Stateful Traits

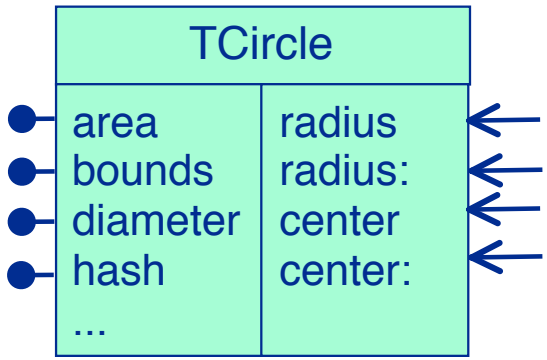
A. Bergel, S. Ducasse, O. Nierstrasz, R. Wuyts

Roadmap

- Traits in a nutshell
- Trait limits
- Stateful traits
- Open questions
- Conclusion

What are Traits?

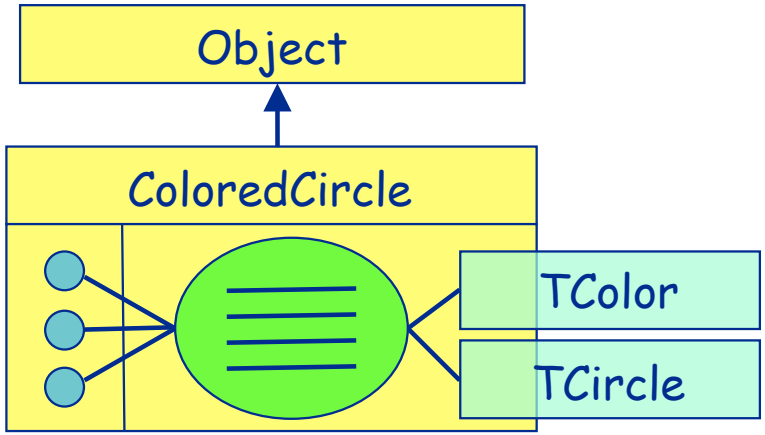
- Traits are *parameterized* behaviors
  - Traits *provide* a set of methods
  - Traits *require* a set of methods
  - Traits are purely behavioral (Traits do not specify any state)



Composing Classes out of Traits

Traits are the behavioral building blocks of classes

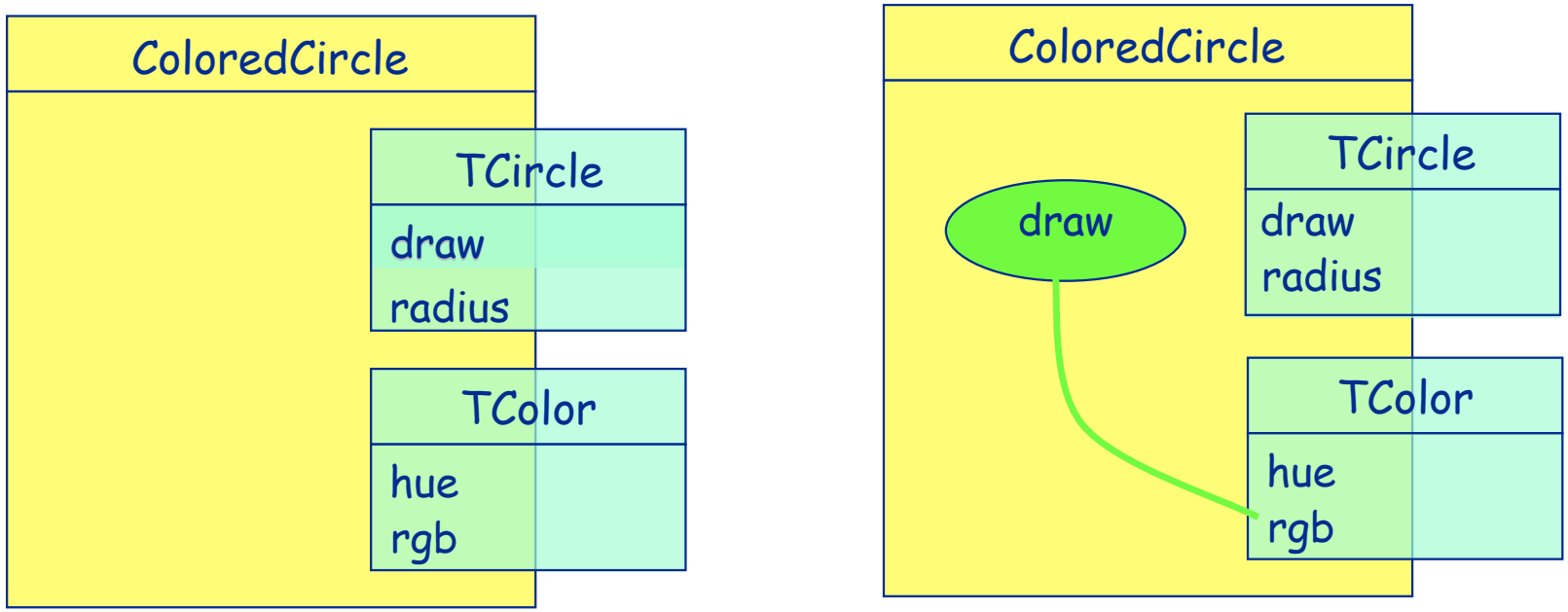
Class = Superclass + State + Traits + Glue Methods



