Script generated by TTT

Title: Petter: Programmiersprachen (20.01.2016)

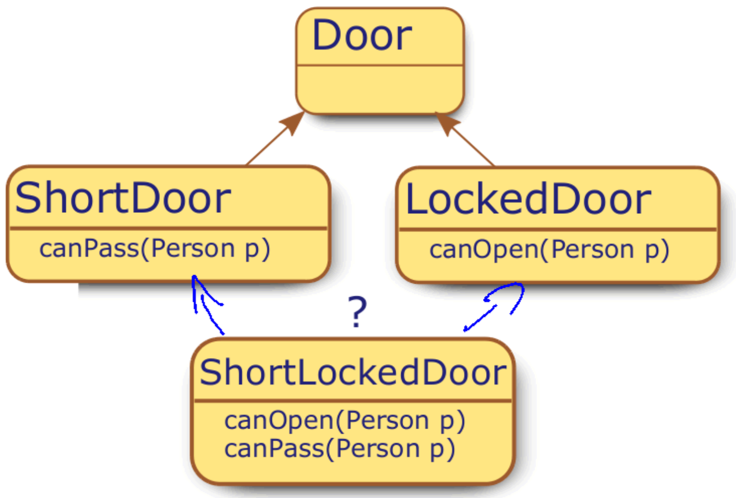
Date: Wed Jan 20 15:15:18 CET 2016

Duration: 45:47 min

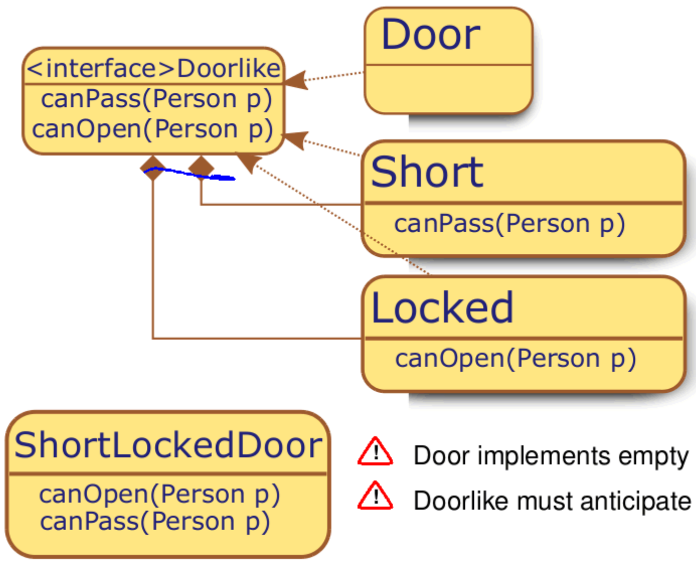
Pages: 24

“What advanced techiques are there besides multiple implementation inheritance?”

