Proxy Pattern

Idaho State University | Computer Science

Isaac Griffith

CS 2263
Department of Informatics and Computer Science
Idaho State University

ROAR

Outcomes

After today's lecture you will be able to:

- Understand the use of the Proxy Design Pattern
- Use and implement the Proxy Pattern

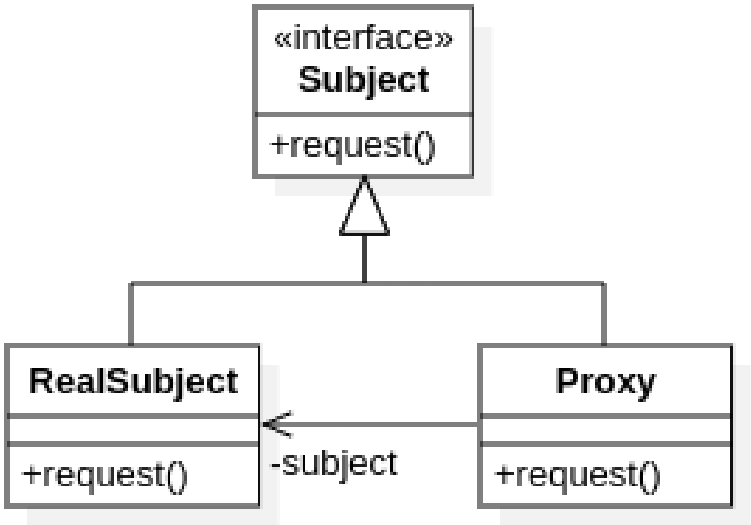
Inspiration

"... the designer of a new system must not only be the implementor and the first large-scale user; the designer should also write the first user manual. ... If I had not participated fully in all these activities, literally hundreds of improvements would never have been made, because I would never have thought of them or perceived why they were important." – Donald Knuth

Proxy Pattern: Definition

- The Proxy Pattern Provides a surrogate or placeholder for another object to control access to it.
  - Use the Proxy Pattern to create a representative object that controls access to another object, which may be remote, expensive to create, or in need of securing
- We'll see several examples of the Proxy pattern in use in this lecture, including the Remote Proxy, Virtual Proxy, and Protection Proxy

Proxy Pattern: Structure



Gumball Revisited

- To Illustrate the Remote Proxy variation of the Proxy Pattern (in which the Proxy and RealSubject objects are on different machines), we return to the Gumball Machine example of the previous lecture
  - Our client would like a way to monitor gumball machines remotely
    - to enable regular status reports and to allow them to do a better job of keeping the gumball machines full of gumballs

Step 1: Update Gumball Machine

```
public class GumballMachine {

  State souldOutState;
  State noQuarterState;
  State hasQuareterState;
  State soldState;
  State winnerState;

  State state = soldOutState;
  int count = 0;
  String location;

  public GumballMachine(String location, int count) {
    soldOutState = new SoldOutState(this);
    noQuarterState = new NoQuarterState(this);
    hasQuarterState = new HasQuarterState(this);
    soldState = new SoldState(this);
    winnerState = new WinnerSTate(this);
    this.count = count;
    if (count > 0) {
      state = noQuarterSTate;
    }
    this.location = location;
  }
}
```

Step 2: Create a Gumball Monitor

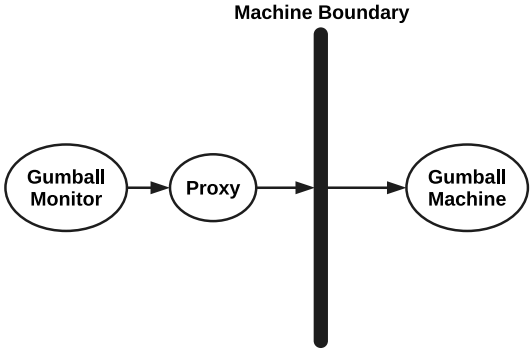
```
public class GumballMonitor {
  GumballMachine machine;

  public GumballMonitor(GumballMachine machine) {
    this.machine = machine;
  }

  public void report() {
    System.out.println("Gumball Machine: " + machine.getLocation());
    System.out.println("Current inventory: " + machine.getCount() + " gumballs");
    System.out.println("Current state: " + machine.getState());
  }
}
```

Simple! The monitor takes an instance of the Gumball machine class and can generate a status report: location, number of gumballs, and the machine's current state.

But something is wrong with design... what?

