Traits in C#

Stefan Reichhart
stefan_reichhart@student.unibe.ch

Oscar Nierstrasz, Stephane Ducasse
Roadmap

- Project Context / Why Traits?

- What’s a Trait? Flattening Traits?

- Traits in C# and STOO (statically typed object oriented) Languages
Why Traits?

Problem / Goal:
avoid code duplication and fragile compositions
share code easily

Current Reuse-Mechanisms:
Single Inheritance ? -- too limited
Multiple Inheritance ?
Mixins ?
} good, but too complex
fragile composition, “ripple effect”

Solution: Traits -- simple, efficient, cool!
What is a “Trait”? 

◆ **first-class** compose-time
  group of pure methods

**Properties:**

◆ **stateless, pure behavior**

◆ **provides** — & **requires** ← a set of methods

---

<table>
<thead>
<tr>
<th>TCircle</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>area</td>
<td>radius</td>
</tr>
<tr>
<td>bounds</td>
<td>radius:</td>
</tr>
<tr>
<td>diameter</td>
<td>center</td>
</tr>
<tr>
<td>hash</td>
<td>center:</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Traits & Composition (1)

- *complements* single inheritance
- composition order is *irrelevant*
- *compose time* entity
- composite entity is in *full control of the composition*

Class = Superclass + State + Traits + Glue methods
Traits & Composition (2)

- class methods >>> trait methods

- *no change* in overriding!

- *self/this, super/base* have the *same meaning as before*

= consequences by the *flattening property*
Flattening Traits

Structured view

```
Object

ColoredCircle
draw

TCircle
draw
radius

TColor
hue
rgb
```

equivalent

Flat view

```
Object

ColoredCircle
draw
radius

hue
rgb
```
explicit conflict resolution by the composite entity

**Alias** -> and **Exclusion**  ^
Traits & Conflicts (2)

... one possible solution: **aliasing**

```
MyCircle = TColor = TCircle = ColorEquals -> CircleEquals
```

```
= OK
```

```
TCircle =
```

```
MyCircle
=
```

```
= OK
```

```
ColorEquals
```

```
CircleEquals
```
class Circle {
    uses {
        TColor { = -> ColorEquals; };
        TCircle { = -> CircleEquals; };
    }
    public boolean operator =(Circle c) {
        return this.ColorEquals(c) && this.CircleEquals(c);
    }
}

trait TColor { ... }
trait TCircle { ... }
Traits & Conflicts (4)

... an alternative? **exclusion**

\[
\begin{align*}
\text{MyCircle} & \quad ^{\wedge} = \quad \text{OK} \\
\text{TColor} & = \\
\text{TCircle} & = 
\end{align*}
\]
class Circle {
    uses {
        TColor { ^=; }; 
        TCircle;
    }
}

trait TColor { ... }
trait TCircle { ... }
Traits

- available in *dynamically typed oo languages*
  
  ➔ Smalltalk, Slate, Perl 6 (role)
  
  **experiences:** *simple* and *expressive*

- what about *statically typed oo languages*, e.g. C# ???
  
  ➔ Scala, Minijava
Traits in C#: Overview

Trait Container and Definition

Typed Syntax/Declaration

3 Kinds of Typed Traits in STOO languages

Generics and Generic Constraints

Typing Traits and Type Problems

Respect modifiers and keywords

Handling of libraries / packages

Trait Interfaces
Typed Syntax (I)

Typed Trait Declaration:
- typed alias, exclusion and requirements

Solution (conceptually simple):
- alias, exclusion → argument types
- requirements → argument (and return) types
Object subclass: #MyCircle
instanceVariableNames: 'color'
uses {
    TColor @{ #invertAliased -> #invert }.
    TShape -{ #resize: }.
}
MyCircle>> color
    ^ color

Trait named: #TColor
uses {}
requires { #color }
MyColor>> invert
    ^ ... self color ...

public class MyCircle {
    uses {
        TColor { invert() -> invertAliased; };
        TShape { ^ resize(int); };
    }
    private IColor color;
    public IColor Color() {
        return this.color;
    }
}

trait TColor {
    requires {
        IColor Color();
    }
    public IColor Invert() {
        ... return this.Color();
    }
}
Typing Traits (1)

Traits
- code-reuse
- share-ability

Goal
- simplicity
- flexibility

Types
- argument
- return
- temporary
Typing Traits (2)

Typed Trait methods

- what *argument* types?
- what types should be *returned* by methods?
- how to return the *type of the class* using the trait?
- *temporary* types?

