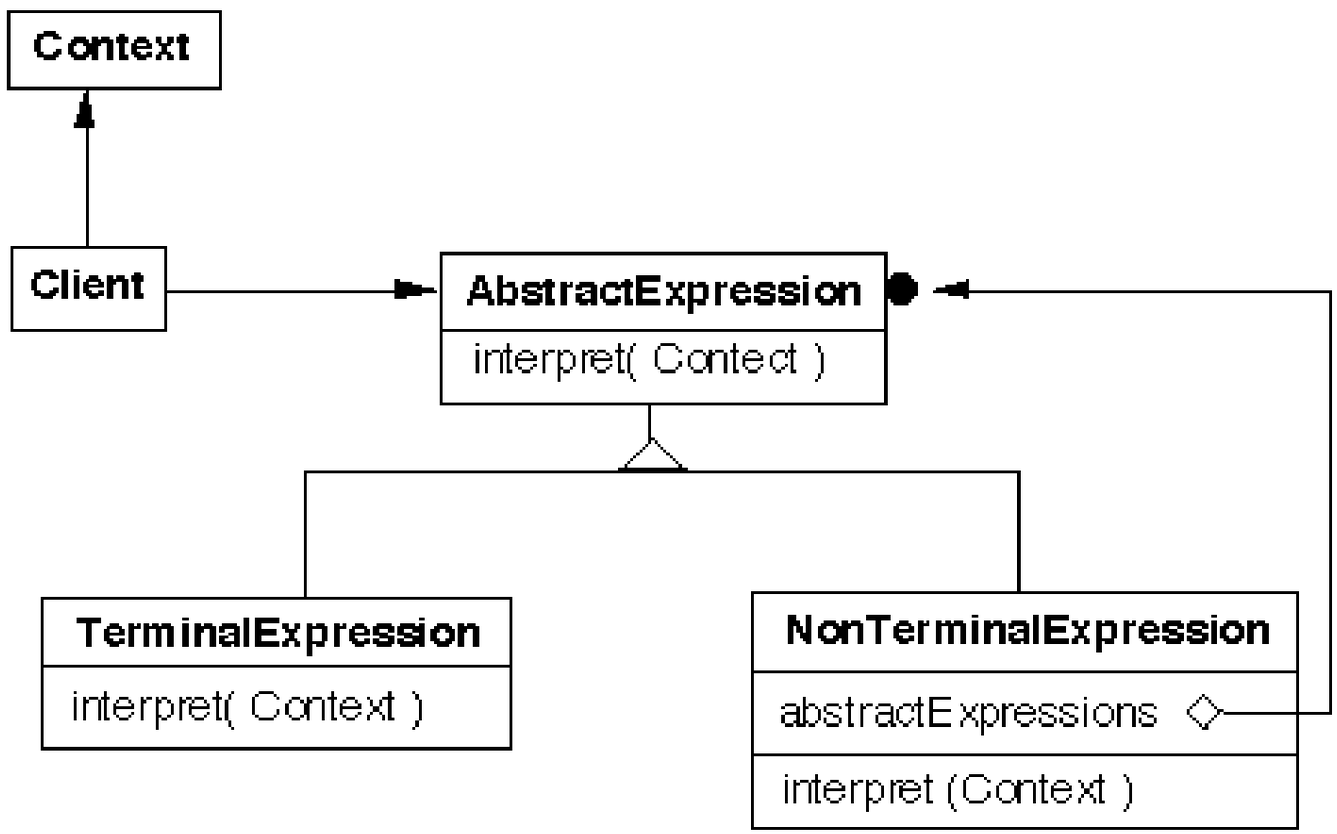
CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2016
Doc 11 Interpreter, Factory Method, Effective Java, Builder
Mar 15, 2016