Going Remote

- The Gumball Monitor is coded to accept a pointer to a local Gumball Machine object
- But we need to monitor Gumball Machines that are not physically present
  - Or in computer speak: We need access to a “remote” Gumball Machine object, one that “lives” in a different JavaVM, or address space.
- To do this, we’ll use a technology built into Java, called RMI, short for Remote Method Invocation

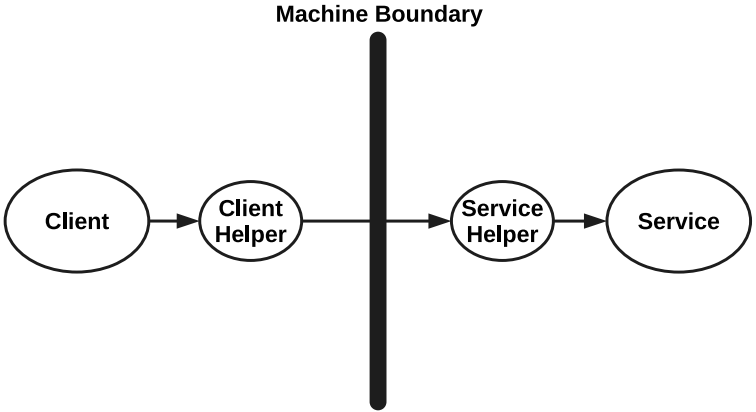


The Gumball Monitor talks to the Proxy object, thinking that its a Gumball Machine object. The Proxy communicates with the “real” Gumball Machine, and returns any results back to the monitor.

Approach

- Quick Introduction to RMI
- Change Gumball Machine so that it becomes a “remote service”
- Create a Proxy object that can talk to this “remote service” while looking like a local Gumball Machine to the Gumball Monitor class

Remote Method Invocation (I)



RMI creates “helper” objects that live on the client and service sides of a remote transaction. The client helper acts as a proxy for the remote service and sends method call information to the service helper. The service helper invokes the requested method on the service and returns the results.

Remote Method Invocation (II)

- RMI provides tools to automate the creation of the “helper” objects
  - The client helper is often called the “stub”
    - since none of the service methods are actually implemented
  - The service helper is often called the “skeleton”
    - since it often has methods that need to be filled in by the developer to hook it up to the actual service object
  - This architecture is common to many distributed computing frameworks
- These architectures do a lot to make the distributed nature of these method calls hidden from the client object… its not possible to entirely hide this process however, and the client does need to be prepared for exceptions that wouldn’t normally occur when invoking methods on a local object.

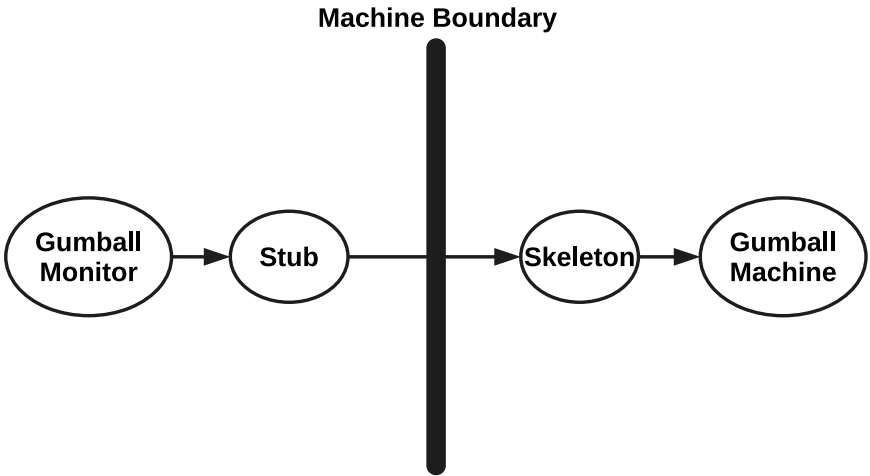
Remote Method Invocation (III)

1. Make a remote interface
   - The interface defines the methods that the remote service provides to the client; both the client helper (stub) and the service implement this interface
2. Make a remote implementation
   - The actual service object, in this case our Gumball Machine
3. Generate the stubs and skeletons
   - Using a tool that ships with the Java SDK: rmic
4. Start the RMI registry
   - So client objects can find service objects at run-time
5. Start the remote service
   - Before clients can make calls on the remote service, it needs to be running
6. Run the Client, which can now access the remote service

Remote Method Invocation (IV)

- Demonstrate on Simple Example
  - Simple server with single method
    - implements MyRemote interface and extends UnicastRemoteObject
  - Server and client run on the same machine
- Warning: I found RMI to be a big fickle
  - For instance, I got this example to work, but only after I set my class path to equal "." (i.e., the current directory) and then performed all steps of the example with that particular classpath (sigh)

Gumball RMI Architecture



Step 1: Create Remote Interface

```
import java.rmi.*;

public interface GumballMachineRemote extends Remote {
  int getCount() throws RemoteException;
  String getLocation() throws REmoteException;
  State getTate() throws RemoteException;
}
```

Simple translation of GumballMachine API into a Remote interface.