The Adventure Game

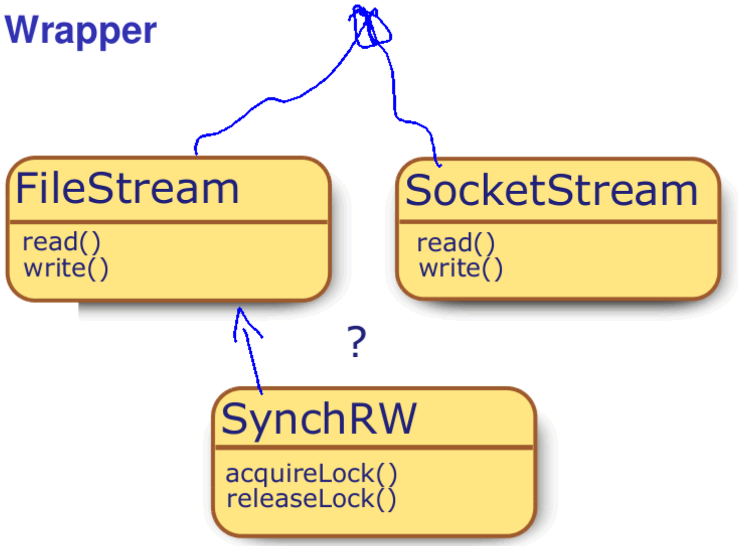


The Adventure Game



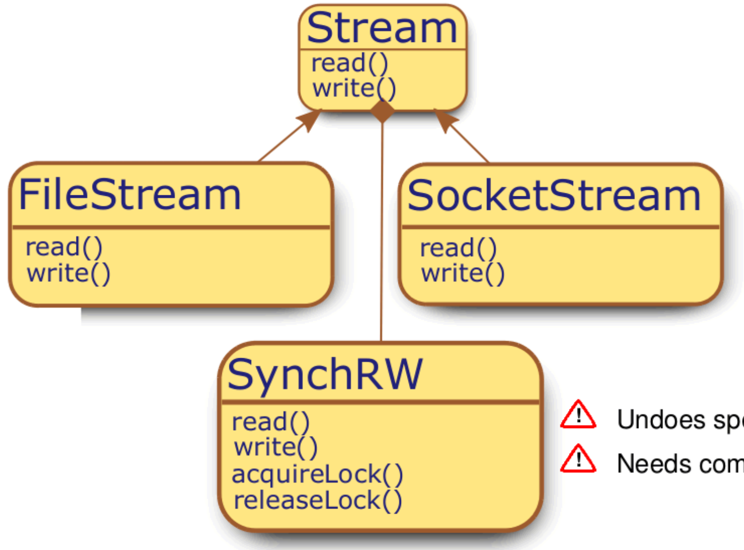
- ⚠ Door implements empty methods
- ⚠ Doorlike must anticipate wrappers

The Wrapper



- ⚠ Cannot inherit from both seperately
- ⚠ Creating new wrapping Classes duplicates code

The Wrapper



- ⚠ Undoes specialization
- ⚠ Needs common ancestor

“Let’s go back to the basics of inheritance”

Classes and Methods

The building blocks for classes are

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

Classes map names to elements from the *flat lattice* $\mathcal{B}$ (called bindings), consisting of:

- method bodies $\in \mathbb{B}$ or classes $\in \mathcal{C}$
- attribute offsets $\in \mathbb{N}^+$
- $\perp$ (yet) undefined
- $\top$ in conflict

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

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

A partial function $c : \mathcal{N} \mapsto \mathcal{B}$ is called abstract class.

Definition (Interface and Class)

An abstract class c is called (with pre beeing the preimage)

interface iff $\forall_{n \in \mathrm{pre}(c)} . \; c(n) = \boxed{\perp.}$

(concrete) class iff $\forall_{n \in \mathrm{pre}(c)} . \; \boxed{\perp \sqsubset} c(n) \boxed{\sqsubset \top.}$

Computing with Classes and Methods

Definition (Family of classes $\mathcal{C}$)

We call the set of all maps from names to bindings the family of abstract classes $\mathcal{C} := \mathcal{N} \mapsto \mathcal{B}$.

Several possibilites for composing maps $\mathcal{C} \,\Box\, \mathcal{C}$:

- the symmetric join $\sqcup$, defined componentwise:

$$(c_1 \sqcup c_2)(n) = b_1 \sqcup b_2 = \begin{cases} b_2 & \text{if } b_1 = \bot \text{ or } n \notin \text{pre}(c_1) \\ b_1 & \text{if } b_2 = \bot \text{ or } n \notin \text{pre}(c_2) \\ b_2 & \text{if } b_1 = b_2 \\ \top & \text{otherwise} \end{cases} \quad \text{where } b_i = c_i(n)$$

- in contrast, the asymmetric join $\boxed{\sqcup\!\!\!\sqcup}$ defined componentwise:

$$\boxed{(c_1 \sqcup\!\!\!\sqcup c_2)(n)} = \begin{cases} c_1(n) & \boxed{\text{if } n \in \text{pre}(c_1)} \\ c_2(n) & \boxed{\text{otherwise}} \end{cases}$$

Example: Smalltalk-Inheritance

Smalltalk inheritance

- is the archetype for inheritance in mainstream languages like Java or C#
- inheriting smalltalk-style establishes a reference to the parent

Definition (Smalltalk inheritance ($\triangleright$))

Smalltalk inheritance is the binary operator $\triangleright : \mathcal{C} \times \mathcal{C} \mapsto \mathcal{C}$, defined by $c_1 \triangleright c_2 = \{\texttt{super} \mapsto c_2\} \sqcup\!\!\!\sqcup (c_1 \sqcup\!\!\!\sqcup c_2)$

Example: `Doors`

$$Door = \{canPass \mapsto \bot, canOpen \mapsto \bot\}$$

$$LockedDoor = \{canOpen \mapsto 0x4204711\} \triangleright Door$$

$$= \{\texttt{super} \mapsto Door\} \sqcup\!\!\!\sqcup (\{canOpen \mapsto 0x4204711\} \sqcup\!\!\!\sqcup Door)$$

$$= \{\boxed{\texttt{super} \mapsto Door}, \boxed{canOpen \mapsto 0x4204711}, \boxed{canPass \mapsto \bot}\}$$

Excursion: Beta-Inheritance

In *Beta*-style inheritance

- the design goal is to provide security from replacement of a method by a different method.
- methods in parents dominate methods in subclass
- the keyword `inner` explicitely delegates control to the subclass

