6.005 Software Construction
Fall 2011
Prof. Rob Miller

L8: Interpreters & Visitors

Today

- Design patterns
- Interpreter pattern
- Visitor pattern

Required Reading (from the Java Tutorial & Thinking in Java)

- Inner classes
- Generic methods & constructors

Recipes for Program Construction

Let's take a moment to review the approaches we've been taking to building software in this course. Each of these is a recipe for tackling a different-sized software construction problem. In the first project, you'll have complete design freedom, so use it well.

Recipe for a method:

1. **Design.** Write the signature and specification. Think from the client's point of view. Common mistake is to put too much detail about how the implementation works into the spec. That reduces the implementer's freedom, and makes the spec harder to understand by itself.
2. **Test.** Write test cases that cover the spec.
3. **Code.** Implement the method. Make sure your test cases cover the code of the method (e.g. using a code coverage tool like EclEmma), and add more test cases until you achieve coverage.

Recipe for an abstract data type (a class of mutable or immutable objects):

1. **Design.**
   a. Choose the operations: the creators, producers, mutators, and observers that a client of the type will need.
      i. If the type is mutable, draw a state machine to define its important states and the events (mutators) that transition between them.
      ii. If the type involves recognizing or generating a structured language, write a grammar describing the language.
      iii. If the type is recursive, write a datatype expression showing how it's structured, and convert it into interfaces and classes.
   b. Write signatures and specifications for each of the operations.
2. **Test.**
   a. Write test cases that cover the transitions of the state machine, and cover the specs of the methods.
3. **Code.**
   a. Choose the rep, and write down the rep invariant and abstraction function.
   b. Implement the methods, running your test suite frequently as you go to check for regressions.
   c. Make sure your tests cover your code.

Finally, the recipe for a program:

1. **Choose data structures**. What data types will we use, and can we get them from a library or build them? Should the hailstone sequence be stored as a list or an array? For a lexer and parser, what do the tokens look like? Data is central to most programs. If you show me your data types, I can guess how your code works. If you just show me your code without your data types, I'll continue to be mystified.[1]
2. **Divide into steps**. For example, the steps of computing a hailstone sequence, then finding the peaks in it. Computing the digits of pi, converting to base-26, replacing with a-z, and searching for words. Lexing into tokens, then parsing into expressions.
3. **Module by module.** Implement each new data structure you'll need using the recipe for data types. Do it in isolation, as a unit. For example, Sudoku vs. Formula. Then implement each step of the process. Some of these steps will be methods in your data types; others may be static methods that stand alone in their own classes. Unit-test everything you can.
4. **Put it together**. Connect modules together, one by one, and test that they work together. Incremental is very important – an engineer expects things to fail, and makes only a small change at a time. Whenever you make a change or fix a bug at this stage, rerun *all* your tests to check for regressions.

Often you can follow these recipes straight through, but not always. You will sometimes discover when you're coding that your design was incomplete, and you need to back up and modify it, rewrite some tests. But if you don't start with a design – if you don't put some thought and preplanning into what you're doing -- then you'll be doing far more rewriting than necessary, and you are likely to end up with something unmaintainable – unsafe, hard to understand, and hard to change.

Functions over Recursive Data Types

Last time we looked briefly at one representation for formulas of boolean variables, which was defined recursively as:

```
Formula = Var(name:String)
          + Not(f: Formula)
          + And(left: Formula, right: Formula)
          + Or(left: Formula, right: Formula)
```

We can implement it as an interface Formula and four classes that implement it:

```
public static interface Formula {
}

public static class Var implements Formula{
    public final String name;
    public Var(String name) { this.name = name; }
}

public static class Not implements Formula{
```

[1] This is a paraphrase of a classic quote by Fred Brooks, who wrote the seminal book on software project management, *The Mythical Man-Month*. Brooks's original quote referred to flowcharts and tables, which were popular notions in the 1960s. Eric Raymond, who wrote the book on open source software development, *The Cathedral and the Bazaar,* paraphrased his quote into data structures and code, for the C programmers of the 90s. Guy Steele, who wrote the book on Lisp and co-invented Java, paraphrased it again into contracts and code, for the new millenium. http://dreamsongs.com/ObjectsHaveNotFailedNarr.html

```
        public final Formula f;
        public Not(Formula f) { this.f = f; }
    }

    public static class And implements Formula{
        public final Formula left, right;
        public And(Formula left, Formula right) { this.left = left; this.right
= right; }
    }

    public static class Or implements Formula{
        public final Formula left, right;
        public Or(Formula left, Formula right) { this.left = left; this.right =
right; }
    }
```

