Patterns in Datatype-Generic Programming

Jeremy Gibbons
University of Oxford
APPSEM II, April 2004

0. Outline

1. Generic programming

Generic programming is a matter of

> *making programs more adaptable by making them more general.*

The intention is to allow the programmer

> *to capture more recurring patterns as abstractions.*

This is achieved by providing a wide variety of kinds of parameter: *non-traditional polymorphism.*

Parameters may have a rich structure: eg types, type constructors, other programs, class hierarchies, programming paradigms. . .

Mostly popular realization today is through the *C++ Standard Template Library.*

2. C++ template meta-programming

Methods and classes parametrized by *types*:

```
template<class T>
void swap (T& a, T& b) { T c = a; a = b; b = c; }
main() {
  int i1 = 3, i2 = 4;          swap<int>(i1,i2);
  double d1 = 3.5, d2 = 4.5; swap<double>(d1,d2);
}
```

... and by (integral, enumerated or pointer) values:

```
template<class T, int size>
class Vector { private: T values[size]; ... };
main() {
  Vector<int, 3> v;
  Vector<Vector<double,100>, 100> matrix;
}
```

2.1. Bluffer's guide to the STL

A library of collection types and algorithms using C++ templates.

Container types: parametrically-polymorphic collection types — arrays, sequences, etc

Iterators: abstraction of pointer; increment, decrement, arithmetic, dereference as l-value or r-value

Algorithms: functions defined over container types: `count`, `sort`, `reverse`, etc

Concepts: abstract requirements of a template parameter; iterator concepts form interface between algorithms and containers

Function objects: objects providing `operator()` method; other parameters to algorithms (eg ordering for sorting)

2.2. Compile-time meta-programming

In fact, there is a lot more to templates than appears in the STL.

The template language forms a simple, purely-functional sublanguage that is *executed at compile time*.

The language is Turing complete: Unruh showed how to compute primes in the compiler, and Czarnecki and Eisenecker implement a rudimentary LISP interpreter via templates. (Compilers typically provide only bounded recursion, though.)

Not just a cute trick: Alexandrescu shows some startling meta-programming paradigms (such as the *generic abstract factory*).

2.3. The problem with templates

A class template is *not a class*: it cannot be manipulated until it is instantiated.

There is *no static checking* (other than some syntax checking) of templates. The instances are statically checked, but not the template itself.

A class template is not a formal construct with its own semantics: it is more like a macro. There is *no hope of a theory* of generic programming with templates.

Template meta-programming is awkward. True meta-programming would *support 'programs as data'* as first-class citizens, with perspicuous rather than obscure techniques for manipulation.

2.4. Exploiting and reasoning about shape

The STL succeeds in providing a generic library of datatypes and algorithms. However, the template mechanism prohibits reasoning about those datatypes and algorithms.

Moreover, the library is rather impoverished: all the datatypes are kinds of sequence. An iterator is a very small window through which to view a data structure. *'The moving finger writes, and having writ, moves on.'*

It is impossible using STL to write generic algorithms that exploit the shape of data (parsers, pretty-printers, encoders, marshallers, etc.) One can only write algorithms that ignore the shape or discard it.

STL does not support an *algebra of programming* well.

3. The algebra of programming

We already have a good understanding of the theory of *first-order parametric polymorphism*, such as for polymorphic but monotypic traversals:

```
> foldL   :: (Maybe (a,b) -> b) -> List a -> b
> unfoldL :: (b -> Maybe (a,b)) -> b -> List a

> foldT   :: (Either a (b,b) -> b) -> Tree a -> b
> unfoldT :: (b -> Either a (b,b)) -> b -> Tree a
```

We also have a good understanding of *higher-order parametric polymorphism*, such as for polymorphic and polytypic folds:

```
> foldF   :: Bifunctor h => (h a b -> b) -> Fix h a -> b
> unfoldF :: Bifunctor h => (b -> h a b) -> b -> Fix h a
```

3.1. Datatype-generic programming

Parametricity suffices for operations like `foldF` and `unfoldF` that preserve shape without manipulating it. But more generally, we want to *compute with shape* (for example, to define a generic marshaller and unmarshaller from complex datatypes to bit-sequences and back again).

We call this kind of parametrization *datatype-generic programming*. (We considered talking about a *type-parametrized–type—parametrized theory of programming*.)

An understanding of this theory would allow us to treat datatype-generic programs (a constrained but disciplined subset of template meta-programs) as formal constructs.

We would then have a sound basis for manipulating them:
static checking, reasoning, derivation, etc.

