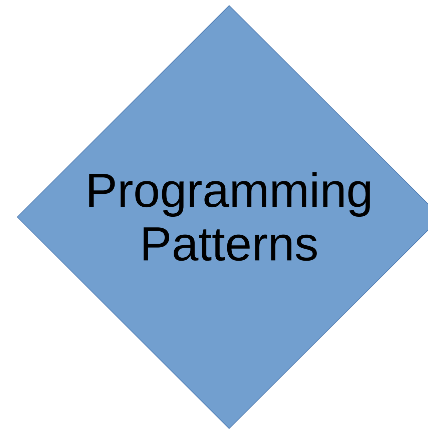
SWEN423 - Lecture 11



Beyond Java

Week5 readings

- The expression problem (Trivially, Scandinavian, Scala)
- Multiple dispatch
- Same problem: how to expand in both directions in a library setting? (that is, no editing pre existing code)
- Composite (non sealed)
  - Data-variants can be added
  - adding operations requires updating the library code
- Composite with visitor (or functional match)
  - Operations can be added
  - adding Data-variants requires updating the visitor interface

```
Library A:
  interface Node{op1, op2, op3}//Not sealed!
  record P(String text)implements Node{op1, op2, op3}
  record H1(String text)implements Node{op1, op2, op3}
  record Html(Head head,Body body)implements Node{op1, op2, op3}
  record Head()implements Node{op1, op2, op3}
  record Body(List<Div>divs)implements Node{op1, op2, op3}
  record Div(List<Node>ns)implements Node{op1, op2, op3}
  record A(String href,String text)implements Node{op1, op2, op3}
  record Ul(List<Li>lis)implements Node{op1, op2, op3}
  record Ol(List<Li>lis)implements Node{op1, op2, op3}
  record Li(Node node)implements Node{op1, op2, op3}
------------
User code:
  record H2(Node node)implements Node{op1, op2, op3} //ok
  //but, how can I add an op4 to Node and all the existing cases?
```

```
Library A: (with visitor)
  interface Node{<T> T accept(Visitor<T> v);}//Not sealed!
  record P(String text)implements Node{ ..accept..}
  record H1(String text)implements Node{..accept..}
  record Html(Head head,Body body)implements Node{..accept..}
  record Head()implements Node{..accept..}
  record Body(List<Div>divs)implements Node{..accept..}
  record Div(List<Node>ns)implements Node{..accept..}
  record A(String href,String text)implements Node{..accept..}
  record Ul(List<Li>lis)implements Node{..accept..}
  record Ol(List<Li>lis)implements Node{..accept..}
  record Li(Node node)implements Node{..accept..}
------------
User code:
  record H2(Node node)implements Node{
    <T> T accept(Visitor<T> v){return v.visitH2(this);}//error
    }
  //but we can easily define
 class ContainsItalian extends PropagatorVisitor{..}
```

