

Lecture 14: Callbacks, Singletons and Wrappers

Callback Pattern

- Context: controlling entity A calls/uses controlled entity B
- Problem:
 - B can have no control over A
 - How can B initiate something that is under the control of A without taking control of A?
- Solution: A gives B a pointer to one of its functions to call, or an object to “execute”, when the situation occurs

Callback Objects

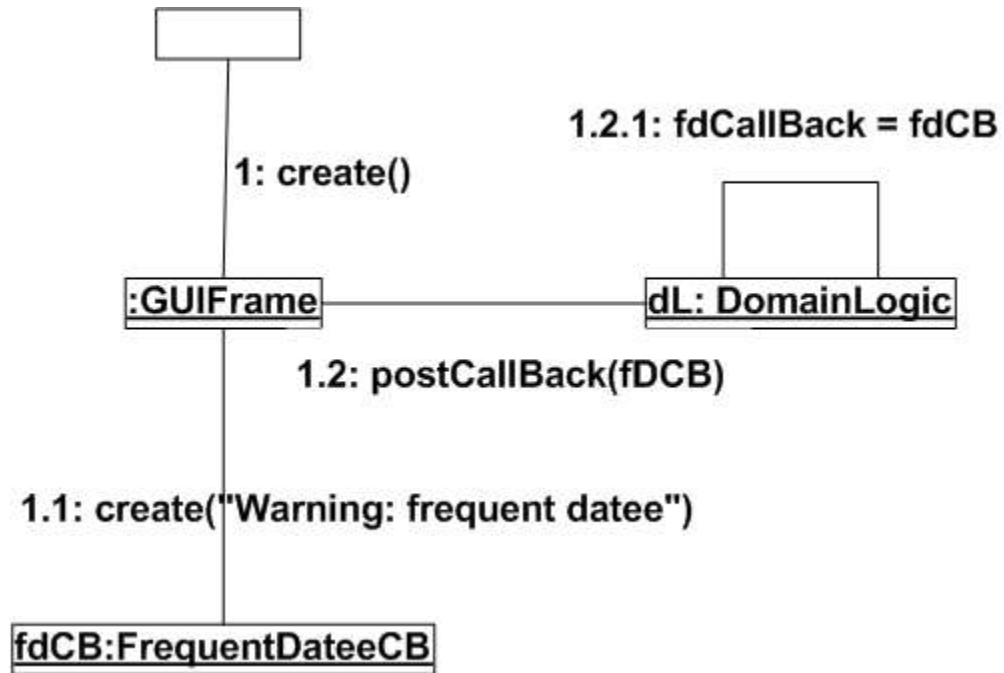
- Controller A will call a method B.b(a) in controllee B with an object a
- a must have an expected method a.callback() that is known to B and which will be called by B when the callback situation arises
- A must know the name of the method B.b() in B to call

Callbacks and MV Separation

- Model View separation architecture(discussed in architecture lecture)
 - separation facilitates change of “look and feel” where Model is retained but View changed
 - controller is view, controllee is model
 - view posts callback object with model by calling appropriate method in model
 - callback object must have expected callback method

DS Callback Example

- Frequent Datee Feature
 - normally GUI will call getDate() in DL when user indicates this user option
 - getDate() will return date information which GUI will display
 - if the found date has been dated many times, we want DL to display a GUI message (i.e. execute something in the GUI)
 - GUI registers a FrequentDatee call back object w



Event Delegation and Callbacks

- Components and events, e.g. button b
- Options
 - i) if b is pushed, b's special response method is called
 - ii) other code tells b what to do when button is pushed. “Register” a callback (actionListener) object x with b by executing `b.addActionListener(x)`
 - Callback pattern? sort of

Related Patterns

- Can be used for CallBack
 - Publisher/Subscriber
 - General pattern
 - Observer/Observable
 - Supported by classes in Java library

Publisher-Subscriber

- Special Event Manager object
- Objects interested in an event, register with event manager
- Objects who recognize event, publish occurrence of event with event manager
- Event manager informs subscribers
 - Expected to have a specific method that will be called

