Script generated by TTT

Title: Petter: Programmiersprachen (10.12.2014)

Date: Wed Dec 10 14:28:09 CET 2014

Duration: 75:57 min

Pages: 31

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Mixins

Dr. Michael Petter
Winter 2014

"What advanced techiques are there besides multiple implementation inheritance?"

Outline

Weak implementation inheritance

1. Decorator Problem
2. Wrapper Problem

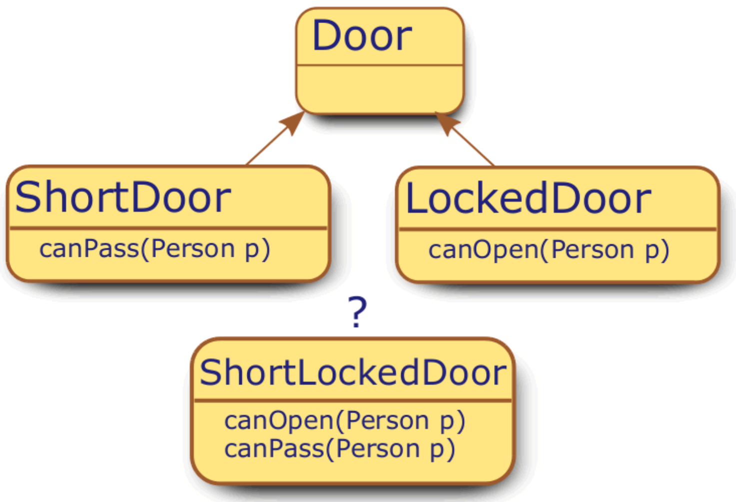
Inheritance in Detail

1. Models for single inheritance
2. Introducing Mixins
3. Modelling Mixins

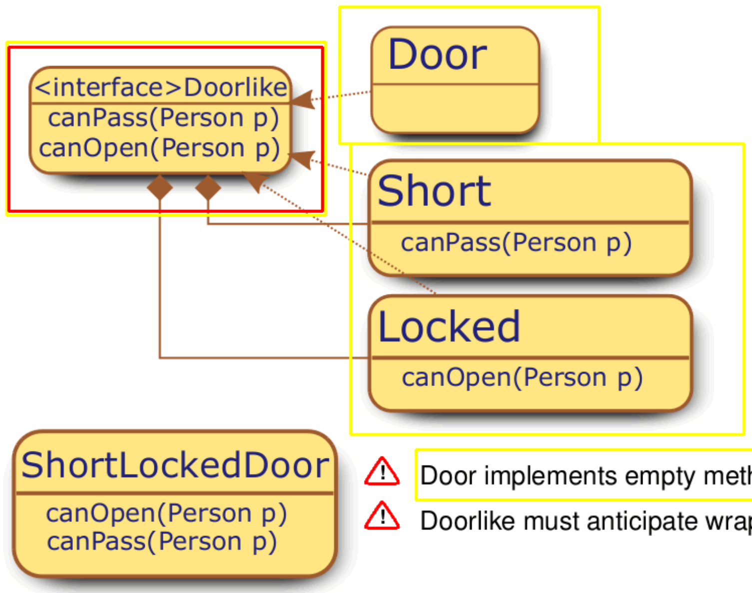
Mixins in the wild

1. Mixins as C++-Pattern
2. Native Mixins

The Adventure Game



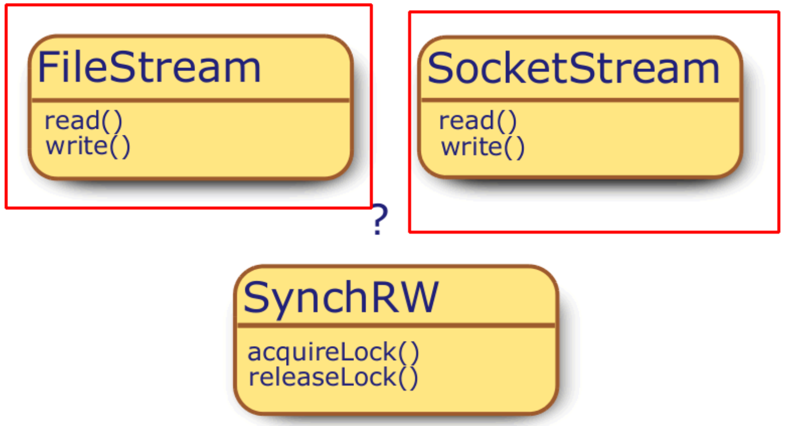
The Adventure Game



⚠ Door implements empty methods

⚠ Doorlike must anticipate wrappers

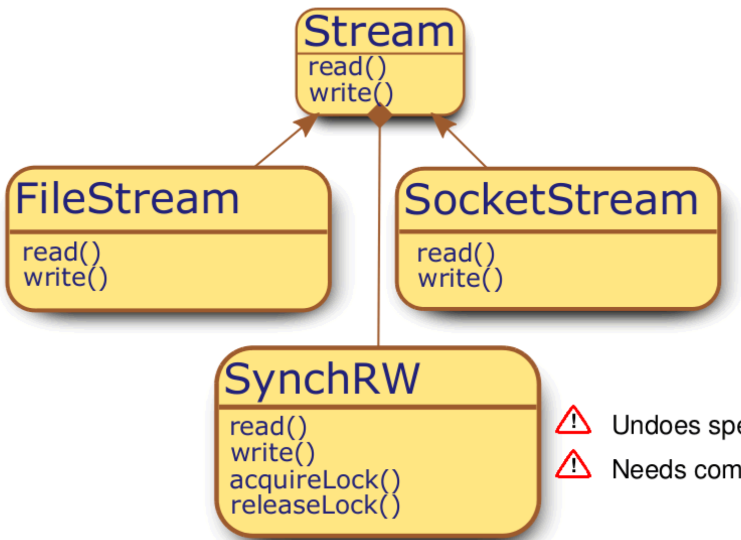
The Wrapper



⚠ Cannot inherit from both seperately

⚠ Creating new wrapping Classes duplicates code

The Wrapper



⚠ Undoes specialization

⚠ Needs common ancestor

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 $\in \mathbb{N}^+$
- $\bot$ (yet) undefined
- $\top$ in conflict

and the partial order $\bot \sqsubseteq m \sqsubseteq \top$ 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 $\forall_{n \in \text{pre}(c)} \, . \, c(n) = \bot$.

(concrete) class iff $\forall_{n \in \text{pre}(c)} \, . \, \bot \sqsubset c(n) \sqsubset \top$.

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:

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

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

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

$c_1(n) = \bot$

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 $\in \mathbb{N}^+$
- $\bot$ (yet) undefined
- $\top$ in conflict

and the partial order $\bot \sqsubseteq m \sqsubseteq \top$ 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 $\forall_{n \in \text{pre}(c)} \, . \, c(n) = \bot$.

