TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Mixins and Traits

Dr. Michael Petter
Winter 2019/20

What modularization techiques are there besides multiple implementation inheritance?

Outline

Design Problems

1. Inheritance vs Aggregation
2. (De-)Composition Problems

Inheritance in Detail

1. A Model for single inheritance
2. Inheritance Calculus with Inheritance Expressions
3. Modeling Mixins

Mixins in Languages

1. Simulating Mixins
2. Native Mixins

Cons of Implementation Inheritance

1. Lack of finegrained Control
2. Inappropriate Hierarchies

A Focus on Traits

1. Separation of Composition and Modeling
2. Trait Calculus

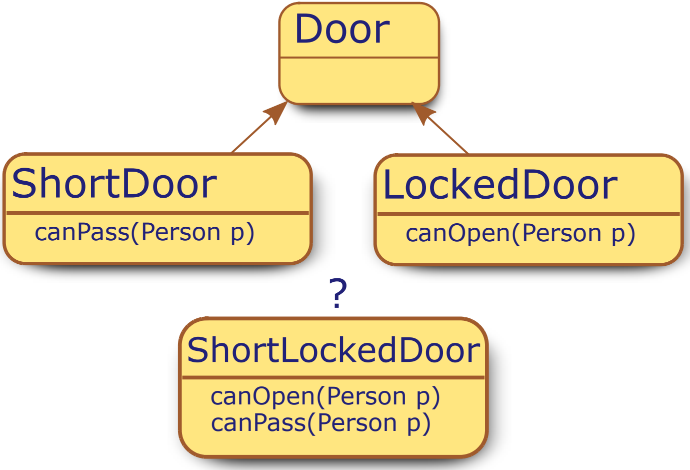
Traits in Languages

1. (Virtual) Extension Methods
2. Squeak

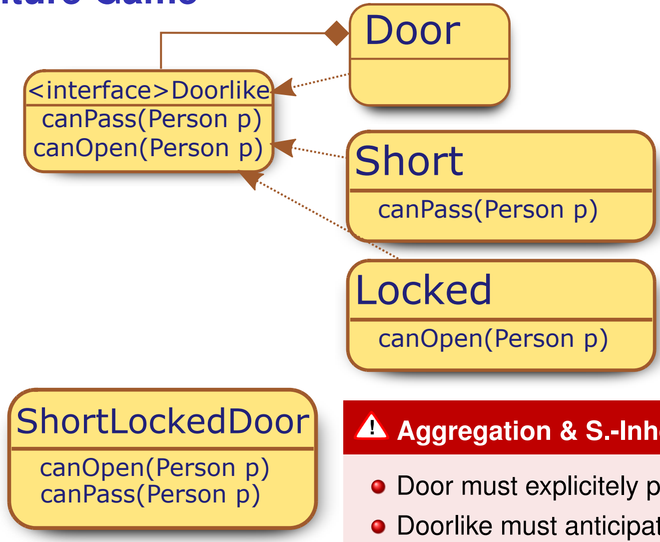
Reusability $\equiv$ Inheritance?

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

The Adventure Game



The Adventure Game

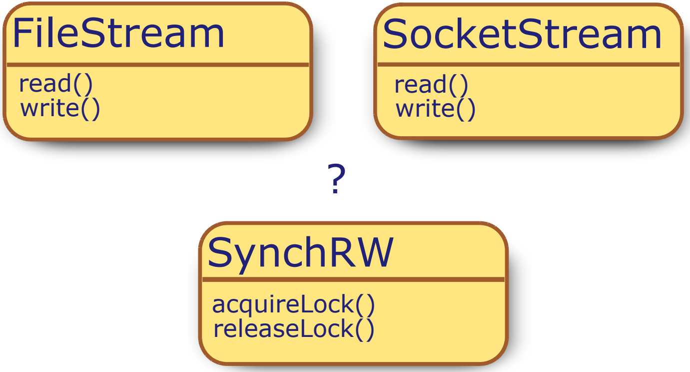


⚠ Aggregation & S.-Inheritance

- Door must explicitely provide chaining
- Doorlike must anticipate wrappers

⇒ **Multiple Inheritance ✓**

The Wrapper



FileStream

read()
write()

SocketStream

read()
write()

?

