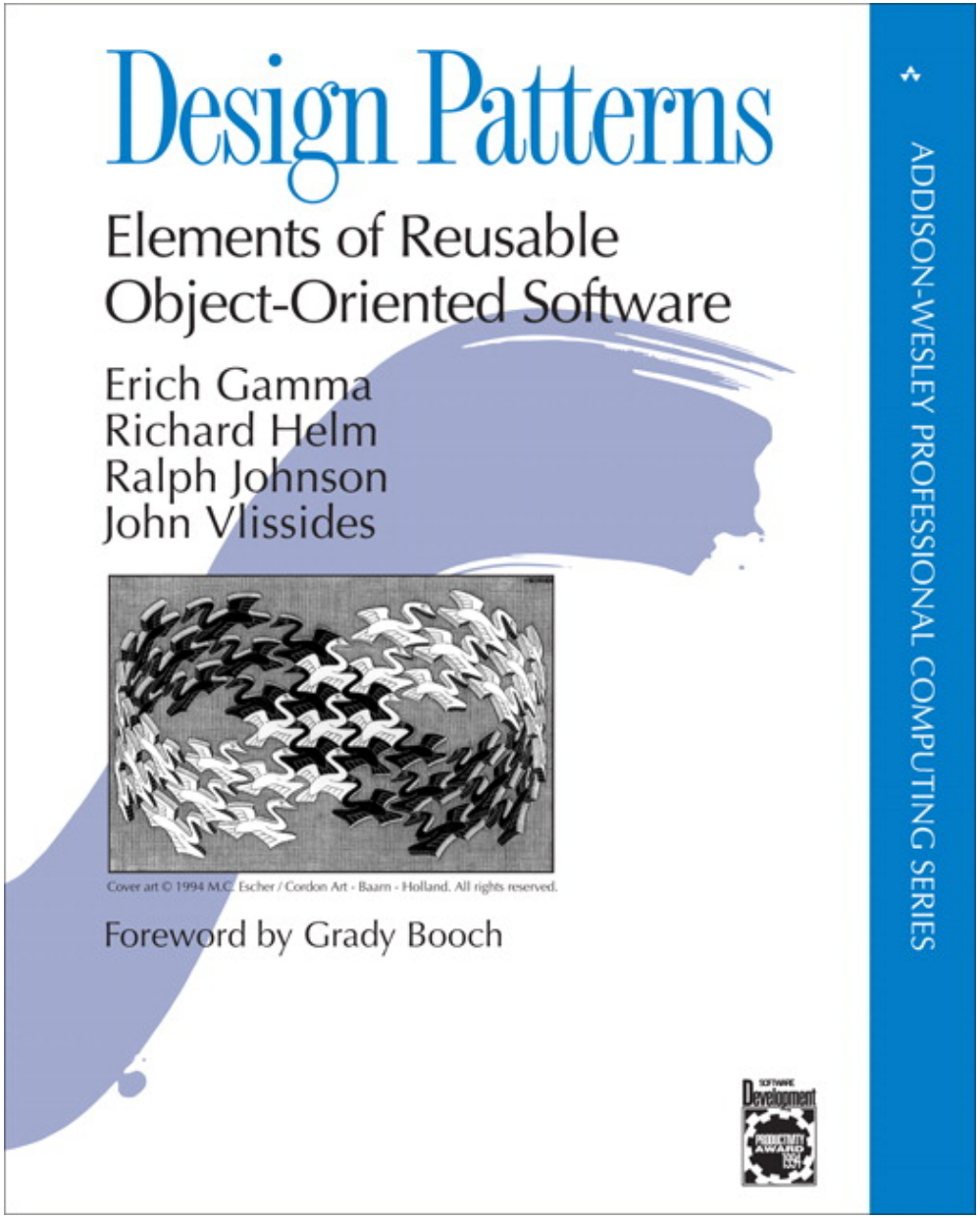
CS 213 – Spring 2016

Lecture 16 – Mar 10

Design Patterns – 1

State and Singleton Patterns

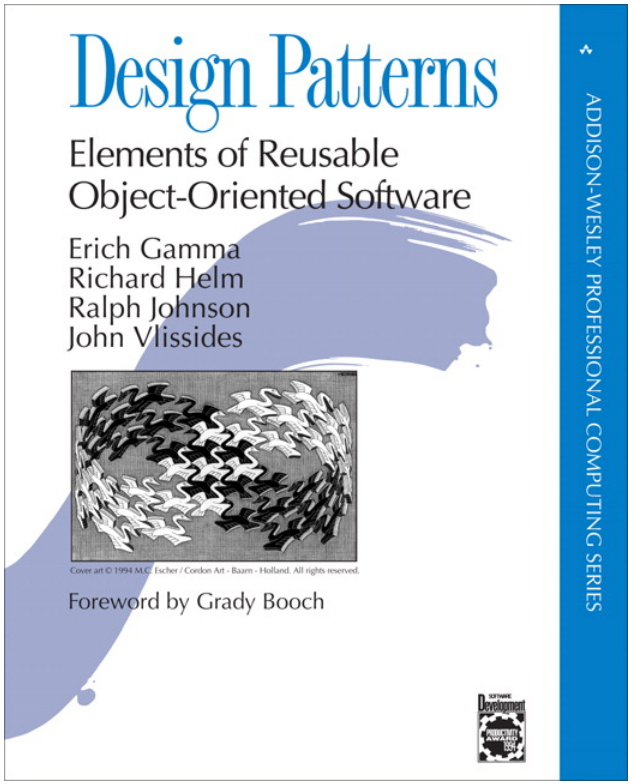
Illustration: State-Based Calculator


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.

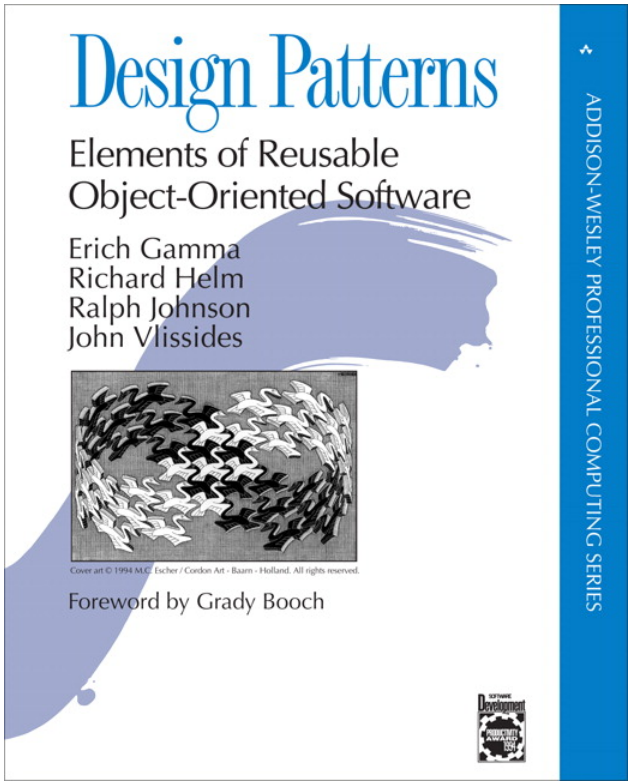