3.2. Polytypic programming

We are making some steps forward in programming using *higher-order ad-hoc polymorphism* (for example, in Generic Haskell). It is typically expressed using induction over the structure of datatypes:

```
show<Unit> ()           = "()"
show<Int> x             = showInt x
show<:+:> f g (Inl u) = "Left (" ++ f u ++ ")"
show<:+:> f g (Inr v) = "Right (" ++ g v ++ ")")
show<:*:> f g (u, v)  = "(" ++ f u ++ ", " ++ g v ++ ")"
```

But we know very little of the theory underlying this approach. In particular, there need be no connection between behaviours at different types. Proofs of correctness can be done by induction over the structure, but how can we derive programs to be correct by construction? Can we combine the safety of HOPP with the flexibility of HOAP?

4. Patterns generically

Claim: inside many *design patterns* is a datatype-generic program trying to get out.

For example, the STL *input iterator* concept roughly corresponds to the ITERATOR design pattern. An STL concept is an informal construct; the DGP analogue is a generic program `flatten<T> :: T a -> List a`.

Dually, there ought to be a COITERATOR design pattern, corresponding to the STL *output iterator*. The DGP analogue would be a generic program `grow<T> :: List a -> T a` that generates a collection (of some otherwise-determined shape) from its contents.

My PhD student Bruno Oliveira is exploring this hypothesis. He is supported by the EPSRC-funded project *Datatype-Generic Programming*.

5. Appendices

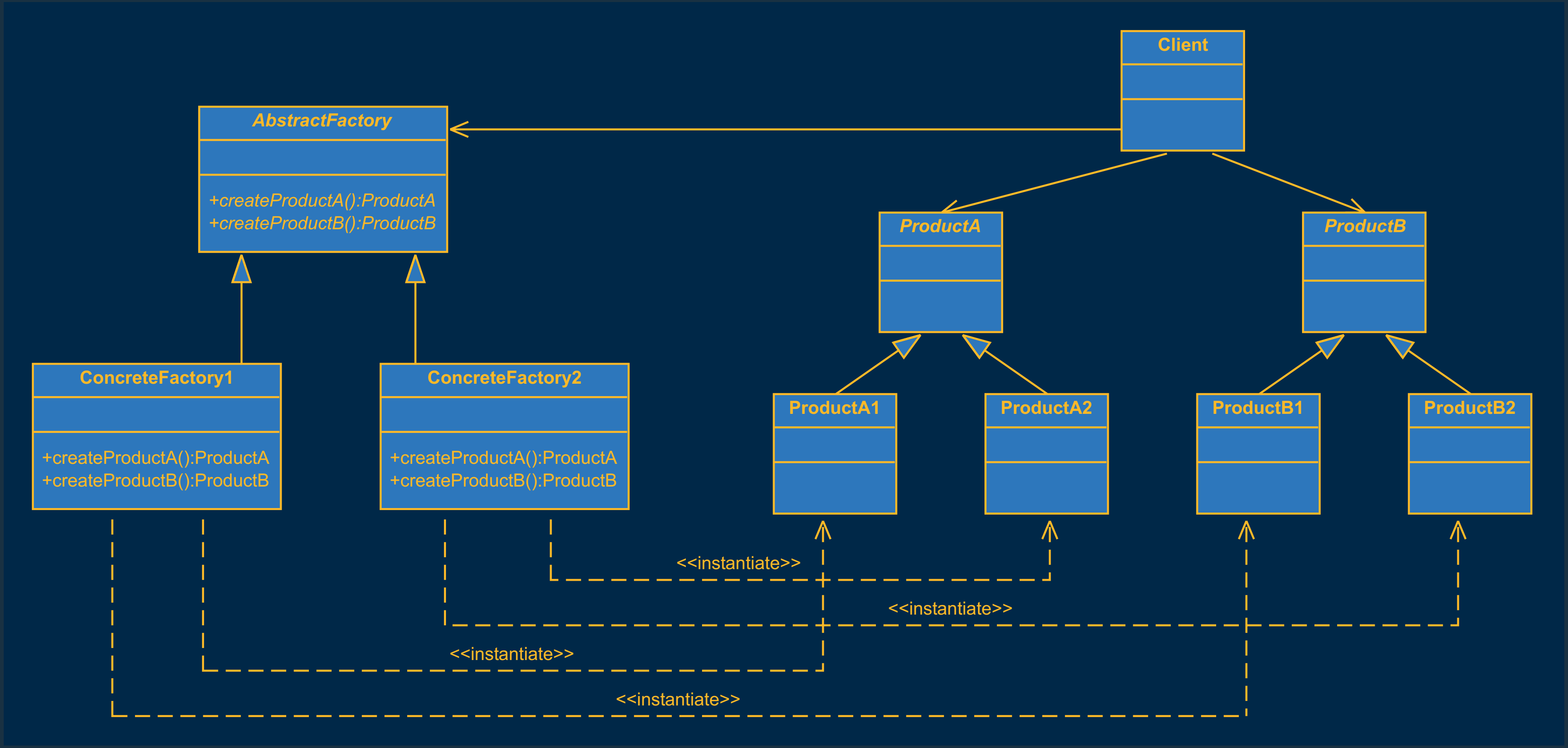
- primes at compile time
- generic abstract factory
- lists in Haskell
- trees in Haskell
- Gofer Goes Bananas
- two cultures
- visitors

