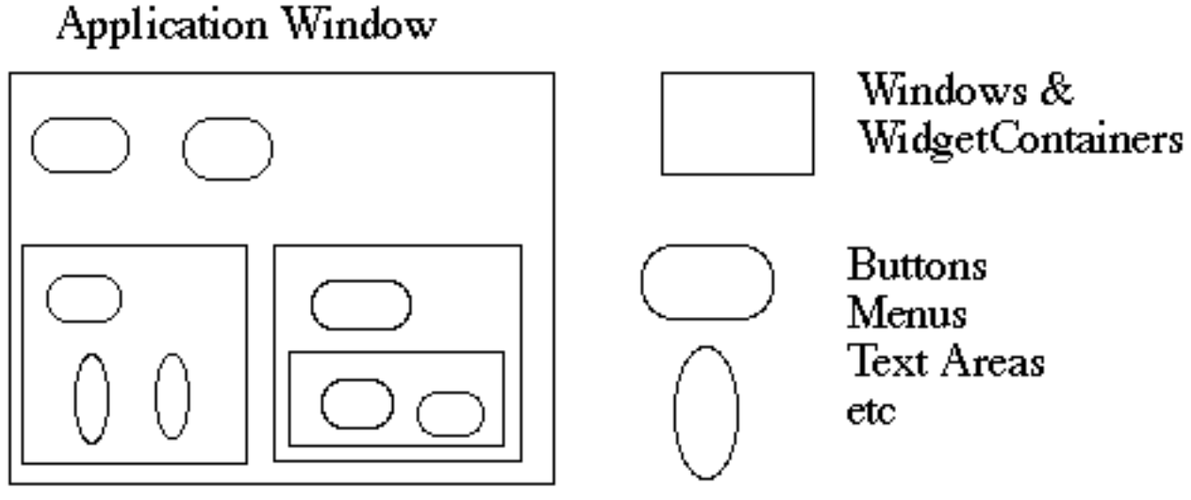
CS 635 Advanced Object-Oriented Design & Programming
Fall Semester, 2020

Doc 17 Composite, Interpreter, Prototype, Builder
Nov 3, 2020

Composite

Composite Motivation



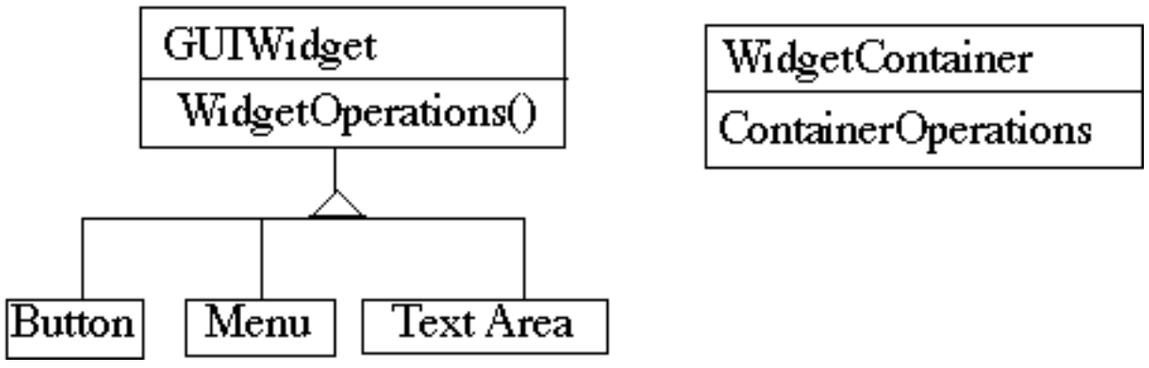
How does the window hold and deal with the different items it has to manage?