Week5 readings

- In the reading we have seen various solutions.
- Here we focus on Further extension / Virtual classes
- In the same way a method can be overridden, we assume an OO language where
  a nested class can be overridden too
- This is also called family polymorphism

```
Library A://Not JAVA, example language with virtual classes
class Dom{
  interface Node{op1, op2, op3}
  record P(String text)implements Node{op1, op2, op3}
  record H1(String text)implements Node{op1, op2, op3}
  record Html(Head head,Body body)implements Node{op1, op2, op3}
  record Head()implements Node{op1, op2, op3}
  record Body(List<Div>divs)implements Node{op1, op2, op3}
  ...
  }
------------
User code:
class RichDom extends Dom{
  @Override interface Node{op4}//new operation
  record H2(String text)implements Node{op1, op2, op3, op4}//new datavariant
  @Override record P{op4}//add the new operations to the existing data variants
  @Override record H1{op4}
  @Override record Html{op4}
  @Override record Head{op4}
  @Override record Body{op4}
  ..
  }
```

```
//Not JAVA, example language with virtual classes

class Graph{//Graph,Node and Arc are a family of classes
  List<Node> nodes=new ArrayList<>();
  public String toString(){
    String res="";
    for(Node n:nodes){res+="Node("+n.arcs+")\n";}
    return res;
    }

  class Node{ List<Arch> arcs=new ArrayList<>(); }//Nested class Node
  class Arc{ Node left; Node right; }//Nested class Arc
  }

class ColorGraph extends Graph{
  @Override public String toString(){//method overriding (virtual methods can be overridden)
    String res="";
    for(Node n:nodes){res+="Node("+n.color+", "+n.arcs+")\n";}
    return res;
    }
  @Override class Node{//nested class overriding (classes are now virtual too)
    //Node implicitly further extends Graph.Node
    String color;//added field
    }
  }
```

```
//Not JAVA, example language with virtual classes

class Graph{//Graph,Node and Arc are a family of classes
  List<Node> nodes=new ArrayList<>();
  public String toString(){
    String res="";
    for(Node n:nodes){res+="Node("+n.arcs+")\n";}
    return res;
    }

  class Node{ List<Arch> arcs=new ArrayList<>(); }//Nested class Node
  class Arc{ Node left; Node right; }//Nested class Arc
  }

class ColorGraph extends Graph{
  @Override public String toString(){//method overriding (virtual methods can be overridden)
    String res="";
    for(Node n:nodes){res+="Node("+n.color+", "+n.arcs+")\n";}
    return res;
    }
  @Override class Node{//nested class overriding (classes are now virtual too)
    //Node implicitly further extends Graph.Node
    String color;//added field
    }
  }
```

```java
class Graph{//same code of last slide, here for reference
  List<Node> nodes=new ArrayList<>();
  public String toString(){
    String res=""; for(Node n:nodes){res+="Node("+n.arcs+")\n";} return res;}
  class Node{ List<Arch> arcs=new ArrayList<>(); }
  class Arc{ Node left; Node right; }}
class ColorGraph extends Graph{
  @Override public String toString(){
    String res=""; for(Node n:nodes){res+="Node("+n.color+", "+n.arcs+")\n";} return res;}
  @Override class Node{String color;}}
//------------------------------------------------------------------

class User{static void main(String[]arg){
  Graph g=new Graph();//no use of subtyping; all good
  g.nodes.add(new Graph.Node());
  System.out.println(g.toString());

  ColorGraph cg=new ColorGraph();//no use of subtyping; all good
  cg.nodes.add(new ColorGraph.Node());
  System.out.println(cg.toString());

  Graph gg=new ColorGraph();//is this line ok?
  gg.nodes.add(new Graph.Node());//is this line ok?
  System.out.println(gg);//this fail: can not find node.color!
  }}
```

Solutions

- Scala/Scandinavian: Nested class types are dependent types:
  - Graph.Node and ColorGraph.Node: NOT TYPES
  - given a graph g, g.Node is a type.
  - g.nodes.add(new g.Node(..)) would work if g is a final variable.

Solutions

- L42/Separating use and reuse:
  code reuse does not induce subtyping
  - Graph and ColorGraph are not subtype of each other
  - interfaces are needed to have subtyping

Multimethods instead of Virtual classes

- Alternative solution: allow for methods to be dispatched on more then one argument.
- Usually a method belong to a class, so that dynamic dispatch can work on “this”
- Those methods can be seen as “not belonging” to any specific class, and all the arguments are equally dynamically dispatched

Multimethods in practique: python!

```
#pip install multimethod
from multimethod import multimethod

class Zebra:
  pass

class Lion:
  pass

class Elephant:
  pass

class Lake:
  pass

class Savana:
  pass

class Jungle:
  pass
```

```
@multimethod
def action(a:Lion, b:Zebra,c:Savana):
  print("the lion attacks the zebra")

@multimethod
def action(a:Lion, b:Lion,c):
  print("the two lions roar at each other")

@multimethod
def action(a:Lion, b:Lion,c:Lake):
  print("the two lions drink together peacefully")

@multimethod
def action(a:Lion, b:Elephant,c:Savana):
  print("the lion is scared of the elephant")

@multimethod
def action(a:Lion, b:Elephant,c:Jungle):
  print("the elephant mobility is reduced; lion attacks")
```

