CSE 223: Programming -2
09-Structural Design Patterns (II)

Prof. Dr. Khaled Nagi

Department of Computer and Systems Engineering,

Faculty of Engineering, Alexandria University, Egypt.

Agenda

- *Describe common ways that different types of objects can be organized to work with each other*
- *Structural patterns help you compose groups of objects into larger structures, such as complex user interfaces or accounting data.*
- Adapter
- Iterator
- Bridge
- Façade
- Flyweight
- Dynamic linkage
- Virtual proxy
- Decorator
- Cache management

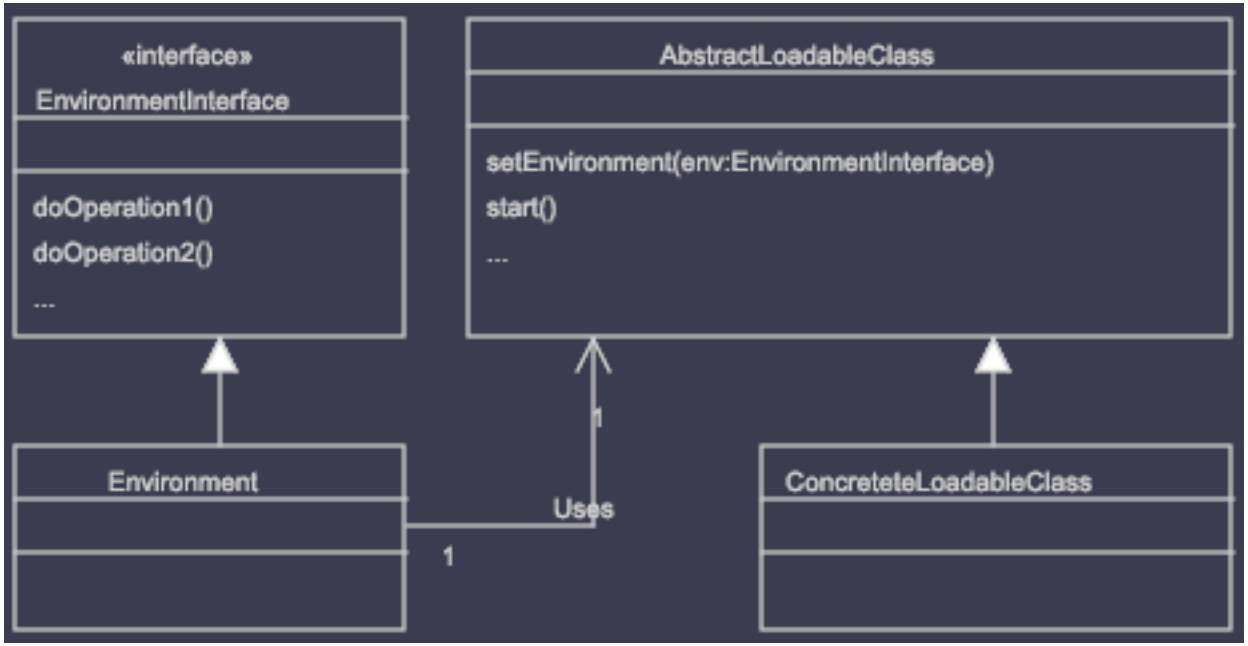
Part II

Dynamic linkage

Dynamic linkage

- Defines procedure of arbitrary loading and usage of classes at runtime.
- Only requirement for the classes, that they implement an interface, known at compile-time.
- Dynamic linkage simply reduces compile-time restrictions on behalf of more run-time freedom regarding class implementation.
- The procedure allows change in implementation of classes at run-time, without neither re-compiling nor stopping the program.
- Forces
  - A program must be able to load and use arbitrary classes that it has no prior knowledge of.
  - A loaded class must be able to call back to the program that loaded it.

Dynamic linkage



- Consequences
  - Subclasses of the AbstractLoadableClass class can be dynamically loaded.
  - The operating environment and the loaded classes do not need any specific foreknowledge of each other.
  - Dynamic linkage increases the total amount of time it takes for a program to load all of the classes that it uses. However, it does have the effect of spreading out, over time, the overhead of loading.


