Script **generated by TTT**

Title: Simon: Programmiersprachen (18.01.2013)

Date: Fri Jan 18 10:05:04 CET 2013

Duration: 74:23 min

Pages: 31

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Traits

Dr. Axel Simon and Dr. Michael Petter
Winter term 2012

"Is Multiple Inheritance the holy grail of reusability?"

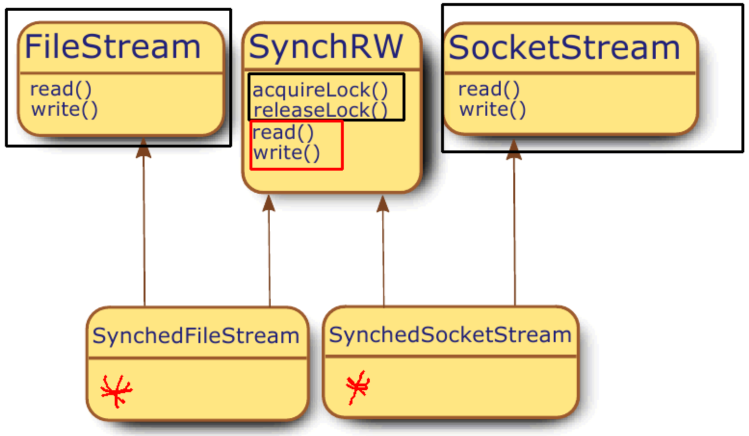
Learning outcomes

1. Identify problems of composition and decomposition
2. Understand semantics of traits
3. Separate function provision, object generation and class relations
4. Traits and existing program languages

Reusability ≡ Inheritance?

- Codesharing in Object Oriented Systems is usually inheritance-centric.
- Inheritance itself comes in different flavours:
  - single inheritance
  - multiple inheritance
  - mixin inheritance
- All flavours of inheritance tackle problems of *decomposition* and *composition*

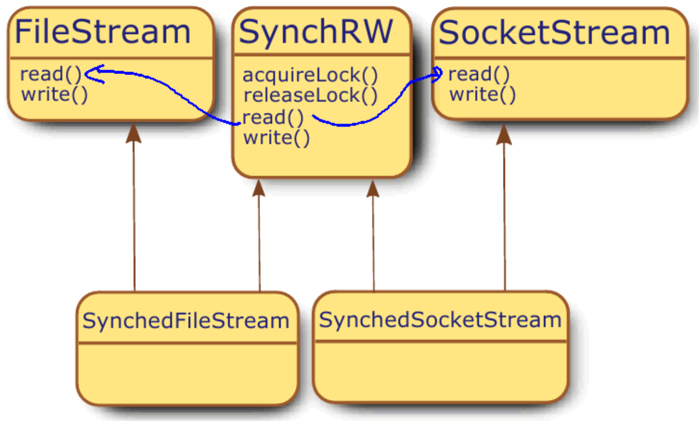
Streams



⚠ **Duplicated Wrappers**

Multiple Inheritance is not applicable as super-References are *statically bound* (⇝ Alternative: Mixins)

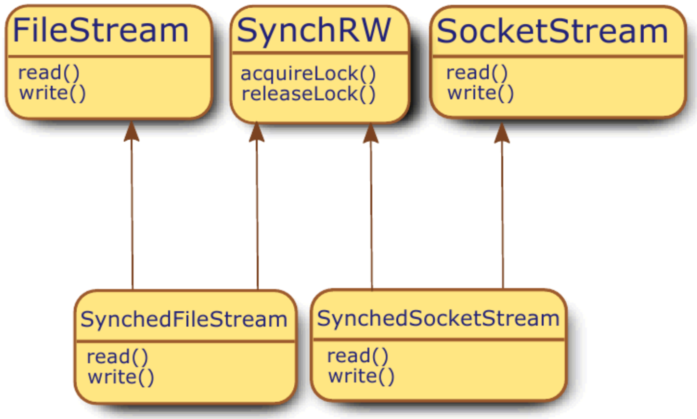
Streams



⚠ **Duplicated Wrappers**

Multiple Inheritance is not applicable as super-References are *statically bound* (⇝ Alternative: Mixins)

