Mediator, Memento

CSCI 4448/5448: Object-Oriented Analysis & Design

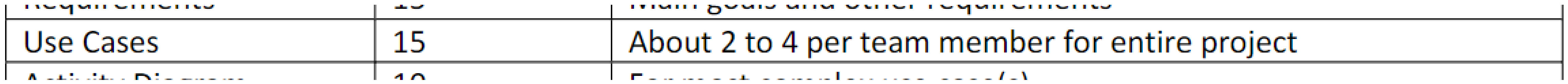
Lecture 30

Acknowledgement & Materials Copyright

- I’d like to start by acknowledging Dr. Ken Anderson
- Ken is a Professor and the Chair of the Department of Computer Science
- Ken taught OOAD on several occasions, and has graciously allowed me to use his copyrighted material for this instance of the class
- Although I will modify the materials to update and personalize this class, the original materials this class is based on are all copyrighted © Kenneth M. Anderson; the materials are used with his consent; and this use in no way challenges his copyright

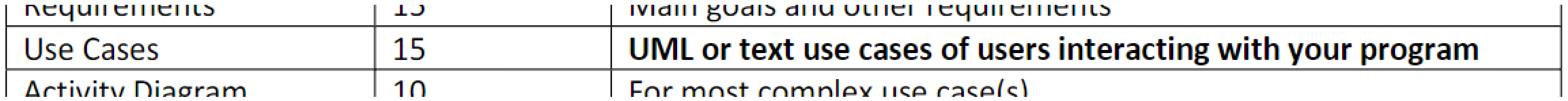
Before We Start – Project 4 correction

- Thanks to an eagle-eyed student team, I've updated the Project 4 instructions
- I made an error in describing the use case element of the project in the points for the project notes (it's correct in the assignment details)
- I said:



Requirements	15	Main goals and other requirements
Use Cases	15	About 2 to 4 per team member for entire project
Activity Diagram	10	For most complex use case(s)

- This was referring back to the scoping exercise for your project, that's not what the use cases in Project 4 are about…
- I changed it to:



Requirements	15	Main goals and other requirements
Use Cases	15	**UML or text use cases of users interacting with your program**
Activity Diagram	10	For most complex use case(s)

- Which is what the use cases in Project 4 are intended to be, a good thorough description of the scenarios of users interacting with your code, what tasks will they be doing. Ok?

Before We Start – The Rest of the Semester

- Topics/Tasks planned
  - Close out last OO Patterns
  - Pattern Coding Exercise
  - More on Design Techniques
  - ORMs
  - Refactoring
  - Dependency Injection
  - Reflection
  - Antipatterns
  - (New) OO for APIs and GUIs
  - Alternatives to OO Design
  - Graduate Pecha Kuchas
  - Graduate Project Submissions
  - Project 5 and 6 Demonstrations
  - Final (Optional)
  - Any topics you'd like me to cover?

Before We Start – Dealing with Distance

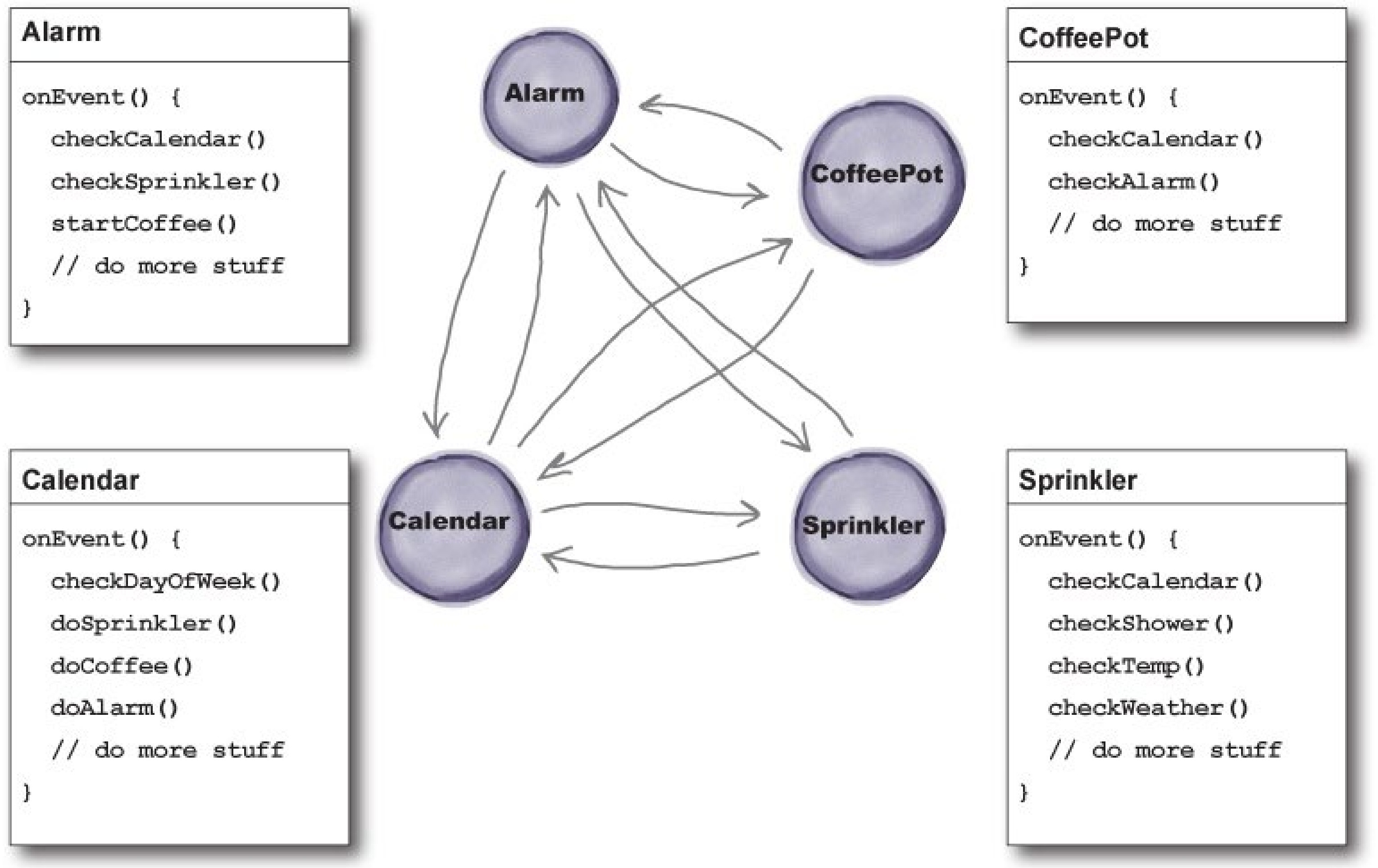
- Per the administration directives, you should avoid in-person meetings whenever possible
- For your group projects
  - You should be able to use Zoom: https://cuboulder.zoom.us/
  - Other approaches: Google hangouts/docs, Slack IM, etc.
  - Avoid in-person meetings where possible
- Demonstrations for project 5 & 6 can be either Zoom-based or recorded videos to support social distancing – more on this as they get closer
- If you're leaving campus, you should still be able to finish out the course – keep in mind you still will have team activities and you must plan for them!
- I (and the class staff) will be watching Piazza and e-mail, don't be afraid to ask for a meeting outside of class and office hours
- Kindness, patience, open communication…
- Questions?