Traits

- code-reuse
- share-ability
- simplicity
- flexibility
Typing Traits (3)

class MyCollection { uses {TSequenceable;} }

trait TSequenceable {
    public ??? Reverse() {
        ??? reversedList = new ???();
        ...
        return reversedList;
    }
    public void Concat(??? c1, ??? c2) {...}
}
Typing Traits (4)

Possible Solutions:

- Simple/Concrete types
- Interfaces
- Explicit / Parameterized
- Implicit / Type Keyword
- Implicit Types & Generics
- ...
Explicit / Parameterized

class MyCollection {
    uses { TSequenceable<MyCollection>; } OR
    uses { TSequenceable<ICollection>; }
}

trait TSequenceable<A> {
    public A Reverse() { ... }
    public void Concat(A.. c1, A.. c2) { ... }
}
Explicit / Parameterized

**Advantage:**

- *simple, extremely flexible*
- *template-like* use, on top of any language

**Disadvantage:**

- multiple/variable use of the type parameter
- *inconsistent syntax/use* when generics are used
Implicit Types & Generics

class MyCollection\<T> \{ 
    uses { TSequenceable\<T>; } ... 
\}

trait TSequenceable\<A> \{ 
    public TSequenceable Reverse() {...} 
    public void Concat(TSeq.. c1, TSeq.. c2) 
    {...} 
\}

Implicit Types & Generics

**Advantage:**
- *simple*
- *consistent syntax* for generics and non-generics

**Disadvantage:**
- *limited flexibility* for return types
- *not suitable for any use*
Traits in C# : Overview

Trait **Container** and **Definition**

Typed **Syntax/Declaration**

3 Kinds of Typed Traits in STOO languages

**Generics** and Generic Constraints

**Typing Traits** and **Type Problems**

Respect **modifiers** and **keywords**

Handling of **libraries / packages**

Trait **Interfaces**
Prototype Implementation (1)

- Preprocessor based on the *flattening property*
- *Simple Typed* and *Generic Traits*  
  (no var-binding, strict-matchings only)
- Simple and minimal *Typed-Trait-syntax* for declarations and requirements

... ...
Prototype Implementation (2)

- *Generic parser* (most C-like languages supported)
- Simple CodeDOM framework
- Language-*independant flattening logic*

- ....
Future for Traits in C# / STOO

- Finding a *satisfying solution* for *modifier/keyword* and *typing problems*
- Combining Template-like & Generic Traits (?)
- Requirement constraints for methods (?)
- .......

- **Clean Implementation/Integration in**
  *Rotor/*.NET*

S. Ducasse, N. Schärli, O. Nierstrasz, R. Wuyts, and A. Black. \textbf{Traits: A mechanism for fine-grained reuse}. Transactions on Programming Languages and Systems, 2005. under revision.
References (2)

N. Schärli.

O. Nierstrasz, S. Ducasse, and N. Schärli.

S. Reichhart
**Traits in C#,** Technical Report and prototype implementation, University of Bern, Switzerland, Sept. 2005
References (3)

Software Composition Group

Webpage
http://www.iam.unibe.ch/~scg/Research/Traits/index.html
Traits

- *simple* but *effective reuse-mechanism*

- *no “ripple-effects”*

- *no change* to the existing composition model (single inheritance, overriding, ...)

- *fully-controlled composition*
Traits in C# : Overview

Trait *Container* and *Definition*

Typed *Syntax/Declaration*

3 Kinds of Typed Traits in STOO languages

*Generics* and Generic Constraints

Typing Traits and Type Problems ~OK

Respect modifiers and keywords

Handling of libraries / packages

Trait *Interfaces* .... ....
3 Kinds Of Traits

- Simple Traits
- Template-like traits
- (full) Generic Traits
Simple Traits

- uses *identities* of *concrete argument* and *return types*
- *type-problems* not handled
- very simple, but *limited* code reuse

```java
class MyCollection {
    uses { TSequenceable; } ...
}

trait TSequenceable {
    public *ICollection* Reverse() { ... }  
    public void Concat(*IC.. c1, *IC.. c1) { ... }
}
```
Template-like Traits (1)

- not like C++ templates!
- simple, variable, *flexible type parameter*
- *type-problems* might be handled
**Template-like Traits (2)**

```java
class MyCollection {
    uses { TSequenceable<ICollection>; } ...
}

trait TSequenceable<A> {
    public A Reverse() {...}
    public void Concat(A c1, A c1) {...}
}

or like this ...

class MyCollection {
    uses { TSequenceable<MyCollection>; } ...
}
```
Generic Traits (1)