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



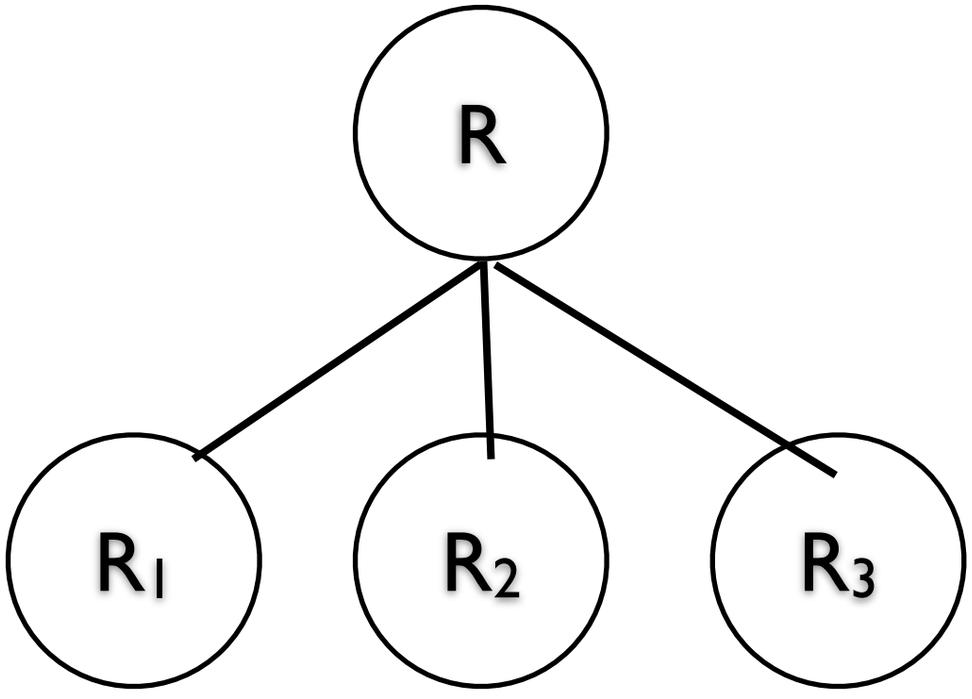
Grammar & Classes

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

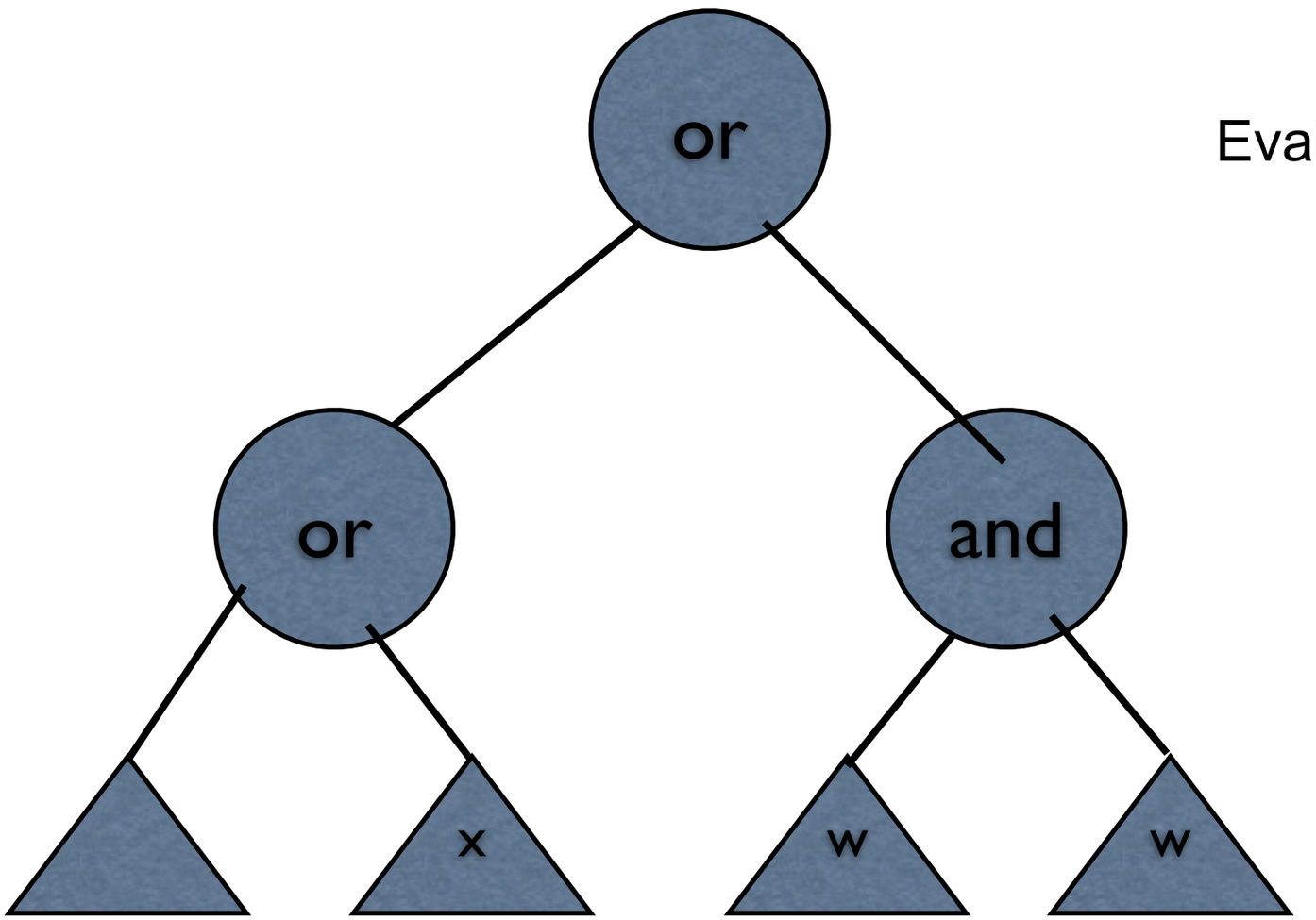


