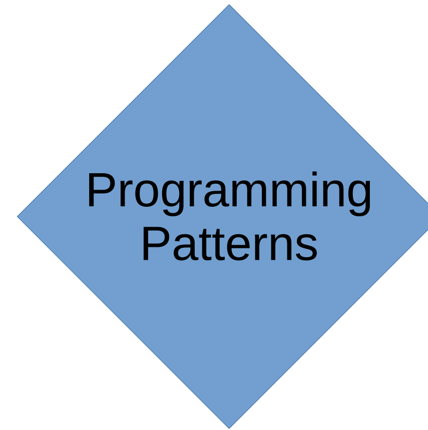
SWEN423 - Lecture 14



MockTermTest2 ModelSolutions

Date change for Final term test

- The school has discussed how to mitigate the stress in the final weeks of this trimester, and we decided to move SWEN423 final test earlier to 19 October 2020.
- We will not introduce new material in the very last lecture of the course but just review the already presented material.

Mock Term test 2: the exercise is about a variation of a binary tree

- Pure OO
- Functional: no state mutation
- Flyweight

```
public abstract class Tree<T>{
  interface On0<T,R>{R of(T label);}
  interface On1<T,R>{R of(T label,Tree<T>child1);}
  interface On2<T,R>{R of(T label,Tree<T>child1,Tree<T>child2);}
  private static class Tree0<T> extends Tree<T>{/*the tree with no children*/}
  private static class Tree1<T> extends Tree<T>{/*the tree with 1 child*/}
  private static class Tree2<T> extends Tree<T>{/*the tree with 2 children*/}
  private Tree(){}
  private static HashMap<Object,Tree<?>> cache0=new HashMap<>();
  @SuppressWarnings("unchecked") public static <T> Tree<T> of(T label){
    return (Tree<T>)cache0.computeIfAbsent(label,Tree0::new);}
  public abstract <R> R match(On0<T,R>a,On1<T,R>b,On2<T,R>c);
  public abstract Tree<T> withChild(Tree<T>child,Function<Tree<T>,Tree<T>>orElse);
  public Tree<T> withoutFirst(){return this;}
  public Tree<T> withoutSecond(){return this;}
  public T label(){return match(l->l,(l,c1)->l,(l,c1,c2)->l);}
  public Tree<T> first(Function<Tree<T>,Tree<T>>orElse){
    return match(l->orElse.apply(this),(l,c1)->c1,(l,c1,c2)->c1);}
  public Tree<T> second(Function<Tree<T>,Tree<T>>orElse){
    return match(l->orElse.apply(this),(l,c1)->orElse.apply(this),(l,c1,c2)->c2);}
  public void save(Path fileName,Function<T,ElemRepr<? extends T>>f){
    Persistence.save(fileName,TreeRepr.of(this,f));}
  public static <T> Tree<T> load(Path fileName){
    TreeRepr<T> s=Persistence.load(fileName);
    return s.get();}
  }
```

```java
public abstract class Tree<T>{
  interface On0<T,R>{R of(T label);}
  interface On1<T,R>{R of(T label,Tree<T>child1);}
  interface On2<T,R>{R of(T label,Tree<T>child1,Tree<T>child2);}
  private static class Tree0<T> extends Tree<T>{/*the tree with no children*/}
  private static class Tree1<T> extends Tree<T>{/*the tree with 1 child*/}
  private static class Tree2<T> extends Tree<T>{/*the tree with 2 children*/}
  private Tree(){}
  private static HashMap<Object,Tree<?>> cache0=new HashMap<>();
  @SuppressWarnings("unchecked") public static <T> Tree<T> of(T label){
    return (Tree<T>)cache0.computeIfAbsent(label,Tree0::new);}
  public abstract <R> R match(On0<T,R>a,On1<T,R>b,On2<T,R>c);
  public abstract Tree<T> withChild(Tree<T>child,Function<Tree<T>,Tree<T>>orElse);
  public Tree<T> withoutFirst(){return this;}
  public Tree<T> withoutSecond(){return this;}
  public T label(){return match(l->l,(l,c1)->l,(l,c1,c2)->l);}
  public Tree<T> first(Function<Tree<T>,Tree<T>>orElse){
    return match(l->orElse.apply(this),(l,c1)->c1,(l,c1,c2)->c1);}
  public Tree<T> second(Function<Tree<T>,Tree<T>>orElse){
    return match(l->orElse.apply(this),(l,c1)->orElse.apply(this),(l,c1,c2)->c2);}
  public void save(Path fileName,Function<T,ElemRepr<? extends T>>f){
    Persistence.save(fileName,TreeRepr.of(this,f));}
  public static <T> Tree<T> load(Path fileName){
    TreeRepr<T> s=Persistence.load(fileName);
    return s.get();}
  }
```

```java
public abstract class Tree<T>{
  interface On0<T,R>{R of(T label);}
  interface On1<T,R>{R of(T label,Tree<T>child1);}
  interface On2<T,R>{R of(T label,Tree<T>child1,Tree<T>child2);}
  private static class Tree0<T> extends Tree<T>{/*the tree with no children*/}
  private static class Tree1<T> extends Tree<T>{/*the tree with 1 child*/}
  private static class Tree2<T> extends Tree<T>{/*the tree with 2 children*/}
  private Tree(){}
  private static HashMap<Object,Tree<?>> cache0=new HashMap<>();
  @SuppressWarnings("unchecked") public static <T> Tree<T> of(T label){
    return (Tree<T>)cache0.computeIfAbsent(label,Tree0::new);}
