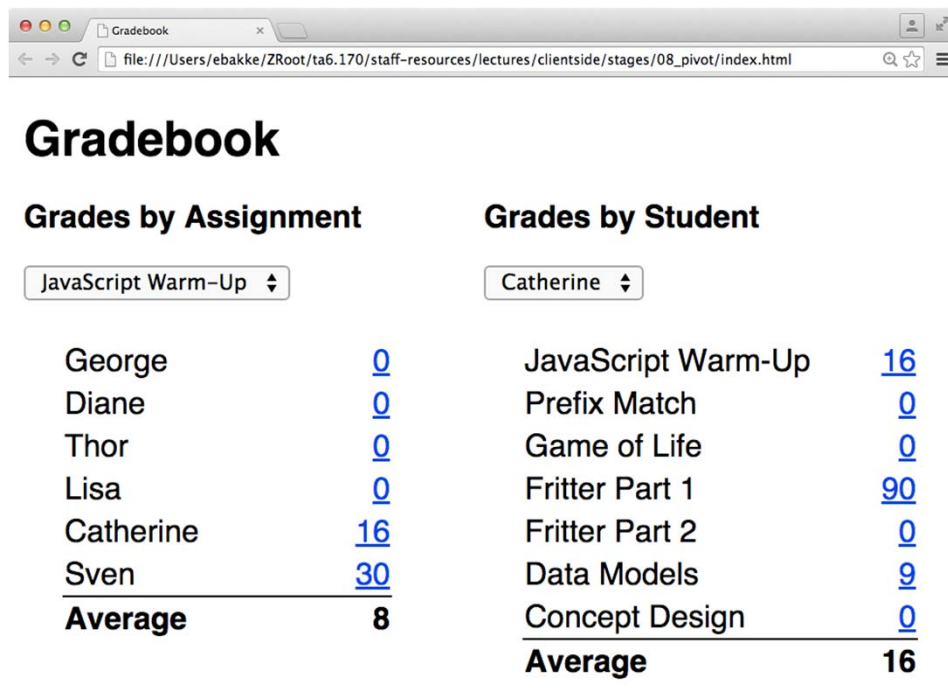
JavaScript Warm-Up	16
Prefix Match	0
Game of Life	0
Fritter Part 1	90
Fritter Part 2	0
Data Models	9
Concept Design	0
Average	16

Show what the app does. Point out that this is all client-side.

http://www.mit.edu/~ebakke/misc/6170_gradebook_example

http://www.mit.edu/~ebakke/misc/6170_gradebook_example/08_pivot/index.html

How would you structure this app?



Grades by Assignment		Grades by Student	
JavaScript Warm-Up ▾		Catherine ▾	
George	<u>0</u>	JavaScript Warm-Up	<u>16</u>
Diane	<u>0</u>	Prefix Match	<u>0</u>
Thor	<u>0</u>	Game of Life	<u>0</u>
Lisa	<u>0</u>	Fritter Part 1	<u>90</u>
Catherine	<u>16</u>	Fritter Part 2	<u>0</u>
Sven	<u>30</u>	Data Models	<u>9</u>
Average	8	Concept Design	<u>0</u>
		Average	16

Give the class 5 minutes to think about how they would structure this app, then discuss possible architectures on the board.

http://www.mit.edu/~ebakke/misc/6170_gradebook_example

Common User Interface Architectures

Spaghetti Code



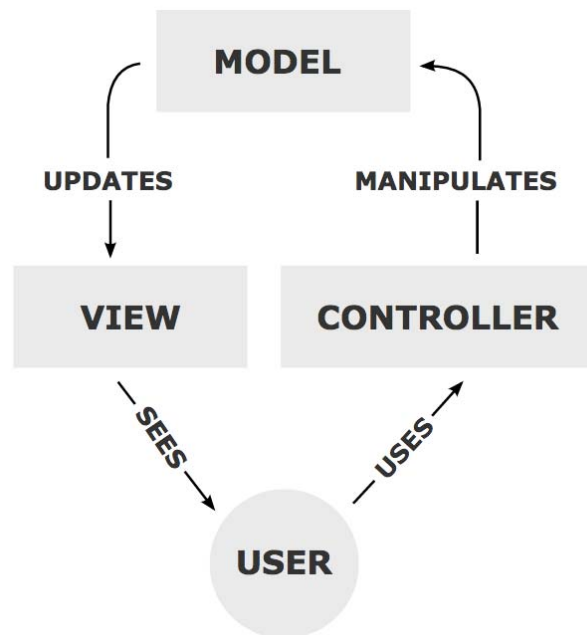
Spaghetti Code

- Mixes data and presentation
- Everything depends on everything
- State is everywhere
- Repeats logic
- Small features require changes throughout codebase



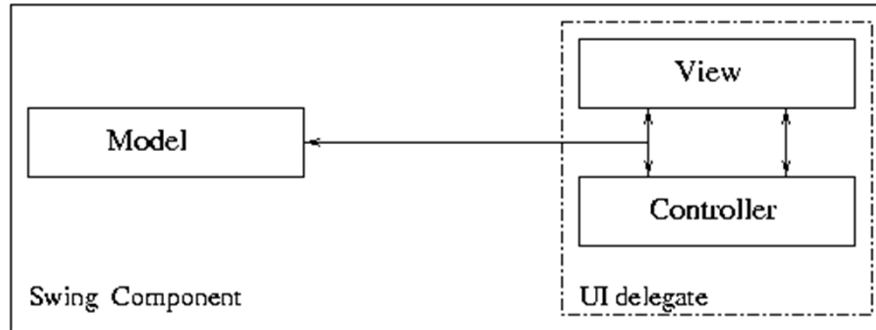
Spaghetti code originally referred to programs with tangled control structures (e.g. using GOTO everywhere). I'll use it to denote software with tangled dependencies. This list is not exhaustive.

