Code reuse with 42

Many mains in 42

```
reuse L42.is/AdamTowel

Main0: Debug(S"Hello world1")

Main1:{
  Debug(S"Let's do something more")
  x=7Num*6Num
  Debug(S"Hello "++x)
  return {
    class method S foo()=S"Surprise"
    }
  }
//Our computation can return CODE; code in 42 is of type 'Library'

Main2: Debug(S"Hello "++Main1.foo())
//Main2 can use Main1, since is declared later.
//Classes are compiled top-bottom
```

My first class

```
reuse L42.is/AdamTowel
Point:Data<><{Num x,Num y}
Main2: Debug(S"Hello "++Point(x:4Num, y:2Num))
//-lets dissect the code above:
{Num x,Num y}//(0)an unnamed class-body
Num x //(1)a final field x of type Num; it is equivalent to a getter
method Num x() //(2)the equivalent getter.  Note:  is abstract/no body
var Num x //(3)a non final field.  Equivalent to getter+setter
Point:Data<><{..} //(4) this is just a main that return code.
//That code will be the definition for class 'Point'
Data<><{..} //(5)Data is a class decorator.
//In 42 operators are method calls; you can read it as
//Data.babelFish(Num x,Num y)
//Data adds features to the argument:  factory, equality, toS,...
Point(x:4Num, y:2Num) //(6) factory call.  Parameter names are required.
```

My first Trait compostion

```
Part1:Trait<><{
  class method S message()

  class method S quoteMessage()=
    S"The message is :'"++this.message()++S"'."
  }

Part2:Trait<><{ class method S message()=S"42" }

Result:Part1+Part2

Main:Debug(Result.quoteMessage())//S"The message is :'42'."

//-lets dissect the code above:
Trait<><{..} //The trait wraps the input code to support operations.
t1+t2//simmetric composition, match by member name
//error if a method is implemented in both t1 and t2
Result:{//was Part1+Part2; flattening semantics:
  class method S message()=S"42"

  class method S quoteMessage()=
    S"The message is :'"++this.message()++S"'."
  }
```

Modularization with traits

```
Game:{//example game code, NOT MODULARIZED
  Map:Data<><{/*map implementation*/
    method Item get(Point that)=..,,,,,method Void set(Item that)=..
    }
  Item:{interface,,,,,Point point, method Item break()}//Item can have many methods
  Rock:Data<><{implements Item,,,,,Num weight
    method break()=Rock(weight:this.weight()-1Num)
    }
  Wall:Data<><{implements Item,,,,,Num height
    method break()=Rock(weight:100Num)
    } //we could have more kinds of Items, not just Rock and Wall
  class method Map load(S fileName)=..//Alice writes load(_)
    /*read from file, divide in lines, call read(line)*/
  class method Void load(Map map,S line)={//example line:  S"Rock 23 in 12, 7"
    ns=line.readNums()
    if line.startsWith(S"Rock") (
      map.set(Rock(weight:1ns,point:Point(x:2ns,x:3ns)))
      )
    if line.startsWith(S"Wall") (..)
    ...
    }
  class method Void run()=..this.load(..)..
  }
Main:Game.run()
```

Modularized code

```
Alice:Trait<><{//could be in its own file
  Map:{class method Map(),,,,,method Void set(Item that)}
  Item:{interface,,,,,Point point}
  Rock:{implements Item,,,,,class method Rock(Num weight, Point point) }
  //the above is all abstract
  class method Map load(S fileName)=..//Alice writes load(_)
    /*read from file, divide in lines, call read(line)*/
  class method Void load(Map map,S line)={//example line:  S"Rock 23 in 12, 7"
    ns=line.readNums()
    if line.startsWith(S"Rock") (
      map.set(Rock(weight:1ns,point:Point(x:2ns,x:3ns)))
      )
    if line.startsWith(S"Wall") (..)
    ...
    }
  }
Bob:{/*the Game code as before, without the two methods of Alice*/
  class method Map load(S fileName)// we declare an abstract load method
  }
Game=Alice+Bob
Main:Game.run()
//both Alice and Bob are independent.  Map,Item,Rock inside of Alice
//are unrelated with Map,Item,Rock inside Bob.
//the sum then merges the members with the same name.
//The flattened result is exactly the code of before.
```

