CS 213 – Software Methodology
Spring 2017

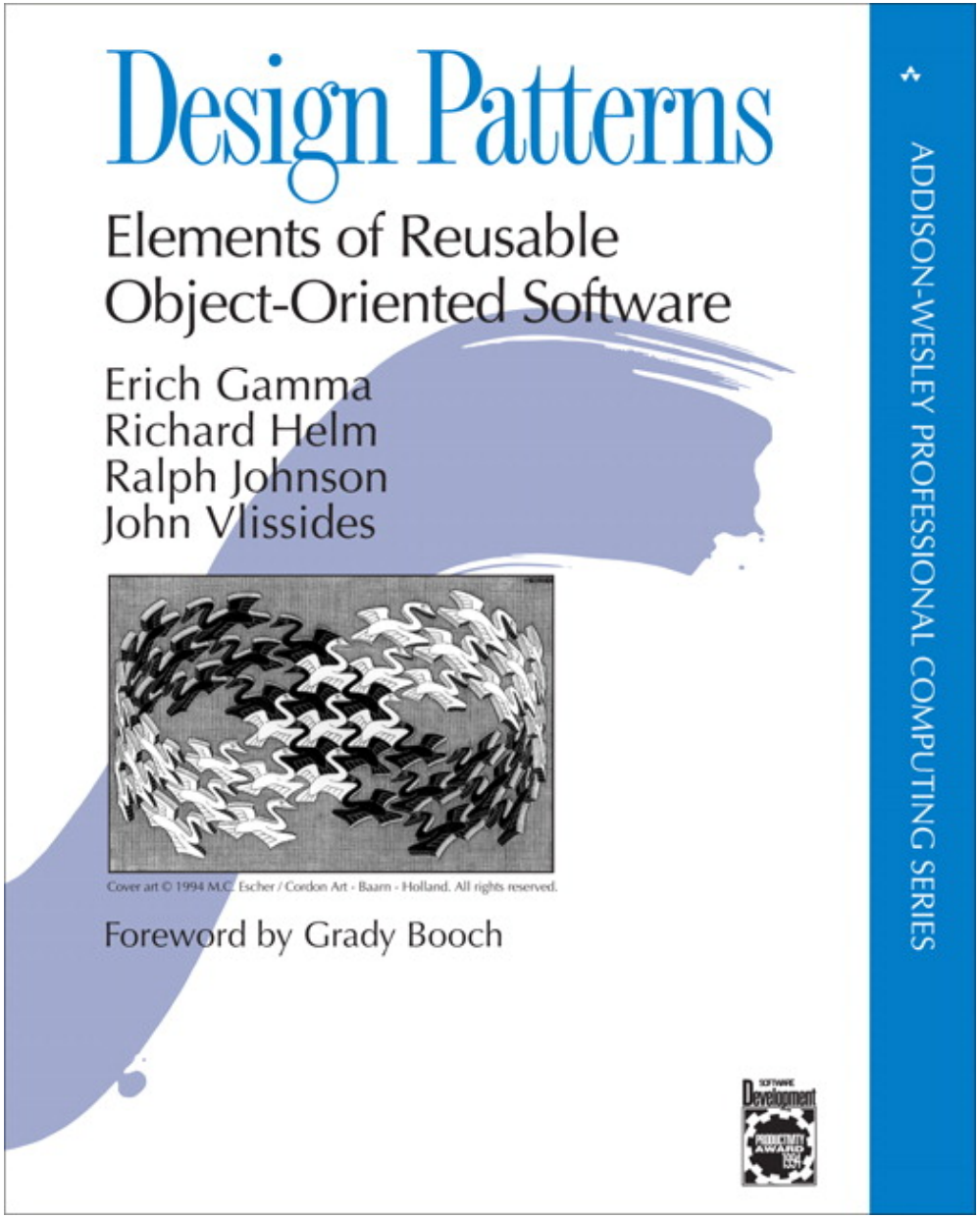
Sesh Venugopal

Lecture 17 – Mar 23

Design Patterns – 1

State and Singleton Patterns

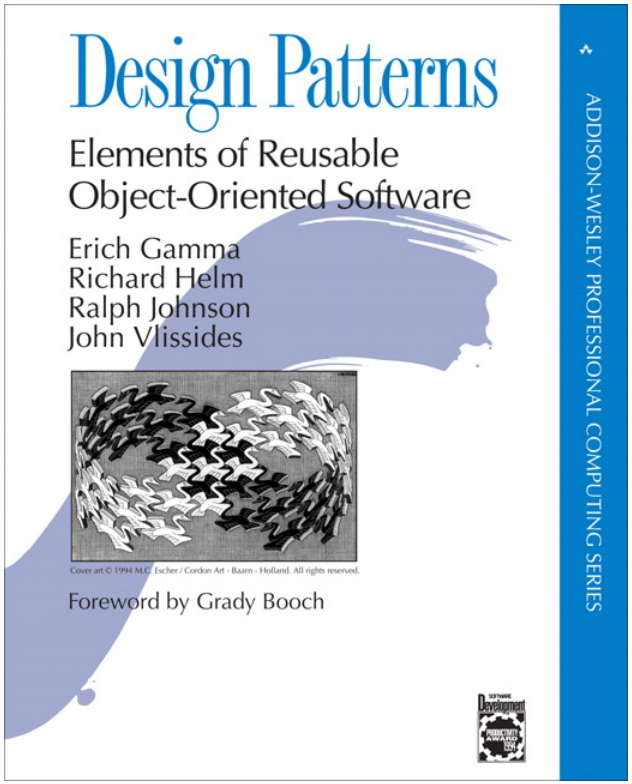
Illustration: State-Based Calculator


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Categories of Patterns

- Design patterns are classified into three categories:
  - **Creational** patterns: to do with the object creation process
  - **Structural** patterns: to do with the static composition and structure of classes and objects
  - **Behavioral** patterns: to do with the dynamic interaction between classes and objects

