CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2005
Doc 15 Prototype & Builder

Contents

Prototype.......... 3

Intent.......... 3

Applicability.......... 3

Copying Issues.......... 5

Shallow Copy Verse Deep Copy..........5

Cloning Issues..........8

Consequences.......... 11

Implementation Issues.......... 11

Builder.......... 12

Applicability.......... 12

Example – XML Parser.......... 14

Consequences.......... 17

References

Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995, pp. 117-126, 97-106

The Design Patterns Smalltalk Companion, Alpert, Brown, Woolf, Addision-Wesley, 1998, pp. 77-90, 47-62

Prototype
Intent

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.

    It may be easier to have the proper number of prototypes and clone them rather than instantiating the class manually each time

Insurance Example

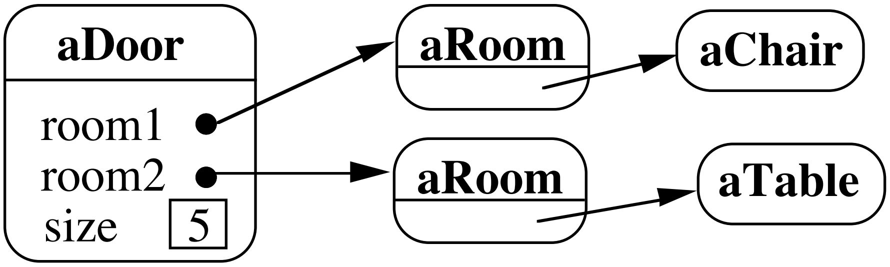
Insurance agents start with a standard policy and customize it

Two basic strategies:

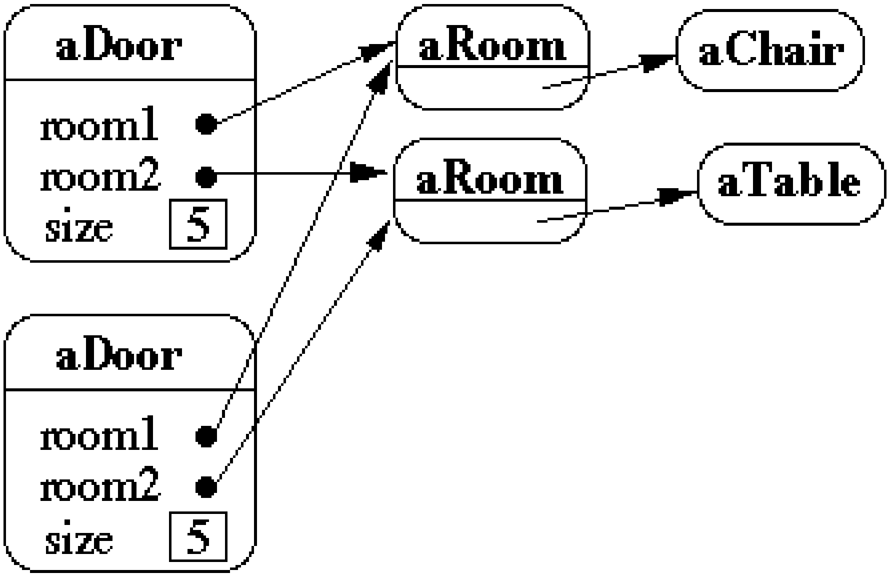
- Copy the original and edit the copy
- Store only the differences between original and the customize version in a decorator