Last time we also saw that we could define recursive functions over datatypes, such as:

eval: Formula x Env → boolean

eval(Var(name:String), env) = env.lookup(name)

eval(Not(f:Formula) , env) = ¬ eval(f, env)

eval(And(left, right: Formula) , env) = eval(left, env) ∧ eval(right, env)

eval(Or(left, right: Formula) , env) = eval(left, env) ∨ eval(right, env)

which translates directly into an eval( ) method implemented on the variant classes:

```
    public static interface Formula {
        public boolean eval(Env env);
    }

    public static class Var implements Formula {
        ...
        public boolean eval(Env env) { return env.lookup(name); }
    }

    public static class Not implements Formula {
        ...
        public boolean eval(Env env) { return !f.eval(env); }
    }

    public static class And implements Formula {
        ...
        public boolean eval(Env env) { return left.eval(env) &&
right.eval(env); }
    }

    public static class Or implements Formula {
        ...
        public boolean eval(Env env) { return left.eval(env) ||
right.eval(env); }
    }
```

This is a great approach for defining core operations of the datatype, like eval().

But this approach won't work for all the functions we might want. Suppose some client wants a hasNot() function that tests whether there's any negation in the formula:

hasNot: Formula → boolean

hasNot(Var(name:String)) = false

hasNot(Not(f:Formula)) = true

hasNot(And(left, right: Formula)) = hasNot(left) || hasNot(right)

hasNot(Or(left, right: Formula)) = hasNot(left) || hasNot(right)

Another might want a countVars() function that counts the variables, or a varSet() function that collects all the variables, or a hasLowercaseVariable() that looks for a particular kind of variable name that presumably has semantic meaning for the client... We can't fill up our Formula interface with methods needed by only one client; that would destroy its coherence as an abstract data type.

A final problem with this approach is that it spreads out the code that implements the function into multiple classes – in fact, multiple *files* in a language like Java. That makes it much harder to understand, because a programmer has to piece together the way eval() works by looking in multiple places, and make sure that those places are kept consistent with each other when the function is changed. It doesn't show everything about eval() in one place, like our mathematical function definition does.

Design Patterns

The recursive-function-as-method approach has a standard name in software engineering: it's the **Interpreter** design pattern. We'll see another approach in a moment, but first a word about the notion of design patterns.

A design pattern is a standard solution to a common programming problem.

Design patterns are like standard tools in your toolbox. When you see a common problem, you can turn to your collection of design patterns and decide whether one fits the situation or not.

Design patterns are also a useful **vocabulary** for talking about software design. You can tell another programmer to "throw an *exception*" or "use a *factory method*" or "create a *visitor*," and they will know what you are talking about. So design patterns are useful abstractions for communicating the meaning of your program to other humans, too, not just for communicating with the compiler.

We've already encountered some important design patterns in this course:

Encapsulation (aka information hiding)

Problem: Exposed fields can be directly manipulated from outside, leading to violations of the representation invariant or undesirable dependences that prevent changing the implementation.
Solution: Hide some components, permitting only stylized access to the object – public getX() methods rather than direct access to a private field x.
Disadvantages: The interface may not (efficiently) provide all desired operations. Indirection may reduce performance.

Iterator

Problem: Clients that wish to access all members of a collection must perform a specialized traversal for each data structure. This introduces undesirable dependences and does not extend to other collections.
Solution: Implementations, which have knowledge of the representation, perform traversals and do bookkeeping. The results are communicated to clients via a standard interface (called Iterator in Java).
Disadvantages: Iteration order is fixed by the implementation and not under the control of the client. You can't go backwards through the elements, or jump around in a pseudorandom sequence.