Head First Design Patterns

- Chapter 13 – Leftover Patterns
  - Bridge
  - Builder
  - Flyweight
  - Interpreter
  - Chain of Responsibility
  - Mediator
  - Memento
  - Prototype
  - Visitor

Mediator

- The Mediator Pattern is used to centralize complex communications and control between related objects.
- Head First example:
  Elements of an automated home



Alarm
```
onEvent() {
  checkCalendar()
  checkSprinkler()
  startCoffee()
  // do more stuff
}
```

CoffeePot
```
onEvent() {
  checkCalendar()
  checkAlarm()
  // do more stuff
}
```

Calendar
```
onEvent() {
  checkDayOfWeek()
  doSprinkler()
  doCoffee()
  doAlarm()
  // do more stuff
}
```

Sprinkler
```
onEvent() {
  checkCalendar()
  checkShower()
  checkTemp()
  checkWeather()
  // do more stuff
}
```

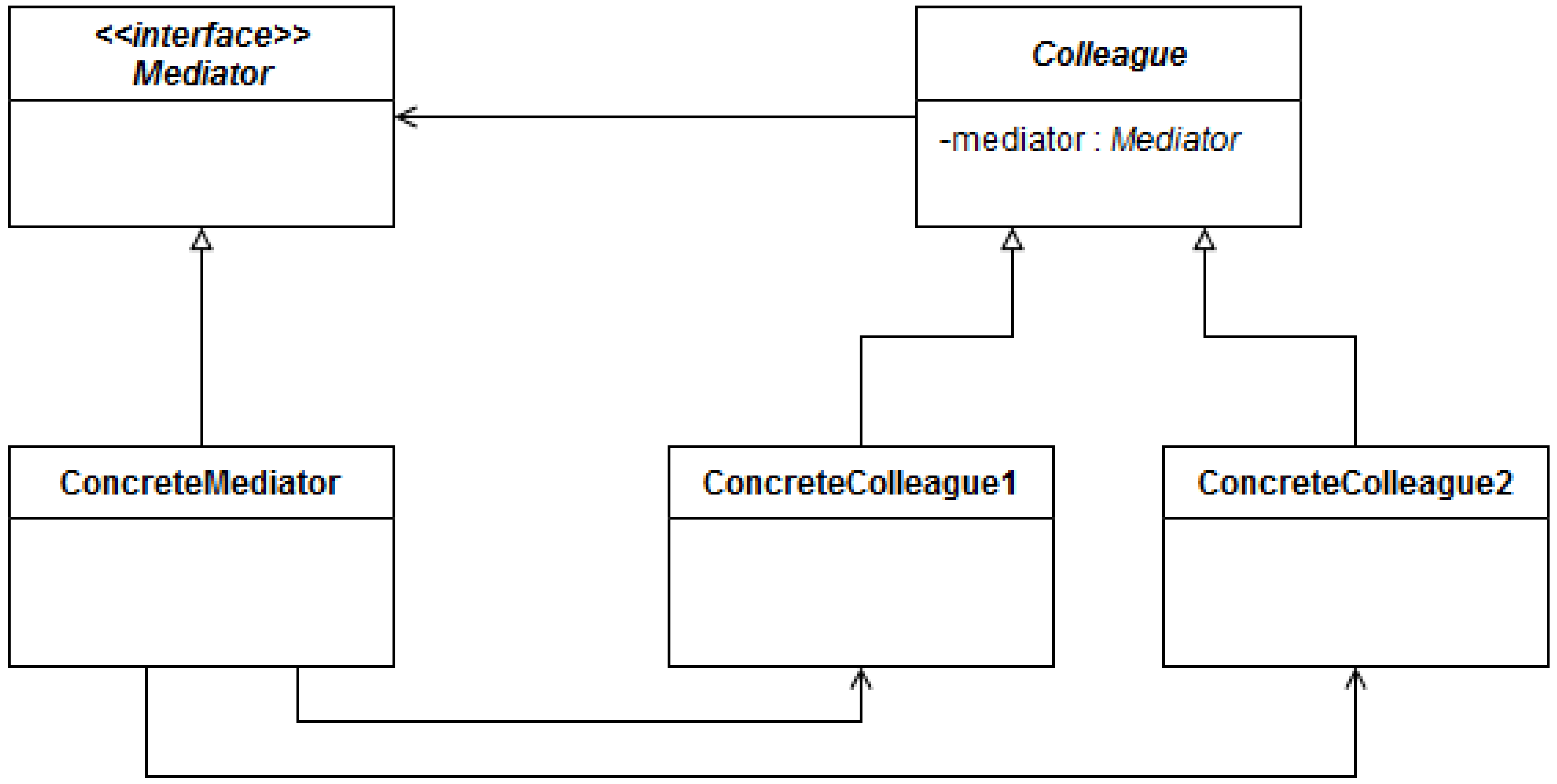
Mediator Problem/Solution

- Problem: We want to design reusable component classes, but dependencies between the potentially reusable pieces are highly coupled
  - Demonstrates the "spaghetti code" phenomenon
- Solution: 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
  - Design an intermediary to decouple many peers