Builder (97) Separate the construction of a complex object from its representation so that the same construction process can create different representations.

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.

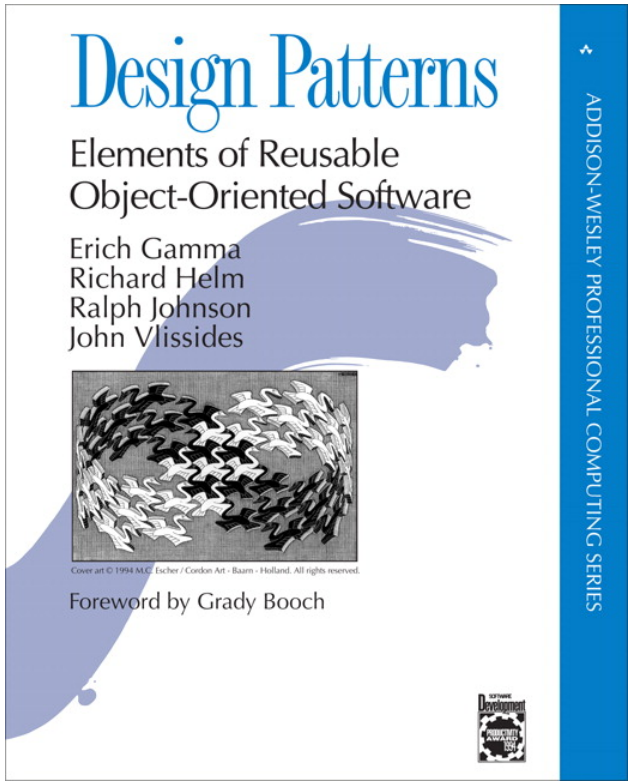
Composite (163) Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