The *composing* class retains control of the composition

Composition Rules

Class methods take precedence over trait methods



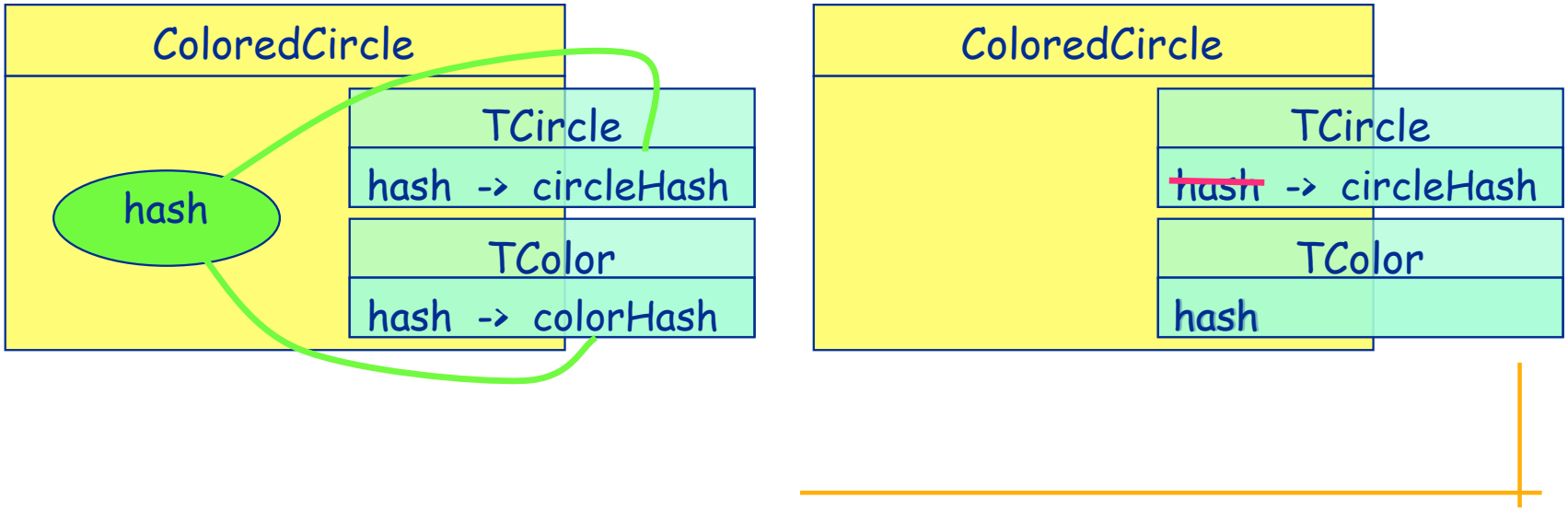
Conflicts are *explicitly* resolved

Override the conflict with a glue method

- Aliases provide access to the conflicting methods

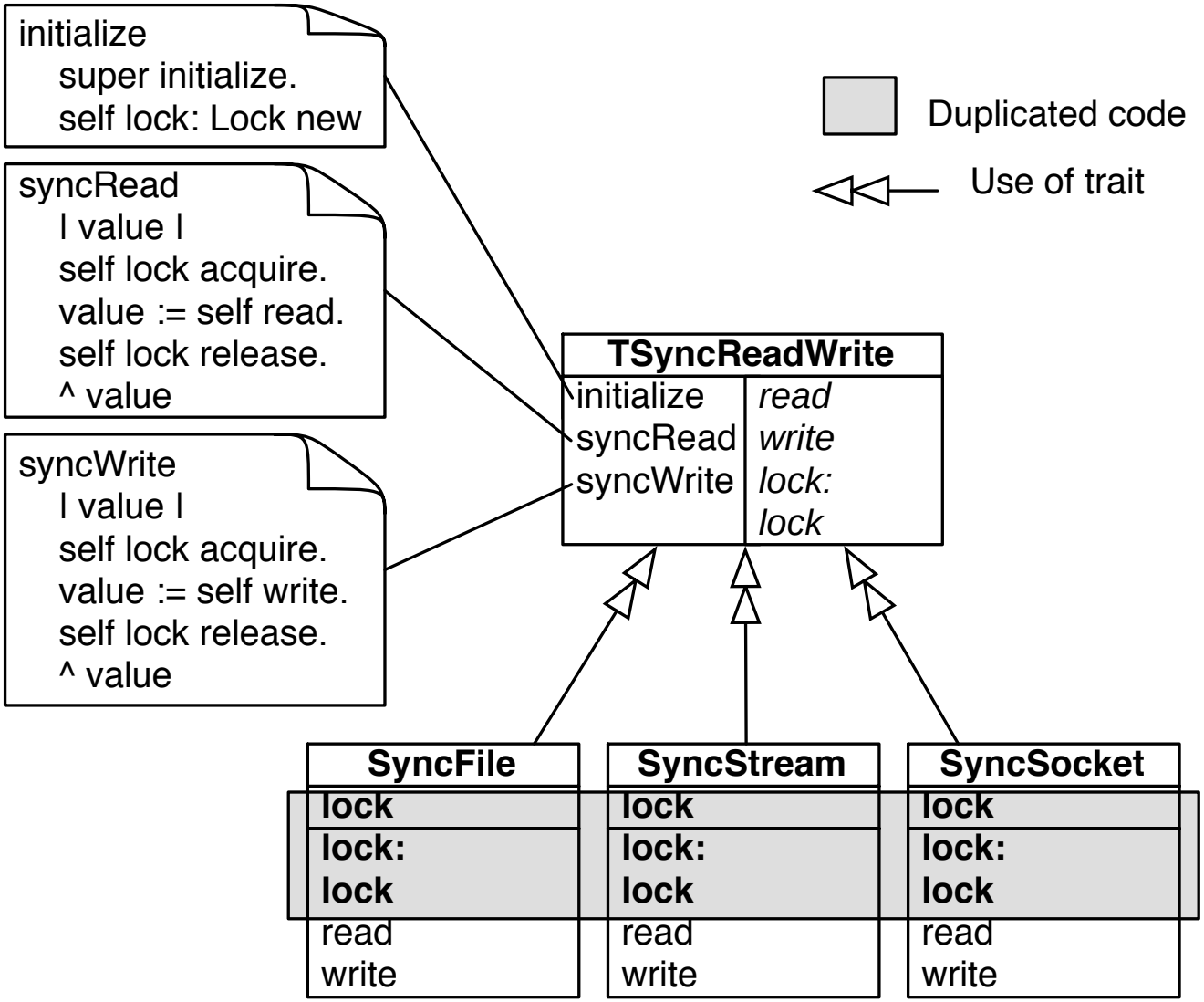
Avoid the conflict

