Design Patterns as Higher-Order Datatype-Generic Programs

Jeremy Gibbons
Oxford University Computing Laboratory
http://www.comlab.ox.ac.uk/jeremy.gibbons/

Abstract

Design patterns are reusable abstractions in object-oriented software. However, using current mainstream programming languages, these elements can only be expressed extra-linguistically: as prose, pictures, and prototypes. We believe that this is not inherent in the patterns themselves, but evidence of a lack of expressivity in the languages of today. We expect that, in the languages of the future, the code parts of design patterns will be expressible as reusable library components. Indeed, we claim that the languages of tomorrow will suffice; the future is not far away. All that is needed, in addition to commonly-available features, are *higher-order* and *datatype-generic* constructs; these features are already or nearly available now. We argue the case by presenting higher-order datatype-generic programs capturing ORIGAMI, a small suite of patterns for recursive data structures.

Categories and Subject Descriptors F.3.3 [*Logics and meanings of programs*]: Studies of program constructs—object-oriented constructs; D.3.3 [*Programming languages*]: Language constructs and features—Patterns, polymorphism, control structures, recursion; D.3.2 [*Programming languages*]: Language classifications—Functional languages, design languages, object-oriented languages.

General Terms Languages, Design, Algorithms, Theory.

Keywords Design patterns, generic programming, higher-order functions, functional programming, folds, unfolds.

1. Introduction

Design patterns, as the subtitle of the seminal book [11] has it, are 'elements of reusable object-oriented software'. However, within the confines of existing mainstream programming languages, these supposedly reusable elements can only be expressed extra-linguistically: as prose, pictures, and prototypes. We believe that this is not inherent in the patterns themselves, but evidence of a lack of expressivity in the languages of today. We expect that, in the languages of the future, the code parts of design patterns will be expressible as directly-reusable library components. The benefits will be considerable: patterns may then be reasoned about, type-checked, applied and reused, just as any other abstractions can.

Indeed, we claim that the languages of tomorrow will suffice; the future is not far away. All that is needed, in addition to what is provided by essentially every programming language, are *higher-order* (parametrization by code) and *datatype-generic* (parametrization by type constructor) features. Higher-order constructs have been available for decades in functional programming languages such as ML [37] and Haskell [42]. Datatype genericity can be simulated in existing programming languages [7, 24, 39], but we already have significant experience with robust prototypes of languages that support it natively [25, 33].

We argue our case by capturing as higher-order datatype-generic programs a small subset ORIGAMI of the Gang of Four (GOF) patterns. (For the sake of rhetorical style, we equate 'GOF patterns' with 'design patterns'.) These programs are parametrized along three dimensions: by the *shape* of the computation, which is determined by the shape of the underlying data, and represented by a type constructor (an operation on types); by the *element type* (a type); and by the *body* of the computation, which is a higher-order argument (a value, typically a function).

Although our presentation is in a functional programming style, we do not intend to argue that functional programming is the paradigm of the future (whatever we might feel personally!). Rather, we believe that functional programming languages are a suitable test-bed for experimental language features — as evidenced by parametric polymorphism and list comprehensions, for example, which are both now finding their way into mainstream programming languages such as Java and C#. We expect that the evolution of programming languages will continue to follow the same trend: experimental language features will be developed and explored in small, nimble laboratory languages, and the successful experiments will eventually make their way into the outside world. Specifically, we expect that the mainstream languages of tomorrow will be broadly similar to the languages of today — strongly and statically typed, object-oriented, with an underlying imperative mindset — but incorporating additional features from the functional world — specifically, higher-order operators and datatype genericity.

2. Parametrization

We start with a brief review of the kinds of parametrization required to express design patterns as programs: as the title of the paper suggests, the necessary features are *higher-order* and *datatype-generic* constructs. We then present a little suite of well-known higher-order datatype-generic recursion operators: folds, unfolds, and the like. These operators turn out, we claim, to capture the essence of a number of familiar design patterns.

2.1 Higher order programs

Design patterns are patterns in program structure. They capture commonalities in the large-scale structure of programs, abstracting from differences in the small-scale structure. Thus, (at least the ex-

WGP'06 September 16, 2006, Portland, Oregon, USA.
ACM 1-59593-492-6/06/0009.

tensional parts of) design patterns can be seen as program schemes: operators on programs, taking small-scale program fragments as arguments and returning large-scale pattern instances as results. It is quite natural, therefore, to model design patterns as higher-order operators.

Higher-order operators, programs that take other programs as arguments or return them as results, are the focus of functional programming languages, which take the view that functions are first-class citizens, with the same rights as any other kind of data. For example, consider the following definition of a datatype *ListI* of lists of integers.

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

Various programs over this datatype have a common structure: definition by case analysis on a *ListI* argument; two clauses, one per *ListI* variant; and the only use of the tail of the analysed argument as an argument to an identical recursive call.

$$\begin{aligned}
&\mathit{sumI} :: \mathit{ListI} \to \mathit{Integer}\\
&\mathit{sumI}\ \mathit{NilI} &&= 0\\
&\mathit{sumI}\ (\mathit{ConsI}\ x\ xs) &&= x + \mathit{sumI}\ xs\\
&\mathit{appendI} :: \mathit{ListI} \to \mathit{ListI} \to \mathit{ListI}\\
&\mathit{appendI}\ \mathit{NilI} &&= ys\\
&\mathit{appendI}\ (\mathit{ConsI}\ x\ xs)\ ys &&= \mathit{ConsI}\ x\ (\mathit{appendI}\ xs\ ys)
\end{aligned}$$

Higher-order features allow the abstraction of the common pattern of computation in these two programs — in this case, as a fold.

$$\begin{aligned}
&\mathit{foldLI} :: b \to (\mathit{Integer} \to b \to b) \to \mathit{ListI} \to b\\
&\mathit{foldLI}\ n\ c\ \mathit{NilI} &&= n\\
&\mathit{foldLI}\ n\ c\ (\mathit{ConsI}\ x\ xs) &&= c\ x\ (\mathit{foldLI}\ n\ c\ xs)\\
&\mathit{sumI} &&= \mathit{foldLI}\ 0\ (+)\\
&\mathit{appendI}\ xs\ ys &&= \mathit{foldLI}\ ys\ \mathit{ConsI}\ xs
\end{aligned}$$

For more about higher-order programming, see any textbook on functional programming [41, 4].

2.2 Datatype genericity

The datatype *ListI* and the corresponding higher-order operator *foldLI* can be made more useful may making them *parametrically polymorphic*, abstracting away from the fixed element type *Integer*.

