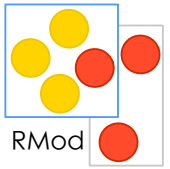


Traits in Perspective

Stéphane Ducasse
stephane.ducasse@inria.fr
http://stephane.ducasse.free.fr/

Our expertise

Supporting software evolution and software composition

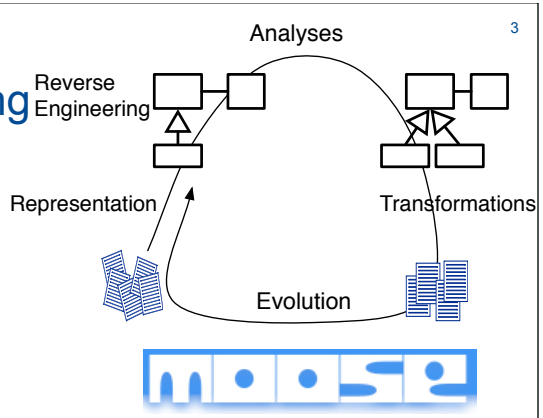
Axis 1: Reengineering

Maintaining large software systems
Moose: a platform for reengineering
Nokia, Daimler, Harman-Becker

Axis 2: Dynamic languages to support evolution

How to support reuse?
Revisiting fundamental aspects of OO languages
Traits (SUN...), Classboxes

Axis 1: Reengineering



Topics

Metamodeling,
Software metrics,
Program understanding,
Visualization, evolution analysis,
Duplicated code detection,
Code analysis, refactorings...

Contributions

Moose: an open-source language independent extensible reengineering environment: (Lugano, Bern, Annecy,Louvain la neuve, ULB, UTSL)

Contacts

Harman-Becker (3 Millions C++), Bedag (Cobol), Nokia, ABB, IMEC

Axis 2: Dynamic Languages Infrastructure

La perfection est atteinte, non pas lorsqu'il n'y a plus rien à ajouter, mais lorsqu'il n'y a plus rien à retirer. St-Exupery

Topics

Components for field devices (Pecos IST Project)
Classboxes: Modules for open-classes [OOPSLA'05]
OOPAL: OOP + APL Generalizing message passing [OOPSLA'03]
Language symbiosis (Jour. Program)
Encapsulation for dynamic languages [ECOOP '04, OOPSLA'04]
Reusable behavior: Traits [ECOOP'03, OOPSLA'03, Toplas, ..., OOPSLA'07, TOOLS'09]

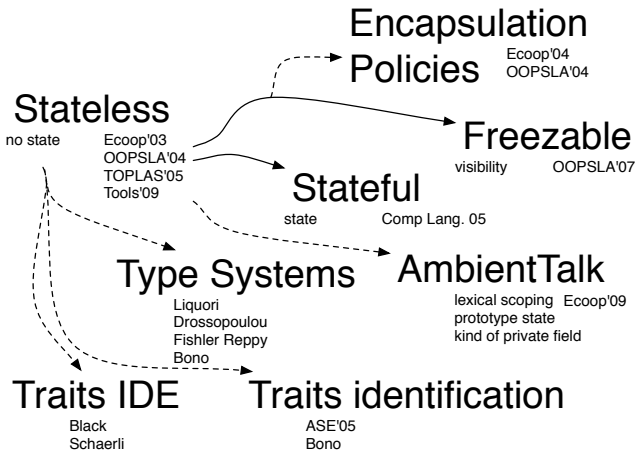
Impacts

Traits used by Fortress (SUN Microsystems), Scala (EPFL), Perl-6, Squeak, Slate, Dr-Scheme, Multiple type systems (Drossopoulos, Reppy, Liquori, Bono...)

Road Map

- Motivation
- Traits - Fine grained structural composition
- Results
- Stateful Traits - state into traits?
- Freezable Traits - full method conflict
- Conclusion

Towards Traits Only?



