Design Patterns on Recursive Data Types
Spring 2013
Today’s Topics

Review: Recursive Data Types
Design Patterns
Design Patterns On ASTs: Interpreter and Visitor
Recursive Data Types

Data type definition for ImList
- ImList = Empty + Cons(first:E, rest:ImList)

ImList appears on both the left and right sides
- ImList is a recursive data type

Another example: binary tree
- Tree = Empty + Node(element:E, left:Tree, right:Tree)
Functions on Recursive Data Types

Using data type definitions makes defining operations easier

- Just think of operations in terms of one case per variant

Example: size()

- size: ImList → Int
- Data type definition: ImList = Empty + Cons(first:E, rest:ImList)
- So, two cases
  - size(Empty) = 0
  - size(Cons(first:E, rest:ImList)) = 1 + size(rest)
- This recursive definition leads naturally to simple, understandable, recursive code
Boolean Satisfiability (SAT)

Given a formula made up of boolean variables and operators $\land$ (and), $\lor$ (or), $\neg$ (not), find an assignment of variables that makes the formula true

Example: $(P \lor Q) \land (\neg P \lor R)$
- Not true if $P=false$, $Q=false$, $R=false$
- Satisfied when $P=false$, $Q=true$, $R=true$
- Other solutions as well (E.g. $P=true$, $Q=false$, $R=true$)

Conjunctive Normal Form (CNF)
- Write a boolean formula as a “product of sums”
- i.e. Each clause is a sum (just $\lor$) and the clauses are combined with $\land$
- Standard way to write boolean formulas
Recursive Data Type for SAT

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

Example: \((P \lor Q) \land (\neg P \lor R)\)

\[
\land (\lor (\neg P, \lor (P, R)) )
\]

This is an abstract syntax tree for SAT
Implementing Satisfiability

We’ll use the simplest possible strategy: enumerate all variables, then try every boolean value for each variable

1. Extract the set of variables from a formula
2. Try all assignments of true/false for each variable
   • Represent the assignment with a data type Environment, which just maps each variable to its current assignment
3. Evaluate the boolean expression for each environment
4. If an environment results in the expression being true, return the assignment

Create functions functions over our Formula data type

- satisfiable: Formula → boolean
- getAllVariables: Formula → Set<Var>
- eval: Formula, Environment → boolean
Implementing Satisfiability

Easy to see how eval works (assume Environment is just a Java Map)

- eval(Var(name:String), Environment) = Environment.get(name)
- eval(Not(formula:Formula), Environment) = !eval(formula, Environment)
- eval(And(left:Formula, right:Formula), Environment) = eval(left, Environment) && eval(right, Environment)
- eval(Or(left:Formula, right:Formula), Environment) = eval(left, Environment) || eval(right, Environment)
Code: eval()
Evaluating eval()

This approach works well for telling us whether a given assignment of variables evaluates to true.

What about adding additional methods?

- E.g. hasNot: Formula → boolean
  - hasNot(Var(name:String)) = false
  - hasNot(Not(f:Formula)) = true
  - hasNot(Or(left:Formula, right:Formula)) = hasNot(left) || hasNot(right)
  - hasNot(And(left:Formula, right:Formula)) = hasNot(left) || hasNot(right)

- Or countVars() or …

Our interface would need to have all of these methods

- Introducing methods only applicable to a very small number of clients
Evaluating `eval()`

Another problem: code is spread out between multiple classes

- Even worse, in Java it’s multiple files
- Understanding algorithm requires reading code in multiple files
- Changing algorithm may require changing all classes
Interpreter Design Pattern

Our approach for eval() has a standard name: the Interpreter pattern

Define a language
- In our case, a language for boolean expressions

Create separate class for each kind of object in our ADT

Each class implements a method to define how that object should be interpreted
- And the methods are recursively called
- Functional: the data is not mutated
Design Patterns

Idea that began in architecture

Pattern = common problem + an accepted solution

- Everything from construction materials to how common certain kinds of businesses should be
- Not an absolute guide, but ideas based on experience

Adopted by software engineering community

- Particularly OOP
- Patterns describe common design problems & accepted solutions
- Began with Beck, Cunningham, but made most famous by GOF book
Patterns You’ve Seen

Encapsulation/Separation of Concerns
Iterator
Exception
State Machine
Lexer+Parser
Abstract Syntax Tree
Sentinel Objects
Why Design Patterns

A toolbox for sub-problems likely to encounter when building new systems
- Instead of reinventing wheel, look at what others have done in similar situations
- Codify standard practices

A vocabulary for talking about software design
- Help communicate with other programmers
- Help convey design decisions or agreed-upon interfaces between parts
Factory Pattern

“Define an interface for creating an object, but let the classes that implement the interface decide which class to instantiate.” [GOF]

- Allows us to defer class of what’s actually being instantiated

Last time, we forced programmers to use new Empty() instead of empty() for ImList

- This pattern is a way to “solve” this, though not perfect solution
Factory Pattern & \textit{ImList}

```java
public static <E> ImList<E> empty() {
    return new Empty<E>();
}
```

Now have a method that doesn't require users to know about \textit{Empty} class (better rep independence)