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

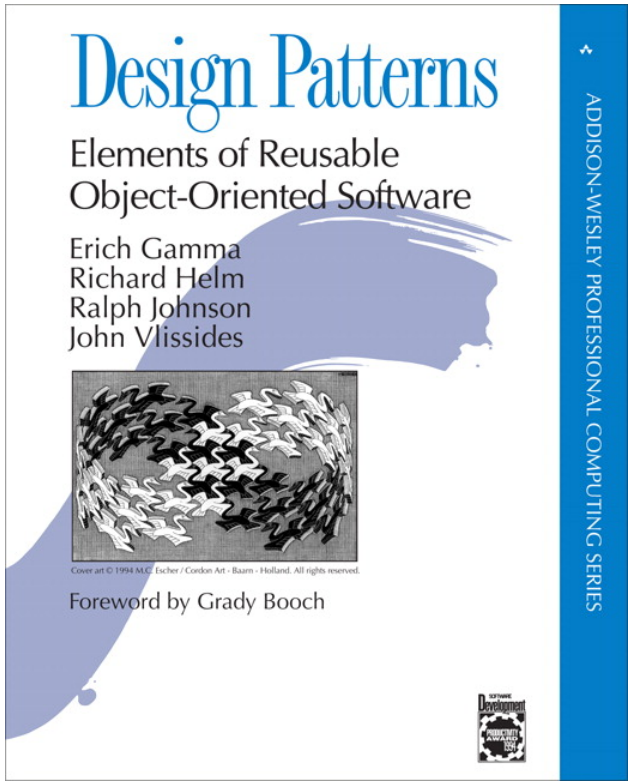
Chain of Responsibility (223) Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

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.

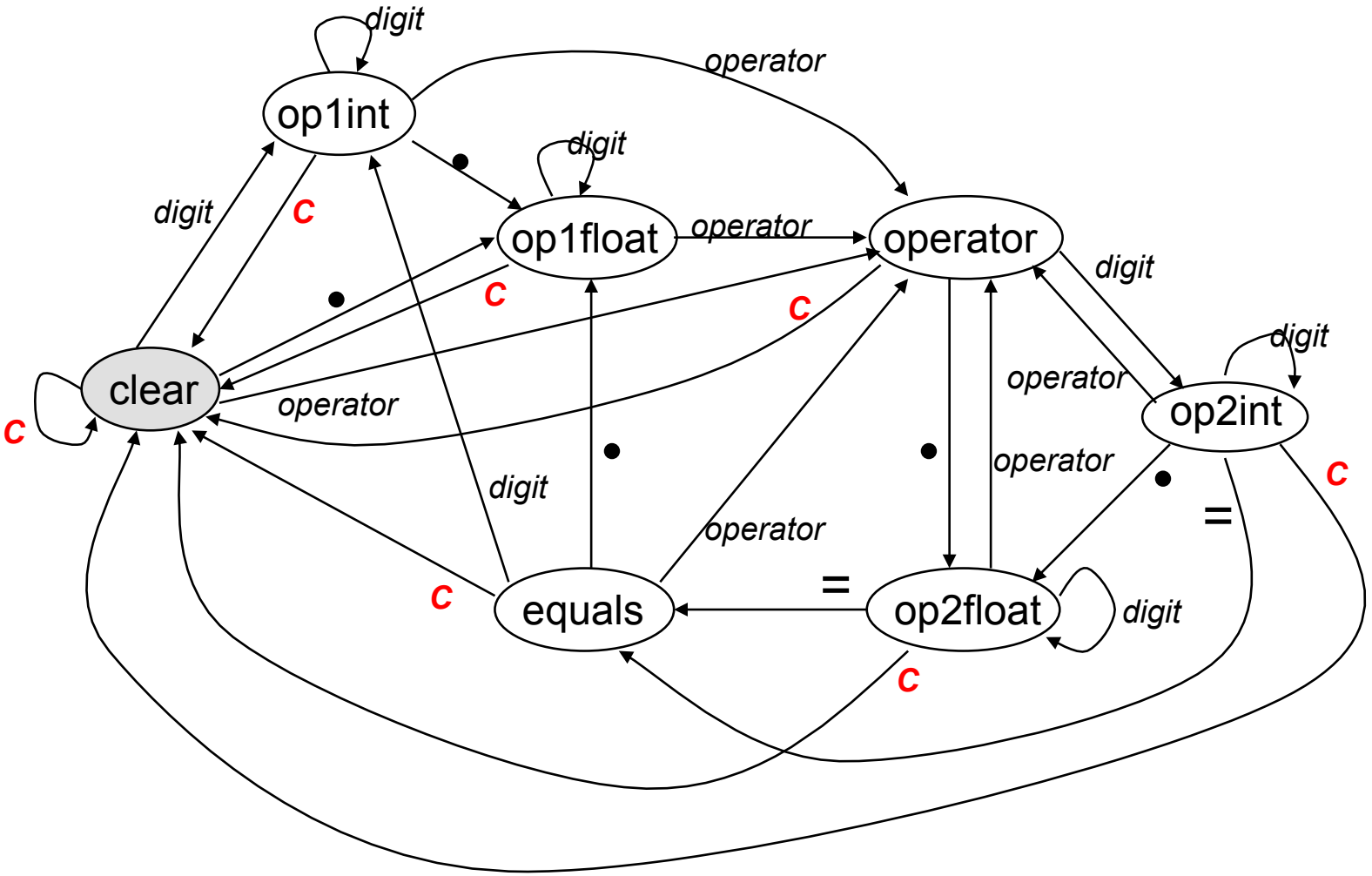
State (305) Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