Creational Patterns



Abstract Factory (87) Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

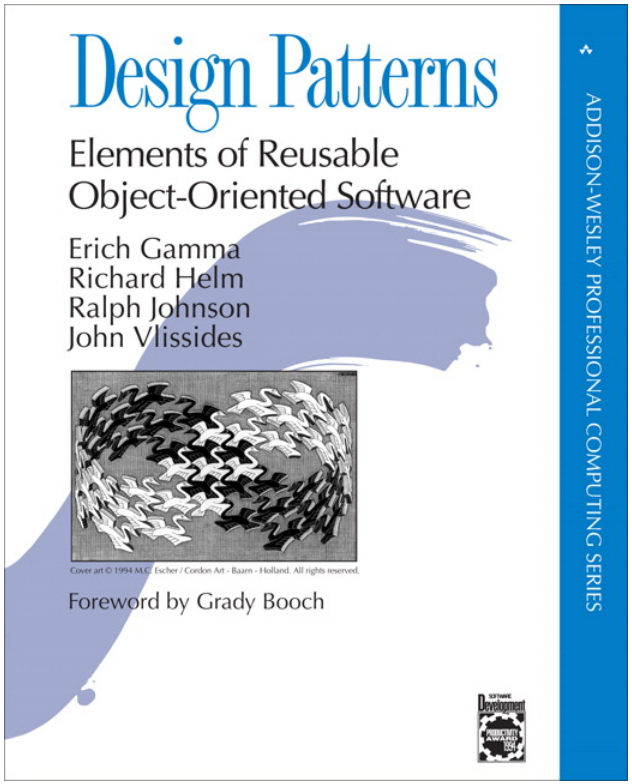
Builder (97) Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Factory Method (107) Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Prototype (117) Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Singleton (127) Ensure a class only has one instance, and provide a global point of access to it.

Structural Patterns



Adapter (139) Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Bridge (151) Decouple an abstraction from its implementation so that the two can vary independently.

