

Design Patterns as Higher-Order Datatype-Generic Programs



Jeremy Gibbons

University of Oxford

London HUG, November 2007

1. Summary

Design patterns are typically expressed *informally*, using *prose, pictures and prototypes*.

That's all we can do with today's mainstream languages.

But with tomorrow's languages, we will be able to capture patterns as *reusable library code*.

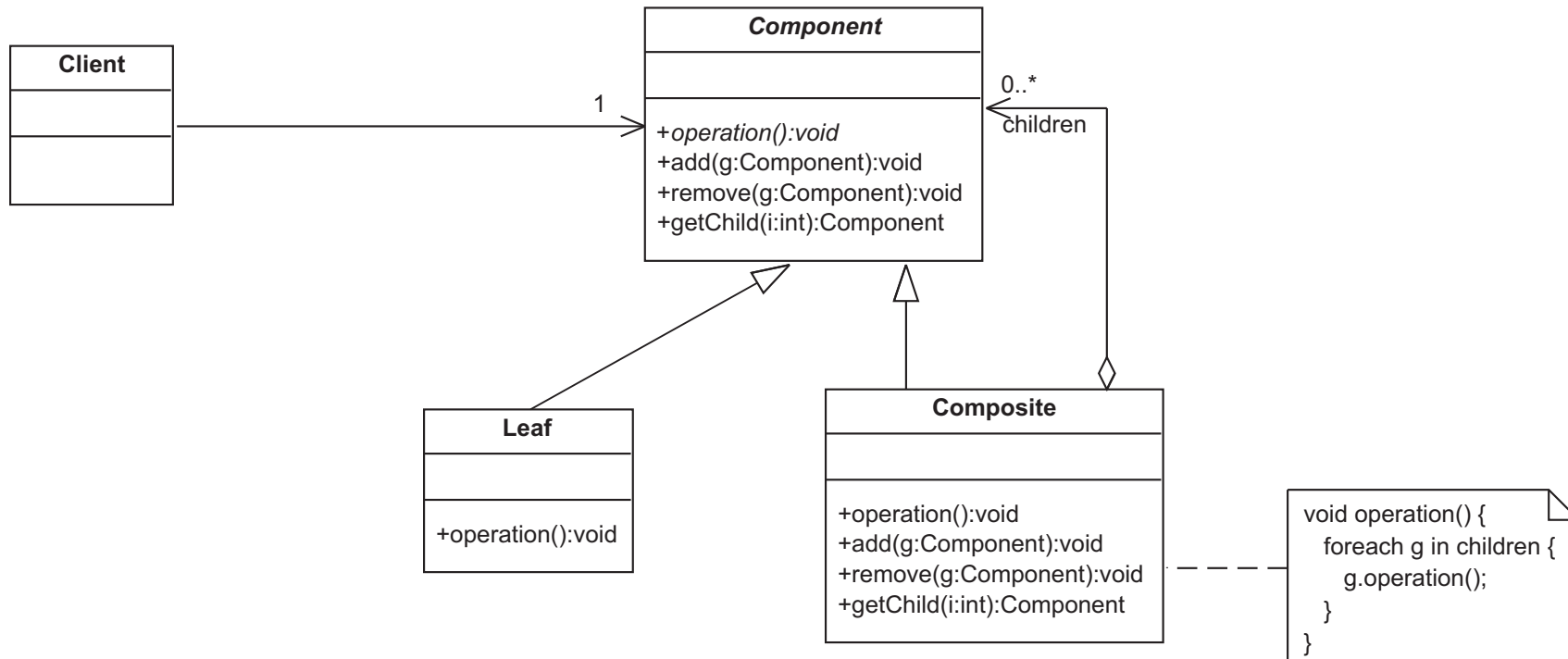
The features we need are

- *higher-order functions* (parametrization by programs), and
- *datatype-genericity* (parametrization by datatypes).