Observer/Observable

- Observers register with Observable
- Observable recognizes event and calls specified method in observer
- Java Observer
 - Interface class, implemented by observers
- Java Observable
 - Class, extended by callback caller classes

Java Observer Class

- Implementers must have an `update()` method which the `Observable` object `x` will call
 - `update(Observable x, Object y)`
- When `Observable` object `x` calls the `update()` method for a registered observer, it passes itself as an identifier and also a data object `y`

Java Observable Class

- Users of this class extend it
- Inherited methods from Observable
 - addObserver(Observer z)
 - Adds the object z to the observable's list of observers
 - notifyObserver(Object y)
 - Calls the update() method for each of the observers in its list of observers, passing y as the data object

Singleton Pattern

- Context: class for which we only want a single instance, and to which we may want global access
- Solution:
 - reference the object globally through its class definition
 - class variable called “instance” whose value is an instance of the object
 - static method called instance() that
 - Creates an instance if not created yet
 - Returns the instance of the class

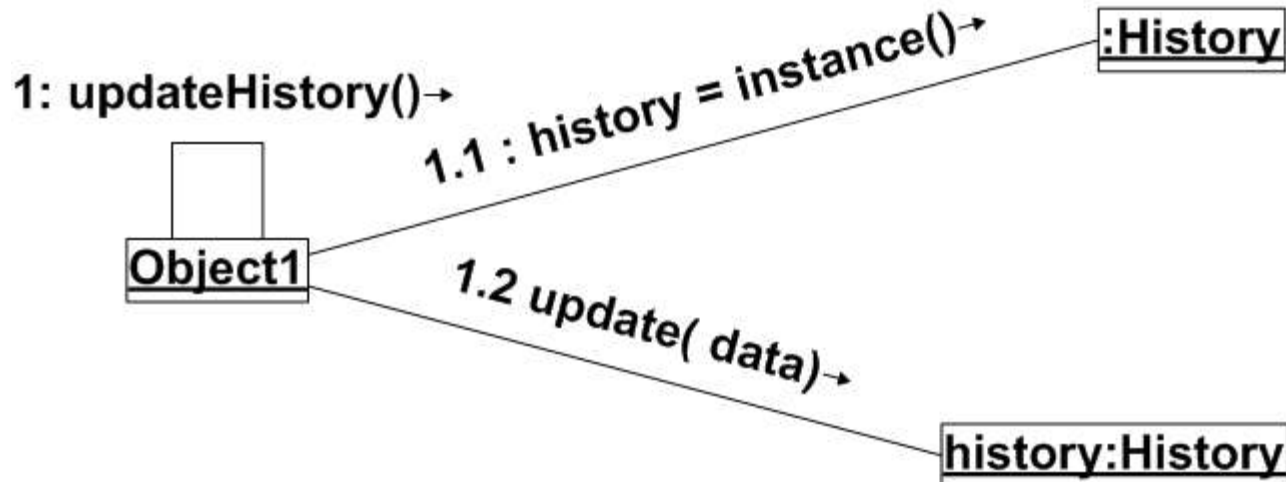
DS Singleton Example

- Want to have a globally accessible object that records session history statistics
- Gets updated from different places, e.g. when different kinds of messages are generated
- Use the singleton pattern with a History class