Contributors

N. Schaerli

A. Black

O. Nierstrasz

S. Ducasse

C. Delaunay

A. Lienhard

A. Bergel

R. Wuyts

D. Cassou

D. Pollet

D. Rothlisberger

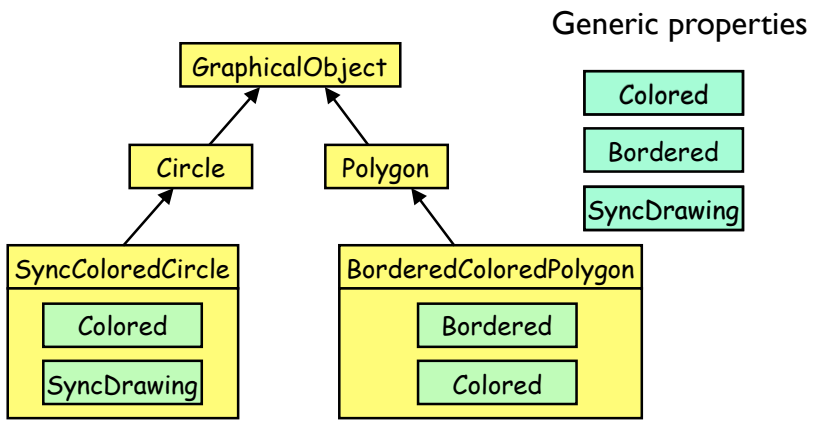
L. Bettini

F. Damiani

T. van Cutsem

V. Bono

Building Classes out of Components...



Are classes schizo...?

Unit of Creation.
Should a complete to be able to create instances

Unit of Reuse.
Should be partially defined/"abstract" to be able to be reused

Traditional Inheritance Limits

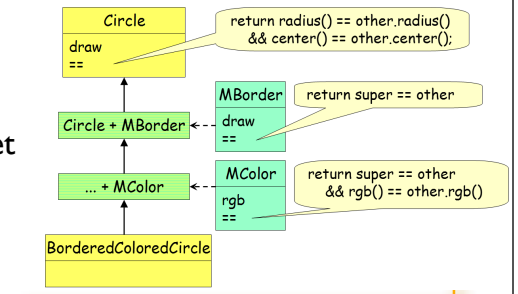
Single inheritance limited
- copy and paste
- tyranny of the dominant abstractions

Multiple inheritance
- Diamond
- Hardcoded superclass refs
- Linearization limits

Mixin

Parametrized by their superclass class
Do not have most MI problems
But
- Implicit conflict resolution
- Composite entity is not in full control
- Dispersal of glue code

BorderedColoredCircle
cannot decide which == to get



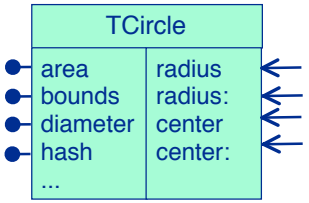
Trait Goals and Design Principles

Goals
- Improve code reuse
- Improve class composition
- Maintain understandability
- No surprises
- Avoid fragile hierarchies

Design Principles
- *Simplicity*
- *Empowering* the composer
- *Compile-time* entities, no run-time costs

What are Traits?

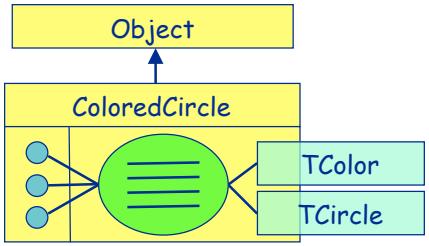
Traits are *parameterized* behaviors
Traits *provide* a set of methods
Traits *require* a set of methods
Traits are purely behavioral (Traits do not specify any state)



Composing Classes out of Traits

Traits are the behavioral building blocks of classes

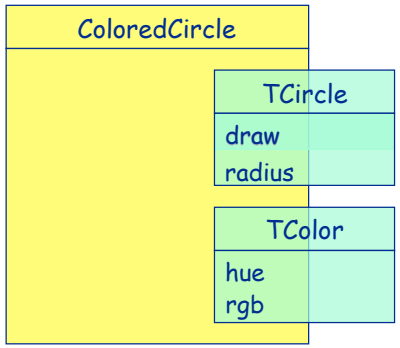
Class = Superclass + State + Traits + Glue Methods