2. Design patterns

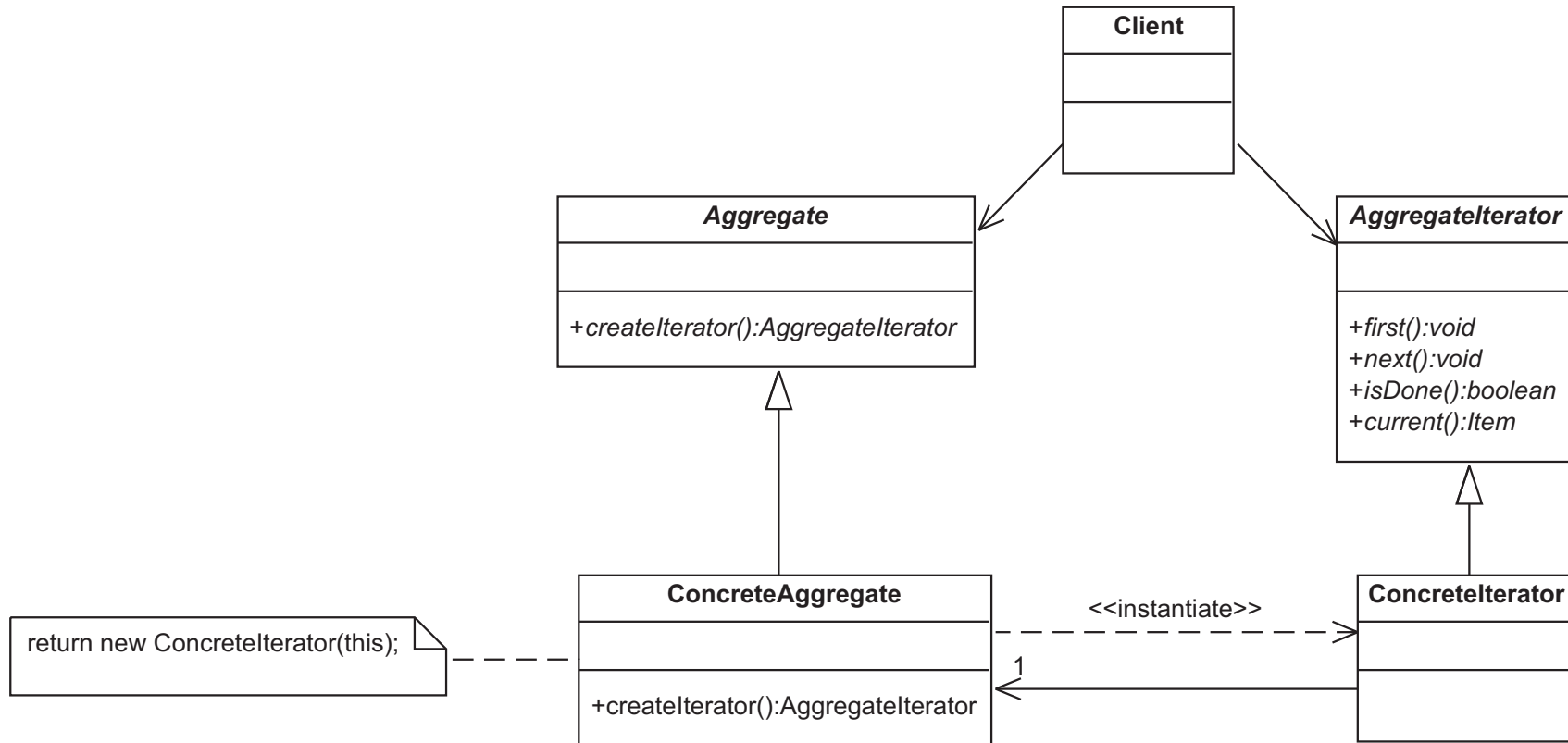
- good ideas in (OO) software design
- reusable micro-architectures
- popularized by the Gang of Four (GoF)
- up-front hooks, or post-hoc refactorings
- meta-linguistic

