"Design Patterns as Higher-Order Datatype-Generic Programs"

Discussion Sheet

Introduction

- Design patterns are reusable software abstractions that tend to be presented extra-linguistically as prose or pictures, yet with a sufficiently powerful type-system they can be expressed directly for immediate reuse by instantiation.
- What is required are *higher-order* –parametrisation by code– and *datatype-generic* –parametrisation by type constructor, the shape of data.
- Common features in functional languages & can be simulated in existing languages.

How do we display menus from distinct cafes?

Suppose two cafes, a pancake house and a lunch diner, are merging into one cafe. Thinking of each as Java classes, we can print their menus by iterating over their items and processing them somehow. Suppose that one uses arrays and the other uses hashmaps, and they have differently named methods that must be used to extract the menu printing information. We can be more uniform by asking them to implement the *interface of tools need for the processing*, then we invoke this processing in two iteration loops. We can reduce the loops to two method calls by *encapsulating iteration control flow* –**iterator**– and using that in the processing method. Importantly, using the iterator hides the particular arrays and hashmaps implementations. *Program to an interface rather than to an implementation!*

Later on, an evening cafe wants to merge in so we resuest it implement the two interfaces –the one for processing and the other for iteration– then invoke the processing method 3 times.

Later on, the evening cafe wants to introduce a dessert sub-menu! Our previous attempt wont work with minor additions. This can get even worse if more cafes join in, then service new sub-menus! We want to be able to treat menu items, menus, submenus, subsubmenus, etc, uniformly. Enter the composite pattern: We use a tree structure. This pattern is used for modelling recursive, hierarchical, structures in a uniform fashion.

Suppose we now want to traverse our menus and the deeply nested an item is, the most costly it should be: In particular, we want to increase cost of an item by its depth. Iteration loses the 'depth structure', we need to recursively pattern match –enter the visitor pattern!

In the same vein, the control flow, the process, of building an object can be encapsulted by the Builder pattern: Make an object, the builder, that has methods to take arguments necessary to build an object. One then makes a builder, provides arguments *as they become available* then asks the builder to actually produce the object. Hence, this can provide a sort of curried intilisation of an object. Neato!

Parametrisation

- Design patterns are patterns in program structure.
- They can be seen as program schemes, capturing commonalities in the large-scale structure of programs and abstracting from differences in the small-scale structure. That is, operations on programs!
- E.g., Common programs can be rendered as loops, but a loop is a particular pattern that traverses some input stream, possibly indirectly, and so the notion of traversals, or 'folds', is the essentially pattern here and it's instantiated by picking a 'body' for the loop –*higher-order!*
  What differs between traversals/loops/folds is the *shape* of the data they operate on —*datatype genericity!*
- *Data structure determines program structure!*
  Hence we abstract away the structure by `s` –such as "pairs of" or "lists of"– over an element type `a` –such as integers of strings, the recursive nature of `s` is captured by in the definition of `Fix` –which may be infinite due to laziness.

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

data Either     a b = Left a  | Right b
data MaybePair  a b = Nothing | Just a b
data EitherList a b = Leaf a  | Just [b]

data List a  = Fix MaybePair a
data ℕ       = Fix (Either ()) a
data Tree a  = Fix EitherList a
-- ≅ (Leaf a | Branches [Tree a]) ≅ (MkTree a [Tree a])
```

'Origami' Patterns

Iterator Iteration control flow

FP Internal iterator $\leq$ functor's `fmap`.
External iterator = a `toList` function.

- That is, we provide a view of the data as a list of elements.

```
contents :: Bifunctor s ⇒ (s a (List a) → List a) → Fix s a → List a
contents combiner (In x) = combiner (bimap id (contents combiner) x)
-- i.e., contents combiner = fold combiner
```

Composite Multibranching hierarchies; uniformal treatment of sub-trees and leafs.

FP `RoseTree a` where both `a` and `RoseTree a` implement a common typeclass, say `Component`, thereby providing uniform treatment of the two.

- This corresponds to `Fix`.

Visitor Provides structured traversal of a composite;

- Actually take into consideration the shape of the container.
- An example would be `fold`.

Builder Curried object construction

FP Currying, partially evaluating functions, `unfold`:

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

Bifunctors

Not all shapes can be fixed, we look at those that can "identify their elements" –the *bifunctors*.

```
class Bifunctor s where
  bimap :: (a → c) → (b → d) → s a b → s c d
  -- Lift functions to act onto the shapes.

instance Bifunctor Either where
  bimap l r (Left x)  = Left (l x)
  bimap l r (Right x) = Right (r x)


map :: Bifunctor s ⇒ (a → b) → Fix s a → Fix s b
map f (In x) = In (bimap f (map f) x)

fold :: Bifunctor s ⇒ (s a b → b) → (Fix s a → b)
fold f (In x) = f (bimap id (fold f) x)

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

- ◇ The datatype-generic definitions are surprisingly short!
- ◇ The structure becomes much clearer with the higher level of abstraction.
- ◇ In Java, we cannot parameterise on a type constructor; here is an un-useful approximation.

```
interface BiFunctor<A,B,C,D, FromInstanceOnAB, ToInstanceOnCD>
{
    ToInstanceOnCD bimap(FromInstanceOnAB sab, Function<A,C> f, Function<B, D> g);
}
```

The Composite Pattern

The Composite Pattern allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly –by having them both implement an interface, say `Components`, supporting desired operations– thereby ignoring the differences between compositions [subtrees] and individuals [leafs]: That is, no having to pattern match, say by `if`'s, of whether a component is a leaf or a sub-tree to ensure we're calling the right methods.

A composite contains components, which may be leafs or composites. A recursive definition. The composite pattern handles child management and leaf operations thereby trading single responsibility for transparency: The ability to treat components uniformly.

Sources Consulted

- ◇ Design Patterns as Higher-Order Datatype-Generic Programs
- ◇ Head First Design Patterns, in Java
- ◇ Example Implementation of Rose Trees
- ◇ Reason isomorphically!

Functors ≅ The Iterator Pattern ... almost

The Iterator Pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

It places the task of traversal on the iterator object, not on the aggregate, which simplifies the aggregate interface and implementation, and places the responsibility where it should be.

```
interface Iterator<T> {
    boolean hasNext();    // Are there more elements to iterate over?
    T       next();       // Actually obtain the next element of type T.

    // The above suffices as an external iterator.
    // The following default method makes it into an internal iterator.

    // Almost like fmap, but loses structure.
    // Consumer<T>  ≅ Function<T,void>.
    default void iterate(java.util.function.Consumer consumer)
    {
        while(this.hasNext())
            consumer.accept(this.next());
    }
}
```

The first two pieces above form an "external iterator" since it lets the client control iteration by calling `next() and hasNext()`. An "internal iterator" controls iteration itself and the client has to tell it how to process the elements as it goes through them. In some sense the internal iterators are easier to use since we just hand them an operation and they do all the work of actually iterating themselves. The former necessitiaes the boilerplate of `iterate`.

Either way, programming to an interface allows more polymorphic code.

Single Responsibility Principle A class should have only one reason to change. Every responsibility of a class is an area of potential change. More than one responsibility means more than one area of change.

Conclusion

Hopefully this work promotes the uptake of higher-order and datatype-generic techniques, and to encourage their incorporation in mainstream programming languages.

Design patterns are invisible or simple using functional programming features!

- ◇ How to capture creational design patterns as higher-order datatype-generic programs?
- ◇ 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.