Copying Issues
Shallow Copy Verse Deep Copy

Original Objects

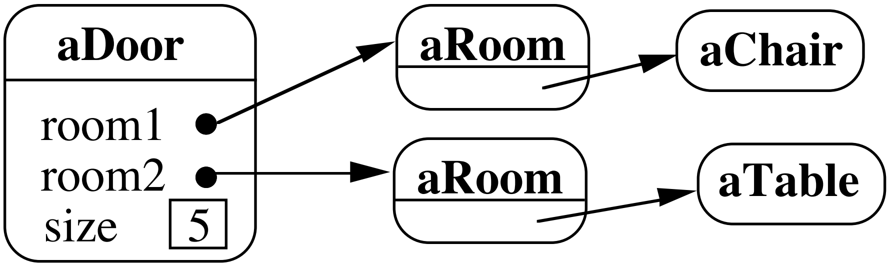


Shallow Copy

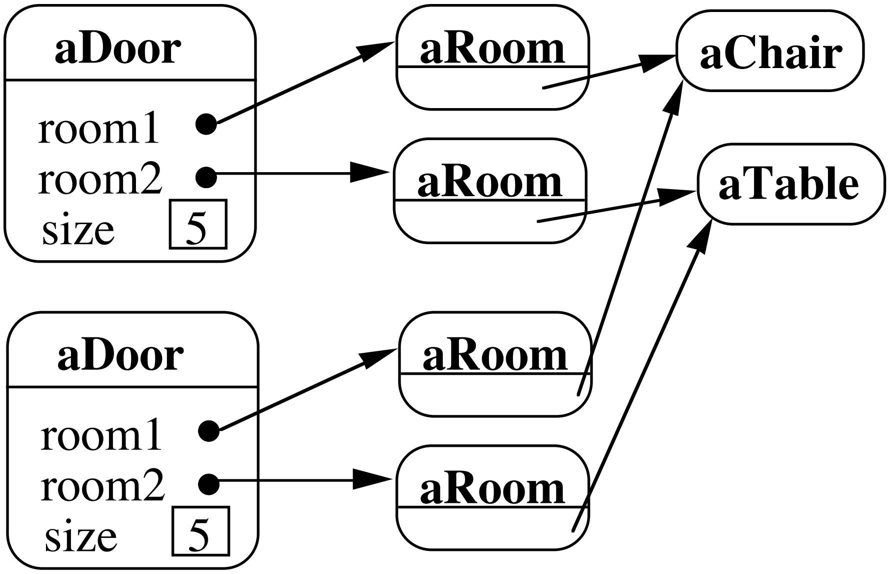


Shallow Copy Verse Deep Copy

Original Objects

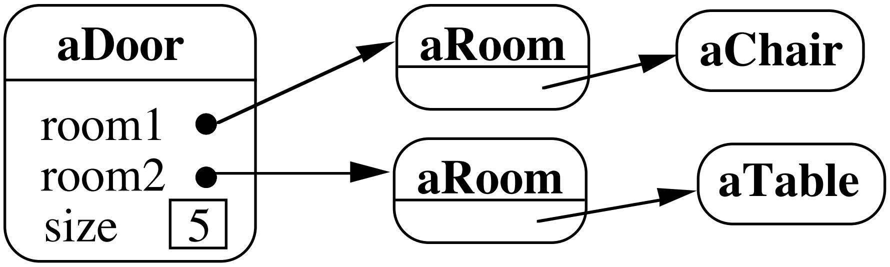


Deep Copy

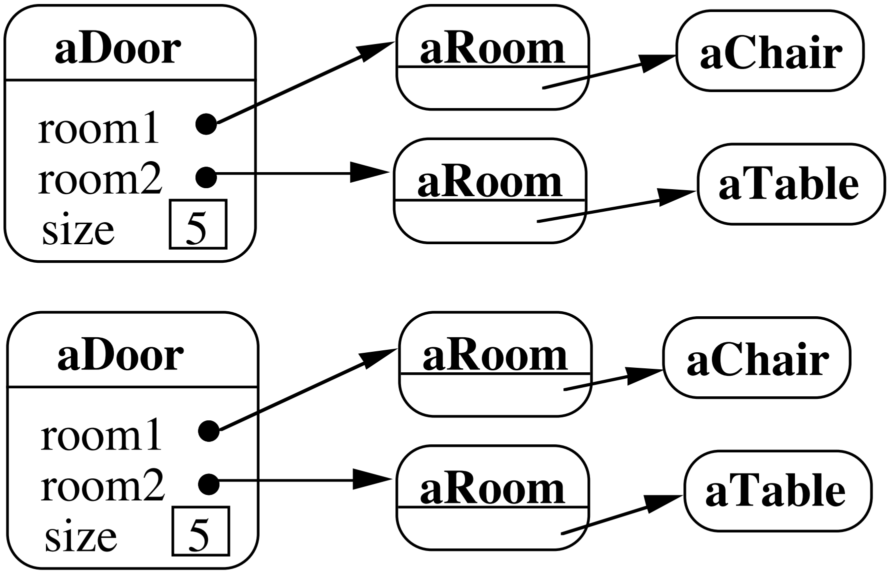


Shallow Copy Verse Deep Copy

Original Objects



Deeper Copy



Cloning Issues
How to in C++ - Copy Constructors

```
class Door
   {
   public:
      Door();
      Door( const Door&);

      virtual Door* clone() const;

      virtual void Initialize( Room*, Room* );
      // stuff not shown
   private:
      Room* room1;
      Room* room2;
   }

Door::Door ( const Door& other ) //Copy constructor
   {
   room1 = other.room1;
   room2 = other.room2;
   }

Door* Door::clone()  const
   {
   return new Door( *this );
   }
```

How to in Java - Object clone()

protected Object clone() throws CloneNotSupportedException

Default is shallow copy

Returns: A clone of this Object.

Throws: OutOfMemoryError

Throws: CloneNotSupportedException

```
class Door implements Cloneable {
   public void Initialize( Room a, Room b)
      { room1 = a; room2 = b; }

   public Object clone() throws
               CloneNotSupportedException {
      // modify this method for deep copy
      // no need to implement this method for shallow copy
      return super.clone();
   }
   Room room1;
   Room room2;
}
```

VisualWorks Smalltalk

```
Object>>shallowCopy
   Does a shallowCopy of the receiver
```

```
Object>>copy
   ^self shallowCopy postCopy

   “Template method for copy”
```

Copy is the primary method for copying an object

Classes override postCopy to do more than shallow copyDoor

```
Smalltalk.CS635 defineClass: #Door
   superclass: #{Core.Object}
   indexedType: #none
   private: false
   instanceVariableNames: 'room1 room2 '

postCopy
   room1 := room1 copy.
   room2 := room2 copy.
```

