Material and some slide content from:
- Atif Kahn

SERVICES

COMPONENTS

OBJECTS

MODULES

Composition vs. Inheritance, Cloud Computing, and REST-based Architectures

Reid Holmes

Store Example

- Starbucks example
  - Beverages: house, dark
  - Toppings: whip, milk
  - Each beverage needs a cost() function
- Build class diagram for this system

Store Example

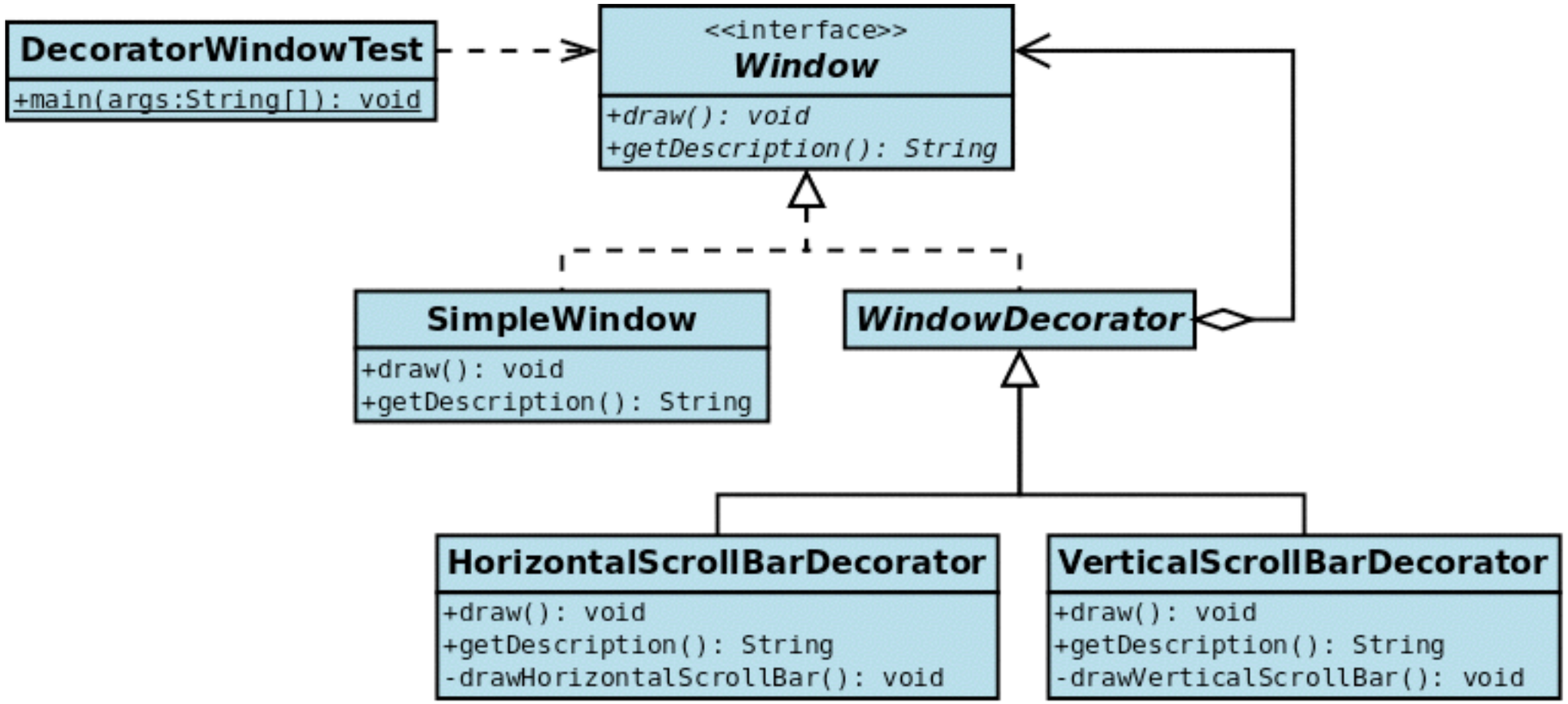
- Starbucks example
  - Beverages: house, dark, decaf, espresso
  - Toppings: whip, milk, soy, mocha, unicorn tears
  - Each beverage needs a cost() function
- Build class diagram for this system

Decorator

- Intent: “Dynamically add additional responsibilities to structures.”
- Motivation: Sometimes we want to add new responsibilities to individual objects, not the whole class. Can enclose existing objects with another object.
- Applicability:
  - Add responsibilities dynamically and transparently.
  - Remove responsibilities dynamically.
  - When subclassing is impractical.