Exceptions

Problem: Errors occurring in one part of the code should often be handled elsewhere. Code should not be cluttered with error-handling code, nor return values preempted by error codes.
Solution: Introduce language structures for throwing and catching exceptions.
Disadvantages: Code may still be cluttered. It can be hard to know where an exception will be handled. Programmers may be tempted to use exceptions for normal control flow, which is confusing and usually inefficient.

These particular design patterns are so important that they are built into Java! Other design patterns are so important that they are built into other languages. Some design patterns may never be built into languages, but are still useful in their place. Other patterns we've thought about include **state machines**, **lexer/parser**, and **abstract syntax tree**.

Factory Pattern

Here's a design pattern that you'll see frequently with abstract data types, particularly in languages like Java that restrict what interfaces and constructors can do: the **Factory Method** pattern.

Recall that abstract data types have several different kinds of operations: creators, producers, observers, and mutators. When we first talked about ADTs, we equated a creator with a constructor. But that's not really enough for abstract data types that want to hide their concrete classes. Remember ImList:

```
    public static interface ImList<E> {
        public ImList<E> cons(E e);
        public E first();
        public ImList<E> rest();
    }

    public static class Empty<E> implements ImList<E> {
        public Empty() { }
        ...
    }
```

We want a client to be able to make an empty list without having to know about the Empty class, so that ImList has the freedom to change it in the future (representation independence). Unfortunately we can just put a constructor in the ImList interface:

```
    public static interface ImList<E> {
        /** make an empty list */
        public ImList<E>() { ... }  // <==== Java forbids constructors in
interfaces
```

A factory method lets us handle this problem. It's simply a static method that creates a value of a type:

```
    /** @return an empty list */
    public <E> ImList<E> empty() {
        return new Empty();
    }
```

Here's another useful factory method for ImList:

```
    /** @param lst  a mutable java.util.List
     *  @return a new immutable ImList representing the same sequence as lst */
    public <E> ImList<E> makeList(List<E> lst) { ... }
```

Factory methods simplify client code and provide a layer of encapsulation between the client and the implementer.

Patterns for Functions over Recursive Datatypes

Interpreter and **Visitor** are the major design patterns we'll see today. We've actually already had a glimpse of Interpreter, which is we were calling the recursive-function-as-method approach. Java has good support for it with interface methods and implementation methods.

Visitor is actually built into some functional programming languages, as pattern matching in ML and Haskell. But it's not built into Java, and it turns out to be very useful for implementing functions over recursive datatypes like lists and trees.

Back to our problem of how to represent a function over a recursive datatype, without adding a new method to the datatype's interface. Let's consider the hasNot function:

hasNot: Formula → boolean

hasNot(Var(name:String)) = false

hasNot(Not(f:Formula)) = true

hasNot(And(left, right: Formula)) = hasNot(left) || hasNot(right)

hasNot(Or(left, right: Formula)) = hasNot(left) || hasNot(right)

The key idea of Visitor is to bring all of these cases into one place, so we wish we could do something like this:

```
public static boolean hasNot(Formula f) {
    // not valid Java code!
    switch (f) {
    case Var v: return false;
    case Not n: return true;
    case And a: return hasNot(a.left) || hasNot(a.right);
    case Or o:  return hasNot(o.left) || hasNot(o.right);
    }
}
```

The switch statement in Java doesn't work that way – this would be the right thing to do if f were an enum, but it doesn't work for object types. Here's another try that is legal Java but very bad style:

```
public static boolean hasNot(Formula f) {
    // not statically checked!
    if (f instanceof Var) {
        return false;
    } else if (f instanceof Not) {
        return true;
    } else if (f instanceof And) {
        And a = (And) f;
        return hasNot(a.left) || hasNot(a.right);
    } else if (f instanceof Or) {
        Or o = (Or) f;
        return hasNot(o.left) || hasNot(o.right);
    } else {
        throw new AssertionError("shouldn't get here!");
    }
}
```

What this code does is determine the type of the list using instanceof (see the Java Tutorial for more details), and then *downcasts* from the interface type Formula to the concrete class types so that it can use their specific fields and methods.