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

$$\begin{aligned}
&\mathit{foldL} :: b \to (a \to b \to b) \to \mathit{List}\ a \to b\\
&\mathit{foldL}\ n\ c\ \mathit{Nil} &&= n\\
&\mathit{foldL}\ n\ c\ (\mathit{Cons}\ x\ xs) &&= c\ x\ (\mathit{foldL}\ n\ c\ xs)
\end{aligned}$$

This kind of parametrization is sometimes called ‘generic programming’; for example, it underlies the kind of generic programming embodied in the C++ Standard Template Library [2]. It is a very well-behaved form of genericity — one can deduce properties of parametrically polymorphic programs from their types alone [45] — but by the same token it is also relatively inflexible. For example, suppose one also had a polymorphic datatype of binary trees:

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

and a corresponding fold operator:

$$\begin{aligned}
&\mathit{foldB} :: (a \to b) \to (b \to b \to b) \to \mathit{Btree}\ a \to b\\
&\mathit{foldB}\ t\ b\ (\mathit{Tip}\ x) &&= t\ x\\
&\mathit{foldB}\ t\ b\ (\mathit{Bin}\ xs\ ys) &&= b\ (\mathit{foldB}\ t\ b\ xs)\ (\mathit{foldB}\ t\ b\ ys)
\end{aligned}$$

The two higher-order, parametrically-polymorphic programs *foldL* and *foldB* have quite a lot in common: both replace constructors by supplied arguments; both have patterns of recursion that follow the datatype definition, with one clause per datatype variant and one recursive call per substructure. But neither parametric polymorphism nor higher-order functions suffice to capture this recurring pattern.

In fact, what differs between the two fold operators is the *shape* of the data on which they operate, and hence the shape of the programs themselves. The kind of parametrization required is by this shape; that is, by the datatype or type constructor (such as *List* or *Tree*) concerned. We call this *datatype genericity*; it allows the capture of recurring patterns in *programs of different shapes*. In Section 2.3 below, we explain the definition of a datatype-generic operation *fold* with the following type:

$$\mathit{fold} :: \mathit{Bifunctor}\ s \Rightarrow (s\ a\ b \to b) \to \mathit{Fix}\ s\ a \to b$$

Here, in addition to the type *a* of collection elements and the fold body (a function of type $s\ a\ b \to b$), the shape parameter *s* varies; the type class *Bifunctor* expresses the constraints we place on its choice. The shape parameter determines the shape of the input data; for one instantiation of *s*, the type *Fix s a* is isomorphic to *List a*, and for another instantiation it is isomorphic to *Btree a*. The same shape parameter also determines the type of the fold body, supplied as an argument with which to replace the constructors.

For more about datatype genericity, see [17].

2.3 Origami programming

As argued above, data structure determines program structure. It therefore makes sense to abstract from the determining shape, leaving only what they have in common. We do this by defining a datatype *Fix*, parametrized both by an element type *a* of basic kind (a plain type, such as integers or strings), and by a shape type *s* of higher kind (a type constructor, such as ‘pairs of’ or ‘lists of’).

data $\mathit{Fix}\ s\ a = \mathit{In}\{\mathit{out} :: s\ a\ (\mathit{Fix}\ s\ a)\}$

The parameter *s* determines the shape; ‘*Fix*’ ties the recursive knot. Here are three instances of *Fix* using different shapes: lists, and internally- and externally-labelled binary trees.

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*

Note that *Fix s a* is a recursive type. Typically, as in the three instances above, the shape *s* has several variants, including a ‘base case’ independent of its second argument. But with lazy evaluation, infinite structures are possible, and so the definition makes sense even with no such base case. For example, *Fix ITreeF a* with *ITreeF a b* = *INodeF a b b* is a type of infinite internally-labelled binary trees.

Not all valid binary type constructors *s* are suitable for *Fix*ing (for example, because of function types). It turns out that we should restrict attention to *bifunctors*, which support a *bimap* operation ‘locating’ all the elements. We capture this constraint as a type class.

class *Bifunctor s* **where**
  $\mathit{bimap} :: (a \to c) \to (b \to d) \to s\ a\ b \to s\ c\ d$

Technically speaking, *bimap* should satisfy some properties:

$$\begin{aligned}
&\mathit{bimap}\ \mathit{id}\ \mathit{id} &&= \mathit{id}\\
&\mathit{bimap}\ f\ g \cdot \mathit{bimap}\ h\ j &&= \mathit{bimap}\ (f \cdot h)\ (g \cdot j)
\end{aligned}$$

These cannot be expressed in Haskell — but we might expect to be able to express them in the languages of tomorrow [8, 44].

All datatypes made from sum and product constructors induce bifunctors. Here are instances for our three example shapes.

instance *Bifunctor ListF* **where**
  $\mathit{bimap}\ f\ g\ \mathit{NilF} = \mathit{NilF}$
  $\mathit{bimap}\ f\ g\ (\mathit{ConsF}\ x\ y) = \mathit{ConsF}\ (f\ x)\ (g\ y)$

instance *Bifunctor BtreeF* **where**
  $\mathit{bimap}\ f\ g\ (\mathit{TipF}\ x) = \mathit{TipF}\ (f\ x)$
  $\mathit{bimap}\ f\ g\ (\mathit{BinF}\ y\ z) = \mathit{BinF}\ (g\ y)\ (g\ z)$

instance *Bifunctor TreeF* **where**
  $\mathit{bimap}\ f\ g\ \mathit{EmptyF} = \mathit{EmptyF}$
  $\mathit{bimap}\ f\ g\ (\mathit{NodeF}\ x\ y\ z) = \mathit{NodeF}\ (f\ x)\ (g\ y)\ (g\ z)$

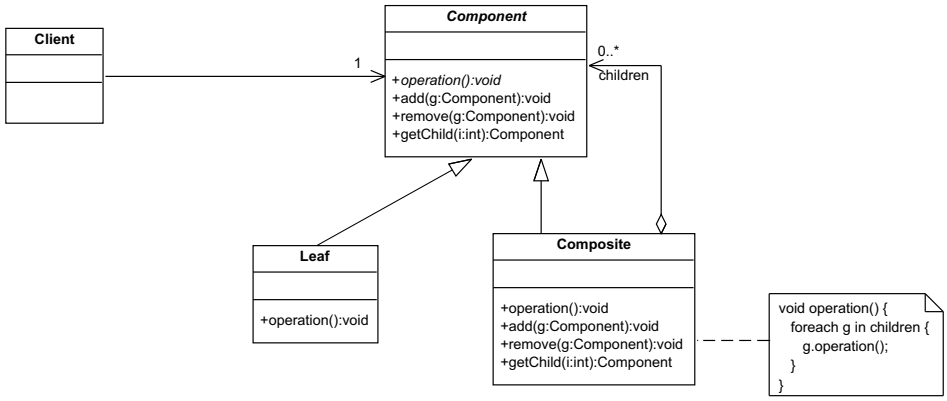

Figure 1. The class structure of the COMPOSITE pattern

The type signature of the operator *bimap* is datatype-generic, since it is parameterized by the shape s of the data:

$$bimap :: Bifunctor\ s \Rightarrow (a \to c) \to (b \to d) \to s\ a\ b \to s\ c\ d$$

However, because *bimap* is encoded as a member function of a type class, the definitions for particular shapes are examples of ad-hoc rather than parametric datatype genericity; each instance entails a proof obligation that the appropriate laws are satisfied.

