Changing the Guts

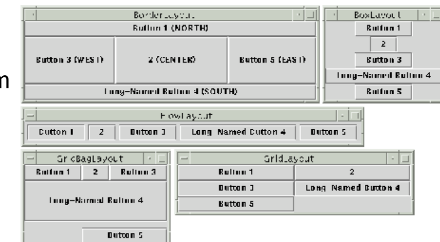
Changing the "Guts" of an Object ...

- Optimize
  - use an alternative algorithm to implement behavior (Strategy)
- Alter
  - change behavior when object's state changes (State)
- Control
  - "shield" the implementation from direct access (Proxy)
- Decouple
  - let abstraction and implementation vary independently (Bridge)

Strategy Pattern

Java Layout Managers

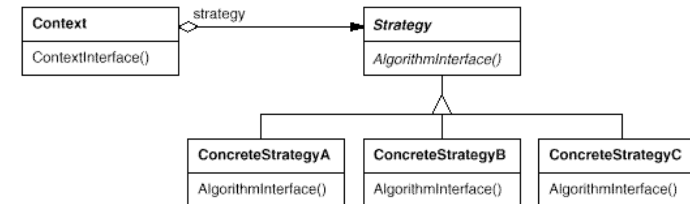
- GUI container classes in Java
  - frames, dialogs, applets (top-level)
  - panels (intermediate)
- Each container class has a layout manager
  - determine the size and position of components
  - 20 types of layouts
  - ~40 container-types
  - imagine to combine them freely by inheritance ;)
- Consider also sorting...
  - open-ended number of sorting criteria



Basic Aspects

- Intent
  - Define a family of algorithms, encapsulate each one, and make them interchangeable
  - Let the algorithm vary independently from clients that use it
- Applicability
  - You need different variants of an algorithm
  - An algorithm uses data that clients shouldn't know about
    - avoid exposing complex, algorithm-specific data structures
  - Many related classes differ only in their behavior
    - configure a class with a particular behavior

Structure



Strategy Applied on Example

```
import java.awt.*;
class  FlowExample  extends Frame  {

  public FlowExample( int  width, int height ) {
     setTitle( "Flow Example" );
     setSize( width, height );
     setLayout(new FlowLayout(FlowLayout.LEFT) );

     for  ( int label = 1; label < 10; label++ )
       add(new Button(String.valueOf( label )));
     show();
  }

  public  static  void  main( String  args[] ) {
     new  FlowExample( 175, 100 );
     new  FlowExample( 175, 100 );
  }
}
```


Participants

- Strategy
  - declares an interface common to all supported algorithms.
  - Context uses this interface to call the algorithm defined by a ConcreteStrategy
- ConcreteStrategy
  - implements the algorithm using the Strategy interface
- Context
  - configured with a ConcreteStrategy object
  - may define an interface that lets Strategy objects to access its data

Positive Consequences

- Families of related algorithms
  - usually provide different implementations of the same behavior
  - choice decided by time vs. space trade-offs
- Alternative to subclassing
  - see examples with layout managers (Strategy solution, here)
  - We still subclass the strategies...Why is this a big deal? ;)
- Eliminates conditional statements
  - many conditional statements → "invitation" to apply Strategy!

Negative Consequences

- Communication overhead between Strategy and Context
  - some ConcreteStrategies don't need information passed from Context
- Clients must be aware of different strategies
  - clients must understand the different strategies

```
SortedList studentRecords = new SortedList(new ShellSort());
```

- Increased number of objects
  - each Context uses its concrete strategy objects
  - can be reduced by keeping strategies stateless (share them)

Implementation

- How does data flow between Context and Strategies?
  - **Approach 1:** take data to the strategy
    - decoupled, but might be inefficient
  - **Approach 2:** pass Context itself and let strategies take data
    - Context must provide a more comprehensive access to its data
    - thus, more coupled
  - In Java strategy hierarchy might be inner classes
- Making Strategy object optional
  - provide Context with default behavior
    - if default used no need to create Strategy object
  - don't have to deal with Strategy unless you don't like the default behavior

When Behavior Depends on State...