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.

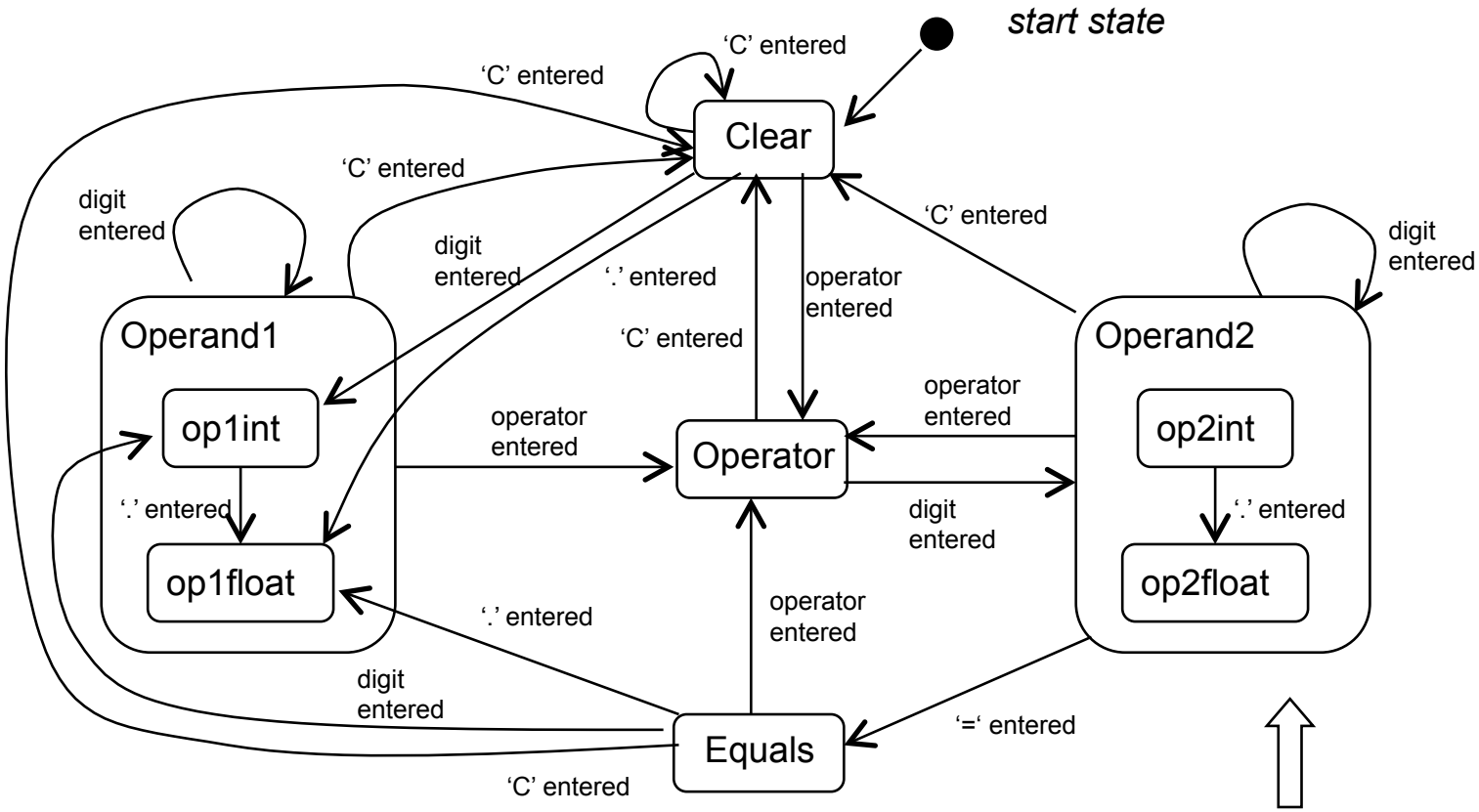
Template Method (325) Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

