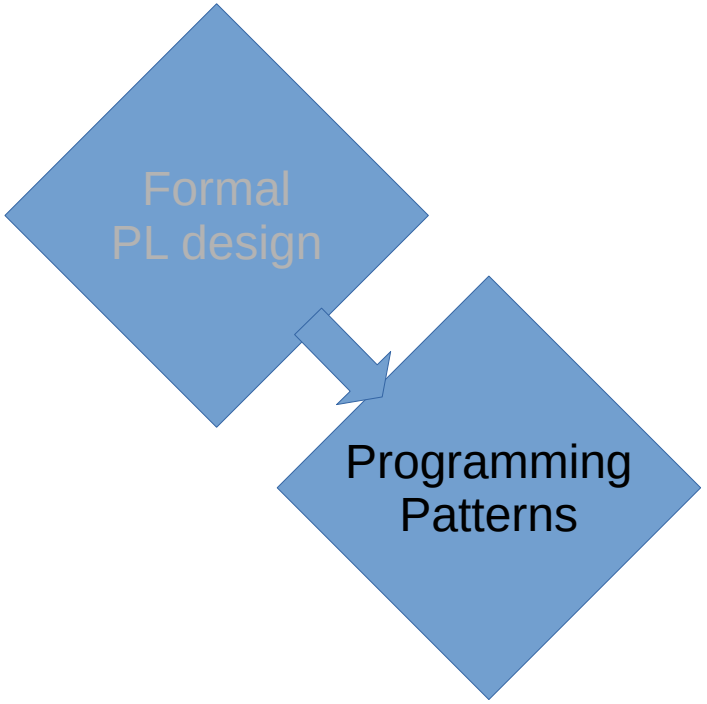
SWEN423 - Lecture 8



The visitor pattern

Java 14 adds records

```
record Point(int x, int y) {/*methods go here*/}


...
  public static void main(String[]arg){
    var m=new HashMap<Point,String>();
    m.put(new Point(0,0),"base");//record automatically adds equals and hash code
    m.put(new Point(1,0),"left");//so we can use them in HashMaps and HashSets
    m.put(new Point(0,1),"down");//fields of records are implicitly final
    System.out.println(m);//automatically adds toString
    //{Point[x=0, y=0]=base, Point[x=0, y=1]=down, Point[x=1, y=0]=left}
    System.out.println(m.containsKey(new Point(0,1)));
    //true
  }
```

Java 14 adds records

```
record Person(String name){
  public Person{//invariant
    assert name.length()>1; //this.name() would be == null
    }  }//all fields are implicitly final

record Positives(int[] ints){
  public Positives{//invariant? Not bullet proof!
    assert IntStream.of(ints).allMatch(i->i>=0);
    }//all fields are implicitly final
  }//but sub objects are still mutable :-(

public class Example {
  public static void main(String[]a){
    new Person("Bob");
    var p=new Positives(new int[]{1,2,3});
    p.ints()[1]=-100;//breaking invariant
    System.out.println(p.ints()[1]);//Records are only shallow immutable
    }  }
```

Java 14 adds records

```
record Person(String name){
  public Person{//invariant
    assert name.length()>1; //this.name() would be == null
    }  }//all fields are implicitly final

record Positives(int[] ints){
  public Positives{//invariant? Not bullet proof!
    assert IntStream.of(ints).allMatch(i->i>=0);
    }//all fields are implicitly final
  }//but sub objects are still mutable :-(

public class Example {
  public static void main(String[]a){
    new Person("Bob");
    var p=new Positives(new int[]{1,2,3});
    p.ints()[1]=-100;//breaking invariant
    System.out.println(p.ints()[1]);//Records are only shallow immutable
    }  }
```

Java 14 adds records

```
record Person(String name){
  public Person{//invariant
    assert name.length()>1; //this.name() would be == null
    }  }//all fields are implicitly final

record Positives(int[] ints){
  public Positives{//invariant? Not bullet proof!
    assert IntStream.of(ints).allMatch(i->i>=0);
    }//all fields are implicitly final
  }//but sub objects are still mutable :-(

public class Example {
  public static void main(String[]a){
    new Person("Bob");
    var p=new Positives(new int[]{1,2,3});
    p.ints()[1]=-100;//breaking invariant
    System.out.println(p.ints()[1]);//Records are only shallow immutable
    }  }
```

Composite pattern - HTML

```
interface Node{
  record P(String text)implements Node{}
  record H1(String text)implements Node{}
  record Html(Head head,Body body)implements Node{}
  record Head()implements Node{}
  record Body(List<Div>divs)implements Node{}
  record Div(List<Node>ns)implements Node{}
  record A(String href,String text)implements Node{}
  record Ul(List<Li>lis)implements Node{}
  record Ol(List<Li>lis)implements Node{}
  record Li(Node node)implements Node{}
  }
```

Terminology:
Node: A **component**
P, H1, Head,A: **leaves**
Html,Body,Div,Ul,Ol,Li: **composites**; they contains other components

- leave = composite with zero components
- The composite pattern requires a tree structure (no cycles!)

Adding operations

```
interface Node{
  String toHtml();//example operation
  record P(String text)implements Node{public String toHtml(){return "<p>"+text+"</p>";}}
  record H1(String text)implements Node{public String toHtml(){return format("h1","",text);}}
  record Html(Head head,Body body)implements Node{
    public String toHtml(){return format("html","",head.toHtml()+body.toHtml());}}
  record Head()implements Node{public String toHtml(){return "<head></head>";}}
  record Body(List<Div>divs)implements Node{
    public String toHtml(){return format("body","",all(divs));}}
  record Div(List<Node>ns)implements Node{
    public String toHtml(){return format("div","",all(ns));}}
  record A(String href,String text)implements Node{
    public String toHtml(){return format("a"," href=\""+href+"\"",text);}}
  record Ul(List<Li>lis)implements Node{
    public String toHtml(){return format("ul","",all(lis));}}
  record Ol(List<Li>lis)implements Node{
    public String toHtml(){return format("ol","",all(lis));}}
  record Li(Node node)implements Node{
    public String toHtml(){return format("li","",node.toHtml());}}

  static String format(String tag,String attributes,String content){//auxiliary method
    return "<"+tag+attributes+">"+content+"</"+tag+">";
    }
  static String all(List<? extends Node> ns){//auxiliary method
    return ns.stream().map(e->e.toHtml()).collect(Collectors.joining());
    }  }
```