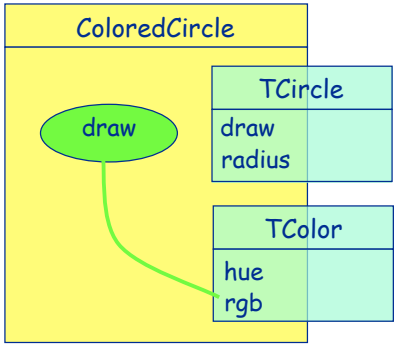


The *composing* class retains control of the composition

Composition Rules

Class methods take precedence over trait methods





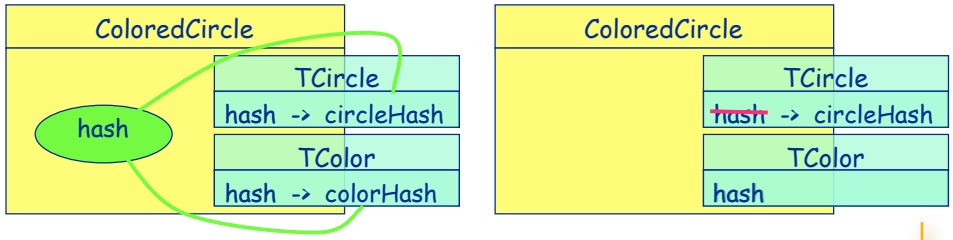
Conflicts are *explicitly* resolved

Override the conflict with a glue method

- Aliases provide access to the conflicting methods

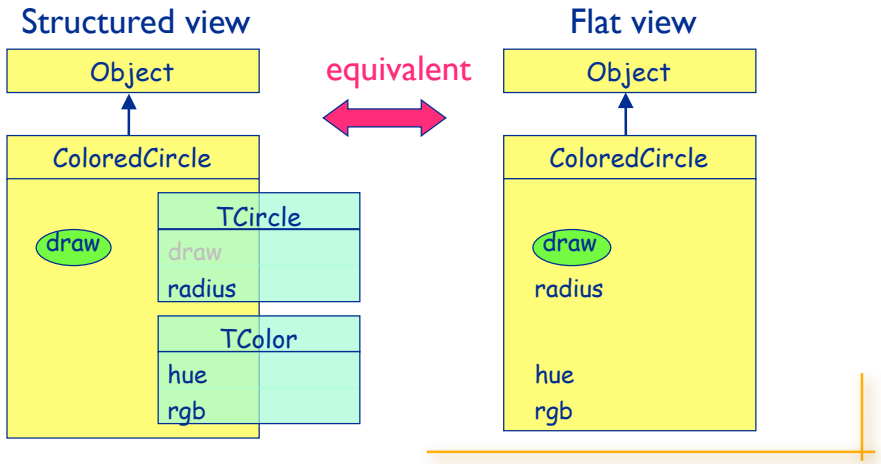
Avoid the conflict

- Exclude the conflicting method from one trait



Flattening Property

The semantics of a method does not depend of whether it is defined in a trait or in a class that uses the trait

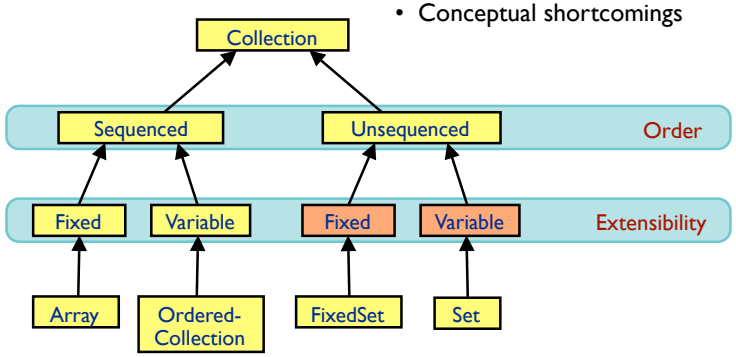


Road Map

Motivation
Traits - Fine grained structural composition
Results
Stateful Traits - state into traits?
Freezable Traits - full method conflict
Conclusion