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.

Calculator: State Diagram



Calculator: UML State(chart) Diagram

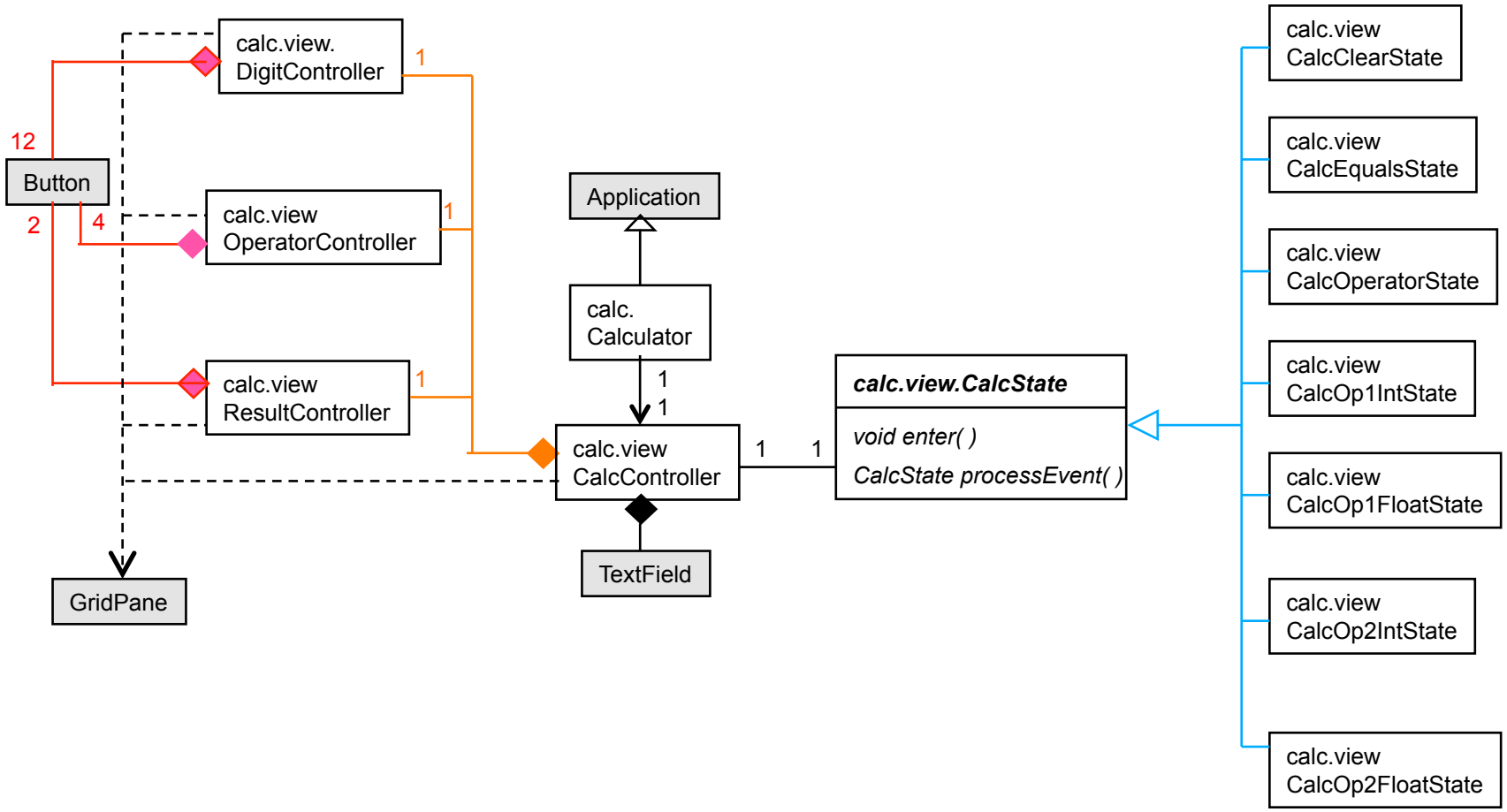


Every state is responsible for enabling and disabling buttons as appropriate