Adding operations

```
interface Node{
  String toHtml();//example operation
  record P(String text)implements Node{public String toHtml(){return "<p>"+text+"</p>";}}
  record H1(String text)implements Node{public String toHtml(){return format("h1","",text);}}
  record Html(Head head,Body body)implements Node{
    public String toHtml(){return format("html","",head.toHtml()+body.toHtml());}}
  record Head()implements Node{public String toHtml(){return "<head></head>";}}
  record Body(List<Div>divs)implements Node{
    public String toHtml(){return format("body","",all(divs));}}
  record Div(List<Node>ns)implements Node{
    public String toHtml(){return format("div","",all(ns));}}
  record A(String href,String text)implements Node{
    public String toHtml(){return format("a"," href=\""+href+"\"",text);}}
  record Ul(List<Li>lis)implements Node{
    public String toHtml(){return format("ul","",all(lis));}}
  record Ol(List<Li>lis)implements Node{
    public String toHtml(){return format("ol","",all(lis));}}
  record Li(Node node)implements Node{
    public String toHtml(){return format("li","",node.toHtml());}}

  static String format(String tag,String attributes,String content){//auxiliary method
    return "<"+tag+attributes+">"+content+"</"+tag+">";
    }
  static String all(List<? extends Node> ns){//auxiliary method
    return ns.stream().map(e->e.toHtml()).collect(Collectors.joining());
    }  }
```

Adding operations

```
interface Node{

  String toHtml();//example operation

  record P(String text)implements Node{
    public String toHtml(){
      return "<p>"+text+"</p>";
      }
    }
  ..
  ..
  ..
  ..
  }
```

Adding operations

```
interface Node{
  String toHtml();//example operation
  record P(String text)implements Node{public String toHtml(){return "<p>"+text+"</p>";}}
  record H1(String text)implements Node{public String toHtml(){return format("h1","",text);}}
  record Html(Head head,Body body)implements Node{
    public String toHtml(){return format("html","",head.toHtml()+body.toHtml());}}
  record Head()implements Node{public String toHtml(){return "<head></head>";}}
  record Body(List<Div>divs)implements Node{
    public String toHtml(){return format("body","",all(divs));}}
  record Div(List<Node>ns)implements Node{
    public String toHtml(){return format("div","",all(ns));}}
  record A(String href,String text)implements Node{
    public String toHtml(){return format("a"," href=\""+href+"\"",text);}}
  record Ul(List<Li>lis)implements Node{
    public String toHtml(){return format("ul","",all(lis));}}
  record Ol(List<Li>lis)implements Node{
    public String toHtml(){return format("ol","",all(lis));}}
  record Li(Node node)implements Node{
    public String toHtml(){return format("li","",node.toHtml());}}

  static String format(String tag,String attributes,String content){//auxiliary method
    return "<"+tag+attributes+">"+content+"</"+tag+">";
    }
  static String all(List<? extends Node> ns){//auxiliary method
    return ns.stream().map(e->e.toHtml()).collect(Collectors.joining());
    }  }
```

Adding operations

```
interface Node{
  String toHtml();//example operation
  record P(String text)implements Node{public String toHtml(){return "<p>"+text+"</p>";}}
  record H1(String text)implements Node{public String toHtml(){return format("h1","",text);}}
  record Html(Head head,Body body)implements Node{
    public String toHtml(){return format("html","",head.toHtml()+body.toHtml());}}
  record Head()implements Node{public String toHtml(){return "<head></head>";}}
  record Body(List<Div>divs)implements Node{
    public String toHtml(){return format("body","",all(divs));}}
  record Div(List<Node>ns)implements Node{
    public String toHtml(){return format("div","",all(ns));}}
  record A(String href,String text)implements Node{
    public String toHtml(){return format("a"," href=\""+href+"\"",text);}}
  record Ul(List<Li>lis)implements Node{
    public String toHtml(){return format("ul","",all(lis));}}
  record Ol(List<Li>lis)implements Node{
    public String toHtml(){return format("ol","",all(lis));}}
  record Li(Node node)implements Node{
    public String toHtml(){return format("li","",node.toHtml());}}

  static String format(String tag,String attributes,String content){//auxiliary method
    return "<"+tag+attributes+">"+content+"</"+tag+">";
    }
  static String all(List<? extends Node> ns){//auxiliary method
    return ns.stream().map(e->e.toHtml()).collect(Collectors.joining());
    }  }
```

Adding operations

```
interface Node{
  String toHtml();//example operation
  record P(String text)implements Node{public String toHtml(){return "<p>"+text+"</p>";}}
  record H1(String text)implements Node{public String toHtml(){return format("h1","",text);}}
  record Html(Head head,Body body)implements Node{
    public String toHtml(){return format("html","",head.toHtml()+body.toHtml());}}
  record Head()implements Node{public String toHtml(){return "<head></head>";}}
  record Body(List<Div>divs)implements Node{
    public String toHtml(){return format("body","",all(divs));}}
  record Div(List<Node>ns)implements Node{
    public String toHtml(){return format("div","",all(ns));}}
  record A(String href,String text)implements Node{
    public String toHtml(){return format("a"," href=\""+href+"\"",text);}}
  record Ul(List<Li>lis)implements Node{
    public String toHtml(){return format("ul","",all(lis));}}
  record Ol(List<Li>lis)implements Node{
    public String toHtml(){return format("ol","",all(lis));}}
  record Li(Node node)implements Node{
    public String toHtml(){return format("li","",node.toHtml());}}

  static String format(String tag,String attributes,String content){//auxiliary method
    return "<"+tag+attributes+">"+content+"</"+tag+">";
    }
  static String all(List<? extends Node> ns){//auxiliary method
    return ns.stream().map(e->e.toHtml()).collect(Collectors.joining());
    }  }
```