The code above is bad because it gives up static checking. One way that datatypes often change is by adding new variants, new concrete classes that implement Formula. Suppose somebody extends the boolean formulas to support existential and universal quantification -- ThereExists and ForAll. If that happened, the compiler would *force* the maintainer to implement the eval() method for these new variants because it was mentioned in the Formula interface. The program wouldn't even compile unless the maintainer took care of eval(), so functions implemented with the Interpreter pattern are statically checked. But hasNot() written as shown above would continue to compile, happily but wrongly, with no static checking to remind us that we need to change that function too.

The code above does do one thing right: at least it defends itself against unexpected new variants of Formula that it encounters at runtime. It doesn't assume that Var, Not, And, and Or are the only possible implementations of Formula, and it throws an error if it discovers a new one, rather than returning wrong answers. The following code would be much much worse:

```
public static boolean hasNot(Formula f) {
    if (f instanceof Var) {
        return false;
    } else if (f instanceof And) {
        And a = (And) f;
        return hasNot(a.left) || hasNot(a.right);
    } else if (f instanceof Or) {
        Or o = (Or) f;
        return hasNot(o.left) || hasNot(o.right);
    } else {
        // must be a Not    <=== not defensive!
        return true;
    }
}
```

I just can't put enough big red X's on this code.

In general, instanceof should be avoided. There is almost always a way to write your code that doesn't require testing the type of an object with instanceof and downcasting. Use of instanceof is a sign of a bad smell. There are exceptions to this – instanceof is frequently used to implement equals(), as we'll see – but generally instanceof should not be part of a good object-oriented design.

The Visitor Pattern

We saw two almost-right approaches. Here's how the visitor pattern actually works. We'll follow the model of the switch statement we wished we could write, and show how each part of it can be represented in correct, typesafe, statically-checked Java.

Let's start by thinking about the body of our imaginary switch statement, which has the four cases for the function. To represent that, we'll define a **visitor class** with four methods, one for each variant:

```
    class HasNot implements Visitor<Boolean> {
        public Boolean onVar(Var v) { return false; }
        public Boolean onNot(Not n) { return true; }
        public Boolean onAnd(And a) { return hasNot(a.left) || hasNot(a.right);
}
        public Boolean onOr(Or o) { return hasNot(o.left) || hasNot(o.right); }
    }
```

Note the Visitor interface that this class implements. This interface describes the general pattern for writing cases of recursive functions over the variants of Formula:

```
    public interface Visitor<R> {
        public R onVar(Var v);
        public R onNot(Not n);
        public R onAnd(And a);
        public R onOr(Or o);
    }
```

It's parameterized by type R, which is the return value of the function you're defining. hasNot() returns a boolean, so HasNot implements Visitor<Boolean>.

Now we need a mechanism for invoking this visitor class – something that switches on the actual type of the formula object and calls the appropriate case in our visitor object. The trick here is to use the Interpreter pattern to invoke the visitor, by adding a new method to the datatype, conventionally called *accept*:

```
    public interface Formula {
        ...
        public <R> R accept(Visitor<R> v);
    }

    public class Var implements Formula{
        ...
        public <R> R accept(Visitor<R> v) { return v.onVar(this); }
    }

    public class Not implements Formula{
        ...
        public <R> R accept(Visitor<R> v) { return v.onNot(this); }
    }

    public class And implements Formula{
        ...
        public <R> R accept(Visitor<R> v) { return v.onAnd(this); }
    }

    public class Or implements Formula{
        ...
        public <R> R accept(Visitor<R> v) { return v.onOr(this); }
    }
```

Study accept() closely, because some magic is happening there. First a minor piece of Java syntax: in order to allow accept() to handle visitors of any return type, we need to parameterize the entire accept() method by the type variable R, so <R> appears at the start of the signature. But the key magic of the visitor pattern is what happens in the body of the method. Each case of the accept function chooses a different method of the visitor object to invoke: onVar, onNot, onAnd, onOr. And it always passes along "this," which has a declared type corresponding to the concrete class in which it appears (Var, Not, And, or Or, respectively), rather than the abstract type Formula. So the Java compiler can look at the onVar call inside Var and say – yes! this is guaranteed to be a Var here, so it's safe for you to pass it here. We have static type checking.