Model/View/Controller (MVC)



This is just the classic MVC.

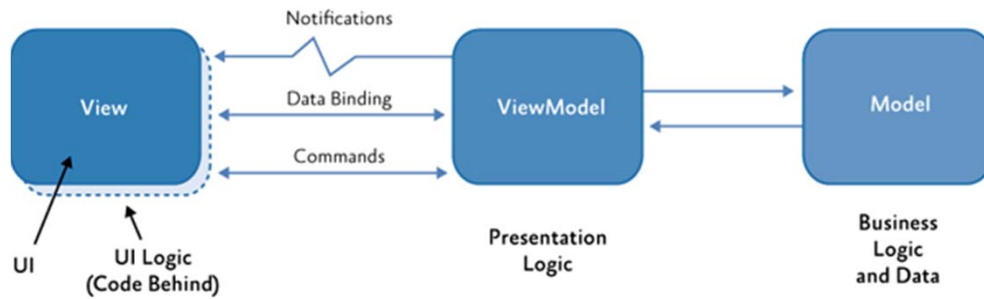
Model/ViewController



(from Java's Swing toolkit)

This will not be on the test. I'm just showing you a couple of architectures people have proposed.

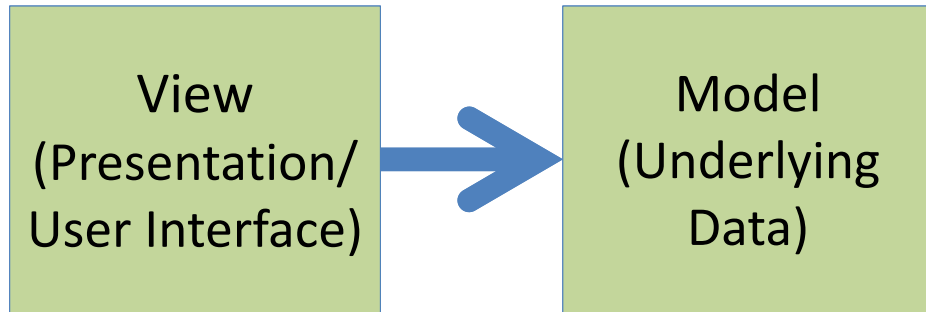
View/ViewModel/Model



(e.g. the Knockout framework)

This one adds another layer. The thing called “View” here is actually the DOM in this model.

The Point Is



- Keep the model and the view separate
- Your model should **not** depend on your view
- You should be able to write and test your model without writing a line of HTML or CSS (or other user interface code).

Explain clearly what is meant by “view” and “model”. The model is the data underlying your app. The view is the user interface. The view may have some state of its own (e.g. the pressed state of buttons, or the immediate content of text boxes), but the core state of your application, which may potentially be shared among multiple user interface components or even users, should be stored in a separate “model” part of your application that is independently testable and does not depend on anything else.

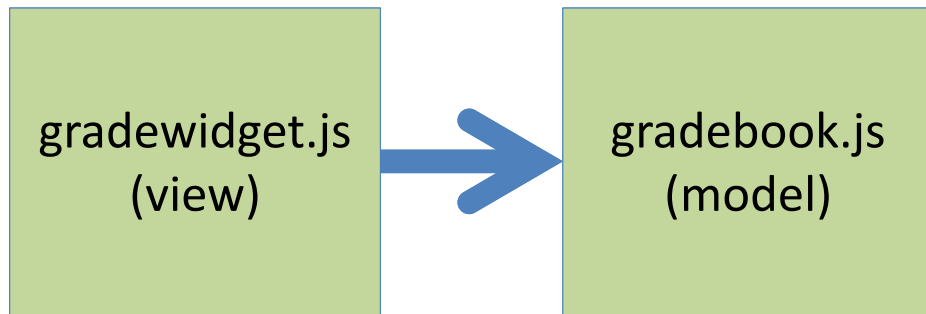
Back to Our Example

Grades by Assignment		Grades by Student	
JavaScript Warm-Up ▾		Catherine ▾	
George	<u>0</u>	JavaScript Warm-Up	<u>16</u>
Diane	<u>0</u>	Prefix Match	<u>0</u>
Thor	<u>0</u>	Game of Life	<u>0</u>
Lisa	<u>0</u>	Fritter Part 1	<u>90</u>
Catherine	<u>16</u>	Fritter Part 2	<u>0</u>
Sven	<u>30</u>	Data Models	<u>9</u>
Average	8	Concept Design	<u>0</u>
		Average	16