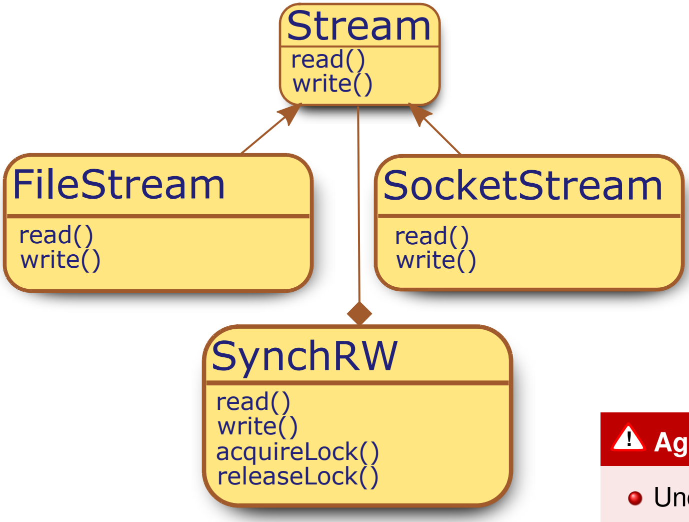
SynchRW

acquireLock()
releaseLock()

⚠ Unclear relations

⤳ Cannot inherit from both in turn with Multiple Inheritance (*Many-to-One* instead of *One-to-Many* Relation)

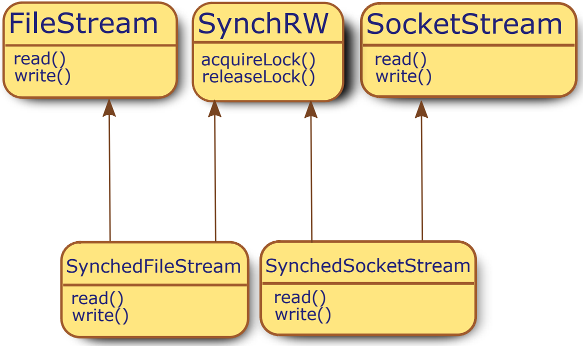
The Wrapper – Aggregation Solution



⚠ Aggregation

- Undoes specialization
- Needs common ancestor

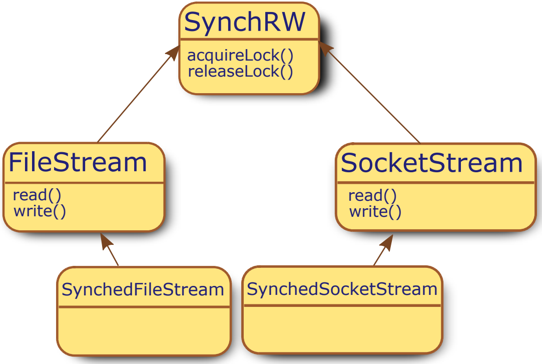
The Wrapper – Multiple Inheritance Solution



⚠ Duplication

With multiple inheritance, `read/write` Code is essentially *identical but duplicated for each particular wrapper*

Fragility



⚠ Inappropriate Hierarchies

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

(De-)Composition Problems

All the problems of

- Relation
- Duplication
- Hierarchy

are centered around the question

“How do I distribute functionality over a hierarchy”

⇝ *functional (de-)composition*

Classes and Methods

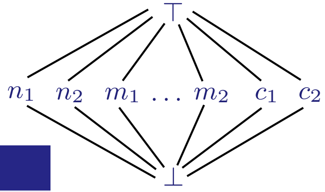
The building blocks for classes are

- 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 $\in \mathcal{C}$
- $\perp$ *abstract*
- $\top$ *in conflict*

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



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

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

Definition (Interface and Class)

A class c is called (with pre beeing the preimage)

interface iff $\forall_{n \in \text{pre}(c)} \,.\, c(n) = \perp$.

abstract class iff $\exists_{n \in \text{pre}(c)} \,.\, c(n) = \perp$.

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

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 classes $\mathcal{C} := \mathcal{N} \mapsto \mathcal{B}$.