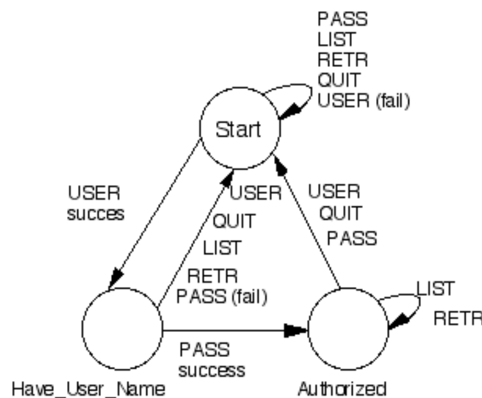
Example: SPOP

- SPOP = **S**imple **P**ost **O**ffice **P**rotocol
  - used to download emails from server
- SPOP supports the following commands:
  - USER <username>
  - PASS <password>
  - LIST
  - RETR <message number>
  - QUIT
- USER & PASS commands
  - USER with a username must come first
  - PASS with a password or QUIT must come after USER
  - If the username and password are valid, the user can use other commands

SPOP (contd.)

- LIST command
  - Arguments: a message-number (optional)
  - Returns: size of message in octets
    - if message number, returns the size of that message
    - otherwise return size of all mail messages in the mail-box
- RETR command
  - Arguments: a message number
  - Returns: the mail message indicated by that number
- QUIT command
  - Arguments: none
  - updates mailbox to reflect transactions taken during the transaction state, the logs user out
  - if session ends by any method except the QUIT command, the updates are not done

SPOP States



Our "Dear, Old" Switches in Action ;)

- ...as you see in the code (on next slide)
  - long functions
  - complex switches
  - same switches occur repeatedly in different functions
- Think about adding a new state to the protocol...
  - changes all the code
  - not Open-Closed
- Why?
  - object's behavior depends on its state

SPOP with Switches

```
class Spop {
   static final int HAVE_USER_NAME = 2;
   static final int START = 3;
   static final int AUTHORIZED = 4;
   private int state = START;

   String userName;
   String password;

   public void user( String userName ) {
    switch (state) {
      case START:
        this.userName = userName;
        state = HAVE_USER_NAME;
        break;
      case HAVE_USER_NAME:
      case AUTHORIZED:
        endLastSessionWithoutUpdate();
        goToStartState()
      }
   }
 //...
```

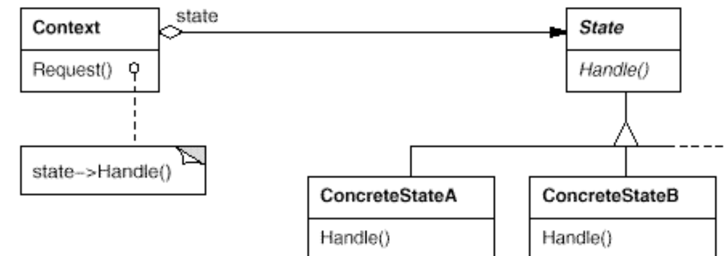
```
public void pass( String password ) {
      switch (state) {
         case START:
            giveWarningOfIllegalCommand();
        break;
         case HAVE_USER_NAME:
            this.password = password;
            if (validateUser())
               state = AUTHORIZED;
            else {
               sendErrorMessageOrWhatEver();
               userName = null;
               password = null;
               state = START;
            }
            break;
         case AUTHORIZED:
            endLastSessionWithoutUpdate();
            goToStartState()
         }
  }
  // ...
```


State Pattern

Basic Aspects of State Pattern

- Intent
  - allow an object to alter its behavior when its internal state changes
    - object will appear to change its class
- Applicability
  - object's behavior depends on its state
  - it must change behavior at run-time depending on that state
  - operations with multipart conditional statements depending on the object's state
    - state represented by one or more enumerated constants
    - several operations with the same (or similar) conditional structure

Structure



Participants

- Context
  - defines the interface of interest for clients
  - maintains an instance of ConcreteState subclass
- State
  - defines an interface for encapsulating the behavior associated with a particular state of the Context
- ConcreteState
  - each subclass implements a behavior associated with a state of the Context

