Recap

- Last time...
- Examples of *design patterns*
  - **Observer**: publisher/subscriber mechanism
  - **Strategy**: vary an algorithm at runtime
  - **Singleton**: global object
- UML Notation
  - Sequence Diagrams

Objectives

- Cover more *design patterns*
  - **Decorator**
  - **Builder**
  - **State**

The Decorator Pattern

- Way to attach behavior or responsibilities to an object at runtime
- Also called “wrapper”
- Essential Classes:
  - Component Interface
  - Base Classes
  - Decorator Classes

Decorator

“Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.” *GoF*

Example: UI Widgets and Views

- Windowed user interfaces are often used to illustrate the Decorator pattern[1]
- How would you organize the classes that implement a windowing system for a UI?
  - For the base window class you would need a simple box that contains text
  - Also, variations on the basic text box that allow for scroll bars, menus, fancy borders etc.
- Inheritance is not the best solution because it requires all possibilities to be known at the time of design
  - Would have to write a separate class for each possible variation of our desired UI Features (for example, scrollable window with a menu but without fancy border)
  - Not possible to change at runtime

[1] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

Example: Streams and File Readers

```
BufferedReader br = new BufferedReader(
                          new FileReader("./text3.txt")
                        );
```

- The Decorator pattern used in the `java.io` classes
- The code above shows a common idiom for reading a file in Java
- New functionality is obtained by passing the `FileReader` in the constructor of the `BufferedReader`
- A plain `FileReader` has been *decorated* with a `BufferedReader`

java.io Classes

- Many classes in the Java io library follow a similar pattern
- This code shows the example for an `InputStream`

```
InputStream is =
new MyInputStream(
  new BufferedInputStream(
    new FileInputStream("./text3.txt")));

BufferedReader bufr =
  new BufferedReader(
    new InputStreamReader(is));
StringBuilder sb = new StringBuilder();
String line = new String();
while((line = bufr.readLine()) != null) {
  sb.append(line);
}
bufr.close();
is.close();

dString = sb.toString();
```

Examining BufferedInputStream

```
public class BufferedInputStream extends FilterInputStream
```

- If we follow the pattern then we can write our own classes to extend the functionality of input streams in new ways
- As a starting point, we might examine the source code of `BufferedInputStream` code to see what our own class would need to do
- We could use our implementation just as we would a `BufferedInputStream`

```
InputStream is =
    new MyInputStream(
        new BufferedInputStream(
            new FileInputStream("./text3.txt")));
```

Decorator Example

- For illustration, we can develop a simple example of decorator for printing formatted text to the console
- Notice usage of the Java *reflection*

```
public static void main(String[] args) {
  TextComponent tx = new TextBase("The Text!");
  System.out.println(tx.getClass().toString());
  System.out.println(tx.produceText());

  tx = new CapitalDecorator(tx);
  System.out.println(tx.getClass().toString());
  System.out.println(tx.produceText());

  tx = new BorderDecorator(tx);
  System.out.println(tx.getClass().toString());
  System.out.println(tx.produceText());

  tx = new DashBorderDecorator(tx);
  System.out.println(tx.getClass().toString());
  System.out.println(tx.produceText());

}
```

Decorator Example (cont.)

- Intended output from the application is shown below
- As we decorate the base class with more, the object assumes more responsibilities

```
class com.example.textdecorator.TextBase
The Text!
class com.example.textdecorator.CapitalDecorator
THE TEXT!
class com.example.textdecorator.BorderDecorator
*** THE TEXT! ***
class com.example.textdecorator.DashBorderDecorator
--- *** THE TEXT! *** ---
```

The Base Class

- There are two different kinds of classes in the decorator pattern: the base classes and the decorator classes
- The code below shows the base class

```
public class TextBase extends TextComponent {

  private String s;

  public TextBase(String s) {
    this.s = s;
  }

  @Override
  public String produceText() {
    return s;
  }
}
```

The Decorator Interface

- The Decorator interface contains a component field called `next` to which it delegates the functionality of `produceText()`

```
public abstract class TextDecorator extends TextComponent {
  protected TextComponent next;

  public TextDecorator(TextComponent t) {
    this.next = t;
  }

  public String produceText() {
    return this.next.produceText();
  }
}
```

A Concrete Decorator Implementation

- We can introduce new behaviors in our decorator implementations

```
public class DashBorderDecorator extends TextDecorator {

  public DashBorderDecorator(TextComponent t) {
    super(t);
  }

  @Override
  public String produceText() {
      return "--- " + super.produceText() + " ---";
  }
}
```

Open-Closed Principle and Delegation

Open-Closed Principle

Classes should be open for extension but closed for modification

- Want to extend functionality without having to modify source code
- When using a deep inheritance tree the design can become rigid or there may be too much implementation code in the base classes
- Delegation:
  - The Decorator delegates part of its responsibilities to the object that it wraps