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

- 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 $\Lsh\!\sqcup$, defined componentwise:

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

Example: Smalltalk-Inheritance

Smalltalk inheritance

- children's methods dominate parents' methods
- 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
$c_1 \triangleright c_2 = \{\mathtt{super} \mapsto c_2\} \mathbin{\Lsh\!\sqcup} (c_1 \mathbin{\Lsh\!\sqcup} c_2)$

Example: `Doors`

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

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

$$= \{\mathtt{super} \mapsto Door\} \mathbin{\Lsh\!\sqcup} (\{canOpen \mapsto 0x4204711\} \mathbin{\Lsh\!\sqcup} Door)$$

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

Excursion: Beta-Inheritance

In *Beta*-style inheritance

- the design goal is to provide security wrt. 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

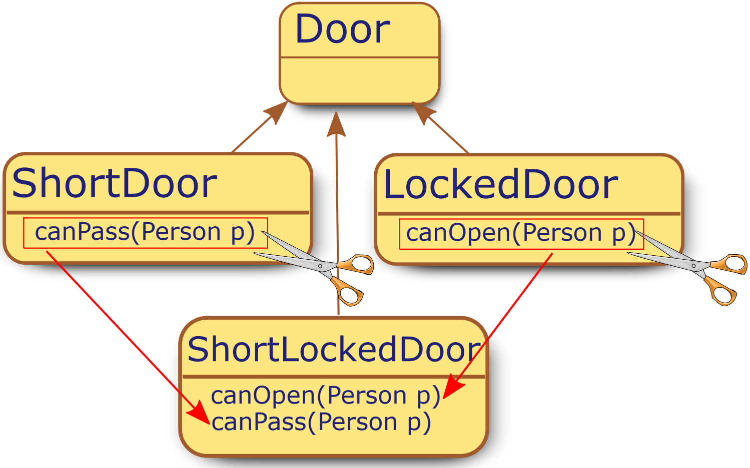
$$c_1 \triangleleft c_2 = \{\texttt{inner} \mapsto c_1\} \Lsh\!\sqcup (c_2 \Lsh\!\sqcup c_1)$$

Example (equivalent syntax):

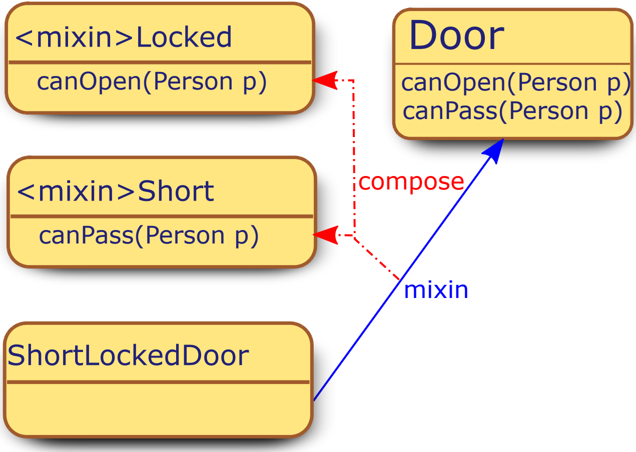
```
class Person {
  String name ="Axel Simon";
  public String toString(){ return name+inner.toString();};
};
class Graduate extends Person {
  public extension String toString(){ return ", Ph.D."; };
};
```

So what do we really want?

Adventure Game with Code Duplication



Adventure Game with Mixins



<mixin>Locked

canOpen(Person p)

<mixin>Short

canPass(Person p)

ShortLockedDoor

Door

canOpen(Person p)
canPass(Person p)

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:

