State Pattern
Proxy Pattern

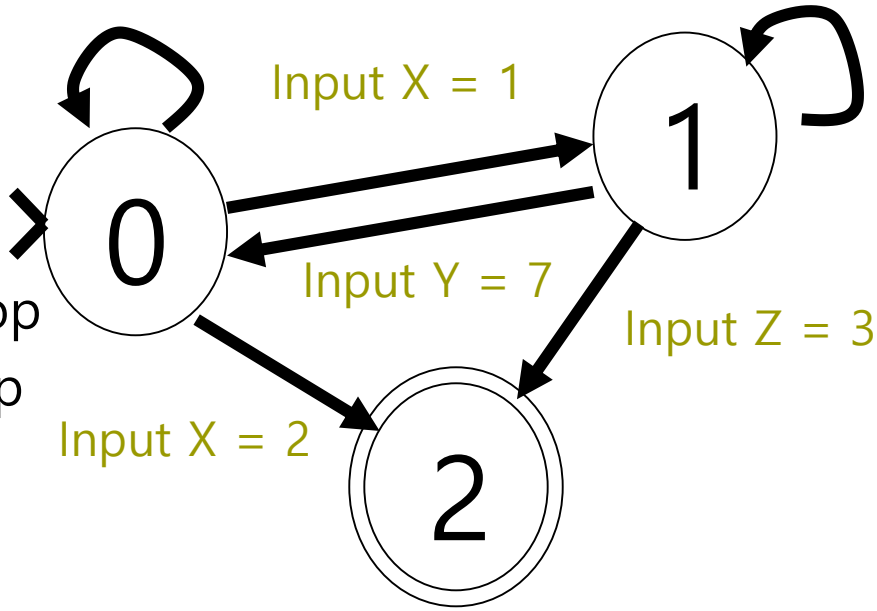
514770-1
Fall 2025
11/26/2025
Kyoung Shin Park
Computer Engineering
Dankook University

State Pattern

- "Allow an object to alter its behavior when its internal state changes. The object will appear to change its class."
- Also known as "**Objects for States**"
- An **object-oriented state machine**
- The State pattern is used when an object changes its behavior based on its internal state.
- In State pattern we create objects which represent various **states** and a **context** object whose behavior varies as its state object changes.
- The State pattern is closely related to the concept of a **Finite State Machine**.

Finite State Machine

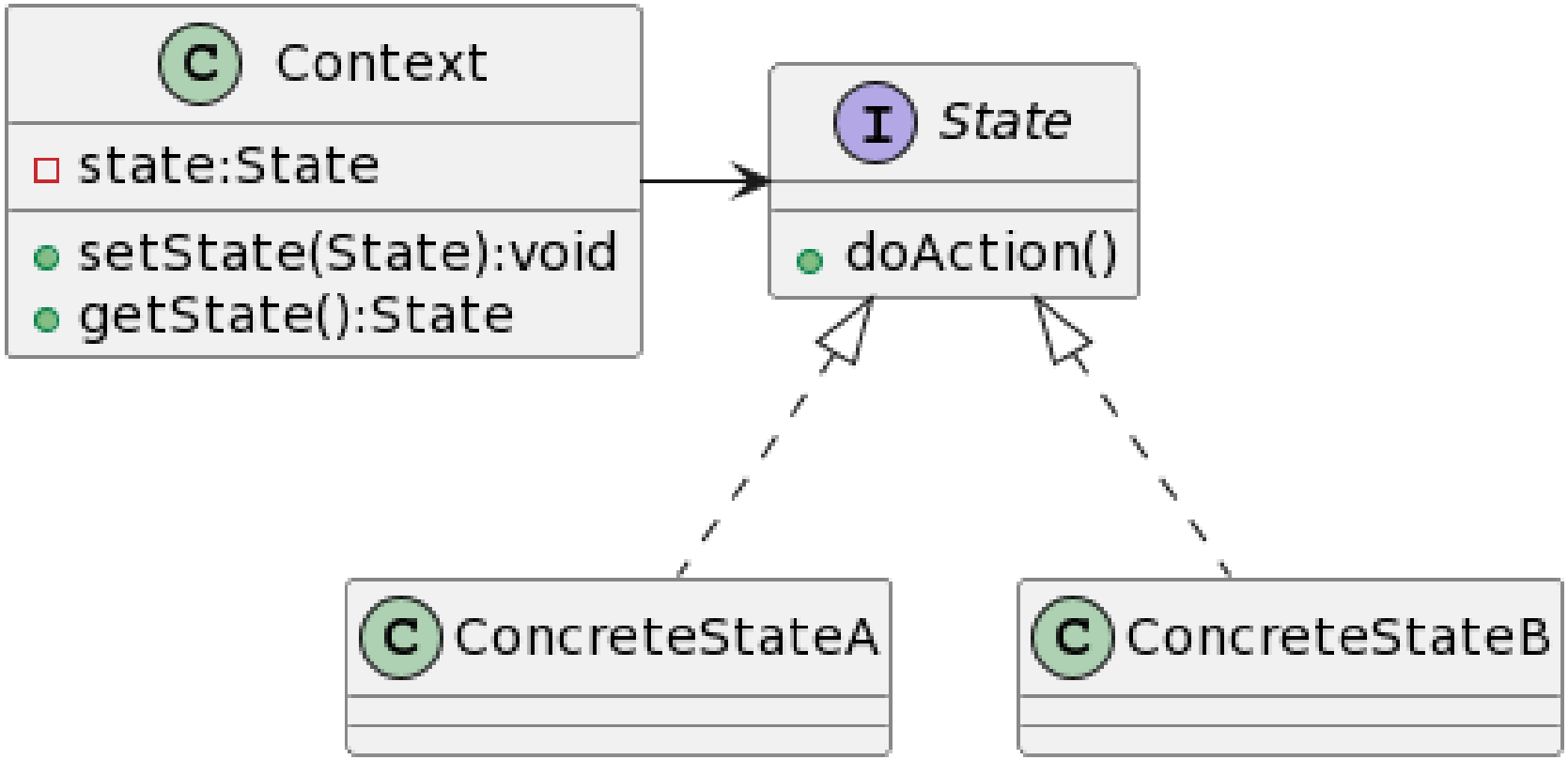
- **Finite State Machine (FSM)** or **Finite Automata**, or simply a **state machine**.
- An FSM is defined by a list of its states, its initial state, and the inputs that trigger each transition.
  - States
  - Inputs
  - Transitions
- For example,
  - Game character: walk, run, stop
  - Electronic goods: on, off, sleep
  - Turnstile: locked, unlocked



State Pattern

	Description
Pattern	State
Problem	State machines are usually implemented with lots of conditional operators (**if** or **switch**) that select the appropriate behavior depending on the current state of the object.
Solution	The State pattern allows the object for changing its behavior without changing its class.
Result	Single Responsibility Principle, Open/Closed Principle, Cleaner and more maintainable code

