Today’s Topics

Functionals
- Objects representing executable code
- Visitor Design Pattern

Domain-specific languages
- Music language – subset of ABC player of Project 2
Consider a datatype representing language syntax

- Formula is the language of propositional logic formulas
- a Formula value represents program code in a data structure; i.e.
  
  ```
  new And(new Var("x"), new Var("y"))
  ```
  has the same semantic meaning as the Java code
  ```
  x && y
  ```

- but a Formula value is a **first-class** object
  - first-class: a value that can be passed, returned, stored, manipulated
  - the Java expression “x && y” is *not* first-class
Want to define a new operation **calcCor** on **WorldElement** subclass objects

- Need to provide implementations for each of the subclasses -- harder
Visitor Design Pattern

Set up WorldElement so arbitrary operations can be added easily

- Factor out the operations in a separate hierarchy of classes, all extending the abstract class Visitor
  - Each subclass represents one operation
Visitor Design Pattern

Two levels of dispatch
Visitor abstract class can provide skeletal implementation for each WorldElement subclass
Two Levels of Dispatch

WorldElement elm = ...
Visitor vis = ...

elm.Accept(vis);

cableObj.Accept(vis)
vis.VisitCable(cableObj)

PlanObj.Accept(vis)
vis.VisitPlanet(planObj)

First dispatch

GravVObj.VisitPlanet(planObj)
CorVObj.VisitPlanet(planObj)

Second dispatch

GravVObj.VisitCable(cableObj)
CorVObj.VisitCable(cableObj)
GravVObj.VisitPlanet(planObj)
CorVObj.VisitPlanet(planObj)
Overloading

Just syntactic sugar
Do not use if you find it confusing!
Duality Between Interpreter and Visitor

**Operation using interpreter pattern**
- Adding new operation is hard (must add a method to every existing class)
- Adding new class is easy (changes only one place: the new class)

**Operation using visitor pattern**
- Adding new operation is easy (changes only one place: the new visitor)
- Adding new class is hard (must add a method to every existing visitor)
We can now write code like:

```java
for(i = 0; i < WElements.size(); i++) {
    elm = WElements.get(i);
    for(j = 0; j < visitorsVector.size(); j++) {
        vis = visitorsVector.get(j);
        elm.Accept(vis);
    }
}
```

The above code does not have to change even if we add new `WorldElement` subclasses or new Visitor subclasses (operations). Adding new operations is easy: Just add a new Visitor. Adding new types of elements is harder: Need to add visit methods for each Visitor subclass for each new element.
Visitor Summary

A Visitor can visit elements that are unrelated through inheritance. In our example Cable and Planet do not need to be related.

A Visitor can accumulate state as they visit each element of an object structure in an instance variable of the visitor. In normal operations such state would have to be passed as arguments to the operation.

A visitor represents code as a first-class object, too

- A visitor is an **object** that can be passed around, returned, and stored
- But it’s also a **function** that can be invoked
Interesting music tends to have a lot of repetition

- Let’s look at rounds, canons, fugues
- A familiar simple **round** is “Row Row Row Your Boat”: one voice starts, other voices enter after a delay

  Row row row your boat, gently down the stream, merrily merrily ...

  Row row row your boat, gently down the stream...

**Recall our MIDI piano from early lectures**

- A song could be represented by Java code doing a sequence of calls on a state machine:

  machine.play(E); machine.play(D); machine.play(C); ...

- We want to capture the code that operates this kind of machine as first-class **data objects** that we can manipulate, transform, and repeat easily

- **What does our Music data type look like?**
Let’s start by representing simple tunes

Music = Note(duration:double, pitch:Pitch, instr:Instrument)
+ Rest(duration:double)
+ Concat(m1:Music, m2:Music)

- duration is measured in beats
- Pitch represents note frequency (e.g. C, D, E, F, G; essentially the keys on the piano keyboard)
- Instrument represents the instruments available on a MIDI synthesizer

Design questions
- Is Concat a tree or a list? what would it look like defined the other way?
  - Concat as defined above is a tree
  - Concat = List<Note + Rest> would be a list
A Few of Music’s Operations

notes : String x Instrument → Music

requires string is in a subset of abc music notation

  e.g. notes(“E D C D | E E E2 |”, PIANO)

  1 beat note       2-beat note

duration : Music → double

  returns total duration of music in beats

  e.g. duration(Concat(m1, m2)) = duration(m1) + duration(m2)

transpose : Music x int → Music

  returns music with all notes shifted up or down in pitch by the given
  number of semitones (i.e., steps on a piano keyboard)

play : Music → void

  effects plays the music
public class RowYourBoatSimple {

    public static void main(String[] args) throws MidiUnavailableException {

        Music rowYourBoat = 
        notes("C C C3/4 D/4 E |
            +"E3/4 D/4 E3/4 F/4 G2 |
            +"C'/3 C'/3 C'/3 G/3 G/3 G/3 E/3 E/3 E/3 C/3 C/3 C/3 |
            +"G3/4 F/4 E3/4 D/4 C2",
            PIANO); 
        Music rowTwice = 
        concat(rowYourBoat, transpose(rowYourBoat, Pitch.OCTAVE));
        new MusicPlayer().play(rowTwice);
    }
}