Collaborations

- Context delegates state-specific requests to the State objects
  - the Context may pass itself to the State object
    - if the State needs to access it in order to accomplish the request
- State transitions are managed either by Context or by State
  - see discussion on the coming slides
- Clients interact exclusively with Context
  - but they might configure contexts with states
    - e.g initial state

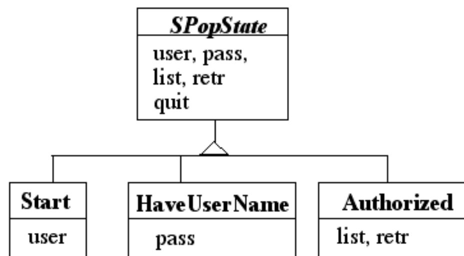
Consequences

- **Localizes** state-specific behavior and **partitions** behavior for different states
  - put all behavior associated with a state in a state-object
  - easy to add new states and transitions
    - context becomes O-C
  - Behavior spread among several State subclasses
    - number of classes increases, less compact than a single class
    - good if many states...
- Makes state transitions explicit
  - not only a change of an internal value
  - **states receive a full-object status!**
  - Protects Context from inconsistent internal states

Applying State to SPOP

```
class SPop {
     private SPopState state = new Start();
     public void user( String userName ) {
       state = state.user( userName );
     }
     public void pass( String password ) {
       state = state.pass( password );
     }
     public void list( int messageNumber ) {
       state = state.list( massageNumber );
     }
   //  .  . .
}
```

SPOP States



SPOP States

```
class SPopState {
   public SPopState user( String userName ) {
      return goToStartState(); }
   public SPopState pass( String password ) {
      return goToStartState(); }
   public SPopState list( int massageNumber ) {
      return goToStartState(); }
   public SPopState retr( int massageNumber ) {
      return goToStartState();
   }
   public SPopState quit() {
      return goToStartState(); }
   protected SPopState goToStartState()  {
      endLastSessionWithoutUpdate();
      return new StartState();
   }
}

class Start extends SPopState {
  public SPopState user(String uName )
 {
  return new HaveUserName(uName);
 }
}
```

```
class HaveUserName extends SPopState {
   String userName;

   public HaveUserName( String userName ) {
      this.userName = userName;
   }

   public SPopState pass( String password ) {
      if ( validateUser( userName, password )
         return new Authorized( userName );
      else
         return goToStartState();
   }
}
```


How much State in the State?

- Let's identify the roles...
  - **`SPop`** is the Context
  - **`SPopState`** is the abstract State
  - **`Start, HaveUserName`** are ConcreteStates
- All the state and real behavior is in SPopState and subclasses
  - this is an extreme example
- In general Context has data and methods
  - besides State & State methods
  - this data will not change states
- Only some aspects of the Context will alter its behavior

Who Defines the State transition?

- The Context if ...
  - ...states will be **reused** in different state machines with different transitions
  - ... the criteria for changing states are fixed
  - SPOP Example
- The States
  - More flexible to let State subclasses specify the next state
  - as we have seen before

Transition Triggered by Context

```
class Spop {
   private SPopState state = new Start();

   public void user( String userName ) {
      state.user( userName );
      state = new HaveUserName( userName );
   }

   public void pass( String password ) {
      if ( state.pass( password ) )
         state = new Authorized( );
      else
         state = new Start();
   }
}
```


Transition Triggered by States

```
class Spop {
   private SPopState state = new Start();

   public void user( String userName ) {
      state = state.user( userName );
   }

   public void pass( String password ) {
      state = state.pass( password );
   }

   public void list( int messageNumber ) {
      state = state.list( messageNumber );
   }

   // . . .
}
```

```
class Start extends SPopState {
 public SPopState user(String uName ) {
      return new HaveUserName( uName );
   }
}
```

```
class HaveUserName extends SPopState {
   String userName;

   public HaveUserName( String userName ) {
      this.userName = userName;
   }

   public SPopState pass( String password ) {
      if ( validateUser( userName, password )
         return new Authorized( userName );
      else return goToStartState();
   }
}
```