Consequences

- Adding and removing products at run-time
- Specifying new objects by varying values
- Specifying new objects by varying structure
- Reducing subclassing (from factory method)
- Configuring an application with classes dynamically

Implementation Issues

- Using a prototype manager
- Implementing the Clone operation
- Initializing clones

Builder
Intent

Separate the construction of a complex object from its representation so that the same construction process can create different representations

Applicability

Use the Builder pattern when

- The algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled

- The construction process must allow different representations for the object that's constructed

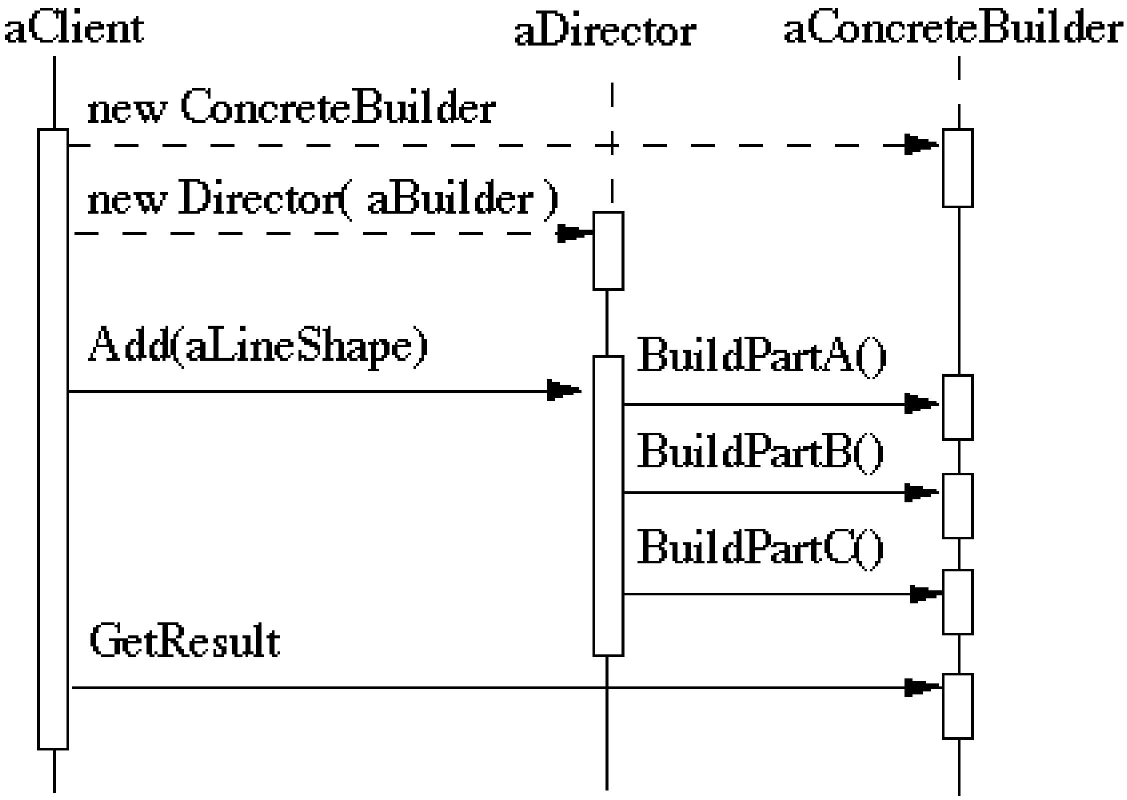
Collaborations

The client creates the Director object and configures it with the desired Builder object

Director notifies the builder whenever a part of the product should be built

Builder handles requests from the director and adds parts to the product

The client retrieves the product from the builder



Example – XML Parser

Director

XML Parser

Abstract Builder Class

XML.SAXDriver (Smalltalk)
org.xml.sax.helpers.DefaultHandler (Java)
DefaultHandler (C++)

Concrete Builder Class

Your subclass of the abstract builder

Client

Your code that uses the tree built

Java Example

```
public static void main(String argv[])
   {
   SAXDriverExample handler = new SAXDriverExample();

   // Use the default (non-validating) parser
   SAXParserFactory factory = SAXParserFactory.newInstance();
   try
      {
      SAXParser saxParser = factory.newSAXParser();
      saxParser.parse( new File("sample"), handler );
      }
   catch (Throwable t)
      {
      t.printStackTrace();
      }
   System.out.println( handler.root());
   }
```

Smalltalk Example

```
| builder exampleDispatcher |
```

```
builder := SAXDriverExample new.
exampleDispatcher := SAXDispatcher new contentHandler: builder.
XMLParser
   processDocumentInFilename: 'page'
   beforeScanDo:
      [:parser |
      parser
         saxDriver:(exampleDispatcher);
         validate: true].
builder root.
```

Consequences

- It lets you vary a product's internal representation
- It isolates code for construction and representation
- It gives you finer control over the construction process

Implementation

- Assembly and construction interface

  Builder may have to pass parts back to director, who will then pass them back to builder

- Why no abstract classes for products
- Empty methods as default in Builder