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

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

Example: `Doors`

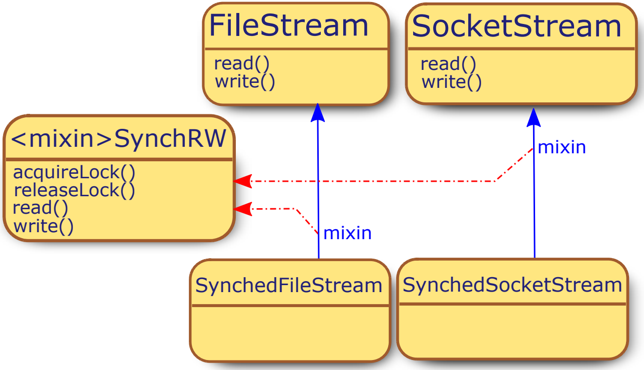
$$Locked = \{canOpen \mapsto 0x1234\}$$

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

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

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

Wrapper with Mixins

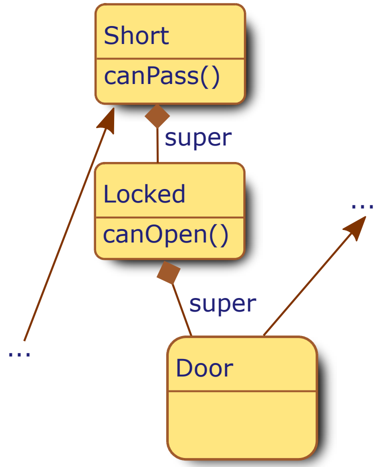


Mixins for wrappers

- avoids duplication of `read/write` code
- keeps specialization
- even compatible to single inheritance systems

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++

```cpp
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
- Composable mixins
- Hassle-free simple alternative to multiple inheritance

C++ Mixins

- Mixins reduced to templated superclasses
- Can be seen as coding pattern
- C++ Type system not modular

⇝ Mixins have to stay source code

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
```

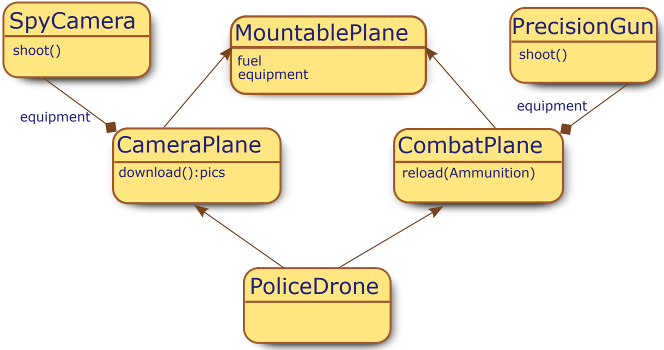
```
module ShortLocked
  include Short
  include Locked
end
class Person
  attr_accessor :size
  def initialize
    @size = 160
  end
  def hasKey
    true
  end
end

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

puts d.canPass(p)
```

Is Inheritance the Ultimate Principle in Reusability?

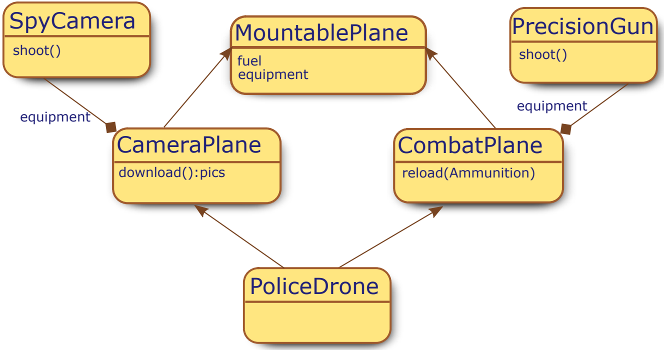
Lack of Control



⚠ Control

- Common base classes are shared or duplicated at class level

Lack of Control

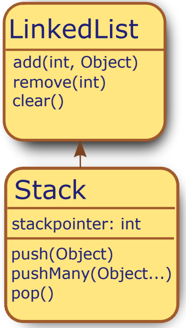


⚠ Control

- Common base classes are shared or duplicated at class level
- `super` as ancestor reference vs. qualified specification

⇝ No *fine-grained specification* of duplication or sharing

Inappropriate Hierachies



LinkedList

add(int, Object)
remove(int)
clear()

Stack

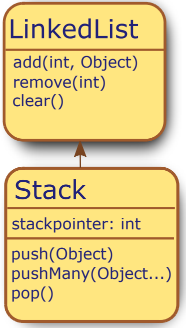
stackpointer: int

push(Object)
pushMany(Object...)
pop()

⚠ Inappropriate Hierarchies

- High up specified methods *turn obsolete*, but there is no statically safe way to remove them

Inappropriate Hierachies



⚠ Inappropriate Hierarchies

- High up specified methods *turn obsolete*, but there is no statically safe way to remove them

⚠ Liskov Substitution Principle!