Finally, we put all the pieces together with a static method hasNot:

```
public static boolean hasNot(Formula f) {
    return f.accept(new HasNot());
}
```

Here, we make the visitor object that contains the code for the cases of the hasNot function, and then dispatch to it through accept(). Walk through the code for a case like

And(Not(Var("x")), Var("y"))

to make sure you understand how it works. Draw a snapshot diagram of the formula first, so that you can keep track of what's going on.

A few tweaks

Here are a few ways to tune the visitor pattern to make it more readable in Java:

- Since we may have several recursive data types that need to use the visitor pattern, it helps to nest the Visitor interface inside the datatype that needs it, in this case Formula.
- We can exploit Java's method overloading – using the same name for different methods that take different types – to change onVar, onNot, etc. to all use the same name, "on".
- Finally, we don't have to declare the HasNot visitor as a named class; we can use Java's anonymous class syntax instead, and put it right inside the hasNot() method, the only place where it's used.

Here's the result:

```
public interface Formula {
    public interface Visitor<R> {
        public R on(Var v);
        public R on(Not n);
        public R on(And a);
        public R on(Or o);
    }
    public <R> R accept(Visitor<R> v);
}
...
public static boolean hasNot(Formula f) {
    return f.accept(new Formula.Visitor<Boolean>() {
        public Boolean on(Var v) { return false; }
        public Boolean on(Not n) { return true; }
        public Boolean on(And a) { return hasNot(a.left)
                                        || hasNot(a.right); }
        public Boolean on(Or o) { return hasNot(o.left
                                        || hasNot(o.right); }
    });
}
```

There's also a performance tweak. If you trace through a invocation of hasNot() on a formula tree, you'll notice that every time hasNot() is called recursively, it creates a new visitor object, which then goes away when hasNot() returns. Since all these objects are identical and immutable, this is unnecessary overhead that can be eliminated simply by creating the object once and reusing it:

```
public static boolean hasNot(Formula f) {
    return f.accept(HAS_NOT_VISITOR);
}
private static Formula.Visitor<Boolean> HAS_NOT_VISITOR
  = new Formula.Visitor<Boolean>() {
    public Boolean on(Var v) { return false; }
    public Boolean on(Not n) { return true; }
    public Boolean on(And a) { return hasNot(a.left) || hasNot(a.right); }
    public Boolean on(Or o) { return hasNot(o.left) || hasNot(o.right); }
};
```

If hasNot() is used rarely, however, the cost in readability may not be worth the performance benefit. This performance tweak is also incompatible with having multiple arguments to the function, as we'll see next.

Passing parameters to a visitor

hasNot() is a function with only one argument: the Formula. What if we need additional arguments, like eval(Formula, Env)? We can make these additional arguments available to the cases of the function by storing them as fields of the visitor object:

```
    public static boolean eval(Formula f, Env env) {
        return f.accept(new Eval(env));
    }

    static class Eval implements Formula.Visitor<Boolean> {
        private Env env;
        public Eval(Env env) { this.env = env; }
        public Boolean on(Var v) { return env.lookup(v.name); }
        public Boolean on(Not n) { return !eval(n.f, env); }
        public Boolean on(And a) { return eval(a.left, env) && eval(a.right, 
env); }
        public Boolean on(Or o)  { return eval(o.left, env) || eval(o.right, 
env); }
    }
```

In Java, we can also express this visitor as an anonymous class:

```
    public static boolean eval(Formula f, final Env env) {
        return f.accept(new Formula.Visitor<Boolean>() {
            public Boolean on(Var v) { return env.lookup(v.name); }
            public Boolean on(Not n) { return !eval(n.f, env); }
            public Boolean on(And a) { return eval(a.left, env)
                                           && eval(a.right, env); }
            public Boolean on(Or o)  { return eval(o.left, env)
                                           || eval(o.right, env); }
        });
    }
```

Key to note here is that any local variables declared *final* are automatically in scope for the anonymous class. So we didn't have to do anything special to make env available to the cases of the visitor; Java handles all that automatically.