Decorator

- Structure



- Participants:
  - Component / concrete component
  - Decorator / concrete decorator

Decorator (code ex)

```java
// the Window interface
interface Window {
    public void draw(); // draws the Window
    public String getDescription();
}

// implementation of a simple Window
class SimpleWindow implements Window {
    public void draw() {
        // draw window
    }

    public String getDescription() {
        return "simple window";
    }
}
// abstract decorator class
abstract class WindowDecorator implements Window {
    protected Window decoratedWindow;

    public WindowDecorator (Window decoratedWindow) {
        this.decoratedWindow = decoratedWindow;
    }
   public void draw() {
        decoratedWindow.draw();
    }
}

 public class DecoratedWindowTest {
     public static void main(String[] args) {
         Window decoratedWindow = new HorizontalScrollBarDecorator (
                 new VerticalScrollBarDecorator(new SimpleWindow()));
         // print the Window's description
         System.out.println(decoratedWindow.getDescription());}}
```

```java
// adds vertical scrollbar functionality
class VerticalScrollBarDecorator extends WindowDecorator {
    public VerticalScrollBarDecorator (Window decoratedWindow) {
        super(decoratedWindow);
    }
    public void draw() {
        drawVerticalScrollBar();
        super.draw();
    }
    private void drawVerticalScrollBar() { .. }
    public String getDescription() {
        return decoratedWindow.getDescription() +" and vert sb";
    }
}
// adds horizontal scrollbar functionality
class HorizontalScrollBarDecorator extends WindowDecorator {
    public HorizontalScrollBarDecorator (Window decoratedWindow) {
        super(decoratedWindow);
    }
    public void draw() {
        drawHorizontalScrollBar();
        super.draw();
    }
    private void drawHorizontalScrollBar() { .. }
    public String getDescription() {
        return decoratedWindow.getDescription() + "and horiz sb";
    }
}
```

Decorator

- Collaborations
  - Decorators forward requests to component object.
- Consequences:
  - More flexible.
    - (than static inheritance; arbitrary nesting possible)
  - Avoids feature-laden classes.
    - (KISS and add functionality as needed.)
  - Warn: Decorator & component are not identical.
    - (equality can be thrown off because decorator != decorated)
  - Negative: Many of little objects.
    - (Lots of small, similar-looking classes differentiated by how they are connected. hard to understand and debug.)

Decorator

- Implementation:
  - 1) Interface conformance. (decorator interface required)
  - 2) Abstract decorator not needed if only one decoration is required.
  - 2) Keep component classes lightweight. (too heavyweight can overwhelm decorators
  - 3) Changing a skin instead of changing the guts. (if component is heavy, consider strategy instead)
- Related to: Decorators are a kind of single-node Composite. Decorators can change the skin, Strategy pattern can change the guts.

Decorator (io ex)

```
new File(“446.csv”);
```

Decorator (io ex)

```
new FileInputStream(
new File("446.csv"));
```

Decorator (io ex)

```
new BufferedInputStream(
new FileInputStream(
 new File(“446.csv”)
));
```

Decorator (io ex)

```
new LineNumberInputStream(
 new BufferedInputStream(
  new FileInputStream(
  new File("446.csv")
)));
```

Composition vs Inheritance

- Has-a relationships are more flexible than is-a
- With composition:
  - Behaviour can be extended by dynamically
    - Provides natural extension points
  - Comprehension challenges (many small classes)
  - Delegation can impact performance
- With inheritance:
  - Subclasses ‘reuse’ superclass code
  - Changing parent types can cause large changes in client code
  - Supertype changes can impact all subtypes

Cloud precursors

- Grid Computing:
  - Combination of computing resources from multiple administrative domains applied to common tasks.
    - Usually used to create ‘super computers’ that can work on specific parallel computation tasks.
- Utility Computing:
  - Combining computation, storage, and services metered like utilities.