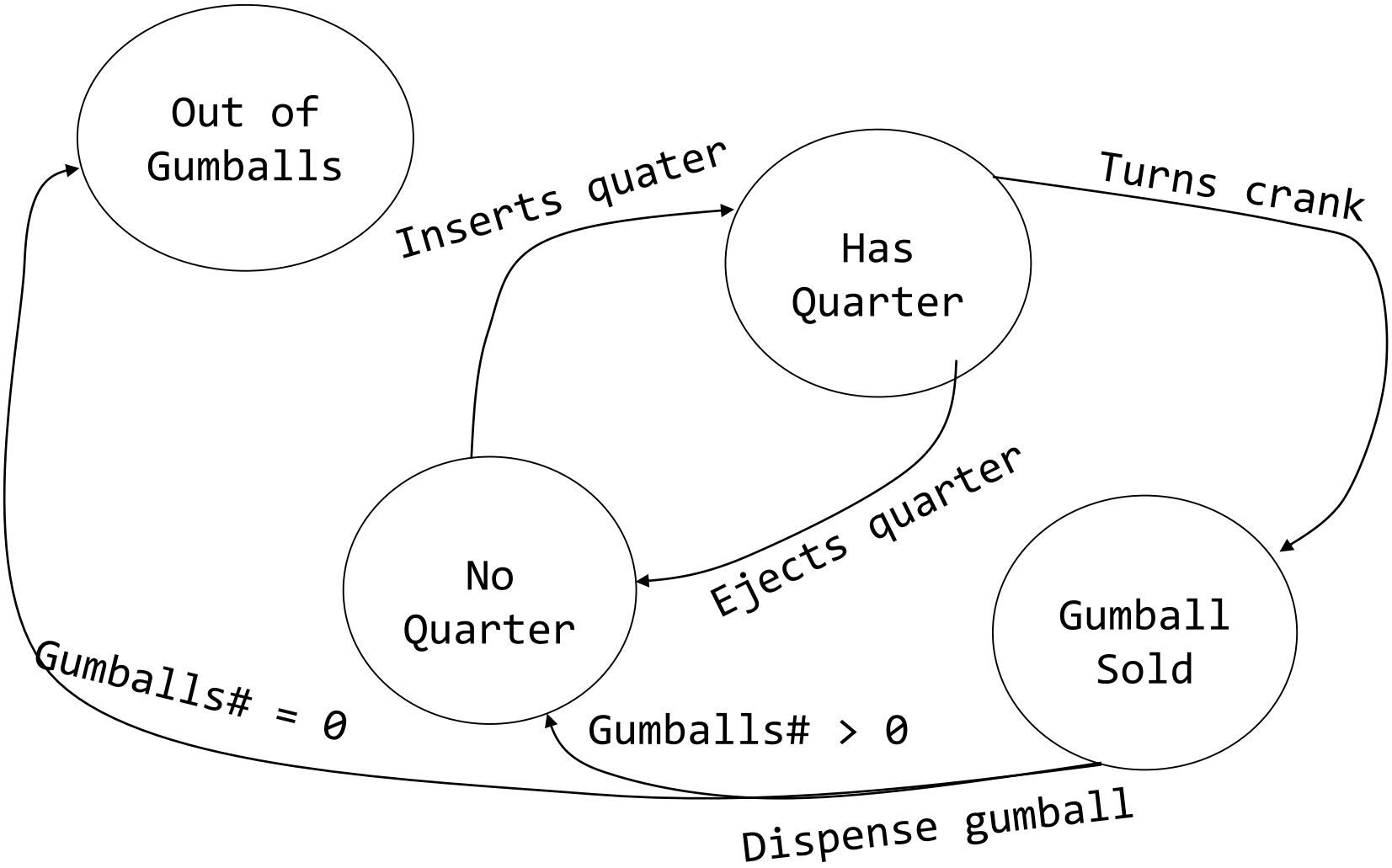
State Pattern



Define State Pattern

- Context
  - Context stores **a reference to one of the concrete state objects** and delegates to it all state-specific work. Context communicates with the state object via the state interface. Context exposes a **setter for passing it a new state object**.
- State
  - The State interface declares the **state-specific methods** (**what each concrete state should do**).
- ConcreteStateA, ConcreteStateB
  - They provide their own **implementations for state-specific methods**. To avoid duplication of similar code across multiple states, you may provide intermediate abstract classes that encapsulate some common behavior.

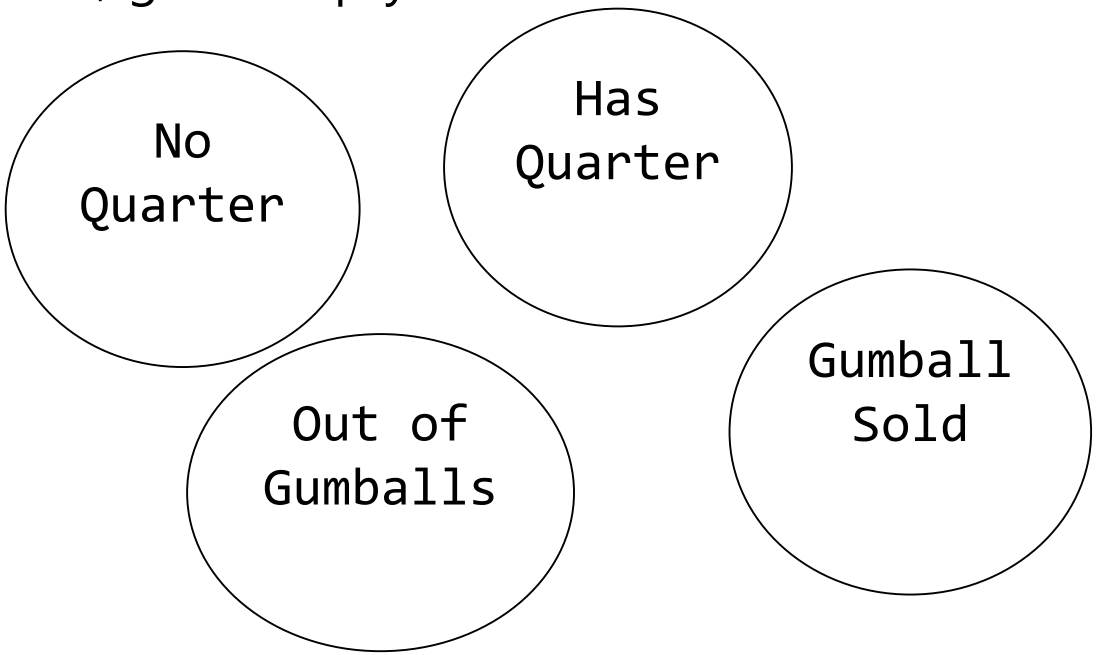
Gumball Machine (HFDP Ch. 10)



Finite State Machine

Gumball Machine (HFDP Ch. 10)

- Implementing **state machines**
  - First, gather up your states:



Gumball Machine (HFDP Ch. 10)

- Next, create an instance variable to hold the current state, and define values for each of the states:

```
final static int SOLD_OUT = 0;
final static int NO_QUARTER = 1;
final static int HAS_QUARTER = 2;
final static int SOLD = 3;

int state = SOLD_OUT;
```

- Now, gather up all the actions that can happen in the system.

Ejects quarter

Inserts quarter

Turns crank

dispense

Gumball Machine (HFDP Ch. 10)

- Now, create a class that acts as the state machine.

```
public class GumballMachine {
  final static int SOLD_OUT = 0;
  final static int NO_QUARTER = 1;
  final static int HAS_QUARTER = 2;
  final static int SOLD = 3;

  int state = SOLD_OUT;
  int count = 0;

  public GumballMachine(int count) {
    this.count = count;
    if (count > 0) {
      state = NO_QUARTER;
    }
  }
```

Gumball Machine (HFDP Ch. 10)

- Implement the actions as methods.

```
public void insertQuarter() {
  if (state == HAS_QUARTER) {
    System.out.println("You can’t insert another
quarter.");
  } else if (state == SOLD_OUT) {
    System.out.println("You can’t insert a quarter,
the machine is sold out.");
  } else if (state == SOLD) {
    System.out.println("Please wait, we’re already
giving you a gumball.");
  } else if (state = NO_QUARTER) {
    state = HAS_QUARTER;
    System.out.println("You inserted a quarter.");
  }
}
```

Gumball Machine (HFDP Ch. 10)

```
  public void ejectQuarter() {
    if (state == HAS_QUARTER) {
      System.out.println("Quarter returned.");
      state = NO_QUARTER;
    } else if (state == NO_QUARTER) {
      System.out.println("You haven't inserted a
quarter.");
    } else if (state == SOLD) {
      System.out.println("Sorry, you already turned
the crank.");
    } else if (state = SOLD_OUT) {
      System.out.println("You can't eject, you
haven't inserted a quarter yet. ");
    }
  }
```

Gumball Machine (HFDP Ch. 10)

```
  public void turnCrank() {
    if (state == SOLD) {
      System.out.println("Turing twice doesn't get
you another gumball!");
    } else if (state == NO_QUARTER) {
      System.out.println("You turned, but there's
no quarter.");
    } else if (state == SOLD_OUT) {
      System.out.println("You turned, but there are
no gumballs.");
    } else if (state = HAS_QUARTER) {
      System.out.println("You turned..");
      state = SOLD;
      dispense();
    }
  }
```

```
  public void dispense() {
    if (state == SOLD) {
      System.out.println("A Gumball comes rolling
out the slot.");
      count = count - 1;
      if (count == 0) {
        System.out.println("Oops, out of gumballs!
");
        state = SOLD_OUT;
      } else {
        state = NO_QUARTER;
      }
    } else if (state == NO_QUARTER) {
      System.out.println("You need to pay first.");
    } else if (state == SOLD_OUT) {
      System.out.println("No gumball dispensed.");
    } else if (state = HAS_QUARTER) {
      System.out.println("No gumball dispensed.");
    }
  }
  // other methods..
}
```

Gumball Machine (HFDP Ch. 10)

```
public class GumballMachineTestDrive {
  public static void main(String[] args) {
    GumballMachine gumballMachine = new
GumballMachine(5);
    System.out.println(gumballMachine);

    gumballMachine.insertQuarter();
    gumballMachine.turnCrank();

    System.out.println(gumballMachine);

    gumballMachine.insertQuarter();
    gumballMachine.turnCrank();
    gumballMachine.insertQuarter();
    gumballMachine.turnCrank();
    gumballMachine.ejectQuarter();

    System.out.println(gumballMachine);
```

Gumball Machine (HFDP Ch. 10)