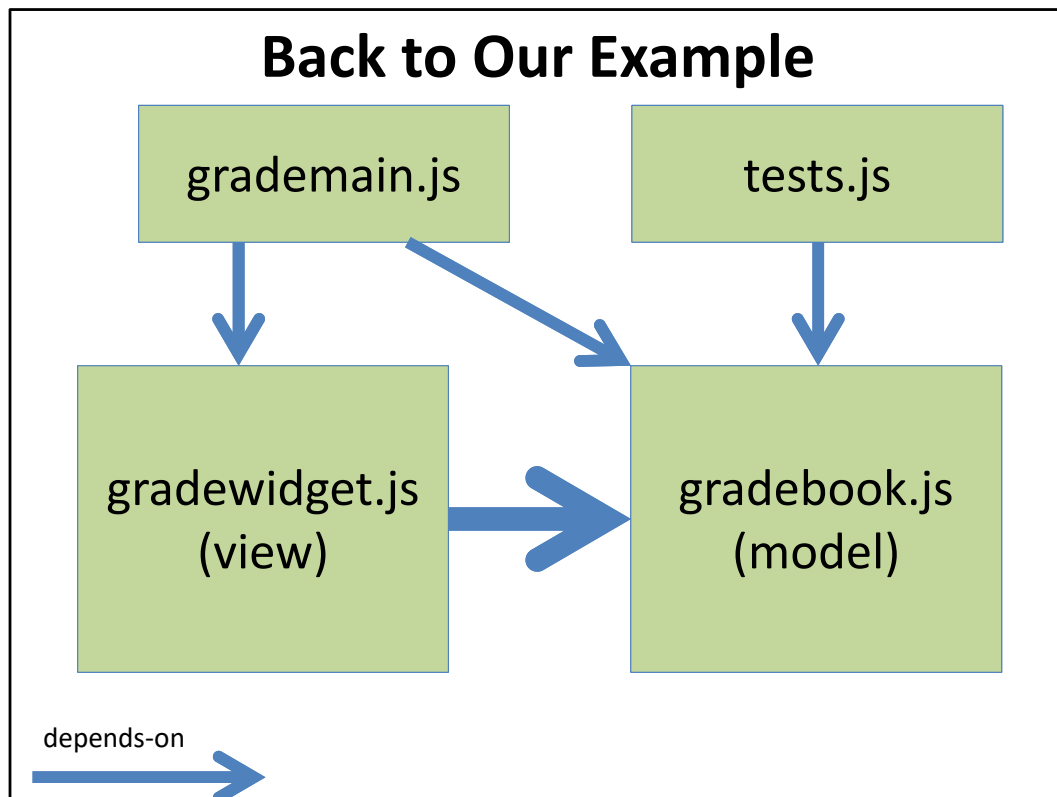
Give the class 5 minutes to think about how they would structure this app, then discuss possible architectures on the board.

http://www.mit.edu/~ebakke/misc/6170_gradebook_example

Back to Our Example



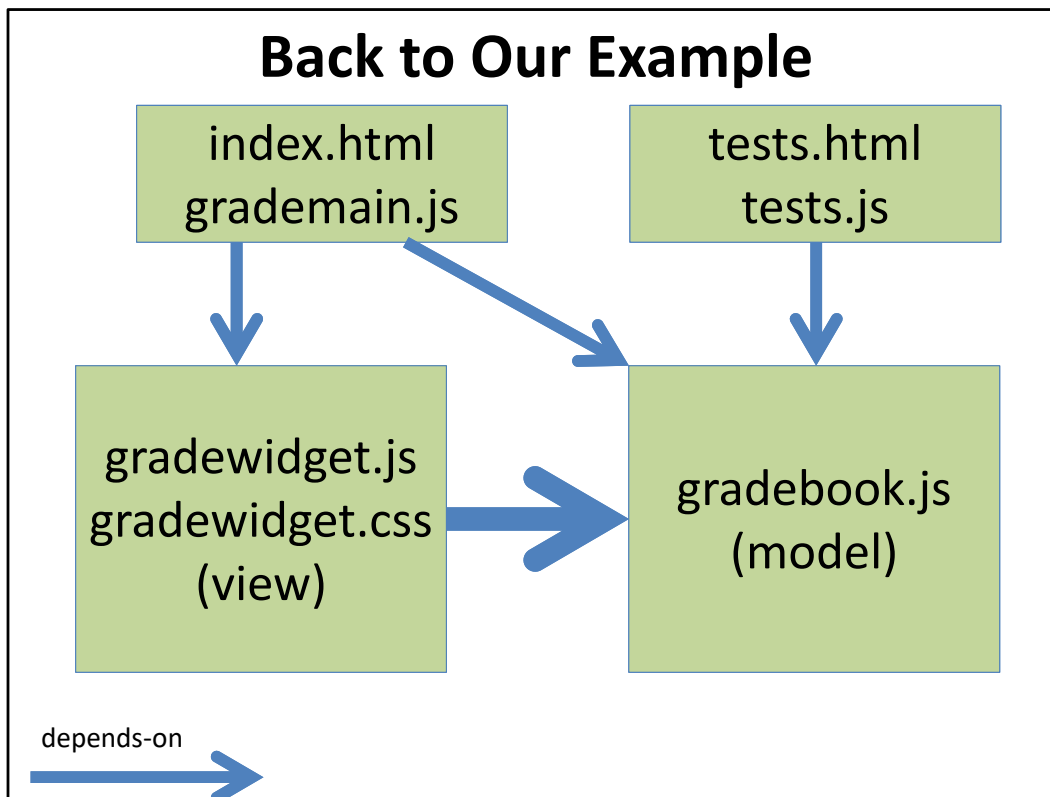
- Our app consists of two separate modules
- The model does not depend on the view
- We will write and rigorously test our model, gradebook.js, before doing any work on the view (gradewidget.js)