Adding operations

```
interface Node{
  String toHtml();//example operation
  record P(String text)implements Node{public String toHtml(){return "<p>"+text+"</p>";}}
  record H1(String text)implements Node{public String toHtml(){return format("h1","",text);}}
  record Html(Head head,Body body)implements Node{
    public String toHtml(){return format("html","",head.toHtml()+body.toHtml());}}
  record Head()implements Node{public String toHtml(){return "<head></head>";}}
  record Body(List<Div>divs)implements Node{
    public String toHtml(){return format("body","",all(divs));}}
  record Div(List<Node>ns)implements Node{
    public String toHtml(){return format("div","",all(ns));}}
  record A(String href,String text)implements Node{
    public String toHtml(){return format("a"," href=\""+href+"\"",text);}}
  record Ul(List<Li>lis)implements Node{
    public String toHtml(){return format("ul","",all(lis));}}
  record Ol(List<Li>lis)implements Node{
    public String toHtml(){return format("ol","",all(lis));}}
  record Li(Node node)implements Node{
    public String toHtml(){return format("li","",node.toHtml());}}

  static String format(String tag,String attributes,String content){//auxiliary method
    return "<"+tag+attributes+">"+content+"</"+tag+">";
    }
  static String all(List<? extends Node> ns){//auxiliary method
    return ns.stream().map(e->e.toHtml()).collect(Collectors.joining());
    }  }
```

Adding operations

```
interface Node{
  String toHtml();//example operation
  record P(String text)implements Node{public String toHtml(){return "<p>"+text+"</p>";}}
  record H1(String text)implements Node{public String toHtml(){return format("h1","",text);}}
  record Html(Head head,Body body)implements Node{
    public String toHtml(){return format("html","",head.toHtml()+body.toHtml());}}
  record Head()implements Node{public String toHtml(){return "<head></head>";}}
  record Body(List<Div>divs)implements Node{
    public String toHtml(){return format("body","",all(divs));}}
  record Div(List<Node>ns)implements Node{
    public String toHtml(){return format("div","",all(ns));}}
  record A(String href,String text)implements Node{
    public String toHtml(){return format("a"," href=\""+href+"\"",text);}}
  record Ul(List<Li>lis)implements Node{
    public String toHtml(){return format("ul","",all(lis));}}
  record Ol(List<Li>lis)implements Node{
    public String toHtml(){return format("ol","",all(lis));}}
  record Li(Node node)implements Node{
    public String toHtml(){return format("li","",node.toHtml());}}

  static String format(String tag,String attributes,String content){//auxiliary method
    return "<"+tag+attributes+">"+content+"</"+tag+">";
    }
  static String all(List<? extends Node> ns){//auxiliary method
    return ns.stream().map(e->e.toHtml()).collect(Collectors.joining());
    }  }
```

Adding operations

```
interface Node{
  String toHtml();//example operation
  record P(String text)implements Node{public String toHtml(){return "<p>"+text+"</p>";}}
  record H1(String text)implements Node{public String toHtml(){return format("h1","",text);}}
  record Html(Head head,Body body)implements Node{
    public String toHtml(){return format("html","",head.toHtml()+body.toHtml());}}
  record Head()implements Node{public String toHtml(){return "<head></head>";}}
  record Body(List<Div>divs)implements Node{
    public String toHtml(){return format("body","",all(divs));}}
  record Div(List<Node>ns)implements Node{
    public String toHtml(){return format("div","",all(ns));}}
  record A(String href,String text)implements Node{
    public String toHtml(){return format("a"," href=\""+href+"\"",text);}}
  record Ul(List<Li>lis)implements Node{
    public String toHtml(){return format("ul","",all(lis));}}
  record Ol(List<Li>lis)implements Node{
    public String toHtml(){return format("ol","",all(lis));}}
  record Li(Node node)implements Node{
    public String toHtml(){return format("li","",node.toHtml());}}

  static String format(String tag,String attributes,String content){//auxiliary method
    return "<"+tag+attributes+">"+content+"</"+tag+">";
    }
  static String all(List<? extends Node> ns){//auxiliary method
    return ns.stream().map(e->e.toHtml()).collect(Collectors.joining());
    }  }
```

Adding operations

```
interface Node{
  String toHtml();//example operation
  record P(String text)implements Node{public String toHtml(){return "<p>"+text+"</p>";}}
  record H1(String text)implements Node{public String toHtml(){return format("h1","",text);}}
  record Html(Head head,Body body)implements Node{
    public String toHtml(){return format("html","",head.toHtml()+body.toHtml());}}
  record Head()implements Node{public String toHtml(){return "<head></head>";}}
  record Body(List<Div>divs)implements Node{
    public String toHtml(){return format("body","",all(divs));}}
  record Div(List<Node>ns)implements Node{
    public String toHtml(){return format("div","",all(ns));}}
  record A(String href,String text)implements Node{
    public String toHtml(){return format("a"," href=\""+href+"\"",text);}}
  record Ul(List<Li>lis)implements Node{
    public String toHtml(){return format("ul","",all(lis));}}
  record Ol(List<Li>lis)implements Node{
    public String toHtml(){return format("ol","",all(lis));}}
  record Li(Node node)implements Node{
    public String toHtml(){return format("li","",node.toHtml());}}

  static String format(String tag,String attributes,String content){//auxiliary method
    return "<"+tag+attributes+">"+content+"</"+tag+">";
    }
  static String all(List<? extends Node> ns){//auxiliary method
    return ns.stream().map(e->e.toHtml()).collect(Collectors.joining());
    }  }
```

Adding operations

```
interface Node{
  String toHtml();//example operation
  record P(String text)implements Node{public String toHtml(){return "<p>"+text+"</p>";}}
  record H1(String text)implements Node{public String toHtml(){return format("h1","",text);}}
  record Html(Head head,Body body)implements Node{
    public String toHtml(){return format("html","",head.toHtml()+body.toHtml());}}
  record Head()implements Node{public String toHtml(){return "<head></head>";}}
  record Body(List<Div>divs)implements Node{
    public String toHtml(){return format("body","",all(divs));}}
  record Div(List<Node>ns)implements Node{
    public String toHtml(){return format("div","",all(ns));}}
  record A(String href,String text)implements Node{
    public String toHtml(){return format("a"," href=\""+href+"\"",text);}}
  record Ul(List<Li>lis)implements Node{
    public String toHtml(){return format("ul","",all(lis));}}
  record Ol(List<Li>lis)implements Node{
    public String toHtml(){return format("ol","",all(lis));}}
  record Li(Node node)implements Node{
    public String toHtml(){return format("li","",node.toHtml());}}

  static String format(String tag,String attributes,String content){//auxiliary method
    return "<"+tag+attributes+">"+content+"</"+tag+">";
    }
  static String all(List<? extends Node> ns){//auxiliary method
    return ns.stream().map(e->e.toHtml()).collect(Collectors.joining());
    }  }
```

