Visitors and Embedded Languages

Spring 2012
Back to our equation solver

Expression :=

\textbf{NumExpr} (\texttt{val}:\texttt{double}) +
\textbf{VarExpr} (\texttt{var}:\texttt{String}) +
\textbf{PlusExpr} (\texttt{left}:Expression, \texttt{right}:Expression) +
\textbf{MinusExpr} (\texttt{left}:Expression, \texttt{right}:Expression) +
\textbf{TimesExpr} (\texttt{left}:Expression, \texttt{right}:Expression) +
\textbf{DivExpr} (\texttt{left}:Expression, \texttt{right}:Expression) 

\begin{align*}
Expression & = Expression \\
\Rightarrow & \\
Var & 
\end{align*}
Back to our Equation Solver

solve(Expression e, Expression res)

march e:

VarExpr( var ): return res

PlusExpr(l, r): if(l.hasVar())

return solve(l, MinusExpr(res, r))

else

return solve(r, MinusExpr(res, l))

MinusExpr(l, r): if(l.hasVar())

return solve(l, PlusExpr(res, r))

else

return solve(r, MinusExpr(l, res))

TimesExpr(l, r): if(l.hasVar())

return solve(l, DivExpr(res, r))

else

return solve(r, DivExpr(res, l))

...

You can now implement this with the interpreter pattern!
Issues with the interpreter pattern

Pros
➢ It’s easy to implement  
➢ Code is simple and pretty

Cons
➢ Every time you want to add a new function you have to modify all your variant classes!
➢ Imagine if every time you wanted to implement a function over a list you had to go into the source code of the List library and modify it.

This is a common problem, so yes, there is a design pattern to address it!
Visitor Design Pattern

Set up **Expression** 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 Expression subclass
Two Levels of Dispatch

Expression \( expr = \ldots \)
Visitor \( vis = \ldots \)

\[ \text{expr.Accept}(vis); \]

First dispatch

- \( \text{NumExpr.Accept}(vis) \)
  - \( \text{vis.VisitNum}(\text{NumExpr}) \)

Second dispatch

- \( \text{PlusExpr.Accept}(vis) \)
  - \( \text{vis.VisitPlus}(\text{planObj}) \)
  - \( \text{PrintVisitor.VisitPlus}(\text{planObj}) \)
  - \( \text{SolveVisitor.VisitPlus}(\text{planObj}) \)

- \( \text{PrintVisitor.VisitNum}(\text{NumExpr}) \)
  - \( \text{SolveVisitor.VisitNum}(\text{NumExpr}) \)
Overloading

Abstract class `Expression`:
- `Accept(Visitor v)`
  - `v.Visit(this)`

Abstract class `Visitor`:
- `Visit(NumExpr c)`
- `Visit(PlusExpr p)`

Visitor classes:
- `PrintVisitor`:
  - `VisitNumExpr(NumExpr c)`
  - `VisitPlusExpr(PlusExpr p)`
- `SolveVisitor`:
  - `Visit(NumExpr c)`
  - `Visit(PlusExpr p)`

Diagram:
- 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)
Representing Code as Data

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

We can now write code like this:

```java
for (Expression expr : exprList) {
    for (Visitor vis : visitorList) {
        expr.Accept(vis);
    }
}
```

Today’s lecture will see more examples of code as data
Today’s Problem: Music

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...
- Bach was a master of this kind of music

Example: A simple player

- A song could be represented by Java code doing a sequence of method calls
  - 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
Music Data Type

Let’s start by representing simple tunes

\[ \text{Music} = \text{Note}(\text{duration:double, pitch:Pitch, instr:Instrument}) \]
\[ + \text{Rest}(\text{duration:double}) \]
\[ + \text{Concat}(\text{m1:Music, m2:Music}) \]

- duration is measured in \textit{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 this a tree or a list? what would it look like defined the other way?
- what is the “empty” Music object?
  - it’s usually good for a data type to be able to represent \textit{nothing}
  - avoid null
- what are the rep invariants for Note, Rest, Concat?
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

all these operations also have precondition that parameters are non-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>
  • We would not be able to concat two Togethers