- https://sourcemaking.com/design_patterns/mediator

Mediator Pattern: UML Class Diagram

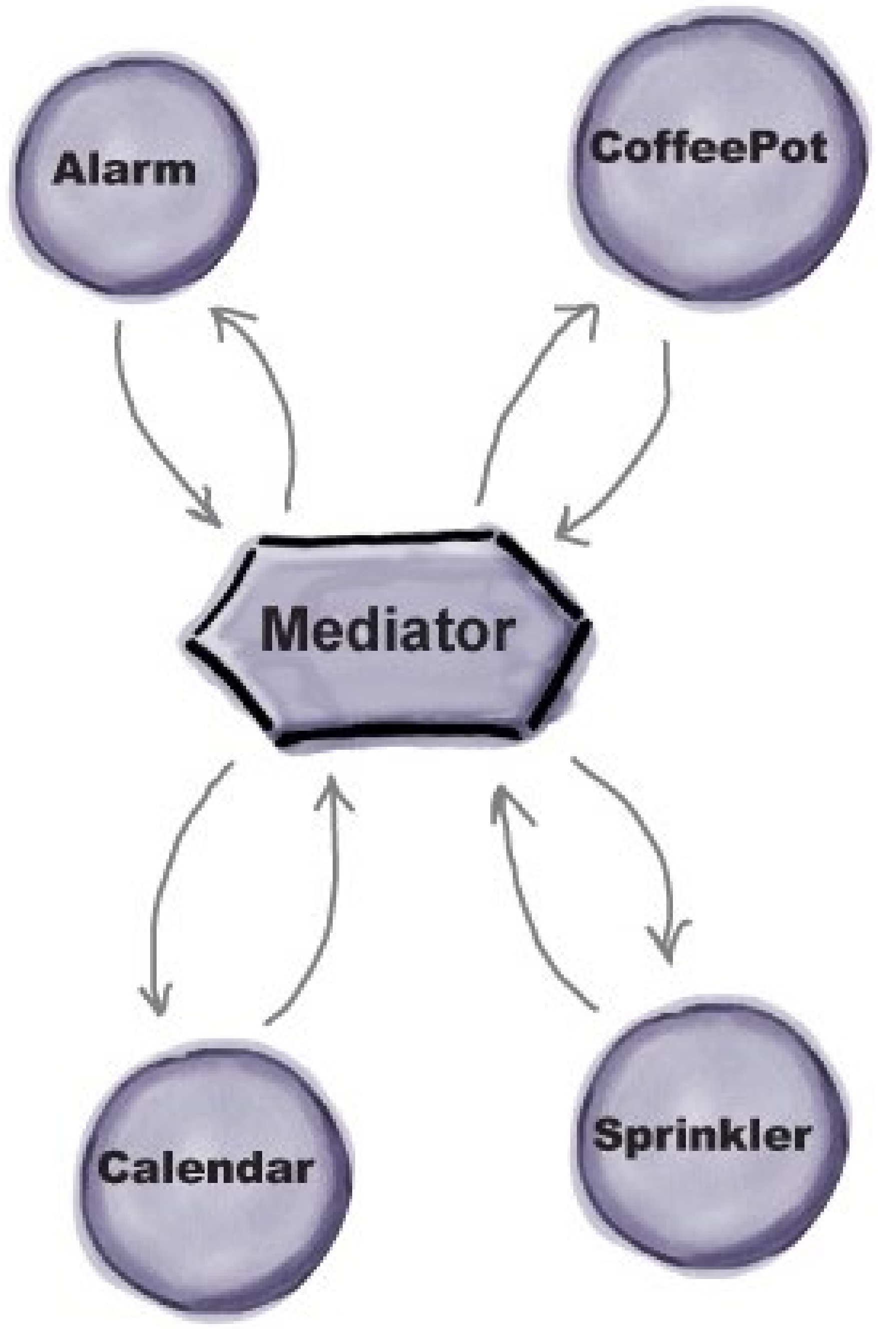
- Colleagues (or peers) are not coupled to one another
- Each talks to the Mediator, which in turn knows and conducts the orchestration of the others
- The "many to many" mapping between colleagues that would otherwise exist, has been represented in the mediator.
- The diagram shows directional references from the concrete mediator, but these could be 2-way