Modularized code=testable code

```
Alice:Trait<><{/*as before*/}
AliceMock:Alice+{
  Item:{interface,,,,,Point point}
  Map:Collections.hashmap(Point to:Item)+{
    method Void set(Item that)=this.put(key:that.point(),val:that)
    }
  Rock:Data<><{implements Item,,,,,Num weight}

  class method Void test(S fileName,S expected)={
    map=MapReader.read(fileName:fileName)
    Debug.test(map, expected:expected)
    }
Test1:AliceMock.test(S"justARock.txt",S"HashMap[Point(..)->Rock(..)]"
Test2:...
...
```

Those tests are completely independent of Bob code, and can be run even before Bob started writing any code. By composing independently testable code, you can put you project together after every component has been independently tested. You can (and should) do it also in Java. However, it is so much harder to do so.

Dependency injection Hell in Java

```
//common code:
interface Map{..} class Point{..}//most of those requires their own file
interface Item{..}, interface Rock extends Item {..},..
interface ItemFactory{Rock makeRock(Point p, weight w); ..}
interface MapFactory{Map makeMap();}
//Alice code:
class MapLoader{
  ItemFactory items; MapFactory maps;
  MapLoader(ItemFactory i,MapFactory m){items=i;maps=m;}
  Map load(String fileName){..maps.makeMap()..}
  Void load(Map map,String fileName){..items.makeRock(..)..}
  }
//Alice mocking code
class MockMap implements Map{...}
class MockMapFactory implements MapFactory{//this may be a lambda in Java8
  public Map makeMap(){return new MockMap();}  }
class MockItemFactory implements ItemFactory{//but this can not
  public Rock makeRock(..){return new MockRock(..);}
  public Wall makeWall(..){return new MockWall(..);}  } ..
class MockRock implements Rock{..} class MockWall implements Rock{..}
class Tester{
  static void test(String fileName,String expected)={
    MapLoader m=new MapLoader(new MockMapFactory(),new MockItemFactory());
    Map map=m.load(fileName:fileName)
    assert map.toString().equals(expected);
    }
...
```

First exposure to 42

```
Point: {(N x, N y)  'class point and field declarations
  method N distance(Point that){ 'method declaration
    var tmpX=this.x()-that.x()  'fields are getters
    tmpX:=tmpX*tmpX  'allowed since declared var
    var tmpY=this.y()-that.y()  'omitted var type
    tmpY*=tmpY   'usual Java/C++ operators
    return (tmpX+tmpY).sqrt()  'pure oo view
    }
  method Point  add(N x){ 'method declaration
    return Point(x:this.x()+x, y:this.y()) 'constructor from fields
    }
  }
```

Minimal OO capabilities:

- (final) classes with state (constructor composed by fields) and behaviour;
- single dispatch method call, with nominal method selectors;
- interfaces as the only way to induce subtyping; that is, all class types are exact types.

First exposure to 42

```
Point: { (N x, N y) ... }
Main:{
  p1= Point(x:12N,y:22N)  'this is declaring a final local var
  p2= Point(x:12N,y:22N)
   'check=p1==p2 'equivalent of Java .equals
   'method == is undefined
  }
```

With **`Point`** as defined before, we can do very little with points. We may want to manually add

- equality, inequality, hashCode
- since all the fields can be compared (by using $>=$) then we may want to implement a method comparing points lexicographically
- toString, serialize, deserialize/parse, toHTML, clone,...

