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

Door
- Copy Code
- Multiple Links
- SW Engineering

ShortDoor
- canPass(Person p)

LockedDoor
- canOpen(Person p)

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

The Wrapper

FileStream
- read()
- write()

SocketStream
- read()
- write()

SynchRW
- acquireLock()
- releaseLock()

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

The Adventure Game

Door

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

Short
- canPass(Person p)

Locked
- canOpen(Person p)

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

⚠️ Door implements empty methods
⚠️ Doorlike must anticipate wrappers

The Wrapper

FileStream
- read()
- write()

SocketStream
- read()
- write()

SynchRW
- read()
- write()
- acquireLock()
- releaseLock()

⚠️ Undoes specialization
⚠️ Needs common ancestor
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
- $\Delta(P) \rightsquigarrow \Delta$ the increment $\Delta$ can delegate calls to its parent $P$ via keyword (in Smalltalk via `super`)
- The combination operator $\oplus$ merges maps, preferring the left argument

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

**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):**

```java
class Person {
    String name = "Axel Simon";
    public virtual String toString() { return name + inner(); };
}

class Graduate extends Person {
    public extended String toString() { return "", Ph.D."; }
}

class Doctor extends Person {
    public String toString() { return "Dr. " + super.toString(); }
}

class ResearchingDoctor extends Doctor, Graduate
```

In Beta-like Inheritance, a subtype clause is defined as $\Delta \circ P = P \Delta \oplus \Delta$

Excursion: CLOS-Inheritance

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

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

**Example (equivalent syntax):**

```java
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 subtype clause: $\Delta_1 \triangleright (\Delta_2 \triangleright (\ldots \triangleright P \ldots))$
“So what do we really want?”

```
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(LockedDoor);
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 \( \Delta \) be a constant map from identifiers to fully qualified identifiers, then \( M \) is

\[
\lambda.\Delta \triangleright \lambda = \lambda.\Delta(\lambda) \triangleq \lambda
\]

**Example: Doors**

\[
\begin{align*}
Door &= \{ \text{canPass} \mapsto \text{Door.canPass}, \text{canOpen} \mapsto \text{Door.canOpen} \} \\
Locked &= \{ \text{canOpen} \mapsto \text{Locked.canOpen} \} \\
Short &= \{ \text{canPass} \mapsto \text{Locked.canPass} \} \\
ShortLocked_M &= \lambda. Short \triangleright (Locked \triangleright \lambda)
\end{align*}
\]

Let \( \Delta \) be the Mixin \( M \)'s map. Then, \( M \) is connected to a class \( C \) via parameter binding:

\[
MC = \Delta \triangleright C = \Delta(C) \triangleright C
\]

Implementing Mixins

```java
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();
```

Wrapper with Mixins

```
FileStream
  read()
  write()

SocketStream
  read()
  write()

<mixin>SynchRW
  acquireLock()
  releaseLock()

SynchedFileStream
  read()
  write()

SynchedSocketStream
  read()
  write()
```

“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

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)

Lessons Learned
- Formalisms to model inheritance
- Mixins provide soft multiple inheritance
- Multiple inheritance can not compensate super reference
- Full extent of mixins only when mixins are 1st class language citizens
Further reading...


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.