ALEXANDRIA UNIVERSITY

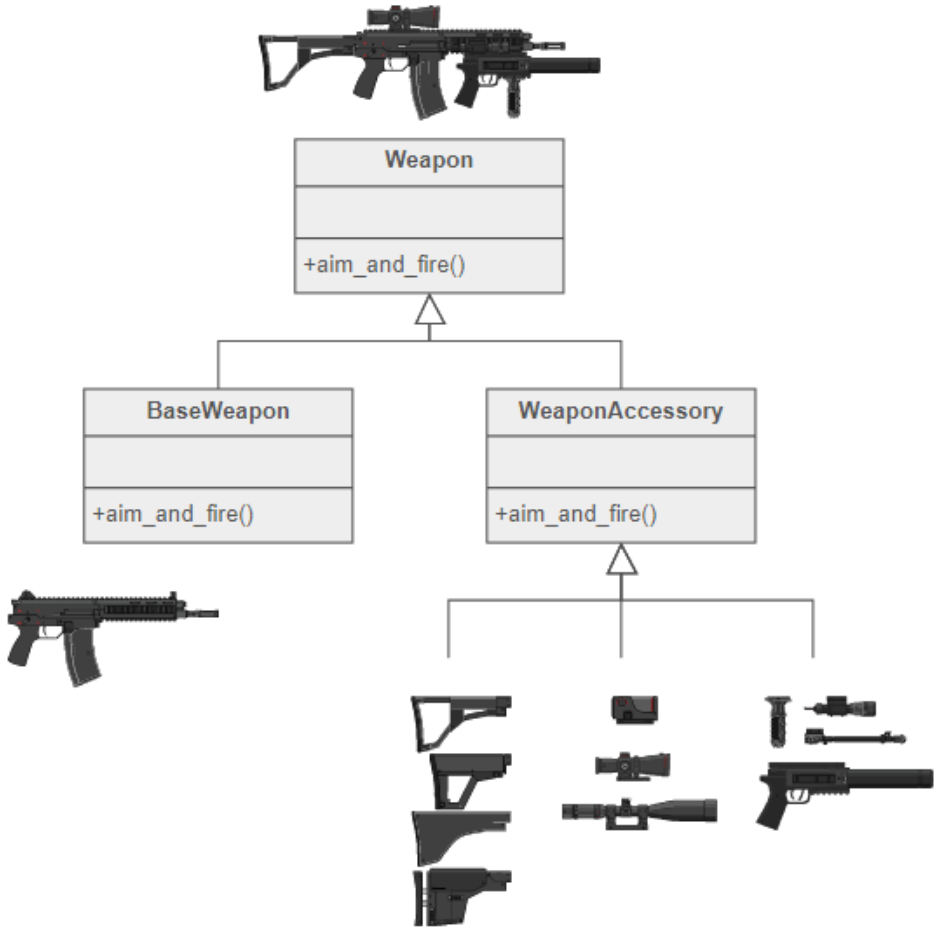
Decorator

Decorator

- Intent
  - Attach additional responsibilities to an object dynamically.
  - Decorators provide a flexible alternative to subclassing for extending functionality.
  - Client specified embellishment of a core object by recursively wrapping it.
  - Wrapping a gift, putting it in a box, and wrapping the box.
- Problem
  - You want to add behavior or state to individual objects at run time.
  - Inheritance is not feasible because it is static and applies to an entire class.