Widgets are different that WidgetContainers

Bad News

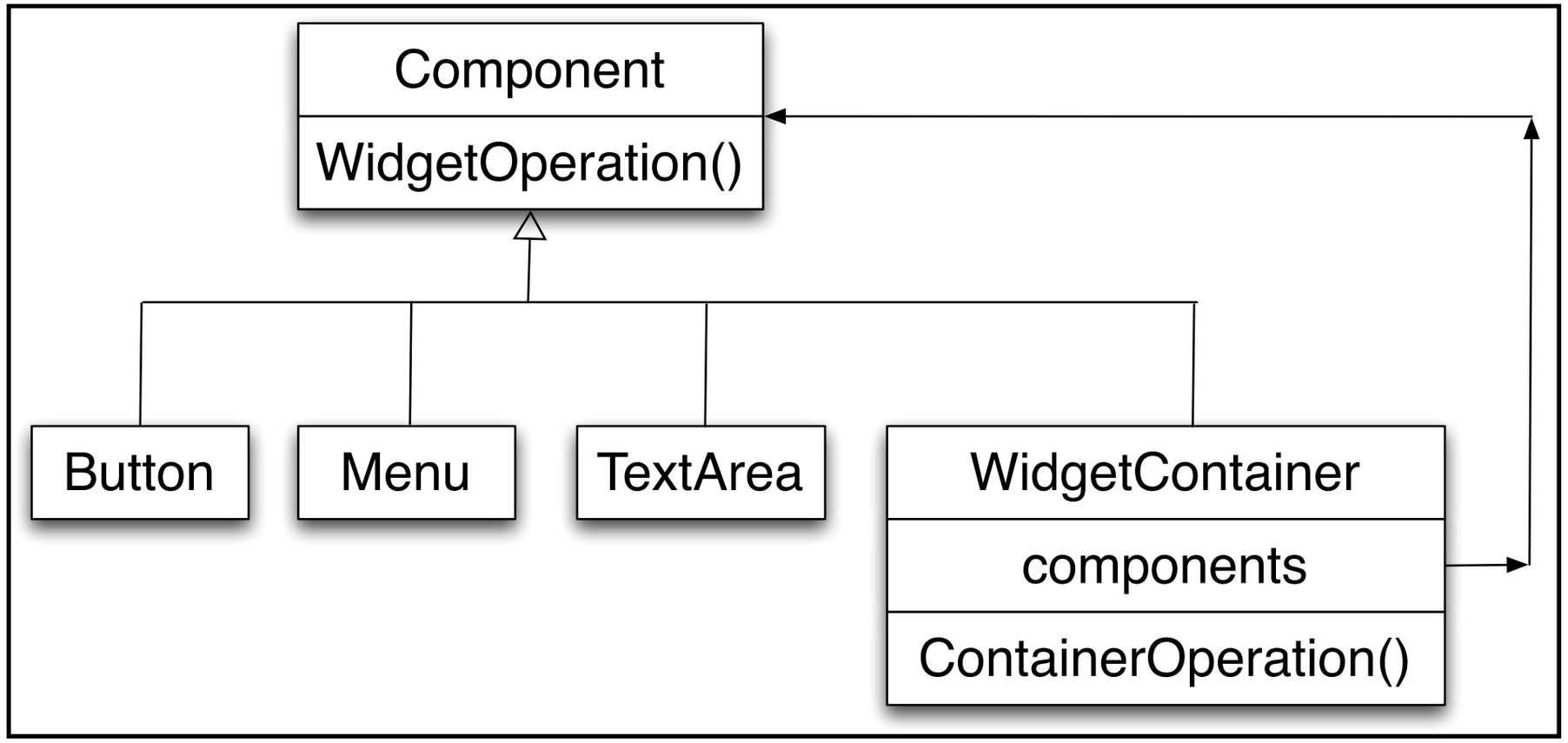
```
class Window {
        Buttons[] myButtons;
        Menus[] myMenus;
        TextAreas[] myTextAreas;
        WidgetContainer[] myContainers;

        public void update() {
                if ( myButtons != null )
                        for ( int k = 0; k < myButtons.length(); k++ )
                                myButtons[k].refresh();
                if ( myMenus != null )
                        for ( int k = 0; k < myMenus.length(); k++ )
                                myMenus[k].display();
                if ( myTextAreas != null )
                        for ( int k = 0; k < myButtons.length(); k++ )
                                myTextAreas[k].refresh();
                if ( myContainers != null )
                        for ( int k = 0; k < myContainers.length(); k++ )
                                myContainers[k].updateElements();
                etc.
        }
        public void fooOperation(){
                if (myButtons != null)
                etc.
```

An Improvement




```
class Window {
        GUIWidgets[] myWidgets;
        WidgetContainer[] myContainers;

        public void update(){
                if ( myWidgets != null )
                        for ( int k = 0; k < myWidgets.length(); k++ )
                                myWidgets[k].update();
                if ( myContainers != null )
                        for ( int k = 0; k < myContainers.length(); k++ )
                                myContainers[k].updateElements();
                etc.
        }
}
```

Composite Pattern



Intent

Compose objects into tree structures to represent part-whole hierarchies.

Lets clients treat individual objects and compositions of objects uniformly

Conflict

Lets clients treat individual objects and compositions of objects uniformly

Composites have add/remove operations that individual objects do not

Composite Pattern

Component implements default behavior for widgets when possible

Button, Menu, etc overrides Component methods when needed

WidgetContainer will have to override all widgetOperations

```
class WidgetContainer {
        Component[] myComponents;

        public void update() {
                if ( myComponents != null )
                        for ( int k = 0; k < myComponents.length(); k++ )
                                myComponents[k].update();
        }
}
```

Issue - WidgetContainer Operations

Should the WidgetContainer operations be declared in Component?

Pro - Transparency

Declaring them in the Component gives all subclasses the same interface

All subclasses can be treated alike. (?)

Con - Safety

Declaring them in WidgetContainer is safer

Adding or removing widgets to non-WidgetContainers is an error

One out is to check the type of the object before using a WidgetContainer operation

Issue - Parent References

```
class WidgetContainer
        {
        Component[] myComponents;

        public void update() {
                if ( myComponents != null )
                        for ( int k = 0; k < myComponents.length(); k++ )
                                myComponents[k].update();
        }

        public add( Component aComponent ) {
                myComponents.append( aComponent );
                aComponent.setParent( this );
        }
}

class Button extends Component {
        private Component parent;
        public void setParent( Component myParent) {
                parent = myParent;
        }

        etc.
```

More Issues

Should Component implement a list of Components?

The button etc. will have a useless data member

Child ordering is important in some cases

Who should delete components?