RowYourBoatSimple.java
Interfaces

public interface Music {
    // @return total duration of this piece
    double duration();

    // @requires v != null
    // @returns result of visitor function v
    <T> T accept(Visitor<T> v);
}

// Visitor represents a function over the Music datatype,
public interface Visitor<T> {
    T visit(Rest m); // could be visitRest to avoid overloading
    T visit(Note m);
    T visit(Concat m);
    T visit(Together m);
    T visit(Forever m);
}

<T> added to tell Java that we are using generics so it will compile!
public class Concat implements Music {
    private final Music first;
    private final Music second;
    // Rep invariant: m2 eventually plays if it's nontrivial
    private void checkRep() {
        assert first != null; assert second != null;
        assert first.duration() < Double.POS_INF || second.duration() == 0;
    }

    // Make a Music sequence that plays m1 followed by m2.
    public Concat(Music m1, Music m2) {
        this.first = m1;
        this.second = m2;
        checkRep();
    }
}
public class MusicPlayer {
    public void play(Music m) throws MidiUnavailableException {
        m.accept(new Visitor<Void>() {
            private Midi midi = new Midi();

            public Void visit(Concat m) {
                m.first().accept(this);  m.second().accept(this);
                return null;
            }
            public Void visit(Note m) {
                midi.play(m.pitch().difference(C) + 60,
                            (int) (m.duration() * MSEC_PER_BEAT), m.instrument());
                return null;
            }
            public Void visit(Rest m) {
                midi.wait((int) (m.duration() * MSEC_PER_BEAT));  return null;
            }
        });
    }
}
Multiple Voices

For a round, the parts need to be sung simultaneously

Music = Note(duration:double, pitch:Pitch, instr:Instrument)
   + Rest(duration:double)
   + Concat(m1:Music, m2:Music)
   + Together(m1:Music, m2:Music) // two concurrent threads

➢ Here’s where our decision to make Concat() tree-like becomes very useful
   • Suppose we instead had:
     Concat = List<Note + Rest>
     Together = List<Concat>
   • What is the limitation of this design?
     We would not be able to concat two Togethers
Multiple Voices

We need one more operation:
\[ \text{delay} : \text{Music} \times \text{double} \rightarrow \text{Music} \]
\[ \text{delay}(m, \text{dur}) = \text{concat}(\text{rest}(\text{dur}), \text{m}) \]

And now we can express Row Row Row Your Boat
\[ \text{rrryb} = \text{notes}(\text{"C C C3/4 D/4 E | E3/4 D/4 E3/4 F/4 G2 | ..."}, \text{PIANO}) \]
\[ \text{together}(\text{rrryb}, \text{delay}(\text{rrryb}, 4)) \]
- Two voices playing together, with the second voice delayed by 4 beats

- Can create rounds, canons, fugues using these operations!

- See music-language package for more details
public class Together implements Music {
    private Music top, m2;

    private void checkRep() {
        assert top != null;
        assert m2 != null;
    }

    // Make a Together of two pieces of music.
    public Together(Music m1, Music m2) {
        this.top = m1;
        this.m2 = m2;
        checkRep();
    }

    ...
}
public class MusicPlayer {
    public void play(Music m) throws MidiUnavailableException {
        m.accept(new Visitor<Void>() {
            private Midi midi = new Midi();

            public Void visit(final Together m) {
                final Visitor<Void> thisVisitor = this;
                final Music shorter, longer;
                if (m.top().duration() < m.bottom().duration)
                    { shorter = m.top(); longer = m.bottom(); }
                else  {  shorter = m.bottom(); longer = m.top(); }
                new Thread(new Runnable() {
                    public void run() {   shorter.accept(thisVisitor); }
                    }).start();
                longer.accept(this); return null;
            }
        });
    }
}

Why are we playing the shorter one in the spawned thread?
Little Languages

We’ve built a new language embedded in Java

- Music data type and its operations constitute a language for describing music generation
- Instead of just solving one problem (like playing Row Row Row Your Boat), build a language or toolbox that can solve a range of related problems (e.g. Pachelbel’s canon)
- This approach gives you more flexibility if your original problem turns out to be the wrong one to solve (which is not uncommon in practice!)
- Capture common patterns as reusable abstractions

Formula was an embedded language too

- Formula combined with SAT solver is a powerful tool that solves a wide range of problems