2.1. Composite



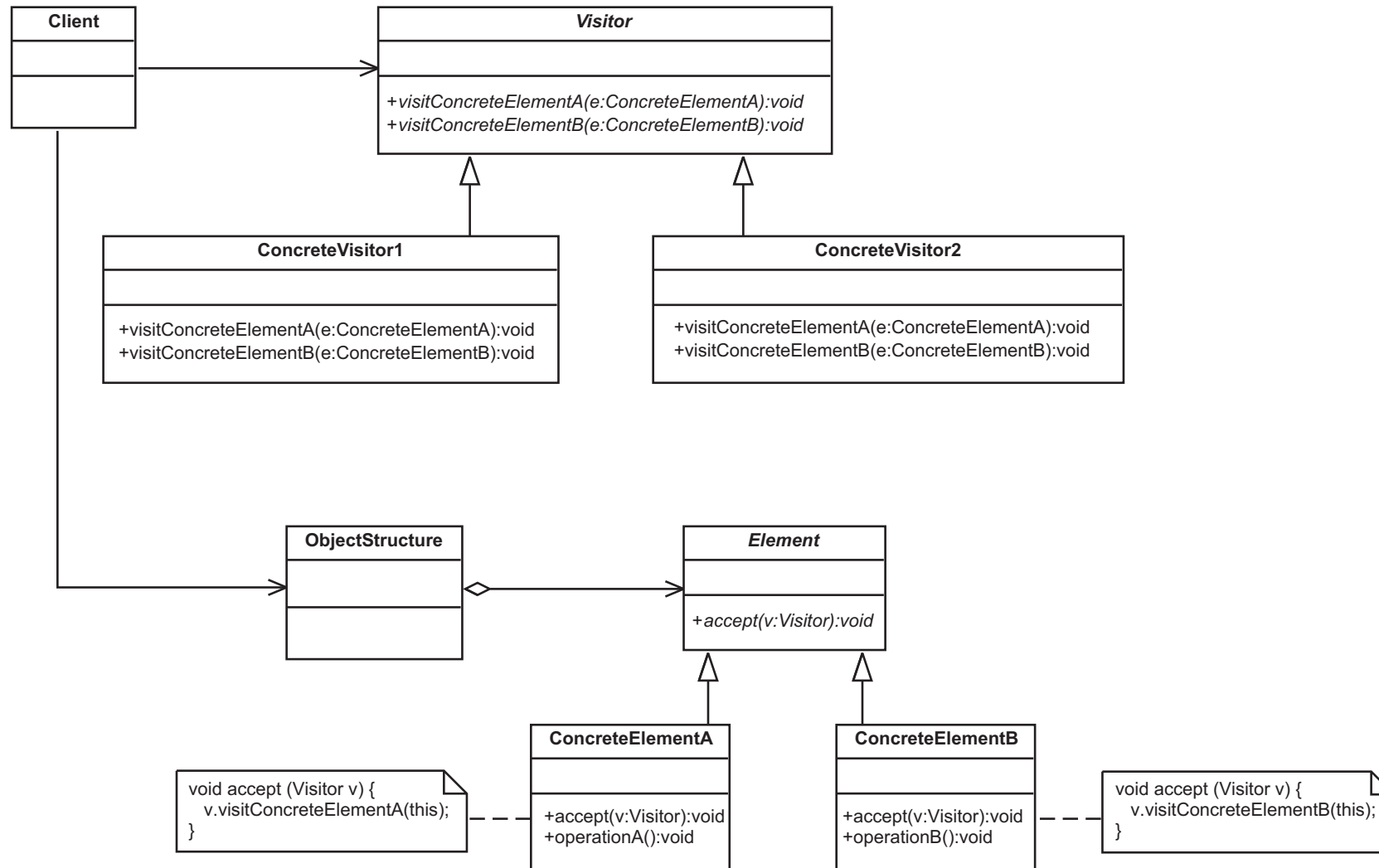
Treat atoms and aggregates the same; hence hierarchical structures.

2.2. Iterator



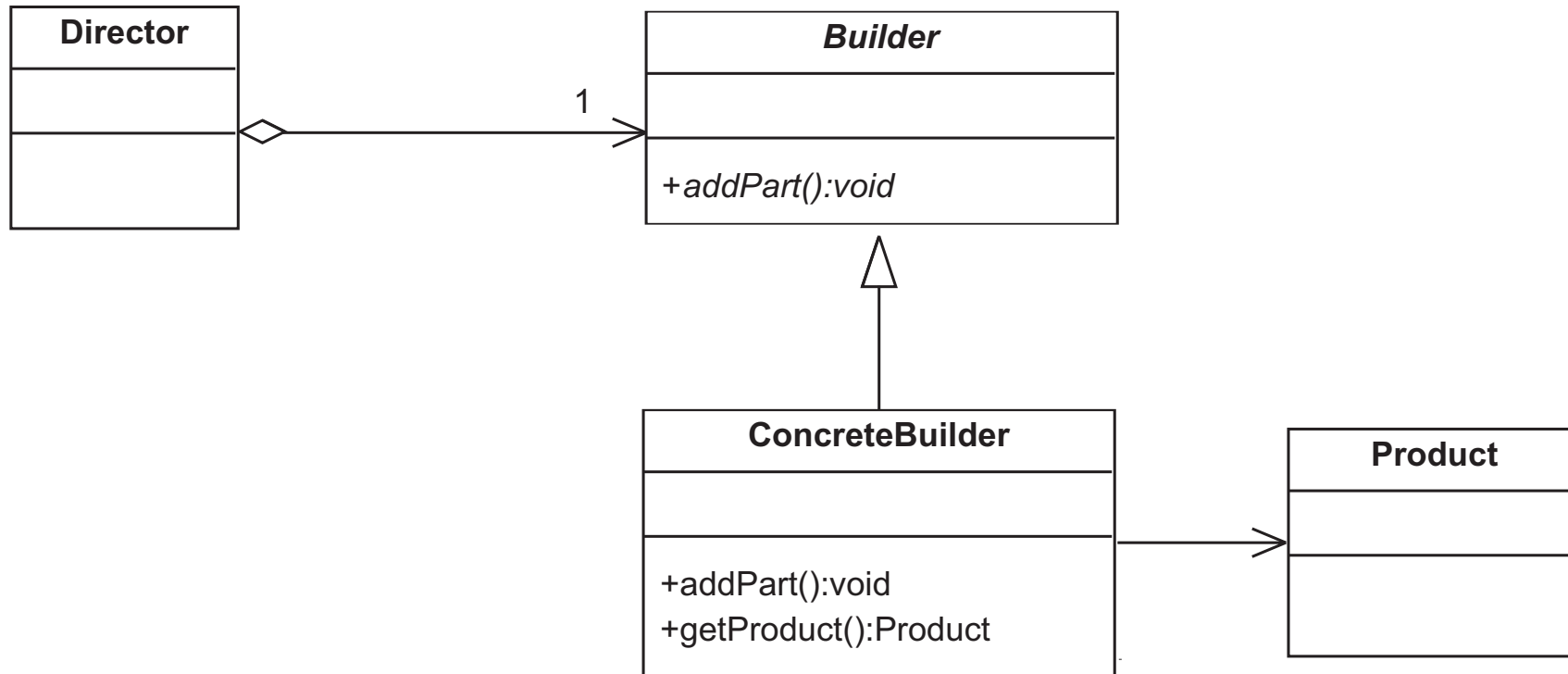
Separate responsibilities of containment and traversal.