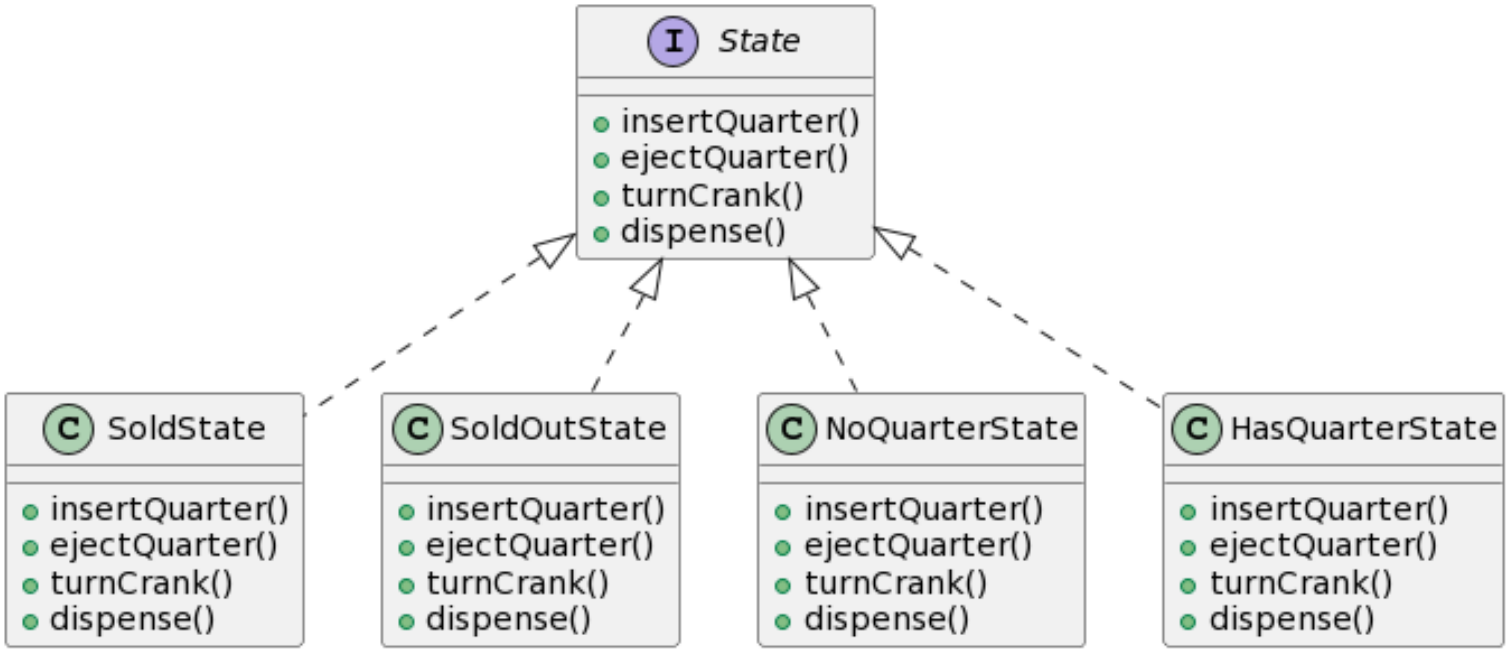
```
        gumballMachine.insertQuarter();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();
        gumballMachine.insertQuarter();
        gumballMachine.turnCrank();

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

Gumball Machine (HFDP Ch. 10)

- A change request
  - 10% of the time, when the crank is turned, the customer gets two gumballs instead of one.
    - **Be a WINNER!** One in ten get a free gumball.
    - First, you'd have to add a new **WINNER state**.
    - .. But then, you'd have to add a new **conditional** in **every single method** (insertQuater, ejectQuarter, dispense) to handle the WINNER state –> **that's a lot of code to modify**.
    - **turnCrank() will get especially messy**, because you'd have to add code to check to see whether you've got a WINNER and then switch to either the WINNER state or the SOLD state.

Gumball Machine (HFDP Ch. 10)

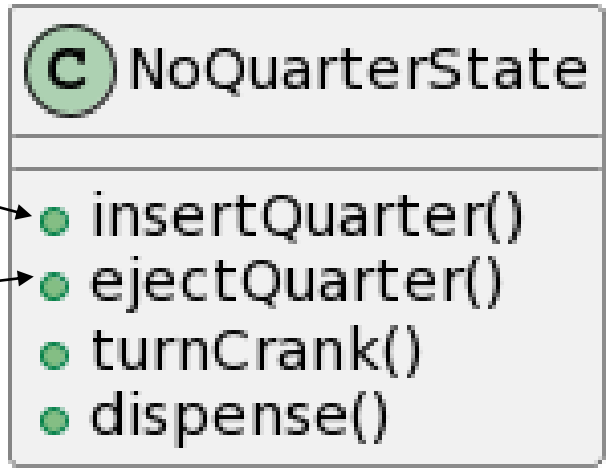
- The new design
  - First, define a **State interface** that contains a method for every action in the Gumball Machine.
  - Then, **implement a State class** for every state of the machine.
  - Finally, get rid of all of our conditional code and instead delegate to the state class to do the work for us.

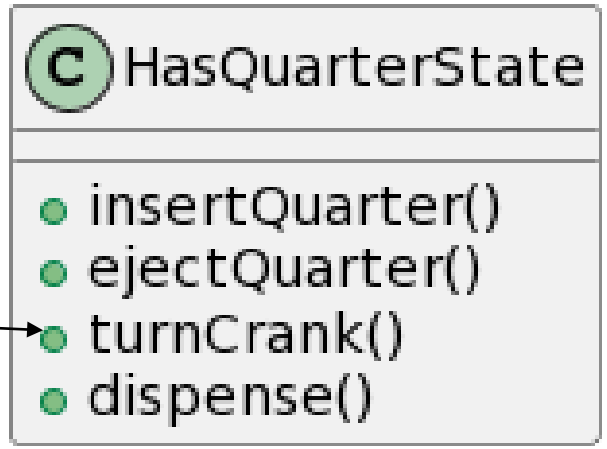


Gumball Machine (HFDP Ch. 10)

Go to HasQuarterState

Tell the customer,
"You haven't inserted
a quarter."

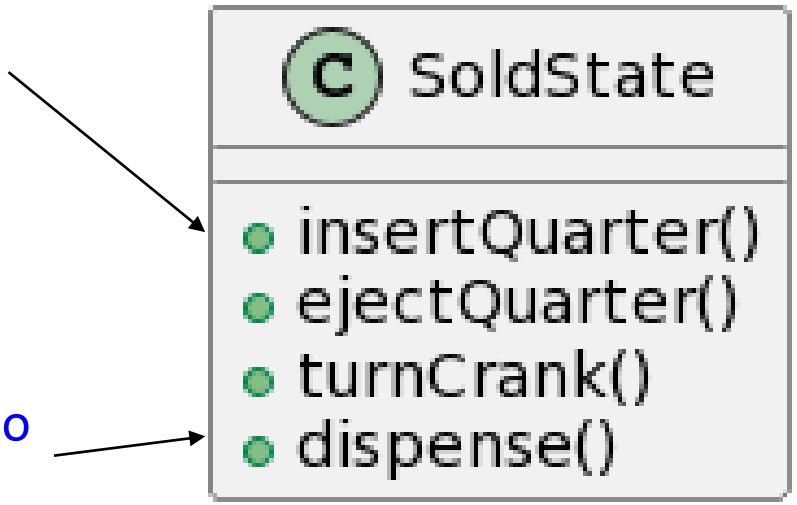




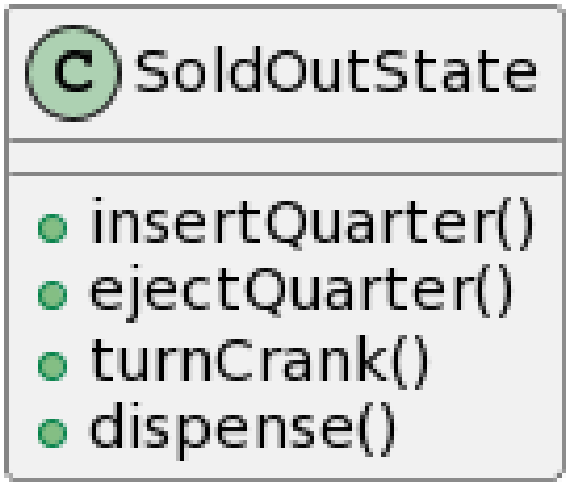
Go to SoldState

Gumball Machine (HFDP Ch. 10)

Tell the customer,
"Please wait, we're already
giving you a gumball."



Dispense one gumball. Check
number of gumballs; if > 0, go
to NoQuarterState, otherwise,
go to SoldOutState.



Tell the customer,
"There are no gumballs."

Gumball Machine (HFDP Ch. 10)

```
public class NoQuarterState implements State {
  GumballMachine gm;

  public NoQuarterState(GumballMachine gm) {
    this.gm = gm;
  }

  public void insertQuarter() {
    System.out.println("You inserted a quarter.");
    gm.setState(gm.getHasQuarterState());
  }

  public void ejectQuarter() {
    System.out.println("You haven't inserted a
quarter.");
  }
```

Gumball Machine (HFDP Ch. 10)

```
  public void turnCrank() {
    System.out.println("You turned, but there's no
quarter.");
  }

  public void dispense() {
    System.out.println("You need to pay first.");
  }
}
```

Gumball Machine (HFDP Ch. 10)

- Reworking the Gumball Machine
  - Switch the code from the state related instance variables using integers to **using state objects**.

```
public class GumballMachine {
  State soldOutState;
  State noQuarterState;
  State hasQuarterState;
  State soldState;

  State state = soldOutState;
  int count = 0;

  public GumballMachine(int numberGumballs) {
    soldOutState = new SoldOutState(this);
    noQuarterState = new NoQuarterState(this);
    hasQuarterState = new HasQuarterState(this);
    soldState = new SoldState(this);
```

Gumball Machine (HFDP Ch. 10)

```
    this.count = numberGumballs;
    if (numberGumballs > 0 ) {
      state = noQuarterState;
    }
  }
  public void insertQuarter() {
    state.insertQuarter();
  }
  public void ejectQuarter() {
    state.ejectQuarter();
  }
  public void turnCrank() {
    state.turnCrank();
    state.dispense();
  }
  void setState(State state) {
    this.state = state;
  }
```

Gumball Machine (HFDP Ch. 10)

```
  void releaseBall() {
    System.out.println("A gumball comes rolling out
the slot…");
    if (count != 0) {
      count = count – 1;
    }
  }

  // more methods including getters for each State
}
```

Gumball Machine (HFDP Ch. 10)

- Implementing HasQuarterState

```
public class HasQuarterState implements State {
  GumballMachine gm;

  public HasQuarterState(GumballMachine gm) {
    this.gm = gm;
  }

  public void insertQuarter() {
    System.out.println("You can’t insert another
quarter.");
  }

  public void ejectQuarter() {
    System.out.println("Quarter returned.");
    gm.setState(gm.getNoQuaterState());
  }
```

Gumball Machine (HFDP Ch. 10)