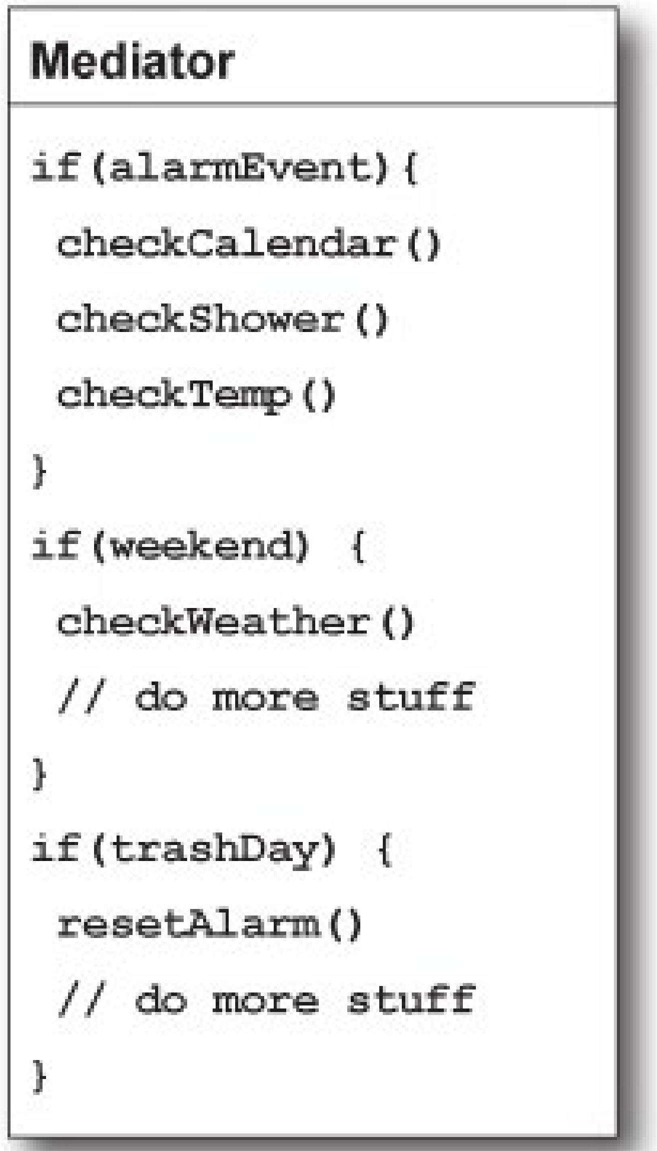


https://www.baeldung.com/java-mediator-pattern

Mediator applied

- With a Mediator added to the system, all of the appliance objects can be greatly simplified:
  - They tell the Mediator when their state changes
  - They respond to requests from the Mediator
- Before we added the Mediator, all of the appliance objects needed to know about each other... they were all tightly coupled
- With the Mediator in place, the appliance objects are all completely decoupled from each other.
- The Mediator contains all of the control logic for the entire system. When an existing appliance needs a new rule, or a new appliance is added to the system, you'll know that all of the necessary logic will be added to the Mediator
  - Hmm…





Mediator

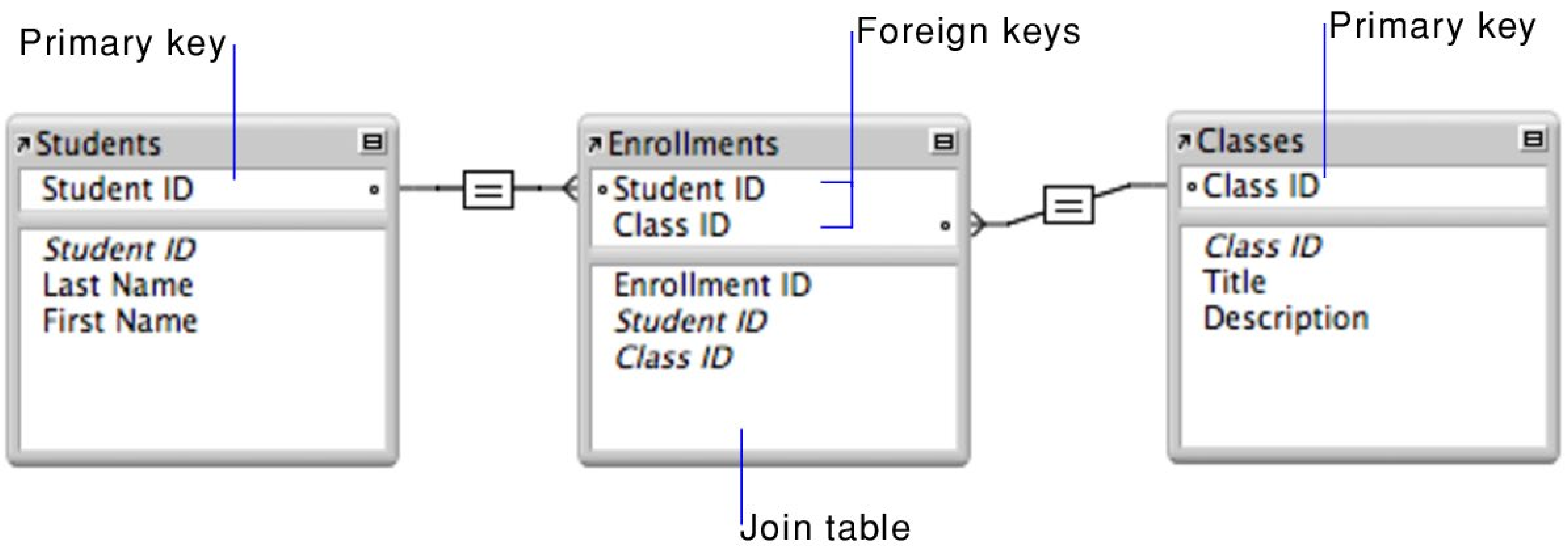
```
if(alarmEvent){
 checkCalendar()
 checkShower()
 checkTemp()
}
if(weekend) {
 checkWeather()
 // do more stuff
}
if(trashDay) {
 resetAlarm()
 // do more stuff
}
```

Mediator Considered

- Be careful not to create a "controller" or "god" object
- Balance the principle of decoupling with the principle of distributing responsibility evenly
- Realize there is a dependency when new objects are added to visit the Mediator to adjust system behavior
- Without proper design, the Mediator object itself can become overly complex

