Script generated by TTT

Title: Simon: Programmiersprachen (10.01.2014)

Date: Fri Jan 10 14:16:12 CET 2014

Duration: 76:13 min

Pages: 28

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

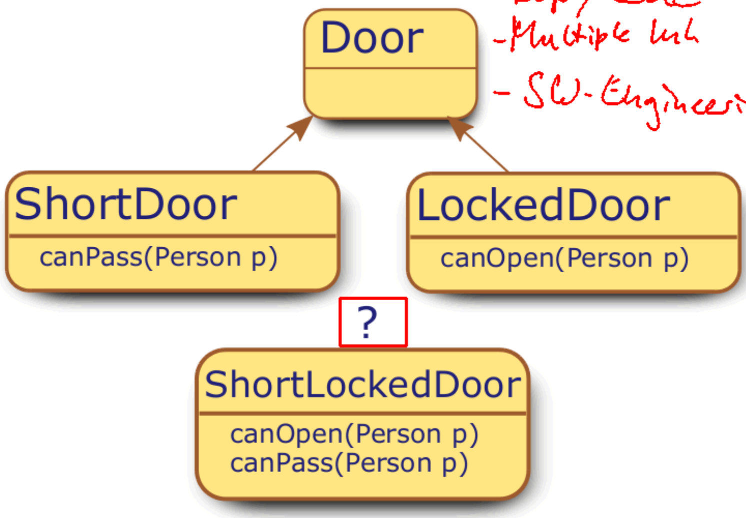
Mixins

Dr. Axel Simon and Dr. Michael Petter
Winter term 2013

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

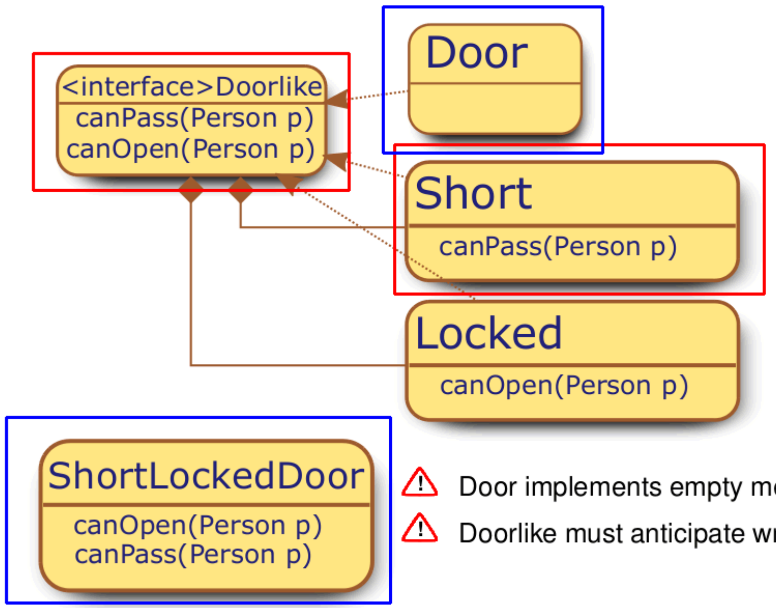
Outline

The Adventure Game



- Kopy Code
- Multiple Inh
- SW-Engineering !