Implementing such methods is not only very boring,

First exposure to 42

```
Point: { (N x, N y) ... }
Main:{
  p1= Point(x:12N,y:22N)  'this is declaring a final local var
  p2= Point(x:12N,y:22N)
   'check=p1==p2 'equivalent of Java .equals
   'method == is undefined
  }
```

With **Point** as defined before, we can do very little with points. We may want to manually add

- equality, inequality, hashCode
- since all the fields can be compared (by using >=) then we may want to implement a method comparing points lexicographically
- toString, serialize, deserialize/parse, toHTML, clone,...

Implementing such methods is not only very boring, It is super tricky!

How can it be so hard?

- It is actually order of magnitude simpler to use objects defining those methods than to define them correctly.

How can it be so hard?

- It is actually order of magnitude simpler to use objects defining those methods than to define them correctly.
- Up to the point that every year multiple articles challenge the establish standard on how to do those kinds of operation on arbitrarily shaped objects/object graphs.

How can it be so hard?

- It is actually order of magnitude simpler to use objects defining those methods than to define them correctly.
- Up to the point that every year multiple articles challenge the establish standard on how to do those kinds of operation on arbitrarily shaped objects/object graphs.
- If automatic generations of those methods was a language feature it would be outdated very fast.

First exposure to class Decorators

```
Point: Data[]<<{ (N x, N y) ... }
Main:{
  p1= Point(x:12N,y:22N)  'this is declaring a final local var
  p2= Point(x:12N,y:22N)
  check=p1==p2  'true
  less=p1>p2  'false
  }
```

`Data` is a class, whose instances are class decorators. A class decorator is an object offering a method named `<<` that takes a library (=a class with nested classes) and return a library; that is: `method Library <<(Library that)`. Since class decorator tends to have many options, they can be instantiated using the square brackets, as in `Data[...optionName1:value1;optionName2:value2;...]`. (That is the syntax for any var-arg method, including convenient collections initialization as `NVec[a;b;c;]`).

More examples of Data

```
Margin: {(N maxX, N maxY, N minX, N minY)} 'just the fields
```

More examples of Data

```
Margin: {(N maxX, N maxY, N minX, N minY)} 'just the fields
```

```
Margin: Data[]<<{(N minX, N minY, N maxX, N maxY)}
 'automatically generated equals/hashcode, no validation
```

More examples of Data

```
Margin: {(N maxX, N maxY, N minX, N minY)} 'just the fields
```

```
Margin: Data[]<<{(N minX, N minY, N maxX, N maxY)}
 'automatically generated equals/hashcode, no validation
```

If `invariant` is provided, it will be dynamically called in the right moments.

```
Margin: Data[]<<{(N minX, N minY, N maxX, N maxY)
  method Void invariant() {
    return Assert[
      this.minX() >0N;
      this.minY() >0N message:S"y should be positive";
      this.maxX() >this.minX();
      this.maxY() >this.minY();  ]}}
```

More examples of Data

```
Margin: {(N maxX, N maxY, N minX, N minY)} 'just the fields
```

```
Margin: Data[]<<{(N minX, N minY, N maxX, N maxY)}
 'automatically generated equals/hashcode, no validation
```

If `invariant` is provided, it will be dynamically called in the right moments.

```
Margin: Data[]<<{(N minX, N minY, N maxX, N maxY)
  method Void invariant() {
    return Assert[
      this.minX() >0N;
      this.minY() >0N message:S"y should be positive";
      this.maxX() >this.minX();
      this.maxY() >this.minY();  ]}}
```

Now we can have points inside a margin.

```
Point: Data[]<<{(Margin that,  N x, N y)
  method Void invariant() {
    return Assert[   that.minX() <=this.x();  that.minY() <=this.y();
       that.maxX() >=this.x();  that.maxY() >=this.y();  ]}}
```

Active libraries and Decorators: What is new here?

- An active library is a library that have some way to generate code.
- A decorator is a special kind of active library, that takes a class and “improve it".
- If you have some former knowledge of generative programming, you will know that I’m not proposing anything new or groundbreaking.

Active libraries and Decorators: What is new here?