Applicability

Use Composite pattern when you want

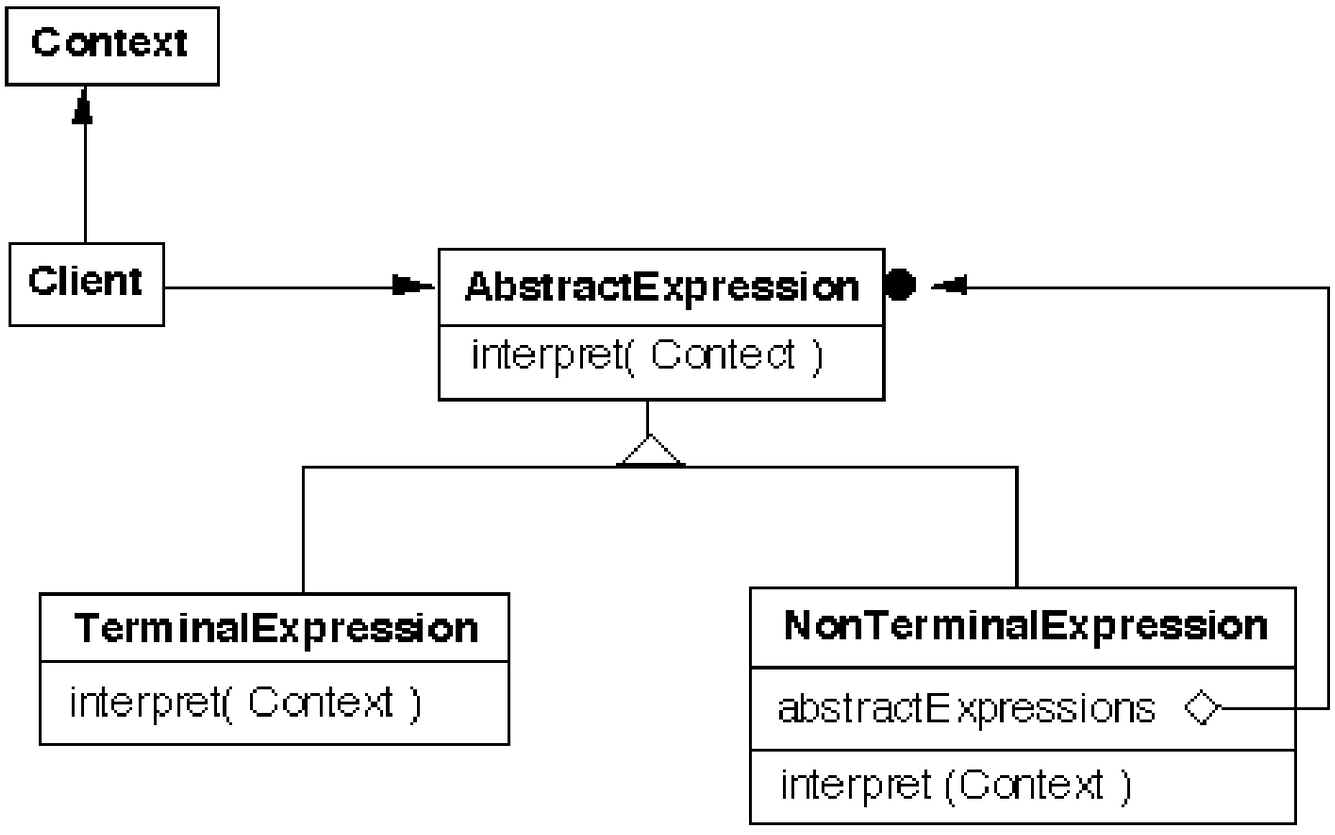
To represent part-whole hierarchies of objects

Clients to be able to ignore the difference between compositions of objects and individual objects

Interpreter

Interpreter

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language



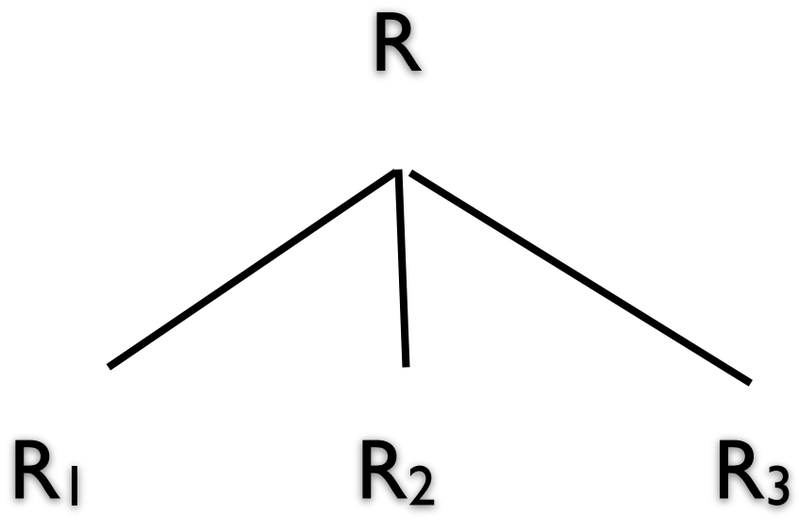
Grammar & Classes

Given a language defined by a grammar like:

$R ::= R_1\ R_2\ R_3$

you create a class for each rule

The classes can be used to construct a tree that represents elements of the language



Example - Boolean Expressions

BooleanExpression ::=

Variable	|
Constant	|
Or	|
And	|
Not	|
BooleanExpression	

And ::= '(' BooleanExpression ‘and’ BooleanExpression ')'

Or ::= '(' BooleanExpression ‘or’ BooleanExpression ')'