Smalltalk Collections

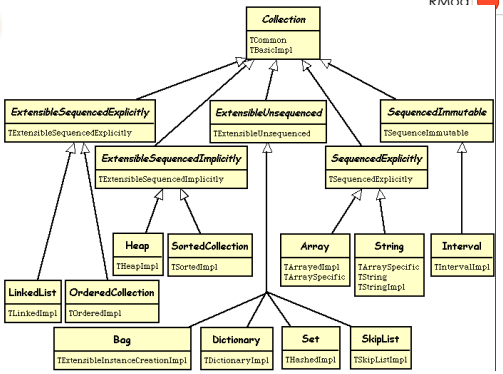
- Methods implemented "too high"
- Code duplication
- Improper inheritance
- Conceptual shortcomings



Resulting trait hierarchies

OOPSLA'04

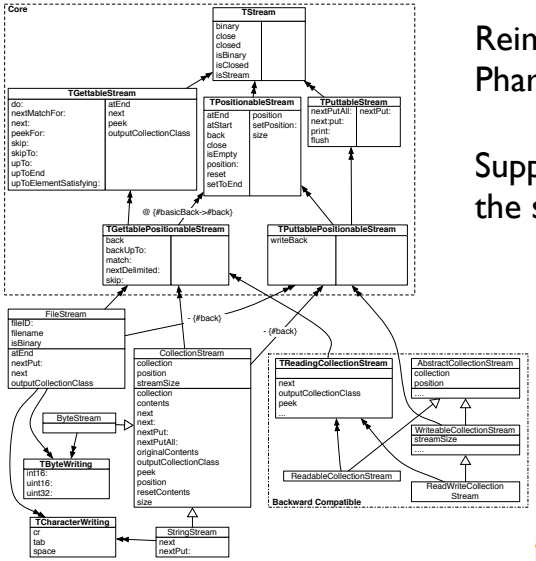
Two trait hierarchies
- Functional Traits
- Implementation Traits



Very fine-grained
- Most traits consist of multiple subtraits

Saved ~12% of source code

Nile

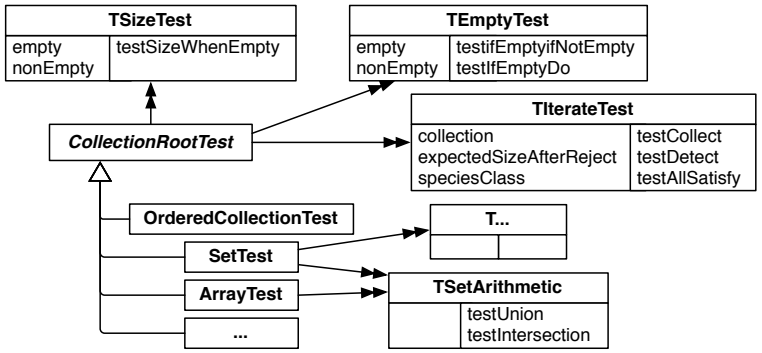


Reimplementing streams in Pharo and Squeak

Supports old and new styles with the same traits recomposed

Composing and reusing Tests...

TOOLS'09



Stateless Traits Current Status

Implemented in Squeak/Pharo Smalltalk
- Fully backwards compatible
- No performance penalty for method lookup
- Refactored Smalltalk collections
- Bootstrapped Squeak kernel (metaclass...)