Example - Boolean Expressions

```
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

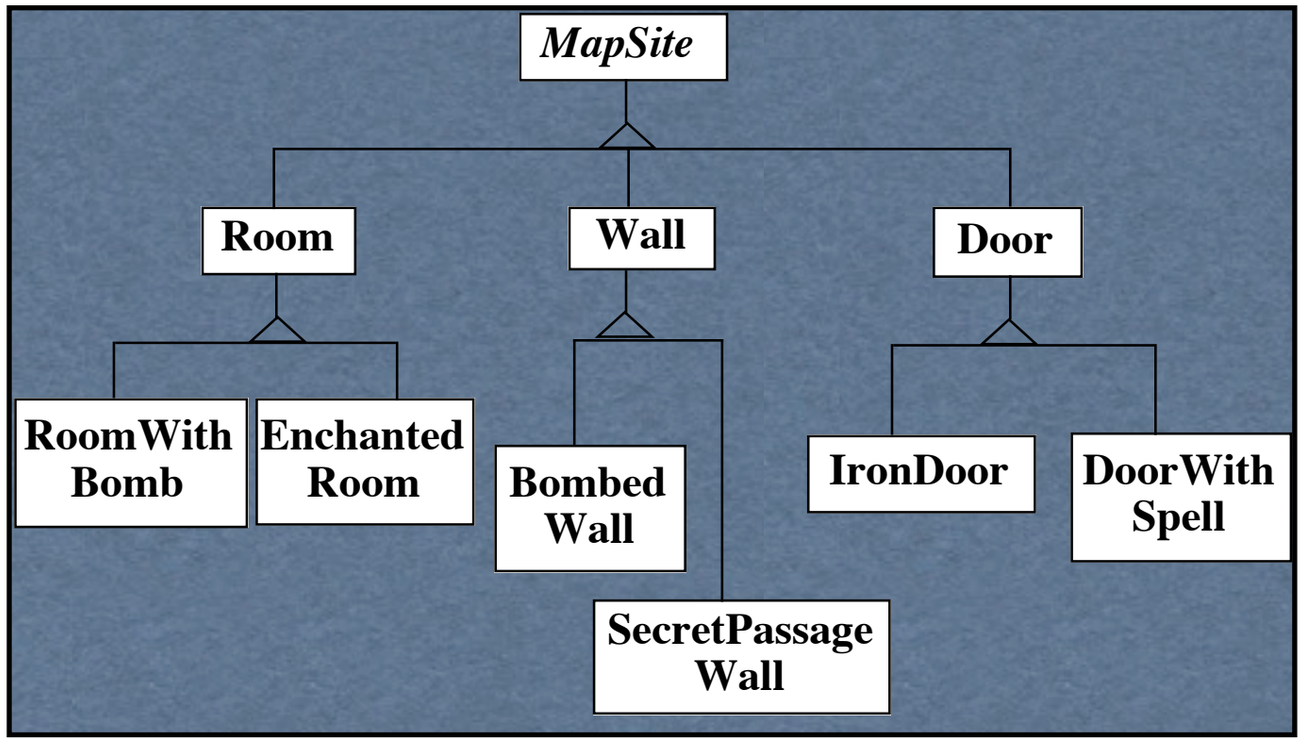
Factory Method

Factory Method

A template method for creating objects