  public abstract <R> R match(On0<T,R>a,On1<T,R>b,On2<T,R>c);//Two abstract methods
  public abstract Tree<T> withChild(Tree<T>child,Function<Tree<T>,Tree<T>>orElse);
  public Tree<T> withoutFirst(){return this;}
  public Tree<T> withoutSecond(){return this;}
  public T label(){return match(l->l,(l,c1)->l,(l,c1,c2)->l);}
  public Tree<T> first(Function<Tree<T>,Tree<T>>orElse){
    return match(l->orElse.apply(this),(l,c1)->c1,(l,c1,c2)->c1);}
  public Tree<T> second(Function<Tree<T>,Tree<T>>orElse){
    return match(l->orElse.apply(this),(l,c1)->orElse.apply(this),(l,c1,c2)->c2);}
  public void save(Path fileName,Function<T,ElemRepr<? extends T>>f){
    Persistence.save(fileName,TreeRepr.of(this,f));}
  public static <T> Tree<T> load(Path fileName){
    TreeRepr<T> s=Persistence.load(fileName);
    return s.get();}
  }
```

```java
public abstract class Tree<T>{
  interface On0<T,R>{R of(T label);}
  interface On1<T,R>{R of(T label,Tree<T>child1);}
  interface On2<T,R>{R of(T label,Tree<T>child1,Tree<T>child2);}
  private static class Tree0<T> extends Tree<T>{/*the tree with no children*/}
  private static class Tree1<T> extends Tree<T>{/*the tree with 1 child*/}
  private static class Tree2<T> extends Tree<T>{/*the tree with 2 children*/}
  private Tree(){}
  private static HashMap<Object,Tree<?>> cache0=new HashMap<>();
  @SuppressWarnings("unchecked") public static <T> Tree<T> of(T label){
    return (Tree<T>)cache0.computeIfAbsent(label,Tree0::new);}
  public abstract <R> R match(On0<T,R>a,On1<T,R>b,On2<T,R>c);
  public abstract Tree<T> withChild(Tree<T>child,Function<Tree<T>,Tree<T>>orElse);
  public Tree<T> withoutFirst(){return this;}//could be done with match instead
  public Tree<T> withoutSecond(){return this;}//but is more efficient if overridden
  public T label(){return match(l->l,(l,c1)->l,(l,c1,c2)->l);}
  public Tree<T> first(Function<Tree<T>,Tree<T>>orElse){
    return match(l->orElse.apply(this),(l,c1)->c1,(l,c1,c2)->c1);}
  public Tree<T> second(Function<Tree<T>,Tree<T>>orElse){
    return match(l->orElse.apply(this),(l,c1)->orElse.apply(this),(l,c1,c2)->c2);}
  public void save(Path fileName,Function<T,ElemRepr<? extends T>>f){
    Persistence.save(fileName,TreeRepr.of(this,f));}
  public static <T> Tree<T> load(Path fileName){
    TreeRepr<T> s=Persistence.load(fileName);
    return s.get();}
  }
```

```
public abstract class Tree<T>{
  interface On0<T,R>{R of(T label);}
  interface On1<T,R>{R of(T label,Tree<T>child1);}
  interface On2<T,R>{R of(T label,Tree<T>child1,Tree<T>child2);}
  private static class Tree0<T> extends Tree<T>{/*the tree with no children*/}
  private static class Tree1<T> extends Tree<T>{/*the tree with 1 child*/}
  private static class Tree2<T> extends Tree<T>{/*the tree with 2 children*/}
  private Tree(){}
  private static HashMap<Object,Tree<?>> cache0=new HashMap<>();
  @SuppressWarnings("unchecked") public static <T> Tree<T> of(T label){
    return (Tree<T>)cache0.computeIfAbsent(label,Tree0::new);}
  public abstract <R> R match(On0<T,R>a,On1<T,R>b,On2<T,R>c);
  public abstract Tree<T> withChild(Tree<T>child,Function<Tree<T>,Tree<T>>orElse);
  public Tree<T> withoutFirst(){return this;}
  public Tree<T> withoutSecond(){return this;}
  public T label(){return match(l->l,(l,c1)->l,(l,c1,c2)->l);}
  public Tree<T> first(Function<Tree<T>,Tree<T>>orElse){
    return match(l->orElse.apply(this),(l,c1)->c1,(l,c1,c2)->c1);}
  public Tree<T> second(Function<Tree<T>,Tree<T>>orElse){
    return match(l->orElse.apply(this),(l,c1)->orElse.apply(this),(l,c1,c2)->c2);}
  public void save(Path fileName,Function<T,ElemRepr<? extends T>>f){
    Persistence.save(fileName,TreeRepr.of(this,f));}
  public static <T> Tree<T> load(Path fileName){
    TreeRepr<T> s=Persistence.load(fileName);
    return s.get();}
  }
```

```java
public abstract class Tree<T>{
  interface On0<T,R>{R of(T label);}
  interface On1<T,R>{R of(T label,Tree<T>child1);}
  interface On2<T,R>{R of(T label,Tree<T>child1,Tree<T>child2);}
  private static class Tree0<T> extends Tree<T>{/*the tree with no children*/}
  private static class Tree1<T> extends Tree<T>{/*the tree with 1 child*/}