2.3. Visitor



Hook for associating new traversals.

2.4. Builder



Separate what to build from how to build it.

3. Datatype-genericity

- *higher-order*: programs parametrized by programs
- *polymorphic*: programs parametrized by types
- *(datatype-)generic*: programs parametrized by datatypes (type constructors)
- *origami programming* (using folds, unfolds, etc) is based on higher-order polymorphic datatype-generic recursion operators

3.1. First-order, monomorphic

data *ListI* = *NilI* | *ConsI Integer ListI*

data *ListC* = *NilC* | *ConsC Char ListC*

sumI :: *ListI* → *Integer*

sumI NilI = 0

sumI (ConsI x xs) = *x* + *sumI xs*

appendI :: *ListI* → *ListI* → *ListI*

appendI NilI ys = *ys*

appendI (ConsI x xs) ys = *ConsI x (appendI xs ys)*

appendC :: *ListC* → *ListC* → *ListC*

appendC NilC ys = *ys*

appendC (ConsC x xs) ys = *ConsC x (appendC xs ys)*

3.2. First-order, polymorphic but datatype-specific

Abstract from over-specific type parameter.

data *List a = Nil | Cons a (List a)*

sum :: List Integer → Integer

sum Nil = 0

sum (Cons x xs) = x + sum xs

append :: List a → List a → List a

append Nil ys = ys

append (Cons x xs) ys = Cons x (append xs ys)

concat :: List (List a) → List a

concat Nil = Nil

concat (Cons xs xss) = append xs (concat xss)

3.3. Higher-order, list-specific

Identify recurring pattern, encapsulate as higher-order function.

$$\textit{foldL} :: b \rightarrow (a \rightarrow b \rightarrow b) \rightarrow \textit{List} a \rightarrow b$$
$$\textit{foldL} n c \textit{Nil} = n$$
$$\textit{foldL} n c (\textit{Cons} x xs) = c x (\textit{foldL} n c xs)$$
$$\textit{sum} = \textit{foldL} 0 (+)$$
$$\textit{append} xs ys = \textit{foldL} ys \textit{Cons} xs$$
$$\textit{concat} = \textit{foldL} \textit{Nil} \textit{append}$$

‘Replace constructors with supplied functions’.

3.4. Higher-order, tree-specific

Similarly for a pattern of computation over binary trees.

data *Btree* *a* = *Tip* *a* | *Bin* (*Btree* *a*) (*Btree* *a*)

foldB :: (*a* → *b*) → (*b* → *b* → *b*) → *Btree* *a* → *b*

foldB *t b* (*Tip* *x*) = *t x*

foldB *t b* (*Bin* *xs ys*) = *b* (*foldB* *t b* *xs*) (*foldB* *t b* *ys*)

reverse :: *Btree* *a* → *Btree* *a*

reverse = *foldB* *Tip nib* **where** *nib* *xs ys* = *Bin* *ys xs*

flatten :: *Btree* *a* → *List* *a*

flatten = *foldB* *wrap append* **where** *wrap* *x* = *Cons* *x Nil*

3.5. Datatype-generic

- higher-order features allow some recurring patterns to be captured
- commonality of program shape, variability in the details
- pattern of program shape expressed as a higher-order function; details encapsulated as function parameters
- eg capturing the commonality between *sum*, *append* and *concat* in terms of *foldL*
- somewhat analogous to the TEMPLATE METHOD design pattern
- *datatype-genericity* allows the capture of recurring patterns in *programs of different shapes*
- eg capturing the commonality between *foldL* and *foldB*

$$\textit{fold} :: \textit{Bifunctor } s \Rightarrow (s \ a \ b \rightarrow b) \rightarrow \textit{Fix } s \ a \rightarrow b$$