Adding operations

```
interface Node{
  String toHtml();//example operation
  record P(String text)implements Node{public String toHtml(){return "<p>"+text+"</p>";}}
  record H1(String text)implements Node{public String toHtml(){return format("h1","",text);}}
  record Html(Head head,Body body)implements Node{
    public String toHtml(){return format("html","",head.toHtml()+body.toHtml());}}
  record Head()implements Node{public String toHtml(){return "<head></head>";}}
  record Body(List<Div>divs)implements Node{
    public String toHtml(){return format("body","",all(divs));}}
  record Div(List<Node>ns)implements Node{
    public String toHtml(){return format("div","",all(ns));}}
  record A(String href,String text)implements Node{
    public String toHtml(){return format("a"," href=\""+href+"\"",text);}}
  record Ul(List<Li>lis)implements Node{
    public String toHtml(){return format("ul","",all(lis));}}
  record Ol(List<Li>lis)implements Node{
    public String toHtml(){return format("ol","",all(lis));}}
  record Li(Node node)implements Node{
    public String toHtml(){return format("li","",node.toHtml());}}

  static String format(String tag,String attributes,String content){//auxiliary method
    return "<"+tag+attributes+">"+content+"</"+tag+">";
    }
  static String all(List<? extends Node> ns){//auxiliary method
    return ns.stream().map(e->e.toHtml()).collect(Collectors.joining());
    }  }
```

Adding operations

```
interface Node{
  String toHtml();//example operation
  record P(String text)implements Node{public String toHtml(){return "<p>"+text+"</p>";}}
  record H1(String text)implements Node{public String toHtml(){return format("h1","",text);}}
  record Html(Head head,Body body)implements Node{
    public String toHtml(){return format("html","",head.toHtml()+body.toHtml());}}
  record Head()implements Node{public String toHtml(){return "<head></head>";}}
  record Body(List<Div>divs)implements Node{
    public String toHtml(){return format("body","",all(divs));}}
  record Div(List<Node>ns)implements Node{
    public String toHtml(){return format("div","",all(ns));}}
  record A(String href,String text)implements Node{
    public String toHtml(){return format("a"," href=\""+href+"\"",text);}}
  record Ul(List<Li>lis)implements Node{
    public String toHtml(){return format("ul","",all(lis));}}
  record Ol(List<Li>lis)implements Node{
    public String toHtml(){return format("ol","",all(lis));}}
  record Li(Node node)implements Node{
    public String toHtml(){return format("li","",node.toHtml());}}

  static String format(String tag,String attributes,String content){//auxiliary method
    return "<"+tag+attributes+">"+content+"</"+tag+">";
    }
  static String all(List<? extends Node> ns){//auxiliary method
    return ns.stream().map(e->e.toHtml()).collect(Collectors.joining());
    }  }
```

Adding operations

```
interface Node{
  String toHtml();//example operation

  record P(String text)implements Node{
    public String toHtml(){
      return "<p>"+text+"</p>";
      }
    }
  ..
  }
```

2 - 5 LINES FOR EACH CASE?? whoa, that is quite a lot!
And we only have one operation!
If we add more operations, we would end up with at least as many extra lines as operations; so if we have like 5 operations it would be at least 6 lines for each case...

Unacceptable!

Moreover, the implementation of an operation is logically tightly connected, But by adding operations as methods, we end up with all the operations dismembered between the cases.
Introducing the Visitor pattern!

- Visitor -

One operation to rule them all

```
interface Node{
  <T> T accept(Visitor<T> v);
  record P(String text)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitP(this);}}
  record H1(String text)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitH1(this);}}
  record Html(Head head,Body body)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitHtml(this);}}
  record Head()implements Node{
    public <T> T accept(Visitor<T> v){return v.visitHead(this);}}
  record Body(List<Div>divs)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitBody(this);}}
  record Div(List<Node>ns)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitDiv(this);}}
  record A(String href,String text)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitA(this);}}
  record Ul(List<Li>lis)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitUl(this);}}
  record Ol(List<Li>lis)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitOl(this);}}
  record Li(Node node)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitLi(this);}}
  }
```

```
import static foo2.Node.*;
interface Visitor<T>{
  T visitP(P e);
  T visitH1(H1 e);
  T visitHtml(Html e);
  T visitHead(Head e);
  T visitBody(Body e);
  T visitDiv(Div e);
  T visitA(A e);
  T visitUl(Ul e);
  T visitOl(Ol e);
  T visitLi(Li e);
  }
```

- Visitor -

One operation to rule them all

```
interface Node{
  <T> T accept(Visitor<T> v);
  record P(String text)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitP(this);}}
  record H1(String text)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitH1(this);}}
  record Html(Head head,Body body)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitHtml(this);}}
  record Head()implements Node{
    public <T> T accept(Visitor<T> v){return v.visitHead(this);}}
  record Body(List<Div>divs)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitBody(this);}}
  record Div(List<Node>ns)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitDiv(this);}}
  record A(String href,String text)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitA(this);}}
  record Ul(List<Li>lis)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitUl(this);}}
  record Ol(List<Li>lis)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitOl(this);}}
  record Li(Node node)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitLi(this);}}
  }
```

```
import static foo2.Node.*;
interface Visitor<T>{
  T visitP(P e);
  T visitH1(H1 e);
  T visitHtml(Html e);
  T visitHead(Head e);
  T visitBody(Body e);
  T visitDiv(Div e);
  T visitA(A e);
  T visitUl(Ul e);
  T visitOl(Ol e);
  T visitLi(Li e);
  }
```

- Visitor -

One operation to rule them all

```
interface Node{
  <T> T accept(Visitor<T> v);
  record P(String text)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitP(this);}}
  record H1(String text)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitH1(this);}}
  record Html(Head head,Body body)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitHtml(this);}}
  record Head()implements Node{
    public <T> T accept(Visitor<T> v){return v.visitHead(this);}}
  record Body(List<Div>divs)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitBody(this);}}
  record Div(List<Node>ns)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitDiv(this);}}
  record A(String href,String text)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitA(this);}}
  record Ul(List<Li>lis)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitUl(this);}}
  record Ol(List<Li>lis)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitOl(this);}}
  record Li(Node node)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitLi(this);}}
  }
```

```
import static foo2.Node.*;
interface Visitor<T>{
  T visitP(P e);
  T visitH1(H1 e);
  T visitHtml(Html e);
  T visitHead(Head e);
  T visitBody(Body e);
  T visitDiv(Div e);
  T visitA(A e);
  T visitUl(Ul e);
  T visitOl(Ol e);
  T visitLi(Li e);
  }
```