end state

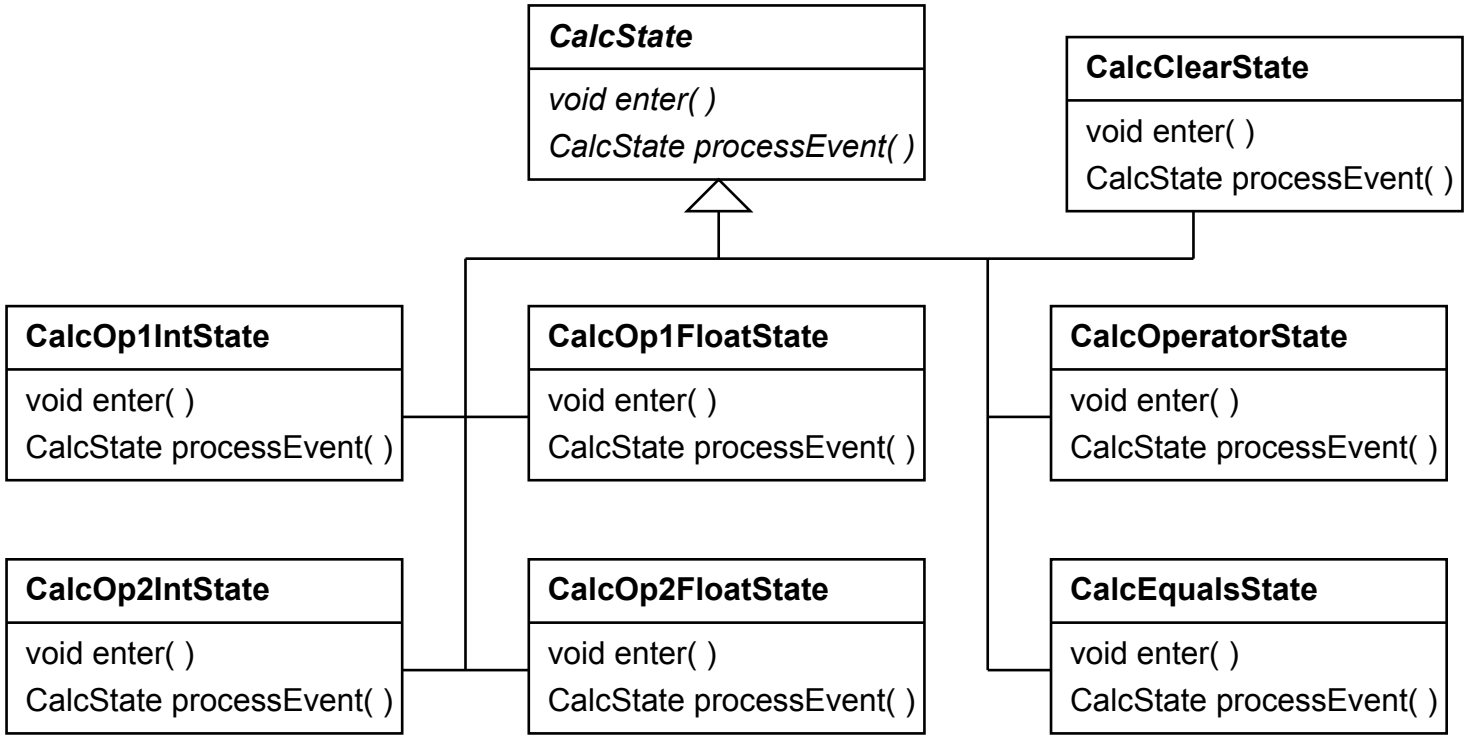
(all states above can go to end state – transitions not shown because all transitions are same, and happen on exiting the application)