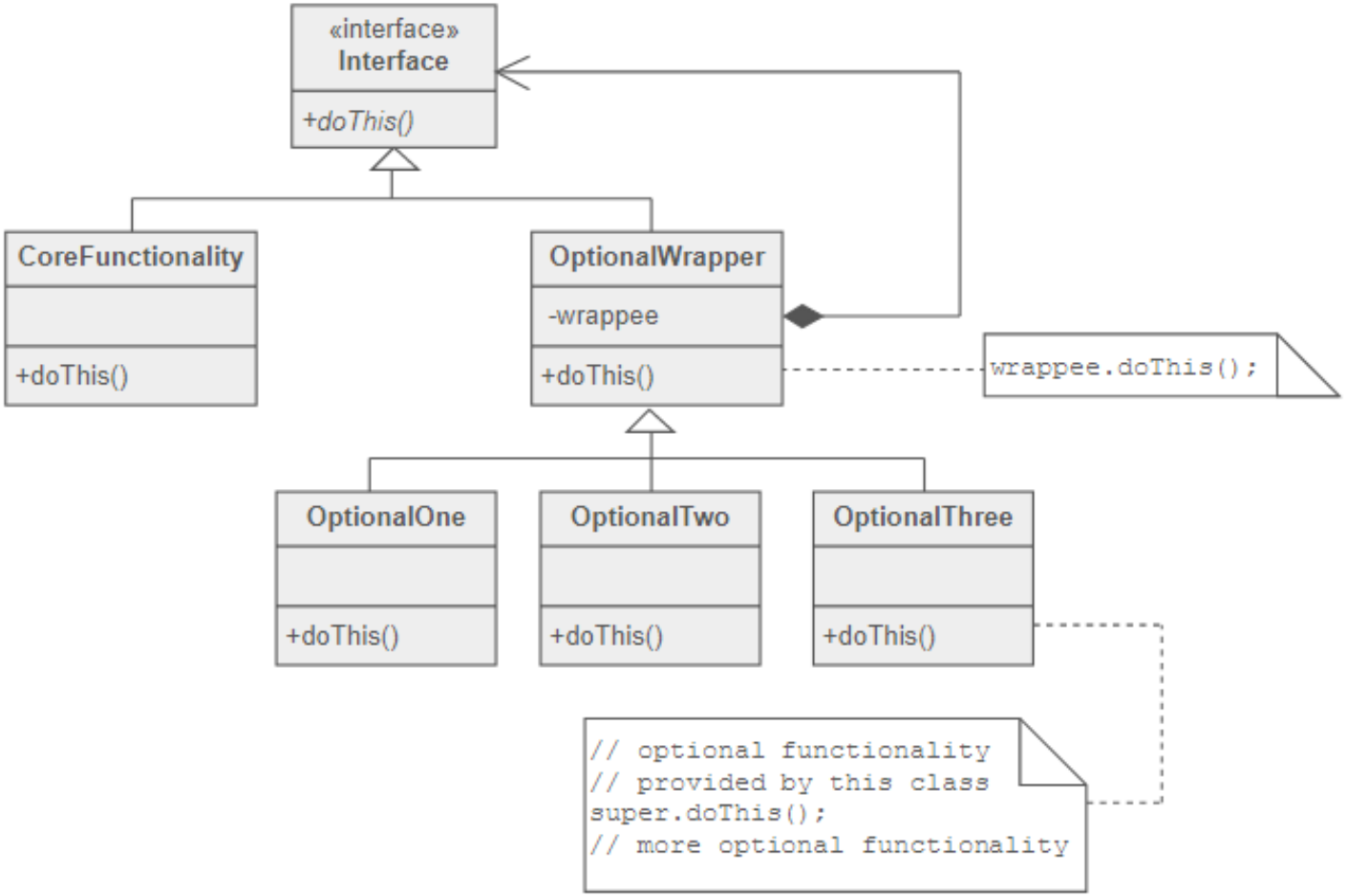
Decorator

- Example



Decorator

- UML



Decorator

- Ensure the context is: a single core (or non optional) component, several optional embellishments or wrappers, and an interface that is common to all.
- Create a "Lowest Common Denominator" interface that makes all classes interchangeable.
- Create a second level base class (Decorator) to support the optional wrapper classes.
- The Core class and Decorator class inherit from the LCD interface.
- The Decorator class declares a composition relationship to the LCD interface, and this data member is initialized in its constructor.
- The Decorator class delegates to the LCD object.
- Define a Decorator derived class for each optional embellishment.
- Decorator derived classes implement their wrapper functionality and delegate to the Decorator base class.
- The client configures the type and ordering of Core and Decorator objects.