Actually, I lied when I said there were only two modules. There is another tiny little module which actually makes use of the `GradeWidget` and the `GradeBook`. You can consider it part of the view, but none of the other modules depend on it.

Note that the model can be tested entirely independently of the view.

On this chart, the arrows mean dependencies.



There's also the HTML and CSS files, but they're not too interesting in this example. Note, though, that the gradewidget has a CSS file that logically belongs in its module together with gradewidget.js.

Implementation Steps shoutkey.com/asparagus

1. Basic **model** class
2. Robust model class, **unit tests**
3. **Documented** model class
4. Read-only grade widget—the **view**
5. Editable grade widget
6. Two widgets, one model (buggy)
7. **Subscriber** pattern
8. Reconfigure one of the widgets

http://www.mit.edu/~ebakke/misc/6170_gradebook_example

https://github.com/eirikbakke/6170_gradebook_example/commits/master

Basic Model Class

shoutkey.com/a

[sparagus](#)

```
Gradebook = function() {  
  var that = Object.create(Gradebook.prototype);  
  var students = {}; // Keys are student names, values are always true.  
  // Keys are assignments, values are maps from student names to grades.  
  var assignments = {};  
  
  that.addStudent = function(student) {  
    students[student] = true;  
  };  
  that.addAssignment = function(assignment) {  
    assignments[assignment] = {};  
  };  
  that.setGrade = function(student, assignment, grade) {  
    assignments[assignment][student] = grade;  
  };  
  that.getGrade = function(student, assignment) {  
    return assignments[assignment][student] || 0;  
  };  
  that.getStudents = function() {  
    return Object.keys(students);  
  };  
  that.getAssignments = function() {  
    return Object.keys(assignments);  
  };  
  return that;  
}
```

Robust Model Class, Unit Tests

```
// Keys are assignments, values are maps from student names to grades
var assignments = {};
```

```
+ var checkStringArgument = function(obj) {
+   if ($.type(obj) !== "string" || obj.length === 0)
+     throw new Error("Expected a non-empty string argument");
+ };
+
+ var checkKey = function(map, key) {
+   checkStringArgument(key);
+   if (!map.hasOwnProperty(key))
+     throw new Error("No entry " + key);
+ };
+
+ that.addStudent = function(student) {
+   checkStringArgument(student);
+   students[student] = true;
+ };
+
+ that.addAssignment = function(assignment) {
+   checkStringArgument(assignment);
+   assignments[assignment] = {};
+ };
```

shoutkey.com/a-sparagus

```
that.setGrade = function(student, assignment) {
+   checkKey(students, student);
+   checkKey(assignments, assignment);
+   if ($.type(grade) !== "number" || !isFinite(grade))
+     throw new Error("Expected a numeric grade");
+   assignments[assignment][student] = grade;
+ };
+
+ that.getGrade = function(student, assignment) {
+   - return assignments[assignment][student];
+   + checkKey(students, student);
+   + checkKey(assignments, assignment);
+   + var ret = assignments[assignment][student];
+   + return ret === undefined ? 0 : ret;
+ };
```

Runtime checks. Repeat what DNJ said before: you shouldn't be checking for types and constraints everywhere, but you should absolutely check them at API boundaries. Especially when supplied objects go into the model state and errors may not otherwise be discovered until much later.

Robust Model Class, Unit Tests

```
Gradebook = function() {  
  var that = Object.create(Gradebook.prototype);  
  var students = {}; // Keys are student names, values are always true.  
  // Keys are assignments, values are maps from student names to grades.  
  var assignments = {};  
  
  that.addStudent = function(student) {  
    students[student] = true;  
  };  
  that.addAssignment = function(assignment) {  
    assignments[assignment] = {};  
  };  
  that.setGrade = function(student, assignment, grade) {  
    assignments[assignment][student] = grade;  
  };  
  that.getGrade = function(student, assignment) {  
    return assignments[assignment][student];  
  };  
  that.getStudents = function() {  
    return Object.keys(students);  
  };  
  that.getAssignments = function() {  
    return Object.keys(assignments);  
  };  
  return that;  
}
```

shoutkey.com/a/sparagus

```
    return Object.keys(assignments)  
  }  
  
  + Object.freeze(that);  
  return that;  
}
```

Mention Object.freeze here.