Is Implementation Inheritance even an *Anti-Pattern*?

Excerpt from the Java 8 API documentation for class `Properties`:

> *"Because `Properties` inherits from `Hashtable`, the `put` and `putAll` methods can be applied to a `Properties` object. Their use is strongly discouraged as they allow the caller to insert entries whose keys or values are not `Strings`. The `setProperty` method should be used instead. If the `store` or `save` method is called on a "compromised" `Properties` object that contains a non-`String` key or value, the call will fail…"*

Excerpt from the Java 8 API documentation for class `Properties`:

> *"Because `Properties` inherits from `Hashtable`, the `put` and `putAll` methods can be applied to a `Properties` object. Their use is strongly discouraged as they allow the caller to insert entries whose keys or values are not `Strings`. The `setProperty` method should be used instead. If the `store` or `save` method is called on a "compromised" `Properties` object that contains a non-`String` key or value, the call will fail..."*

⚠ Misuse of Implementation Inheritance

Implementation Inheritance itself as a pattern for code reusage is often misused!
⇝ All that is not explicitely prohibited will eventually be done!

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 *composing* functionality.

⇝ Use interfaces to design hierarchical signature propagation

⇝ Use *traits* as modules for assembling functionality

⇝ Use classes as frames for entities, which can create objects

Traits – Composition

Definition (Trait $\in \mathcal{T}$)

A class t is without attributes is called *trait*.

The *trait sum* $+ : \mathcal{T} \times \mathcal{T} \mapsto \mathcal{T}$ is the componentwise least upper bound:

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

Trait-Expressions also comprise:

- *exclusion* $- : \mathcal{T} \times \mathcal{N} \mapsto \mathcal{T}$: $\quad (t - a)(n) = \begin{cases} \text{undef} & \text{if } a = n \\ t(n) & \text{otherwise} \end{cases}$
- *aliasing* $[\rightarrow] : \mathcal{T} \times \mathcal{N} \times \mathcal{N} \mapsto \mathcal{T}$: $\quad t[a \rightarrow b](n) = \begin{cases} t(n) & \text{if } n \neq a \\ t(b) & \text{if } n = a \end{cases}$

Traits t can be connected to classes c by the asymmetric join:

$$(c \overset{\leftarrow}{\sqcup} t)(n) = \begin{cases} c(n) & \text{if } n \in \text{pre}(c) \\ t(n) & \text{otherwise} \end{cases}$$

Usually, this connection is reserved for the last composition level.

Traits – Concepts

Trait composition principles

Flat ordering	All traits have the same precedence under $+$ $\rightsquigarrow$ explicit disambiguation with aliasing and exclusion
Precedence	Under asymmetric join $\overleftarrow{\sqcup}$, class methods take precedence over trait methods
Flattening	After asymmetric join $\overleftarrow{\sqcup}$: Non-overridden trait methods have the same semantics as class methods

⚠ Conflicts . . .

arise if composed traits map methods with identical names to different bodies

Conflict treatment

- ✓ Methods can be aliased ($\rightarrow$)
- ✓ Methods can be excluded ($-$)
- ✓ Class methods override trait methods and sort out conflicts ($\overleftarrow{\sqcup}$)

Can we augment classical languages by traits?

Extension Methods (C#)

Central Idea:

Uncouple method definitions from class bodies.

Purpose:

- retrospectively add methods to complex types
  ⇝ *external definition*
- especially provide definitions of *interface methods*
  ⇝ poor man's multiple inheritance!