- An active library is a library that have some way to generate code.
- A decorator is a special kind of active library, that takes a class and “improve it".
- If you have some former knowledge of generative programming, you will know that I'm not proposing anything new or groundbreaking.
- What I'm proposing is to design a language around the idea that decorators are going to be there and massively used.

Active libraries and Decorators: What is new here?

- An active library is a library that have some way to generate code.
- A decorator is a special kind of active library, that takes a class and “improve it".
- If you have some former knowledge of generative programming, you will know that I'm not proposing anything new or groundbreaking.
- What I'm proposing is to design a language around the idea that decorators are going to be there and massively used.
- In the end, the programmers may change their perception about writing code: Instead of directly encoding the behaviour they need, they will give some suggestion to a decorator, that is going to do all the hard job.

Active libraries and Decorators: What is new here?

- An active library is a library that have some way to generate code.
- A decorator is a special kind of active library, that takes a class and “improve it".
- If you have some former knowledge of generative programming, you will know that I'm not proposing anything new or groundbreaking.
- What I'm proposing is to design a language around the idea that decorators are going to be there and massively used.
- In the end, the programmers may change their perception about writing code: Instead of directly encoding the behaviour they need, they will give some suggestion to a decorator, that is going to do all the hard job.
- This also allows for a much simpler language,requiring only a simple nominal (sub-)types and no inheritance.
  Code reuse (with inheritance as a special case), generics (as shown later), enumerations and many other concepts may be encoded as (active) libraries / decorators.

Getting started with Active Libraries

How an Active Library looks like?

Minimal Active Library

```
A:{()  'a class with a method m returning the empty class
  method Library m() {return {()}}
  }
B: A().m()  'an empty class generated using library A
```

rewrites to

```
A:{()  'a class with a method m returning the empty class
  method Library m() {return {()}}
  }
B:{()}  'the empty class
```

Incremental compilation

Like compile time execution in C++, template haskell and others. First class code literals, called `Librar`ies allows classes to be created, from top to bottom, using formerly defined classes. The program can perform side effects while creating classes. Execution is just the production of all the classes.

```
 '(STEP0)
A:{() 'a class with a method k producing an A
  method Library m() { 'm just returns a class with a method k
    return {() method A k(){return A()}}  ' returning an A instance
  }}
B:A().m()
C:B().k().m()
```

During execution becomes

```
 '(STEP1)
A:{() ..}  'as before
B:{() method A k(){return A()}}
C:B().k().m()
```

and finally

```
 '(STEP2)
A:{() ..}  'as before
B:{() method A k(){return A()}}
C:{() method A k(){return A()}}
```

that is the result of the execution of the original program.

Minimal building block

To manipulate libraries we do not use any templating mechanism. Libraries can be composed with other libraries using an algebra of composition operators /decorators. The main one is `Compose`, that decorates a class by summing to it a list of libraries. It is the only one that uses multiple libraries.

`Compose[a;b;c]<<d` composes `a b c d`. Abstract methods can be implemented, nested classes with the same name are recursively summed.

Minimal building block

To manipulate libraries we do not use any templating mechanism. Libraries can be composed with other libraries using an algebra of composition operators /decorators. The main one is **`Compose`**, that decorates a class by summing to it a list of libraries. It is the only one that uses multiple libraries.

`Compose[a;b;c]<<d` composes `a b c d`. Abstract methods can be implemented, nested classes with the same name are recursively summed.

On default the composition is symmetric, but **`Compose`** can take options to automatically solve conflicts (i.e. same method implemented in multiple classes). For example, a right preferential strategy.

Note that generated libraries are decorated copies of old ones. Pre-existing code is never modified.

Minimal building block

In addition to compose there are a plethora of refactoring operators, that takes a single library and adapt it somehow.

Minimal building block

In addition to compose there are a plethora of refactoring operators, that takes a single library and adapt it somehow.

- **`RenamePath`** and **`RenameSelector`** rename nested classes and methods

Minimal building block

In addition to compose there are a plethora of refactoring operators, that takes a single library and adapt it somehow.