Robust Model Class, Unit Tests

```
QUnit.test("Simple gradebook tests", function(assert) {  
    var gradebook = Gradebook();  
    gradebook.addStudent("George");  
    gradebook.addStudent("Diane");  
    gradebook.addStudent("Thor");  
    gradebook.addStudent("Lisa");  
    gradebook.addStudent("Catherine");  
    gradebook.addStudent("Sven");  
    gradebook.addAssignment("JavaScript Warm-Up");  
    gradebook.addAssignment("Prefix Match");  
    gradebook.addAssignment("Game of Life");  
    gradebook.addAssignment("Fritter Part 1");  
    gradebook.addAssignment("Fritter Part 2");  
    gradebook.addAssignment("Data Models");  
    gradebook.addAssignment("Concept Design");  
    assert.deepEqual(gradebook.getStudents(),  
        ["George", "Diane", "Thor", "Lisa", "Catherine", "Sven"]);  
    assert.deepEqual(gradebook.getAssignments(), [  
        "JavaScript Warm-Up", "Prefix Match", "Game of Life", "Fritter Part 1",  
        "Fritter Part 2", "Data Models", "Concept Design"]);  
    assert.equal(gradebook.getGrade("George", "Prefix Match"), 0);  
    assert.raises(function () {  
        gradebook.getGrade("George2", "Prefix Match");  
    });  
    assert.raises(function () {  
        gradebook.getGrade("George", "Prefix Match2");  
    });  
    gradebook.setGrade("Sven", "JavaScript Warm-Up", 15);  
    assert.equal(gradebook.getGrade("Sven", "JavaScript Warm-Up"), 15);  
    gradebook.setGrade("Sven", "JavaScript Warm-Up", 30);  
    assert.equal(gradebook.getGrade("Sven", "JavaScript Warm-Up"), 30);  
});
```

shoutkey.com/a/sparagus

Go and show these unit tests in action.

https://www.mit.edu/~ebakke/misc/6170_gradebook_example/02_robust/tests.html

Spend about 10 minutes on QUnit tests.

Point out the “unit” part of unit tests. They’re testing a unit, that is to say, they are only testing a small piece of the program in isolation.

Unit tests are most valuable when written early in the development process; that’s when they will best help you catch bugs, especially as you make a lot of changes to your application.

Documented Model Class shoutkey.com/asparagus

```
+/**
+ * Create a Gradebook object. A Gradebook contains grades by students and
+ * assignments.
+ * @constructor
+ */
+
+Gradebook = function() {
+  /* 6.170's preferred constructor idiom (d
+  keyword). */
+  @@ -19,16 +24,33 @@ Gradebook = function() {
+    throw new Error("No entry " + key);
+  };
+
+  /**
+  + * Add a student.
+  + *
+  + * @param {String} student the name of the student
+  + */
+  that.addStudent = function(student) {
+    checkStringArgument(student);
+    students[student] = true;
+  };
+
+  /**
+  + * Add an assignment.
+  + *
+  + * @param {String} assignment the name of the assignment
+  + */
+
+  /**
+  + * Set the grade of a student for one assignment.
+  + *
+  + * @param {String} student the name of the student
+  + * @param {String} assignment the name of the assignment
+  + * @param {Number} grade the grade to set
+  + */
+  that.setGrade = function(student, assignment, grade) {
+    checkKey(students, student);
+    checkKey(assignments, assignment);
+    @@ -37,6 +59,11 @@ Gradebook = function() {
+      assignments[assignment][student] = grade;
+    };
+
+  /**
+  + * Get the grade of a student for one assignment.
+  + * @param {String} student the name of the student
+  + * @param {String} assignment the name of the assignment
+  + */
+  that.getGrade = function(student, assignment) {
```

Point out that I'm using the JSDoc format, with the caveat that it doesn't quite understand the `this = that` idiom (with some more annotations it would work).

Spend no more than 5 minutes on this.

Read-Only Grade Widget (the “view”)

Gradebook

Grades by Assignment

JavaScript Warm-Up ▾

George	0
Diane	0
Thor	0
Lisa	0
Catherine	16
Sven	30
Average	8

Read-Only Grade Widget (the “view”)

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8"/>
    <title>Gradebook</title>
    <link type="text/css" rel="stylesheet" href="modules/gradebook.css">
    <link type="text/css" rel="stylesheet" href="modules/gradewidget.css">
  </head>
</html>
<body>
  <h1>Gradebook</h1>
  <section>
    <div id="grades_by_assignment"></div>
  </section>

  <script type="text/javascript" src="lib/jquery-1.11.3.min.js"></script>

  <script type="text/javascript" src="modules/gradebook.js"></script>
  <!-- Depends on gradebook. -->
  <script type="text/javascript" src="modules/gradewidget.js"></script>
  <!-- Depends on gradebook and gradewidget. -->
  <script type="text/javascript" src="modules/grademain.js"></script>
</body>
```

Read-Only Grade Widget (the “view”)