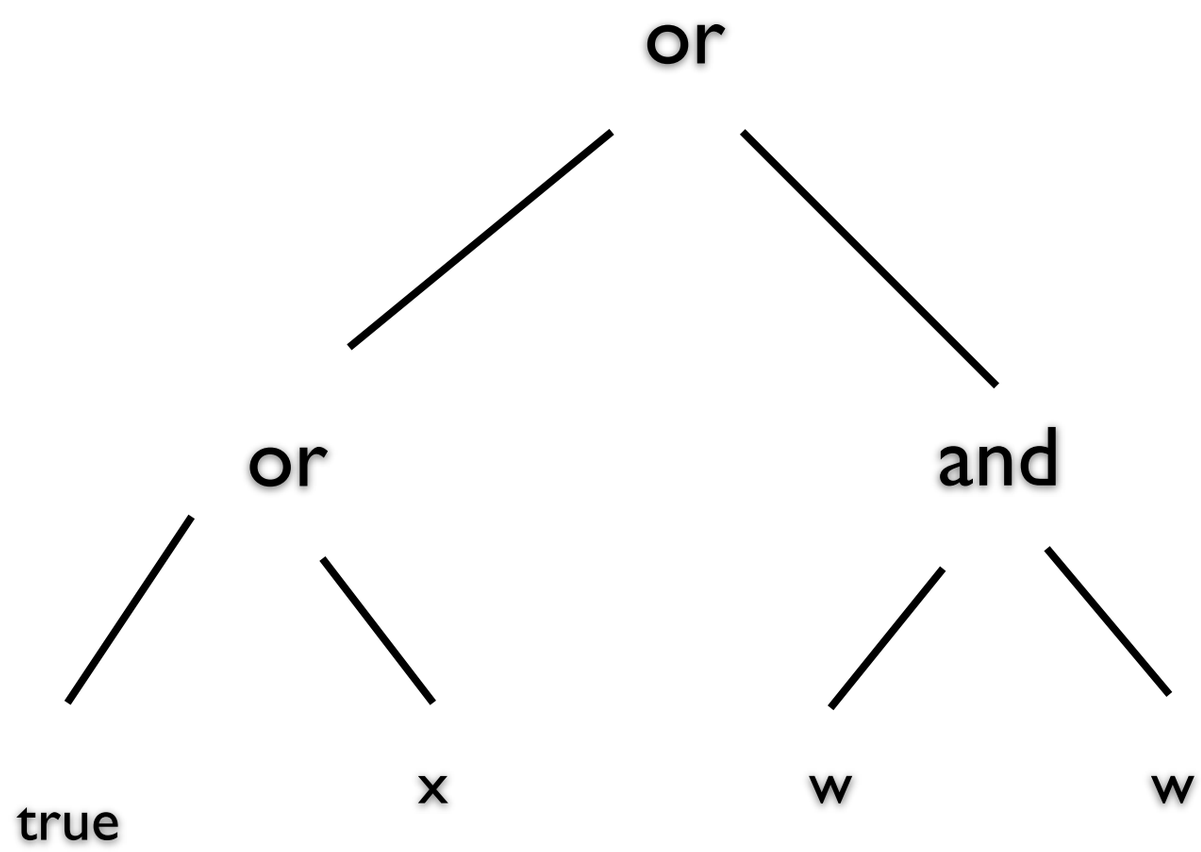
Not ::= ‘not’ BooleanExpression

Constant ::= ‘true’ | ‘false’

Variable ::= String

Sample Expression

((true or x) or (w and x))



Evaluate with

x = true

w = false

Sample Classes

```
public interface BooleanExpression{
    public boolean evaluate( Context values );
    public String toString();
}
```

And

```
public class And implements BooleanExpression {
    private BooleanExpression leftOperand;
    private BooleanExpression rightOperand;

    public And( BooleanExpression leftOperand, BooleanExpression rightOperand) {
        this.leftOperand = leftOperand;
        this.rightOperand = rightOperand;
    }

    public boolean evaluate( Context values ) {
        return leftOperand.evaluate( values ) &&  rightOperand.evaluate( values );
    }

    public String toString(){
        return "(" + leftOperand.toString() + " and " + rightOperand.toString() + ")";
    }
}
```

Constant

```
public class Constant implements BooleanExpression {
    private boolean value;
    private static Constant True = new Constant( true );
    private static Constant False = new Constant( false );

    public static Constant getTrue() {  return True; }

    public static Constant getFalse(){ return False; }

    private Constant( boolean value) { this.value = value; }

    public boolean evaluate( Context values ) { return  value;  }

    public String toString() {
       return String.valueOf( value );
    }
}
```

Variable

```
public class Variable implements BooleanExpression {

    private String name;

    private Variable( String name ) {
        this.name = name;
    }

    public boolean evaluate( Context values ) {
        return values.getValue( name );
    }

    public String toString() { return name; }
}
```

Context

```
public class Context {
   Hashtable<String,Boolean> values = new Hashtable<String,Boolean>();

   public boolean getValue( String variableName ) {
      return values.get( variableName );
   }

   public void setValue( String variableName, boolean value ) {
      values.put( variableName,  value ) );
   }
}
```

((true or x) or (w and x))

```
public class Test {
    public static void main( String args[] ) throws Exception  {
        BooleanExpression left =
            new Or(  Constant.getTrue(), new Variable( "x" ) );
        BooleanExpression right =
            new And(  new Variable( "w" ), new Variable( "x" ) );

        BooleanExpression all = new Or(  left, right );

        System.out.println( all );
        Context values = new Context();
        values.setValue( "x", true );
        values.setValue( "w", false );

        System.out.println( all.evaluate( values ) );
        }
    }
```

Consequences

It's easy to change and extend the grammar

Implementing the grammar is easy

Complex grammars are hard to maintain

- Use JavaCC or SmaCC instead

Adding new ways to interpret expressions

- The visitor pattern is useful here

Complicates design when a language is simple

Supports combinations of elements better than implicit language

Implementation

The pattern does not talk about parsing!

Flyweight

If terminal symbols are repeated many times using the Flyweight pattern can reduce space usage

Composite

Abstract syntax tree is an instance of the composite

Iterator

Can be used to traverse the structure

Visitor

Can be used to place behavior in one class

Prototype

Prototype

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype

Applicability

Use the Prototype pattern when

A system should be independent of how its products are created, composed, and represented; and

When the classes to instantiate are specified at run-time; or

To avoid building a class hierarchy of factories that parallels the class hierarchy of products; or

When instances of a class can have one of only a few different combinations of state.