```
    public void turnCrank() {
        System.out.println("You turned..");
        gm.setState(gm.getSoldState());
    }

    public void dispense() {
        System.out.println("No gumball dispensed.");
    }
}
```

Gumball Machine (HFDP Ch. 10)

- Implementing SoldState

```
public class SoldState implements State {
  GumballMachine gm;

  public SoldState(GumballMachine gm) {
    this.gm = gm;
  }

  public void insertQuarter() {
    System.out.println("Please wait, we're already
giving you a gumball.");
  }

  public void ejectQuarter() {
    System.out.println("Sorry, you already turned
the crank.");
  }
```

Gumball Machine (HFDP Ch. 10)

```
  public void turnCrank() {
    System.out.println("Turing twice doesn't get you
another gumball!");
  }

  public void dispense() {
    gm.releaseBall();
    if (gm.getCount() > 0) {
      gm.setState(gm.getNoQuarterState());
    } else {
      System.out.println("Oops, out of gumballs!");
      gm.setState(gm.getSoldOutState());
    }
  }
}
```

Gumball Machine (HFDP Ch. 10)

- Implementing SoldOutState

```
public class SoldOutState implements State {
  GumballMachine gm;

  public SoldOutState(GumballMachine gm) {
    gm = gm;
  }

  public void insertQuarter() {
    System.out.println("You can't insert a quarter,
the machine is sold out.");
  }

  public void ejectQuarter() {
    System.out.println("You can't eject, you haven't
inserted a quarter yet.");
  }
```

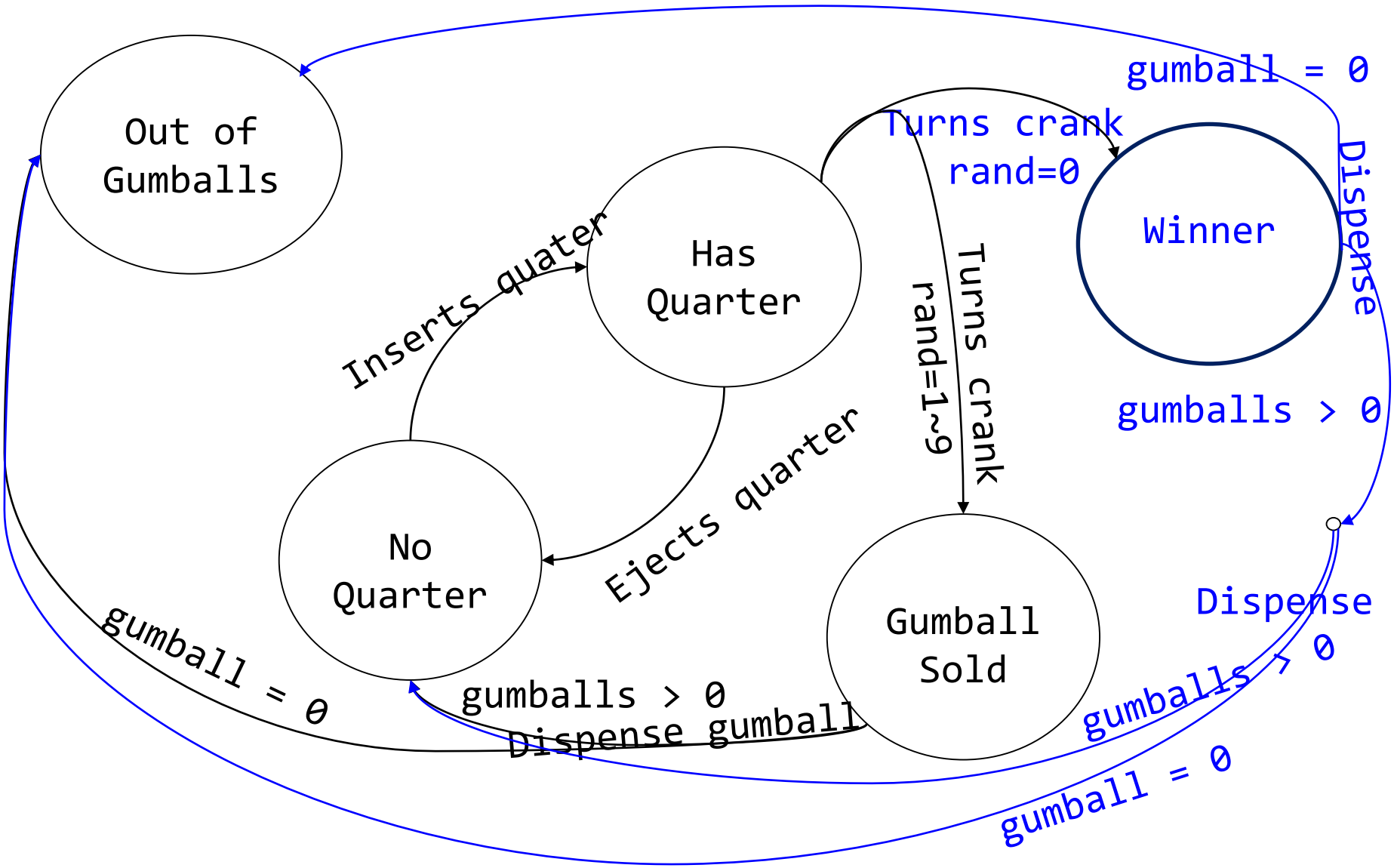
Gumball Machine (HFDP Ch. 10)

```
  public void turnCrank() {
    System.out.println("You turned, but there are no
gumballs!");
  }

  public void dispense() {
    System.out.println("No gumball dispensed.");
  }
}
```

Gumball Machine (HFDP Ch. 10)

- In State pattern, states are class.
- It gets rid of if-statements.
- State machine is open to extensions that add new state classes, such as Winner State.

Gumball Machine (HFDP Ch. 10)



Gumball Machine (HFDP Ch. 10)

- To make a gumball machine that gives you an extra gumball every ten times

```
public class WinnerState implements State {
  GumballMachine gm;

  public WinnerState(GumballMachine gm) {
    this.gm = gm;
  }

  public void insertQuarter() {
    System.out.println("Please wait, we're already
giving you a Gumball.");
  }

  public void ejectQuarter() {
    System.out.println("Please wait, we're already
giving you a Gumball.");
  }
```

```
  public void turnCrank() {
    System.out.println("Turning again doesn't get you
another Gumball!");
  }

  public void dispense() {
    gm.releaseBall();
    if (gm.getCount() == 0) {
      gm.setState(gm.getSoldOutState());
    } else {
      gm.releaseBall();
      System.out.println("YOU'RE A WINNER! You got
two gumballs for your quarter.");
      if (gm.getCount() > 0) {
        gm.setState(gm.getNoQuarterState());
      } else {
        System.out.println("Oops, out of gumballs!");
        gm.setState(gm.getSoldOutState());
      }
    }
  }
}
```

Gumball Machine (HFDP Ch. 10)

- Reworking HasQuarterState