3.6. Datatype-genericity vs traditional ‘generics’

- Ada, C++, C#, Java all have ‘generics’
- largely a mechanism for parametric polymorphism (eg *List Integer*)
- C++ ‘generic programming’ as in STL is parametric polymorphism plus higher-order functions (eg sorting by supplied ordering)
- datatype-genericity is a different kind of parametrization
- some progress using C++ templates (eg Alexandrescu’s *Modern C++ Design*: generic ABSTRACT FACTORY)
- ...but datatype-genericity is currently on the boundary of expressibility

4. Origami programming

The shape of data determines the shape of the program.

So abstract from this shape.

```
data Fix s a = In (s a (Fix s a))
```

```
out :: Fix s a → s a (Fix s a)
```

```
out (In x) = x
```

or equivalently

```
data Fix s a = In { out :: s a (Fix s a) }
```

4.1. Specific shapes

The parameter *s* determines the shape; '*Fix*' ties the recursive knot.

Here are three instances of *Fix* using different shapes.

```
data ListF a b = NilF | ConsF a b
```

```
type List a = Fix ListF a
```

```
data TreeF a b = EmptyF | NodeF a b b
```

```
type Tree a = Fix TreeF a
```

```
data BtreeF a b = TipF a | BinF b b
```

```
type Btree a = Fix BtreeF a
```

4.2. Bifunctors

Restrict attention to shape parameters supporting a *bimap* operation to locate their elements.

class *Bifunctor* *s* **where**

bimap :: (a → c) → (b → d) → s a b → s c d

Technically speaking, *bimap* should satisfy functorial properties:

bimap id id = id

bimap f g · *bimap* h j = *bimap* (f · h) (g · j)

4.3. Bifunctor instances

instance *Bifunctor ListF* **where**

bimap f g NilF = *NilF*

bimap f g (ConsF x y) = *ConsF (f x) (g y)*

instance *Bifunctor BtreeF* **where**

bimap f g (TipF x) = *TipF (f x)*

bimap f g (BinF y z) = *BinF (g y) (g z)*

instance *Bifunctor TreeF* **where**

bimap f g EmptyF = *EmptyF*

bimap f g (NodeF x y z) = *NodeF (f x) (g y) (g z)*

4.4. Origami operators

map :: Bifunctor *s* ⇒ (*a* → *b*) → Fix *s* *a* → Fix *s* *b*

map f = In · bimap *f* (map *f*) · out

fold :: Bifunctor *s* ⇒ (*s* *a* *b* → *b*) → Fix *s* *a* → *b*

fold f = *f* · bimap id (fold *f*) · out

unfold :: Bifunctor *s* ⇒ (*b* → *s* *a* *b*) → *b* → Fix *s* *a*

unfold f = In · bimap id (unfold *f*) · *f*

hylo :: Bifunctor *s* ⇒ (*b* → *s* *a* *b*) → (*s* *a* *c* → *c*) → *b* → *c*

hylo f g = *g* · bimap id (hylo *f g*) · *f*

build :: Bifunctor *s* ⇒ (∀ *b*. (*s* *a* *b* → *b*) → *b*) → Fix *s* *a*

build f = *f* In

— short, sweet and symmetric.

4.5. Parametricity vs ad-hockery

This *bimap* is ad-hoc datatype-generic: the *Bifunctor* instances depend on the shape, must be given (or derived) for each new datatype, and entail a proof obligation.

But given *bimap* and its properties, *fold* etc are parametric: *higher-order natural transformations*.

A little bit of ad-hockery goes a long way.

5. Inside every pattern is a HODGP

My thesis:

a number of GoF patterns can be expressed formally,
as *higher-order datatype-generic programs*,
rather than informally as prose and pictures

Higher-order and datatype-generic features provide great expressivity.

I don't mean that mainstream programmers should give up Java and switch to Haskell.

But I do hope that tomorrow's mainstream languages will combine the benefits of OO and of HODGP languages.

5.1. Composite in HODGP

COMPOSITES are just recursive data structures.

```
data Fix s a = In { out :: s a (Fix s a) }
```

These come essentially for free in functional programming languages.

5.2. Visitor in HODGP

In the normal OO paradigm, the definition of each traversal is spread across the whole hierarchy.