It is a bit tedious to have to provide a new instance of *Bifunctor* for each new datatype shape; one would of course prefer a single datatype-generic definition. This is the kind of feature for which Generic Haskell [25] is designed, and one can almost achieve the same effect in Haskell [39]. One might hope that these instance definitions could in fact be inferred, in the languages of tomorrow [26]. But whatever the implementation mechanism, the result will still be ad-hoc datatype-generic: it is necessarily the case that different code is used to locate the elements within data of different shapes.

It turns out that the class *Bifunctor* provides sufficient flexibility to capture a wide variety of recursion patterns as datatype-generic programs: a little bit of ad-hockery goes a long way. Here are a number of familiar recursion patterns (map [34], fold [28], unfold [21], hylomorphism [36], and build [22]) captured as datatype-generic programs parameterized by a *Bifunctor* shape s.

$$map :: Bifunctor\ s \Rightarrow (a \to b) \to Fix\ s\ a \to Fix\ s\ b$$
$$map\ f = In \cdot bimap\ f\ (map\ f) \cdot out$$

$$fold :: Bifunctor\ s \Rightarrow (s\ a\ b \to b) \to Fix\ s\ a \to b$$
$$fold\ f = f \cdot bimap\ id\ (fold\ f) \cdot out$$

$$unfold :: Bifunctor\ s \Rightarrow (b \to s\ a\ b) \to b \to Fix\ s\ a$$
$$unfold\ f = In \cdot bimap\ id\ (unfold\ f) \cdot f$$

$$hylo :: Bifunctor\ s \Rightarrow (b \to s\ a\ b) \to (s\ a\ c \to c) \to b \to c$$
$$hylo\ f\ g = g \cdot bimap\ id\ (hylo\ f\ g) \cdot f$$

$$build :: Bifunctor\ s \Rightarrow (\forall b.\ (s\ a\ b \to b) \to b) \to Fix\ s\ a$$
$$build\ f = f\ In$$

The datatype-generic definitions are surprisingly short — shorter even than datatype-specific ones would be. The structure becomes much clearer with the higher level of abstraction. In particular, the duality between *fold* and *unfold* is obvious.

For more about origami programming, see [13, 14].

3. Origami patterns

In this section we describe ORIGAMI, a little suite of patterns for recursive data structures, consisting of four of the Gang of Four design patterns [11]:

- COMPOSITE, for modelling recursive structures;
- ITERATOR, for linear access to the elements of a composite;
- VISITOR, for structured traversal of a composite;
- BUILDER, to generate a composite structure.

These four patterns belong together. They all revolve around the notion of a hierarchical structure, represented as a COMPOSITE. One way of constructing such hierarchies is captured by the BUILDER pattern: a client application knows what kinds of part to add and in what order, but it delegates to a separate object knowledge of their implementation and responsibility for creating and holding them. Having constructed a hierarchy, there are two kinds of traversal we might perform over it: either considering it as a container of elements, in which case we use an ITERATOR for a linear traversal; or considering its shape as significant, in which case we use a VISITOR for a structured traversal.

3.1 Composite

The COMPOSITE pattern 'lets clients treat individual objects and compositions of objects uniformly', by 'composing objects into tree structures'. The essence of the pattern is a common supertype, of which both atomic and aggregated objects are subtypes, as shown in Figure 1.

3.2 Iterator

The ITERATOR pattern 'provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation'. It does this by separating the responsibilities of containment and iteration. The standard implementation is as an *external* or client-driven iterator, illustrated in Figure 2 and as embodied for example in the Java standard library.

In addition to the standard implementation, GOF also discuss *internal* or iterator-driven ITERATORS, illustrated in Figure 3. These might be modelled by the following pair of interfaces:

public interface *Action*{ *Object apply* (*Object o*); }
public interface *Iterator*{ **void** *iterate* (*Action a*); }

An object implementing the *Action* interface provides a single method *apply*, which takes in a collection element and returns (either a new, or the same but modified) element. (The C++ STL calls such objects 'functors', but we avoid that term here to prevent name clashes with type functors.) A collection (implements a FACTORY METHOD to return a separate subobject that) implements the

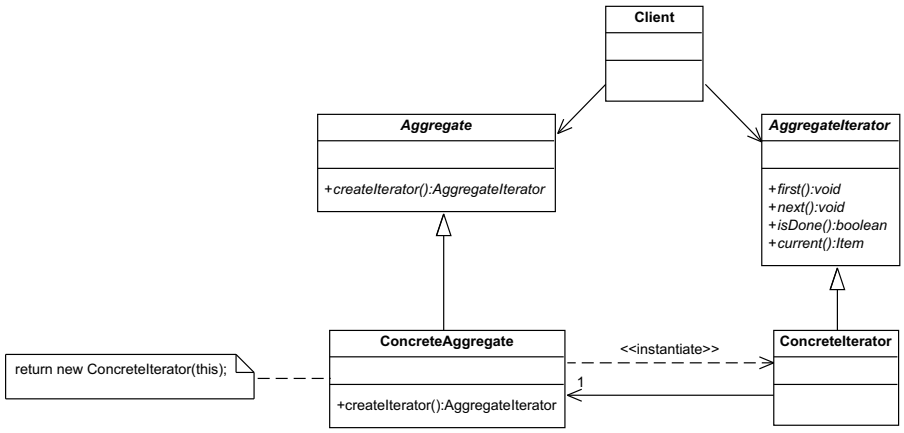

Figure 2. The class structure of the ITERATOR pattern

Iterator interface to accept an *Action*, apply it to each element in turn, and replace the original elements with the possibly new ones returned. Internal ITERATORs are less flexible than external — for example, it is more difficult to have two linked iterations over the same collection, and to terminate an iteration early — but they are correspondingly simpler to use.