DS Singleton History Class

History
+\$instance
+\$instance() : History

\$ indicates a static/class variable/method



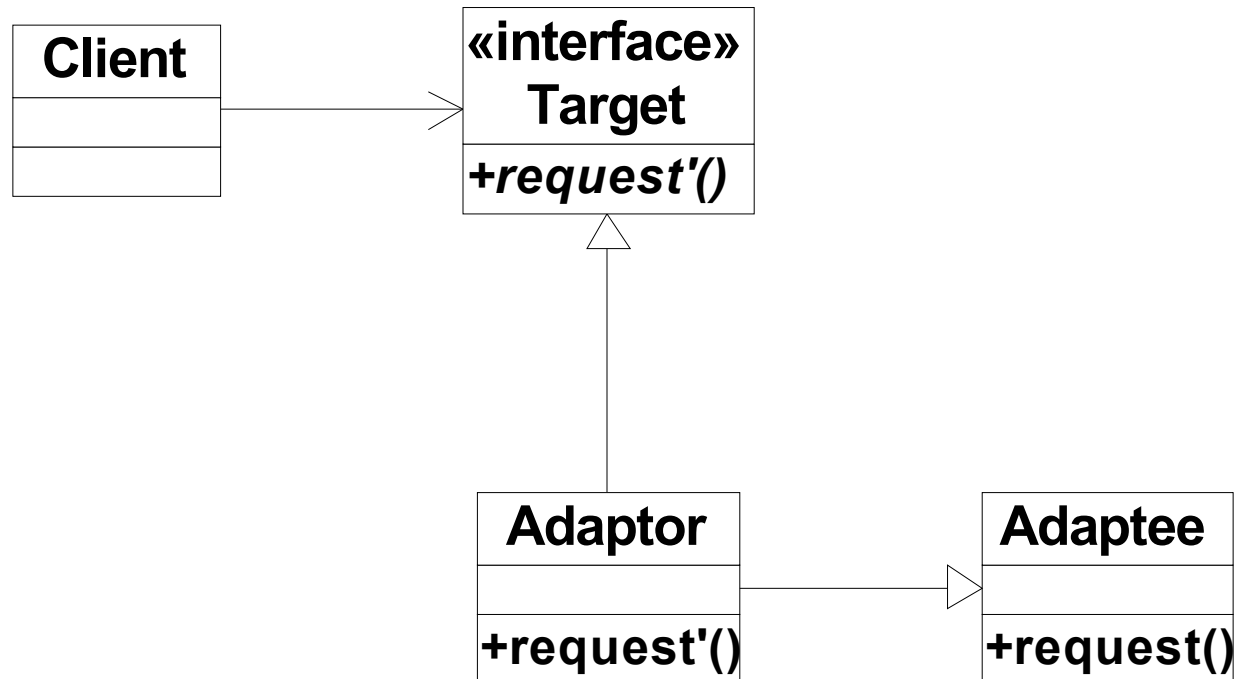
Wrappers

- Take an existing entity and alter its behavior or interface by “embedding” it inside a class
- Types of wrappers
 - Adapter: emphasis is on altering interface, but could add some additional responsibilities responsibilities/behavior
 - Decorator: alters behavior or responsibilities but not interface
 - Combined