Syntax:

1. Declare a static class with definitions of static methods
2. Explicitely declare first parameter as receiver with modifier `this`
3. Import the carrier class into scope (if needed)
4. Call extension method in *infix form* with emphasis on the receiver

```
public class Person{
 public int size = 160;
 public bool hasKey() { return true;}
}
public interface Short {}
public interface Locked {}
public static class DoorExtensions {
 public static bool canOpen(this Locked leftHand, Person p){
  return p.hasKey();
 }
 public static bool canPass(this Short leftHand, Person p){
  return p.size<160;
 }
}
public class ShortLockedDoor : Locked,Short {
 public static void Main() {
  ShortLockedDoor d = new ShortLockedDoor();
  Console.WriteLine(d.canOpen(new Person()));
 }
}
```

Extension Methods as Traits

Extension Methods

- transparently extend arbitrary types externally
- provide quick relief for plagued programmers

. . . but not traits

- Interface declarations empty, thus kind of purposeless
- Flattening not implemented
- Static scope only

Static scope of extension methods causes unexpected errors:

```
public interface Locked {
  public bool canOpen(Person p);
}
public static class DoorExtensions {
 public static bool canOpen(this Locked leftHand, Person p){
  return p.hasKey();
 }
}
```

Extension Methods as Traits

Extension Methods

- transparently extend arbitrary types externally
- provide quick relief for plagued programmers

. . . but not traits

- Interface declarations empty, thus kind of purposeless
- Flattening not implemented
- Static scope only

Static scope of extension methods causes unexpected errors:

```
public interface Locked {
  public bool canOpen(Person p);
}
public static class DoorExtensions {
 public static bool canOpen(this Locked leftHand, Person p){
  return p.hasKey();
 }
}
```

Virtual Extension Methods (Java 8)

Java 8 advances one step further:

```
interface Door {
  boolean canOpen(Person p);
  boolean canPass(Person p);
}
interface Locked {
  default boolean canOpen(Person p) { return p.hasKey(); }
}
interface Short {
  default boolean canPass(Person p) { return p.size<160; }
}
public class ShortLockedDoor implements Short, Locked, Door {
}
```

Implementation

…consists in adding an interface phase to `invokevirtual`’s name resolution

⚠ Precedence

Still, default methods do not override methods from *abstract classes* when composed

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 the language with real traits?!

Squeak

Smalltalk

Squeak is a smalltalk implementation, extended with a system for traits.

Syntax:

- `name: param1 and: param2`
  declares method `name` with `param1` and `param2`
- `| ident1 ident2 |`
  declares Variables `ident1` and `ident2`
- `ident := expr`
  assignment
- `object name:content`
  sends message `name` with `content` to `object` (≡ call: `object.name(content)`)
- `.`
  line terminator
- `^ expr`
  return statement

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.
```

Disambiguation

Traits vs. Mixins vs. Class-Inheritance

All different kinds of type expressions:

- Definition of curried *second order type operators* + Linearization

Explicitly: Traits differ from Mixins

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

Disambiguation

Traits vs. Mixins vs. Class-Inheritance

All different kinds of type expressions:

- Definition of curried *second order type operators* + Linearization
- Finegrained flat-ordered *composition of modules*

Explicitly: Traits differ from Mixins

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

Disambiguation

Traits vs. Mixins vs. Class-Inheritance

All different kinds of type expressions:

- Definition of curried *second order type operators* + Linearization
- Finegrained flat-ordered *composition of modules*
- Definition of (local) partial order on precedence of types wrt. MRO

Explicitly: Traits differ from Mixins

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

Disambiguation

Traits vs. Mixins vs. Class-Inheritance

All different kinds of type expressions:

- Definition of curried *second order type operators* + Linearization
- Finegrained flat-ordered *composition of modules*
- Definition of (local) partial order on precedence of types wrt. MRO
- Combination of principles

Explicitly: Traits differ from Mixins

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

Lessons learned

Mixins

- Mixins as *low-effort* alternative to multiple inheritance
- Mixins lift type expressions to *second order type expressions*

Traits

- Implementation Inheritance based approaches leave room for improvement in modularity in real world situations
- Traits offer *fine-grained control* of composition of functionality
- Native trait languages offer *separation of composition* of functionality from *specification* of interfaces

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/`.

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.

Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black.
Traits: Composable units of behaviour.
European Conference on Object-Oriented Programming (ECOOP), 2003.