Sharing State Objects

- Multiple contexts can use the same state object
  - if the state object has no instance variables
- State object has no instance variables if the object ...
  - ... has **no need** for instance variables
  - ... stores its instance variables elsewhere
    - Variant 1 – Context stores them and passes them to states
    - Variant 2 – Context stores them and State gets them

Stateless State Objects Variant 1

```
class Spop {
   private SPopState state = new Start();

   String userName;
   String password;

   public void user( String newName ) {
      this.userName = newName;
      state.user( newName );
   }

   public void pass( String password ) {
      state.pass( userName , password);
   }
//...
```

Stateless State Objects Variant 2

```
class SPop {
   private SPopState state = new Start();
   String userName;
   String password;

   public String userName() { return userName; }
   public String password() { return password; }

   public void user( String newName ) {
      this.userName = newName ;
      state.user( this );
   }
 // . . .

class HaveUserName extends SPopState {
   public SPopState pass( SPop mailServer ) {
      String useName = mailServer.userName();
      etc.
   }
}
```


State versus Strategy

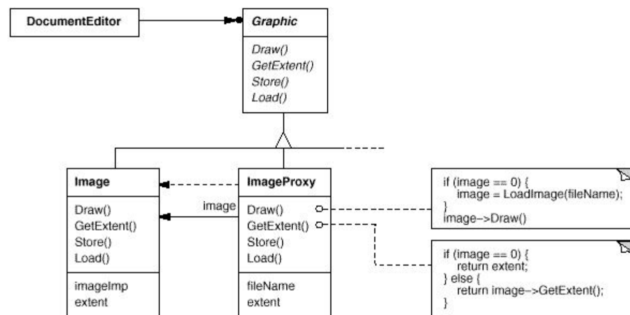
- Rate of Change
  - Strategy
    - Context object usually contains one of several possible **ConcreteStrategy** objects
  - State
    - Context object often changes its **ConcreteState** object over its lifetime
- Visibility of Change
  - Strategy
    - All **ConcreteStrategy** do the same thing, but differently
    - Clients do not see any difference in behavior in the Context
  - State
    - **ConcreteState** acts differently
    - Clients see different behavior in the Context

Proxy Pattern

Loading "Heavy" Objects

- Document Editor that can embed multimedia objects
  - MM objects are expensive to create ⇒ opening of document slow
  - avoid creating expensive objects
    - they are not all necessary as they are not all visible at the same time
- Creating each expensive object on demand !
  - i.e. when image has to be displayed
- What should we put instead?
  - hide the fact that we are "lazy"!
  - don't complicate the document editor!

Idea: Use a Placeholder!



- create only when needed for drawing
- keeps information about the dimensions (extent)

Basic Aspects

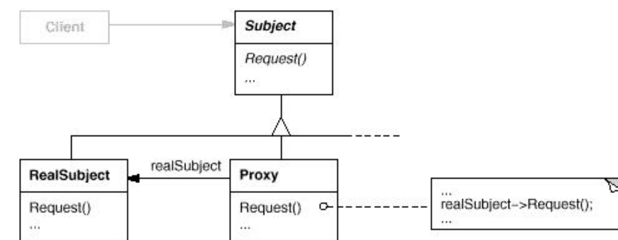
proxy (n. pl prox-ies) The agency for a person who acts as a substitute for another person, authority to act for another

- Intent
  - provide a surrogate or placeholder for another object to control access to it
- Applicability: whenever there is a need for a more flexible or sophisticated reference to an object than a simple pointer
  - remote proxy ... if real object is "far away"
  - virtual proxy ... if real object is "expensive"
  - protection proxy ... if real object is "vulnerable"
  - enhancement proxies (smart pointers)
    - prevent accidental delete of objects (counts references)

Further Reasons to Use Proxies

- Virtual Proxy
  - Creates/accesses expensive objects on demand
    - may wish to delay creating an expensive object until it is really accessed
  - It may be too expensive to keep entire state of the object in memory at one time
- Protection Proxy
  - Provides different levels of access to original object
- Cache Proxy (Server Proxy)
  - Multiple local clients can share results from expensive operations
    - remote accesses or long computations