3.3 Visitor

In the normal object-oriented paradigm, the definition of each traversal operation is spread across the whole class hierarchy of the structure being traversed — typically but not necessarily a COMPOSITE. This makes it easy to add new variants of the datatype (for example, new kinds of leaf node in the COMPOSITE), but hard to add new traversal operations.

The VISITOR pattern 'represents an operation to be performed on the elements of an object structure', allowing one to 'define a new operation without changing the classes of the elements on which it operates'. This is achieved by providing a hook for associating new traversals (the method *accept* in Figure 4), and an interface for those traversals to implement; the effect is to simulate *double dispatch* on the types of two arguments, the element type and the operation, by two consecutive single dispatches. It is a kind of *aspect-oriented programming* [32], modularizing what would otherwise be a cross-cutting concern. It reverses the costs: it is now easy to add new traversals, but hard to add new variants. (Wadler [48] has coined the term *expression problem* for this tension between dimensions of easy extension.)

3.4 Builder

Finally, the BUILDER pattern 'separates the construction of a complex object from its representation, so that the same construction process can create different representations'. As Figure 5 shows, this is done by delegating responsibility for the construction to a separate object — in fact, a STRATEGY for performing the construction.

The GOF motivating example of the BUILDER pattern involves assembling a product that is basically a simple collection; that is necessarily the case, because the operations supported by a builder object add parts and return void. However, they also suggest the possibility of building a more structured product, in which the parts are linked together. For example, to construct a tree, each operation to add a part could return a unique identifier for the part added, and take an optional identifier for the parent to which to add it; a directed acyclic graph requires a set of parents for each node, and construction in topological order; a cyclic graph requires the possibility of 'forward references', adding parts as children of yet-to-be-added parents.

GOF also suggest the possibility of BUILDERs that compute. Instead of constructing a large *Product* and eventually collapsing it, one can provide a separate implementation of the *Builder* interface that makes the *Product* itself the collapsed result, computing it on the fly while building.

3.5 An example

As an example of applying the ORIGAMI patterns, consider the little document system illustrated in Figure 6. (The complete code is given in an appendix, for reference.)

- The focus of the application is a COMPOSITE structure of documents: *Section*s have a *title* and a collection of sub-*Component*s, and *Paragraph*s have a *body*.
- One can iterate over such a structure using an internal ITERATOR, which acts on every *Paragraph*. For instance, iterating with a *SpellCorrector* might correct the spelling of every paragraph body. (For brevity, we have omitted the possibility of acting on the *Section*s of a document, but it would be easy to extend the *Action* interface to allow this. We have also made the *apply* method return *void*, so providing no way to change the identity of the document elements; more generally, *apply* could optionally return new elements.)
- One can also traverse the document structure with a VISITOR, for example to compute some summary of the document. For instance, a *PrintVisitor* might yield a string array with the section titles and paragraph bodies in order.
- Finally, one can construct such a document using a BUILDER. We have used the structured variant, adding *Section*s and *Paragraph*s as children of existing *Component*s via unique *int* identifiers. A *ComponentBuilder* constructs a *Component* as expected, whereas a *PrintBuilder* incorporates the printing behaviour of the *PrintVisitor* incrementally, actually constructing a string array instead.

This one application is a paradigmatic example of each of the four ORIGAMI patterns. We therefore claim that any alternative representation of the patterns cleanly capturing this structure is a faithful rendition of those patterns. In Section 4 below, we provide just such a representation, in terms of the higher-order datatype-generic programs from Section 2.3. Section 4.5 justifies our claim

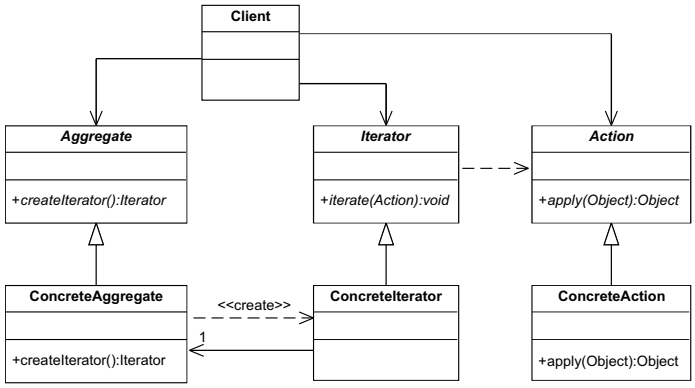

Figure 3. The class structure of an internal ITERATOR

of a faithful rendition by capturing the structure of the document application in this alternative representation.

4. Patterns as HODGPs

We revisit the ORIGAMI patterns from Section 3, showing that each of the four patterns can be captured as a higher-order datatype-generic program (HODGP). However, we consider them in a slightly different order; it turns out that the datatype-generic representation of the ITERATOR pattern builds on that of VISITOR.

4.1 Composite in HODGP

COMPOSITEs are recursive data structures; in the OO setting, they are packaged together with some operations, but in a functional setting the operations are represented separately. So actually, these correspond not to programs, but to types. Recursive data structures come essentially for free in functional programming languages.

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

What is datatype-generic about this definition is that it is parametrized by the shape s of the data structure; thus, one recursive datatype serves to capture *all* (regular) recursive data structures, whatever their shape.

4.2 Visitor in HODGP

The VISITOR pattern collects fragments of each traversal into one place, and provides a hook for performing such traversals. The resulting style matches the normal functional programming paradigm, in which traversals are entirely separate from the data structures traversed. No explicit hook is needed; the connection between traversal and data is made within the traversal by dispatching on the data, either by pattern matching or (equivalently) by applying a destructor. What was a double dispatch in the OO setting becomes in HODGP the choice of a function to apply, followed by a case analysis on the variant of the data structure. A common case of such traversals, albeit not the most general, is the fold operator introduced above.

$$fold :: Bifunctor\ s \Rightarrow (s\ a\ b \to b) \to Fix\ s\ a \to b$$
$$fold\ f = f \cdot bimap\ id\ (fold\ f) \cdot out$$

This too is datatype-generic, parametrized by the shape s: the same function *fold* suffices to traverse any shape of COMPOSITE structure.