(concrete) class iff $\forall_{n \in \text{pre}(c)} \, . \, \bot \sqsubset c(n) \sqsubset \top$.

Mixins Mixins Modelling Inheritance 9 / 30

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:

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

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

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

$c_1(n) = \bot$

Mixins Mixins Modelling Inheritance 10 / 30

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 $\triangleright : \mathcal{C} \times \mathcal{C} \mapsto \mathcal{C}$, definied by
$c_1 \triangleright c_2 = \{\mathtt{super} \mapsto c_2\} \,⩏\, (c_1 \,⩏\, c_2)$

Example: `Doors`

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

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

$$= \{\mathtt{super} \mapsto Door\} \,⩏\, (\{canOpen \mapsto 0x4204711\} \,⩏\, Door)$$

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

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
$c_1 \triangleleft c_2 = \{\mathtt{inner} \mapsto c_1\} \,⩏\, (c_2 \,⩏\, c_1)$

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."; };
};
```


Extension: Attributes

Remark: Modelling attributes is not in our main focus. Anyway, most mainstream languages nowadays are designed so that attributes are not overwritten:

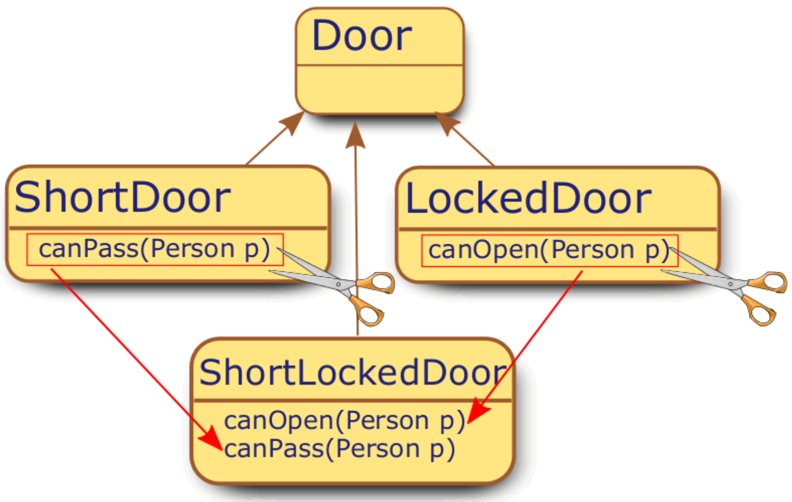
Definition (Mainstream inheritance ($\triangleright'$))

The extended mainstream inheritance $\triangleright' : \mathcal{C} \times \mathcal{C} \mapsto \mathcal{C}$ binds attributes statically:

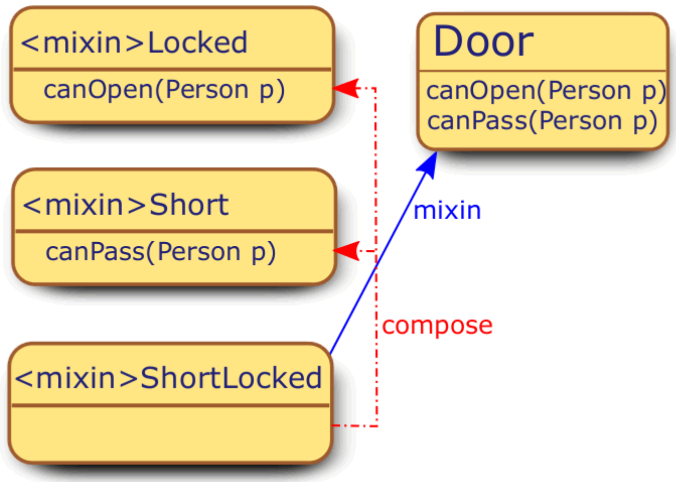
$$(c_1 \triangleright' c_2)(n) = \begin{cases} c_2 & \text{if } n = \mathtt{super} \\ \top & \text{if } n \in \mathsf{pre}(c_1) \ \wedge\ c_2(n) \in \mathbb{N}^+ \\ c_1(n) & \text{if } n \in \mathsf{pre}(c_1) \\ c_2(n) & \text{otherwise} \end{cases}$$

“So what do we really want?”

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);
```