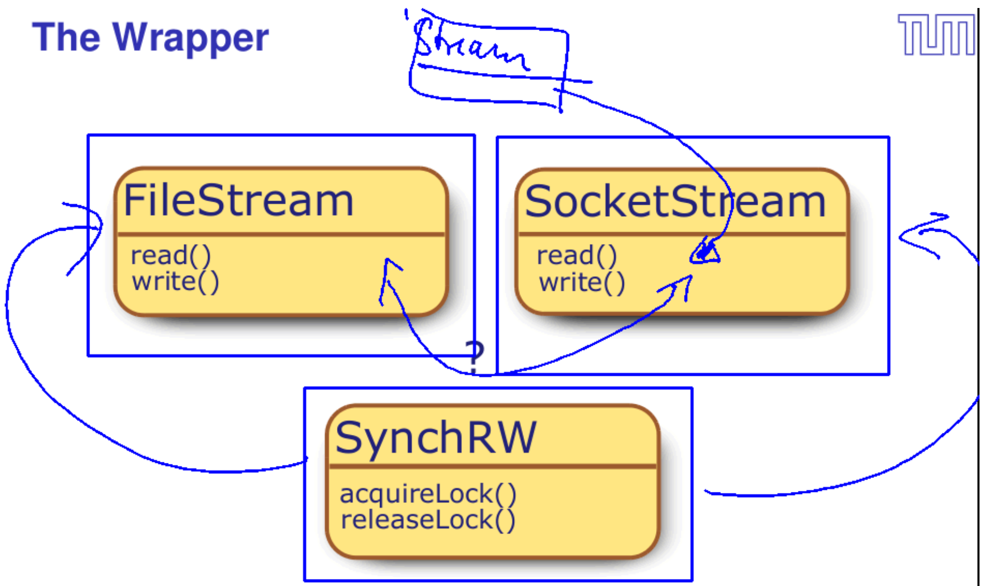
The Adventure Game



⚠ Door implements empty methods

⚠ Doorlike must anticipate wrappers

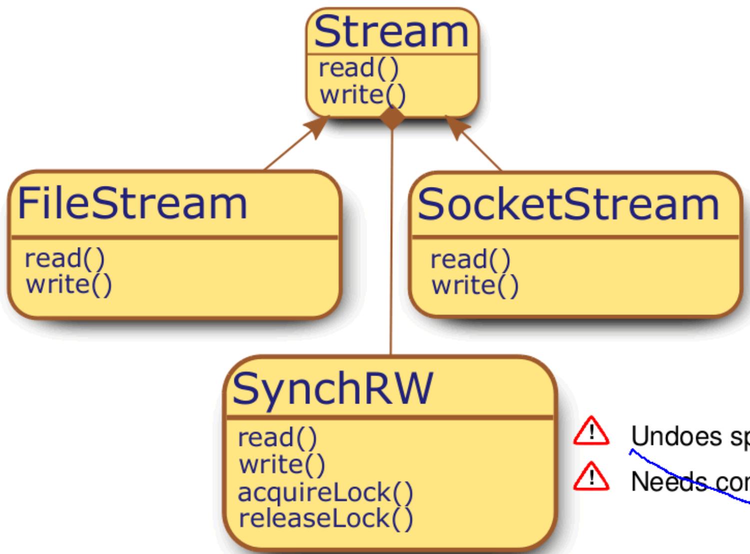
The Wrapper



⚠ 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"

Abstract model for Smalltalk-Inheritance

Smalltalk inheritance is the archetype for inheritance in mainstream languages like Java or C#.

- Types of Classes abstracted to maps from identifiers to qualified methods
- $\Delta(P) \rightsquigarrow$ the increment Δ can delegate calls to its parent P via keyword (in Smalltalk via `super`)
- The combination operator $\oplus$ merges maps, prefering the left argument

In Smalltalk-like inheritance, a subtype clause is defined as $\Delta \triangleright P = \Delta(P) \oplus P$

Example: `Doors`

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

$$LockedDoor = (\{canOpen \mapsto LockedDoor.canOpen\}(Door)) \oplus Door$$

Excursion: Beta-Inheritance

Beta-style inheritance is designed to provide security from replacement of a method by a different method.

- methods in parent overwrite methods in subclass
- `inner` as keyword to delegate control to subclass ($\rightsquigarrow$ `super`)
- $\rightsquigarrow$ parent arranges the exact spot, where the subclass can take over

Example (equivalent syntax):

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

In Beta-like Inheritance, a subtype clause is defined as $\boxed{\Delta \triangleleft P} = \boxed{P(\boxed{\Delta})} \oplus \boxed{\Delta}$

Excursion: CLOS-Inheritance

CLOS(Common Lisp Object System)-style inheritance offers multiple implementation inheritance featuring linearization.

- methods in childs overwrite methods in parents
- `super` as keyword to delegate control to direct parent ($\rightsquigarrow$ linearization)

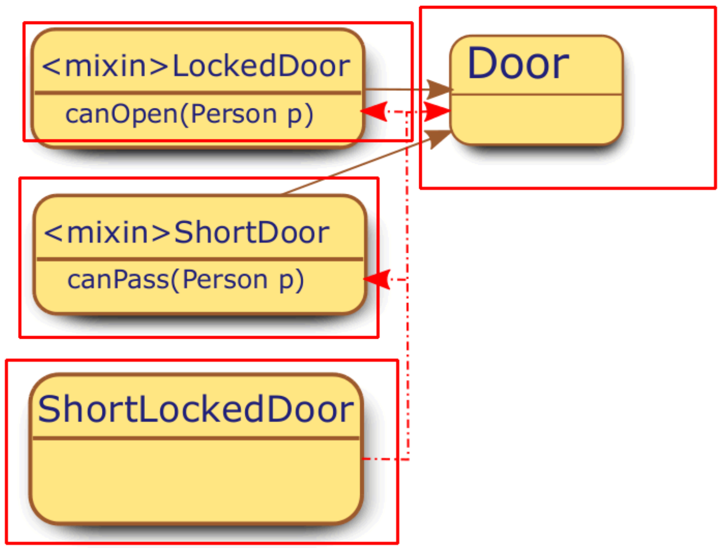
Example (equivalent syntax):

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

CLOS-like Multiple-Inheritance sybtype clause: $\boxed{\Delta_1 \triangleright (\Delta_2 \triangleright (\ldots \triangleright P)}\ldots)$

"So what do we really want?"