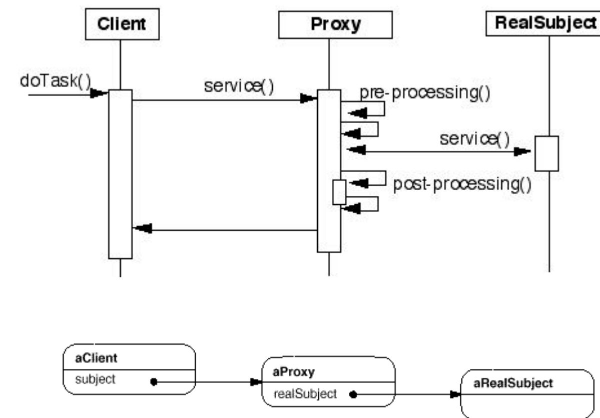
Structure



Participants

- Proxy
  - maintains a reference that lets the proxy access the real subject.
  - provides an interface identical to Subject's
    - so that proxy can be substituted for the real subject
  - controls access to the real subject
    - may be responsible for creating or deleting it
- Subject
  - defines the common interface for RealSubject and Proxy
- RealSubject
  - defines the real object that the proxy holds place for

Collaborations



Synchronization Proxy

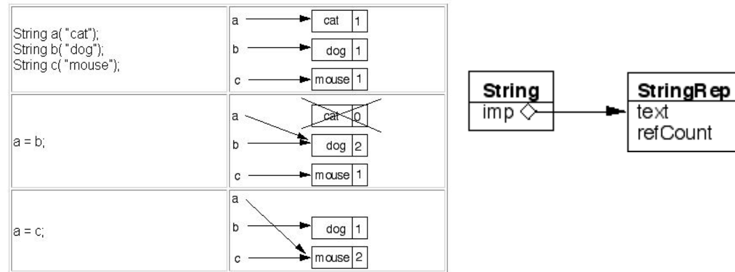
- Synchronize multiple accesses to real subject

```
public class Table {
  public Object elementAt(int row, int column){ blah }
  public void setElementAt(Object element,int row,int col)
  { blah} }

public class RowLockTable {
   Table realTable;
   Integer[] locks;

   public RowLockTable( Table toLock) {
      realTable = toLock;
      locks = new String[ toLock.numberOfRows() ];
      for (int row = 0; row< toLock.numberOfRows(); row++ )
         locks[row] = new Integer(row); }

   public Object elementAt( int row, int column ) {
      synchronized (locks[row]) {
         return realTable.elementAt( row, column);
      } }
   public void setElementAt(Object element,int row,int col)
   {
      synchronized ( locks[row] )  {
        return realTable.setElementAt(element, row, col);
      }
   } }
```


Consequences

- Proxies introduce a level of indirection
  - used differently depending on the kind of proxy:
    - hide different address space (remote p.)
    - creation on demand (virtual p.)
    - allow additional housekeeping activities (protection, smart pointers)
- Copy-On-Write [J.Coplien92 - The "Handle Class" Idiom]
  - copying large and complicated objects is expensive
  - use proxy to pay the price of copying only when the new object is modified
  - Subject must be reference counted
    - proxy increments counter on copy
    - when modified, do copy and decrement counter
    - if (counter == 0) delete Subject

Handle Class Idiom



- String contains a StringRep object
  - StringRep holds the text and reference count
- String passes actual string operations to StringRep object
- String handles pointer operations and deleting StringRep object when reference count reaches zero


```
class StringRep   {
  friend String;

private:
  char *text;
  int refCount;

  StringRep() { *(text = new char[1]) = '\0'; }

  StringRep( const StringRep& s )    {
     strcpy( text = new char[strlen(s.text) + 1], s.text); }

  StringRep( const char *s)    {
     strcpy( text = new char[strlen(s) + 1], s); }

  ~StringRep()   { delete[] text; }

  int length() const    { return strlen( text ); }

  void print() const    { printf("%s\n", text ); }
}

StringRep StringRep::operator+(const StringRep& s) const {
 //  concatenate the two representations
}
```