4.3 Iterator in HODGP

External ITERATORs give sequential access to the elements of collection. The functional approach would be to provide a view of the collection as a list of elements, at least for read-only access. Seen this way, the ITERATOR pattern can be implemented using the VISITOR pattern, traversing using a body *combiner* that combines the element lists from substructures into one overall element list.

$$contents :: Bifunctor\ s \Rightarrow (s\ a\ (List\ a) \to List\ a) \to Fix\ s\ a \to List\ a$$
$$contents\ combiner = fold\ combiner$$

With lazy evaluation, the list of elements can be generated incrementally on demand, rather than eagerly in advance: 'lazy evaluation means that lists and iterators over lists are identified' [49].

In the formulation above, the *combiner* argument has to be provided to the *contents* operation. Passing different *combiner*s allows the same COMPOSITE to yield its elements in different orders; for example, a tree-shaped container could support both preorder and postorder traversal. On the other hand, it is clumsy always to have to specify the *combiner*. One could specify it once and for all, in the class *Bifunctor*, in effect making it another datatype-generic operation parametrized by the shape s. In the languages of tomorrow, one might expect that at least one, obvious implementation of *combiner* could be inferred automatically.

Of course, some aspects of external ITERATORs can already be expressed linguistically; the interface *java.util.Iterator* has been available for years in the Java API, the iterator concept has been explicit in the C++ Standard Template Library for even longer, and recent versions of Java and C# even provide language support ('**foreach**') for iterating over the elements yielded by such an operator. Thus, element consumers can be written datatype-generically today. But still, one has to implement the *Iterator* anew for each datatype defined; element producers are still datatype-specific.

An internal ITERATOR is basically a map operation, iterating over a collection and yielding one of the same shape but with different or modified elements; it therefore supports write access to the collection as well as read access. In HODGP, we can give a *single generic* definition of this.

$$map :: Bifunctor\ s \Rightarrow (a \to b) \to Fix\ s\ a \to Fix\ s\ b$$
$$map\ f = In \cdot bimap\ f\ (map\ f) \cdot out$$

This is in contrast with the object-oriented approach, in which *Iterator* implementations are datatype-specific. Note also that the HODGP version is more general than the OO version, because it can return a collection of elements of a different type.

Although the internal ITERATOR explains both read and write access to a collection, it doesn't explain imperative access, with impure aspects such as side-effects, I/O and so on. Moreover, it does not subsume the HODGP external ITERATOR, because it does not allow *accumulation* of some measure of the elements (for example, to compute the size of the collection in passing). Recent work on *idiomatic traversals* [35, 19] overcomes both of these

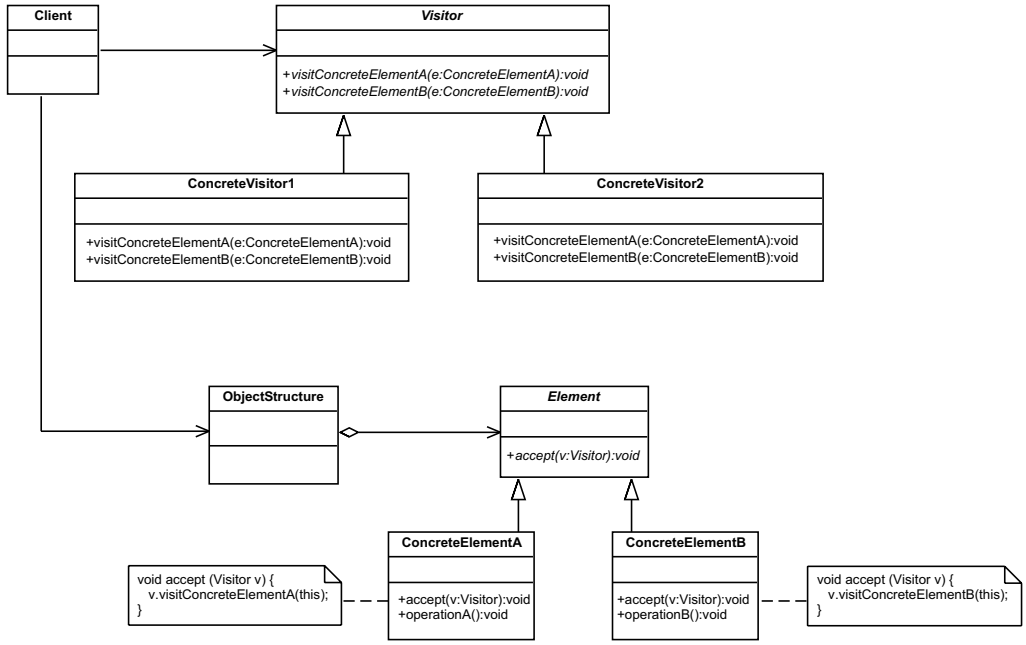

Figure 4. The class structure of the VISITOR pattern

shortcomings: idiomatic traversals support imperative features and mapping and accumulating aspects simultaneously, using *idioms* or *applicative functors*, a slight generalization of monads [47]. One small extra piece of ad-hockery is required: a mechanism for pulling an idiomatic effect out of the shape of a data structure.

class *Bifunctor s* ⇒ *Bitraversable s* **where**
bidist :: *Idiom m* ⇒ *s* (*m a*) (*m b*) → *m* (*s a b*)

Given this tool, a datatype-generic *traverse* operator turns out to be an instance of *fold*:

instance *Bitraversable s* ⇒ *Traversable* (*Fix s*) **where**
traverse f = *fold* (*fmap In* · *bidist* · *bimap f id*)

Applications of *traverse* include maps, accumulations and imperative iterations over collections [19].

4.4 Builder in HODGP

The standard protocol for the BUILDER pattern involves a *Director* sending *Part*s one by one to a *Builder* for it to assemble, and then retrieving from the *Builder* a *Product*. Thus, the product is assembled in a step-by-step fashion, but is unavailable until assembly is complete. With lazy evaluation, we can in some circumstances construct the *Product* incrementally: we can yield access to the root of the product structure while continuing to assemble its substructures. In the case that the data structure is assembled in a regular fashion, this corresponds in the HODGP style to an unfold operation.

unfold :: *Bifunctor s* ⇒
(*b* → *s a b*) → *b* → *Fix s a*
unfold f = *In* · *bimap id* (*unfold f*) · *f*

When the data structure is assembled irregularly, a build operator has to be used instead.

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

These are both datatype-generic programs, parametrized by the shape of product to be built. In contrast, the GOF BUILDER pattern states the general scheme, but requires code specific for each *Builder* interface and each *ConcreteBuilder* implementation.

Turning to GOF's computing builders, with lazy evaluation there is not so pressing a need to fuse building with postprocessing. If the structure of the consumer computation matches that of the producer — in particular, if the consumer is a fold and the producer a build or an unfold — then consumption can be interleaved with production, and the whole product never need be in existence.