Insurance Example

Insurance agents start with a standard policy and customize it

Two basic strategies:

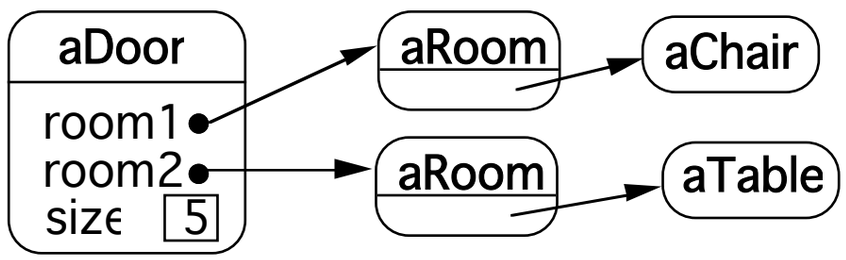
Copy the original and edit the copy

Store only the differences between original and the customize version in a decorator

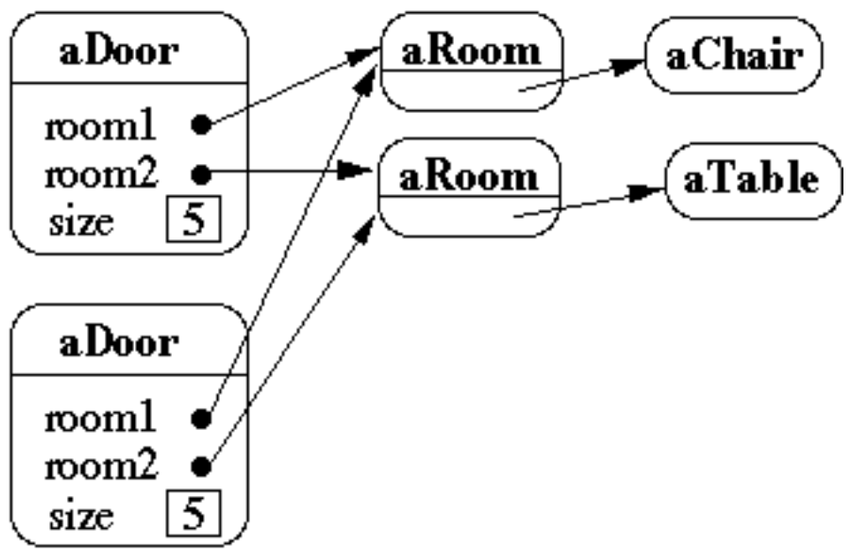
Copying Issues

Shallow Copy Verse Deep Copy

Original Objects

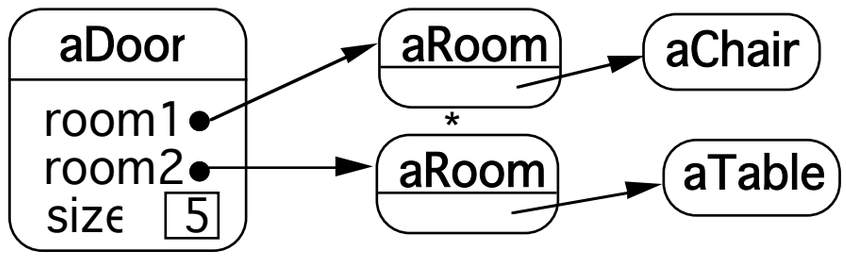


Shallow Copy

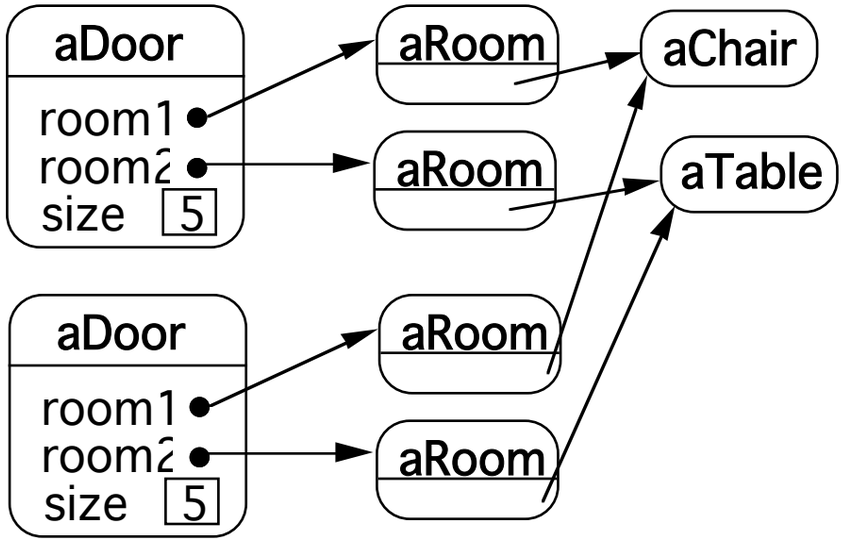


Shallow Copy Verse Deep Copy

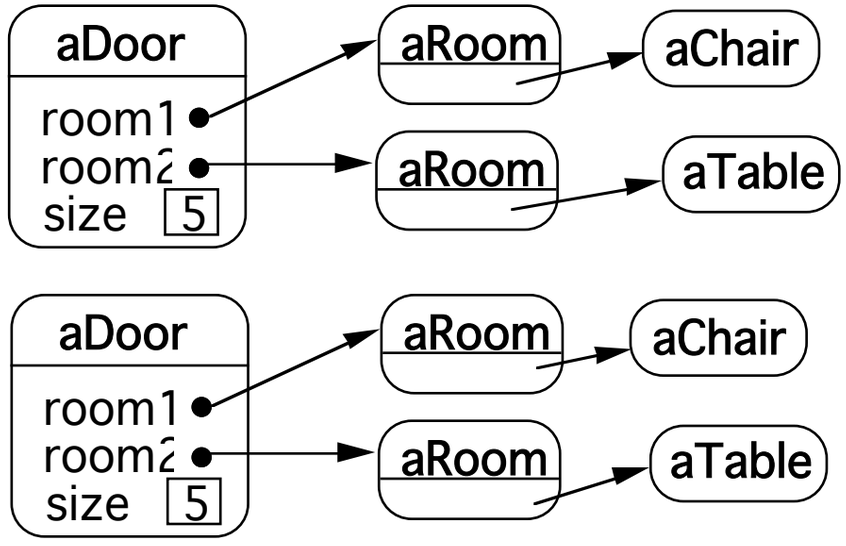
Original Objects



Deep Copy



Deeper Copy



Prototype-based Languages

No classes

Behavior reuse (inheritance)