- `RenamePath` and `RenameSelector` rename nested classes and methods
- `RemoveImplementationPath` make a nested class (and the whole subtree) fully abstract. (See also `RemoveImplementationSelector`)

Minimal building block

In addition to compose there are a plethora of refactoring operators, that takes a single library and adapt it somehow.

- `RenamePath` and `RenameSelector` rename nested classes and methods
- `RemoveImplementationPath` make a nested class (and the whole subtree) fully abstract. (See also `RemoveImplementationSelector`)
- `Redirect` delete a nested class and redirect all the reference to it to an external one.

Minimal building block

In addition to compose there are a plethora of refactoring operators, that takes a single library and adapt it somehow.

- `RenamePath` and `RenameSelector` rename nested classes and methods
- `RemoveImplementationPath` make a nested class (and the whole subtree) fully abstract. (See also `RemoveImplementationSelector`)
- `Redirect` delete a nested class and redirect all the reference to it to an external one.
- `Introspect` is a class allowing to query a library on its methods and nested classes names and structural shapes.

Minimal building block

In addition to compose there are a plethora of refactoring operators, that takes a single library and adapt it somehow.

- `RenamePath` and `RenameSelector` rename nested classes and methods
- `RemoveImplementationPath` make a nested class (and the whole subtree) fully abstract. (See also `RemoveImplementationSelector`)
- `Redirect` delete a nested class and redirect all the reference to it to an external one.
- `Introspect` is a class allowing to query a library on its methods and nested classes names and structural shapes.
- `MakePrivatePath` and `MakePrivateSelector` to tune privateness.

Minimal building block

In addition to compose there are a plethora of refactoring operators, that takes a single library and adapt it somehow.

- `RenamePath` and `RenameSelector` rename nested classes and methods
- `RemoveImplementationPath` make a nested class (and the whole subtree) fully abstract. (See also `RemoveImplementationSelector`)
- `Redirect` delete a nested class and redirect all the reference to it to an external one.
- `Introspect` is a class allowing to query a library on its methods and nested classes names and structural shapes.
- `MakePrivatePath` and `MakePrivateSelector` to tune privateness.
- `AddDocumentationPath`, `AddDocumentationSelector`, `RemoveDocumentationPath`, `RemoveDocumentationSelector` support generation of documentation with metaprogramming.

Collection generation / not with generics

```
NVector: Collections.vector(N)

PointSet:Collections.set(Point)

StudentList:Collections.orderedList(Student, orderedBy:{
  method Boolean(Student a, Student b) {return a.name>b.name}
  })
```

Collections are generated on demand one for each different type. Multiple incompatible collections can be generated for the same type, in order to underline a different role for different collections.

Collection generation / not with generics

Thanks to **Refactor::Redirect** we can emulate generics, just redirect a class to the final destination. Internally, the code of **Collections**.vector may look like the following

```
Collection:{...
  Library vector(type Any that){
    return Refactor::Redirect[Path"T" to: that ]<<{()
      T:{}
      method Void add(T that){...}
     ...}
    }
  }
```

And the result of **NVec:Collections**.vector**(N)** could be:

```
 'NVec:Collections.vector(N)
NVec:{()
  method Void add(N that){...}
  ...}
```

Note how in this way we only need a simple nominal type system.

Pseudo higher order functions

In the same way, it would be possible to have maps/filters/folds as in functional languages, without the need of polymorphic types

```
myStudents= PersonList[....]
FindOlder: Accumulate[PersonList]<<{
  method Person accumulate (Person left, Person right){
    if left.age()>right.age() ( return left )
    return right
  }}
older= FindOlder.from(myStudents)
 'haskell-like equivalent
 'older=accumulate (left,right->if left.age()>right.age() left else right) myStudents
```

While do not look as nice as an haskell-like equivalent, our approach works with a simple pure nominal type system.