```
/**
 * Install a GradeWidget in the specified DOM container. A GradeWidget
 * is a user interface for editing Gradebook data.
 * @param domContainer a jQuery wrapper around a single empty div
 *       element to install the GradeWidget in.
 * @param {Gradebook} gradebook the Gradebook object to use as a model
 *       for the data being displayed and edited by this GradeWidget.
 */
GradeWidget_install = function(domContainer, gradebook) {
  var dropdownElm = $("<select>");
  var tableElm = $("<table>");
  $.each(gradebook.getAssignments(), function (index, value) {
    dropdownElm.append($("<option>", { "value": value, text: value }));
  });
  domContainer.append($("<h2>Grades by Assignment</h2>"));
  domContainer.append(dropdownElm);
  domContainer.append(tableElm);

  var rebuild_table = function() { /* ...next slide... */ }
  rebuild_table();
  dropdownElm.change(rebuild_table);
}
```

Note here how the GradeWidget takes a domContainer and a model object. It does not install itself arbitrarily on the page.

Read-Only Grade Widget (the “view”)

```
var rebuild_table = function() {
  var newTableElm = $("<table class='GradeWidget'>");
  var dropdownValue = dropdownElm.val();
  var total = 0, N = 0;
  if (dropdownValue.length > 0) {
    $.each(gradebook.getStudents(), function (index, rowKey) {
      var grade = gradebook.getGrade(rowKey, dropdownValue);
      var tableRow = $("<tr>");
      tableRow.append($("<td>", { text: rowKey }));
      tableRow.append($("<td>", { text: grade }));
      newTableElm.append(tableRow);
      total += grade; N++;
    });
    newTableElm.append(
      $("<tr class='avg'>" +
        "<td>Average</td><td>" +
          (total / N).toFixed(0) +
        "</td></tr>"));
  }
  tableElm.replaceWith(newTableElm);
  tableElm = newTableElm;
}
```

Gradebook

Grades by Assignment

JavaScript Warm-Up ▾

George	0
Diane	0
Thor	0
Lisa	0
Catherine	16
Sven	30
Average	8

The main point here is that we are building DOM elements from a model object. There also exists client-side templating libraries that help with this.

Point out where we get the grade from—it comes from the model.

Constructing DOM “behind the scenes” (good practice).

Also discuss the technique of regenerating the entire view every time there is an update (less complexity in code, but can upset UI state such as scroll position, focus, and tooltip state).

```

var grade = gradebook.getGrade(rowKey);
var tableRow = $("<tr>");
tableRow.append("<td>", { text: rowKey });
- tableRow.append("<td>", { text: grade });
+ var gradeCell = "<td>";
+ if (rowKey !== currentlyEditedRowKey) {
+   var gradeElm = "<a>", { text: grade, href: "#" });
+   gradeCell.append(gradeElm);
+   gradeCell.click(function() {
+     acceptCurrentEdit();
+     currentlyEditedRowKey = rowKey;
+     rebuild_table();
+     if (gradeInputElm !== null) {
+       gradeInputElm.select();
+       gradeInputElm.focus();
+     }
+   });
+ } else {
+   gradeCell.append(gradeInputElm);
+   gradeInputElm.val(grade);
+ }
+ tableRow.append(gradeCell);
+ newTableElm.append(tableRow);
+ total += grade; N++;
});

```

```

+ var gradeInputElm = "<input type='number'>";

```

```

var rebuild_table = function() {
+ gradeInputElm.detach();
var newTableElm = "<table class='
var dropdownValue = dropdownElm.val
var total = 0, N = 0;

```

Editable Grade Widget

Don't go into too much detail here; just point out that as part of our refresh, we assign a click handler to each grade, and put an input field in the right position if any grade has previously been clicked.

About `detach()`: jQuery will automatically detach any listeners when an element is removed from the DOM. Use `detach()` to remove `gradeInputElm` without removing its listeners, since we'll be reusing it.

Editable Grade Widget

```
+ var gradeInputElm = $("<input type='number'>");
+ var currentlyEditedRowKey = null;
+ var acceptCurrentEdit = function() {
+   if (currentlyEditedRowKey !== null) {
+     var student = currentlyEditedRowKey;
+     currentlyEditedRowKey = null;
+     var assignment = dropdownElm.val();
+     var gradeInput = gradeInputElm.val();
+     if (gradeInput.length > 0) {
+       gradebook.setGrade(student, assignment, parseInt(gradeInput));
+     }
+     rebuild_table();
+   }
+ };
+ gradeInputElm.keydown(function (evt) {
+   // Enter or tab keys.
+   if (evt.keyCode === 13 || evt.keyCode === 9) {
+     // Avoid the default focus-traversing behavior of the tab key.
+     evt.preventDefault();
+     acceptCurrentEdit();
+   }
+ });
```

Editable Grade Widget