Builder Pattern

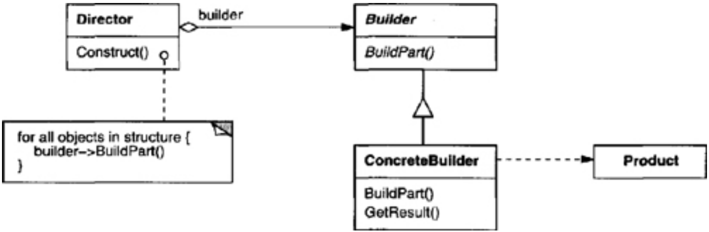
- Classes Involved:
  - Builder: interface for building the parts of an object
  - ConcreteBuilder: implements the Builder interface
  - Director: constructs an object using a Builder
  - Product: domain object that is being constructed
- Motivation:
  - We have a complex object with many fields that need to be set etc. But we want to make sure that an invalid object cannot be instantiated
- Example of a *creational* design pattern

Builder[2]

"Separate the construction of a complex object from its representation so that the same construction process can create different representations."

[2] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

Builder[3]



- The Builder pattern is often implemented using a *fluent* interface
  - Call the methods of the builder in a single large statement via method chaining
  - At the end of the method chain call `build()` to actually return the object

[3] Class diagram from ibid.

Example: Airline Reservation System (1)


```
public class Ticket {
    private String origin;
    private String destination;
    private String date;
    private String travelClass;

    public Ticket(String orig,
                  String dest,
                  String date,
                  String trav) {
        this.origin = orig;
        this.destination = dest;
        this.date = date;
        this.travelClass = trav;
    }
}
```

- Imagine you are working on a system for an Airline to manage reservations
- Use the class `Ticket` to model a ticket
    - How could we apply the *Builder* pattern in this context?

Example: Airline Reservation System (2)

```
public class TicketBuilder {
    String origin;
    String destination;
    String date;
    String travelClass;

    // methods to construct Ticket
    // ...
}
```

- First step would be to create a new class `TicketBuilder` that has the same fields as Ticket
- We will need methods to set the values of these fields (origin, destination, ...)
- To have a fluent interface the setter methods can all return instances of the builder
  - Question: how should we name these setter methods?

Example: Airline Reservation System (3)

```
public class TicketBuilder {
    // ...

    public TicketBuilder from(String f) {
        this.origin = f;
        return this;
    }

    public TicketBuilder to(String t) {
        this.destination = t;
        return this;
    }

    // ...
}
```

- If we think about the problem in terms of natural language some method names suggest themselves
- With this naming, the chain of method calls on the builder reads more like a sentence

Example: Airline Reservation System (4)


```
public class TicketBuilder {
    // ...

    public Ticket build() {
        return new Ticket(this);
    }

    // ...
}
```

- The `build()` method returns the actual `Ticket`
- Note that we can put any logic related to the creation of the ticket here
  - For example, (origin, destination) should be an actual route of this airline and there should be a flight on the date
- Notice also that we (implicitly) defined a new constructor for `Ticket` that takes a `TicketBuilder` as a parameter

Example: Airline Reservation System (5)

```
public class Ticket {
    // ...

    public Ticket(TicketBuilder tb) {
        this(tb.origin,
             tb.destination,
             tb.date,
             tb.travelClass);
    }

    // ...
}
```

- The new constructor would look something like this
- We have used *constructor chaining* to utilize the constructor we already have

Using the Builder

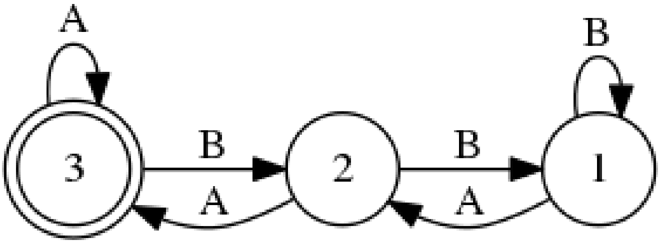
```
public class App {
    public static void main(String[] args) {
        TicketBuilder tb = new TicketBuilder();

        Ticket t = tb.from("Almaty")
                     .on("March 12")
                     .to("Frankfurt")
                     .withClass("Economy")
                     .build();

        System.out.println(t);
    }
}
```

- This example shows how we could construct a `Ticket` from a `TicketBuilder`
- Notice that the order of the method calls does not matter (as long as we finish with `build()`)

Extensions

- *Builder* is an object creational pattern
- One of the primary motivations is to ensure that an object cannot be created in an invalid state
  - Suppose we exposed public setters on the `Ticket` class
  - Our system might end with `Ticket` objects with null fields or invalid routes
  - *Builder* allows us to keep that logic out of the Ticket class
- We can go further and use the same trick from *singleton*
  - Mark the constructor private
  - Add the builder as an inner class of `Ticket`

State Machines and DFAs



- Example of a basic state machine used in computer science: basic model of computation, matching patterns
- Two types of states: accept and reject (double-circle)
- Machine consumes a character and follows the appropriate edge to change state
- The patterns that can be recognized with a DFA are equivalent to the patterns we can specify with strict *regular expression* syntax

Other uses of State Machines

- State Machines can also be used to model simple mechanistic systems such as vending machines and turnstiles
- To function properly, a vending machine needs to change state based on actions: inserting coins, selecting items, etc.