Hence easy to add new variants, but hard to add new traversals.

VISITOR collects fragments of each traversal into one aspect, and provides a hook for performing such traversals. This reverses the costs: easy to add new traversals, but hard to add new variants.

This matches the normal FP paradigm. Here, no explicit hook is needed; the connection is made by pattern matching.

$$\begin{aligned} \textit{fold} &:: \textit{Bifunctor } s \Rightarrow (s \ a \ b \rightarrow b) \rightarrow \textit{Fix } s \ a \rightarrow b \\ \textit{fold } f &= f \cdot \textit{bimap } \textit{id} \ (\textit{fold } f) \cdot \textit{out} \end{aligned}$$

5.3. Iterator in HODGP

Standard external (client-driven) ITERATORS give sequential access to elements of collection.

The functional approach would be to provide a view of the collection as a list.

contents :: Bifunctor s => (s a (List a) -> List a) -> Fix s a -> List a
contents combiner = fold combiner

With lazy evaluation, the list can be generated incrementally on demand.

Of course, external ITERATORS can already be expressed linguistically; eg *java.util.Iterator*. But still, one has to program the iterator for each new datatype.

5.4. Internal iterator in HODGP

GOF discuss internal (iterator-driven) ITERATORS too.

```
public interface Action { void apply (Object o); }
```

```
public interface InternalIterator { void go (Action a); }
```

Internal ITERATORS are less flexible than external (for example, for two linked iterations over the same collection), but simpler.

An internal ITERATOR is basically a map.

In HODGP, we can give a *single generic* definition of this.

$$\text{map} :: \text{Bifunctor } s \Rightarrow (a \rightarrow b) \rightarrow \text{Fix } s \ a \rightarrow \text{Fix } s \ b$$
$$\text{map } f = \text{In} \cdot \text{bimap } f \ (\text{map } f) \cdot \text{out}$$

(There's a fuller story involving *idioms*.)

5.5. Builder in HODGP

GOF motivating example involves building a product that is basically a simple collection; but they also suggest the possibility of building a more structured product.

The standard protocol involves a *Director* adding *Parts* to a *Builder*, and finally retrieving a *Product*. Lazily, we can construct the *Product* incrementally — in a regular fashion:

$$\begin{aligned} \text{unfold} &:: \text{Bifunctor } s \Rightarrow (b \rightarrow s \ a \ b) \rightarrow b \rightarrow \text{Fix } s \ a \\ \text{unfold } f &= \text{In} \cdot \text{bimap } \text{id} \ (\text{unfold } f) \cdot f \end{aligned}$$

or without enforcing this regularity:

$$\begin{aligned} \text{build} &:: \text{Bifunctor } s \Rightarrow (\forall b. (s \ a \ b \rightarrow b) \rightarrow b) \rightarrow \text{Fix } s \ a \\ \text{build } f &= f \ \text{In} \end{aligned}$$

5.6. Computing builders

GOF also suggest the possibility of BUILDERS that compute. Instead of constructing a large *Product* and eventually collapsing it, make the *Product* itself be the collapsed result, and compute it while building.

Lazily, there is no pressing need to fuse building with processing.

Alternatively, BUILDER pattern is unfold with subsequent fold, and built-in deforestation: a hylomorphism.

$$\begin{aligned} \text{hylo} &:: \text{Bifunctor } s \Rightarrow (b \rightarrow s a b) \rightarrow (s a c \rightarrow c) \rightarrow b \rightarrow c \\ \text{hylo } f \ g &= g \cdot \text{bimap id (hylo } f \ g) \cdot f \end{aligned}$$

or irregularly, using fold-build fusion:

$$\begin{aligned} \text{foldBuild} &:: \text{Bifunctor } s \Rightarrow (\forall b. (s a b \rightarrow b) \rightarrow b) \rightarrow (s a b \rightarrow b) \rightarrow b \\ \text{foldBuild } f \ g &= f \ g \end{aligned}$$

6.1. The document example as a HODGP

```

data DocF a b = Para a | Sec String [ b]
type Doc = Fix DocF String
instance Bifunctor DocF where
    bimap f g (Para s) = Para (f s)
    bimap f g (Sec s xs) = Sec s (map g xs)

corrector :: Doc → Doc
corrector = map correct

printDoc :: Doc → [String]
printDoc = fold combine where
    combine (Para s) = [s]
    combine (Sec s xs) = s : concat xs

data XML = Text String
          | Entity Tag Attrs [XML]
type Tag = String
type Attrs = [(String, String)]