Composite pattern

> The composite pattern means that groups of objects (composites) can be treated the same way as single objects (primitives)
> \[ T = C_1(...,T) +...+ C_n(...,T) + P_1(...) +...+ P_m(...) \]

Music and Formula are composite data types.
Simple Rounds

We need one more operation:

delay : Music x double → Music

delay(m, dur) = concat(rest(dur), m)

And now we can express Row Row Row Your Boat

together(rrryb, delay(rrryb, 4))

• Two voices playing together, with the second voice delayed by 4 beats

This pattern is found in all rounds, not just Row Row Row Your Boat

Abstract out the common pattern

round : Music x double x int → Music

round(m, dur, n) = m if n == 1

together(m, round(delay(m, dur), dur, n-1)) if n > 1

The ability to capture a general pattern like round() is one of the advantages of music as a first-class object rather than merely a sequence of play() calls
Distinguishing Voices

We want each voice in the round to be distinguishable

- e.g. an octave higher, or lower, or using a different instrument
- So these operations over Music also need to be first-class objects that can be passed to round()
- Fortunately operations implemented as visitors already are objects

**canon() applies a visitor to the repeated melody**

```plaintext
canon : Music x double x Visitor<Music> x int → Music

e.g. canon(rrryb, 4, new TransposeVisitor(OCTAVE), 4)
```

produces 4 voices, each one octave higher than the last

**canon() is a higher-order function**

- A higher-order function takes a function as an argument or returns a function as its result
Functional Objects

Not all operations are visitors

- Let’s generalize the idea of a music transformer function

```java
interface UnaryFunction<T,U> {
    U apply(T t);
}
```

- An instance of UnaryFunction is a **functional object**, representing some function \( f : T \rightarrow U \)

- For example:

```java
new UnaryFunction<Music,Music>() {
    public Music apply(Music m) { return delay(m, 4); }
}
```

- In general, we might want a `delayer()` method that produces a delay transformer with an arbitrary delay (not just 4 beats):

```java
delayer : int \rightarrow \text{UnaryFunction<Music,Music>}
```

**note that `delayer` is a higher-order function too**

let’s write it this way, the abstract type that `UnaryFunction` represents

this anonymous class is essentially a lambda expression producing a functional object
Counterpoint

A canon is a special case of a more general pattern

- **Counterpoint** is \( n \) voices singing related music, not necessarily delayed
  
  \[
  \text{counterpoint : Music} \times (\text{Music} \rightarrow \text{Music}) \times \text{int} \rightarrow \text{Music}
  \]

- Expressed as counterpoint, a canon applies two functions to the music:
  delay and transform

  \[
  \text{canon}(m, \text{delay}, f, n) = \text{counterpoint}(m, f \circ \text{delay}(\text{delay}), n)
  \]

**Another general pattern**

function composition \( \circ : (U \rightarrow V) \times (T \rightarrow U) \rightarrow (T \rightarrow V) \)

```java
public static <T,U,V> UnaryFunction<T,V> compose(final UnaryFunction<U,V> g,
final UnaryFunction<T,U> f) {
    return new UnaryFunction<T,V>() {
        public V apply(T t) { return g.apply(f.apply(t)); }
    };
}
```
Repeating

A line of music can also be repeated by the same voice

\[ \text{repeat} : \text{Music} \times (\text{Music} \rightarrow \text{Music}) \times \text{int} \rightarrow \text{Music} \]

e.g. \( \text{repeat}(\text{rrryb}, \text{octaveHigher}, 2) = \text{concat}(\text{rryb}, \text{octaveHigher}(\text{rryb})) \)

- Note the similarity to \text{counterpoint}():
  - counterpoint: \( m \text{ together } f(m) \text{ together } ... \text{ together } f^{n-1}(m) \)
  - repetition: \( m \text{ concat } f(m) \text{ concat } ... \text{ concat } f^{n-1}(m) \)