```
+ var gradeInputElm = $("<input type='number'>");
+ var currentlyEditedRowKey = null;
+ var acceptCurrentEdit = function() {
+   if (currentlyEditedRowKey !== null) {
+     var student = currentlyEditedRowKey;
+     currentlyEditedRowKey = null;
+     var assignment = dropdownElm.val();
+     var gradeInput = gradeInputElm.val();
+     if (gradeInput.length > 0) {
+       gradebook.setGrade(student, assignment, parseInt(gradeInput));
+     }
+     rebuild_table();
+   }
+ };
+ gradeInputElm.keydown(function (evt) {
+   // Enter or tab keys.
+   if (evt.keyCode === 13 || evt.keyCode === 9) {
+     // Avoid the default focus-traversing behavior of the tab key.
+     evt.preventDefault();
+     acceptCurrentEdit();
+   }
+ });
```

new grade set on the **model**,
not on the view!

The most important point here is that the setting of the grade is done on the model, not somewhere in the view. The view gets updated in turn when `rebuild_table()` is called.

Two Widgets, One Model (buggy)

	index.html
	@@ -10,7 +10,8 @@
10	<body>
11	<h1>Gradebook</h1>
12	<section>
	- <div id="grades_by_assignment"></div>
13	+ <div id="grades_by_assignment1"></div>
14	+ <div id="grades_by_assignment2"></div>
15	</section>
	@@ -19,6 +19,7 @@ \$(fu
19	gradebook.setGrade("Catherine", "JavaScript Warm-Up", 16);
20	gradebook.setGrade("Catherine", "Fritter Part 1", 90);
21	
	- GradeWidget_install(\$("#grades_by_assignment"), gradebook);
22	+ GradeWidget_install(\$("#grades_by_assignment1"), gradebook);
23	+ GradeWidget_install(\$("#grades_by_assignment2"), gradebook);
24	})

There's also a little bit of CSS here.

What's the bug?

Gradebook

Grades by Assignment

JavaScript Warm-Up ↕

George	<u>0</u>
Diane	<u>0</u>
Thor	<u>0</u>
Lisa	<u>0</u>
Catherine	<u>16</u>
Sven	<u>30</u>
Average	8

Grades by Assignment

JavaScript Warm-Up ↕

George	<u>0</u>
Diane	<u>0</u>
Thor	<u>0</u>
Lisa	<u>0</u>
Catherine	<u>16</u>
Sven	<u>30</u>
Average	8

Ask the class what is wrong. Get a good answer before moving on.

Challenge: Fix the Bug

tinyurl.com/pq9gm8v

Gradebook

Grades by Assignment

JavaScript Warm-Up ↕

George	0
Diane	0
Thor	0
Lisa	0
Catherine	16
Sven	30
Average	8

Grades by Assignment

JavaScript Warm-Up ↕

George	0
Diane	0
Thor	0
Lisa	0
Catherine	16
Sven	30
Average	8

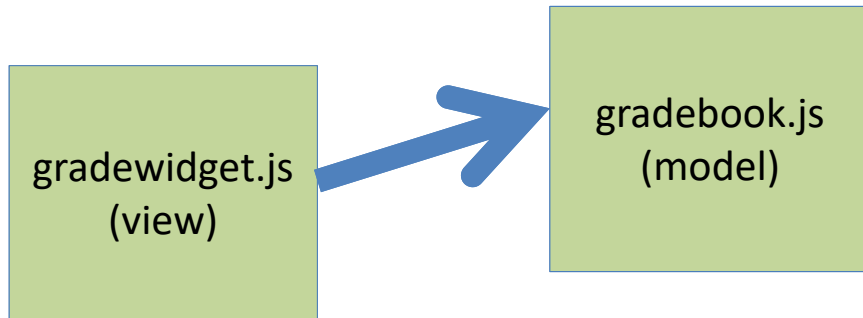
Tell students not to look at the solution yet.

http://www.mit.edu/~ebakke/misc/6170_gradebook_example/challenge.zip
<http://tinyurl.com/pq9gm8v>

Might need more than 5 minutes on this one; though students can continue to try to implement the solution as I move along.

Subscriber Pattern

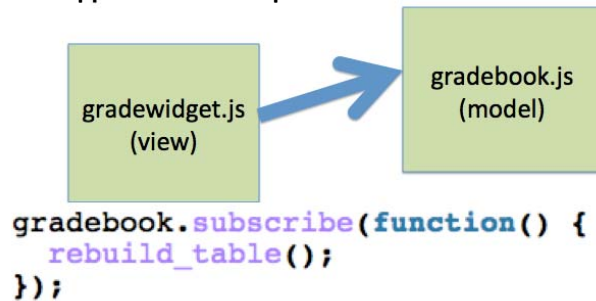
```
/**
 * Subscribe to changes to this object.
 * @param subscriber a function that is called
 *                 whenever the Gradebook is changed
 */
that.subscribe = function(subscriber)
```



```
gradebook.subscribe(function() {  
    rebuild_table();  
});
```


Subscriber Pattern

- The subscriber pattern is used to notify the view of changes in the model, **without requiring the model to depend on the view**
- Subscribers are analogous to DOM event handlers; except the events are on changes to your model, and your model class is responsible for managing and calling subscribers.
- Also known as “observer” pattern or “publish-subscribe” pattern



Stress that we are able to avoid making the model dependent on the view.

Compare subscribers to event handlers.

Fixing the bug: Subscriber Pattern