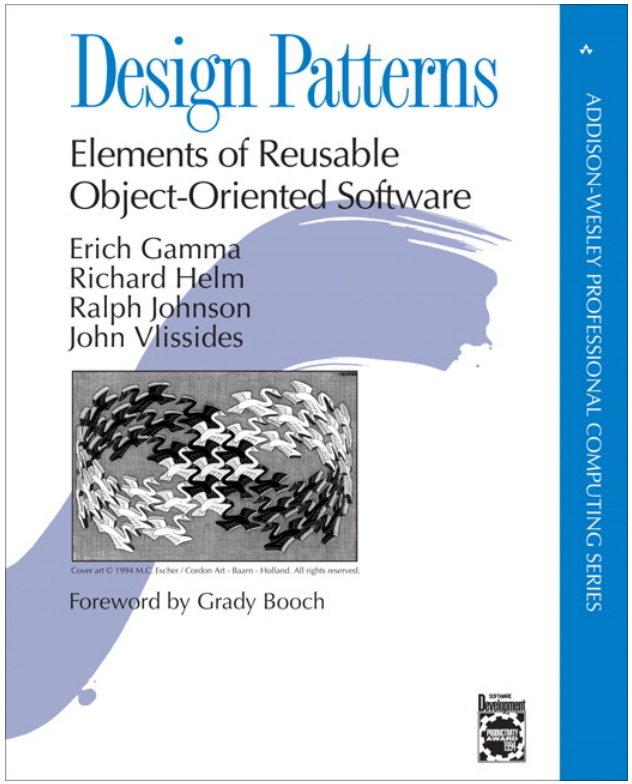
Composite (163) Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Decorator (175) Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Facade (185) Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Flyweight (195) Use sharing to support large numbers of fine-grained objects efficiently.

Proxy (207) Provide a surrogate or placeholder for another object to control access to it.



Behavioral Patterns

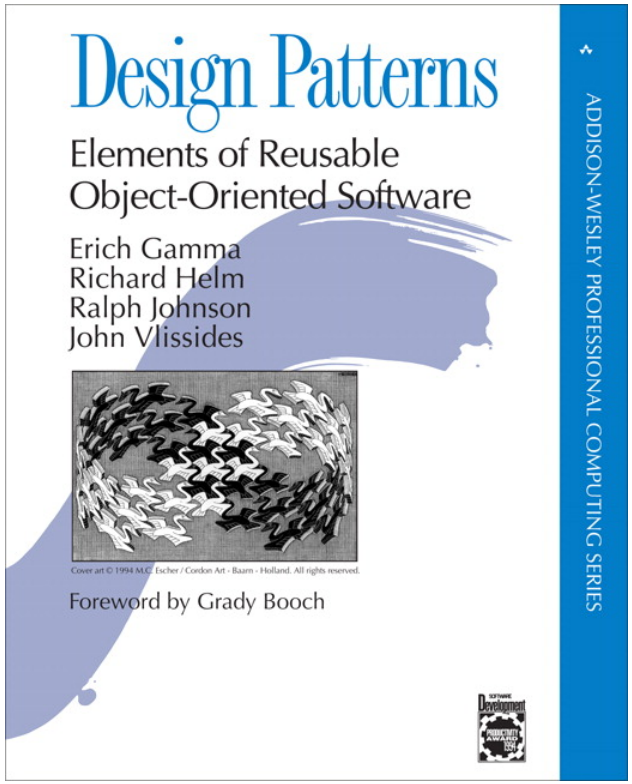
Chain of Responsibility (223) Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Command (233) Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Interpreter (243) Given a language, define a represention for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Iterator (257) Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Mediator (273) Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.