5.1. Primes at compile-time (after Unruh)

```
template<int p, int d>
  struct noDiv {
  enum {
    ans = (p==2) ||
      ((p%d) &&
      noDiv<(d>2?p:0),d-1>::ans )
  };
};
template<> struct noDiv<0,1> {
  enum { ans = 1 };
};
template<> struct noDiv<0,0> {
  enum { ans = 1 };
};

template<int n> struct isPrime {
  enum { ans = noDiv<n,n-1>::ans };
};
```

```
template <int i> struct D {
  D(void*);
};

template<int n> struct primes {
  primes<n-1> rest;
  void f() {
    D<n> current =
      isPrime<n>::ans ? 1 : 0;
    rest.f();
  }
};
template<> struct primes<1> {
  void f() { D<1> current = 0; }
};

main() {
  primes<10> junk; junk.f();
}
```

5.2. Generic abstract factory (Alexandrescu)



5.3. Lists in Haskell

```
> data List a = Nil | Cons (a, List a)

> foldL :: b -> ((a,b)->b) -> List a -> b
> foldL e f Nil              = e
> foldL e f (Cons (a,x)) = f (a, foldL e f x)

> sumL :: List Int -> Int
> sumL = foldL 0 plus  where plus (a,b) = a+b

> concatL :: (List a, List a) -> List a
> concatL (x,y) = foldL Cons y x

> mapL :: (a->b) -> List a -> List b
> mapL f = foldL Nil (\ (a,y) -> Cons (f a,y))
```

5.4. Binary trees in Haskell

```
> data Tree a = Tip a | Bin (Tree a, Tree a)

> foldT :: (a->b) -> ((b,b)->b) -> Tree a -> b
> foldT f g (Tip a)      = f a
> foldT f g (Bin (t,u)) = g (foldT f g t, foldT f g u)

> sumT :: Tree Int -> Int
> sumT = foldT id plus

> flattenT :: Tree a -> List a
> flattenT = foldT wrap concatL

> mapT :: (a->b) -> Tree a -> Tree b
> mapT f = foldT (Tip . f) Bin
```

5.5. Generic definitions in Haskell

```
> class Bifunctor h where bimap :: ...
> data Bifunctor h => Fix h a = In (h a (Fix h a))

> foldF :: Bifunctor h => (h a b->b) -> Fix h a -> b
> foldF f (In x) = f (bimap (id, foldF f) x)

> mapF :: Bifunctor h => (a->b) -> Fix h a -> Fix h b
> mapF f = foldF (In . bimap (f,id))

> data ListF a b = NilF | ConsF (a,b)
> instance Bifunctor ListF where ...
> type List’ a = Fix ListF a

> data TreeF a b = TipF a | BinF (b,b)
> instance Bifunctor TreeF where ...
> type Tree’ a = Fix TreeF a
```