- But where does this method go?
  - In Java, create a new class: \textit{ImListFactory}
  - Only static methods
  - Each static method returns a new list

- How would we classify this method?
- Other useful factory methods?
Design Patterns On ASTs

Today we’re going to talk about Interpreter and Visitor

- Interpreter is well-supported by Java via interfaces & classes implementing the methods required
- Visitor is fundamental pattern in some other languages like Scala, Haskell & ML

One problem with Interpreter: code distributed across many classes & files

- Idea in Visitor: bring all the code into a one place
- Make it easier to add new operations
Visitor: Attempt #1

We’d like to write some code that looks like this:

```java
public static boolean hasNot(Formula f) {
  // this won't compile!
  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);
  }
}
```

Won’t work--- Java switch statement doesn’t work for choosing between object types.
Visitor: Attempt #2

```java
public static boolean hasNot(Formula f) {
    if (f instanceof Var) { return false; }
    if (f instanceof Not) { return true; }
    if (f instanceof Or) {
        Or o = (Or)f;
        return hasNot(o.left) || hasNot(o.right)
    }
    if (f instanceof And) {
        And a = (And)f;
        return hasNot(a.left) || hasNot(a.right)
    }
}
```

This is legal Java, but considered poor style

- **Downcasts** the object after checking its actual (runtime) type
- Gives up on static type checking--- Java won’t ensure new classes implementing Formula will be covered by hasNot
- In general, **avoid instanceof** except in equals() (Wednesday)
Visitor: Let’s Try Again

First, we’ll create an interface for recursive functions over Formula.

Then, our visitors will implement this interface.

Let’s look at the code
Visitor: Modifying Formula

Need some way to invoke the visitor & to dispatch to the correct onSomething() method

We’ll use something like the Interpreter pattern to invoke the visitor

- Modify the Formula interface to support visitors
- Each class implementing it also implements the method required to make visitors work
Cleaning Things Up

The Visitor interface we implemented is specific to Formula, so we should nest it within Formula

- Other ADTs may have their own visitor interfaces/implementations

We can use method overloading to change the interface methods from onFoo to just on()

- Overloaded on the passed-in class
- Now a class implementing visitor just implements a bunch of on() methods
Passing Arguments to a Visitor

What if we wanted to pass in an argument, like in `eval()`?

- Store the argument as an instance variable of the visitor
- Code in the visitor can reference the instance variable
Ways to Think About Visitor

Visitor == switch() over types
➢ Structure of hasNot() using a visitor is almost the same as the original switch statement we wanted
➢ Implemented a mechanism for type dispatch (choosing what code to run based on the actual type of the object)

Visitor == function over a data type
➢ A visitor object represents a function over a recursive data type
➢ We can instantiate this object and pass it around with references
➢ Different from method calls we’ve seen so far
Ways to Think About Visitor

**Visitor == special iterator**

- The visitor iterates over the AST, processing each node once
- Client controls the iteration order, unlike iterator
- Visitors can mutate themselves, though we haven’t seen an example of this
Visitor and Interpreter can be thought of as duals

- Interpreter: code grouped by column, in that all the code for a variant is contained in that variant class
- Visitor: code grouped by row, in that all the code for an operation is grouped together

- Interpreter makes it easy to add new variants
- Visitor makes it easy to add new operations

Which to use?
Summary

Design patterns are accepted solutions to common scenarios & problems. Useful for communicating among programmers & not having to reinvent the wheel.

Visitor & Interpreter: two common design patterns over recursive data types (and particularly ASTs).

Use design patterns in thinking about code at program level, data structure level, module level, and method level.
A High-Level Review

We’ve now talked about designing & implementing code at 3 different levels: whole programs, ADTs, and methods

Implementing a method

- Design: write a good signature and specification, thinking about how a client will use the method. Be wary of over- or under-constraining the spec. Use design patterns where appropriate.
- Test: Design & write test cases that cover the spec. Make sure not implementation-specific.
- Code: Implement the method, making sure to add test cases as necessary to cover all code paths. Make sure glass box tests adhere to specification.
Implementing an ADT

Design the ADT

- Reuse where appropriate (including using design patterns)
- Choose operations (constructors, producers, mutators, and observers)
- If the type is mutable, draw a state machine to think about the important states & transitions
- If the type is recognizing, generating, or working on a structured language, write out the grammar
- If the type is recursive, write out a data type definition and convert into interfaces & classes. Think about operations in terms of variants.
- Write specs and signatures for each operation

Test

- Write test cases to cover the specs, and state transitions if applicable

Code

- Choose rep, and write down RI and AF
- Implement methods using specs
- Run & revise test cases based on your code
Designing A Program

Choose data structures
➢ Reuse when appropriate, compose new ones otherwise

Divide into steps
➢ Think about the overall process as a series of steps, each with one or more modules

Implement module-by-module
➢ Draw on design patterns to find solutions to common problems

Put it together
➢ Incrementally!
➢ Run all tests on any change
Summary

Design patterns are accepted solutions to common scenarios & problems. Useful for communicating among programmers & not having to reinvent the wheel.

Visitor & Interpreter: two common design patterns over recursive data types (and particularly ASTs).

Use design patterns in thinking about code at program level, data structure level, module level, and method level.