Mediator Echoes: Relational or Join Tables

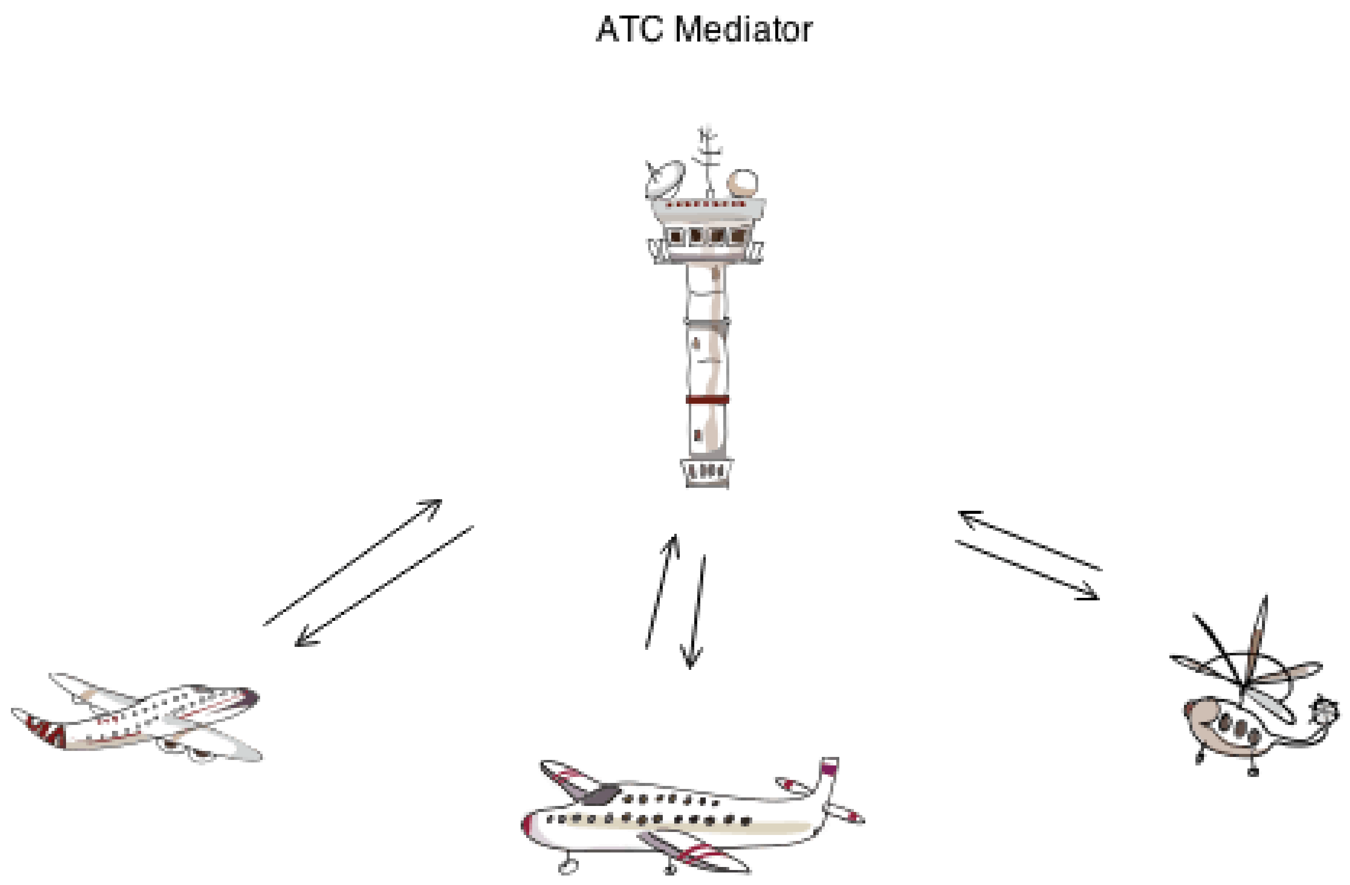
- It's common when you're putting a relational table together quickly to combine multiple elements in a single table or to make connections with duplications of key information between two tables
- During normalization, you may find it makes sense to separate elements to tables containing only data pertaining to them, and then use a join table to relate information to each other…
- To me, this is an echo of mediator, using a central element to increase cohesiveness in the collogues and centralize relationship logic



https://fmhelp.filemaker.com/help/18/fmp/en/index.html#page/FMP_Help/many-to-many-relationships.html

Another Mediator Eample

- Loose coupling between colleague objects is achieved by having colleagues communicate with the Mediator, rather than with each other
- The control tower at a controlled airport demonstrates Mediator very well
- The pilots of the planes approaching or departing the terminal area communicate with the tower rather than explicitly communicating with one another
- The constraints on who can take off or land are enforced by the tower
- It is important to note that the tower does not control the whole flight; it exists only to enforce constraints in the terminal area
- https://sourcemaking.com/design_patterns/mediator

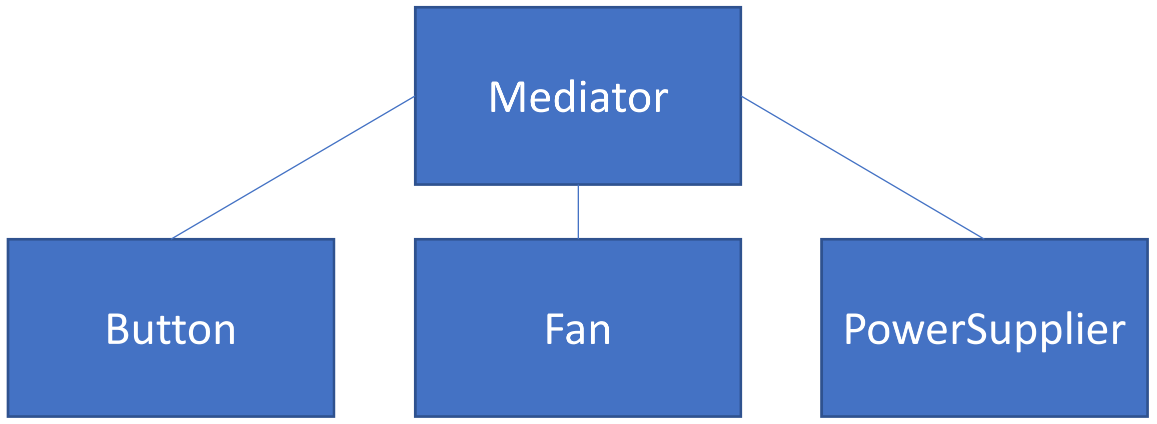


A Java Mediator

```
// two colleagues, a button and a fan
// another, powerSupplier, not shown
public class Button {
   private Mediator mediator;
   // constructor, getters and setters
   public void press() {
      mediator.press();
   }
}
public class Fan {
   private Mediator mediator;
   private boolean isOn = false;
   // constructor, getters and setters
   public void turnOn() {
      mediator.start();
      isOn = true;
   }
   public void turnOff() {
      isOn = false;
      mediator.stop();
   }
}
```

https://www.baeldung.com/java-mediator-pattern

```
// their mediator
public class Mediator {
   private Button button;
   private Fan fan;
   private PowerSupplier powerSupplier;
   // constructor, getters and setters
   public void press() {
      if (fan.isOn()) {
         fan.turnOff();
      } else {
         fan.turnOn();
      }
   }
   public void start() {
      powerSupplier.turnOn();
   }
   public void stop() {
      powerSupplier.turnOff();
   }
}
```



Mediator and other Patterns

- Mediator is similar to Façade in that it abstracts functionality of existing classes
  - Mediator abstracts/centralizes arbitrary communication between colleague objects, it routinely "adds value", and it is known/referenced by the colleague objects (i.e. it defines a multidirectional protocol)
  - In contrast, Facade defines a simpler interface to a subsystem, it doesn't add new functionality, and it is not known by the subsystem classes
- Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs
  - Chain of Responsibility passes a sender request along a chain of potential receivers
  - Command normally specifies a sender-receiver connection with a subclass
  - Mediator has senders and receivers reference each other indirectly
  - Observer defines a very decoupled interface that allows for multiple receivers to be configured at run-time