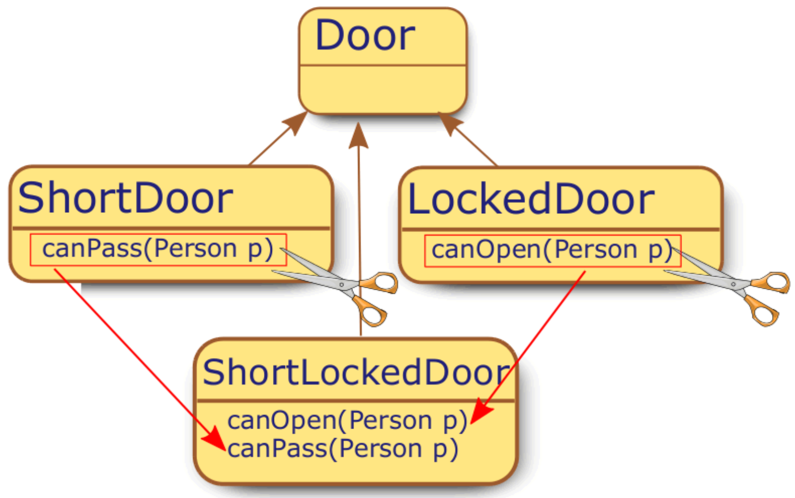
Definition (Beta inheritance ($\triangleleft$))

Beta inheritance is the binary operator $\triangleleft : \mathcal{C} \times \mathcal{C} \mapsto \mathcal{C}$, definied by $\boxed{c_1 \triangleleft c_2} = \{\boxed{\texttt{inner} \mapsto c_1}\} \sqcup\!\!\!\sqcup (\boxed{c_2 \sqcup\!\!\!\sqcup c_1})$

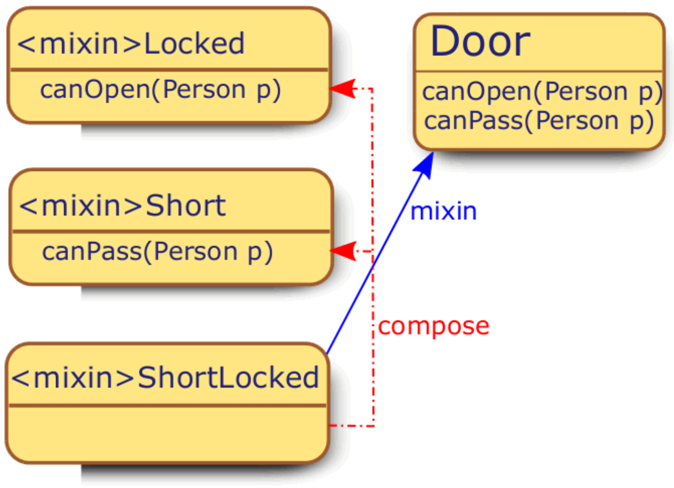
Example (equivalent syntax):

```
class Person {
  String name ="Axel Simon";
  public String toString(){ return name+inner.toString();};
};
class Graduate extends Person {
  public extension String toString(){ return ", Ph.D."; };
};
```


Adventure Game with Code Duplication



Adventure Game with Mixins



Adventure Game with Mixins

```
class Door {
 boolean canOpen(Person p) { return true; };
 boolean canPass(Person p) { return p.size() < 210; };
}
mixin Locked {
 boolean canOpen(Person p){
  if (!p.hasItem(key)) return false; else return super.canOpen(p);
 }
}
mixin Short {
 boolean canPass(Person p){
  if (p.height()>1) return false; else return super.canPass(p);
 }
}
class ShortDoor = Short(Door);
class LockedDoor = Locked(Door);
mixin ShortLocked = Short o Locked;
class ShortLockedDoor  = Short(Locked(Door))
class ShortLockedDoor2 = ShortLocked(Door);
```


Back to the blackboard!

Abstract model for Mixins

A Mixin is a *unary second order type expression*. In principle it is a curried version of the Smalltalk-style inheritance operator. In certain languages, programmers can create such mixin operators:

Definition (Mixin)

The mixin constructor $mixin : \mathcal{C} \mapsto (\mathcal{C} \mapsto \mathcal{C})$ is a unary class function, creating a unary class operator, defined by:

$$mixin(c) = \lambda x \, . \, c \rhd x$$

⚠ Note: Mixins can also be composed $\circ$:

Example: Doors

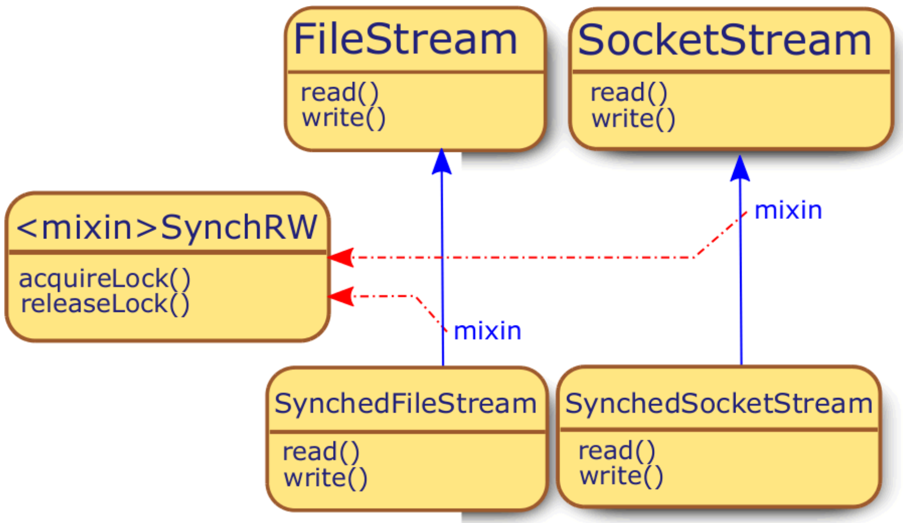
$$Locked = \{canOpen \mapsto 0x1234\}$$

$$Short = \{canPass \mapsto 0x4711\}$$

$$Composed = mixin(Short) \circ (mixin(Locked)) = \lambda x \, . \, \boxed{Short \rhd (Locked \rhd x)}$$

$$= \lambda x \, . \, \{\text{super} \mapsto Locked\} \uplus (\{canOpen \mapsto 0x1234, canPass \mapsto 0x4711\} \rhd x)$$

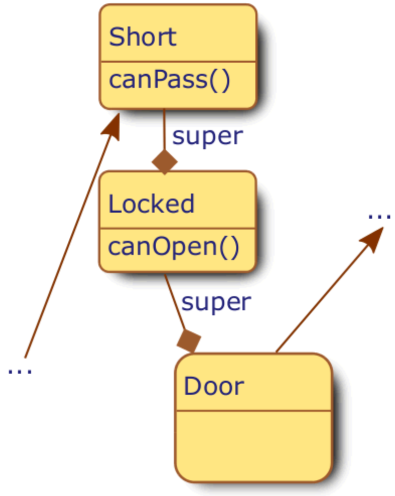
Wrapper with Mixins



Mixins on Implementation Level

```
class Door {
 boolean canOpen(Person p)...
 boolean canPass(Person p)...
}
mixin Locked {
 boolean canOpen(Person p)...
}
mixin Short {
 boolean canPass(Person p)...
}
class ShortDoor
    = Short(Door);
class ShortLockedDoor
    = Short(Locked(Door));
...

ShortDoor d
    = new ShortLockedDoor();
```



⚠ *non-static* `super`-References
⇝ dynamic dispatching without precomputed virtual table

“Surely multiple inheritance is powerful enough to simulate mixins?”

Simulating Mixins in C++

```
template <class Super>
class SyncRW : 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 Mixins in C++

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


True Mixins vs. C++ Mixins

True Mixins

- `super` natively supported
- Mixins as Template do not offer composite mixins
- C++ Type system not modular
- ⇝ Mixins have to stay source code
- Hassle-free simplified version of multiple inheritance

C++ Mixins

- Mixins reduced to templated superclasses
- Can be seen as coding pattern

Common properties of Mixins

- Linearization is necessary
- ⇝ Exact sequence of Mixins is relevant

Ruby

```
class Person
  attr_accessor :size
  def initialize
    @size = 160
  end
  def hasKey
    true
  end
end
class Door
  def canOpen (p)
    true
  end
  def canPass(person)
    person.size < 210
  end
end
```

```
module Short
  def canPass(p)
    p.size < 160 and super(p)
  end
end
module Locked
  def canOpen(p)
    p.hasKey() and super(p)
  end
end

class ShortLockedDoor < Door
  include Short
  include Locked
end

p = Person.new
d = ShortLockedDoor.new
puts d.canPass(p)
```


Ruby

```
class Door
  def canOpen (p)
    true
  end
  def canPass(person)
    person.size < 210
  end
end
module Short
  def canPass(p)
    p.size < 160 and super(p)
   end
end
module Locked
  def canOpen(p)
    p.hasKey() and super(p)
  end
end
```

```
module ShortLocked
  include Short
  include Locked
end
class Person
  attr_accessor :size
  def initialize
    @size = 160
  end
  def hasKey
    true
  end
end

p = Person.new
d = Door.new
d.extend ShortLocked

puts d.canPass(p)
```

Lessons Learned

Lessons Learned

1. Formalisms to model inheritance
2. Mixins provide soft multiple inheritance
3. Multiple inheritance can not compensate the lack of `super` reference
4. Full extent of mixins only when mixins are 1st class language citizens