“What advanced techniques are there besides multiple implementation inheritance?”
The Adventure Game

Door

ShortDoor
- canPass(Person p)
- canOpen(Person p)

LockedDoor
- canPass(Person p)
- canOpen(Person p)

ShortLockedDoor
- canOpen(Person p)
- canPass(Person p)

<interface>Doorlike
- canPass(Person p)
- canOpen(Person p)

Short
- canPass(Person p)

Locked
- canOpen(Person p)

⚠️ Door implements empty methods
⚠️ Doorlike must anticipate wrappers

The Wrapper

FileStream
- read()
- write()

SocketStream
- read()
- write()

SynchRW
- acquireLock()
- releaseLock()

⚠️ Cannot inherit from both separately
⚠️ Creating new wrapping Classes duplicates code

⚠️ 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 \( \in \mathbb{N}^+ \)
- \( \perp \) (yet) undefined
- \( \top \) in conflict
and the partial order \( \perp \subseteq m \subseteq \top \) for each \( m \in \mathcal{B} \)

Definition (Abstract Class \( \in \mathcal{C} \))
A partial function \( c : \mathcal{N} \rightarrow \mathcal{B} \) is called abstract class.

Definition (Interface and Class)
An abstract class \( c \) is called
- **interface** if \( \forall n \in \text{pre}(c) \cdot c(n) = \perp \).
- **(concrete) class** if \( \forall n \in \text{pre}(c) \cdot \perp \subseteq c(n) \subseteq \top \).

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 \( \mathcal{C} := \mathcal{N} \rightarrow \mathcal{B} \).

Several possibilities for composing maps \( \mathcal{C} \sqcap \mathcal{C} \):
- the symmetric join \( \sqcup \), defined componentwise:
  \[
  (c_1 \sqcup c_2)(n) = b_1 \sqcup b_2 = \begin{cases} 
  b_2 & \text{if } b_1 = \perp \\
  b_1 & \text{if } b_2 = \perp \\
  b_2 & \text{if } b_1 = b_2 \\
  \top & \text{otherwise} 
  \end{cases} 
  \]
  where \( b_i = c_i(n) \)
- in contrast, the asymmetric join \( \sqcup \), defined componentwise:
  \[
  (c_1 \sqcup c_2)(n) = \begin{cases} 
  c_1(n) & \text{if } n \in \text{pre}(c_1) \\
  c_2(n) & \text{otherwise} 
  \end{cases} 
  \]

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 \( \perp \subseteq m \subseteq \top \) for each \( m \in \mathcal{B} \)

Definition (Abstract Class \( \in \mathcal{C} \))
A partial function \( c : \mathcal{N} \rightarrow \mathcal{B} \) is called abstract class.

Definition (Interface and Class)
An abstract class \( c \) is called
- **interface** if \( \forall n \in \text{pre}(c) \cdot c(n) = \perp \).
- **(concrete) class** if \( \forall n \in \text{pre}(c) \cdot \perp \subseteq c(n) \subseteq \top \).
**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 (\(\supset\))**

Smalltalk inheritance is the binary operator \(\supset : C \times C \rightarrow C\), defined by

\[
c_1 \supset c_2 = \{\text{super} \mapsto c_2\} \cup (c_1 \cup c_2)
\]

**Example: Doors**

\[
\text{Door} = \{\text{canPass} \mapsto \top, \text{canOpen} \mapsto \bot\}
\]

\[
\text{LockedDoor} = \{\text{canOpen} \mapsto 0x4204711\} \supset \text{Door}
\]

\[
= \{\text{super} \mapsto \text{Door}\} \cup \{\text{canOpen} \mapsto 0x4204711\} \cup \text{Door}
\]

\[
= \{\text{super} \mapsto \text{Door}, \text{canOpen} \mapsto 0x4204711, \text{canPass} \mapsto \bot\}
\]

**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 explicitly delegates control to the subclass

**Definition (Beta inheritance (\(<\))**

Beta inheritance is the binary operator \(< : C \times C \rightarrow C\), defined by

\[
c_1 < c_2 = \{\text{inner} \mapsto c_1\} \cup (c_2 \cup c_1)
\]

**Example (equivalent syntax):**

```java
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 (\(')\))**

The extended mainstream inheritance \(\cdot : C \times C \rightarrow C\) binds attributes statically:

\[
(c_1 \cdot c_2)(n) =
\begin{cases}
  c_2 & \text{if } n = \text{super} \\
  \top & \text{if } n \in \text{pre}(c_1) \land c_2(n) \in \mathbb{N}^+
\end{cases}
\]

\[
\begin{cases}
  c_1(n) & \text{if } n \in \text{pre}(c_1) \\
  c_2(n) & \text{otherwise}
\end{cases}
\]

“So what do we really want?”
**Adventures Game with Code Duplication**

```
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 ◦ Locked;
class ShortLockedDoor = Short(Locked(Door));
class ShortLockedDoor2 = ShortLocked(Door);
```

**Adventures Game with Mixins**

```
< mixin> Locked
  canOpen(Person p)
  canPass(Person p)
< mixin> Short
  canPass(Person p)
< mixin> ShortLocked
Door
  canOpen(Person p)
  canPass(Person p)
```

**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 $\text{mixin}: [C] \to [C \to C]$ is a unary class function, creating a unary class operator, defined by:

$$\text{mixin}(c) = \lambda x. c \circ x$$

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

**Example: Doors**

Locked = \{canOpen \mapsto 0x1234\}
Short = \{canPass \mapsto 0x4711\}

Composed = \text{mixin}(Short) [\text{mixin}(Locked)] = \lambda x. Short \circ (Locked \circ x)
= \lambda x. super \Rightarrow Locked \cup \{canOpen \mapsto 0x1234, canPass \mapsto 0x4711\} \circ 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

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>> {};
```

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
- 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

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

```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

module ShortLocked
  include Short
  include Locked
end

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

class ShortLockedDoor < Door
  include Short
  include Locked
  def canOpen(p)
    p.size < 160 and super(p)
  end
end

p = Person.new
p.puts p.canPass(p)
d = Door.new
d.extend ShortLocked

d.puts d.canPass(p)
```

Lessons Learned

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

Further reading...