Memento (283) Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Observer (293) Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

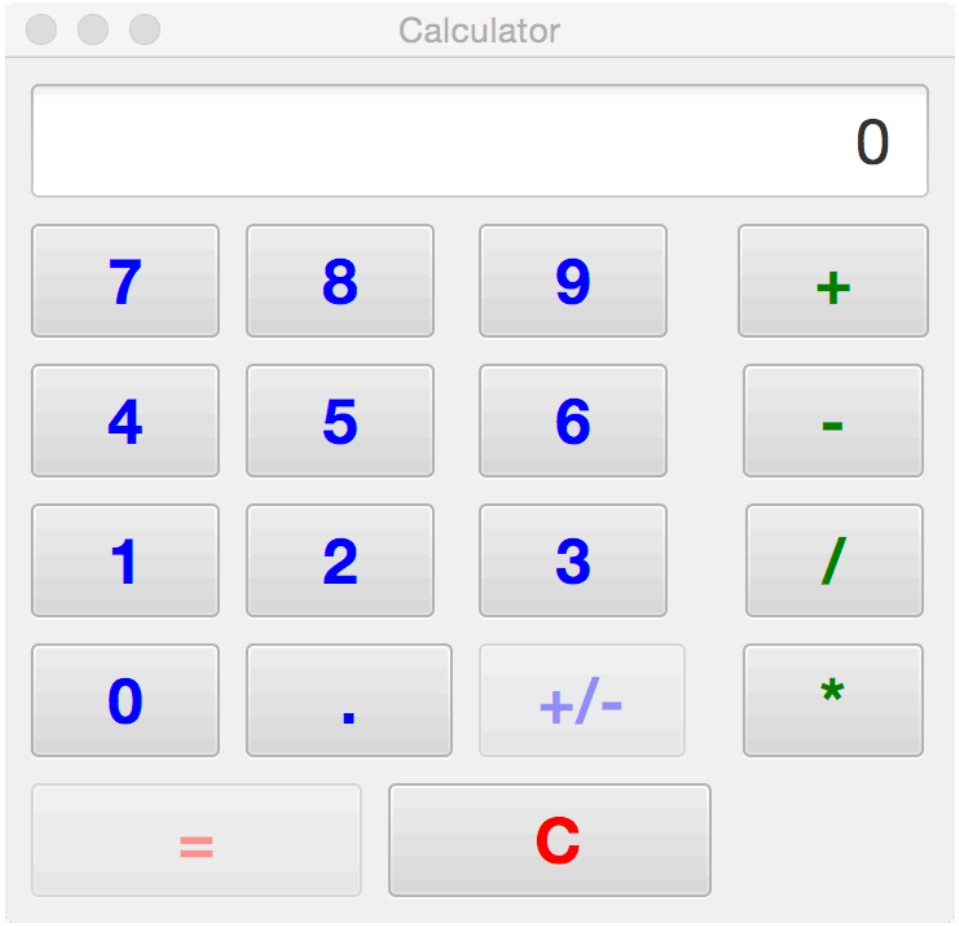
State (305) Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Strategy (315) Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