```
13 // Keys are assignments, values are maps from student names to grades
14 var assignments = {};
15
16 + var subscribers = [];
17 + /**
18 +  * Subscribe to changes to this object.
19 +  * @param subscriber a function that is called when the object has
20 +  *    changed
21 +  */
22 + that.subscribe = function(subscriber) {
23 +   subscribers.push(subscriber);
24 + };
25 + var publishChanges = function() {
26 +   var i;
27 +   for (i = 0; i < subscribers.length; i++)
28 +     subscribers[i]();
29 + };
30 +
@@ -32,6 +47,7 @@ Gradebook = function() {
47   that.addStudent = function(student) {
48     checkStringArgument(student);
49     students[student] = true;
50 +   publishChanges();
51   };
52
53   /**
@@ -42,6 +58,7 @@ Gradebook = function() {
58   that.addAssignment = function(assignment) {
59     checkStringArgument(assignment);
60     assignments[assignment] = {};
61 +   publishChanges();
62   };
63
64   /**
@@ -57,6 +74,7 @@ Gradebook = function() {
74   if ($.type(grade) !== "number" || isNaN(grade))
75     throw new Error("Expected a numeric grade");
76   assignments[assignment][student] = grade;
77 +   publishChanges();
78   };
79
80   /**
```

gradebook.js

Fixing the bug: Subscriber Pattern

```
92 +  
93 + gradebook.subscribe(function() {  
94 +   acceptCurrentEdit();  
95 +   rebuild_table();  
96 +   // Ideally we should also update the dropdown here.  
97 + });
```

gradewidget.js

Reconfigure One of the Widgets

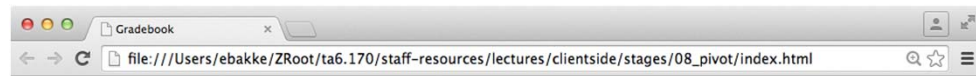
```
38     var tableElm = $("<table>");
39 - $.each(gradebook.getAssignments(), function (index, value) {
40 + $.each(getDropdownChoices(), function (index, value) {
41     dropdownElm.append($"<option>", { "value": value, text: value });
42 });
43 - domContainer.append($"<h2>Grades by Assignment</h2>");
42 + domContainer.append($"<h2>", { text: getHeading() });
43     domContainer.append(dropdownElm);
44     domContainer.append(tableElm);
45     var gradeInputElm = $("<input type='number'>");
@@ -27,7 +51,7 @@ GradeWidget_install =
51     var assignment = dropdownElm.val();
52     var gradeInput = gradeInputElm.val();
53     if (gradeInput.length > 0) {
54 -     gradebook.setGrade(student, assignment, parseInt(gradeInput));
54 +     setGrade(assignment, student, parseInt(gradeInput));
55     }
56     rebuild_table();
57 }
@@ -51,8 +75,8 @@ GradeWidget_install =
75     var dropdownValue = dropdownElm.val();
76     var total = 0, N = 0;
77     if (dropdownValue.length > 0) {
78 -     $.each(gradebook.getStudents(), function (index, rowKey) {
79 -     var grade = gradebook.getGrade(rowKey, dropdownValue);
78 +     $.each(getRowKeys(), function (index, rowKey) {
79 +     var grade = getGrade(dropdownValue, rowKey);
80     var tableRow = $("<tr>");
81     tableRow.append($"<td>", { text: rowKey });
82     var gradeCell = $("<td>");
```

We can generalize the GradeWidget a bit so that grades can be displayed either by assignment or by student. That way we can show off a useful reason why you might want two widgets based on the same model. (The exact logic in this part of the example is not so important.)

Reconfigure One of the Widgets

```
9 + * @param {boolean} byAssignment if true, show grades by assignment, otherwise,
10 + *   show them by student.
11 + */
12 GradeWidget_install =
13 - function(domContainer, gradebook)
14 + function(domContainer, gradebook, byAssignment)
15 {
16 + var getHeading = function() {
17 +   return byAssignment ? "Grades by Assignment" : "Grades by Student";
18 + }
19 + var getDropdownChoices = function() {
20 +   return byAssignment ? gradebook.getAssignments() : gradebook.getStudents();
21 + }
22 + var getRowKeys = function() {
23 +   return byAssignment ? gradebook.getStudents() : gradebook.getAssignments();
24 + }
25 + var getGrade = function(dropdownValue, rowKey) {
26 +   return byAssignment
27 +     ? gradebook.getGrade(rowKey, dropdownValue)
28 +     : gradebook.getGrade(dropdownValue, rowKey);
29 + }
30 + var setGrade = function(dropdownValue, rowKey, newGrade) {
31 +   if (byAssignment) {
32 +     gradebook.setGrade(rowKey, dropdownValue, parseInt(newGrade));
33 +   } else {
34 +     gradebook.setGrade(dropdownValue, rowKey, parseInt(newGrade));
35 +   }
36 + }
```

Reconfigure One of the Widgets



Gradebook

Grades by Assignment

JavaScript Warm-Up ▾

George	0
Diane	0
Thor	0
Lisa	0
Catherine	16
Sven	30
Average	8

Grades by Student

Catherine ▾

JavaScript Warm-Up	16
Prefix Match	0
Game of Life	0
Fritter Part 1	90
Fritter Part 2	0
Data Models	9
Concept Design	0
Average	16

Conclusion

- Keep your app's data (the “model”) separate from its user interface (the “view”)
- The model should not depend on the view
- Write and test the model, with unit tests and runtime checks, before writing any user interface code
- The subscriber pattern can be used to notify the view of changes in the model, without requiring the model to depend on the view
- When the user edits data, apply the change to the model, not the view

A Game of Life solution organized like this can be done in less than 300 lines of JavaScript (+tests).