Pseudo higher order functions

In the same way, it would be possible to have maps/filters/folds as in functional languages, without the need of polymorphic types

```
myStudents= PersonList[....]
FindOlder: Accumulate[PersonList]<<{
  method Person accumulate (Person left, Person right){
    if left.age()>right.age() ( return left )
    return right
  }}
older= FindOlder.from(myStudents)
 'haskell-like equivalent
 'older=accumulate (left,right->if left.age()>right.age() left else right) myStudents
```

While do not look as nice as an haskell-like equivalent, our approach works with a simple pure nominal type system.

Question:

Is it easier to reason about decorators as producing code taking some hint in input, or is it easier to reason about higher order polymorphic function composition?

Compose

`Compose` decorates a class by summing to it a list of libraries. Since there is no “extends”, code reuse is obtained with decorators. For example, to define a gui with callbacks, we can have a html form where buttons have ids, and we can implement the onclick method:

```
Gui:Compose[GuiFromHtml(myHtml)]<<{
  ButtonStart:{ method Void onClick(Gui g){
    g.resultText().value(g.inputText().value())   }}
```

`GuiFromHtml` have a set of conventions: generate nested classes for elements with associated events, and getters for elements with an `id` attribute.
The programmer can/must rely on those conventions!
As an alternative design, `GuiFromHtml` could be a decorator;
this it could examine the input code and check conventions are respected.

```
Gui:GuiFromHtml[myHtml]<<{
  ButtStart:{ method Void onClick(Gui g){ 'error is discovered
    g.resultText().value(g.inputText().value())   }}
```

Resolver

The need of super is satisfied with resolvers. In the following code

```
Compose[
  { method Bool a(){return True()} };
  { method Bool a(){return False()} };
  resolver:{
    Bool a(){return this.left() & this.right()}
    Bool left()
    Bool right()
    };
  ]
```

The `Compose` instance will compose the two classes successfully and the resulting method `a()` will perform the logic and of the result of the two methods.

How Data could work? - Example: equals on fields

Example of input and expected result

```
Point:Data[]<<{(N x,N y)} 'example just on ==
```

reduces to

```
Point:{(N x,N y)
  method Bool == (Any that){
    with that (
      on Outer0  return this.internEquals(that)
      default    return false
      )
    }

  method  '@private
  Bool internEquals(Outer0 that){
    return this.x() ==that.x() & this.y() ==that.y()
    }
  }
```

How Data could work? - Example: equals on fields

```
method Library equal(type Any class,Selector name) { 'equal on single field
  return Refactor::Redirect[Path"T" to: class]<<
         Refactor::RenameSelector[Selector"f()" to:name]<<{
    T:{ method Bool ==(Any that)}
    method T f()
    method Bool internEquals(Outer0 that) {return this.f()==that.f()}
    }
  }
```

How Data could work? - Example: equals on fields

```
method Library equal(type Any class,Selector name) { 'equal on single field
  return Refactor::Redirect[Path"T" to: class]<<
         Refactor::RenameSelector[Selector"f()" to:name]<<{
    T:{ method Bool ==(Any that)}
    method T f()
    method Bool internEquals(Outer0 that) {return this.f()==that.f()}
    }
  }
method Library addEquals(Library l){ 'produces a class with == and internEquals
```

How Data could work? - Example: equals on fields

```
method Library equal(type Any class,Selector name) { 'equal on single field
  return Refactor::Redirect[Path"T" to: class]<<
         Refactor::RenameSelector[Selector"f()" to:name]<<{
    T:{ method Bool ==(Any that)}
    method T f()
    method Bool internEquals(Outer0 that) {return this.f()==that.f()}
    }
  }
method Library addEquals(Library l){ 'produces a class with == and internEquals
  var composer=Compose[resolver:{ ' how to merge two equal on single field
    method Bool internEquals(Outer0 that) {
      return this.left(that) & this.right(that) }
    method Bool left(Outer0 that)
    method Bool right(Outer0 that)    }]
```