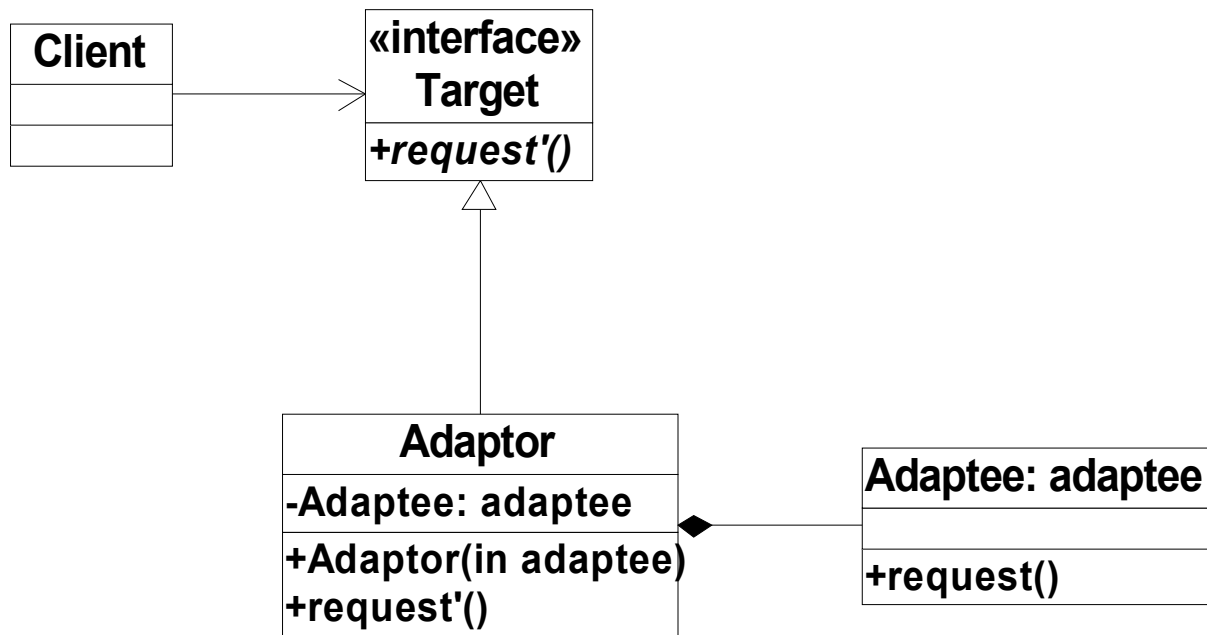
Adapter Strategies

- Inheritance
 - Adaptor class implements the required new interface
 - Adaptor extends the adaptee, with new methods for performing the adaptee responsibilities, which conform to the new interface
- Composition
 - Adaptor class implements the required new interface
 - Adaptor's constructor takes an adaptee object as an argument
 - Adaptor methods reference the adaptee object via its adaptee class methods

Adapter - Inheritance



Adaptor - Composition



Adaptor Example – Primitive Data Reification

- Objectification of primitive data types in Java, i.e. wrap an instance in a class
- Why?
 - When primitive is used as a parameter, it will be passed as call by value.
 - Cannot change its value inside the method
 - May want to add to functionality,
 - e.g. `Int` is the objectification for `int`
 - `Int.parseInt(String s)` will return the `int` for `s`.

User Defined Reification Example

- Interface

```
BoolRefInterface { set(boolean b); boolean get() }
```

- Adaptee - boolean value

- Adaptor

```
class BooleanRef implements BoolRefInterface  
{ public boolean val;  
  public BooleanRef(boolean x){this.val = x}  
  public set(boolean b) {this.val = b;}  
  public get() {return this.val;}  
}
```

Adaptor Example – Interface Adaptors

- In order to carry out certain operations, a class may be required to implement a special interface
- Interface may contain many functions only a few of which are normally relevant
- Tedious to continually supply null definitions for the unneeded functions

Interface Adaptor Strategy

- Construct a utility class that implements the interface and supplies null definitions for all the methods
- User extends the utility adaptor class instead of implementing the interface
- User supplies definitions for the methods of interest