```
public class HasQuarterState implements State {
   Random random = new Random(
                              System.currenTimeMillis());
   public void turnCrank() {
      System.out.println("You turned...");
      int winner = random.nextInt(10);
      if ((winner == 0)
                  && (gumballMachine.getCount() > 1)) {
         gumballMachine.setState(
            gumballMachine.getWinnerState());
      } else {
         gumballMachine.setState(
            gumballMachine.getSoldState());
      }
   }
}
```

Proxy Pattern

- "Provide a **surrogate** or **placeholder** for another object to control access to it."
- A proxy controls access to the original object, allowing you to perform something **either before or after the request gets through to the original object**.

Proxy Pattern

- Used for **access control**
  - "A class functioning as an interface to something else"
  - Processing on behalf of the object to be used
    - **A branch processes work on behalf of the head office of a bank**
  - **Calling a remote object on the server** (calling an object on a different JVM)
    - A **stub** on the client side is a **proxy.**
    - Acting as a **proxy for the stub**
      - Processing the client's request for a remote object on the server locally.
      - The client must have permission to request processing.

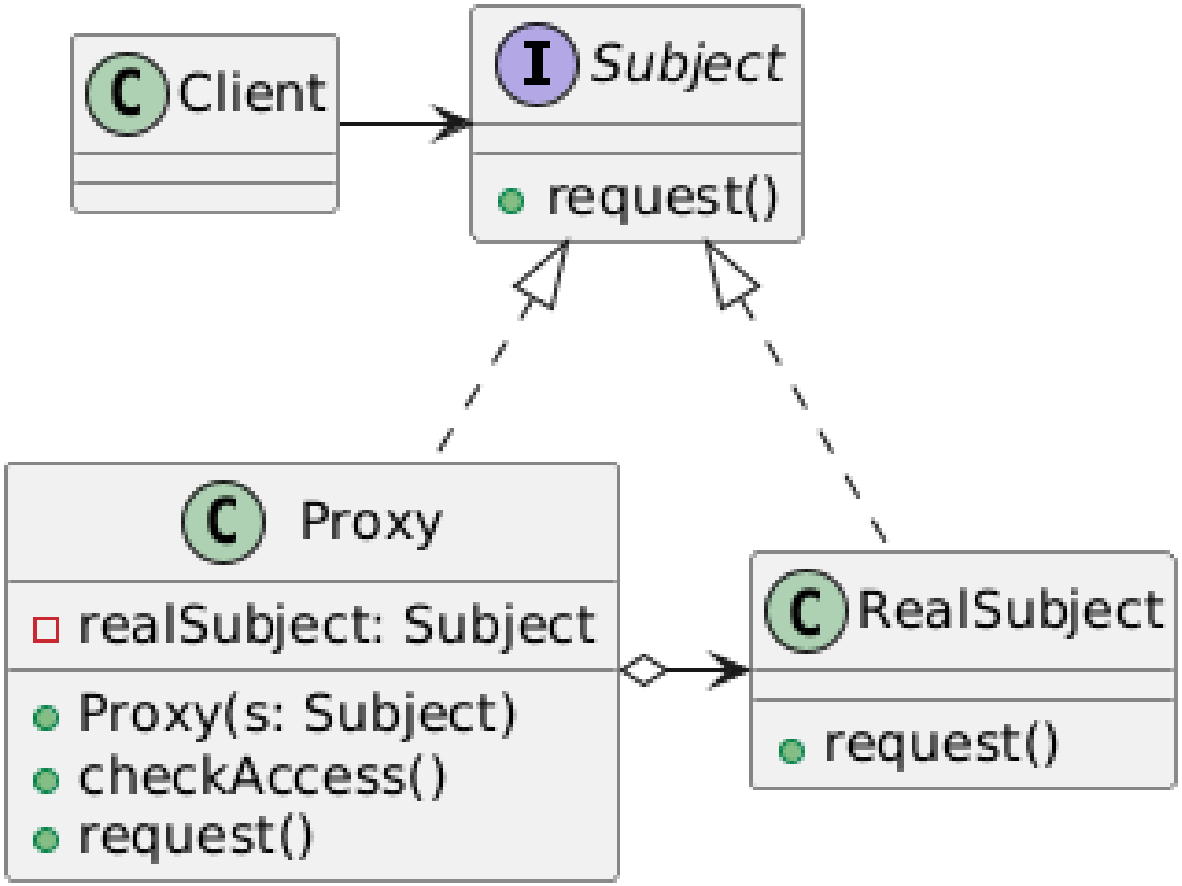
Proxy Pattern

- **Local execution of a remote service (Remote proxy)**
  - A proxy that is created on the local JVM on behalf of a remote object in a distributed network environment
  - The proxy receives a request and connects to an object in another remote JVM
- **Lazy initialization of a heavyweight object (Virtual proxy)**
  - Delay the object's initialization to a time when it's really needed
  - Image proxy (until loading, use icon)
- **Access control (Protection proxy)**
  - Invocation handler
- Logging requests (Logging proxy)
- Caching request results (Caching proxy)
- Smart reference

Proxy Pattern

	Description
Pattern	Proxy
Problem	The object you want to use is far away, busy, large, or difficult to use directly
Solution	Create a proxy object
Result	Decoupling of request and processing; Reducing the load on the object you actually want to use; Implementation becomes complex

Proxy Pattern



Define Proxy Pattern

- Subject
  - Subject interface declares the interface of service
- Proxy
  - The Proxy class has **a reference field that points to a real subject object**. After the proxy finishes its processing (e.g., lazy initialization, logging, access control, caching, etc.), it passes the request to the real subject object.
- RealSubject
  - They provide their own **implementations for state-specific methods**. To avoid duplication of similar code across multiple states, you may provide intermediate abstract classes that encapsulate some common behavior.

Coding the Monitor (HFDP Ch. 11)

- New requirements to Gumball Machine
  - Want to know the **inventory** and **current state** of gumball machine
  - Also want to include the **location** of gumball machine

```
public class GumballMachine {
    // other instance variables
    String location;

    public GumballMachine(String location, int count) {
        // other constructor code here
        this.location = location;
    }
    public String getLocation() {
        return location;
    }
    // other methods here …
}
```

Coding the Monitor (HFDP Ch. 11)

- GumballMonitor
  - **report** the location of the gumball machine, the inventory of gumballs, and the current machine state.

```
public class GumballMonitor {
    GumballMachine machine;
    public GumballMonitor(GumballMachine machine) {
        this.machine = machine;
    }
    public void report() {
        System.out.println("Location:" +
machine.getLocation());
     System.out.println("Inventory:" +
machine.getCount());
        System.out.println("Current State:" +
machine.getState());
    }
}
```

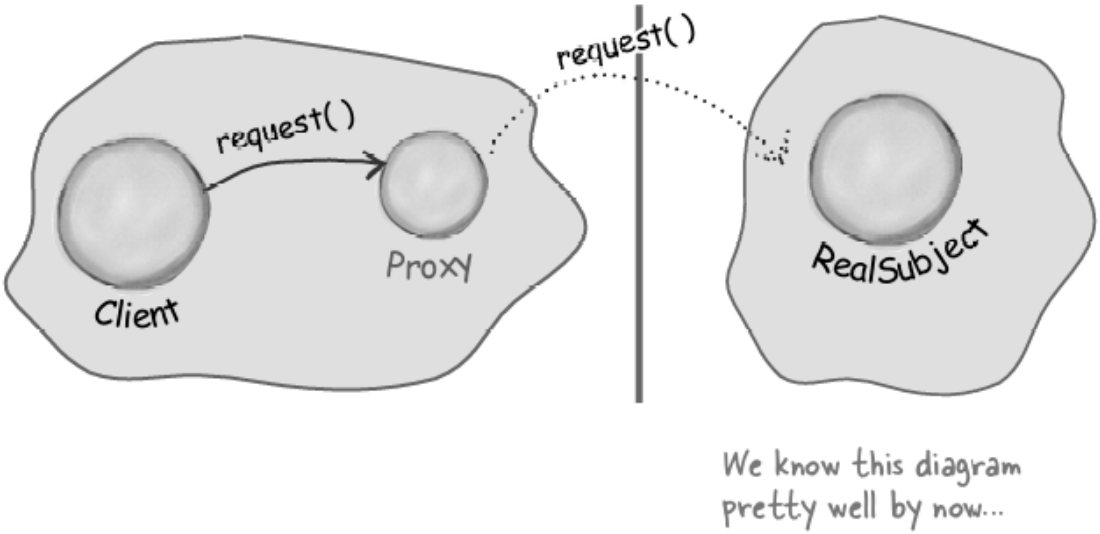
Testing the Monitor (HFDP Ch. 11)