- https://sourcemaking.com/design_patterns/mediator

Mediator vs. Observer

- Mediator and Observer are competing patterns
  - The difference between them is that Observer distributes communication by introducing "observer" and "subject" objects, whereas a Mediator object encapsulates the communication between other objects
  - **It's easier to make reusable Observers and Subjects than to make reusable Mediators**
- Mediator could leverage Observer for dynamically registering colleagues and communicating with them

- https://sourcemaking.com/design_patterns/mediator

Mediator – Key Points

- The Mediator Pattern is a good choice if we have to deal with a set of objects that are tightly coupled and hard to maintain
- Increases the reusability of the objects supported by the Mediator by decoupling them from the system
- Simplifies maintenance of the system by centralizing control logic
- Simplifies and reduces the variety of messages sent between objects in the system
- The Mediator is commonly used to coordinate related GUI components
- BUT we may have too many tightly coupled objects due to a faulty design of the system
  - If this is the case, we should not apply the Mediator Pattern
  - Rather, take one step back and rethink the way we've modeled classes

Memento

- Memento Pattern is for when you need to be able to return an object to one of its previous states
  - for instance, if your user requests an “undo”
- Head First example: Maintaining state in a game…
  - Your interactive role playing game is hugely successful, and has created a legion of addicts, all trying to get to the fabled “level 13”
  - As users progress to more challenging game levels, the odds of encountering a game-ending situation increase
  - Users ask for a “save progress” command, so that players can store their game progress and at least recover most of their efforts when their character is unfairly extinguished
  - The “save progress” function needs to be designed to return a resurrected player to the last level they completed successfully

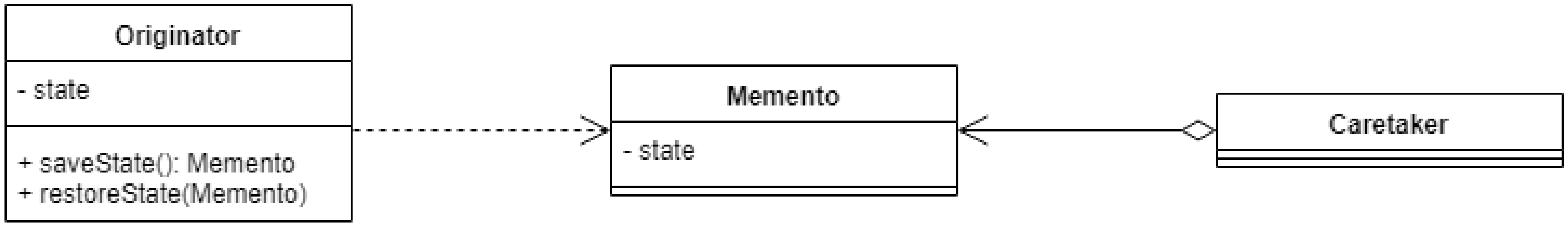
Memento Problem & Solution

- Problem: Need to restore an object back to its previous state (e.g. "undo" or "rollback" operations)
- Solution:
  - Without violating encapsulation, capture and externalize an object's internal state so that the object can be returned to this state later
  - A magic cookie that encapsulates a "check point" capability
  - Promote undo or rollback to full object status

- https://sourcemaking.com/design_patterns/memento

Memento Pattern: UML Class Diagram

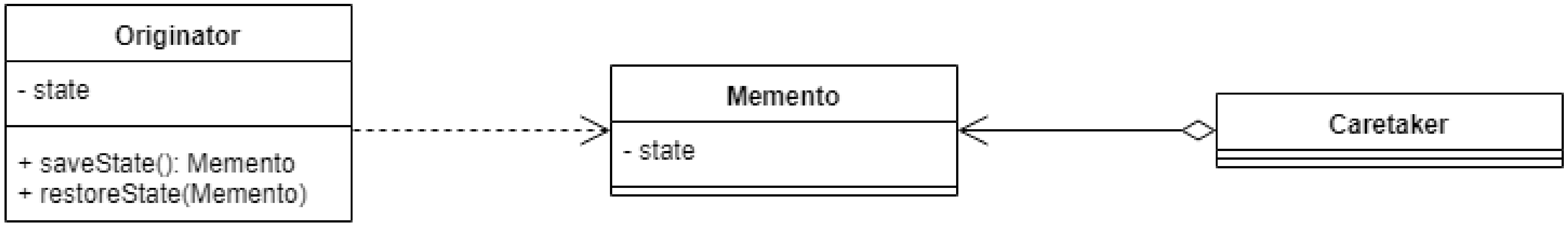
- The object whose state needs to be saved is called an Originator
- The Caretaker is the object triggering the save and restore of the state, which is called the Memento
- The Memento object should expose as little information as possible to the Caretaker
  - This is to ensure that we don't expose the internal state of the Originator to the outside world, as it would break encapsulation
  - However, the Originator should access enough information in order to restore to the original state