```

```

fromXML :: XML → Doc
fromXML = unfold step where
    step (Text s) = Para s
    step (Entity t kvs xs) = Sec (title t kvs) xs

title :: Tag → Attrs → String
title t [] = t
title t kvs = t ++ paren (join (map attr kvs)) where
    paren s = " (" ++ s ++ ")"
    join [s] = s
    join (s : ss) = s ++ ", " ++ join ss
    attr (k, v) = k ++ "=" ++ v ++ "'"

printXML :: XML → [String]
printXML = hylo step combine

buildDoc :: (DocF String b → b) → b
buildDoc f = f (Sec "Heading" [f (Para "p1"),
                                f (Para "p2")])

myDoc :: Doc
myDoc = build buildDoc

printMyDoc :: [String]
printMyDoc = buildDoc combine

```

7. The document example, in Java

```
public interface Component {  
    void accept (Visitor v);  
    Iterator getIterator ();  
}  
  
public class Paragraph implements Component {  
    protected String body;  
    public Paragraph (String body) {  
        setBody (body);  
    }  
    public void setBody (String s) {  
        body = s;  
    }  
    public String getBody () {  
        return body;  
    }  
    public Iterator getIterator () {  
        return new ParagraphIterator (this);  
    }  
    public void accept (Visitor v) {  
        v.visitParagraph (this);  
    }  
}
```

```
import java.util.Vector;  
import java.util.Enumeration;  
public class Section implements Component {  
    protected Vector children;  
    protected String title;  
    public Section (String title) {  
        children = new Vector ();  
        this.title = title;  
    }  
    public String getTitle () {  
        return title;  
    }  
    public void addComponent (Component c) {  
        children.addElement (c);  
    }  
    public Enumeration getChildren () {  
        return children.elements ();  
    }  
    public Iterator getIterator () {  
        return new SectionIterator (this);  
    }  
    public void accept (Visitor v) {  
        v.visitSection (this);  
    }  
}
```

7.1. ... continued...

```
public interface Iterator {
    void iterate (Action a);
}

import java.util.Enumeration;
public class SectionIterator implements Iterator {
    protected Section s;
    public SectionIterator (Section s) {
        this.s = s;
    }
    public void iterate (Action a) {
        for (Enumeration e = s.getChildren ();
            e.hasMoreElements ();) {
            ((Component) (e.nextElement ())).
                getIterator ().iterate (a);
        }
    }
}
```

```
public class ParagraphIterator implements Iterator {
    protected Paragraph p;
    public ParagraphIterator (Paragraph p) {
        this.p = p;
    }
    public void iterate (Action a) {
        a.apply (p);
    }
}

public interface Action {
    void apply (Paragraph p);
}

public class SpellCorrector implements Action {
    public void apply (Paragraph p) {
        p.setBody (correct (p.getBody ()));
    }
    public String correct (String s) {
        return s.toLowerCase ();
    }
}
```

7.2. ... continued...

```

public interface Visitor {
    void visitParagraph (Paragraph p);
    void visitSection (Section s);
}

import java.util.Enumeration;
import java.util.Vector;
public class PrintVisitor implements Visitor {

    protected String indent = "";
    protected Vector lines = new Vector ();

    public String [] getResult () {
        String [] ss = new String [0];
        ss = (String []) lines.toArray (ss);
        return ss;
    }

    public void visitParagraph (Paragraph p) {
        lines.addElement (indent + p.getBody ());
    }
}

```

```

public void visitSection (Section s) {
    String currentIndent = indent;
    lines.addElement (indent + s.getTitle ());
    for (Enumeration e = s.getChildren ();
        e.hasMoreElements ();) {
        indent = currentIndent + "  ";
        ((Component) e.nextElement ()).accept (this);
    }
    indent = currentIndent;
}

public interface Builder {
    int addParagraph (String body, int parent)
        throws InvalidBuilderId;
    int addSection (String title, int parent)
        throws InvalidBuilderId;
}

public class InvalidBuilderId extends Exception {
    public InvalidBuilderId (String reason) {
        super (reason);
    }
}

```

7.3. ... continued...

```

import java.util.AbstractMap;
import java.util.HashMap;
public class ComponentBuilder implements Builder {
    protected int nextId = 0;
    protected AbstractMap comps = new HashMap ();
    public int addParagraph (String body, int pId)
        throws InvalidBuilderId {
        return addComponent (new Paragraph (body), pId);
    }
    public int addSection (String title, int pId)
        throws InvalidBuilderId {
        return addComponent (new Section (title), pId);
    }
    public Component getProduct () {
        return (Component) comps.get (new Integer (0));
    }
    protected int addComponent (Component c, int pId)
        throws InvalidBuilderId {
        if (pId < 0) {
            if (comps.isEmpty ()) {
                comps.put (new Integer (nextId), c);
                return nextId++;
            }