  private static class Tree2<T> extends Tree<T>{/*the tree with 2 children*/}
  private Tree(){}
  private static HashMap<Object,Tree<?>> cache0=new HashMap<>();
  @SuppressWarnings("unchecked") public static <T> Tree<T> of(T label){
    return (Tree<T>)cache0.computeIfAbsent(label,Tree0::new);}
  public abstract <R> R match(On0<T,R>a,On1<T,R>b,On2<T,R>c);
  public abstract Tree<T> withChild(Tree<T>child,Function<Tree<T>,Tree<T>>orElse);
  public Tree<T> withoutFirst(){return this;}
  public Tree<T> withoutSecond(){return this;}
  public T label(){return match(l->l,(l,c1)->l,(l,c1,c2)->l);}
  public Tree<T> first(Function<Tree<T>,Tree<T>>orElse){
    return match(l->orElse.apply(this),(l,c1)->c1,(l,c1,c2)->c1);}
  public Tree<T> second(Function<Tree<T>,Tree<T>>orElse){
    return match(l->orElse.apply(this),(l,c1)->orElse.apply(this),(l,c1,c2)->c2);}
  public void save(Path fileName,Function<T,ElemRepr<? extends T>>f){
    Persistence.save(fileName,TreeRepr.of(this,f));}
  public static <T> Tree<T> load(Path fileName){
    TreeRepr<T> s=Persistence.load(fileName);
    return s.get();}
  }
```

```java
public abstract class Tree<T>{
  interface On0<T,R>{R of(T label);}
  interface On1<T,R>{R of(T label,Tree<T>child1);}
  interface On2<T,R>{R of(T label,Tree<T>child1,Tree<T>child2);}
  private static class Tree0<T> extends Tree<T>{/*the tree with no children*/}
  private static class Tree1<T> extends Tree<T>{/*the tree with 1 child*/}
  private static class Tree2<T> extends Tree<T>{/*the tree with 2 children*/}
  private Tree(){}
  private static HashMap<Object,Tree<?>> cache0=new HashMap<>();
  @SuppressWarnings("unchecked") public static <T> Tree<T> of(T label){
    return (Tree<T>)cache0.computeIfAbsent(label,Tree0::new);}
  public abstract <R> R match(On0<T,R>a,On1<T,R>b,On2<T,R>c);
  public abstract Tree<T> withChild(Tree<T>child,Function<Tree<T>,Tree<T>>orElse);
  public Tree<T> withoutFirst(){return this;}
  public Tree<T> withoutSecond(){return this;}
  public T label(){return match(l->l,(l,c1)->l,(l,c1,c2)->l);}
  public Tree<T> first(Function<Tree<T>,Tree<T>>orElse){
    return match(l->orElse.apply(this),(l,c1)->c1,(l,c1,c2)->c1);}
  public Tree<T> second(Function<Tree<T>,Tree<T>>orElse){
    return match(l->orElse.apply(this),(l,c1)->orElse.apply(this),(l,c1,c2)->c2);}
  public void save(Path fileName,Function<T,ElemRepr<? extends T>>f){
    Persistence.save(fileName,TreeRepr.of(this,f));}
  public static <T> Tree<T> load(Path fileName){
    TreeRepr<T> s=Persistence.load(fileName);
    return s.get();}
  }
```

```
private static class Tree0<T> extends Tree<T>{//the tree with 0 children
  T label; Tree0(T label){this.label=label;}
  private HashMap<Tree<T>,Tree<T>> cache1=new HashMap<>();//map of objects created by this
  public <R> R match(On0<T,R>a,On1<T,R>b,On2<T,R>c){return a.of(label);}//match/visitor
  public Tree<T> withChild(Tree<T>child,Function<Tree<T>,Tree<T>>orElse){//factory
    return cache1.computeIfAbsent(child,c->new Tree1<T>(this,c));
    }
  }
private static class Tree1<T> extends Tree<T>{//the tree with 1 child
  Tree0<T> t0;Tree<T> c1; Tree1(Tree0<T> t0,Tree<T> c1){this.t0=t0;this.c1=c1;}
  private HashMap<Tree<T>,Tree<T>> cache2=new HashMap<>();
  public <R> R match(On0<T,R>a,On1<T,R>b,On2<T,R>c){return b.of(t0.label,c1);}
  public Tree<T> withChild(Tree<T>child,Function<Tree<T>,Tree<T>>orElse){
    return cache2.computeIfAbsent(child,c->new Tree2<T>(this,c));
    }
  public Tree<T> withoutFirst(){return t0;}
  }
private static class Tree2<T> extends Tree<T>{//the tree with 2 children
  Tree1<T> t1;Tree<T> c2; Tree2(Tree1<T> t1,Tree<T> c2){this.t1=t1;this.c2=c2;}
  public <R> R match(On0<T,R>a,On1<T,R>b,On2<T,R>c){return c.of(t1.t0.label,t1.c1,c2);}
  public Tree<T> withChild(Tree<T>child,Function<Tree<T>,Tree<T>>orElse){
    return orElse.apply(this);}
  public Tree<T> withoutFirst(){return t1.t0.withChild(c2,t->t);}//t->t is dead code
  public Tree<T> withoutSecond(){return t1;}
  }
```

```java
private static class Tree0<T> extends Tree<T>{//the tree with 0 children
  T label; Tree0(T label){this.label=label;}