Note: RMI has a restriction that all return types and parameters need to be “serializable” which means that RMI needs to know how to “dismantle” an object of a type, send it across the network, and then assemble the information coming across the wire back into the original object.

See page 451 of the book to see how “State” is made serializable...

Step 2: Update Gumball Machine

```
import java.rmi.*;
import java.rmi.server.*;

public class GumballMachine
  extends UnicastRemoteObject implements GumballMachineRemote{

  // Fields are the same...

  public GumballMachine(String location, int count)
         throws RemoteException {
    // code is the same
  }
}
```

Step 3: Update Gumball Monitor

```
import java.rmi.*;

public class GumballMonitor {
  GumballMachineRemote machine;

  public GumballMonitor(GumballMachineRemote machine) {
    this.machine = machine;
  }

  public void report() {
    try {
      System.out.println("Gumball Machine: " + machine.getLocation());
      System.out.println("Current inventory: " + machine.getCount() + " gumballs");
      System.out.println("Current state: " + machine.getState());
    } catch (RemoteException e) {
      e.printStackTrace();
    }
  }
}
```

Step 4: Create "main" Program for Service

```
import java.rmi.*;

public class GumballMachineTestDrive {

  public static void main(String[] args) {
    GumballMachineRemote gumballMachine = null;
    int count;

    if (args.length < 2) {
      System.out.println("GumballMachineTestDrive <name> <inventory>");
      System.exit(1);
    }

    try {
      count = Integer.parseInt(args[1]);

      gumballMachine =
          new GumballMachine(args[0], count);
      Naming.rebind(args[0], gumballMachine);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Idaho State University Computer Science

Step 5: Create “main” Program for Client

```
import java.rmi.*;

public class GumballMonitorTestDrive {

  public static void main(String[] args) {
    String[] location = ("rmi://localhost/santafe",
                         "rmi://localhost/boulder",
                         "rmi://localhost/seattle");
    GumballMonitor[] monitor = new GumballMonitor[location.length];

    for (int i = 0; i < location.length; i++) {
      try {
        GumballMachineRemote machine =
            (GumballMachineRemote) Naming.lookup(location[i]);
        monitor[i] = new GumballMonitor(machine);
      } catch (Exception e) {
        e.printStackTrace();
      }
    }

    for(int i = 0; i < monitor.length; i++) {
      monitor[i].report();
    }
```

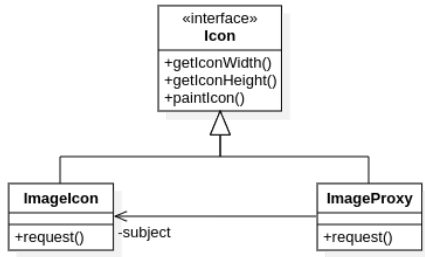
Step 6: Compile, Generate, Run

1. Set CLASSPATH equal to "." (i.e. the current directory)
2. `javac \*.java`
3. `rmic GumballMachine`
4. `rmiregistry &`
5. `java GumballMachineTestDrive boulder 50 &`
6. `java GumballMachineTestDrive seattle 250 &`
7. `java GumballMachineTestDrive santafe 150 &`
8. `java GumballMonitorTestDrive`

- **Demonstration**

Virtual Proxy

- Virtual Proxy is a variation of Proxy that provides control over when “expensive” objects are created
  - whereby expensive typically means
    - “takes a long time to create” or
    - “object takes up a lot of memory”
- The virtual proxy ensures that the object is only created when it is absolutely needed and “stands in” for the real object while the “expensive” creation process takes place
- Example: Loading Images Over a Network
  - Icon: Swing Interface
  - ImageIcon: Display Image
  - ImageProxy: Acts like ImageIcon while the image is loading…
- **Demonstration**

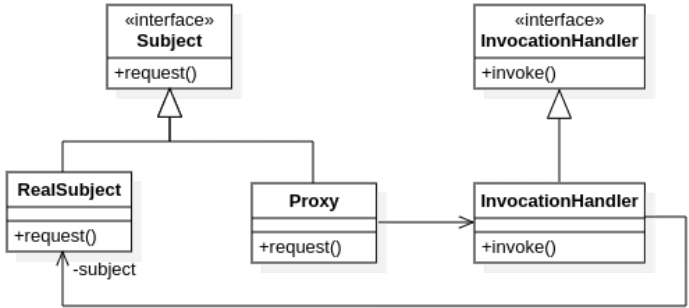


Protection Proxy

- A protection proxy is a variation on the Proxy pattern in which the proxy looks at the caller and the method being called and decides if it wants to forward the method call to the real subject
  - In this variation, the proxy is implementing a form of access control on top of the real subject; without this access control, any object that got a reference to the real subject could call any of its methods
- Example:
  - A person rating site
    - Model Class: Person
  - Problems:
    - Owners calling “setRating(10)” over and over to inflate their rating
    - Non-Owners calling various setter methods to capriciously (or maliciously) change the details of another person.

Java's Built-In Proxy Services

- In this example, the book decides to look at Java's built in support of the Proxy pattern: aka Dynamic Proxies
  - Structure of Java's Built-In Proxy support



Proxy is now an auto-generated class that is configured at creation time with an invocation handler. This handler determines what requests get forwarded to the RealSubject.

Specify Interface for Subject