Nevertheless, naive interleaving of production and consumption of parts of the product still involves the creation and immediate disposal of those parts. Even the individual parts need never be constructed; often, they can be deforested [46], with the attributes of a part being fed straight into the consumption process. When the producer is an unfold, the composition of producer and consumer is (under certain mild strictness conditions) a hylomorphism.

hylo :: *Bifunctor s* ⇒
(*b* → *s a b*) → (*s a c* → *c*) → *b* → *c*
hylo f g = *g* · *bimap id* (*hylo f g*) · *f*

More generally, but less conveniently for reasoning, the producer is a build, and the composition simply replaces the constructors in the builder by the body of the fold.

foldBuild :: *Bifunctor s* ⇒
(∀*b*. (*s a b* → *b*) → *b*) → (*s a b* → *b*) → *b*
foldBuild f g = *f g*

Once again, both definitions are datatype-generic; both take as arguments a producer *f* and a consumer *g*, both with types parametrized by the shape *s* of the product to be built. Note especially that in both cases, the fusion requires no creativity; in contrast, GOF's computing builders can take considerable insight and ingenuity to program (as we shall see in the appendix).

4.5 The example, revisited

To justify our claim that the higher-order datatype-generic representation of the ORIGAMI patterns is a faithful rendition, we use it to re-express the document application discussed in Section 3.5 and

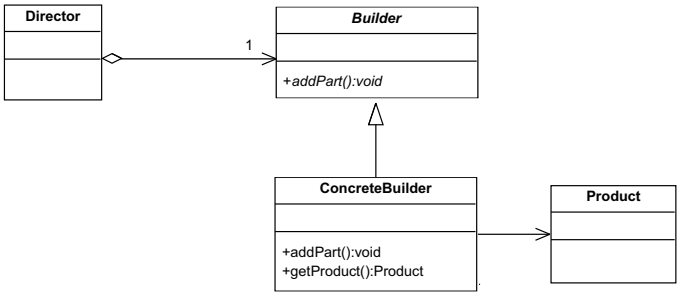

Figure 5. The class structure of the BUILDER pattern

illustrated in Figure 6. (It is instructive to compare these 40 lines of Haskell code with the equivalent Java code in the appendix.)

- The COMPOSITE structure has the following shape.

```
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)
```

  We have chosen to consider paragraph bodies as the 'contents' of the data structure, but section titles as part of the 'shape'; that decision could be varied.

- We used an ITERATOR to implement the *SpellCorrector*; this would be modelled now as an instance of *map*.

```
correct :: String → String    -- definition omitted

corrector :: Doc → Doc
corrector = map correct
```

- The use of VISITOR to print the contents of a document is a paradigmatic instance of a *fold*.

```
printDoc :: Doc → [String]
printDoc = fold combine

combine :: DocF String [String] → [String]
combine (Para s)   = [s]
combine (Sec s xs) = s : concat xs
```

- Finally, in place of the BUILDER pattern, we can use *unfold* for constructing documents, at least when doing so in a structured fashion. For example, consider the following simple representation of XML documents.

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

  From such an XML document we can construct one of our documents, with *Text* elements as paragraphs and *Entity*s as sections with appropriate titles.

```
fromXML :: XML → Doc
fromXML = unfold step

step :: XML → DocF String XML
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
  join [s]    = s
  join (s:ss) = s ++ ", " ++ join ss

attr (k,v)  = k ++ "='" ++ v ++ "'"
paren s     = " (" ++ s ++ ")"
```

  Printing of a document constructed from an XML file is the composition of a fold with an unfold, and so a hylomorphism:

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

- For constructing documents in a less structured fashion, we have to resort to the more general and more complicated *build* operator. For example, here is a builder for a simple document of one section with two sub-paragraphs.

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

  We can actually construct the document from this builder, simply by passing it to the operator *build*, which plugs the holes with document constructors.

```
myDoc :: Doc
myDoc = build buildDoc
```

  If we want to traverse the resulting document, for example to print it, we can do so directly without having to construct the document in the first place; we do so by plugging the holes instead with the body of the *printDoc* fold.

```
printMyDoc :: [String]
printMyDoc = buildDoc combine
```

5. Discussion

We have shown that two advanced language features — *higher-order functions* and *datatype genericity* — suffice (in the presence of other standard features such as datatypes and interfaces) to capture as reusable code a number of the familiar GOF design patterns; specifically, the patterns we have considered are COMPOSITE, ITERATOR, VISITOR and BUILDER, which together we call the ORIGAMI patterns. We also believe that these or similar features are necessary for this purpose, since the design patterns are parametrized by actions and by the shape of datatypes.

Our intentions in doing this work are not so much to criticize the existing informal presentations of these four and other patterns; indeed, as we explain below, the informal presentations contribute much useful information beyond the code. Rather, we aim to promote the uptake of higher-order and datatype-generic techniques, and to encourage their incorporation in mainstream programming languages. In this regard, we are following in the footsteps of Norvig [38], who wrote that 16 of the 23 GOF patterns are 'invisible or simple' in Lisp, and others who argue that design patterns amount to admissions of inexpressiveness in programming languages. However, in contrast to Norvig and the others favouring