- https://www.baeldung.com/java-memento-design-pattern

Memento Considered

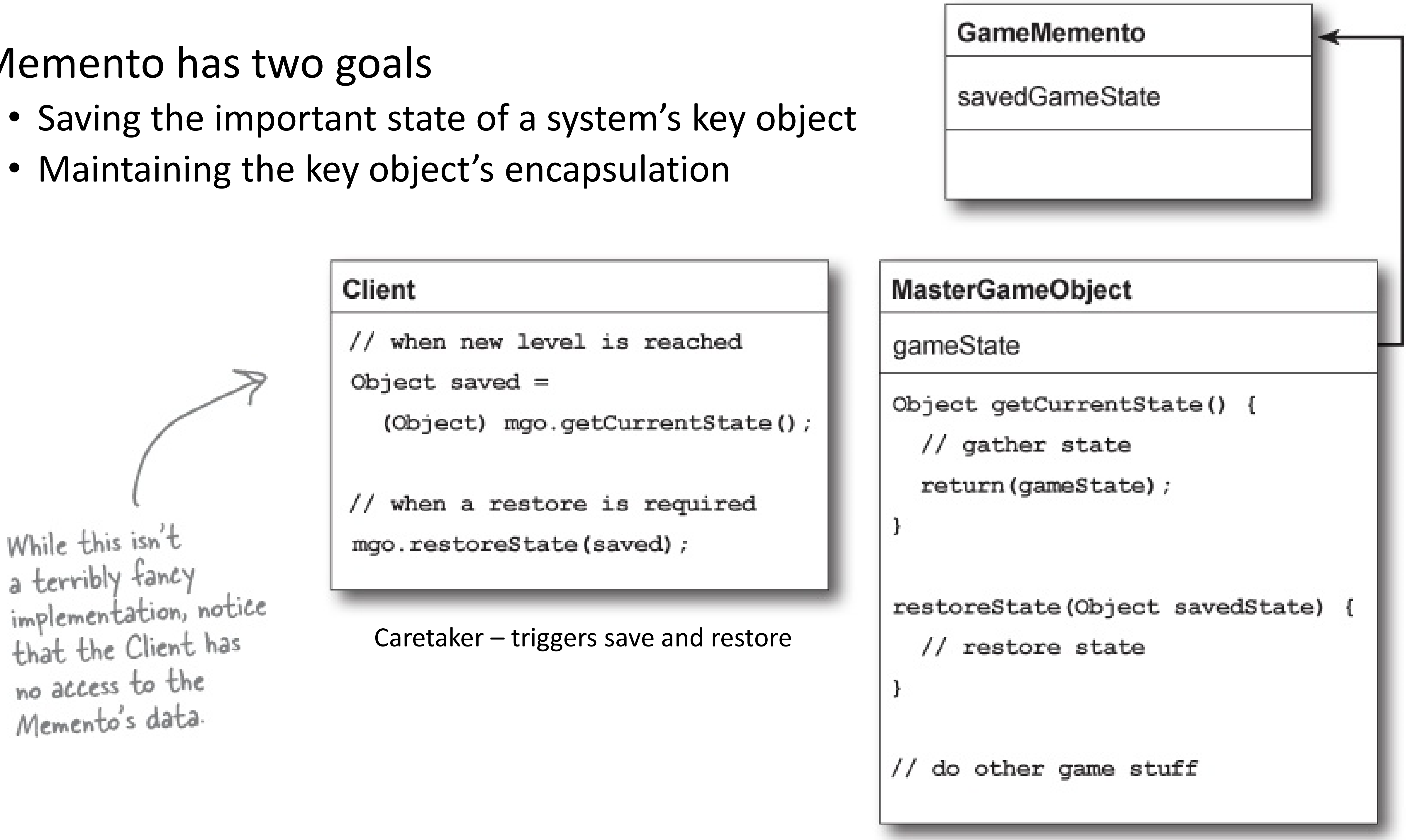
- The Originator can produce and consume a Memento
- The Caretaker only keeps the state before restoring it
- The internal representation of the Originator is kept hidden from the external world.
    - Here, we used a single field to represent the state of the Originator, though we're not limited to one field and could have used as many fields as necessary
    - The state held in the Memento object doesn't have to match the full state of the Originator as long as the kept information is sufficient to restore the state of the Originator



- https://www.baeldung.com/java-memento-design-pattern

Memento applied

- Memento has two goals
  - Saving the important state of a system’s key object
  - Maintaining the key object’s encapsulation



```
// when new level is reached
Object saved =
   (Object) mgo.getCurrentState();

// when a restore is required
mgo.restoreState(saved);
```

```
Object getCurrentState() {
   // gather state
   return(gameState);
}

restoreState(Object savedState) {
   // restore state
}

// do other game stuff
```

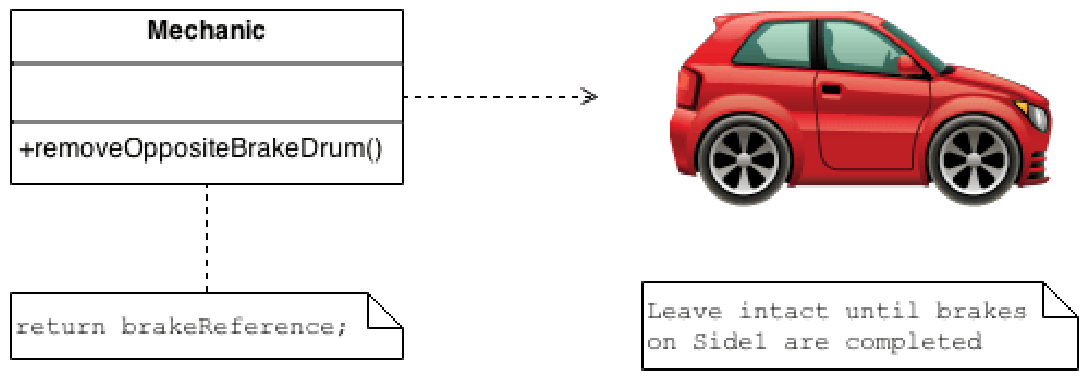
While this isn’t a terribly fancy implementation, notice that the Client has no access to the Memento’s data.

Caretaker – triggers save and restore

Originator – can save or restore

Another Memento example

- The Memento captures and externalizes an object's internal state so that the object can later be restored to that state
- This pattern is common among do-it-yourself mechanics repairing drum brakes on their cars
- The drums are removed from both sides, exposing both the right and left brakes
  - Only one side is disassembled and the other serves as a Memento of how the brake parts fit together
  - Only after the job has been completed on one side is the other side disassembled
  - When the second side is disassembled, the first side acts as the Memento.
- https://sourcemaking.com/design_patterns/memento


Mechanic
+removeOppositeBrakeDrum()
return brakeReference;
Leave intact until brakes on Side1 are completed

Memento in Java – a Text Editor

```
// a text editor…
public class TextEditor {
   private TextWindow textWindow;
   public TextEditor(TextWindow textWindow) {
      this.textWindow = textWindow;
   }
}
// text windows hold and add text…
public class TextWindow {
   private StringBuilder currentText;
   public TextWindow() {
      this.currentText = new StringBuilder();
   }
   public void addText(String text) {
      currentText.append(text);
   }
}
```

https://www.baeldung.com/java-memento-design-pattern

```
// To implement save and undo…
// add Memento
public class TextWindowState {
   private String text;
    public TextWindowState(String text) {
      this.text = text;
   }
   public String getText() {
      return text;
   }
}

//add Originator code to Text Window
public TextWindowState save() {
   return new TextWindowState(wholeText.toString());
}
public void restore(TextWindowState save) {
   currentText = new StringBuilder(save.getText());
}

// add this Caretaker code to Text Editor
private TextWindowState savedTextWindow;
public void hitSave() {
   savedTextWindow = textWindow.save();
}
public void hitUndo() {
   textWindow.restore(savedTextWindow);
}
```