```
public interface PersonBean {
  String getName();
  String getGender();
  String getInterests();
  String getStringRep();
  double getRating();

  void setName(String name);
  void setGender(String gender);
  void setInterests(String interests);
  void setRating(double rating);
}
```

Note: interface is slightly different from the book. It was changed to make the output of the test program more understandable!

Differences:

- Added `String getStringRep();`
- Changed "rating" methods to work with `double` not `int`

Implement RealSubject

```
public class PersonBeanImpl implements PersonBean {

  String name;
  String gender;
  String interests;
  double rating;
  int ratingCount = 0;

  public String getName() { return name; }

  public String getGender() { return gender; }

  public String getInterests() { return interests; }

  public double getRating() {
    if (ratingCount == 0) return 0;
    return (rating/ratingCount);
  }

  ...
}
```

- Standard “information holder” object with getter/setter routines
- HotOrNot methods keep track of number of ratings submitted and return an average value.
- `getStringRep()` produces a report of all current values of the object

Implement InvocationHandlers

```
import java.lang.reflect*;

public class OwnerInvocationHandler implements InvocationHandler {
  PersonBean person;

  public OwnerInvocationHandler(PersonBean person) {
    this.person = person;
  }

  public Object invoke(Object proxy, Method name, Object[] args)
          throws IllegalAccessException {
    try {
      if (method.getName().startsWith("get")) {
        return method.invoke(person, args);
      } else if (method.getName().equals("setRating")) {
        throw new IllegalAccessException();
      } else if (method.getName().startsWith("set")) {
        return method.invoke(person, args);
      }
    } catch (InvocationTargetException e) {
      e.printStackTrace();
    }
    return null;
```

Implement InvocationHandlers

```
import java.lang.reflect*;

public class NonOwnerInvocationHandler implements InvocationHandler {
  PersonBean person;

  public NonOwnerInvocationHandler(PersonBean person) {
    this.person = person;
  }

  public Object invoke(Object proxy, Method name, Object[] args)
          throws IllegalAccessException {
    try {
      if (method.getName().startsWith("get")) {
        return method.invoke(person, args);
      } else if (method.getName().equals("setRating")) {
        return method.invoke(person, args);
      } else if (method.getName().startsWith("set")) {
        throw new IllegalAccessException();
      }
    } catch (InvocationTargetException e) {
      e.printStackTrace();
    }
    return null;
```

Create Dynamic Proxies

```
PersonBean getOwnerProxy(PersonBean person) {
  return (PersonBean) Proxy.newProxyInstance(
      person.getClass().getClassLoader(),
      person.getClass().getInterfaces(),
      new OwnerInvocationHandler(person));
}

PersonBean getNonOwnerProxy(PersonBean person) {
  return (PersonBean) Proxy.newProxyInstance(
      person.getClass().getClassLoader(),
      person.getClass().getInterfaces(),
      new NonOwnerInvocationHandler(person));
}
```

Call Proxy's `newProxyInstance()` method; provide access to `RealSubject`'s class loader, its interfaces, and the desired invocation handler.

Client code then gets an instance of `RealSubject` and wraps it.

Demonstration

Wrapping Up

- Proxy is an extremely flexible pattern that allows you to control access to a particular object
  - We've seen examples of proxies that enable distributed access, control of "expensive" objects, and protection of an object's methods
- The book also mentions the use of proxies to
  - mimic a firewall in controlling access to network resources
  - keep track of the number of objects pointing at a subject
  - cache the results of expensive operations
  - protect an object from being accessed by multiple threads
  - and more

Are there any questions?