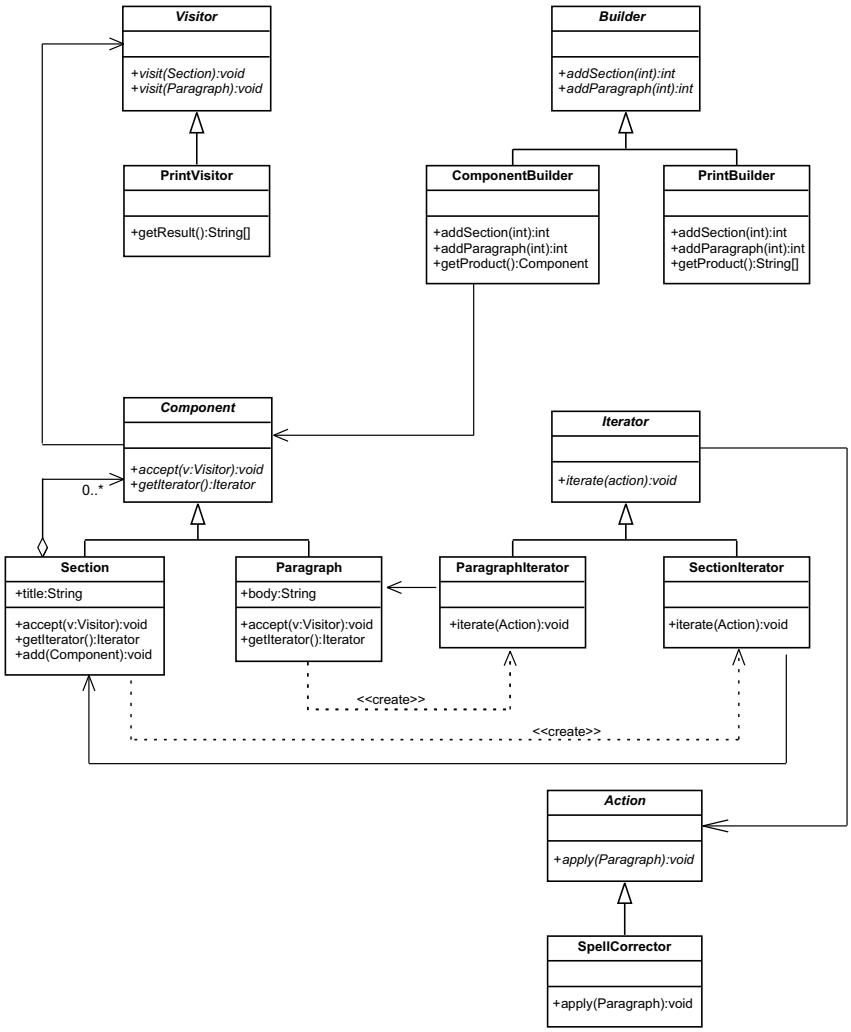

Figure 6. An application of the ORIGAMI patterns

dynamic languages [43], our presentation provides genericity while preserving strong static typing.

We do not claim to have captured all 23 of the GOF patterns, or for that matter any deuterocanonical ones either. In particular, we do not see yet how to capture *creational* design patterns as higher-order datatype-generic programs. This is perhaps because our approach is to model object-oriented ideas in a functional framework, and that framework has no direct analogue of object creation. However, we hope and expect that the languages of tomorrow will provide higher-order datatype-generic features in a more traditional framework, and then we may be able to make better progress. Indeed, Alexandrescu's *type list* implementation of a GENERIC ABSTRACT FACTORY [1] is essentially a datatype-generic metaprogram written using C++ templates.

We also appreciate that there is more to design patterns than their extensional characteristics, which can be expressed as class and sequence diagrams and captured as programs or programming constructs. Also important are their intensional characteristics: motivation for their use, paradigmatic examples, trade-offs in their application, and other aspects of the 'story' behind the pattern. Our presentation impinges only on the limited extensional aspects of those patterns we treat.

6. Related work

This paper is based on ideas from the Algebra of Programming ('Squiggol') community, and especially the work of Roland Backhouse and Grant Malcolm [34, 3], Richard Bird and Oege de Moor [5, 6], Maarten Fokkinga, Erik Meijer and Ross Paterson [10, 36], Johan Jeuring and Ralf Hinze [29, 23, 25], and John Hughes [27]. For their inspiration, I am indebted. For further details on the datatype-generic style presented here, see [13, 14] and the above references.

Barry Jay has an alternative approach to datatype-generic programming, which he calls *shape polymorphism* [31, 30]. He and Jens Palsberg have also done some work on a generic representation of the VISITOR pattern [40], but this relies heavily on reflection rather than his work on shape.

For other recent discussions of the meeting between functional and object-oriented views of genericity, see [9, 12].

7. Conclusions

Design patterns are traditionally expressed informally, using prose, pictures and prototypes. In this paper we have argued that, given the right language features, certain patterns at least could be expressed more usefully as reusable library code. The language features re-

quired, in addition to those provided by mainstream languages, are *higher-order functions* and *datatype genericity*; for some aspects, *lazy evaluation* also turns out to be helpful. These features are familiar in the world of functional programming; we hope to see them soon in more mainstream programming languages.

8. Acknowledgements

This paper elaborates on arguments developed in a course presented while on sabbatical at the University of Canterbury in New Zealand in early 2005, and explored further at tutorials at ECOOP [15] and OOPSLA [16] later that year. The contribution of participants at those venues and at less formal presentations of the same ideas is gratefully acknowledged, as is that of several anonymous referees (and one brave soul who signed his review). The work was carried out as part of the EPSRC-funded *Datatype-Generic Programming* project at Oxford and Nottingham; we thank members of the project for advice and encouragement.

References

[1] A. Alexandrescu. *Modern C++ Design*. Addison-Wesley, 2001.

[2] M. H. Austern. *Generic Programming and the STL*. Addison-Wesley, 1999.

[3] R. C. Backhouse, P. Jansson, J. Jeuring, and L. G. L. T. Meertens. Generic programming: An introduction. In *Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, pages 28–115, 1998.

[4] R. Bird. *Introduction to Functional Programming Using Haskell*. Prentice-Hall, 1998.

[5] R. Bird and O. de Moor. *The Algebra of Programming*. Prentice-Hall, 1996.

[6] R. Bird, O. de Moor, and P. Hoogendijk. Generic functional programming with types and relations. *Journal of Functional Programming*, 6(1):1–28, 1996.

[7] J. Cheney and R. Hinze. A lightweight implementation of generics and dynamics. In *Haskell Workshop*, pages 90–104, 2002.

[8] K. Claessen and J. Hughes. Specification-based testing with QuickCheck. In Gibbons and de Moor [20], pages 17–40.

[9] G. Dos Reis and J. Järvi. What is generic programming? In *Library-Centric Software Design*, 2005. OOPSLA workshop.

[10] M. M. Fokkinga and E. Meijer. Program calculation properties of continuous algebras. Technical Report CS-R9104, CWI, Amsterdam, Jan. 1991.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[12] R. Garcia, J. Järvi, A. Lumsdaine, J. G. Siek, and J. Willcock. A comparative study of language support for generic programming. In *Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2003.

[13] J. Gibbons. Calculating functional programs. In R. Backhouse, R. Crole, and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *Lecture Notes in Computer Science*, pages 148–203. Springer-Verlag, 2002.

[14] J. Gibbons. Origami programming. In Gibbons and de Moor [20], pages 41–60.

[15] J. Gibbons. Design patterns as higher-order datatype-generic programs. `http://2005.ecoop.org/8.html`, June 2005. Tutorial presented at ECOOP.

[16] J. Gibbons. Design patterns as higher-order datatype-generic programs. `http://www.oopsla.org/2005/ShowEvent.do?id=121`, Oct. 2005. Tutorial presented at OOPSLA.