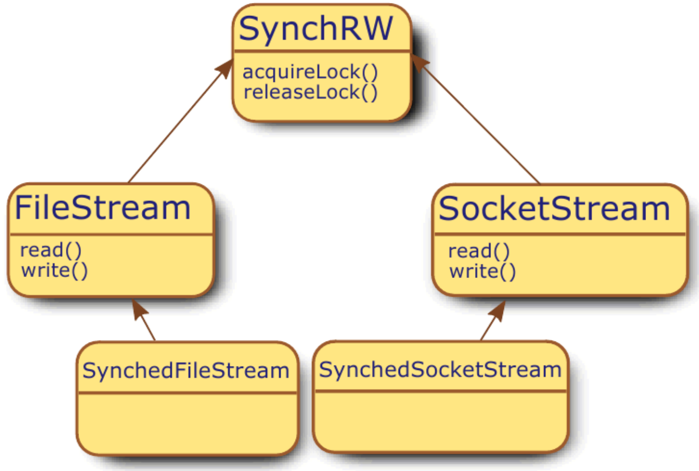
Streams modified



⚠ **Duplicated Features**

`read`/`write` Code is essentially *identical but duplicated*

Oh my god, streams!



⚠ **Inappropriate Hierarchies**

Implement methods (`acquireLock`/`releaseLock`) *to high*

Decomposition problems

All the problems of

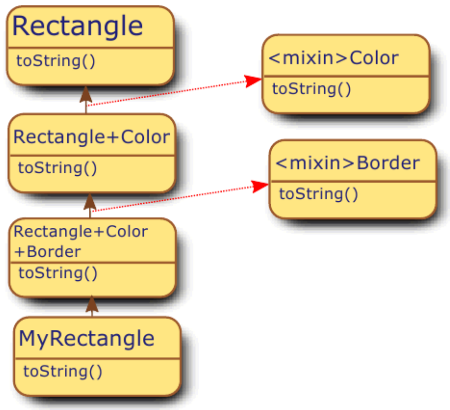
- duplicated Wrappers
- duplicated Features
- inappropriate Hierarchies

are centered around the question

"How do I distribute functionality over a hierarchy"

⇝ *functional decomposition*

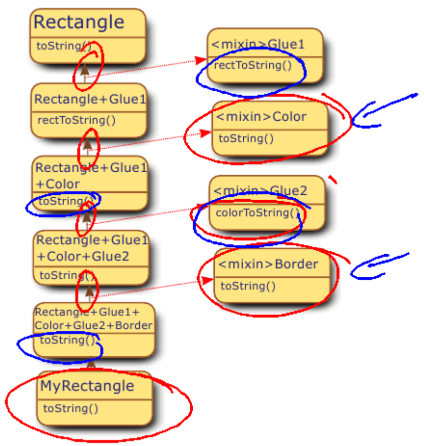
Are Mixins the solution?



⚠ **Fragile Hierarchies**

- Linearization overrides identically named methods earlier in the chain
- `super` is not enough to sufficiently qualify inherited features, while explicit qualification makes refactoring difficult

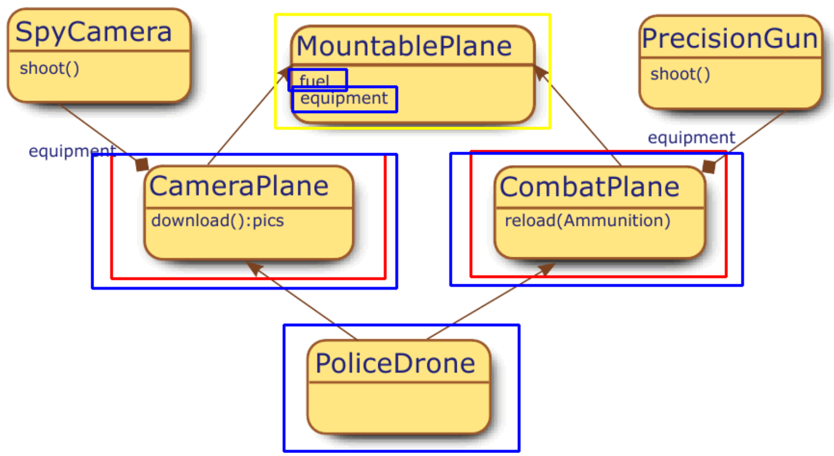
Are Mixins the solution?



⚠ **Lack of Control and Dispersal of Glue Code**

Overriding methods always happens in parallel ⇝ *lack of control*
Glue code penetrates the whole hierarchy ⇝ *dispersal of glue code*

And Multiple Inheritance?