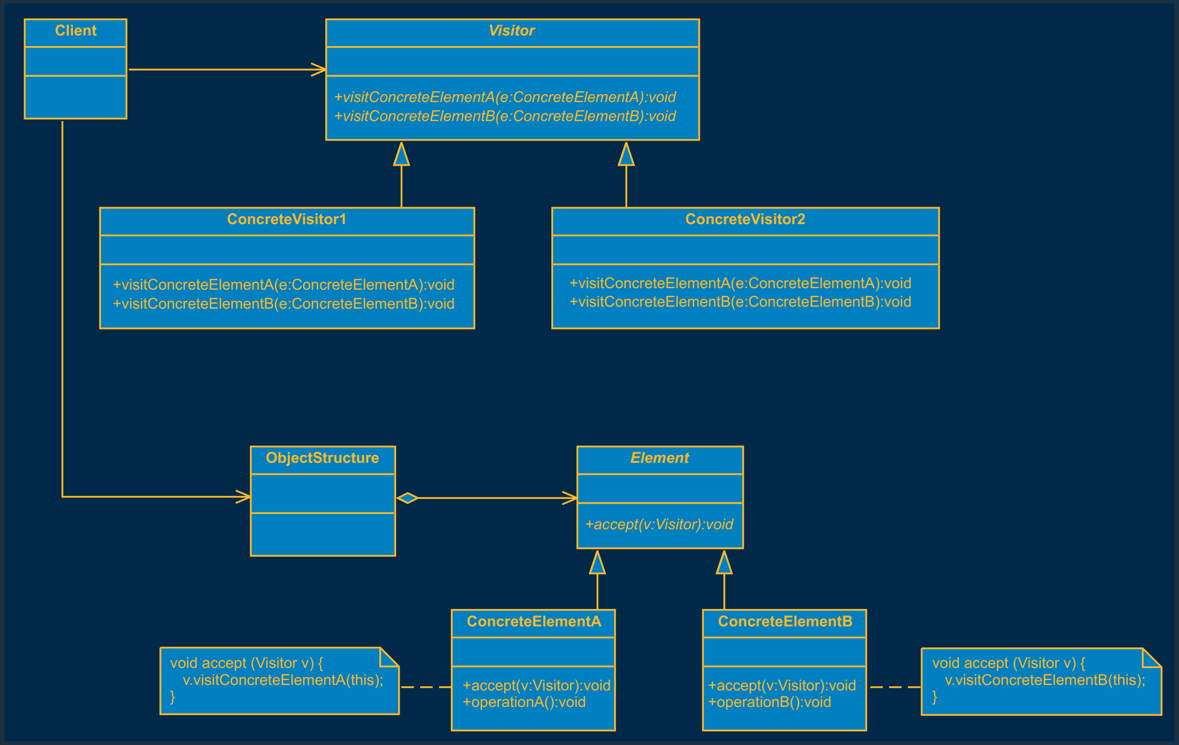
5.6. Two cultures

Proceedings of the IFIP TC2 Working Conference on Generic Programming, J. Gibbons and J. Jeuring (eds), Kluwer Academic Publishers, 2003.

1. *Generic Programming within Dependently-Typed Programming*
2. *Generic Haskell, Specifically*
3. *Generic Accumulations*
4. *A Generic Algorithm for Minimum Chain Partitioning*
5. *Concrete Generic Functionals*
6. *Making the Usage of STL Safe*
7. *Static Data Structures*
8. *Adaptive Extensions of Object-Oriented Systems*
9. *Complete Traversals as General Iteration Patterns*
10. *Efficient Implementation of Run-Time Generic Types for Java*

5.7. Visitors

The VISITOR pattern is approximately the OO equivalent of the AoP `fold`. It provides a means to traverse a data structure, accumulating some result.


Client
Visitor
+visitConcreteElementA(e:ConcreteElementA):void
+visitConcreteElementB(e:ConcreteElementB):void
ConcreteVisitor1
+visitConcreteElementA(e:ConcreteElementA):void
+visitConcreteElementB(e:ConcreteElementB):void
ConcreteVisitor2
+visitConcreteElementA(e:ConcreteElementA):void
+visitConcreteElementB(e:ConcreteElementB):void
ObjectStructure
Element
+accept(v:Visitor):void
ConcreteElementA
+accept(v:Visitor):void
+operationA():void
ConcreteElementB
+accept(v:Visitor):void
+operationB():void
void accept (Visitor v) { v.visitConcreteElementA(this); }
void accept (Visitor v) { v.visitConcreteElementB(this); }

New traversals can be added without changing the class hierarchy; yet it is completely type-safe (no downcasting).

5.8. The essence of visitors

Palsberg and Jay have tried to produce *generic visitors*.

Their `Walkabout` class uses reflection to find all the fields of an object, and recursively visits each.

For a known class hierarchy, `Walkabout` can be specialized, avoiding reflection and essentially recovering the standard VISITOR.

5.9. Adaptive object-oriented programming

Lieberherr's *adaptive object-oriented programming* or *Demeter method* is related.

A program is parametrized by the class hierarchy: hence, generic.

Propagation pattern on class hierarchy (**from** root node **to** target nodes, **through** certain edges and **bypassing** others) specifies a subgraph.

Automatic generation of traversal code to visit every vertex of subgraph (depth-first?). *Wrapper* code fragments **prefix** or **suffix** visits to particular vertices.

(Reminiscent of *XQuery* and *XSLT* for XML.)

5.10. Datatype-generic visitors

It seems to me that datatype-specificity and reflection could both be avoided, by developing *datatype-generic visitors*. They are essentially just folds, after all.