  private HashMap<Tree<T>,Tree<T>> cache1=new HashMap<>();
  public <R> R match(On0<T,R>a,On1<T,R>b,On2<T,R>c){return a.of(label);}
  public Tree<T> withChild(Tree<T>child,Function<Tree<T>,Tree<T>>orElse){
    return cache1.computeIfAbsent(child,c->new Tree1<T>(this,c));
    }
  }//Tree1 contains a Tree0, Tree2 contains a Tree1
private static class Tree1<T> extends Tree<T>{//the tree with 1 child
  Tree0<T> t0;Tree<T> c1; Tree1(Tree0<T> t0,Tree<T> c1){this.t0=t0;this.c1=c1;}
  private HashMap<Tree<T>,Tree<T>> cache2=new HashMap<>();//map of objects created by this
  public <R> R match(On0<T,R>a,On1<T,R>b,On2<T,R>c){return b.of(t0.label,c1);}
  public Tree<T> withChild(Tree<T>child,Function<Tree<T>,Tree<T>>orElse){
    return cache2.computeIfAbsent(child,c->new Tree2<T>(this,c));
    }
  public Tree<T> withoutFirst(){return t0;}//just a field access!
  }
private static class Tree2<T> extends Tree<T>{//the tree with 2 children
  Tree1<T> t1;Tree<T> c2; Tree2(Tree1<T> t1,Tree<T> c2){this.t1=t1;this.c2=c2;}
  public <R> R match(On0<T,R>a,On1<T,R>b,On2<T,R>c){return c.of(t1.t0.label,t1.c1,c2);}
  public Tree<T> withChild(Tree<T>child,Function<Tree<T>,Tree<T>>orElse){
    return orElse.apply(this);}
  public Tree<T> withoutFirst(){return t1.t0.withChild(c2,t->t);}//t->t is dead code
  public Tree<T> withoutSecond(){return t1;}
  }
```

```java
private static class Tree0<T> extends Tree<T>{//the tree with 0 children
  T label; Tree0(T label){this.label=label;}
  private HashMap<Tree<T>,Tree<T>> cache1=new HashMap<>();
  public <R> R match(On0<T,R>a,On1<T,R>b,On2<T,R>c){return a.of(label);}
  public Tree<T> withChild(Tree<T>child,Function<Tree<T>,Tree<T>>orElse){
    return cache1.computeIfAbsent(child,c->new Tree1<T>(this,c));
    }
  }
private static class Tree1<T> extends Tree<T>{//the tree with 1 child
  Tree0<T> t0;Tree<T> c1; Tree1(Tree0<T> t0,Tree<T> c1){this.t0=t0;this.c1=c1;}
  private HashMap<Tree<T>,Tree<T>> cache2=new HashMap<>();
  public <R> R match(On0<T,R>a,On1<T,R>b,On2<T,R>c){return b.of(t0.label,c1);}
  public Tree<T> withChild(Tree<T>child,Function<Tree<T>,Tree<T>>orElse){
    return cache2.computeIfAbsent(child,c->new Tree2<T>(this,c));
    }
  public Tree<T> withoutFirst(){return t0;}
  }
private static class Tree2<T> extends Tree<T>{//the tree with 2 children
  Tree1<T> t1;Tree<T> c2; Tree2(Tree1<T> t1,Tree<T> c2){this.t1=t1;this.c2=c2;}
  public <R> R match(On0<T,R>a,On1<T,R>b,On2<T,R>c){return c.of(t1.t0.label,t1.c1,c2);}
  public Tree<T> withChild(Tree<T>child,Function<Tree<T>,Tree<T>>orElse){
    return orElse.apply(this);}
  public Tree<T> withoutFirst(){return t1.t0.withChild(c2,t->t);}//t->t is dead code
  public Tree<T> withoutSecond(){return t1;}
  }
```

```java
private static class Tree0<T> extends Tree<T>{//the tree with 0 children
  T label; Tree0(T label){this.label=label;}
  private HashMap<Tree<T>,Tree<T>> cache1=new HashMap<>();
  public <R> R match(On0<T,R>a,On1<T,R>b,On2<T,R>c){return a.of(label);}
  public Tree<T> withChild(Tree<T>child,Function<Tree<T>,Tree<T>>orElse){
    return cache1.computeIfAbsent(child,c->new Tree1<T>(this,c));
    }
  }
private static class Tree1<T> extends Tree<T>{//the tree with 1 child
  Tree0<T> t0;Tree<T> c1; Tree1(Tree0<T> t0,Tree<T> c1){this.t0=t0;this.c1=c1;}
  private HashMap<Tree<T>,Tree<T>> cache2=new HashMap<>();
  public <R> R match(On0<T,R>a,On1<T,R>b,On2<T,R>c){return b.of(t0.label,c1);}
  public Tree<T> withChild(Tree<T>child,Function<Tree<T>,Tree<T>>orElse){
    return cache2.computeIfAbsent(child,c->new Tree2<T>(this,c));
    }
  public Tree<T> withoutFirst(){return t0;}
  }
private static class Tree2<T> extends Tree<T>{//the tree with 2 children
  Tree1<T> t1;Tree<T> c2; Tree2(Tree1<T> t1,Tree<T> c2){this.t1=t1;this.c2=c2;}
  public <R> R match(On0<T,R>a,On1<T,R>b,On2<T,R>c){return c.of(t1.t0.label,t1.c1,c2);}
  public Tree<T> withChild(Tree<T>child,Function<Tree<T>,Tree<T>>orElse){
    return orElse.apply(this);}
  public Tree<T> withoutFirst(){return t1.t0.withChild(c2,t->t);}//t->t is dead code
  public Tree<T> withoutSecond(){return t1;}
  }
```