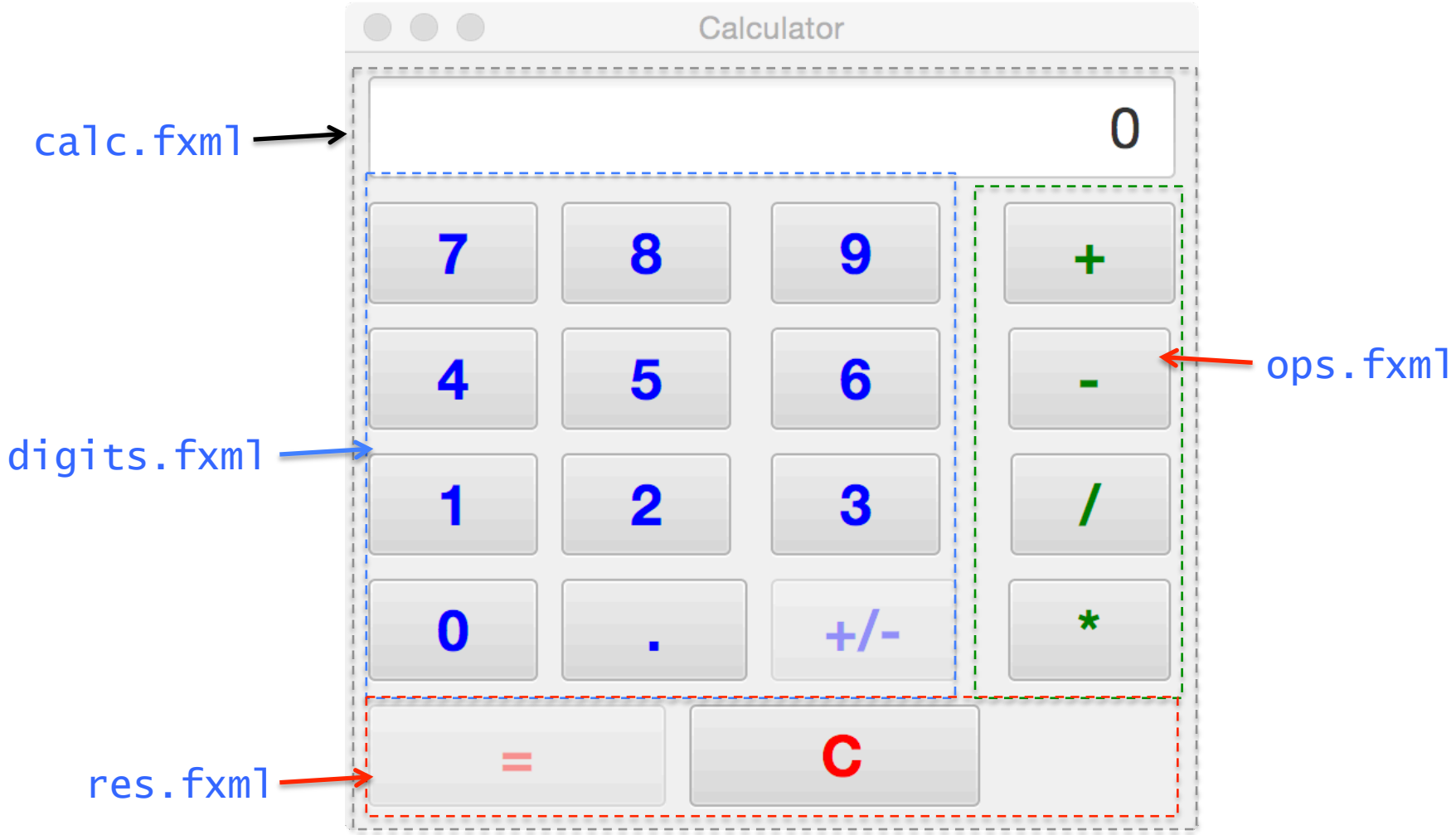
Template Method (325) Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Visitor (331) Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Building a Calculator - UI



Calculator UI – Nested UI Entities



Calculator UI – Nested UI Entities/FXMLs

calc.fxml

```
...
<?import java.net.*?>

<GridPane
    ...
    fx:controller="view.CalcController">
    ...
    <TextField fx:id="display" editable="false" alignment="CENTER_RIGHT"
               GridPane.columnSpan="2" />

    <fx:include fx:id="digits" source="digits.fxml" GridPane.rowIndex="1" />
    <fx:include fx:id="ops" source="ops.fxml" GridPane.columnIndex="1"
                GridPane.rowIndex="1" />
    <fx:include fx:id="res" source="res.fxml" GridPane.rowIndex="2"/>

    <stylesheets>
        <URL value="@calc.css" />
    </stylesheets>

</GridPane>
```

Need this for URL tag used with CSS styling (see bottom)

You can nest UI entities with their own FXML layouts, with fx:include

```
.button {
  -fx-font-size: 18pt;
  -fx-font-weight: bold;
}
.text-field {
  -fx-font-size: 18pt;
}
```