- Visitor -

One operation to rule them all

```
interface Node{
  <T> T accept(Visitor<T> v);

  P(String text)  visitP

  H1(String text) visitH1

  Html(Head head,Body body) visitHtml

  Head()  visitHead

  Body(List<Div>divs)  visitBody

  Div(List<Node>ns)  visitDiv

  A(String href,String text) visitA

  Ul(List<Li>lis)  visitUl

  Ol(List<Li>lis)  visitOl

  Li(Node node)  visitLi
  }
```

```
import static foo2.Node.*;
interface Visitor<T>{
  T visitP(P e);
  T visitH1(H1 e);
  T visitHtml(Html e);
  T visitHead(Head e);
  T visitBody(Body e);
  T visitDiv(Div e);
  T visitA(A e);
  T visitUl(Ul e);
  T visitOl(Ol e);
  T visitLi(Li e);
  }
```

- Visitor -

One operation to rule them all

```
interface Node{
  <T> T accept(Visitor<T> v);
  record P(String text)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitP(this);}}
  record H1(String text)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitH1(this);}}
  record Html(Head head,Body body)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitHtml(this);}}
  record Head()implements Node{
    public <T> T accept(Visitor<T> v){return v.visitHead(this);}}
  record Body(List<Div>divs)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitBody(this);}}
  record Div(List<Node>ns)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitDiv(this);}}
  record A(String href,String text)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitA(this);}}
  record Ul(List<Li>lis)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitUl(this);}}
  record Ol(List<Li>lis)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitOl(this);}}
  record Li(Node node)implements Node{
    public <T> T accept(Visitor<T> v){return v.visitLi(this);}}
  }
```

```
import static foo2.Node.*;
interface Visitor<T>{
  T visitP(P e);
  T visitH1(H1 e);
  T visitHtml(Html e);
  T visitHead(Head e);
  T visitBody(Body e);
  T visitDiv(Div e);
  T visitA(A e);
  T visitUl(Ul e);
  T visitOl(Ol e);
  T visitLi(Li e);
  }
```

```
class ToHtml implements Visitor<String>{  //usage: myNode.accept(new ToHtml())
  String format(String tag,String attributes,String content){
    return "<"+tag+attributes+">"+content+"</"+tag+">";}
  String all(List<? extends Node> ns){
    return ns.stream().map(e->e.accept(this)).collect(Collectors.joining());}
  public String visitP(P e){
    return format("p","",e.text());}
  public String visitH1(H1 e){
    return format("h1","",e.text());}
  public String visitHtml(Html e){
    return format("html","",visitHead(e.head())+visitBody(e.body()));}
  public String visitHead(Head e){
    return format("head","","");}
  public String visitBody(Body e){
    return format("body","",all(e.divs()));}
  public String visitDiv(Div e){
    return format("div","",all(e.ns()));}
  public String visitA(A e){
    return format("a",e.href(),e.text());}
  public String visitUl(Ul e){
    return format("ul","",all(e.lis()));}
  public String visitOl(Ol e){
    return format("ol","",all(e.lis()));}
  public String visitLi(Li e){
    return format("li","",e.node().accept(this));}
  }
```

```
class ToHtml implements Visitor<String>{  //usage: myNode.accept(new ToHtml())
  String format(String tag,String attributes,String content){
    return "<"+tag+attributes+">"+content+"</"+tag+">";}
  String all(List<? extends Node> ns){
    return ns.stream().map(e->e.accept(this)).collect(Collectors.joining());}
  public String visitP(P e){
    return format("p","",e.text());}
  public String visitH1(H1 e){
    return format("h1","",e.text());}
  public String visitHtml(Html e){
    return format("html","",visitHead(e.head())+visitBody(e.body()));}
  public String visitHead(Head e){
    return format("head","","");}
  public String visitBody(Body e){
    return format("body","",all(e.divs()));}
  public String visitDiv(Div e){
    return format("div","",all(e.ns()));}
  public String visitA(A e){
    return format("a",e.href(),e.text());}
  public String visitUl(Ul e){
    return format("ul","",all(e.lis()));}
  public String visitOl(Ol e){
    return format("ol","",all(e.lis()));}
  public String visitLi(Li e){
    return format("li","",e.node().accept(this));}
  }
```

```
class ToHtml implements Visitor<String>{  //usage: myNode.accept(new ToHtml())
  String format(String tag,String attributes,String content){
    return "<"+tag+attributes+">"+content+"</"+tag+">";}
  String all(List<? extends Node> ns){
    return ns.stream().map(e->e.accept(this)).collect(Collectors.joining());}

  // format(_) of the following:

  (P e)= "p","",e.text()

  (H1 e)= "h1","",e.text()

  (Html e)="html","",visitHead(e.head())+visitBody(e.body())

  (Head e)="head","",""

  (Body e)="body","",all(e.divs())

  (Div e)= "div","",all(e.ns())

  (A e)= "a",e.href(),e.text()

  (Ul e)= "ul","",all(e.lis())

  (Ol e)= "ol","",all(e.lis())

  (Li e)= "li","",e.node().accept(this)

  }
```

```
class ToHtml implements Visitor<String>{  //usage: myNode.accept(new ToHtml())
  String format(String tag,String attributes,String content){
    return "<"+tag+attributes+">"+content+"</"+tag+">";}
  String all(List<? extends Node> ns){
    return ns.stream().map(e->e.accept(this)).collect(Collectors.joining());}
  public String visitP(P e){
    return format("p","",e.text());}
  public String visitH1(H1 e){
    return format("h1","",e.text());}
  public String visitHtml(Html e){
    return format("html","",visitHead(e.head())+visitBody(e.body()));}
  public String visitHead(Head e){
    return format("head","","");}
  public String visitBody(Body e){
    return format("body","",all(e.divs()));}
  public String visitDiv(Div e){
    return format("div","",all(e.ns()));}
  public String visitA(A e){
    return format("a",e.href(),e.text());}
  public String visitUl(Ul e){
    return format("ul","",all(e.lis()));}
  public String visitOl(Ol e){
    return format("ol","",all(e.lis()));}
  public String visitLi(Li e){
    return format("li","",e.node().accept(this));}
  }
```

```
class CloneVisitor implements Visitor<Node>{
  public static <T> List<T>map(List<T>ts,Function<T,T>m){
    return ts.stream().map(m).collect(Collectors.toList());}
  public String visitText(String text){return text;}
  public String visitUrl(String url){return url;}
  public Node visitH1(H1 e){
    return new H1(visitText(e.text()));}
  public Node visitP(P e){
    return new P(visitText(e.text()));}
  public Node visitHtml(Html e){
    return new Html(visitHead(e.head()),visitBody(e.body()));}
  public Head visitHead(Head e){
    return new Head();}
  public Body visitBody(Body e){
    return new Body(map(e.divs(),ei->visitDiv(ei)));}
  public Div visitDiv(Div e){
    return new Div(map(e.ns(),ei->ei.accept(this)));}
  public A visitA(A e){
    return new A(visitUrl(e.href()),visitText(e.text()));}
  public Node visitUl(Ul e){
    return new Ul(map(e.lis(),ei->visitLi(ei)));}
  public Node visitOl(Ol e){
    return new Ol(map(e.lis(),ei->visitLi(ei)));}
  public Li visitLi(Li e){
    return new Li(e.node().accept(this));}
  }
```