Serialization

```
interface ElemRepr<T> extends Serializable{T get();}

interface TreeRepr<T> extends Serializable{
  Tree<T> get();
  static<T> TreeRepr<T> of(Tree<T> t,Function<T,ElemRepr<? extends T>>f){
    return t.match((l)->tree0(l),(l,c1)->tree1(t,f),(l,c1,c2)->tree2(t,f));
    }
  static <T> TreeRepr<T> tree0(T l){return ()->Tree.<T>of(l);}
  static <T> TreeRepr<T> tree1(Tree<T>t,Function<T,ElemRepr<? extends T>>f){
    var c1R=TreeRepr.of(t.first(x->x),f);
    var t0R=TreeRepr.of(t.withoutFirst(),f);
    return ()->t0R.get().withChild(c1R.get(),x->x);
    }
  static <T> TreeRepr<T> tree2(Tree<T>t,Function<T,ElemRepr<? extends T>>f){
    var c2R=TreeRepr.of(t.second(x->x),f);
    var t1R=TreeRepr.of(t.withoutSecond(),f);
    return ()->t1R.get().withChild(c2R.get(),x->x);
    }
  }
```

The two operations

```
class Operations{
  static int sum(Tree<Integer>tree){
    return tree.match(
      l->l,
      (l,c1)->l+sum(c1),
      (l,c1,c2)->l+sum(c1)+sum(c2));
    }
  static Tree<Integer> twice(Tree<Integer>tree){
    return tree.match(
      l->Tree.of(l*2),
      (l,c1)->Tree.of(l*2).withChild(twice(c1),x->x),
      (l,c1,c2)->Tree.of(l*2).withChild(twice(c1),x->x).withChild(twice(c2),x->x));
    }
  }
```