Used in Scala (but looks more like mixins)
Replace classes in Fortress (Sun MicroSystems)
Introduced in Perl6, Slate, DrScheme, AmbiantTalk,

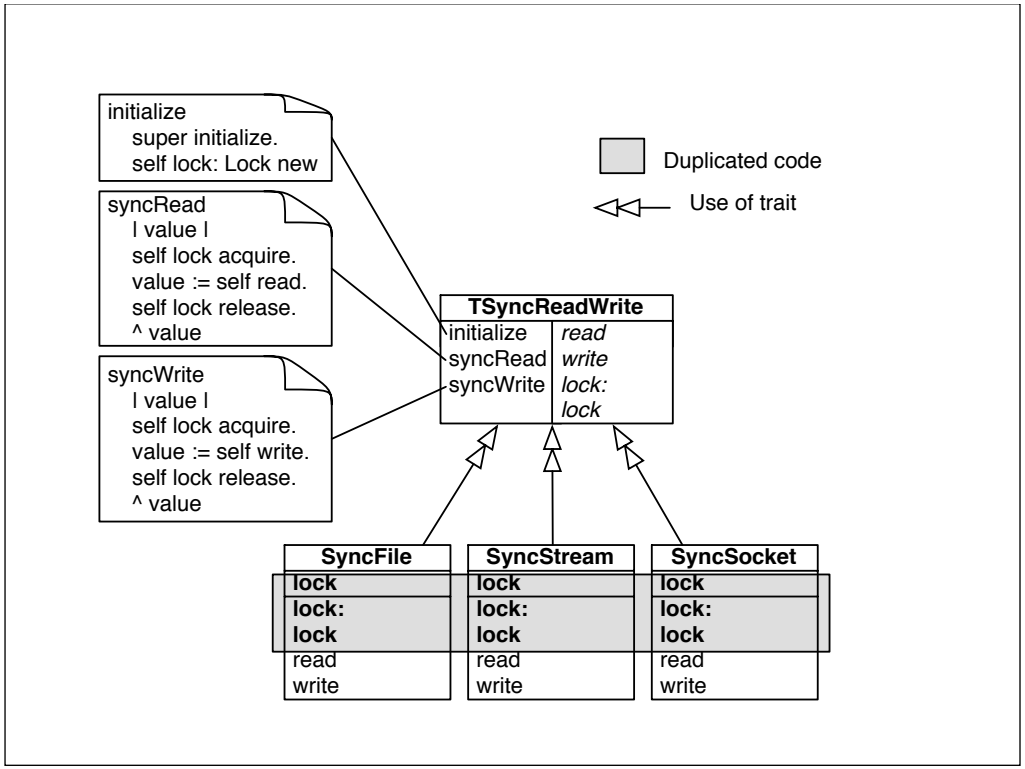
References: ECOOP'03, OOPSLA'03, TOPLAS'05

Road Map

Motivation
Traits - Fine grained structural composition
Results
Stateful Traits - state into traits?
Freezable Traits - full method conflict
Conclusion

Trait Limits

- Trait users should define missing traits state
- Important required methods and required state are mixed
- Boilerplate glue code
- Propagation of required accessors

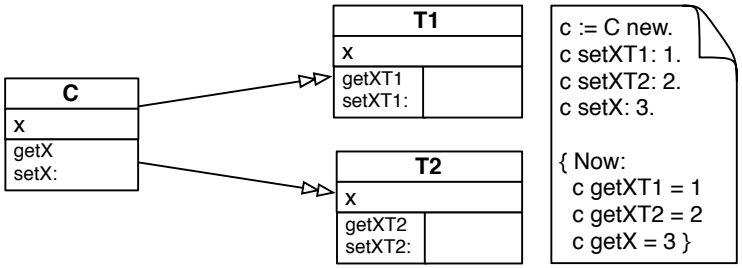

initialize
super initialize.
self lock: Lock new
syncRead
| value |
self lock acquire.
value := self read.
self lock release.
^ value
syncWrite
| value |
self lock acquire.
value := self write.
self lock release.
^ value
Duplicated code
Use of trait
TSyncReadWrite
initialize
syncRead
syncWrite
read
write
lock:
lock
SyncFile
SyncStream
SyncSocket
lock
lock:
lock
read
write