- Exclude the conflicting method from one trait



Trait limits

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

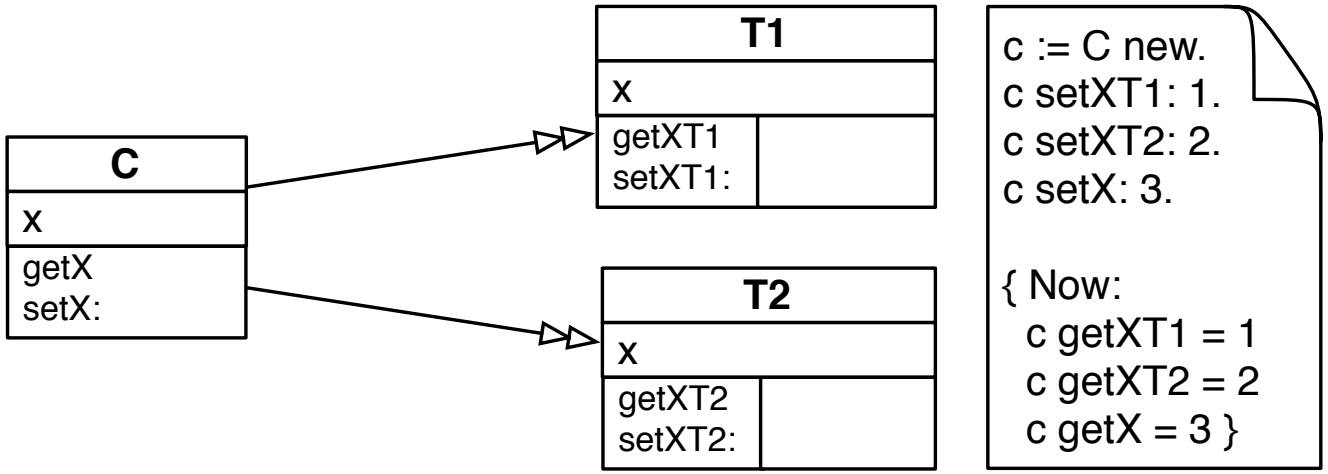

initialize
super initialize.
self lock: Lock new
syncRead
| value |
self lock acquire.
value := self read.
self lock release.
^ value
syncWrite
| value |
self lock acquire.
value := self write.
self lock release.