Different Perspectives on Visitor

Visitor as a kind of type switch

Compare our final code for hasNot() with the original switch statement we wanted to write – the structure is identical. Essentially we have implemented our own type dispatch – a mechanism for looking at the actual type of an object and choosing which piece of code to run in response. That's what a switch statement does for an enumerated type; that's what calling a method through an interface ordinarily does for methods defined on the interface itself; what we've done with visitor is created our own version of that, which allows the pieces of code to be stored in a separate object.

There are other ways we can use this type-dispatch mechanism as well. For example, we can use it for *multiple dispatch* -- choosing a piece of code to run based on the types of more than one argument. Suppose you have a function that takes *two* Formulas as arguments – can we dispatch all 4x4 = 16 cases of this function, without using instanceof? Yes we can. Fasten your seatbelts, let's implement a function that tests whether two formulas are syntactically identical.

same: Formula x Formula -> boolean

same(Var(name1:String), Var(name2:String)) = (name1 = name2)

same(Var(name1:String), Not(formula2:Formula)) = false

...

same(Not(formula1:Formula), Var(name2:String)) = false

same(Not(formula1:Formula), Not(formula2:Formula)) = same(formula1, formula2)

…

We've only shown four cases here, the other 12 are similar. The code for this starts out like this:

```
    public static boolean same(final Formula f1, final Formula f2) {
        return f1.accept(new Formula.Visitor<Boolean>() {
            public Boolean on(final Var v1) {
                return f2.accept(new Formula.Visitor<Boolean>() {
                    public Boolean on(Var v2) { return v1.name
                                                    .equals(v2.name); }
                    public Boolean on(Not n2) { return false; }
                    public Boolean on(And a2) { return false; }
                    public Boolean on(Or o2)  { return false; }
 ...
```

Frankly multiple dispatch is rarely needed in practice (and equality tests tend to be written with instanceof, because they can't assume that both objects are Formulas to begin with). But the visitor pattern can do it.

Visitor as a function over a datatype

Our original reason for the Visitor pattern in this lecture was to solve the problem of representing a function over a recursive datatype. But one interesting feature to call out here is that in the final pattern, we have an *object* that represents a function. A visitor object is not a state machine, it's not a value of an abstract data type, it's a new kind of beast – an object that is essentially represents a method. Like all objects, this method-as-an-object is *first class*, which means that we can name it with a variable, pass it around, store it in data structures, and return it from other methods:

hasnot = new HasNotVisitor();

To actually call the visitor, we hand it to accept(). Notice the minor difference in syntax:

f.eval();		// if eval uses the interpreter pattern

f.accept(hasnot); // if hasNot uses the visitor pattern

But where hasnot is a reference to a first-class object, eval is just a method name. We can't store a reference to it or treat it like data.

Visitor as a kind of iterator

You may have noticed that the visitor pattern for Formula effectively iterates over the formula tree, visiting each node. That's why it's called the visitor pattern, in fact.

Compare an iterator over a collection:

```
// compute the sum of a list
List<Integer> lst = ...;
int sum = 0;
for (int x: lst) {
    sum = sum + x;
}
```

with a visitor over a recursive data type:

```
// count the variables in a formula
Formula f = ...;
f.accept(new Formula.Visitor<Integer> () {   // aka "for each node..."
    public Integer on(Var v) { return 1; }
    public Integer on(Not n) { return n.f.accept(this); }
    public Integer on(And a) { return a.left.accept(this)
```

```
                                              + a.right.accept(this); }
            public Integer on(Or o)   { return o.left.accept(this)
                                              + o.right.accept(this); }
        }
```

The iteration steps through a collection, iteratively assigning the variable *x* to different elements, and executing the body of the *for* loop once for each element. The visitor pattern steps through the formula tree, assigning one of the variables v, n, a, or o to each node (depending on its type), and executing the appropriate case for that type.

Unlike Iterator, which basically steps through all the elements in the collection in an order that's up to the implementer of the collection, the Visitor pattern gives the client more control over navigation. Our hasNot visitor was able to choose not even to drill down into Not's subformula (since it doesn't affect the final answer), and its And/Or cases could look at their left or right subtrees in either order.