Cloud Computing

- "Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model promotes availability and is composed of five essential characteristics, three service models, and four deployment models." [NIST]

NIST Essential Characteristics

- On-demand self-service:
  - Consumers can provision computing capabilities without human interaction.
- Broad network access:
  - Capabilities are available over the network through standard mechanisms.
- Resource pooling:
  - Computing resources are pooled to serve multiple consumers.
  - Location independence. [perfomance/security]

NIST Essential Characteristics

- Rapid elasticity
  - Resources can be easily added and removed.
- Measured service [services and/or resources]
  - Metering of storage, processing, bandwidth, etc.

Benefits

- Agility [quickly respond to changes]
- Scalability [resources can be added, peak load]
- Cost [resources can be released; multi-tenancy (amortization)]
- Reliability [different sites, experts in control]
- Security [works both ways]

Technology

- Thick and thin clients
- Broadband
- Data centres
  - Large capacity
  - Globally distributed
- APIs
  - Administration
  - Development
  - Resource migration

Virtualization

- Virtualization [decoupling physical & computing resources]
  - Emulation (QEMU) [VM simulates partial HW]
  - Paravirtualization (Xen) [SW int to VM]
  - Full (VMWare) [complete sim of HW]
  - Network [abstract network e.g., VPNs]

Cloud Layers

- SaaS (e.g., Google Docs) [multi-tenancy, single release for all users]
  - Vendor-controlled remote applications.
  - Concerns: control, performance, security, privacy.
- PaaS (e.g., AppEngine)
  - Vendor-controlled environment.
  - Concerns: as for SaaS w/ limited technology choices.
- IaaS (e.g., Amazon EC2)
  - Vendor-provided resources; consumer provisions VM.
  - Concerns: more expertise needed to leverage flexibility.

Cloud Spectrum

less flexible
more constrained
less effort

more flexible
less constrained
more effort

Google docs

salesforce.com

TurboTax

Windows Azure

amazon web services

Eucalyptus

the rackspace cloud

Layers of Control

In-house Deployment	Hosted Deployment	IaaS Cloud	PaaS Cloud	SaaS Cloud
Data	Data	Data	Data	Data
APP	APP	APP	APP	APP
VM	VM	VM	Services	Services
Server	Server	Server	Server	Server
Storage	Storage	Storage	Storage	Storage
Network	Network	Network	Network	Network

Organization controlled | Organization & service provider share control | Service Provider controlled

[1] **Visualizing the Boundaries of Control in the Cloud. Dec 2009.**
http://kscottmorrison.com/2009/12/01/visualizing-the-boundaries-of-control-in-the-cloud/

Cloud Security NFPs

- Users want assurances of:
  - Confidentiality [keep unauthorized users out]
  - Integrity [data has not altered]
  - Authenticity [data provenance]
  - Anonymity [users are unidentifiable]
  - Privacy [user data is properly controlled]
- Data remanence is problematic:
  - How can you purge data from the cloud?

REST

- Representational state transfer (REST)
- Key constraints:
  - Client/server
  - Servers to not maintain session state
  - Clients must not depend on direct server access
  - Clients communicate using a uniform interface
    - e.g., URIs and self-descriptive payloads

REST Operations

- Four main operations: GET, POST, PUT, DELETE
- Operation can change functionality:
  - GET /resources/ —> list resources
  - PUT /resources/ —> replace resources
  - POST /resources/ —> append to resources
  - DELETE /resources/ —> delete resources
- URIs are often versioned:
  - /api/v2.0/list/
  - /api/v3.0/list/
- REST endpoints enable direct testing:
  - e.g., `curl --include https://api.github.com/users/rtholmes`