^ value
Duplicated code
Use of trait
TSyncReadWrite
initialize
syncRead
syncWrite
read
write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

Stateful traits

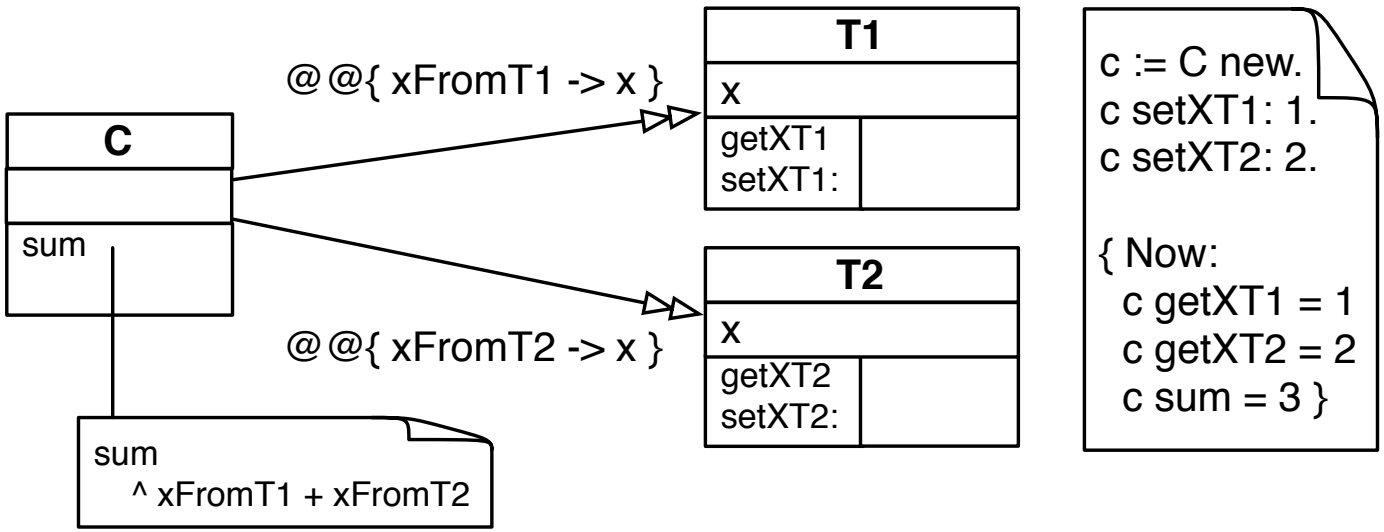
- Instance variables are per default private to the trait

- Via composition
  - Client classes may get access to state
  - Client classes may merge the accessed state