```
public class GumballMachineTestDrive {
    public static void main(String[] args) {
        int count = 0;
        if (args.length < 2) {
            System.out.println("GumballMachine <name>
<inventory>");
            System.exit(1);
        }
        count = Integer.parseInt(argv[1]);
        GumballMachine machine = new
GumballMachine(args[0], count);
        GumballMonitor monitor = new
GumballMonitor(gumballMachine);
        // rest of test code here..
        monitor.report();
    }
}
```

Remote Proxy

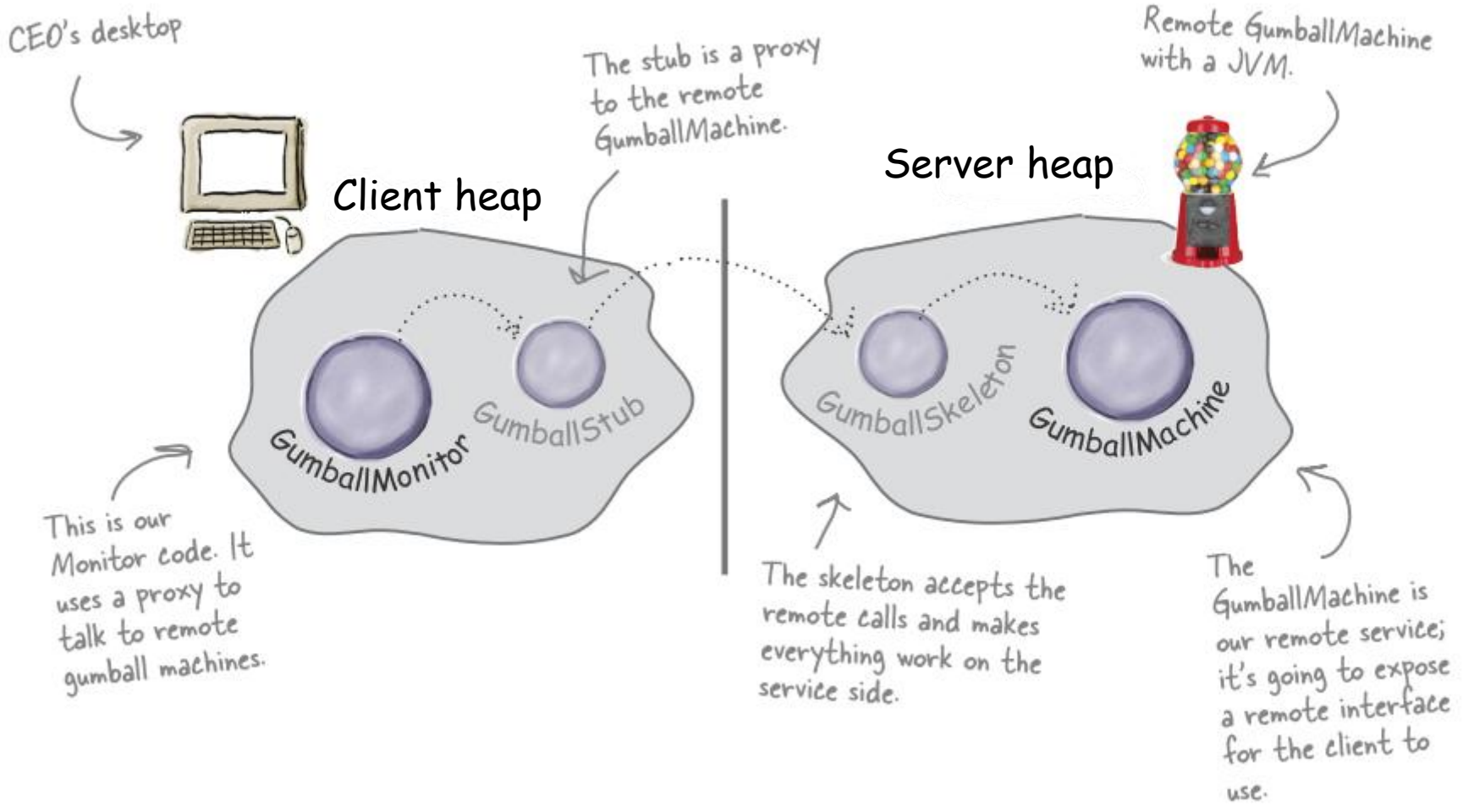
- Remote Proxy
  - Acts as a local representative for a remote object
  - Remote Object
    - An object in another JVM (an object running in a different address space)
  - Local Representative
    - When a method of a local representative is called, it forwards the method call to another remote object.

Remote Proxy



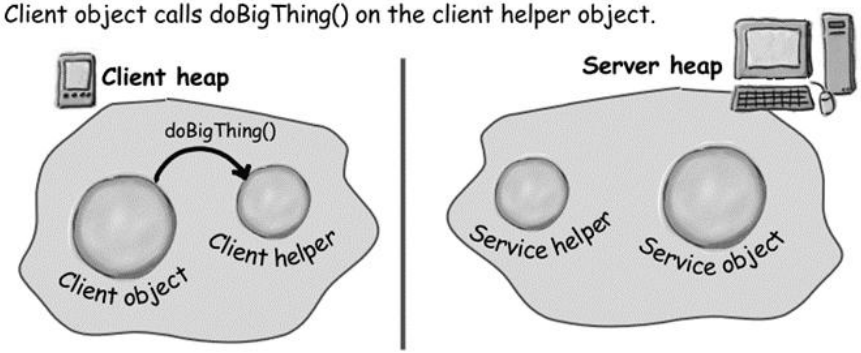
- The client object **acts as if it is calling a method on a remote object**, but in reality it is **calling a method on a "proxy" object that is stored on the local heap**.
- The proxy object is responsible for managing low-level network communication task.

RMI(Remote Method Invocation)

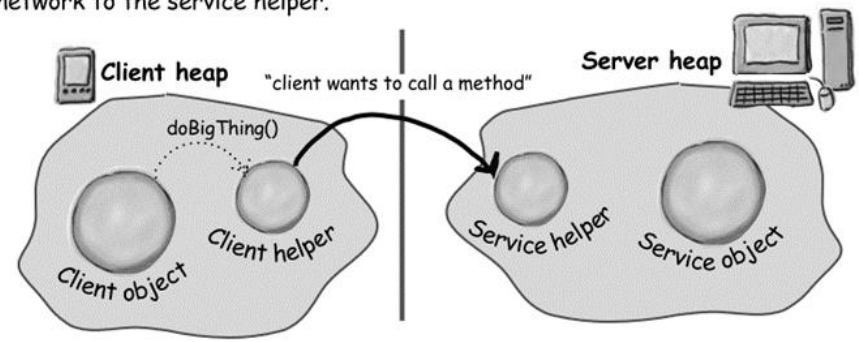


How the method call happens

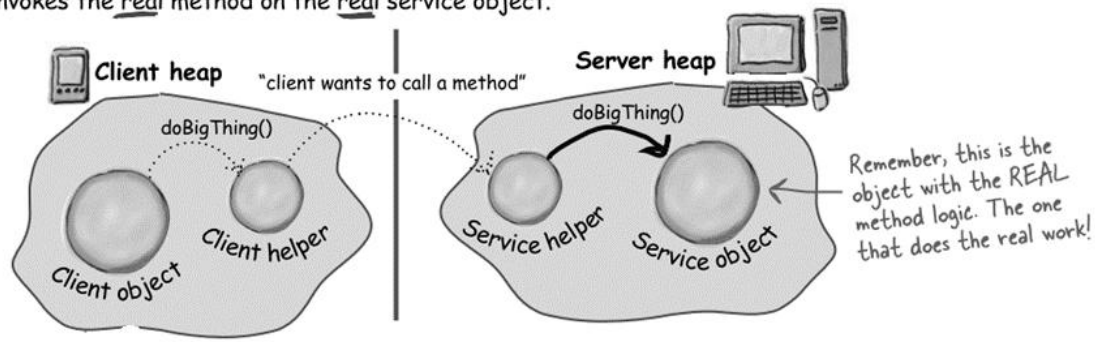
1. Client object calls doBigThing() on the client helper object.



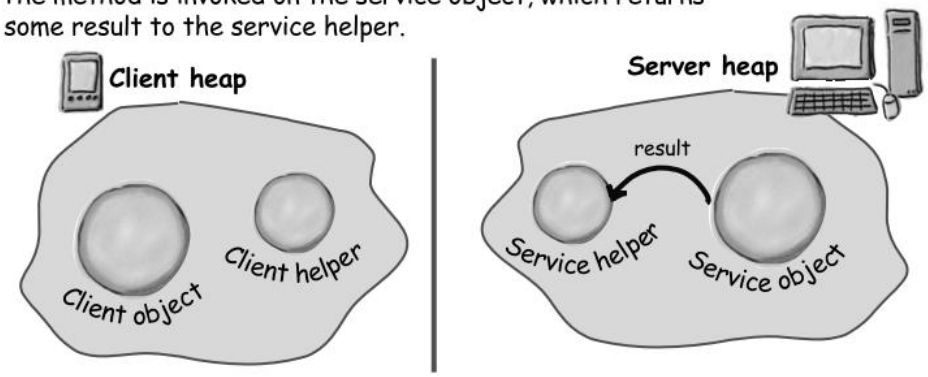
2. Client helper packages up information about the call (arguments, method name, etc.) and ships it over the network to the service helper.



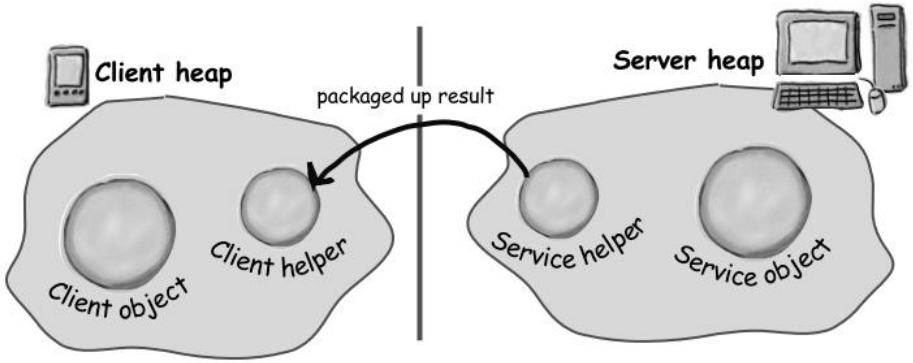
3. Service helper unpacks the information from the client helper, finds out which method to call (and on which object) and invokes the real method on the real service object.



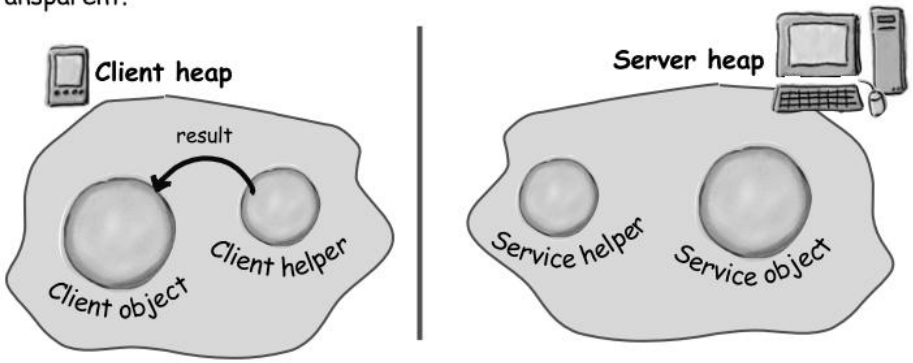
④ The method is invoked on the service object, which returns some result to the service helper.



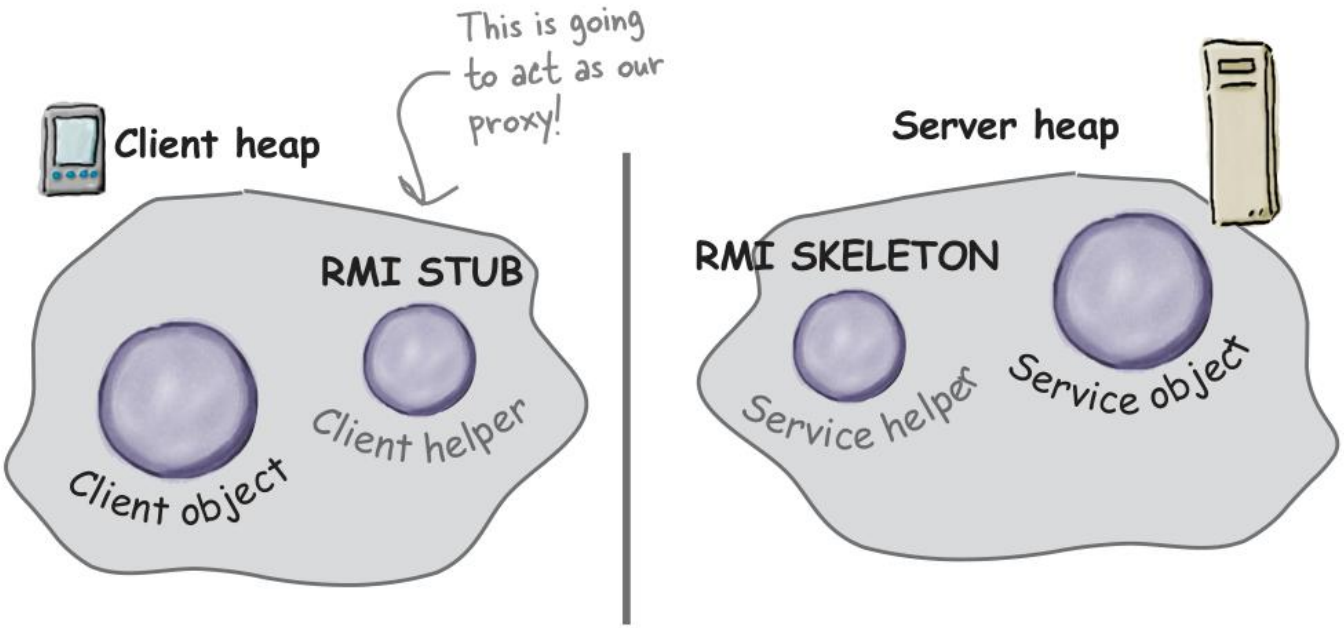
⑤ Service helper packages up information returned from the call and ships it back over the network to the client helper.



⑥ Client helper unpackages the returned values and returns them to the client object. To the client object, this was all transparent.



Java RMI


This is going to act as our proxy!
Client heap
RMI STUB
Client helper
Client object
Server heap
RMI SKELETON
Service helper
Service object

Remote Service

- Create a remote interface
  - Define the methods that the client calls remotely
- Create a service implementation class
  - Actually implement the functions that are called remotely
- Run the RMI registry
  - Phonebook
- Start the remote service
  - Create a service object and register it in the RMI registry

Remote Interface

- Create Remote Interface
  - Extends java.rmi.Remote