Cloning existing objects which serve as prototypes

Some Prototype-based languages

Self

JavaScript

Squeak (eToys)

Perl with Class::Prototyped module

JavaScript - Copying

```
var Animal = {
  type: 'Invertebrates',
  displayType: function() {
    console.log(this.type);
  }
};
```

Animal is the Prototype

```
var animal1 = Object.create(Animal);
animal1.displayType();              // Output:Invertebrates

var fish = Object.create(Animal);
fish.type = 'Fishes';
fish.displayType();                 // Output:Fishes

var copy = {...Animal};
```

JavaScript Prototype

Every object has has a prototype

When looking for a property or method in an object

- Look in the object if not there
- Look in prototype, if not there
- Look in the prototype's prototype, if not there
- Continue looking in the prototype chain until find it or reach the end

A parent class's prototype is added to the subclasses prototype chain

But prototype chains are dynamic

- They can be changed at runtime

JavaScript Class

```
class Rectangle {
    constructor(height, width) {
      this.height = height;
      this.width = width;
    }

  }
```

```
const q = new Rectangle(10, 45);
q.height;
q['height'];
```

Prototype Example

```
class Cat {
  constructor(name) {
    this.name = name;
  }
}

class Dog {
  constructor(name) {
    this.name = name;
  }
}
```

```
let fluffy = new Cat("Fluffy");
let spot = new Dog('Spot');
fluffy['food'];                    //undefined

 Object.color = 'red';
 Object.prototype.food = "mouse";
 Cat.prototype.size = 'small';

  fluffy['food'];                  // 'mouse'
  fluffy.size;                     // 'small'
  fluffy.color;                    // undefined
  spot.food;                       // 'mouse'

  Dog.prototype.food = 'rabbit';
  spot.food;                       // 'rabbit'
  fluffy.food;                     // 'mouse'
```

JavaScript Classes are Functions

```
class Rectangle {
    constructor(height, width) {
      this.height = height;
      this.width = width;
    }
 }

 console.log(typeof Rectangle);                    // function

 console.log(Object.getOwnPropertyNames(Rectangle));       // [ 'length', 'prototype', 'name' ]

 console.log(Rectangle.name);                      // Rectangle

 console.log(Rectangle.length);                    // 2
```

Most Things are Objects in JavaScript

Object

- Function
- Array
- Date
- RegExp

```
function adder(x, y) {
  return x + y;
}

console.log(Object.getOwnPropertyNames(adder))
```

```
[ 'length', 'name', 'arguments', 'caller', 'prototype' ]
```