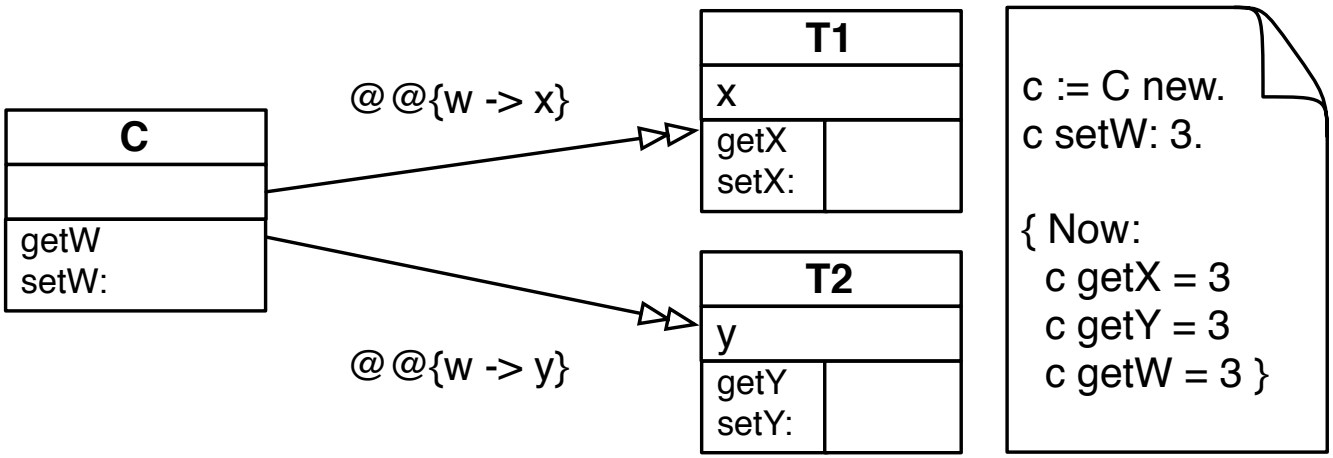
By default state is private



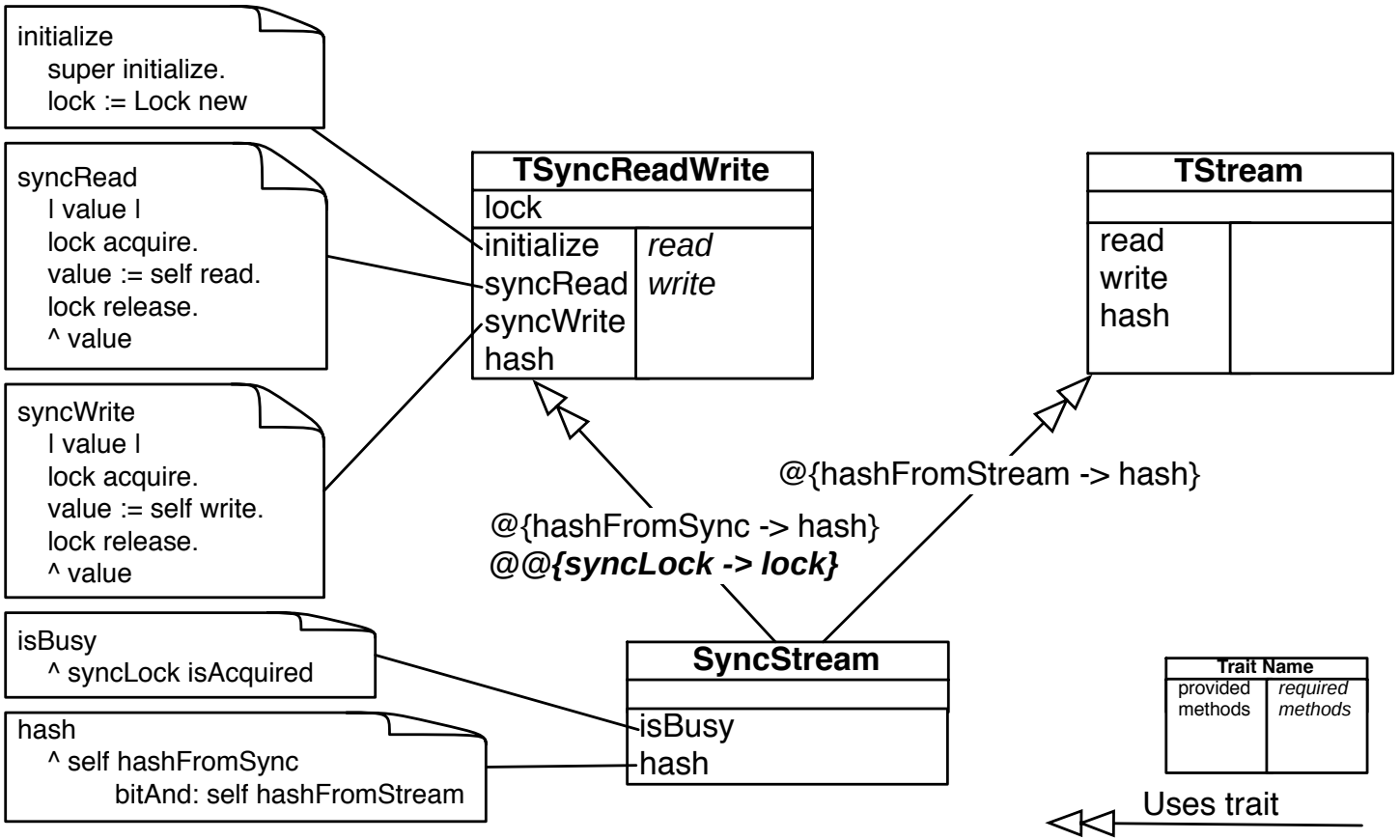
But can be accessed



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 }

An example

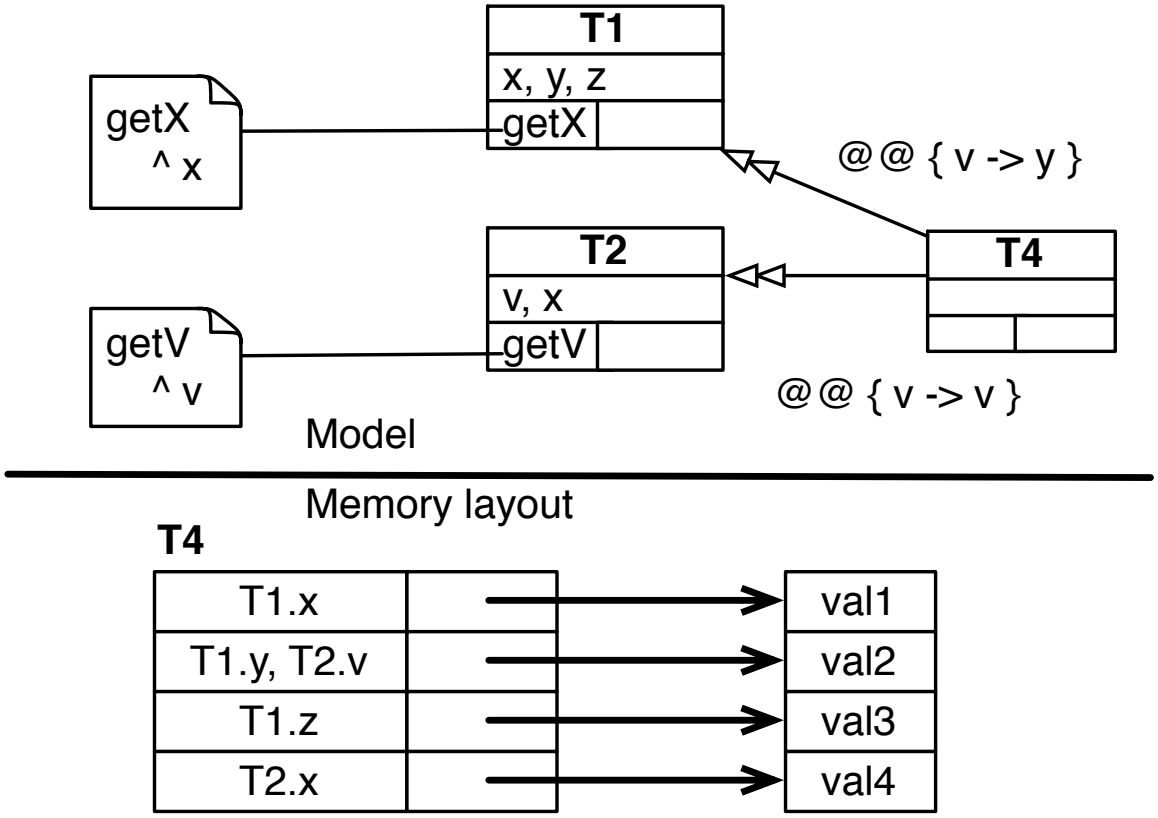

initialize
super initialize.
lock := Lock new
syncRead
| value |
lock acquire.
value := self read.
lock release.
^ value
syncWrite
| value |
lock acquire.
value := self write.
lock release.
^ value
isBusy
^ syncLock isAcquired
hash
^ self hashFromSync
bitAnd: self hashFromStream
TSyncReadWrite
lock
initialize
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

Analysis

- Stateless traits are just a special case of stateful traits
- No accessor required
- No boiler plate code
- Trait can be “complete”

- Flattening property can be rescued using instance variable mangling

A word of implementation



Open Questions

- Do we really need merge?
- Do we need stateful traits if we get private methods?

Conclusion

- Traits and state reconciled
- Traits are complete
- State is private but can be made accessible at composition time

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