Possibly the most useful visitor: the CloneVisitor!

Note how it is doing absolutely nothing!

How can it be the most useful one if it does nothing?

```
class CloneVisitor implements Visitor<Node>{
  public static <T> List<T>map(List<T>ts,Function<T,T>m){
    return ts.stream().map(m).collect(Collectors.toList());}
  public String visitText(String text){return text;}
  public String visitUrl(String url){return url;}
  public Node visitH1(H1 e){
    return new H1(visitText(e.text()));}
  public Node visitP(P e){
    return new P(visitText(e.text()));}
  public Node visitHtml(Html e){
    return new Html(visitHead(e.head()),visitBody(e.body()));}
  public Head visitHead(Head e){
    return new Head();}
  public Body visitBody(Body e){
    return new Body(map(e.divs(),ei->visitDiv(ei)));}
  public Div visitDiv(Div e){
    return new Div(map(e.ns(),ei->ei.accept(this)));}
  public A visitA(A e){
    return new A(visitUrl(e.href()),visitText(e.text()));}
  public Node visitUl(Ul e){
    return new Ul(map(e.lis(),ei->visitLi(ei)));}
  public Node visitOl(Ol e){
    return new Ol(map(e.lis(),ei->visitLi(ei)));}
  public Li visitLi(Li e){
    return new Li(e.node().accept(this));}
  }
```

Possibly the most useful visitor: the CloneVisitor!

Note how it is doing absolutely nothing!

How can it be the most useful one if it does nothing?

```
class CloneVisitor implements Visitor<Node>{
  public static <T> List<T>map(List<T>ts,Function<T,T>m){
    return ts.stream().map(m).collect(Collectors.toList());}
  public String visitText(String text){return text;}
  public String visitUrl(String url){return url;}
  public Node visitH1(H1 e){
    return new H1(visitText(e.text()));}
  public Node visitP(P e){
    return new P(visitText(e.text()));}
  public Node visitHtml(Html e){
    return new Html(visitHead(e.head()),visitBody(e.body()));}
  public Head visitHead(Head e){
    return new Head();}
  public Body visitBody(Body e){
    return new Body(map(e.divs(),ei->visitDiv(ei)));}
  public Div visitDiv(Div e){
    return new Div(map(e.ns(),ei->ei.accept(this)));}
  public A visitA(A e){
    return new A(visitUrl(e.href()),visitText(e.text()));}
  public Node visitUl(Ul e){
    return new Ul(map(e.lis(),ei->visitLi(ei)));}
  public Node visitOl(Ol e){
    return new Ol(map(e.lis(),ei->visitLi(ei)));}
  public Li visitLi(Li e){
    return new Li(e.node().accept(this));}
  }
```

Possibly the most useful visitor: the CloneVisitor!

Note how it is doing absolutely nothing!

How can it be the most useful one if it does nothing?

```
class CloneVisitor implements Visitor<Node>{
  public static <T> List<T>map(List<T>ts,Function<T,T>m){
    return ts.stream().map(m).collect(Collectors.toList());}
  public String visitText(String text){return text;}
  public String visitUrl(String url){return url;}
  public Node visitH1(H1 e){
    return new H1(visitText(e.text()));}
  public Node visitP(P e){
    return new P(visitText(e.text()));}
  public Node visitHtml(Html e){
    return new Html(visitHead(e.head()),visitBody(e.body()));}
  public Head visitHead(Head e){
    return new Head();}
  public Body visitBody(Body e){
    return new Body(map(e.divs(),ei->visitDiv(ei)));}
  public Div visitDiv(Div e){
    return new Div(map(e.ns(),ei->ei.accept(this)));}
  public A visitA(A e){
    return new A(visitUrl(e.href()),visitText(e.text()));}
  public Node visitUl(Ul e){
    return new Ul(map(e.lis(),ei->visitLi(ei)));}
  public Node visitOl(Ol e){
    return new Ol(map(e.lis(),ei->visitLi(ei)));}
  public Li visitLi(Li e){
    return new Li(e.node().accept(this));}
  }
```

Possibly the most useful visitor: the CloneVisitor!

Note how it is doing absolutely nothing!

How can it be the most useful one if it does nothing?

```
class CloneVisitor implements Visitor<Node>{
  public static <T> List<T>map(List<T>ts,Function<T,T>m){
    return ts.stream().map(m).collect(Collectors.toList());}
  public String visitText(String text){return text;}
  public String visitUrl(String url){return url;}
  public Node visitH1(H1 e){
    return new H1(visitText(e.text()));}
  public Node visitP(P e){
    return new P(visitText(e.text()));}
  public Node visitHtml(Html e){
    return new Html(visitHead(e.head()),visitBody(e.body()));}
  public Head visitHead(Head e){
    return new Head();}
  public Body visitBody(Body e){
    return new Body(map(e.divs(),ei->visitDiv(ei)));}
  public Div visitDiv(Div e){
    return new Div(map(e.ns(),ei->ei.accept(this)));}
  public A visitA(A e){
    return new A(visitUrl(e.href()),visitText(e.text()));}
  public Node visitUl(Ul e){
    return new Ul(map(e.lis(),ei->visitLi(ei)));}
  public Node visitOl(Ol e){
    return new Ol(map(e.lis(),ei->visitLi(ei)));}
  public Li visitLi(Li e){
    return new Li(e.node().accept(this));}
  }
```

Possibly the most useful visitor: the CloneVisitor!