```
public interface MyRemote extends Remote { … }
```

- Declare all methods as throwing RemoteException

```
import java.rmi.*;

public interface MyRemote extends Remote {
    public String sayHello() throws RemoteExcepetion;
}
```

- The **arguments** and **return** values of remote methods must be declared as **primitive** or **Serializable** types.
  - If you pass a class you created yourself, implement the Serializable interface.

Service Implementation

- Implementing a remote interface

```
public class MyRemoteImpl implements MyRemote {
    public String sayHello() {
        return "Server says, 'Hey'";
    }
    // …
}
```

- **Create a stub object** using the UnicastRemoteObject.exportObject() function
- **Get the registry** by calling the LocateRegistry.getRegistry() function
- **Register the stub** by name using the rebind() or bind() function

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

public class MyRemoteImpl implements MyRemote {
  public String sayHello() {
    return "Server says, 'Hey'";
  }

  public static void main(String[] args) {
    try {
      MyRemote stub = (MyRemote)
UnicastRemoteObject.exportObject(new MyRemoteImpl(), 0);
      Registry registry = LocateRegistry.getRegistry();
      registry.rebind("RemoteHello", stub);
    }
    catch (Exception e) { e.printStackTrace(); }
  }
}
```

The Remaining Staps

- Run rmiregistry
  - Must be run from the directory where the service implementation class is located.

```
> rmiregistry
```

- Run service

```
> java MyRemoteImpl
```

Client Class

- **Get the registry** using LocateRegistry.getRegistry()
- Search for the service **name** in the registry and **get the stub**
- **Call the function using the stub**

- Run service

```
> java MyRemoteClient
```

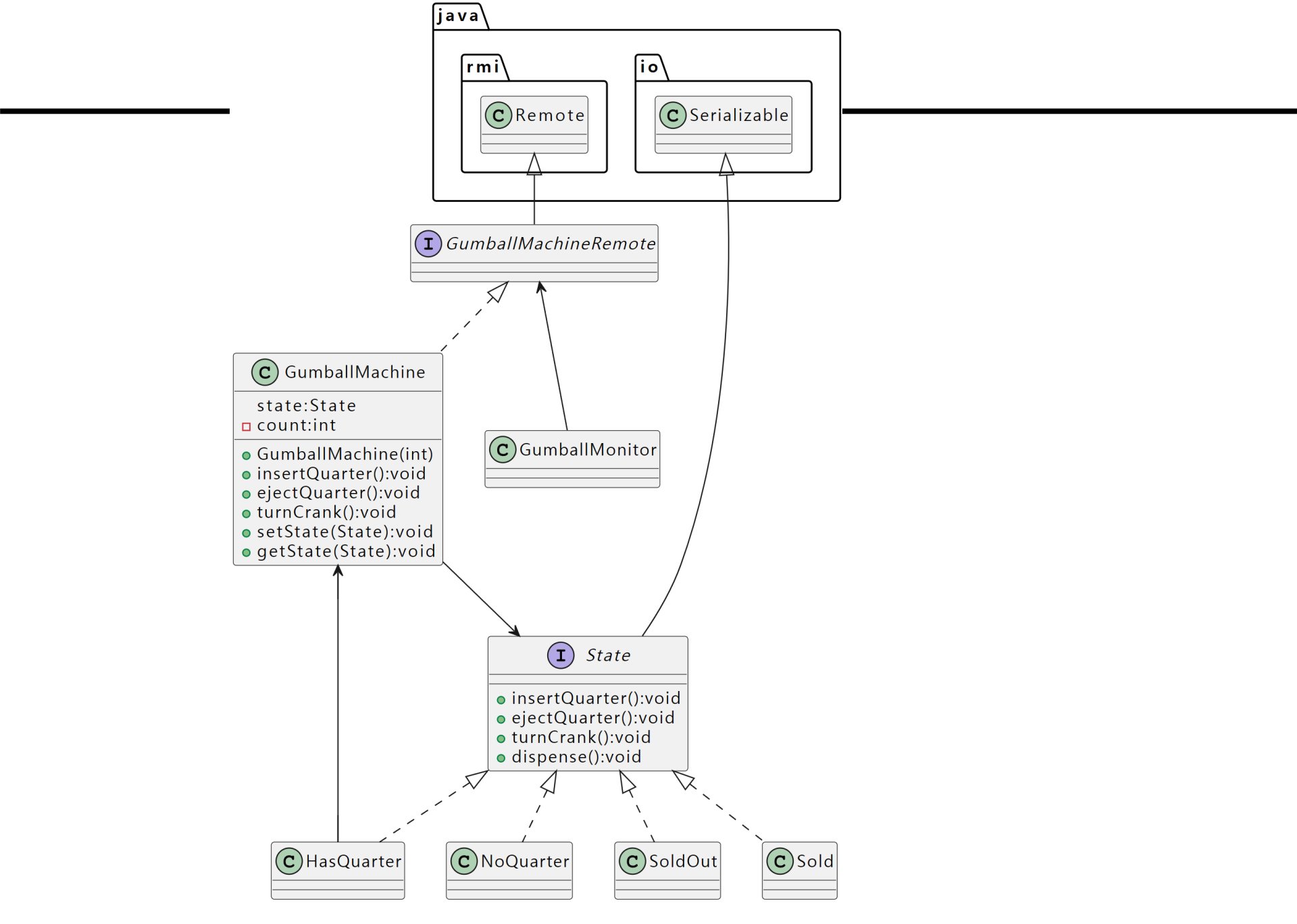
Client Class

```
import java.rmi.*;
import java.rmi.registry.*;

public class MyRemoteClient {
  public static void main(String[] args) {
    try {
      Registry registry = LocateRegistry.getRegistry();
      MyRemote stub = (MyRemote)
registry.lookup("RemoteHello");
      System.out.println(stub.sayHello());
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

GumballMachine as a Remote Service

- Write a **remote interface** for GumballMachine
- Make sure **all return types** in the interface are **serializable**
- Implement **the interface** in your class

```
import java.rmi.*;