Matching Nested Entities with Controllers

calc.fxml

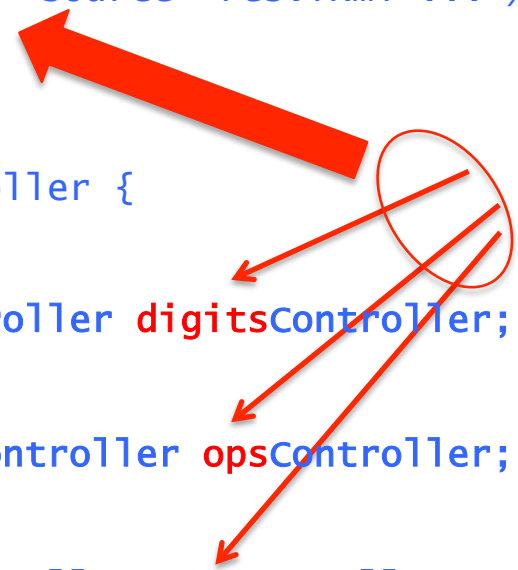
```
...
fx:controller="view.CalcController">
...
<fx:include fx:id="digits" source="digits.fxml" ... />
<fx:include fx:id="ops" source="ops.fxml" ... />
<fx:include fx:id="res" source="res.fxml"... />
```

CalcController.java

```
public class CalcController {
   ...
   @FXML
   protected DigitController digitsController;

   @FXML
   protected OperatorController opsController;

   @FXML
   protected ResultController resController;
   ...
}
```



IMPORTANT!!

The names of the controllers for the contained UI FXMLs must match the ids that go with fx:include in the container's FXML

fx:define and fx:reference

digits.fxml

```
<GridPane
  ...
  fx:controller="view.DigitController">
   ...
   <Button fx:id="d7" onAction="#digitPressed" text="  7  " />
   <Button fx:id="d8" onAction="#digitPressed" text="  8  " ... />
   <Button fx:id="d9" onAction="#digitPressed" text="  9  " ... />
   <Button fx:id="d4" onAction="#digitPressed" text="  4  " ... />
   <Button fx:id="d5" onAction="#digitPressed" text="  5  " ... />
   <Button fx:id="d6" onAction="#digitPressed" text="  6  " ... />
   <Button fx:id="d1" onAction="#digitPressed" text="  1  " ... />
   <Button fx:id="d2" onAction="#digitPressed" text="  2  " ... />
   <Button fx:id="d3" onAction="#digitPressed" text="  3  " ... />
   <Button fx:id="d0" onAction="#digitPressed" text="  0  " ... />
   ...
   <fx:define>
     <ArrayList fx:id="digitButtons">
      <fx:reference source="d0"/>
      <fx:reference source="d1"/>
      ...
      <fx:reference source="d8"/>
      <fx:reference source="d9"/>
     </ArrayList>
   </fx:define>
   ...
</GridPane>
```

needs <?import java.util.*?>

source values are the ids declared elsewhere in the FXML

This set up does away with the tedium of defining one @FXML field per button in the controller class

Matching ArrayList of fx:define to Java Code

digits.fxml

```
...
<fx:define>
   <ArrayList fx:id="digitButtons">
      <fx:reference source="d0"/>
      <fx:reference source="d1"/>
      ...
      <fx:reference source="d9"/>
   </ArrayList>
</fx:define>
...
```

```
public class DigitController {
   ...
   @FXML
   protected List<Button> digitButtons;
   ...
}
```

Container's styling is inherited by contained UIs

calc.css

```
.button {
   -fx-font-size: 18pt;
   -fx-font-weight: bold;
}
.text-field {
   -fx-font-size: 18pt;
}
```

digits.css

```
.button {
   -fx-text-fill: blue;
}
```

digits will be boldfaced and have font size of 18 pt, as defined in calc.css, plus color blue as defined in digits.css

Building a Calculator - The State Design Pattern

State Design Pattern

Allow an object to alter its behavior when the internal state changes.
The object will appear to change its class.

Calculator: State Diagram