Note how it is doing absolutely nothing!

How can it be the most useful one if it does nothing?

```
class CloneVisitor implements Visitor<Node>{
  public static <T> List<T>map(List<T>ts,Function<T,T>m){
    return ts.stream().map(m).collect(Collectors.toList());}
  public String visitText(String text){return text;}
  public String visitUrl(String url){return url;}
  public Node visitH1(H1 e){
    return new H1(visitText(e.text()));}
  public Node visitP(P e){
    return new P(visitText(e.text()));}
  public Node visitHtml(Html e){
    return new Html(visitHead(e.head()),visitBody(e.body()));}
  public Head visitHead(Head e){
    return new Head();}
  public Body visitBody(Body e){
    return new Body(map(e.divs(),ei->visitDiv(ei)));}
  public Div visitDiv(Div e){
    return new Div(map(e.ns(),ei->ei.accept(this)));}
  public A visitA(A e){
    return new A(visitUrl(e.href()),visitText(e.text()));}
  public Node visitUl(Ul e){
    return new Ul(map(e.lis(),ei->visitLi(ei)));}
  public Node visitOl(Ol e){
    return new Ol(map(e.lis(),ei->visitLi(ei)));}
  public Li visitLi(Li e){
    return new Li(e.node().accept(this));}
  }
```

Possibly the most useful visitor: the CloneVisitor!

Note how it is doing absolutely nothing!

How can it be the most useful one if it does nothing?

```
class CloneVisitor implements Visitor<Node>{
  public static <T> List<T>map(List<T>ts,Function<T,T>m){
    return ts.stream().map(m).collect(Collectors.toList());}

  // ~ == propagate the operation
       (String text)=text
       (String url)=url
  Node (H1 e)=~e.text()
  Node (P e)= ~e.text()
  Node (Html e)= ~e.head(),~e.body()
       (Head e)= e
       (Body e)= map(e.divs(),~)
       (Div e)= map(e.ns(),~)
       (A e)= ~e.href(),~e.text()
  Node (Ul e)=map(e.lis(),~)
  Node (Ol e)=map(e.lis(),~)
       (Li e)= ~e.node()
  }
```

Possibly the most useful visitor: the CloneVisitor!

Note how it is doing absolutely nothing!

How can it be the most useful one if it does nothing?

```
class CloneVisitor implements Visitor<Node>{
  public static <T> List<T>map(List<T>ts,Function<T,T>m){
    return ts.stream().map(m).collect(Collectors.toList());}
  public String visitText(String text){return text;}
  public String visitUrl(String url){return url;}
  public Node visitH1(H1 e){
    return new H1(visitText(e.text()));}
  public Node visitP(P e){
    return new P(visitText(e.text()));}
  public Node visitHtml(Html e){
    return new Html(visitHead(e.head()),visitBody(e.body()));}
  public Head visitHead(Head e){
    return new Head();}
  public Body visitBody(Body e){
    return new Body(map(e.divs(),ei->visitDiv(ei)));}
  public Div visitDiv(Div e){
    return new Div(map(e.ns(),ei->ei.accept(this)));}
  public A visitA(A e){
    return new A(visitUrl(e.href()),visitText(e.text()));}
  public Node visitUl(Ul e){
    return new Ul(map(e.lis(),ei->visitLi(ei)));}
  public Node visitOl(Ol e){
    return new Ol(map(e.lis(),ei->visitLi(ei)));}
  public Li visitLi(Li e){
    return new Li(e.node().accept(this));}
  }
```

Possibly the most useful visitor: the CloneVisitor!

Note how it is doing absolutely nothing!

How can it be the most useful one if it does nothing?

Using CloneVisitor

```
class ToItalian extends CloneVisitor{//usage Node n = myNode.accept(new ToItalian())
  @Override public String visitText(String text){return "Pizza!!"+text;}
  }
class ToItalianLi extends ToItalian{//usage myNode.accept(new ToItalianLi())
  boolean inLi=false;
  @Override public Li visitLi(Li li){
    boolean oldInLi=inLi;
    inLi=true;
    try{return super.visitLi(li);}
    finally{inLi=oldInLi;}}
  }
class OnlyUL extends CloneVisitor{//usage myNode.accept(new OnlyUL())
  public Node visitOl(Ol e){return new Ul(map(e.lis(),ei->visitLi(ei)));}
  }
class CollectAllText extends CloneVisitor{//usage new CollectAllText().collect(myNode)
  public String collect(Node n){//slow, it also clone it all.
    n.accept(this);//We could define a PropagatorVisitor
    return allText;//that does not clone the data
    }
  String allText="";
  @Override public String visitText(String text){allText+=text;return text;}
  }
```

Using CloneVisitor

```
class ToItalian extends CloneVisitor{//usage myNode.accept(new ToItalian())
  @Override public String visitText(String text){return "Pizza!!"+text;}
  }
class ToItalianLi extends ToItalian{//usage myNode.accept(new ToItalianLi())
  boolean inLi=false;
  @Override public Li visitLi(Li li){
    boolean oldInLi=inLi;
    inLi=true;
    try{return super.visitLi(li);}
    finally{inLi=oldInLi;}}
  }
class OnlyUL extends CloneVisitor{//usage myNode.accept(new OnlyUL())
  public Node visitOl(Ol e){return new Ul(map(e.lis(),ei->visitLi(ei)));}
  }
class CollectAllText extends CloneVisitor{//usage new CollectAllText().collect(myNode)
  public String collect(Node n){//slow, it also clone it all.
    n.accept(this);//We could define a PropagatorVisitor
    return allText;//that does not clone the data
    }
  String allText="";
  @Override public String visitText(String text){allText+=text;return text;}
  }
```