⚠ **Conflicting Features**

Common base classes are shared or duplicated at class level
⇝ No *fine-grained specification* of duplication or sharing

The idea behind Traits

- A lot of the problems originate from the coupling of implementation and modelling
- Interfaces seem to be hierarchical
- Functionality seems to be modular

Central idea

Separate Object creation from modelling hierarchies and assembling functionality.

$\rightsquigarrow$ Use interfaces to design hierarchical signature propagation
$\rightsquigarrow$ Use *traits* as modules for assembling functionality
$\rightsquigarrow$ Use classes as frames for entities, which can create objects

Classes and methods

We will construct our model from the primitive sets of

- a countable set of method *names* $\mathcal{N}$
- a countable set of method *bodies* $\mathcal{B}$
- a countable set of *attribute* names $\mathcal{A}$

Values of method bodies $\mathcal{B}$ are extended to a *flat lattice* $\mathcal{B}^\star$, with elements

- concrete implementations
- $\perp$ undefined
- $\top$ in conflict

and the partial order $\perp \sqsubset m \sqsubset \top$ for each $m \in \mathcal{B}$

Definition (Method)

Partial function, mapping a name to a body

Definition (Method Dictionary $d \in \mathcal{D}$)

Total function $d : \mathcal{N} \mapsto \mathcal{B}^\star$, and $d^{-1}(\top) = \emptyset$

Definition (Class $c \in \mathcal{C}$)

Either nil or $\langle\alpha, d\rangle \cdot c'$ with $\alpha \in \mathcal{A}, d \in \mathcal{D}, c' \in \mathcal{C}$

Traits

A trait $t \in \mathcal{T}$

- is a function $t : \mathcal{N} \mapsto \mathcal{B}^\star$
- has $conflicts : \mathcal{T} \mapsto 2^{\mathcal{N}}$ with $conflicts(t) = \{l \mid t(l) = \top\}$
- $provides : \mathcal{T} \mapsto 2^{\mathcal{N}}$ with $provides(t) = t^{-1}(\mathcal{B})$
- $selfSends : \mathcal{B} \mapsto 2^{\mathcal{N}}$, the set of method names used in self-sends
- $requires : \mathcal{T} \mapsto 2^{\mathcal{N}}$ with $requires(t) = \bigcup_{b \in t(\mathcal{N})} selfSends(b) \setminus provides(t)$

... and differs from Mixins

- Traits are applied to a class *in parallel*, Mixins *incrementally*
- Trait *composition is unordered*, avoiding linearization problems
- Traits do *not contain attributes*, avoiding state conflicts
- With traits, *glue code* is concentrated in particular classes

Trait composition principles

Flat ordering All traits have the same precedence $\rightsquigarrow$ explicit disambiguation

Precedence Class methods take precedence over trait methods

Flattening Non-overridden trait methods have the same semantics as class methods

Trait composition

Composing Classes from Traits:

$$\langle\alpha, d \triangleright t\rangle \cdot c' \qquad \text{with } \langle\alpha, d\rangle \cdot c' \text{ a class}, t \text{ a composition clause}$$

with the overriding operator $\triangleright$:

$$(d \triangleright t)(l) = \begin{cases} t(l) & d(l) = \perp \\ d(l) & \text{otherwise} \end{cases}$$

Composition clauses are based on

- trait sum: $(t_1 + t_2)(l) = t_1(l) \sqcup t_2(l)$
- exclusion: $(t - a)(l) = \begin{cases} \perp & \text{if } a = l \\ t(l) & \text{otherwise} \end{cases}$
- aliasing: $t[a \to b](l) = \begin{cases} t(l) & \text{if } l \neq a \\ t(b) & \text{if } l = a \wedge t(a) = \perp \\ \top & \text{otherwise} \end{cases}$

Trait handling

⚠ Conflicts

Conflicts arise if composed traits posses methods with identical signatures

Conflict traitment

- ✓ Methods can be aliased (→)
- ✓ Methods can be excluded
- ✓ Class Methods override trait methods and sort out conflicts (▷)

Decomposition

✓ Duplicated Features

... can easily be factored out into unique traits.

✓ Inappropriate Hierarchies

Trait composition as means for reusable code frees inheritance to model hierarchical relations.

✓ Duplicated Wrappers

Generic Wrappers can be directly modeled as traits.