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:

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

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

Example: Doors

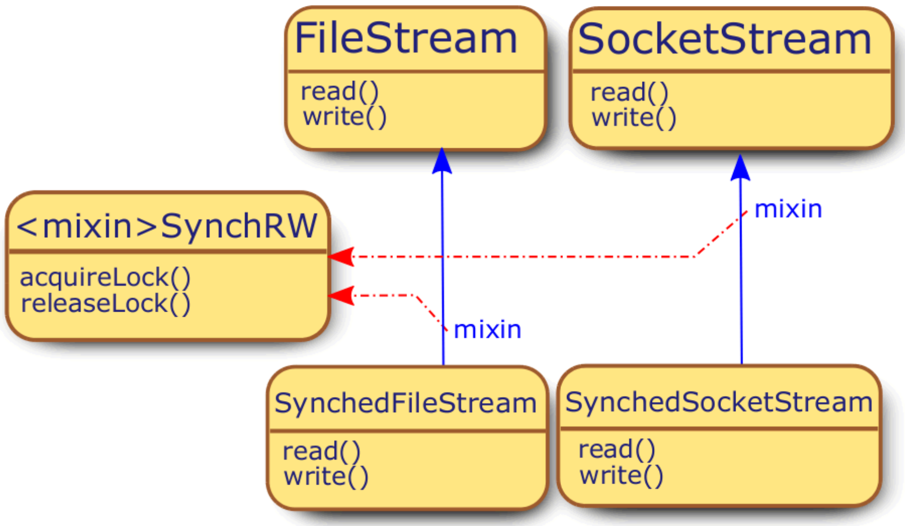
$$Locked = \{canOpen \mapsto 0x1234\}$$

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

$$Composed = mixin(Short) \circ (mixin(Locked)) = \lambda x \,.\, Short \triangleright (Locked \triangleright x)$$

$$= \lambda x \,.\, (\{\texttt{super} \mapsto Locked\} \sqcup (\{canOpen \mapsto 0x1234, canPass \mapsto 0x4711\} \triangleright x))$$

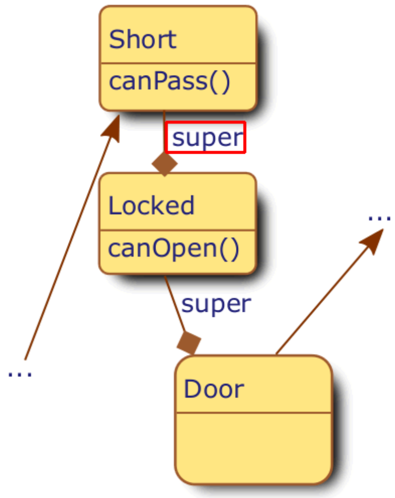
Wrapper with Mixins



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>> {};
```


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

“Ok, ok, show me a language with native mixins!”

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

Further reading...

Gilad Bracha and William Cook.
Mixin-based inheritance.
European conference on object-oriented programming on Object-oriented programming systems, languages, and applications (OOPSLA/ECOOP), 1990.

James Britt.
Ruby 2.1.5 core reference, December 2014.
URL https://www.ruby-lang.org/en/documentation/.

Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen.
Classes and mixins.
Principles of Programming Languages (POPL), 1998.