- And in other domains as well:
  - sum: \( x + f(x) + ... + f^{n-1}(m) \)
  - product: \( x \cdot f(x) \cdot ... \cdot f^{n-1}(m) \)

- There’s a general pattern here, too; let’s capture it
Binary Functionals

We need first-class representation for binary operations like together, concat, plus, times

```java
interface BinaryFunction<T,U,V> {
    V apply(T t, U u);
}
```

- An instance of BinaryFunction represents some \( f : T \times U \rightarrow V \)
  - together: \( \text{Music} \times \text{Music} \rightarrow \text{Music} \)
  - concat: \( \text{Music} \times \text{Music} \rightarrow \text{Music} \)

Now we can capture the pattern

\[
\text{series} : T \times (T \times T \rightarrow T) \times (T \rightarrow T) \times \text{int} \rightarrow T
\]

- initial value
- binary op
- \( f \)
- \( n \)

\[
\text{counterpoint}(m, f, n) = \text{series}(m, \text{together}, f, n)
\]

\[
\text{repeat}(m, f, n) = \text{series}(m, \text{concat}, f, n)
\]
Repeating Forever

Music that repeats forever is useful for canons

\[ \text{Forever}(m) \] plays \( m \) repeatedly, forever

\[ \text{duration}(\text{Forever}(m)) = +\infty \]

Music = Note(duration:double, pitch:Pitch, instr:Instrument) + Rest(duration:double) + Concat(m1:Music, m2:Music) + Together(m1:Music, m2:Music) + \text{Forever}(m:Music)

Here’s the Row Row Row Your Boat round, forever:

canon (\text{Forever}(rrryb), 4, \text{octaveHigher}, 4)
Accompaniment

accompany: Music \times Music \rightarrow Music

repeats second piece until its length matches the first piece

\begin{align*}
\text{melody line} & \quad \text{bass line or drum line,}\n\text{repeated to match melody’s length}
\end{align*}

\[
accompany(m, b) = \\
\begin{cases}
\text{together}(m, \text{repeat}(b, \text{identity}, \text{duration}(m)/\text{duration}(b))) & \text{if duration}(m) \text{ finite} \\
\text{together}(m, \text{forever}(b)) & \text{if duration}(m) \text{ infinite}
\end{cases}
\]
Pachelbel’s Canon

(well, the first part of it, anyway...)

pachelbelBass = notes("D,2  A,,2 | B,,2  ^F ,,  | ...  |“, CELLO)

pachelbelMelody = notes("^F’2  E’2 |  D’2  ^C’2  | ...  | ... | ... | ... | ... | ... | “, VIOLIN)

pachelbelCanon = canon(forever(pachelbelMelody),
    16,
    identity,
    3)

pachelbel = concat(pachelbelBass, accompany(pachelbelCanon, pachelbelBass))
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

- 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
## Embedded Languages

**Useful languages have three critical expressions**

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>Formula language</th>
<th>Music language</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Primitives</strong></td>
<td>3, false</td>
<td>Var, Bool</td>
<td>notes, rest</td>
</tr>
<tr>
<td><strong>Means of Combination</strong></td>
<td>+, *, ==, &amp;&amp;,</td>
<td></td>
<td>, ...</td>
</tr>
<tr>
<td><strong>Means of Abstraction</strong></td>
<td>variables, methods, classes</td>
<td>Java mechanisms</td>
<td>functional objects + Java mechanisms</td>
</tr>
</tbody>
</table>
Summary

Composite pattern
- Composite data types allow a group of objects to be treated the same as a single object

Functionals
- UnaryFunction and BinaryFunction represent functions as Java objects
- So do Runnable and Visitor, in fact

Higher-order functions
- Operations that take or return functional objects

Building languages to solve problems
- A language has greater flexibility than a mere program, because it can solve large classes of related problems instead of a single problem
- Interpreter pattern, visitor pattern, and higher-order functions are useful for implementing powerful languages
- But in fact any well-designed abstract data type is like a new language