- parametric Polymorphism, *flexible* code reuse
- variable binding, a bit *more complex* ...

```
Class_1 <T>
A  -->  T
Class_N <U>
T1 <A>
A  -->  U
Class_1 <T>
Class_N <U>
T1 <T>
T1 <U>
T1 <A>
```
Generic Traits (2)

Class_1 <T>

T1 <A,B>

A --> T
B --> B

no free variables

Class_1 <T>

T1 <T,B>

no free variables
Generic Traits (3)

Class_1 <T>
T1 <A,B>
Class_1 <T,B>
T1 <T,B>
T1 <A,B>

no state change

A --> T
B --> B

no state change
Generic Traits (4)

\[ \text{genericTypeParameters(class)} \geq \sum (\text{genericTypeParameters(Ti)}) \]

*not more or no ‘different’ generic type parameters*
class MyCollection\<T>\> {  
    uses { TSequenceable\<T\>; } ...
}

trait TSequenceable\<A\> {  
    public ??? Reverse() {...}
    public void Concat(??? c1, ??? c1) {...}
}
Modifiers / Keywords (C# only)

**Trait Properties:**
trait methods may want/need to override/hide methods of higher levels (classes or traits)

**C#’s explicitly:**
all methods must declare correct inheritance (and accessibility modifiers)
virtual, override, new, public, ...
Problem & Disadvantage:

- code sharing is *limited*
- might lead to *code duplication* (again)
Modifiers / Keywords (3)

Solution Basics:
tests should *catch conflicts* when processing traits
but: not a real solution to the problem ...

Possible solutions:
- Explicit modifiers for trait declaration ?
- Implicit resolution ?
- Avoid overriding traits ?
- ?
Library / Package propagation

**Situation**
Traits may also *depend on libraries*

**Solution**
*implicit propagation* of libraries to the higher level

```
MyCircle
using System;
TColor
using System.Graphics;
MyCircle
using System;
using System.Graphics;
```
Trait Interfaces (1)

Situation:
Traits may also implement interfaces

```scala
class MyShape { uses {TColor; } ... }
trait TColor : IColorable { ... }
```

Solution:
*implicit propagation* of interfaces to the higher levels

```scala
class MyShape : IColorable { ... }
```
Trait Interfaces (2)

**Advantage:**

*no explicit interface declaration* / maintainance

**!!! Special Case:** Interfaces used with Exclusion
Confusing but correct, no breaking of the interface!
Trait Interfaces (3)
Generic Constraints (C# only)

**Situation**

Traits may also *use constraints* on generic type parameters

```csharp
trait TSequenceable<T>
    where T : INumber

    {...}
```
Generic Constraints (2)

Solution
Test if constraints collide with constraints on higher levels

\[ \text{constraints}(\text{class}) \geq \sum (\text{constraints}(\text{Ti})) \]

not more, not more ‘restrictive’ or different constraints
Generic Constraints (3)

class MyRational<T> where T : IInteger {...}
trait MyAlgebra<A> where A : IFloatingPoint {...}

class MyDictionary<T> where T : IAssociation {...}
trait MySequence<A> where A : INumber {...}
Generic Constraints (4)

class MyDictionary<T> where T : IAssociation {...}

trait MySequence<A> where A : IAssociation, ILockable {...}

class MyRational<T> where T : IFloatingPoint {...}

trait MyAlgebra<A> where A : IInteger {...}
Generic Constraints (5)

Disadvantage:

- Constraints may prevent the ability to share behavior easily
- Use is questionable (?)
Traits Preprocessor

- Interface
- Parser Framework
- Environment
- Trait Flattening Logic
Parser Framework

Simple Parser
Structured Code
Unstructured Code

Type Parser
Class
Interface
...

Type Member Parser
Method
Property
...

Statement Parser
Library / Package
Variable Init
...

Structured Code
...... { ... }

Unstructured Code
...... ;
Environment (singleton)

- InternalEnvironment
- ExternalEnvironment

- code definitions within the ST Image
- directory on disc

...
Trait Flattening Logic (1)

MyClass

T1

T3

T4

T5

T6

T7

T8

MyClass

T1

T3

T4, T6, T7

T5, T7, T8

MyClass

T1

T3 - 8

...
Trait Flattening Logic (2)

TraitFlattenVisitor

xCompilationUnit

lib

lib

class

uses trait

constructor

m1

m2

class

...