```
public class Example {
    protected Bar bar() { return new Bar(); }

    public void foo() {
        blah
        Bar soap = bar();
        blah;
    }
```

Maze Game Example



Maze Game Example

```
class MazeGame{
    public Maze makeMaze() { return new Maze(); }
    public Room makeRoom(int n ) { return new Room( n ); }
    public Wall makeWall() { return new Wall(); }
    public Door makeDoor() { return new Door(); }

    public Maze CreateMaze(){
        Maze aMaze = makeMaze();
        Room r1 = makeRoom( 1 );
        Room r2 = makeRoom( 2 );
        Door theDoor = makeDoor( r1, r2);

        aMaze.addRoom( r1 );
        aMaze.addRoom( r2 );
        etc

        return aMaze;
        }
    }
```

```
class BombedMazeGame extends MazeGame {

    public Room makeRoom(int n )  {
        return new RoomWithABomb( n );
    }

    public Wall makeWall() {
        return new BombedWall();
    }
}
```

Don't repeat your self

```
public class LinkedList extends Collection {
    public OrderedLinkedList() {
        this(defaultOrder());
    }

    public LinkedList(Order listOrder ) {
        this(listOrder, new OrderedCollection());
    }

    public LinkedList(Collection items) {
        this(defaultOrder(), items);
    }

    protected Order defaultOrder() {
        return new RandomOrder();
    }

    public LinkedList(Order listOrder, Collection items) {
        blah
```

Implementation Variation

```
class Hershey {

    public Candy makeChocolateStuff( CandyType id ) {
        if ( id == MarsBars ) return new MarsBars();
        if ( id == M&Ms ) return new M&Ms();
        if ( id == SpecialRich ) return new SpecialRich();

        return new PureChocolate();
    }

class GenericBrand extends Hershey {
    public Candy makeChocolateStuff( CandyType id ) {
        if ( id == M&Ms ) return new Flupps();
        if ( id == Milk ) return new MilkChocolate();
        return super.makeChocolateStuff(id);
    }
}
```

Using C++ Templates

```
template <class ChocolateType>
class Hershey
    {
    public:
        virtual Candy* makeChocolateStuff( );
    }

template <class ChocolateType>
Candy*
Hershey<ChocolateType>::makeChocolateStuff( )
    {
    return new ChocolateType;
    }

Hershey<SpecialRich> theBest;
```

Smalltalk Variant

Return the class, caller creates an object

```
chocolateStuff
        ^SpecialRich
```

```
some code
candy := (self chocolateStuff) new
mode code
```

Use Factory Method When

A class can't anticipate the class of objects it must create

A class wants its subclasses to specify the objects it creates

You want to localize the knowledge of which help classes is used in a class

But when is this?

CS 580 Example - Testing a Server

```
public class SDWitterServer {
    public void run(int port) throws IOException {
        ServerSocket input = new ServerSocket( port );

        while (true) {
            Socket client = input.accept();
            processRequest(
                client.getInputStream(),
                client.getOutputStream());
            client.close();
        }
    }

    void processRequest(InputStream in,OutputStream out) {
        do a bunch of stuff
    }

    etc.
```


This code requires us to send/receive data over the network, which at times is not convenient in unit tests.

Using Factory Method

```
public class SDWitterServer {
    public void run(int port) throws IOException {
        ServerSocket input = this.serverSocket( port );

        while (true) {
            Socket client = input.accept();
            processRequest(
                client.getInputStream(),
                client.getOutputStream());
            client.close();
        }
    }

    ServerSocket serverSocket( int port) {
        return new ServerSocket(port);
    }

    etc.
```

TestServer

```
public class TestServer extends SDWitterServer {
    MockServerSocket testSocket;