How Data could work? - Example: equals on fields

```
method Library equal(type Any class,Selector name) { 'equal on single field
  return Refactor::Redirect[Path"T" to: class]<<
         Refactor::RenameSelector[Selector"f()" to:name]<<{
    T:{ method Bool ==(Any that)}
    method T f()
    method Bool internEquals(Outer0 that) {return this.f()==that.f()}
    }
  }
method Library addEquals(Library l){ 'produces a class with == and internEquals
  var composer=Compose[resolver:{ ' how to merge two equal on single field
    method Bool internEquals(Outer0 that) {
      return this.left(that) & this.right(that) }
    method Bool left(Outer0 that)
    method Bool right(Outer0 that)    }]
  with field in Introspection.fieldsOf(l).vals() ( 'accumulate equalities
    composer:=composer.add(this.equal(class:field.class(),name:field.name())))
```

How Data could work? - Example: equals on fields

```
method Library equal(type Any class,Selector name) { 'equal on single field
  return Refactor::Redirect[Path"T" to: class]<<
        Refactor::RenameSelector[Selector"f()" to:name]<<{
    T:{ method Bool ==(Any that)}
    method T f()
    method Bool internEquals(Outer0 that) {return this.f()==that.f()}
    }
  }
method Library addEquals(Library l){ 'produces a class with == and internEquals
  var composer=Compose[resolver:{ ' how to merge two equal on single field
    method Bool internEquals(Outer0 that) {
      return this.left(that) & this.right(that) }
    method Bool left(Outer0 that)
    method Bool right(Outer0 that)    }]
  with field in Introspection.fieldsOf(l).vals() ( 'accumulate equalities
    composer:=composer.add(this.equal(class:field.class(),name:field.name())))
  result=composer<<{ 'compute internEquals and add the == method
    method Bool internEquals(Outer0 that)
    method Bool ==(Any that){
      with that (
        on Outer0  return this.internEquals(that)
        default    return false
        ) }    }
```

How Data could work? - Example: equals on fields

```
method Library equal(type Any class,Selector name) { 'equal on single field
  return Refactor::Redirect[Path"T" to: class]<<
         Refactor::RenameSelector[Selector"f()" to:name]<<{
    T:{ method Bool ==(Any that)}
    method T f()
    method Bool internEquals(Outer0 that) {return this.f()==that.f()}
    }
  }
method Library addEquals(Library l){ 'produces a class with == and internEquals
  var composer=Compose[resolver:{ ' how to merge two equal on single field
    method Bool internEquals(Outer0 that) {
      return this.left(that) & this.right(that) }
    method Bool left(Outer0 that)
    method Bool right(Outer0 that)    }]
  with field in Introspection.fieldsOf(l).vals() ( 'accumulate equalities
    composer:=composer.add(this.equal(class:field.class(),name:field.name())))
  result=composer<<{ 'compute internEquals and add the == method
    method Bool internEquals(Outer0 that)
    method Bool ==(Any that){
      with that (
        on Outer0  return this.internEquals(that)
        default    return false
        ) }   }
  return Refactor::MakePrivateSelector[Selector"internEquals(that)"]<<result}
```

Just a different way to generate code

Operations are defined by inductively generated methods:

Just a different way to generate code

Operations are defined by inductively generated methods:

- define a behaviour for base case ("single field" in the example)

Just a different way to generate code

Operations are defined by inductively generated methods:

- define a behaviour for base case (“single field” in the example)
- define composition of behaviours (“logic and” in the example)

Just a different way to generate code

Operations are defined by inductively generated methods:

- define a behaviour for base case (“single field” in the example)
- define composition of behaviours (“logic and” in the example)
- refine the result depending of what you are doing (“result” in the example)

Just a different way to generate code

Operations are defined by inductively generated methods:

- define a behaviour for base case (“single field” in the example)
- define composition of behaviours (“logic and” in the example)
- refine the result depending of what you are doing (“result” in the example)

At any step is possible to check what we are creating and provide good error messages if something is going wrong.

Any operation can be synthesized as a library that would implement the operation on the original input when summed (Compose).