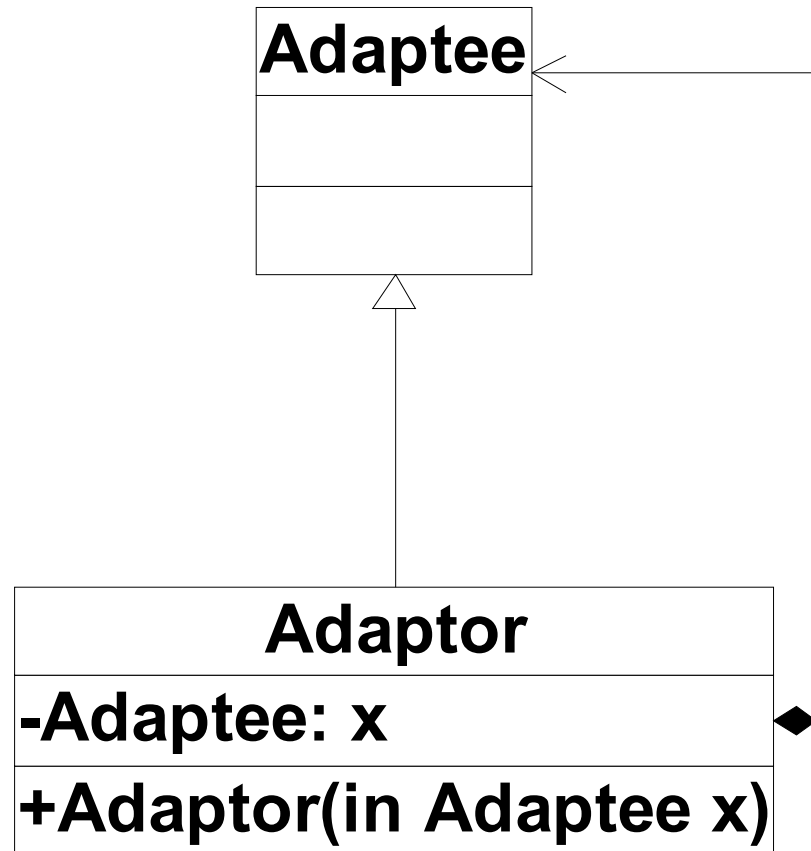
Java Interface Adaptor Example

- WindowListener interface
 - implemented by any object that wants to register itself as a listener for events generated by a Window
 - 7 methods in interface
 - window closing event that occurs when user clicks x in upper right corner requires definition for windowClosing() method only
- WindowAdaptor implements interface with 7 null functions
 - extend WindowAdapter and override windowClosing()

Decorator Strategy

- Decorator wrapper- emphasis:
 - Goal is to alter the run time behavior of an object
 - Wrapper should preserve the interface
- Possible adaptor class approach:
 - adaptor subclasses adaptee so has same interface
 - adaptor constructor takes wrappee as an argument, so it is altering it at runtime

Decorator Class Diagram



Decorator example - DS

- DateRequest objects `x` will look for a date when `x.execute()` is performed
- Want to add a special capability so that if Pamela is logged on, she always gets Dave, who is described using her desired props
- Solution: “decorate” DateRequest with a new DecoratedDateRequest class

Decorator Pattern Variation

- DecoratedDateRequest interface
 - DecoratedDateRequest does not subclass DateRequest but has the same interface
- DecoratedDateRequest constructor
 - instead of taking a DateRequest object as a constructor parameter, it takes the constructor parameters for DateRequest, and uses them to build a DateRequest object if it needs it

```

class DateRequest
{ String requester;
  DaterPreferences desiredProperties;
  public DateRequest(String name, DaterPreferences daterPreferences)
  {
    requester = name;
    desiredProperties = daterPreferences;
  }
  public MemberData execute()
  {
    MemberData memberData;
    memberData = dataBase.selectMember(desiredProperties);
    if (!(memberData == null))
    {
      int dateeCount = ++memberData.adminData.timesDated;
      if (dateeCount > frequentDaterThreshold)
      {
        callBack.execute();
      }
      String name = memberData.name;
      dataBase.updateField(name, "DateeCount", String.valueOf(dateeCount));
      return memberData;
    }
    return(null);
  }
}

```

```

class DecoratedDateRequest
{ String requester;
  DaterPreferences desiredProperties;
  public DecoratedDateRequest(String name, DaterPreferences daterPreferences)
  { requester = name;
    desiredProperties = daterPreferences;
  }
  public MemberData execute()
  { if (requester.equals("Pamela"))
    { MemberData memberData = new MemberData();
      DateeData dateeData = new DateeData();
      memberData.name = "Dave";
      memberData.dateeData.occupation = "Programmer";
      memberData.dateeData.religion = desiredProperties.religion;
      memberData.dateeData.gender = desiredProperties.gender;
      memberData.dateeData.email = "Dave@hotmail.com";
      memberData.adminData.timesDated = 12;
      return memberData;
    }
    else
    { return(new DateRequest(requester, desiredProperties).execute());
    }
  }
}

```

Decorator Example – Java Stream Wrapper

- FileInputStream has a simple read() operation that reads a single byte
- Very inefficient, would like to buffer the input
- Solution: “decorate” FileInputStream with a decorator class BufferedInputStream

Decorator Pattern Variation

- BufferedInputStream interface
 - has the same interface as FileInputStream, but it is not a subclass
- BufferedInputStream constructor
 - does follow the pattern, its constructor can take a FileInputStream object as an argument, but actual constructor argument type is more general

FileInputStream and BufferedInputStream