Composition

✓ Conflicting Features

Traits cannot have conflicting states, and offer conflict resolving measures like exclusion, aliasing or overriding.

✓ Lack of Control and Dispersal of Glue Code

The composition entity can individually choose for each feature, which trait has precedence or how composition is done. Glue code can be kept completely within the composed entity.

✓ Fragile Hierarchies

Conflicts can be resolved in the glue code. Navigational glue code is avoided.

Simulating Traits in C++


```
template <class Super>
class SyncRW : virtual public Super {
  public: virtual int read(){
    acquireLock();
    int result = Super::read();
    releaseLock();
    return result;
  };
  virtual void write(int n){
    acquireLock();
    Super::write(n);
    releaseLock();
  };
  // ... acquireLock() & releaseLock()
};
```

Simulating Traits in C++

```
template <class Super>
class LogOpenClose : virtual public Super {
   public: virtual void open(){
    Super::open();
    log("opened");
   };
   virtual void close(){
    Super::close();
    log("closed");
   };
   protected: virtual void log(char*s) { ... };
};

template <class Super>
class LogAndSync :
  virtual public LogOpenClose<Super>,
  virtual public SyncRW<Super>
{};
```


Simulating Traits in C++

What misses for full traits?

Compositional expressions are not available:

- Aliasing
- Exclusion
- Precedence of class methods
- Specifying required methods
- Fine-grained control over duplication
- ⇝ Type system not flexible enough

But does that matter?

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 a language with real traits!"

Traits in PHP

```
trait Rectangular {
 private $l=3, $w=4;
 public function printInfo()   {  echo 'rectangular $l x $w';  }
}
trait Colored {
 public $color = "red";
 public function printInfo() {  echo 'color '. $this->color;  }
}

class ColoredRect {
 use Colored, Rectangular;
 public function printInfo(){
  Rectangular::printInfo();
  echo ' with ';
  Colored::printInfo();
 }
}
$o = new ColoredRect();
$o->printInfo();
```


Aliasing Traits in PHP

```
trait Rectangular {
 private $l=3, $w=4;
 public function printInfo()   {  echo 'rectangular $l x $w';  }
}
trait Colored {
 public $color = "red";
 public function printInfo() {  echo 'color '. $this->color;  }
}

class ColoredRect {
 use Colored, Rectangular {
  Rectangular::printInfo as printShapeInfo;
  Colored::printInfo     as printColorInfo;
 }
 public function printInfo(){ ... }
}

$o = new ColoredRect();
$o->printColorInfo();
```


Alasing Drones as Traits in PHP

```
trait MountablePlane {
 abstract function store($equip);
 abstract function retrieve();
 public function mount($equip){ $this->store($equip); }
 public function shoot()      { $this->retrieve()->fire(); }
}
trait CameraPlane { use MountablePlane; }
trait CombatPlane { use MountablePlane; }
class PoliceDrone { use CameraPlane, CombatPlane {
      CameraPlane::mount as mountCam;
      CombatPlane::mount as mountGun;
      CameraPlane::store as storeCamera;
      CombatPlane::store as storeGun;
      CameraPlane::shoot as shootFoto;
      CombatPlane::shoot as shootTerrorist;
      CameraPlane::retrieve as retrieveCamera;
      CombatPlane::retrieve as retrieveGun;
    }
    private $cam, $gun;
    ...
```


Alasing Drones as Traits in PHP

⚠ Exclusion

Unfortunaly, exclusion does not seem to work in PHP as expected, as well as aliasing ⇝ No real solution for our problem!

⚠ Traits in PHP

- Composable
- Aliasing without excluding the original
- Exclusion virtually not present

⇝ Real traits elsewhere

e.g. in Smalltalk (⇝ *Squeak*)

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.
```


so far so...

✓ good

- Syntax looks really promising
- Aliasing and Exclusion is implemented

⚠ bad

- Especially Squeak features one of the most unconventional IDEs
- . . . and there is no command line mode!

Lessons learned

Lessons Learned

- Single inheritance, multiple Inheritance and Mixins reveal shortcomings in real world problems
- 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.
- Martin Odersky, Lex Spoon, and Bill Venners.
  Programming in Scala: A Comprehensive Step-by-step Guide.
  Artima Incorporation, USA, 1st edition, 2008.
  ISBN 0981531601, 9780981531601.
- Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black.
  Traits: Composable units of behaviour.
  European Conference on Object-Oriented Programming (ECOOP), 2003.