Visitors aren't limited to pure functions with no side-effects or local state. A visitor object can be a state machine that updates itself as it traverses the tree, in the similar way that the for loop updates the sum variable as it steps through the collection. Consider an operation that finds all the variables in a formula and accumulates them in a Set<String>:

```
public static Set<String> varSet (Formula f) {
    VarSet vs = new VarSet();
    f.accept(vs);
    return vs.vars;
}

private class VarSet implements Formula.Visitor<Void> {
    public final Set<String> vars = new HashSet<String>();
    public Void on(Var v) { vars.add(v.name); return null; }
    public Void on(Not n) { n.f.accept(this); return null; }
    public Void on(And a) { a.left.accept(this); a.right.accept(this);
                            return null; }
    public Void on(Or o)  { o.left.accept(this); o.right.accept(this);
                            return null; }
}
```

VarSet is a state machine whose transitions are the four on() methods. Two things to clarify here: first, since the on methods have become mutators, we don't care about their return value anymore, so we use the Java builtin class Void (which is the object type that corresponds to the void primitive type). This type has no values, but Java still requires a return value, so we return null. Second, the recursive calls in the cases for Not, And, and Or want to use the *same* visitor object for the recursive part of the traversal, so we can't just call varSet() recursively. Instead we directly apply the visitor object (this!) to the subtree, using its accept method.

Here's a more compact way to write the same operation, using an anonymous class so that all the code is contained within varSet():

```
public static Set<String> varSet (Formula f) {
     final Set<String> vars = new HashSet<String>();
     f.accept(new Formula.Visitor<Void>() {
         public Void on(Var v) { vars.add(v.name); return null; }
         public Void on(Not n) { n.f.accept(this); return null; }
         public Void on(And a) { a.left.accept(this); a.right.accept(this);
                                 return null; }
         public Void on(Or o)  { o.left.accept(this); o.right.accept(this);
                                 return null; }
     });
     return vars;
 }
```

Dual nature of Interpreter and Visitor

Interpreter and Visitor are duals of each other, in the following sense. Let's think about a recursive type in terms of its operations (the rows) and its variant classes (the columns). Each operation needs a piece of code to handle each variant, so each cell in this table is essentially a method body somewhere.

	Var	Not	And	Or	*ThereExists?*	*ForAll?*
eval						
hasNot						
same						
varSet						
toCNF?						

Interpreter is column-centric: it groups all the code for a variant together in one place, inside the variant class. Visitor is row-centric: it groups all the code for a single operation together in one place, inside the visitor class.

Interpreter makes it easy to add a new variant – you just create a new variant class with all the cases for that variant in it -- but hard to add a new operation, because you'll be touching every single variant class to do it. Every variant has to have a complete list of the operations that use the Interpreter pattern.

Visitor, by contrast, makes it easy to add a new operation, as a new visitor class, but hard to add a new variant. Every visitor has to have a complete list of all the variants.

So one design tradeoff between these two patterns boils down to **which kind of change you expect** in the future. Is the set of variants pretty fixed, but clients are likely to want to invent new operations? Then you want visitors. Or are clients likely to need new variants – maybe even variants that the client creates themselves? Then you want interpreters.

Interpreter: function as methods

- Has privileged access to reps of the variant classes
- Harder to understand the function, because it's split over multiple files
- Adding a new function requires changing every variant class
- Use for operations that are essential to the abstract data type

Visitor: function as visitor object

- Reduces rep independence: variants must be visible to client
- Need observer operations to get parts of the rep
- Function is found all in one place, so is easier to understand & change
- Adding a new variant requires changing every visitor class
- Use for operations that are essential to some client's processing

Both patterns are statically checked, though!

Summary

We've looked at the notion of **design patterns**, which are standard solutions to common design problems. Some design patterns are formalized in programming languages, and some need to be brought in by a programmer. Design patterns have names that are worth committing to memory (factory method, interpreter, visitor) so that you can communicate with other programmers.

We've also delved deeply into the pros and cons of two design patterns for recursive data types, interpreter and visitor. Interpreter is well-supported by Java language features; visitor requires us to implement more of the machinery ourselves. But the benefits that you get from a visitor, in the form of pulling together the code for a single operation in one place, with full static type checking, are extremely powerful.