Decorator – Example in JAVA

```
public class DecoratorStream {
  static BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

  interface LCD {
    void write( String[] s );
    void read(  String[] s );
  }

  static class Core implements LCD {
    public void write( String[] s ) {
      System.out.print( "INPUT:    " );
      try {
        s[0] = in.readLine();
      } catch (IOException ex) { ex.printStackTrace(); }
    }
    public void read( String[] s ) {
      System.out.println( "Output:   " + s[0] );
  } }

  static class Decorator implements LCD {
    private LCD inner;
    public Decorator( LCD i ) { inner = i; }
    public void write( String[] s ) { inner.write( s ); }
    public void read(  String[] s ) { inner.read(  s ); }
  }
```

Decorator – Example in JAVA

```
static class Authenticate extends Decorator {
  public Authenticate( LCD inner ) { super( inner ); }
  public void write(  String[] s ) {
    System.out.print( "PASSWORD: " );
    try {
      in.readLine();
    } catch (IOException ex) { ex.printStackTrace(); }
    super.write( s );
  }
  public void read( String[] s ) {
    System.out.print( "PASSWORD: " );
    try {
      in.readLine();
    } catch (IOException ex) { ex.printStackTrace(); }
    super.read( s );
} }
```

```
static class Scramble extends Decorator {
  public Scramble( LCD inner ) { super( inner ); }
  public void write( String[] s ) {
    super.write( s );
    System.out.println( "encrypt:" );
    StringBuffer sb = new StringBuffer( s[0] );
    for (int i=0; i < sb.length(); i++)
      sb.setCharAt( i, (char) (sb.charAt( i ) - 5) );
    s[0] = sb.toString();
  }
  public void read( String[] s ) {
    StringBuffer sb = new StringBuffer( s[0] );
    for (int i=0; i < sb.length(); i++)
      sb.setCharAt( i, (char) (sb.charAt( i ) + 5) );
    s[0] = sb.toString();
    System.out.println( "decrypt:" );
    super.read( s );
} }
```

Decorator – Example in JAVA

```
  public static void main( String[] args ) {
    LCD stream = new Authenticate( new Scramble( new Core() ) );
    String[] str = { new String() };
    stream.write( str );
    System.out.println( "main:     " + str[0] );
    stream.read( str );
} }
```