Using CloneVisitor

```
class ToItalian extends CloneVisitor{//usage myNode.accept(new ToItalian())
  @Override public String visitText(String text){return "Pizza!!"+text;}
  }
class ToItalianLi extends ToItalian{//usage myNode.accept(new ToItalianLi())
  boolean inLi=false;
  @Override public Li visitLi(Li li){
    boolean oldInLi=inLi;
    inLi=true;
    try{return super.visitLi(li);}
    finally{inLi=oldInLi;}}
  }
class OnlyUL extends CloneVisitor{//usage myNode.accept(new OnlyUL())
  public Node visitOl(Ol e){return new Ul(map(e.lis(),ei->visitLi(ei)));}
  }
class CollectAllText extends CloneVisitor{//usage new CollectAllText().collect(myNode)
  public String collect(Node n){//slow, it also clone it all.
    n.accept(this);//We could define a PropagatorVisitor
    return allText;//that does not clone the data
    }
  String allText="";
  @Override public String visitText(String text){allText+=text;return text;}
  }
```

Using CloneVisitor

```
class ToItalian extends CloneVisitor{//usage myNode.accept(new ToItalian())
  @Override public String visitText(String text){return "Pizza!!"+text;}
  }
class ToItalianLi extends ToItalian{//usage myNode.accept(new ToItalianLi())
  boolean inLi=false;
  @Override public Li visitLi(Li li){
    boolean oldInLi=inLi;
    inLi=true;
    try{return super.visitLi(li);}
    finally{inLi=oldInLi;}}
  }
class OnlyUL extends CloneVisitor{//usage myNode.accept(new OnlyUL())
  public Node visitOl(Ol e){return new Ul(map(e.lis(),ei->visitLi(ei)));}
  }
class CollectAllText extends CloneVisitor{//usage new CollectAllText().collect(myNode)
  public String collect(Node n){//slow, it also clone it all.
    n.accept(this);//We could define a PropagatorVisitor
    return allText;//that does not clone the data
    }
  String allText="";
  @Override public String visitText(String text){allText+=text;return text;}
  }
```

Using CloneVisitor

```
class ToItalian extends CloneVisitor{//usage myNode.accept(new ToItalian())
  @Override public String visitText(String text){return "Pizza!!"+text;}
  }
class ToItalianLi extends ToItalian{//usage myNode.accept(new ToItalianLi())
  boolean inLi=false;
  @Override public Li visitLi(Li li){
    boolean oldInLi=inLi;
    inLi=true;
    try{return super.visitLi(li);}
    finally{inLi=oldInLi;}}
  }
class OnlyUL extends CloneVisitor{//usage myNode.accept(new OnlyUL())
  public Node visitOl(Ol e){return new Ul(map(e.lis(),ei->visitLi(ei)));}
  }
class CollectAllText extends CloneVisitor{//usage new CollectAllText().collect(myNode)
  public String collect(Node n){//slow, it also clone it all.
    n.accept(this);//We could define a PropagatorVisitor
    return allText;//that does not clone the data
    }
  String allText="";
  @Override public String visitText(String text){allText+=text;return text;}
  }
```

The power of code reuse!

- The CloneVisitor defines and invoke a lot of useful handles.

- CloneVisitor can be extended!
- Many apparently unrelated operations are now very easy!

- Generally speaking, many programming patterns look like pointless code at first!

PropagatorVisitor

```
class PropagatorVisitor implements Visitor<Boolean>{
  //return true == continue the visit
  public Boolean visitText(String text){return true;}
  public Boolean visitUrl(String url){return true;}
  public Boolean visitH1(H1 e){return visitText(e.text());}
  public Boolean visitP(P e){return visitText(e.text());}
  public Boolean visitHtml(Html e){return visitHead(e.head()) && visitBody(e.body());}
  public Boolean visitHead(Head e){return true;}
  public Boolean visitBody(Body e){return all(e.divs(),ei->visitDiv(ei));}
  public Boolean visitDiv(Div e){return all(e.ns(),ei->ei.accept(this));}
  public Boolean visitA(A e){return visitUrl(e.href()) && visitText(e.text());}
  public Boolean visitUl(Ul e){return all(e.lis(),ei->visitLi(ei));}
  public Boolean visitOl(Ol e){return all(e.lis(),ei->visitLi(ei));}
  public Boolean visitLi(Li e){return e.node().accept(this);}
  public static <T extends Node> Boolean all(List<T>ts,Predicate<T>test){
    return ts.stream().allMatch(test);}//short-circuit on false
  }
```

Using PropagatorVisitor

```
class CollectAllText2 extends PropagatorVisitor{//better version of CollectAllText
  public String collect(Node n){//usage new CollectAllText2().collect(myNode)
    n.accept(this);
    return allText;
    }
  String allText="";
  @Override public Boolean visitText(String text){allText+=text;return true;}
  }

class ContainsItalian extends PropagatorVisitor{
  //usage new ContainsItalian().check(myNode)
  public boolean check(Node n){return !n.accept(this);}//short-circuit on false
  @Override public Boolean visitText(String text){return !text.contains("Pizza");}
  }
```

- In some cases, extending PropagatorVisitor get more efficient code than just extending CloneVisitor.

Using PropagatorVisitor

```
class CollectAllText2 extends PropagatorVisitor{//better version of CollectAllText
  public String collect(Node n){//usage new CollectAllText2().collect(myNode)
    n.accept(this);
    return allText;
    }
  String allText="";
  @Override public Boolean visitText(String text){allText+=text;return true;}
  }

class ContainsItalian extends PropagatorVisitor{
  //usage new ContainsItalian().check(myNode)
  public boolean check(Node n){return !n.accept(this);}//short-circuit on false
  @Override public Boolean visitText(String text){return !text.contains("Pizza");}
  }
```

- In some cases, extending PropagatorVisitor get more efficient code than just extending CloneVisitor.

Using PropagatorVisitor

```
class CollectAllText2 extends PropagatorVisitor{//better version of CollectAllText
  public String collect(Node n){//usage new CollectAllText2().collect(myNode)
    n.accept(this);
    return allText;
    }
  String allText="";
  @Override public Boolean visitText(String text){allText+=text;return true;}
  }

class ContainsItalian extends PropagatorVisitor{
  //usage new ContainsItalian().check(myNode)
  public boolean check(Node n){return !n.accept(this);}//short-circuit on false
  @Override public Boolean visitText(String text){return !text.contains("Pizza");}
  }
```

- In some cases, extending PropagatorVisitor get more efficient code than just extending CloneVisitor.

Code Code Code

- So much code, so much repetitive code
- Learn to see the patterns inside the code
- All the code shown today is basically the same in FJ!
- Patterns are all about ways to inject handlers to reuse behaviour!