    ServerSocket serverSocket( int port) {
        return testSocket;
    }
}
```

Other than using a different type of socket it performs the operations as the parent class

```
public class Tests extends Testcase {
    public void testLogin() {
        TestServer server = new TestServer();
        server.testSocket = new MockServerSocket("client command to login");
        server.run();
        assertTrue(server.testSocket.serverResponse() = "the correct response here");
    }
}
```


The situation is more complex than shown here. I simplified it to avoid details that might confuse people.

MockServerSocket

Returns a fake (Mock) client connection

Fakes client connection

- Does not use network
- Contains fixed requests
- Records server responses

Effective Java

Effective Java

Book by Joshua Bloch

First Edition 2001
Second Edition 2008

Item 1. Consider Static Factory methods

Consider using static Factory methods instead of constructors

Java String class

```
public static String valueOf(boolean b) {
    return b ? "true" : "false";
}

public static String valueOf(char c) {
    char data[] = {c};
    return new String(data, true);
}
```

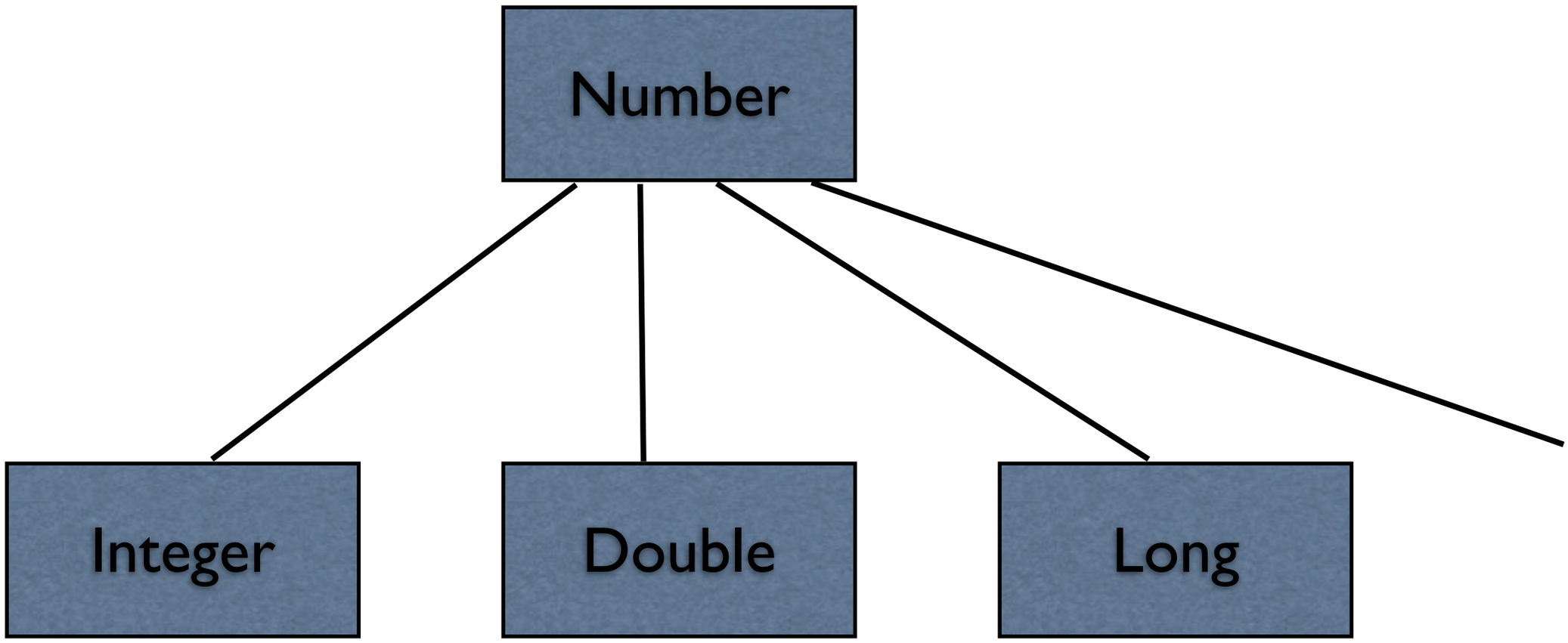
Advantages of Static Factory methods

They have names

Don’t need to create a new object each time

They can return an Object of any type

Java Boxing of Primitives