Multimethods in practique: python!

```
>>> action(Lion(),Lion(),3)
  the two lions roar at each
other
```

```
>>> action(Lion(),Lion(),Lake())

  the two lions drink together
peacefully
```

```
@multimethod
def action(a:Lion, b:Zebra,c:Savana):
  print("the lion attacks the zebra")

@multimethod
def action(a:Lion, b:Lion,c):
  print("the two lions roar at each other")

@multimethod
def action(a:Lion, b:Lion,c:Lake):
  print("the two lions drink together peacefully")

@multimethod
def action(a:Lion, b:Elephant,c:Savana):
  print("the lion is scared of the elephant")

@multimethod
def action(a:Lion, b:Elephant,c:Jungle):
  print("the elephant mobility is reduced; lion attacks")
```

```
#pip install multimethod
from multimethod import multimethod
from collections import namedtuple
P = namedtuple('P', 'text')
H1 = namedtuple('H1', 'text')
Html = namedtuple('Html', 'head
body')
Body = namedtuple('Body', 'divs')
Div = namedtuple('Div', 'ns')
A = namedtuple('A', 'href text')
Ul = namedtuple('Ul', 'lis')
Ol = namedtuple('Ul', 'lis')
Li = namedtuple('Ul', 'node')
Url = namedtuple('Url', 'href')

class Clone:
  pass

@multimethod def op(v:Clone, e:str):
  return e

@multimethod def op(v:Clone, e:P):
  return P(op(v,e.text))

@multimethod def op(v:Clone, e:H1):
  return H1(op(v,e.text))

@multimethod def op(v:Clone, e:Html):
  return Html(op(v,e.head),op(v,e.body))

@multimethod def op(v:Clone, e:Body):
  return Body([op(v,ei) for ei in e.divs])

@multimethod def op(v:Clone, e:Div):
  return Div([op(v,ei) for ei in e.ns])

@multimethod def op(v:Clone, e:A):
  return A(op(v,e.href),op(v,e.text))

@multimethod def op(v:Clone, e:Ul):
  return Ul([op(v,ei) for ei in e.lis])

@multimethod def op(v:Clone, e:Ol):
  return Ol([op(v,ei) for ei in e.lis])

@multimethod def op(v:Clone, e:Li):
  return Li(op(v,e.node))

@multimethod def op(v:Clone, e:Url):
  return e
```

ToItalian with multi-methods

```
class ToItalian(Clone):
  pass

@multimethod
def op(v:ToItalian, e:str):
  return "Pizza!!"+e

node=Div([
  A(Url('google.com'),"hi"),
  Div([P("world"),H1("Mario")]),
  ])

print(op(Clone(),node))
print(op(ToItalian(),node))

-----PrintOuts------
Div(ns=[
  A(href=Url(href='google.com'), text='hi'),
  Div(ns=[P(text='world'), H1(text='Mario')])])
Div(ns=[
  A(href=Url(href='google.com'), text='Pizza!!hi'),
  Div(ns=[P(text='Pizza!!world'), H1(text='Pizza!!Mario')])])
```

- Multi-methods can give all the advantages of visitors, with less boilerplate.
- Python multi-methods are not typesafe, that is, we could forget to define a case and we would only discover it at run time.

Prolog?

- If you have seen Prolog, you may notice the similarity of prolog clauses and multimethods.
- The main difference is that unification only work on types and not values.

  However, subtyping allows for some flexibility in the unification process.

Concluding

- Java is quite a minimalist language with very few features.
- There is a lot of research going on about more language features.
- Dynamic languages often allows to implement those features as libraries (as for @multimethod), but place more burden on the programmer to check that the behavior is well defined in all the cases
- Expressive Tools, like refactoring, usually need more automatic reasoning that what is available in Dynamic languages.
- Most languages are not just “Java/C” with a little syntactic twist, they need to be learned carefully.