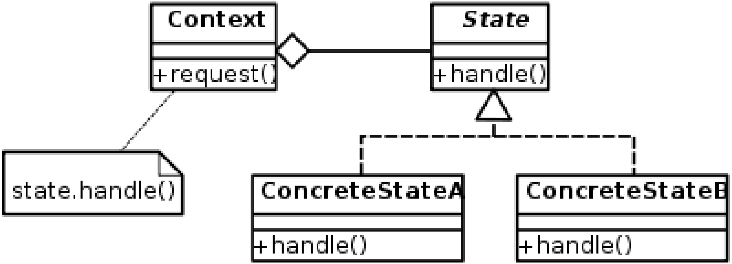
The State Pattern

- Create a state machine in an object-oriented fashion
- Classes Involved:
  - Context
  - State Interface
  - Concrete State Classes

State

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

State Pattern Class Diagram



- The Context delegates its public methods to a state object that it aggregates
- State is an abstraction
- Concrete implementations of State define the behavior of the context

The Context Class


```
public class StateContext {
    final State state1 = new State1(this);
    final State state2 = new State2(this);
    final State state3 = new State3(this);
    private State currentState;

    public StateContext() {
        this.currentState = state1;
    }

    public void actionA() {
        this.currentState.actionA();
    }
    public void actionB() {
        this.currentState.actionB();
    }

    public boolean inAcceptState() {
        return this.currentState.isAccept();
    }

    public State getCurrentState() {
        return currentState;
    }
    public void setCurrentState(State currentState) {
        this.currentState = currentState;
    }
}
```

- The context class for our example

The State Abstraction

```
public abstract class State {
    protected StateContext sc;
    protected boolean accept = false;

    public abstract void actionA();

    public abstract void actionB();

    public boolean isAccept() {
        return this.accept;
    }
}
```

- Concrete States inherit from this abstract class

A Concrete State


```
public class State1 extends State {
    public State1(StateContext stateContext) {
        this.sc = stateContext;
        this.accept = false;
    }

    @Override
    public void actionA() {
        this.sc.setCurrentState(this.sc.state2);
    }

    @Override
    public void actionB() {
        this.sc.setCurrentState(this.sc.state1);
    }
}
```

- A concrete implementation of State 1 from the example

Using the State Machine


```
public class StateTest {
   public static void main(String[] args) {
      StateContext sc = new StateContext();

      sc.actionA();
      sc.actionA();
      sc.actionB();

      // Is machine in an accept state
      // after receiving AAB
      sc.inAcceptState();
   }

}
```

- Example of the behavior we would like from our class
- Sequence of actions executed on the context will make it appear to change its behavior from the outside
- Internal state evolves in pre-defined way

State Pattern Summary

- State Pattern allows an object to have an internal state that changes its behavior
- Each state is represented by a class (will increase the number of classes in your design)
- The class diagrams for State and Strategy are the same
  - *Strategy*: alternative to subclassing
  - *State*: prevent a lot of conditional statements from appearing in your main class

Git

- Git is the version control system created by Linus Torvalds while he was developing the first versions of linux
- Allows you to work both locally and collaboratively
- Maintain a history of all edits in a project
- Work in separate *branches*
- The command line tool is simply `git` and is available from `https://git-scm.com/`

Initializing a Git Project

- A project in Git is managed by a special directory called `.git` in the root of your project
- Git projects are usually called repos (repository)
- You can create a new git project in one of two ways
  - Call `git init` in the root directory of your project
  - Clone an existing repository with `git clone <url>`

Status and Staging

- `git status` will show the current state of your project: changes that have been staged but not committed, files that have been modified but not staged
- Git uses a two-stage commit process, changes are moved first to a "staging area", and then are eventually committed using the `git commit` command

Making Changes

NAZARBAYEV UNIVERSITY
SCHOOL OF ENGINEERING AND DIGITAL SCIENCES

- Edit your files as you would normally
- When you reach a point where you want to mark your progress use `git add <file>` to stage the changes you made
- Run `git commit -m <message>` to actually commit your changes
- `git diff` tells you what is staged but not committed

Working on Different Branches

- Git allows you to split your development into separate branches
- See which branch you are on by running `git branch`
- Create a new branch with `git branch <branch name>`
- Switch between branches using `git checkout <branch name>`

Creating a Pull Request

- To contribute to a project on github (such as an open source project) you create a *pull request*
- The process is roughly as follows
  - Make your own copy of the repository you want to contribute to, called a "fork"
  - Clone your fork to your local machine
  - Make any changes you need, and test them
  - Push the changes to your remote: `git push origin master` for example
  - Use the *New Pull Request* option at the original repository you want to contribute to

Collaboration

- The remote is shared by all members of a team, and each developer has their own complete copy
- Changes made by others can be retrieved from the remote by issuing a `git pull` command
- You share your update by pushing them upstream using `git push`