```
Integer x = new Integer( 5);
Boolean y = new Boolean( true);
```

Objective C Boxing of Primitives

Uses static factory methods in Number

```
Number x = Number.value(5);
Number y = Number.value(true);
```

Programmers only need to know Number class

Class Cluster

Smalltalk

No constructors

Just static factory methods

Item 12 Minimize accessibility

Rule of thumb

Make each class or member as inaccessible as possible

Item 13 Favor Immutability

Immutable objects are simple

Immutable objects are thread-safe

Immutable objects can be shared freely

Immutable objects are good building blocks for other objects

Item 13 Favor Immutability

Don’t provide any methods that modify the object (setters)

Ensure that no methods may be overridden

Make all fields final

Make all fields private

Ensure exclusive use to any mutable components

Make defensive copies of data provided/given to client

Item 24 Make Defensive Copies when Needed

```
public final class Period {
    private final Date start;
    private final Date end;

    public Period(Date start, Date End) {
        if (start.compareTo(end) > 0 )
            throw new IllegalArgumentException(start + " is after " + end);
        this.start = start;
        this.end = end;
    }

    public Date start() {
        return start;
    }
```

Item 24 Make Defensive Copies when Needed

```
public final class Period {
    private final Date start;
    private final Date end;

    public Period(Date start, Date End) {
        this.start = new Date(start.getTime());
        this.end = new Date(end.getTime());
        if (this.start.compareTo(this.end) > 0 )
            throw new IllegalArgumentException(start + “ is after “ + end);
    }

    public Date start() {
        return start.clone();
    }
```

Item 14 Favor Composition over Inheritance

Inheritance breaks encapsulation

- Safe to use inheritance when
  - Superclass and subclass in same package
  - When superclass is designed for inheritance

Item 16 Prefer Interfaces to Abstract Classes

Existing classes can be modified to implement a new interface

Interfaces are ideal for defining mixins

Interfaces allow construction of nonhierarchical frameworks

Provide skeletal implementation class to go with nontrivial interface

Item 30 Know and use the Libraries

Item 32 Avoid strings if other types are better

```
String compoundKey = name + “#” + i.next();
```

What happens if “#” is in name?

Create CompoundKey class

Item 34 Refer to objects by their Interfaces

Your code will be more flexible

