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

Outline

Weak implementation inheritance
- Decorator Problem
- Wrapper Problem

Inheritance in Detail
- Models for single inheritance
- Introducing Mixins
- Modelling Mixins

Mixins in the wild
- Mixins as C++-Pattern
- Extension methods
- Native Mixins
The Adventure Game

Door

ShortDoor
  canPass(Person p)
  canOpen(Person p)

LockedDoor
  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
  acquireLock()
  releaseLock()

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

⚠️ Undo 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
- Subtypes are specified as increments $\Delta$ to their parents
- super calls are delegated to the parent
- $\rightsquigarrow$ Parent is connected to the increment as a parameter $\Delta(Parent)$
- Combination operator $\oplus$ merges operands, preferring the left argument

Smalltalk-like Inheritance is defined as $C = \Delta(P) \oplus P$

**Example: Doors**

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

$\quad\text{LockedDoor} = \{\text{canOpen} \mapsto \text{LockedDoor}.\text{canOpen}, \text{Door} \mapsto \text{Door}\}$

**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
- $\rightsquigarrow$ inner as keyword to delegate control to subclass (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."; };
};
```

Beta-like Inheritance is defined as $C(inner) = P(\Delta(inner)) \oplus \Delta(inner)$

$\rightsquigarrow$ Types in Beta are $\rightsquigarrow$ Lambda-Expressions

**Generalizing Beta- and Smalltalk-Inheritance**

We introduce the combination operator, which joins attributes and performs super/inner bindings:

$$A \triangleright B = A(\Delta) \oplus B$$

*Smalltalk* $C = \Delta \triangleright P$

*Beta* $\lambda.C(\lambda) = \lambda.(P \triangleright \Delta(\lambda))$

$\rightsquigarrow$ Both Systems differ only in the direction of growth (and the lambda-expression)
**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 (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 {}  // Mixin
```

CLOS-like Multiple-Inheritance: $C = \Delta_1 \triangleright \Delta_2 \triangleright \ldots \triangleright P \ldots$

---

**Adventures Game with Mixins**

```
< mixin > LockedDoor
  canOpen(Person p)

< mixin > ShortDoor
  canPass(Person p)
```

```
Door
```

```
< mixin > LockedDoor
  canOpen(Person p)

< mixin > ShortDoor
  canPass(Person p)
```

```
Door
```
**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 = ShortLocked(Door);
class ShortLockedDoor2 = ShortLocked(Door);

**Abstract model for Mixins**

Mixin Composition: \( \lambda (M_1 \ast M_2)(\lambda) = \lambda. M_1(M_2(\lambda) \oplus \lambda) \oplus (M_2(\lambda) \oplus \lambda) \)

**Example: Doors**

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

Standard classes are handled as degenerated Mixins, binding \( \emptyset \) as composite reference; Mixins are connected to classes via inheritance \( \triangleright \):

\[
M(\emptyset) = M(\emptyset) \oplus \emptyset = M \triangleright \emptyset
\]

**Example: Doors**

\[
\begin{align*}
    \text{ShortLockedDoor} &= (\text{Short} \ast \text{Locked}) \triangleright \text{Door} \\
    &= (\text{Short(Locked(Door)} \oplus \text{Door} \oplus \text{Locked(Door)}) \oplus \text{Door}
\end{align*}
\]

**Wrapper with Mixins**

- **FileStream**
  - read()
  - write()

- **SocketStream**
  - read()
  - write()

- **<mixin>SynchRW**
  - acquireLock()
  - releaseLock()

- **SynchedFileStream**
  - read()
  - write()

- **SynchedSocketStream**
  - read()
  - write()

**Types of Mixins**

**Subtype Relation** \( \lll \):

- \( \bot \lll \text{Object} \)
- \( T \lll T_1 \land T_1 \lll T_2 \land T_1 \lll T_3 \lll \text{Reflexively and transitively closed} \)

**Example: Doors**

\[
\begin{align*}
    \text{ShortLocked} &\lll \text{Locked} \land \text{ShortLocked} \lll \text{Short} \\
    \text{ShortLockedDoor} &\lll \text{Short} \land \text{ShortLockedDoor} \lll \text{Locked} \\
    \text{ShortLockedDoor} &\lll \text{ShortLocked} \land \text{ShortLockedDoor} \lll \text{ShortDoor}
\end{align*}
\]
Implementing Mixins

```java
class Door {
    boolean canOpen(Person p) {
        return true;
    }
    boolean canPass(Person p) {
        return true;
    }
}
.mixin Locked extends Door {
    boolean canOpen(Person p) {
        return true;
    }
}
.mixin Short extends Door {
    boolean canPass(Person p) {
        return true;
    }
}  

class ShortDoor {
    ShortDoor();
    class ShortLockedDoor {
        ShortLockedDoor();
    }
}

ShortDoor d = new ShortLockedDoor();
```

Programming Mixins

There are different ideas to bring mixins into daily programming:

- C++: Templates & Multiple Inheritance
- C#: Extension Methods
- Java: Aspect Orientation or Virtual Extension methods
- Ruby/Python: Native mixins

Simulating Mixins in C++

```cpp
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
};
```

"Surely multiple inheritance is powerful enough to simulate mixins?"
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

**Common properties of Mixins**
- Linearization is necessary
- ≈ Exact sequence of Mixins is relevant

Extension Methods (C#)

**Central Idea:**
Uncouple method definitions and implementations from class bodies.

**Purpose:**
- retrospectively add methods to complex types
- especially provide implementations for interface methods

**Syntax:**
1. Specify a static class with static methods
2. Explicitly specify receiver type as first first parameter with keyword `this`
3. Bring the carrier class into scope (if needed)
4. Call extension method in *infix form*
public class Person{
    public int size = 160;
    public bool hasKey() { return true; }
}
public interface Short {
}
public static class PersonExtensions {
    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 Mixins

Pro Extension Methods
- transparently extend arbitrary types
- for many cases offer enough flexibility

Contra Extension Methods
- Interface declarations empty, thus kind of purposeless
- Inherited properties always of higher priority than extensions
- Class-code is distributed over several class bodies
- Still no super reference

⚠ Limited scope of extension methods prohibits expected behaviour:
public interface Locked {
    public bool canOpen(Person p) {
        return p.hasKey();
    }
}
public static class DoorExtensions {
    public static bool canOpen(this Locked leftHand, Person p) {
        return p.hasKey();
    }
}

Excursion: Virtual Extension Methods (Java 8)

Project Lambda from the upcoming Java version advances one pace further:
interface Door {
    boolean canOpen(Person p);
    boolean canPass(Person p);
}
interface Locked extends Door {
    boolean canOpen(Person p) default { return p.hasKey(); }
}
interface Short extends Door {
    boolean canPass(Person p) default { return p.size < 160; }
}
public class ShortLockedDoor implements Short, Locked, Door {
}

Implementation
... consists in adding an interface phase to invokevirtual's name resolution

⚠ Polymorphic Overwriting
Still, default methods can not overwrite abstract methods from abstract classes

“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
  end
end

class ShortLockedDoor < Door
  include Short
  include Locked
end
```

Further reading...

- Stéphane Ducasse, Oscar Nierstrasz, Nathanaël Schaëli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2006.