Memento in Java – a Text Editor

```
TextEditor textEditor = new TextEditor(new TextWindow());
textEditor.write("The Memento Design Pattern\n");
textEditor.write("How to implement it in Java?\n");
textEditor.hitSave();

textEditor.write("Buy milk and eggs before coming home\n");

textEditor.hitUndo();

assertThat(textEditor.print()).isEqualTo("The Memento Design Pattern\nHow to
implement it in Java?\n");
```

https://www.baeldung.com/java-memento-design-pattern

Memento and other Patterns

- Command and Memento act as magic tokens to be passed around and invoked at a later time
  - In Command, the token represents a request
  - In Memento, it represents the internal state of an object at a particular time
- Polymorphism is important to Command, but not to Memento because its interface is so narrow that a memento can only be passed as a value
- Command can use Memento to maintain the state required for an undo operation
- State could use Memento to be able to return to earlier transitions
- Memento is often used in conjunction with Iterator
  - An Iterator can use a Memento to capture the state of an iteration
  - The Iterator stores the Memento internally
- https://sourcemaking.com/design_patterns/memento

Memento Key Points

- The Memento is used to save state
- Keeping the saved state external from the key object helps to maintain cohesion
- Keeps the key object’s data encapsulated
- Provides easy-to-implement recovery capability
- A drawback to using Memento is that saving and restoring state can be time consuming
- In Java (and other) systems, consider using Serialization to save a system’s state

Java Serialization and Deserialization

- Serialization is the conversion of the state of an object into a byte stream
- Deserialization does the opposite
- Stated differently, serialization is the conversion of a Java object into a static stream (sequence) of bytes which can then be saved to a database or transferred over a network
- We saw this as an issue with network proxy object connections and communication, which could only transfer primitive types or serialized objects

- https://www.baeldung.com/java-serialization

Java Serialization and Deserialization

- Classes that are eligible for serialization need to implement a special marker interface Serializable
- Example of a class Person

```
public class Person implements Serializable {
    private static final long serialVersionUID = 1L;
    static String country = "ITALY";
    private int age;
    private String name;
    transient int height;  // will not be saved in serialized object
     // getters and setters
}
```

- Static fields belong to a class (as opposed to an object) and are not serialized
- The keyword *transient* is used to ignore class fields during serialization
- The JVM associates a version (long) number with each serializable class;
- If a serializable class doesn't declare a serialVersionUID, the JVM will generate one automatically at run-time. However, it is highly recommended that each class declares its serialVersionUID as the generated one is compiler dependent
- https://www.baeldung.com/java-serialization

Java Use of Serialization

```
public void whenSerializingAndDeserializing_ThenObjectIsTheSame() ()
  throws IOException, ClassNotFoundException {
    Person person = new Person();
    person.setAge(20);
    person.setName("Joe");

    FileOutputStream fileOutputStream = new FileOutputStream("yourfile.txt");
    ObjectOutputStream objectOutputStream = new ObjectOutputStream(fileOutputStream);
    objectOutputStream.writeObject(person);  //serializing person here
    objectOutputStream.flush();
    objectOutputStream.close();

    FileInputStream fileInputStream = new FileInputStream("yourfile.txt");
    ObjectInputStream objectInputStream = new ObjectInputStream(fileInputStream);
    Person p2 = (Person) objectInputStream.readObject();   //deserializing person here, note the cast
    objectInputStream.close();

    assertTrue(p2.getAge() == p.getAge());
    assertTrue(p2.getName().equals(p.getName()));
}
```

https://www.baeldung.com/java-serialization

Python Serialization - Pickle

- The pickle module is used for implementing binary protocols for serializing and de-serializing a Python object structure
  - Pickling is the process where a Python object hierarchy is converted into a byte stream
  - Unpickling is the inverse of Pickling process where a byte stream is converted into an object hierarchy
- dumps() – This function is called to serialize an object hierarchy
- loads() – This function is called to de-serialize a data stream
- https://www.geeksforgeeks.org/pickle-python-object-serialization/

```
# Python program to illustrate
# pickle.dumps and .loads()
import pickle
import pprint

data1 = [ { 'a':'A', 'b':2, 'c':3.0 } ]
print 'BEFORE:',
pprint.pprint(data1)

data1_string = pickle.dumps(data1)

data2 = pickle.loads(data1_string)
print 'AFTER:',
pprint.pprint(data2)

print 'SAME?:', (data1 is data2)   #False
print 'EQUAL?:', (data1 == data2)   #True
```

Serialization Alternatives

- Capture the object data in a flat or nested data representation
  - Flat: { "Type" : "A", "field1": "value1", "field2": "value2", "field3": "value3" }
  - Nested: {"A" { "field1": "value1", "field2": "value2", "field3": "value3" } }
- Most languages provide functions or libraries for dealing with conversions into and between these formats
- Examples:
  - CSV – Comma Separated Values

    “Vehicle”,”year”,1997,”make”,"Ford",”model”,"E350"
  - JSON – JavaScript Object Notation

    {“Vehicle”:[{“year”:1997,”make”:”Ford”,”model”:”E350”}]}
  - XML – Extensible Markup Language

    <vehicle year=1997 make=“Ford”>E350</vehicle>
  - YAML - YAML Ain't Markup Language

```
- vehicle
      year: 1997
      make: Ford
      model: E350
```

Next steps

- Close to caught up on Grading…
- Project 4 due noon Wed 11/4
  - First part of three for the semester project
  - Get started sooner than later – a lot of parts
- Graduate Draft Presentation due noon Wed 11/11
  - Expecting a thorough research effort, not a surface topic review
- Quizzes are back this weekend, right Dwight? ☺
- Article Reviews are available for extra bonus points…
- New discussion topic is up… Visit Piazza often – it is for your participation grade, so participate!
- Coming up: Finishing up the patterns this week – Prototype, Visitor, more OO concepts, bonus exercises, more…
- If you need help – Office hours, Piazza, e-mail – we are here for you!