Stateful traits

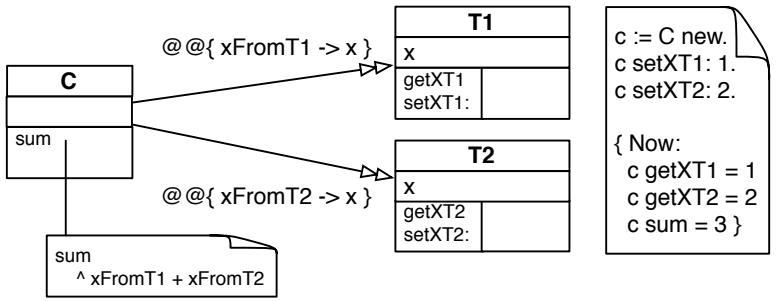
Instance variables are per default *private* to the trait

Via composition
Composing classes may get access to state
Composing classes may merge the accessed state

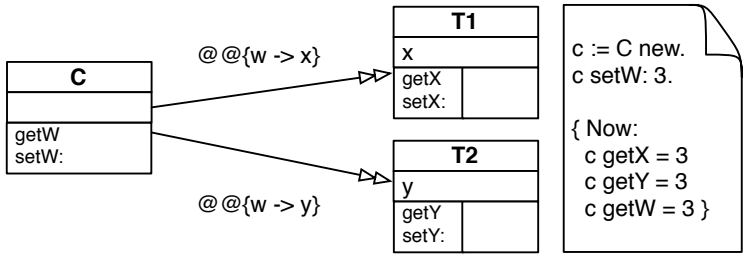
By default state is private



But can be accessed



And merged


C
getW
setW:
@@{w -> x}
@@{w -> y}
T1
x
getX
setX:
T2
y
getY
setY:
c := C new.
c setW: 3.
{ Now:
c getX = 3
c getY = 3
c getW = 3 }

Analysis

Traits and state reconciled
Stateless traits are just a special case of stateful traits
No accessor required
No boiler plate code
Flattening property can be rescued using instance variable mangling
    more engineering work for tools and to lie to the developer

State is private but can be made accessible at composition time

Open Questions

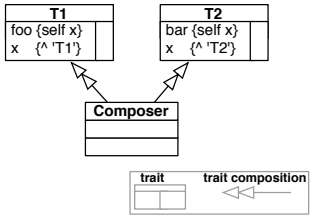
Do we really need merge?
Do we need classes?
What is the difference between a stateful traits and a class?
Do we need stateful traits if we unify state and methods and have visibility mechanisms?
Can we have far less operators?

Road Map

Why
Traits - Fine grained structural composition
Results
Stateful Traits - Can we get state into traits?
Freezable Traits - Full method conflict
Conclusion

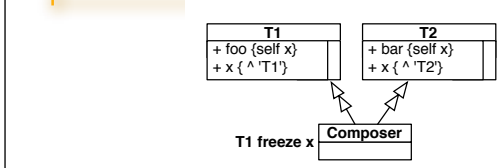
Another Trait Limit

Methods may conflict while they are private to the traits



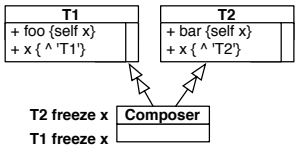
You can only have one x in Composer: either T1x, T2x or Cx

User Changeable Method Visibility



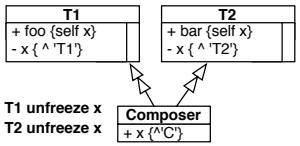

```
Composer new foo -> 'T1'
Composer new bar -> 'T2'
Composer new x -> 'T2'
```

Freezing T1 x



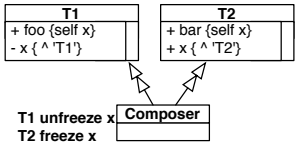

```
Composer new foo -> 'T1'
Composer new bar -> 'T2'
Composer new x -> Error
```

Freezing T2 x and T2 x




```
Composer new foo -> 'C'
Composer new bar -> 'C'
Composer new x -> 'C'
```

Conflict resolution via method redefinition in Composer

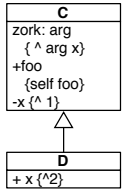



```
Composer new foo -> 'T1'
Composer new bar -> 'T2'
Composer new x -> 'T1'
```

Unfreezing T1 x and freezing T2 x

“private” methods without static types?