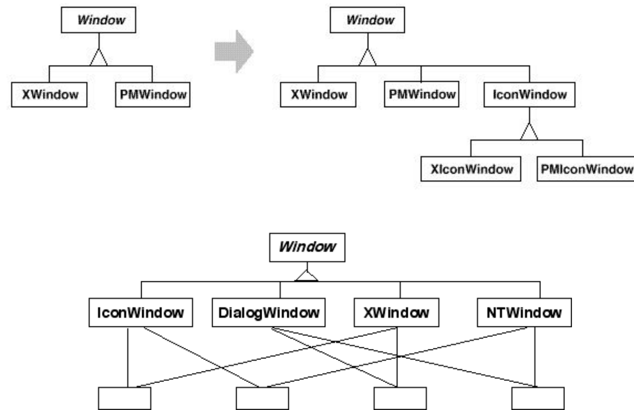


```
class String   {
  friend StringRep;
private:
  StringRep *imp;
public:
  String() {
    imp = new StringRep();
    imp->refCount = 1;  }
  String(const char* charStr)    {
    imp = new StringRep(charStr);
    imp->refCount = 1;}

  String operator=( const String& q){
    imp->refCount--;
    if (imp->refCount <= 0
        && imp != q.imp ) delete imp;
    imp = q.imp;
    imp->refCount++;
    return *this;
  }

  ~String()    {
    imp->refCount--;
    if (imp->refCount <= 0 )
      delete imp;
  }
```

```
String operator+(const String& add) {
    imp = *imp + add.imp;    // proxy behavior
    return *this;
  }
  //...
};
```


Inheritance that Leads to Explosion!

Inheritance that Leads to Explosion!



Bridge Pattern

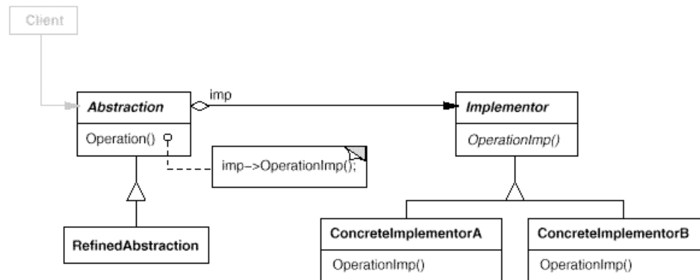
Basic Aspects of Bridge Pattern

- Intent
  - decouple an abstraction from its implementation
  - allow implementation to vary independently from its abstraction
  - abstraction defines and implements the interface
    - all operations in abstraction call methods from its implementation obj.
- In the Bridge pattern ...
  - ... an abstraction can use different implementations
  - ... an implementation can be used in different abstractions

Applicability

- Avoid permanent binding btw. an abstraction and its implementation
- Abstractions and their implementations should be independently extensible by subclassing
- Hide the implementation of an abstraction completely from clients
  - their code should not have to be recompiled when impl. changes
- Share an implementation among multiple objects
  - and this fact should be hidden from the client

Structure



Participants

- Abstraction
  - defines the abstraction's interface
  - maintains a reference to an object of type Implementor
- Implementor
  - defines the interface for implementation classes
    - does not necessarily correspond to the Abstraction's interface
    - Implementor contains primitive operations,
    - Abstraction defines the higher-level operations based on these primitives
- RefinedAbstraction
  - extends the interface defines by Abstraction
- ConcreteImplementer
  - implements the Implementor interface, defining a concrete impl.

Consequences

- Decoupling interface and implementation
  - implem. **configurable** and **changeable** at run-time
  - reduce compile-time dependencies
    - implement. changes do not require Abstraction to recompile
- Improved extensibility
  - extend by subclassing independently Abstractions and Implementations
- Hiding implementation details from clients
  - shield clients from implementations details
    - e.g. sharing implementor objects together with reference counting

BTW ... is the "Handle Class" a Bridge or a Proxy? ;)

Implementation

- Only one Implementor
  - not necessary to create an abstract implementor class
  - degenerate, but useful due to decoupling
- Which Implementor should I use ?
  - Variant 1: let Abstraction know all concrete implem. and choose
  - Variant 2: choose initially default implem. and change later
  - Variant 3: use an Abstract Factory
    - no coupling btw. Abstraction and concrete implem. classes

Windows Example Revisited