Output

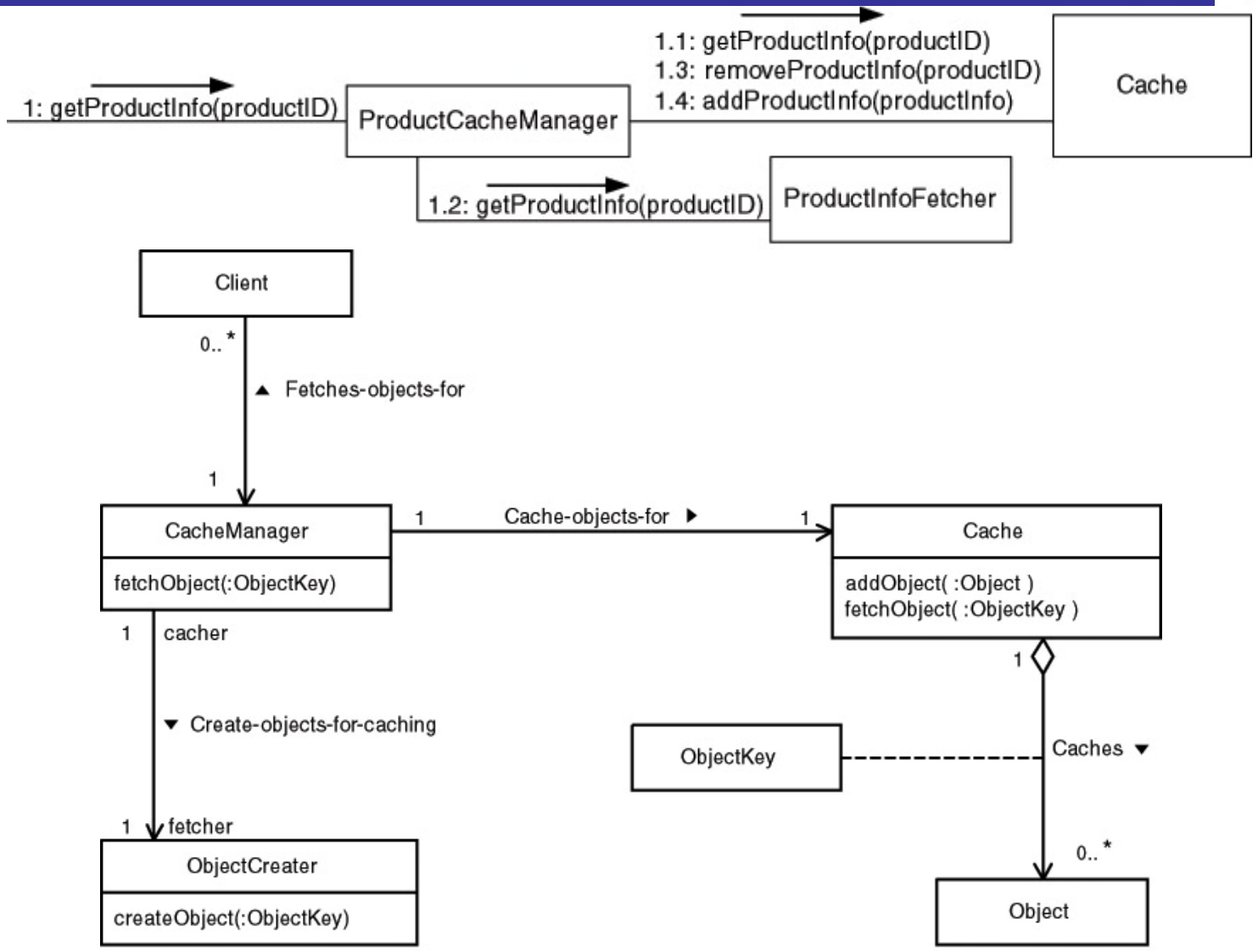
```
PASSWORD: secret
INPUT:    the quick brown fox
encrypt:
main:     oc`xlpd^fe]mjrizajs
PASSWORD: secret
decrypt:
Output:   the quick brown fox
```

Cache management

Cache management

- Context
  - Suppose you are writing a program that allows people to fetch information about products in a catalog.
  - Fetching all of a product's information can take a few seconds because it may have to be gathered from multiple sources.
  - Keeping a product's information in the program's memory allows the next request for the product's information to be satisfied more quickly, since it is not necessary to spend the time to gather the information.
  - Keeping information in memory that takes a relatively long time to fetch into memory for quick access the next time it is needed is called caching.
  - The large number of products in the catalog makes it infeasible to cache information for all the products in memory.
  - What can be done is to keep information for as many products as feasible in memory. Products guessed to be the most likely to be used are kept in memory so they are there when needed.
  - Products guessed to be less likely to be used are not kept in memory.

Cache management



Cache management

- Forces
  - There is a need to access an object that takes a long time to construct or fetch. Typical reasons for the construction of an object being expensive are that its contents must be fetched from external sources or that it requires a lengthy computation. The point is that it takes substantially longer to construct the object than to access the object once it is cached in internal memory.
  - When the number of objects that are expensive to construct is small enough that all of them can fit comfortably in local memory, then keeping all of the objects in local memory will provide the best results. This guarantees that if access to one of these objects is needed again, it will not be necessary to incur the expense of constructing the object again.
  - If very many expensive to construct objects will be constructed, then all of them may not fit in memory at the same time. If they do fit in memory, they may use memory that will later be needed for other purposes. Therefore, it may be necessary to set an upper bound on the number of objects cached in local memory.

Cache management

- Forces
  - An upper bound on the number of objects in a cache requires an enforcement policy. The enforcement policy will determine which fetched objects to cache and which to discard when the number of objects in the cache reaches the upper bound. Such a policy should attempt to predict which objects are most and least likely to be used in the near future.
  - Some objects reflect the state of something outside of the program's own memory. The contents of such objects may not be valid after the time that such objects are created.