```

```

        else
            throw new InvalidBuilderId
                ("Duplicate root");
    } else {
        Component parent = (Component) comps.
            get (new Integer (pId));
        if (parent == null) {
            throw new InvalidBuilderId
                ("Non-existent parent");
        } else {
            if (parent instanceof Paragraph) {
                throw new InvalidBuilderId
                    ("Adding child to paragraph");
            } else {
                Section s = (Section) parent;
                s.addComponent (c);
                comps.put (new Integer (nextId), c);
                return nextId++;
            }
        }
    }
}
}
}
}

```

7.4. ... continued...

```
import java.util.Vector;
public class PrintBuilder implements Builder {
    protected class Record {
        public int id;
        public int last;
        public String line;
        public String indent;
        public Record (int id, int last,
                       String line, String indent) {
            this.id = id;
            this.last = last;
            this.line = line;
            this.indent = indent;
        }
    }
    protected Vector records = new Vector ();
    protected Record recordAt (int i) {
        return (Record) records.elementAt (i);
    }
}
```

```
protected int find (int id, int start) {
    while (start < records.size () &&
           recordAt (start).id != id)
        start++;
    if (start < records.size ())
        return start;
    else
        return - 1;
}
protected int nextId = 0;
protected SpellCorrector c = new SpellCorrector ();
public int addParagraph (String body, int pid)
    throws InvalidBuilderId {
    return addComponent (c.correct (body), pid);
}
public int addSection (String title, int pid)
    throws InvalidBuilderId {
    return addComponent (title, pid);
}
```

7.5. ... continued...

```

public String [] getProduct () {
    String [] ss = new String [ records.size ()];
    for (int i = 0; i < ss.length; i++)
        ss [i] = recordAt (i).indent + recordAt (i).line;
    return ss;
}

protected int addComponent (String s, int pId)
    throws InvalidBuilderId {
    if (pId < 0) {
        if (records.isEmpty ()) {
            records.addElement (new Record
                (nextId, nextId, s, ""));
            return nextId++;
        }
        else
            throw new InvalidBuilderId
                ("Duplicate root");
    }
}

```

```

} else {
    int x = find (pId, 0);
    Record r = recordAt (x);
    String indent = r.indent;
    if (x == - 1) {
        throw new InvalidBuilderId
            ("Non-existent parent");
    } else {
        int y = x;
        while (r.id != r.last) {
            y = x;
            x = find (r.last, x);
            r = recordAt (x);
        }
        records.insertElementAt (new Record
            (nextId, nextId, s, indent + "    "), x + 1);
        recordAt (y).last = nextId;
        return nextId++;
    }
}
}
}
}

```

7.6. ... concluded

```
public abstract class Main {
    public static void build (Builder b) {
        try {
            int rootId = b.addSection ("Doc", -1);
            int sectId = b.addSection ("Sec 1", rootId);
            int subsId = b.addSection ("Subsec 1.1", sectId);
            int id = b.addParagraph ("Para 1.1.1", subsId);
            id = b.addParagraph ("Para 1.1.2", subsId);
            subsId = b.addSection ("Subsec 1.2", sectId);
            id = b.addParagraph ("Para 1.2.1", subsId);
            id = b.addParagraph ("Para 1.2.2", subsId);
            sectId = b.addSection ("Sec 2", rootId);
            subsId = b.addSection ("Subsec 2.1", sectId);
            id = b.addParagraph ("Para 2.1.1", subsId);
            id = b.addParagraph ("Para 2.1.2", subsId);
            subsId = b.addSection ("Subsec 2.2", sectId);
            id = b.addParagraph ("Para 2.2.1", subsId);
            id = b.addParagraph ("Para 2.2.2", subsId);
        } catch (InvalidBuilderId e) {
            System.out.println ("Exception: " + e);
        }
    }
}
```

```
public static void main (String [ ] args) {
    String [ ] lines;
    if (false) {
        ComponentBuilder b = new ComponentBuilder ();
        build (b);
        Component root = b.getProduct ();
        root.getIterator ().iterate (new SpellCorrector ());
        PrintVisitor pv = new PrintVisitor ();
        root.accept (pv);
        lines = pv.getResult ();
    } else {
        PrintBuilder b = new PrintBuilder ();
        build (b);
        lines = b.getProduct ();
    }
    for (int i = 0; i < lines.length; i++)
        System.out.println (lines [ i]);
}
```

8. Conclusion

- design patterns are usually expressed informally
- given the right language features, they could be library code
- *higher-order functions* and *datatype-genericity* are what is needed
- these features are familiar in the FP world
- I hope to see them soon in the OO world!