Decorators instead of Quote/Unquote DSL

What is a Quote/Unquote DSL (also known as a Templating DSL)? Main way to do metaprogramming. Example, Java Mint

```
Code<String> msg=..;
Code<Void> myCode=<|
{
  if(Foo.bar){ Foo.doBla();}
  else { `(msg) }
}
|>;
```

Special parenthesis `<|  |>` denotes first class code literals, with holes, denoted by the symbol `` `() ``. The holes can be filled with arbitrary expressions of `Code` type. This is just a layer of syntactic sugar over regular Meta Object Protocol(MOP), that is basically an object oriented data-structure representing an AST for the language under consideration. Depending on the specific language, it is possible to guarantee a range of properties on the resulting code.

Decorators instead of Quote/Unquote DSL

Templating DSL: a low level mechanism for metaprogramming.
Focus is placed on the code and on the concrete implementation.
Since every kind of code can have holes, there is an enormous error space.

Decorators instead of Quote/Unquote DSL

Templating DSL: a low level mechanism for metaprogramming.
Focus is placed on the code and on the concrete implementation.
Since every kind of code can have holes, there is an enormous error space.

In my opinion Templating DSLs are a fast and dirty way to obtain good initial results using metaprogramming.
The technique does not scale and is very low level.

Decorators instead of Quote/Unquote DSL

Templating DSL: a low level mechanism for metaprogramming.
Focus is placed on the code and on the concrete implementation.
Since every kind of code can have holes, there is an enormous error space.

In my opinion Templating DSLs are a fast and dirty
way to obtain good initial results using metaprogramming.
The technique does not scale and is very low level.
L42 does not rely on Templating DSL.
We focus on behaviour composition instead of “source code” composition, and we take inspiration from class/module composition languages.

Today, a goal for tomorrow

The shift of mind that I'm suggesting here is similar to the one from imperative to object oriented programming.

- With OO, thanks to dynamic dispatch, the programmer do not chose explicitly what behaviour to invoke when a method is called. (as in `a.draw(b)`)
  However, the programmers are still in control, since they are instantiating and passing around such objects.

Today, a goal for tomorrow

The shift of mind that I'm suggesting here is similar to the one from imperative to object oriented programming.

- With OO, thanks to dynamic dispatch, the programmer do not chose explicitly what behaviour to invoke when a method is called. (as in `a.draw(b)`)
  However, the programmers are still in control, since they are instantiating and passing around such objects.
- With class decorators, the programmers do not chose explicitly what methods are in each class.
  (as in `Person:Data[]<<...`)
  However, the programmers are still in control, since they writing and applying such decorator objects.

Today, a goal for tomorrow

The shift of mind that I'm suggesting here is similar to the one from imperative to object oriented programming.

- With OO, thanks to dynamic dispatch, the programmer do not chose explicitly what behaviour to invoke when a method is called. (as in `a.draw(b)`)
  However, the programmers are still in control, since they are instantiating and passing around such objects.
- With class decorators, the programmers do not chose explicitly what methods are in each class.
  (as in `Person:Data[]<<...`)
  However, the programmers are still in control, since they writing and applying such decorator objects.

The difference is in the which programmer keeps the control: In both cases we have a shift: the programmers of the final application can delegate more and more control to *the programmers of the libraries*.

Can you guess what this code is doing?

```
{reuse  L42.is/AdamsTowel
DB: Load[]<<{reuse L42.is/DB}
Gui: Load[]<<{reuse fL42.is/2dGui}
Tables: DB(S"...my connection string...")
StudentList: Collections.list(Tables::Student)
StudentsView: Gui.tableOf(StudentList)
QueryFrom: Tables::Query[result:StudentList](
  DB::SQL"Select * from Student where country=@country")
Main:{
  connection=Tables.connect()
  fromItaly=QueryFrom(connection, country:S"Italy")
  Gui.show(StudentsView(fromItaly))
  return ExitCode.normal()
  }
}
```

Thanks!

Questions?