State-based Calculator – UML Class Diagram



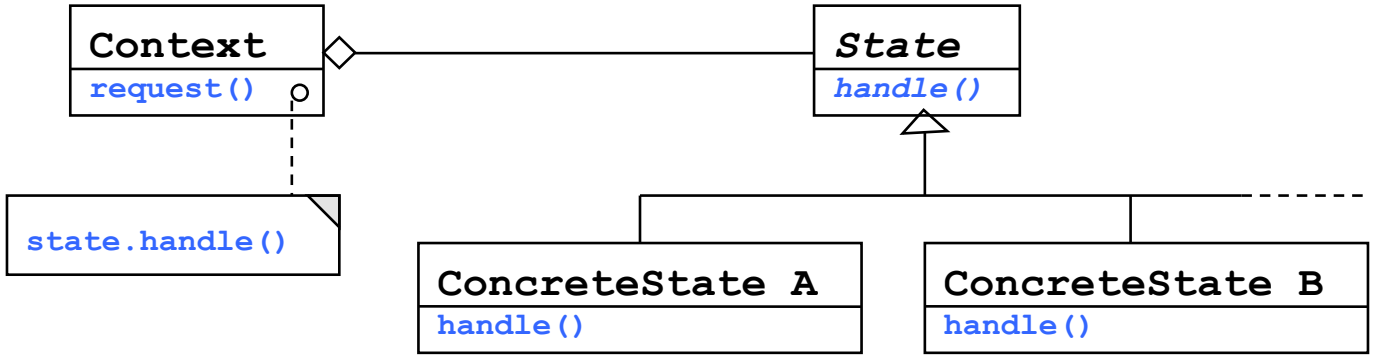
State Design Pattern: Applied to Calculator

- The general implementation of State pattern:
  - There is an abstract class that specifies state methods: in general these could be entry/body/exit methods
  - Subclasses of this abstract class define different specific states



State Design Pattern: Behavioral

- Allow an object to change its behavior when its internal state changes – the "object" is a subclass of an abstract class, thus polymorphism



- Context (client code) has a state object that is one of the concrete instances: the request method executes `handle` on this concrete instance dynamically binding the appropriate concrete class method – neat use of polymorphism

- Example: State classes `CalcState` (abstract) and `CalcClearState`, `CalcEqualsState`, etc. (concrete) in the state-based calculator application. The context is the `CalcController` class.

Singleton Design Pattern: Applied to Calculator

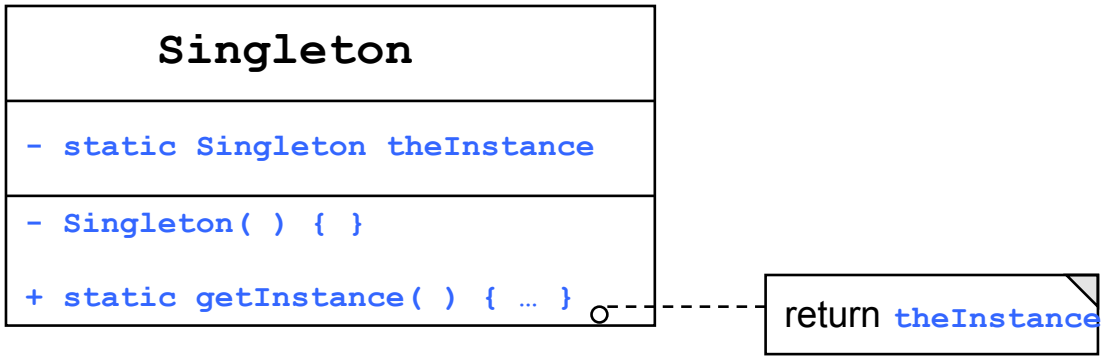
- Each of the concrete state classes implements the Singleton pattern. For instance, the `CalcClearState` class:

```
class CalcClearState {

   . . .
   private static CalcClearState instance = null;
   . . .
   private CalcClearState(CalcController calcController) {
      super(calcController);
   }
   . . .
   public static CalcClearState getInstance(CalcController calcController) {
      if (instance == null) {
         instance = new CalcClearState(calcController);
      }
      return instance;
   }
   ...
}
```

Singleton: Creational

- Ensure that a class has only one object (instance) and provide a global point of access to this single instance



- The single private constructor ensures that an instance of Singleton cannot be created using `new`