Compare and contrast the two following papers

“The power of interoperability” and “Modules as Objects in Newspeak”.

- The power of interoperability discuss the minimal features that a language needs to have to be considered Object Oriented, and shows some OO encoding on functional languages.
- Modules as Objects in Newspeak shows a language design when both fields and constructors are replaced with instance methods, and “nested classes” are just such methods.

- Is there any position where the authors agree?
- is there any position where the authors disagree?
- Which paper gave you more insight in the art of programming, and why?

Compare and contrast the two following papers

The power of interoperability shows how objects are just records of closures that close on the same state: the record itself and the object fields.

Thus showing how fields are not needed as a primitive language feature for OO languages.

Note, this shows objects are a record of their methods, not as a record of their fields, as it could be naively assumed. Thus fields are not fundamental for OO.

Those records can be created by a dedicated function, that would serve the role of a constructor. Thus, also constructors are not fundamental for OO.

The two authors seams to be in great harmony, since those observation come to life in the programming language Newspeak, where methods are truly the protagonist, removing the need of both fields and constructors. Newspeak expands the work of Jonathan showing that also nested classes can be obtained by using methods. Moreover, such nested classes will be “virtual” thus supporting in a clean and direct way the concept of family polymorphism.

Both papers had greatly influenced my understanding of Pure OO; Jonathan work has been more influential, since it is more focused: it presents few clear ideas and make them accessible to a wide audience (of functional programmers).