[17] J. Gibbons. Datatype-generic programming. In *Spring School on Datatype-Generic Programming*. Springer-Verlag, 2006. To appear.

[18] J. Gibbons. Design patterns as higher-order datatype-generic programs (full version). `http://www.comlab.ox.ac.uk/jeremy.gibbons/publications/#hodgp`, June 2006.

[19] J. Gibbons and B. C. d. S. Oliveira. The essence of the Iterator pattern. In T. Uustalu and C. McBride, editors, *Mathematically-Structured Functional Programming*, 2006.

[20] J. Gibbons and O. de Moor, editors. *The Fun of Programming*. Cornerstones in Computing. Palgrave, 2003. ISBN 1-4039-0772-2.

[21] J. Gibbons and G. Jones. The under-appreciated unfold. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, pages 273–279, Baltimore, Maryland, Sept. 1998.

[22] A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture*, 1993.

[23] R. Hinze. Polytypic values possess polykinded types. In R. C. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2000.

[24] R. Hinze. Generics for the masses. *Journal of Functional Programming*, 2006.

[25] R. Hinze and J. Jeuring. Generic Haskell: Practice and theory. In R. Backhouse and J. Gibbons, editors, *Summer School on Generic Programming*, volume 2793 of *Lecture Notes in Computer Science*, pages 1–56. Springer-Verlag, 2003.

[26] R. Hinze and S. Peyton Jones. Derivable type classes. In G. Hutton, editor, *Haskell Workshop*, volume 41.1 of Electronic Notes in Theoretical Computer Science. Elsevier Science, Aug. 2000.

[27] J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, Apr. 1989.

[28] G. Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, July 1999.

[29] P. Jansson and J. Jeuring. PolyP – a polytypic programming language extension. In *Principles of Programming Languages*, pages 470–482, 1997.

[30] B. Jay, G. Bellè, and E. Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, 1998.

[31] C. B. Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995.

[32] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[33] A. Löh. *Exploring Generic Haskell*. PhD thesis, Utrecht University, 2004.

[34] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.

[35] C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, To appear.

[36] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, 1991.

[37] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML*. MIT Press, revised edition, 1997.

[38] P. Norvig. Design patterns in dynamic programming. In *Object World*, Boston, MA, May 1996. Tutorial slides at `http://norvig.com/design-patterns/`.

[39] B. C. d. S. Oliveira and J. Gibbons. TypeCase: A design pattern for type-indexed functions. In D. Leijen, editor, *Haskell Workshop*, 2005.

[40] J. Palsberg and C. B. Jay. The essence of the Visitor pattern. In *22nd Annual International Computer Software and Applications Conference*, pages 9–15, Vienna, Austria, August 1998.

[41] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, second edition, 1996.

[42] S. Peyton Jones. *The Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[43] G. T. Sullivan. Advanced programming language features for executable design patterns: Better patterns through reflection. Artificial Intelligence Laboratory Memo AIM-2002-005, Artificial Intelligence Lab, MIT, Mar. 2002.

[44] The Programatica Team. Programatica tools for certifiable, auditable development of high-assurance systems in Haskell. In *High Confidence Software and Systems Conference*. National Security Agency, April 2003.

[45] P. Wadler. Theorems for free! In *Functional Programming and Computer Architecture*, 1989.

[46] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[47] P. Wadler. Monads for functional programming. In M. Broy, editor, *Program Design Calculi: Proceedings of the Marktoberdorf Summer School*, 1992.

[48] P. Wadler. The expression problem. Posting to java-genericity mailing list, 12th Nov 1998.

[49] P. Wadler. How to solve the reuse problem? Functional programming. In *Internal Conference on Software Reuse*, pages 371–372. IEEE, 1998.

9. Appendix: Java programs

Section 4.5 provides a nearly complete implementation of the document application in a higher-order datatype-generic style; all that is missing is a definition for the spelling corrector *correct*. In contrast, Section 3.5 presents only the outline of a Java implementation of the same application. For completeness, this appendix presents the Java code.

9.1 Component

```
public interface Component{
  void accept (Visitor v);
  Iterator getIterator ();
}
```

9.2 Section

```
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);
  }
}
```

9.3 Paragraph

```
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);
  }
}
```

9.4 Iterator

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

9.5 SectionIterator

```
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);
    }
  }
}
```

9.6 ParagraphIterator

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

9.7 Action

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

9.8 SpellCorrector

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

9.9 Visitor

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

9.10 PrintVisitor

```
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;
  }
}
```

9.11 Builder

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

9.12 InvalidBuilderId

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

9.13 ComponentBuilder

```
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){    // root component
      if (comps.isEmpty ()){
        comps.put (new Integer (nextId), c);
        return nextId++;
      }
      else
        throw new InvalidBuilderId
          ("Duplicate root");
    } else {    // non-root
      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++;
        }
      }
    }
  }
}
```

9.14 PrintBuilder

This is the only class with a non-obvious implementation. It constructs the printed representation (a *String* []) of a *Component* on the fly. In order to do so, it needs to retain some of the tree structure: for each *Component*, in the *last* field of the corresponding *Record*, the unique identifier of its right-most child (or its own identifier, if it has no children). The vector *records* is stored in the order the lines will be returned, namely, preorder. A new *Component* is placed after the rightmost descendent of its immediate parent, located by following the *last* references. (The code would be cleaner using Java generics to declare *records* as a *Vector*⟨*Record*⟩ rather than a plain *Vector* of *Objects*, but we wish to emphasize that the datatype-genericity discussed in this paper is a different kind of genericity to that provided in Java 1.5.)

```
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);
  }
  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){   // root component
      if (records.isEmpty ()){
        records.addElement (new Record
          (nextId,nextId,s,""));
        return nextId++;
      }
      else
        throw new InvalidBuilderId
          ("Duplicate root");
    } else {   // non-root
      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;   // ids [x] = ids [y] = pid
        while (r.id != r.last){
          y = x;
          x = find (r.last,x);
          r = recordAt (x);
        }   // lasts [y] = lasts [x] = ids [x]
        records.insertElementAt (new Record
          (nextId,nextId,s,indent + "    "),x + 1);
        recordAt (y).last = nextId;
          // lasts [y] = lasts [x + 1] = nextId
        return nextId++;
      }
    }
  }
}
```

9.15 Main

```
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){   // build then compute
      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 {   // computing builder
      PrintBuilder b = new PrintBuilder ();
      build (b);
      lines = b.getProduct ();
    }
    for (int i = 0;i < lines.length;i++)
      System.out.println (lines [i]);
  }
}
```