Script generated by TTT

Title: Petter: Programmiersprachen (17.12.2014)

Date: Wed Dec 17 15:12:31 CET 2014

Duration: 35:32 min

Pages: 12

Can we augment classical languages by traits?

Extension Methods (C#)

Central Idea:

Uncouple method definitions from class bodies.

Purpose:

- retrospectively add methods to complex types
  ⇝ *external definition*
- especially provide definitions of *interface methods*
  ⇝ poor man's multiple inheritance!

Syntax:

1. Declare a static class with definitions of static methods
2. Explicitely declare first parameter as receiver with modifier `this`
3. Import the carrier class into scope (if needed)
4. Call extension method in *infix form* with emphasis on the receiver


```
public class Person{
 public int size = 160;
 public bool hasKey() { return true;}
}
public interface Short {}
public interface Locked {}
public static class DoorExtensions {
 public static bool canOpen(this Locked leftHand, Person p){
  return p.hasKey();
 }
 public static bool canPass(this Short leftHand, Person p){
  return p.size<160;
 }
}
public class ShortLockedDoor : Locked,Short {
 public static void Main() {
  ShortLockedDoor d = new ShortLockedDoor();
  Console.WriteLine(d.canOpen(new Person()));
 }
}
```

Extension Methods as Traits

Extension Methods

- transparently extend arbitrary types externally
- provide quick relief for plagued programmers

. . . but not traits

- Interface declarations empty, thus kind of purposeless
- Flattening not implemented
- Static scope only

Static scope of extension methods causes unexpected errors:

```
public interface Locked {
  public bool canOpen(Person p);
}
public static class DoorExtensions {
 public static bool canOpen(this Locked leftHand, Person p){
  return p.hasKey();
 }
}
```


Virtual Extension Methods (Java 8)

Java 8 advances one step further:

```
interface Door {
  boolean canOpen(Person p);
  boolean canPass(Person p);
}
interface Locked {
  default boolean canOpen(Person p) { return p.hasKey(); }
}
interface Short {
  default boolean canPass(Person p) { return p.size<160; }
}
public class ShortLockedDoor implements Short, Locked, Door {
}
```

{ return Locked.superCanOpen }

Implementation

. . . consists in adding an interface phase to `invokevirtual`'s name resolution

⚠ Precedence

Still, default methods do not overwrite methods from *abstract classes* when composed

Traits as General Composition Mechanism

⚠ Central Idea

Separate class generation from hierarchy specification and functional modelling

1. model hierarchical relations with interfaces
2. compose functionality with traits
3. adapt functionality to interfaces and add state via glue code in classes

Simplified multiple Inheritance without adverse effects

So let's do the language with real traits?!

Squeak

Smalltalk

Squeak is a smalltalk implementation, extended with a system for traits.

Syntax:

- `name: param and: param2`
  declares method `name` with `param1` and `param2`
- `| ident1 ident2 |`
  declares Variables `ident1` and `ident2`
- `ident := expr`
  assignment
- `object name:content`
  sends message `name` with `content` to `object`
- `.`
  line terminator
- `^ expr`
  return statement

Traits in Squeak

```
Trait named: #TRStream uses: TPositionableStream
  on: aCollection
    self collection: aCollection.
    self setToStart.
  next
    ^ self atEnd
      ifTrue:  [nil]
      ifFalse: [self collection at: self nextPosition].
Trait named: #TSynch uses: {}
  acquireLock
    self semaphore wait.
  releaseLock
    self semaphore signal.

Trait named: #TSyncRStream uses: TSynch+(TRStream@(#readNext -> #next))
  next
    | read |
    self acquireLock.
    read := self readNext.
    self releaseLock.
    ^ read.
```


Traits: So far so...

✓ good

- Principles fully implemented
- Concept has encouraged mainstream languages to adopt ideas

⚠ bad

- One very unconventional graphical IDE for Squeak, afaik
- ...and there is no separate compiler with command line mode!

Lessons learned

Lessons Learned

- Single inheritance, multiple Inheritance and Mixins leave room for improvement for modularity in real world situations
- Traits offer *fine-grained control* of composition of functionality
- Native trait languages offer *separation of composition* of functionality from *specification* of interfaces
- Practically no language offers full traits in a usable manner

Further reading...

- Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black.
  Traits: A mechanism for fine-grained reuse.
  ACM Transactions on Programming Languages and Systems (TOPLAS), 2006.
- Brian Goetz.
  Interface evolution via virtual extension methods.
  JSR 335: Lambda Expressions for the Java Programming Language, 2011.
- Anders Hejlsberg, Scott Wiltamuth, and Peter Golde.
  C# Language Specification.
  Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 0321154916.
- Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black.
  Traits: Composable units of behaviour.
  European Conference on Object-Oriented Programming (ECOOP), 2003.