What is the x invoked on arg x ?




```
D new foo
D new zork: C new
```

In a dynamic language 3 choices:
Identity of the receiver
Type of the receiver (its class)
Another syntactical messages send

Syntactic difference

self-sends can be private (not visible and early-bound) or public (visible and late-bound)

object-sends (self-sends and super-sends) are normal (late-bound and public)

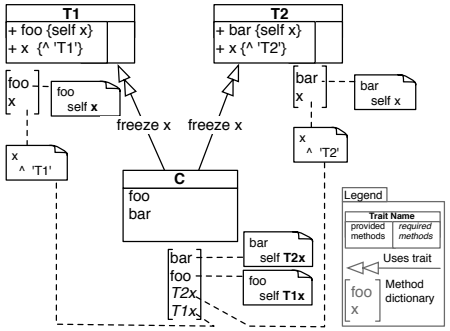
super sends are normal (static)

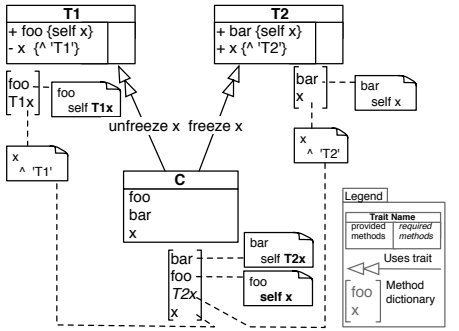
Open Questions

Do we really need static private methods in a dynamic world?

Ruby private methods are dynamically bound

Implementation





Towards a pure trait language

Do we still want classes and traits?
Classes without inheritance and only composition?
Is it Eiffel?
Is it a kind of self with classes
Rename vs. aliasing?

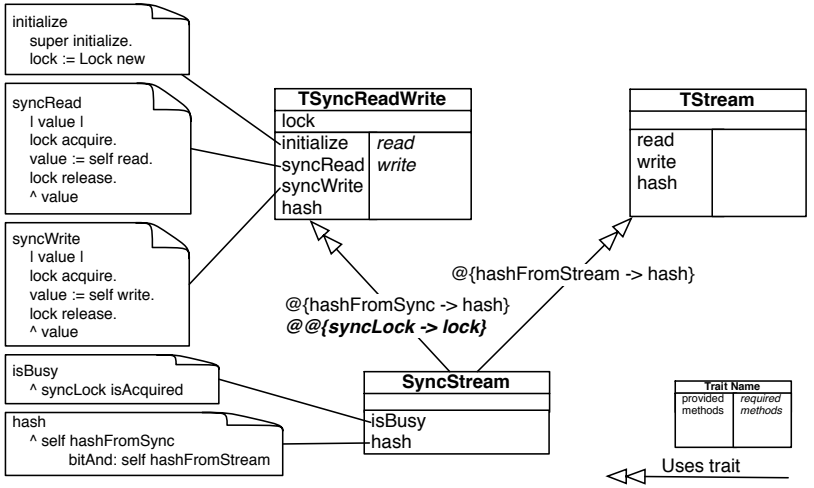
Conclusion and future

Traits are interesting constructs
But
Need more experience building complex software
Do we get patterns (composition vs. inheritance)?
Can we have less operators?

Can we have a trait-based only language?

Let's rethink that... slowly

An example


initialize
super initialize.
lock := Lock new
syncRead
| value |
lock acquire.
value := self read.
lock release.
^ value
syncWrite
| value |
lock acquire.
value := self write.
lock release.
^ value
isBusy
^ syncLock isAcquired
hash
^ self hashFromSync
bitAnd: self hashFromStream
TSyncReadWrite
lock
initialize
syncRead
syncWrite
hash
read
write
TStream
read
write
hash
@{hashFromStream -> hash}
@{hashFromSync -> hash}
@@{syncLock -> lock}
SyncStream
isBusy
hash
Trait Name
provided methods
required methods
Uses trait

A word of implementation