Adventure Game with Mixins



Adventure Game with Mixins

```
class Door {
  boolean canOpen(Person p) { return true; };
  boolean canPass(Person p) { return true; };
}
mixin Locked extends Door {
  boolean canOpen(Person p){
    if (!p.hasItem(key)) return false; else return super.canOpen(p);
  }
}
mixin Short extends Door {
  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 compose Locked;
class ShortLockedDoor  = Short(Locked(Door));
class ShortLockedDoor2 = ShortLocked(Door);
```


Back to the blackboard!

Abstract model for Mixins

Mixin

A Mixin M is a *curried second order type operation*, more precisely: Let Δ be a constant map from identifiers to fully qualified identifiers, then M is

$$\backslash\lambda.\Delta \triangleright \lambda = \backslash\lambda.\Delta(\lambda) \oplus \lambda$$

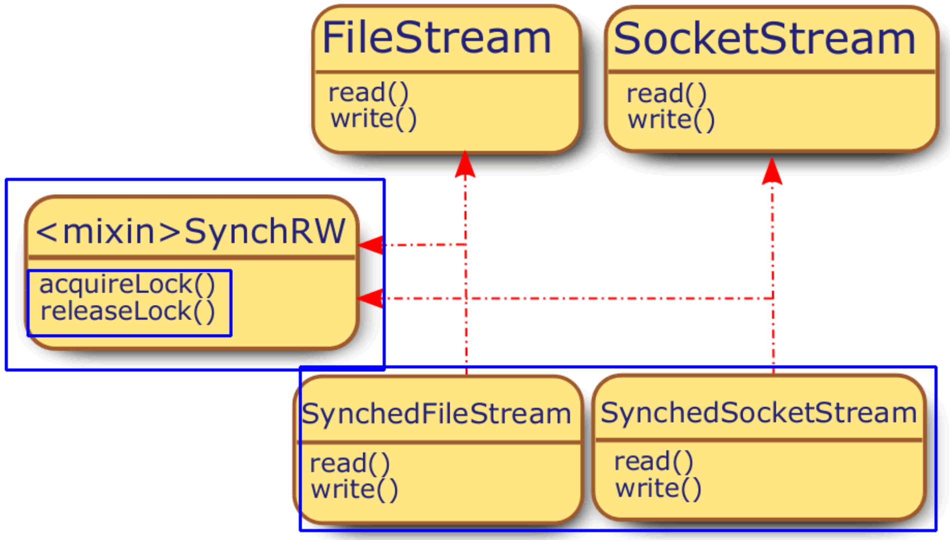
Example: `Doors`

$$\boxed{Door} = \{canPass \mapsto Door.canPass, canOpen \mapsto Door.canOpen\}$$
$$Locked = \{canOpen \mapsto Locked.canOpen\}$$
$$Short = \{canPass \mapsto Locked.canPass\}$$
$$ShortLocked_M = Short_M\ Locked_M = \boxed{\boxed{\backslash\lambda}\ Short \triangleright (Locked \triangleright \lambda)}$$

Let Δ be the Mixin Ms map. Then, M is connected to a class C via parameter binding:

$$MC = \Delta \triangleright C = \Delta(C) \oplus C$$

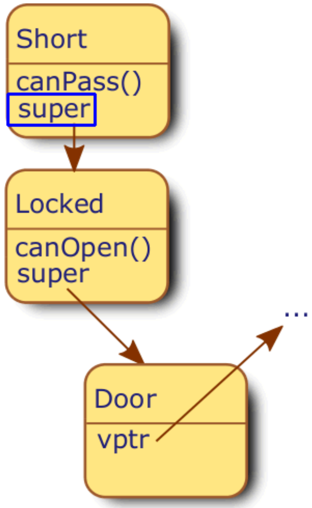
Wrapper with Mixins



Implementing Mixins

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

ShortDoor d
    = new ShortLockedDoor();
```



⚠ `super`-References not statically resolvable

"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 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

"Ok, ok, show me a language with native mixins!"

Ruby

```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
```

```ruby
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)
```


Lessons Learned

Lessons Learned

1. Formalisms to model inheritance
2. Mixins provide soft multiple inheritance
3. Multiple inheritance can not compensate `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.
- 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.
- Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen.
  Classes and mixins.
  Principles of Programming Languages (POPL), 1998.
- Brian Goetz.
  Interface evolution via virtual extension methods.
  JSR 335: Lambda Expressions for the Java Programming Language, 2011.
- Anders Hejlsberg, Scott Wiltamuth, and Peter Golde.
  C# Language Specification.
  Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
  ISBN 0321154916.