public interface GumbalMachineRemote extends Remote {
    public int getCount() throws RemoteException;
    public String getLocation() throws RemoteException;
    public State getState() throws RemoteException;
}
```


java
rmi
io
Remote
Serializable
GumballMachineRemote
GumballMachine
state:State
count:int
GumballMachine(int)
insertQuarter():void
ejectQuarter():void
turnCrank():void
setState(State):void
getState(State):void
GumballMonitor
State
insertQuarter():void
ejectQuarter():void
turnCrank():void
dispense():void
HasQuarter
NoQuarter
SoldOut
Sold

GumballMachine as a Remote Service

- Modify State to be **Serializable**

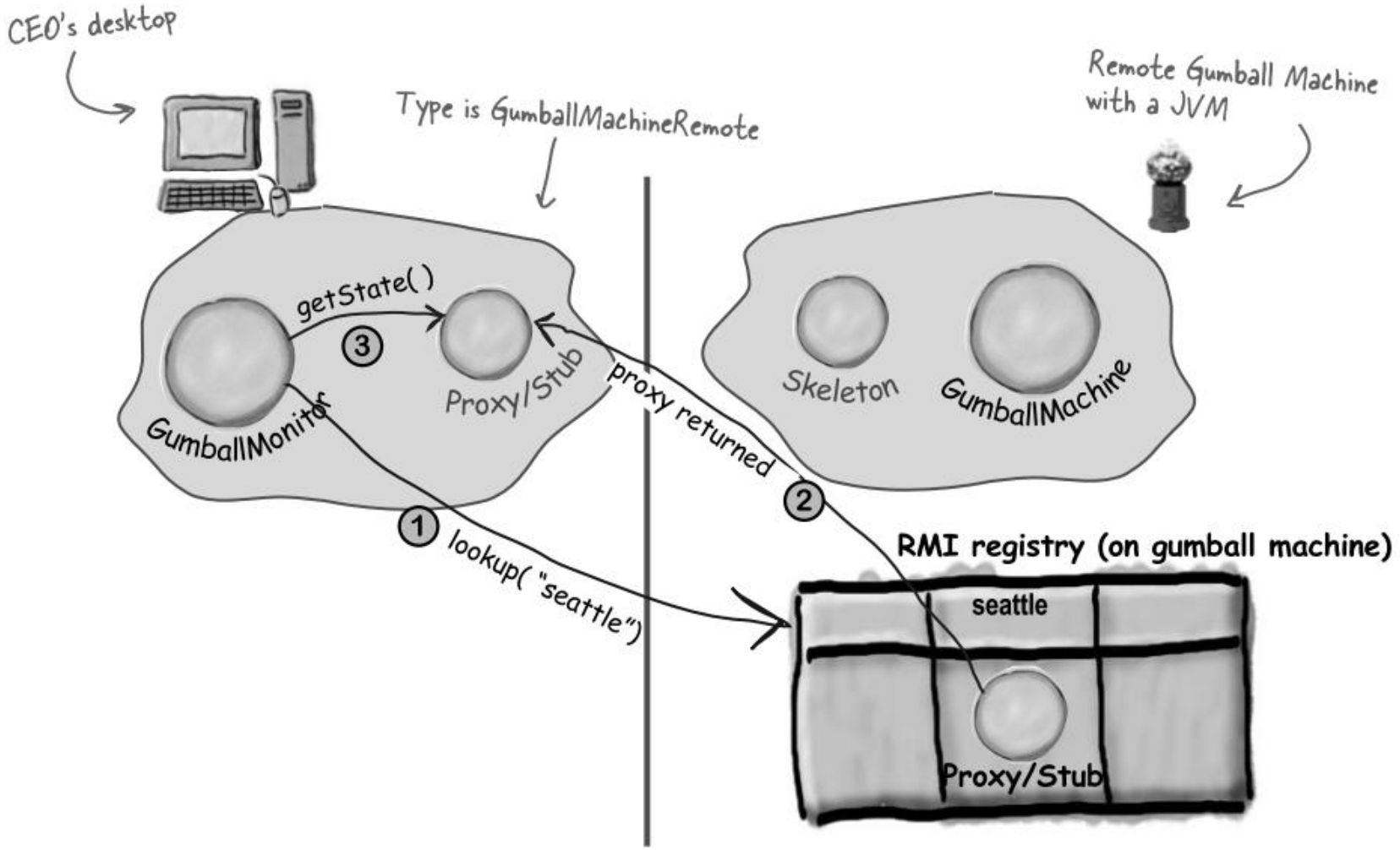
```
import java.io.*;  // Serializable

public interface State extends Serializable {
    public void insertQuarter();
    public void ejectQuarter();
    public void turnCrank();
    public void dispense();
}
```

GumballMachine as a Remote Service

- Modify the State Implementation class

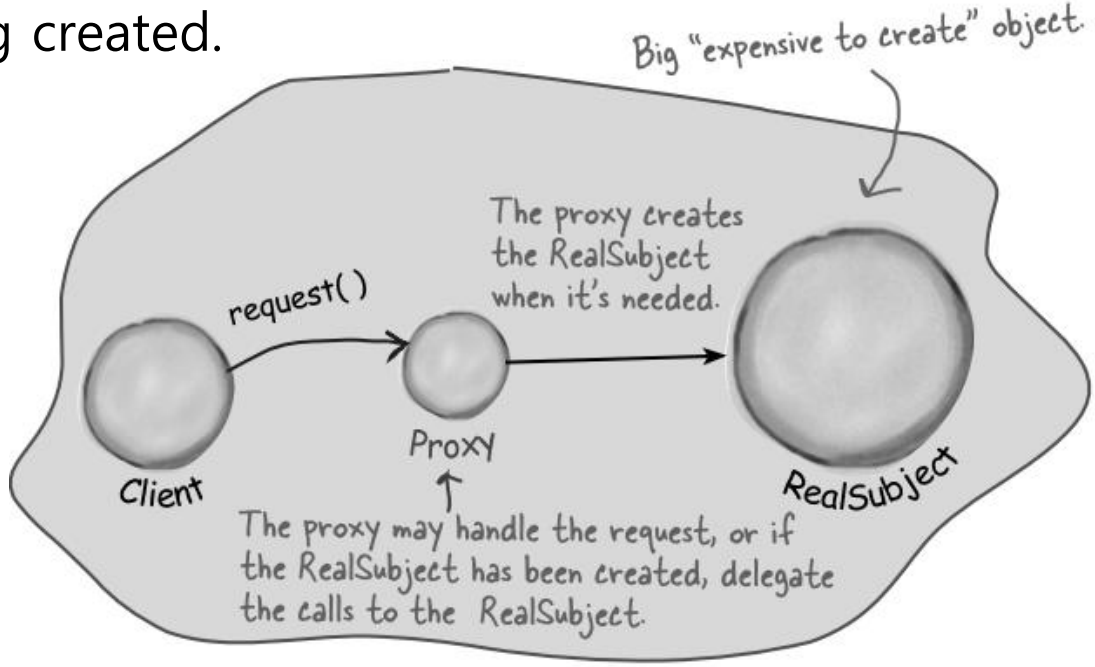
```
public class NoQuarterState implements State {
    private static final long serialVersionUID = 2L;
    transient GumballMachine gumballMachine;

     // rest of the code…
}
```

GumballMachine Remote Proxy Work



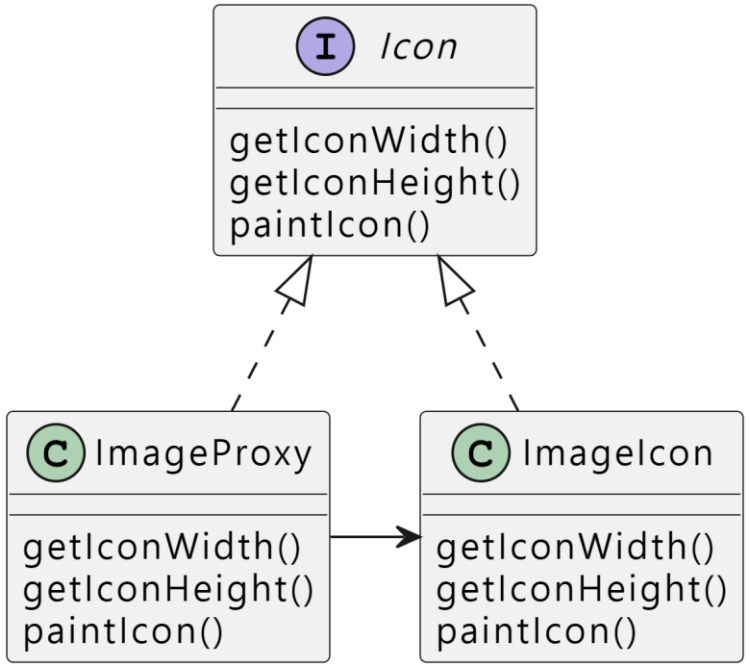
Virtual Proxy

- Virtual Proxy
  - Acts as a representative for an object that may be expensive to create
  - Also provides the ability to postpone the creation of objects until the real object is needed.
  - Also, acts as a proxy for the object before and while it is being created.



CD Cover Viewer

- Let's say you want to create a CD title menu and show images from the internet.
- The **virtual proxy** handles the task of fetching images in the background, and displays a message like "Loading CD cover, please wait..." until the images are fetched.



```
class ImageProxy implements Icon {
  volatile ImageIcon imageIcon;
  final URL imageURL;
  Thread rtThread;

  public ImageProxy(URL url) { imageURL = url; }
  public int getIconWidth() {
    if (imageIcon != null) {
      return imageIcon.getIconWidth();
    }
    else { return 800; }
  }

  public int getIconHeight() {
    if (imageIcon != null) {
      return imageIcon.getIconHeight();
    }
    else { return 800; }
  }
```

```
  public void paintIcon(final Component c, Graphics g,
int x, int y) {
    if (imageIcon != null) {
      imageicon.paintIcon(c, g, x, y);
    }
    else {
      g.drawstring("Loading CD cover, please wait…",
                   x + 300, y + 190);
      if (!retrieving) {
        retrieving = true;
        rtThread = new Thread(new Runnable() {
          public void run() {
            try {
              imageIcon = new ImageIcon(imageURL, "CD
Cover");
              c.repaint();
            }
            catch (Exception e) { e.printStackTrace(); }
        });
        rtThread.start();
} } } }
```

```
class ImageProxyTestDrive {
  ImageComponent imageComponent;
  public static void main(String[] args) {
    ImageProxyTestDrive t = new ImageProxyTestDrive();
  }

  public ImageProxyTestDrive() throws Exception {
    // frame
    // menu
    // …
    Icon icon = new ImageProxy(initialURL);
    imageComponent = new ImageComponent(icon);
    frame.getContentPane().add(imageComponent);
  }
}
```

Other Proxy

- Smart Reference Proxy
  - Provide additional behavior whenever the primacy object is referenced
  - Example: Counting the number of references to an object
- Caching Proxy
  - Temporarily stores the results of expensive operations
  - Can reduce computation time or network latency by allowing multiple clients to share the results