Builder

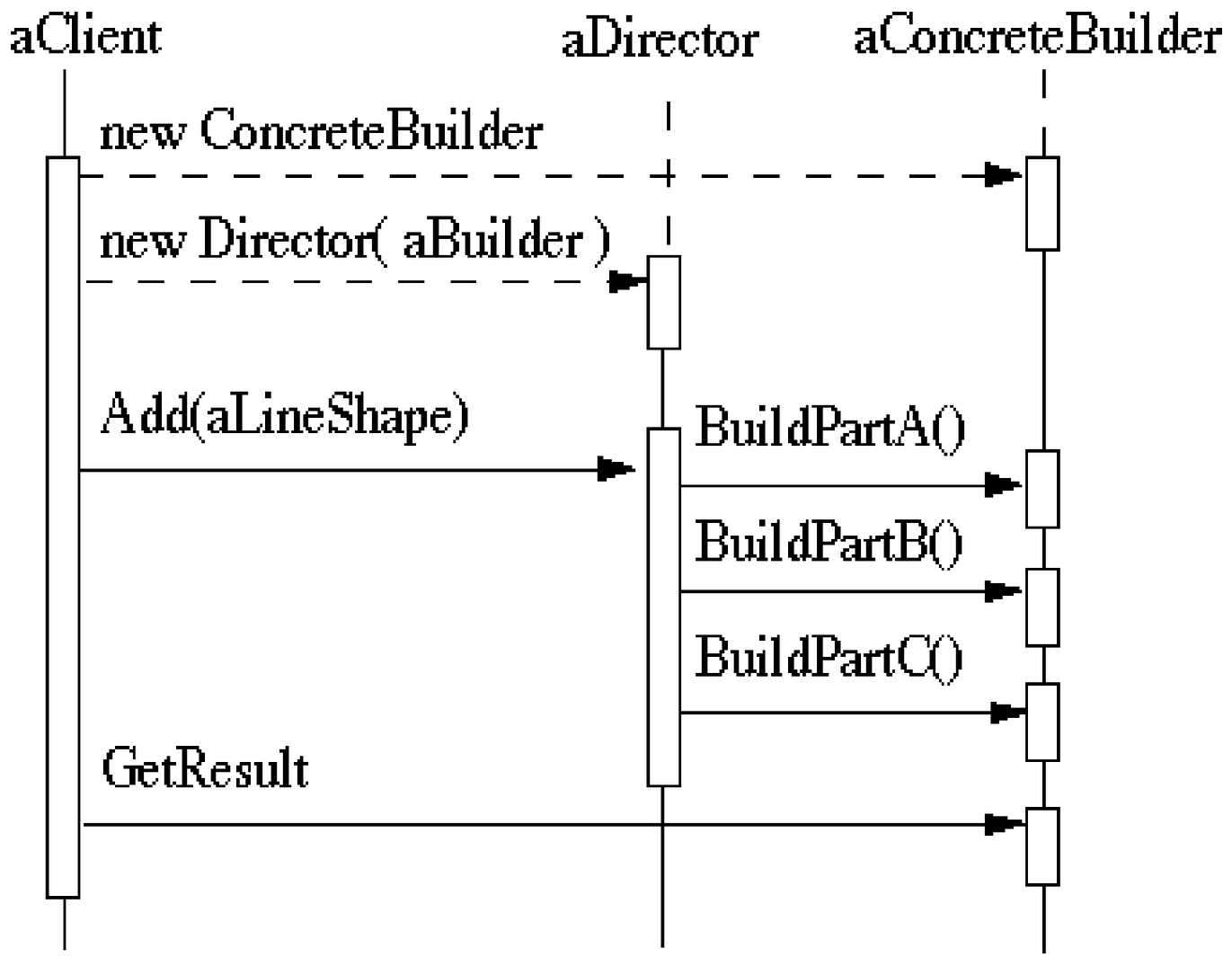
Builder

Separate construction of a complex object from its representation

So same construction process can create different representations

Builder

Client Director Builder



RTF Converter

A word processing document has complex structure

How to convert Rich Text Format (RTF) to

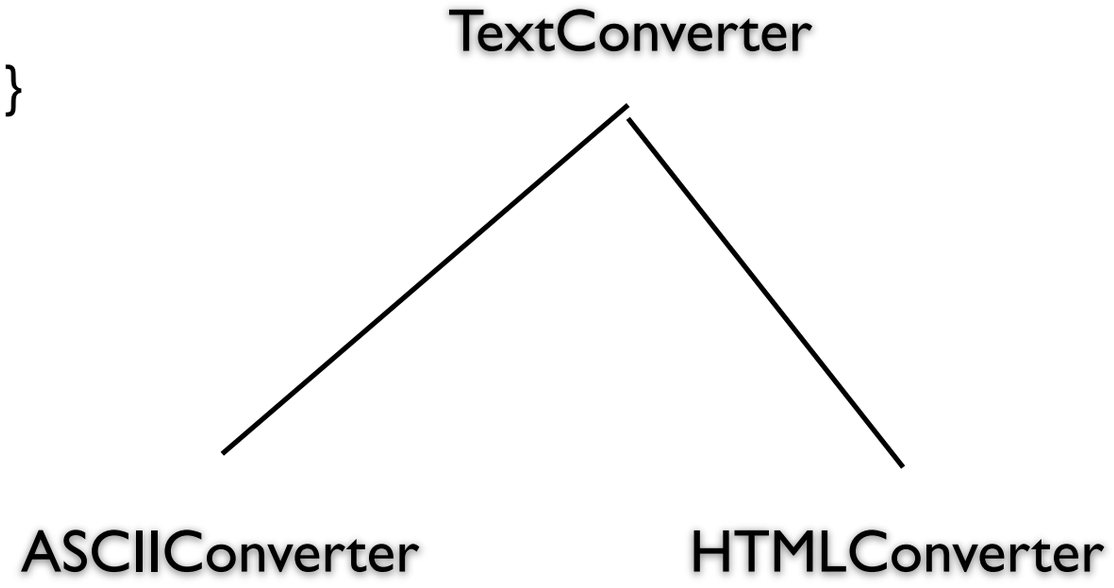
- TeX
- html
- PDF
- etc.

Pseudo Solution

```
class RTF_Reader {
   TextConverter builder;
   String RTF_Text;

   public RTF_Reader( TextConverter aBuilder, String RTFtoConvert ){
      builder = aBuilder;
      RTF_Text = RTFtoConvert;
      }

   public void parseRTF(){
      RTFTokenizer rtf = new RTFTokenizer( RTF_Text );

      while ( rtf.hasMoreTokens() ){
         RTFToken next = rtf.nextToken();

         switch ( next.type() ){
            case CHAR:    builder.character( next.char() ); break;
            case FONT:    builder.font( next.font() ); break;
            case PARA:    builder.newParagraph( ); break;
            etc.
      }
```

Builder Classes

```
abstract class TextConverter {
   public void character( char nextChar ) { }
   public void font( Font newFont )   { }
   public void newParagraph() {}
}
```



Sample Program

```
main(){
   ASCII_Converter simplerText = new ASCII_Converter();
   String rtfText;

   // read a file of rtf into rtfText

   RTF_Reader myReader =
      new RTF_Reader( simplerText, rtfText );

   myReader.parseRTF();

   String myProduct = simplerText.getText();
}
```

The Hard Part

The builder interface

XML Example

```
<?xml version="1.0" encoding="UTF-8"?>
<RootElement param="value">
    <FirstElement>
        Some Text
    </FirstElement>
    <SecondElement param2="something">
        Pre-Text <Inline>Inlined text</Inline> Post-text.
    </SecondElement>
</RootElement>
```

SAX - Builder Pattern

Director	XMLReader
Builder	ContentHandler

ContentHandler Interface

void characters(char[] ch, int start, int length)
Receive notification of character data.

void endDocument()
Receive notification of the end of a document.

void endElement(java.lang.String uri, java.lang.String localName, java.lang.String qName)
Receive notification of the end of an element.

void endPrefixMapping(java.lang.String prefix)
End the scope of a prefix-URI mapping.

void ignorableWhitespace(char[] ch, int start, int length)
Receive notification of ignorable whitespace in element content.

void processingInstruction(java.lang.String target, java.lang.String data)
Receive notification of a processing instruction.

void setDocumentLocator(Locator locator)
Receive an object for locating the origin of SAX document events.

void skippedEntity(java.lang.String name)
Receive notification of a skipped entity.

void startDocument()
Receive notification of the beginning of a document.

void startElement(java.lang.String uri, java.lang.String localName, java.lang.String qName, Attributes att
Receive notification of the beginning of an element.

void startPrefixMapping(java.lang.String prefix, java.lang.String uri)
Begin the scope of a prefix-URI Namespace mapping.