```
List subscribers = new Vector();
```

```
Vector subscribers = new Vector();
```

If no interface exists then ok to refer to object via class

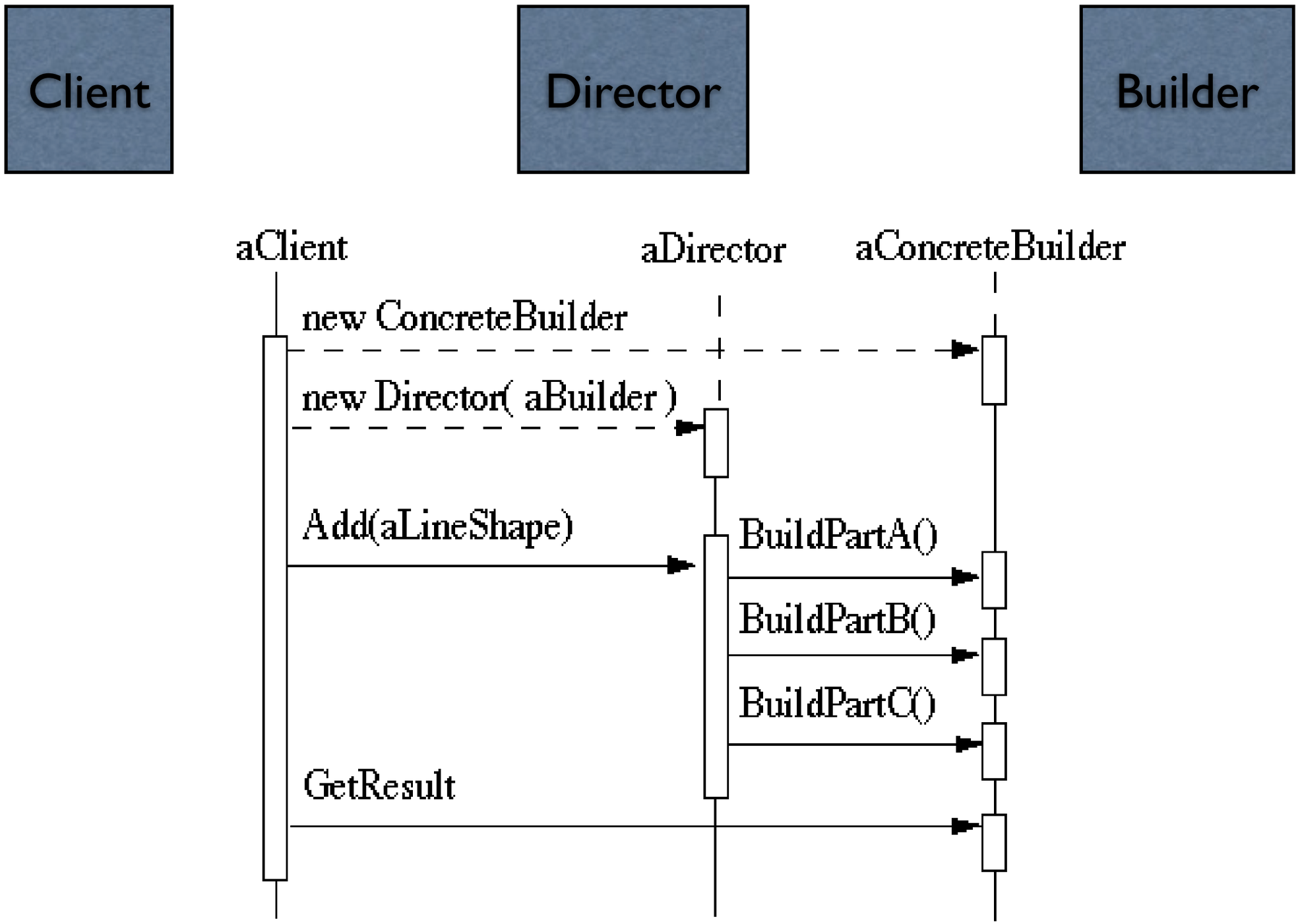
Builder

Builder

Separate construction of a complex object from its representation

So same construction process can create different representations

Builder


Client
Director
Builder
aClient
aDirector
aConcreteBuilder
new ConcreteBuilder
new Director(aBuilder)
Add(aLineShape)
BuildPartA()
BuildPartB()
BuildPartC()
GetResult

RTF Converter

A word processing document has complex structure

How to convert Rich Text Format (RTF) to

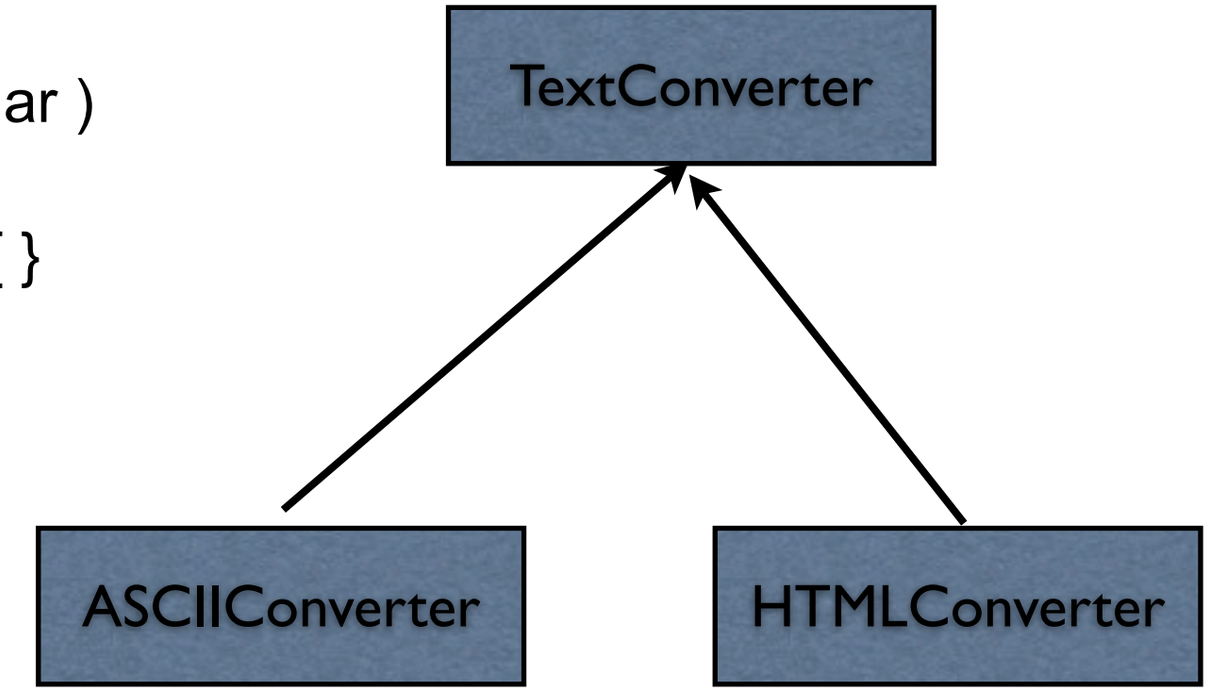
TeX
html
PDF
etc.

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 atts)

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.

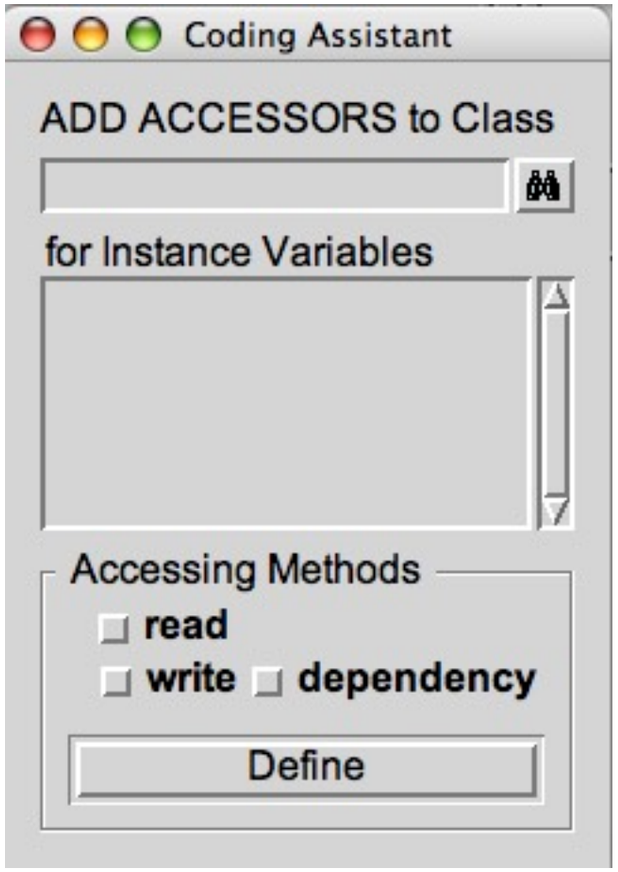
Simple API XML (SAX)

```
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;
```

```
public Person(
      final String newLastName,
      final String newFirstName,
      final String newMiddleName,
      final String newSalutation,
      final String newSuffix,
      final String newStreetAddress,
     ...
      final boolean newIsEmployed) {
                this.lastName = newLastName;
            this.firstName = newFirstName;
          ...
}
```


http://www.javaworld.com/article/2074938/core-java/too-many-parameters-in-java-methods-part-3-builder-pattern.html

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();
```

Java JSON

```
import javax.json.Json;
import javax.json.JsonObject;
...
JsonObject model = Json.createObjectBuilder()
   .add("firstName", "Duke")
   .add("lastName", "Java")
   .add("age", 18)
   .add("streetAddress", "100 Internet Dr")
   .add("city", "JavaTown")
   .add("state", "JA")
   .add("postalCode", "12345")
   .add("phoneNumbers", Json.createArrayBuilder()
      .add(Json.createObjectBuilder()
         .add("type", "mobile")
         .add("number", "111-111-1111"))
      .add(Json.createObjectBuilder()
         .add("type", "home")
         .add("number", "222-222-2222")))
   .build();
```


http://docs.oracle.com/javaee/7/tutorial/jsonp003.htm