Simple API XML (SAX)

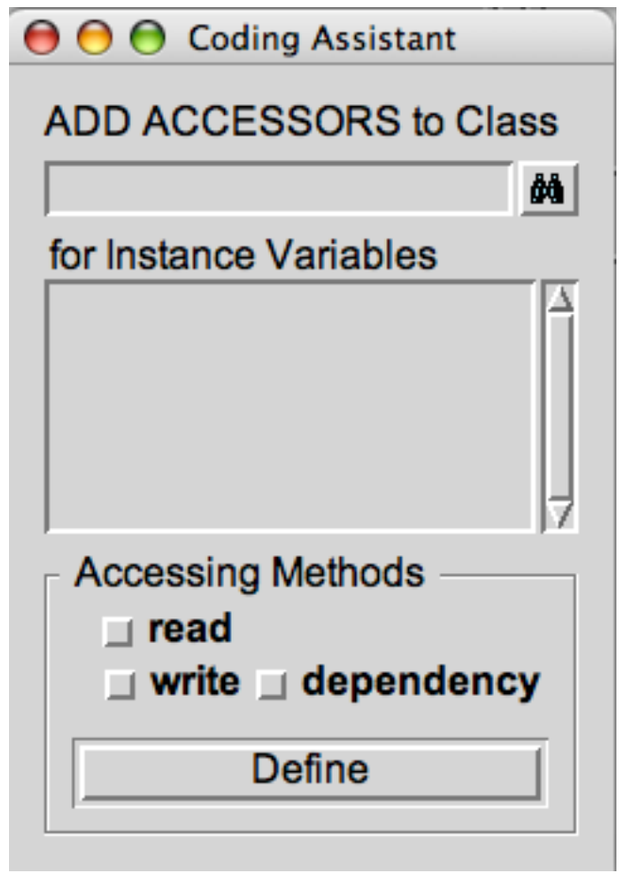
```
public static void main (String args[]) throws Exception {
  XMLReader director = XMLReaderFactory.createXMLReader();
  ContentHandler builder = new MySAXApp();
  director.setContentHandler(builder);
  director.setErrorHandler(builder);

  FileReader source = new FileReader("Foo.xml");
  director.parse(new InputSource(source));
  handler.getResult();
  }
```

Examples - VW Smalltalk

ClassBuilder
MenuBuilder
UIBuilder

UIBuilder

```
#(#{UI.FullSpec}
    #window:
    #(#{UI.WindowSpec}
      #label: #(#{Kernel.UserMessage} #key: #CodingAssistant
            #defaultString: 'Coding Assistant' #catalogID: #UIPainter)
      #min: #(#{Core.Point} 242 320 )
      #max: #(#{Core.Point} 242 320 )
      #bounds: #(#{Graphics.Rectangle} 279 140 521 460 ) )
    #component:
    #(#{UI.SpecCollection}
      #collection: #(
        #(#{UI.LabelSpec}
          #layout: #(#{Graphics.LayoutOrigin} 14 0 12 0 )
          #label: #(#{Kernel.UserMessage} #key: #ADDACCESSORSToClass
            #defaultString: 'ADD ACCESSORS to Class' #catalogID: #UIPainter) )
        #(#{UI.LabelSpec}
          #layout: #(#{Graphics.LayoutOrigin} 16 0 65 0 )
          #label: #(#{Kernel.UserMessage} #key: #forInstanceVariables
              #defaultString: 'for Instance Variables' #catalogID: #UIPainter) )
```



Simplified Builder Pattern

More common the standard Pattern

Used to set multiple fields

Replaces using constructor with many parameters

Person Example

```
public class Person
{
   private final String lastName;
   private final String firstName;
   private final String middleName;
   private final String salutation;
   private final String suffix;
   private final String streetAddress;
   ...
   private final boolean isEmployed;
```

PersonBuilder

```
public class PersonBuilder
{
   private String newLastName;
   private String newFirstName;
   private String newMiddleName;
   private String newSalutation;
   private String newSuffix;
   private String newStreetAddress;
   ...
   private boolean newIsEmployed;

   public PersonBuilder setLastName(String newLastName) {
      this.newLastName = newLastName;
      return this;
   }

   public PersonBuilder setFirstName(String newFirstName) {
      this.newFirstName = newFirstName;
      return this;
   }
```

PersonBuilder - Continued

```
public PersonBuilder setMiddleName(String newMiddleName) {
      this.newMiddleName = newMiddleName;
      return this;
   }
```

The rest of the set methods

```
public Person createPerson() {
      return new Person(newLastName, newFirstName, newMiddleName, newSalutation,
newSuffix, newStreetAddress, newCity, newState, newIsFemale, newIsEmployed,
newIsHomeOwner);
   }
```

Building a Person

```
Person test = new PersonBuilder().
        setLastName("Whitney").
        setFirstName("Roger").
        ...
        setIsEmployed(true).
        createPerson();
```

Improvements

Make Builder an inner class (Java)

Group fields into separate classes

- Name Class
  - firstName
  - lastName
  - middleName
  - salutation
  - suffix

Android Example

Building a Notification

```
Notification note = new Notification.Builder(mContext)
        .setContentTitle("New mail from " + sender.toString())
        .setContentText(subject)
        .setSmallIcon(R.drawable.new_mail)
        .setLargeIcon(aBitmap)
        .build();
```

Strategy
vs
Builder