What advanced techniques are there besides multiple implementation inheritance?
**Reusability ≡ 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**

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

```
Door
  canPass(Person p)
```

```
Locked
  canOpen(Person p)
```

```
Short
  canPass(Person p)
```

```
ShortLockedDoor
  canOpen(Person p)
  canPass(Person p)
```

**Aggregation & S.-Inheritance**

- Door must explicitly provide chaining
- Doorlike must anticipate wrappers

**Multiple Inheritance ✓**

**The Wrapper**

```
FileStream
  read()
  write()
```

```
SocketStream
  read()
  write()
```

```
SynchRW
  acquireLock()
  releaseLock()
```

**Inappropriate Hierarchies**

- Cannot inherit from both in turn with Multiple Inheritance (Many-to-One instead of One-to-Many Relation)
- Creating individual special wrapping Classes duplicates acquire/release code
The Wrapper – Aggregation Solution

Stream
  - read()
  - write()

FileStream
  - read()
  - write()

SocketStream
  - read()
  - write()

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

⚠️ Aggregation
- Undoes specialization
- Needs common ancestor

The Wrapper – Multiple Inheritance Solution

FileStream
  - read()
  - write()

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

SocketStream
  - read()
  - write()

SynchedFileStream
  - read()
  - write()

SynchedSocketStream
  - read()
  - write()

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

(De-)Composition Problems

All the problems of
- Duplication
- Fragility
- Lack of fine-grained control
are centered around the question

“How do I distribute functionality over a hierarchy”

~~ functional (de-)composition

⚠️ Inappropriate Hierarchies
Implemented methods (acquireLock/releaseLock) too high
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:
- attribute offsets \( \in \mathbb{N}^+ \)
- method bodies \( \in \mathbb{B} \) or classes \( \in \mathcal{C} \)
- \( \perp \) abstract
- \( \top \) in conflict

and the partial order \( \perp \subseteq b \subseteq \top \) for each \( b \in \mathcal{B} \)

Definition (Abstract Class \( \in \mathcal{C} \))
A function \( c : \mathcal{N} \mapsto \mathcal{B} \) with at least one abstract image 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 \( \mathcal{C} := \mathcal{N} \mapsto \mathcal{B} \).

Several possibilities for composing maps \( \mathcal{C} \sqcap \mathcal{C} \):
- the symmetric join \( \sqcup \), defined componentwise:

\[
(c_1 \sqcup c_2)(n) = \begin{cases} 
  b_1 & \text{if } b_1 = \perp \text{ or } n \notin \text{pre}(c_1) \\
  b_2 & \text{if } b_2 = \perp \text{ or } n \notin \text{pre}(c_2) \\
  \top & \text{otherwise}
\end{cases}
\]

where \( b_i = c_i(n) \)

- in contrast, the asymmetric join \( \sqcup \), defined componentwise:

\[
(c_1 \sqcup c_2)(n) = \begin{cases} 
  c_1(n) & \text{if } n \in \text{pre}(c_1) \\
  c_2(n) & \text{otherwise}
\end{cases}
\]
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 \(\triangleright\) : \(\mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}\), defined by
\[
c_1 \triangleright c_2 = \{ \text{super} \mapsto c_2 \} \cup (c_1 \setminus c_2)
\]

Example: Doors

\[
\begin{align*}
\text{Door} &= \{ \text{canPass} \mapsto 1 \}, \text{canOpen} \mapsto 1 \} \\
\text{LockedDoor} &= \{ \text{canOpen} \mapsto 0.x4204711 \} \triangleright \text{Door} \\
&= \{ \text{super} \mapsto \text{Door} \} \cup (\{ \text{canOpen} \mapsto 0.x4204711 \} \cup \text{Door}) \\
&= \{ \text{super} \mapsto \text{Door} \}, \text{canOpen} \mapsto 0.x4204711, \text{canPass} \mapsto 1 \}
\end{align*}
\]

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 \(\text{inner}\) explicitly delegates control to the subclass

Definition (Beta inheritance \(\langle\cdot\rangle\))
Beta inheritance is the binary operator \(\langle\cdot\rangle : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}\), defined by
\[
c_1 \langle c_2 = \{ \text{inner} \mapsto c_1 \} \cup (c_2 \setminus c_1)
\]

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."; }
}
```

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 \(\triangleright^\prime\))
The extended mainstream inheritance \(\triangleright^\prime : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}\) binds attribute \(n\) statically and without overwriting:
\[
(c_1 \triangleright^\prime c_2) (n) = \begin{cases} 
    c_2 & \text{if } n = \text{super} \\
    T & \text{if } n \in \text{pre}(c_1) \land c_2 (n) \in (N^+ \cup T) \\
    c_1 (n) & \text{if } n \in \text{pre}(c_1) \\
    c_2 (n) & \text{otherwise}
\end{cases}
\]

Adventure Game with Mixins

```
<mixin>Locked
canOpen(Person p)
canPass(Person p)
</mixin>
```

```
Door
canOpen(Person p)
canPass(Person p)
```

```
<mixin>Short
canPass(Person p)
</mixin>
```

```
ShortLockedDoor
```

mixin
compose
**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 constructor, defined by:

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

**Example: Doors**

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

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

$\text{Composed} = \text{mixin}(\text{Short}) \circ (\text{mixin}(\text{Locked}))$

$$= \lambda x \cdot \text{Short} \circ (\text{Locked} \circ x)$$

$$= \lambda x \cdot \{\text{super} \mapsto \text{Locked} \circ ((\text{canOpen} \mapsto 0x1234, \text{canPass} \mapsto 0x4711) \circ x)\}$$
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();

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>> {};}
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

---

Ruby

class Door
  def canOpen(p)
    true
  end
  def canPass(person)
    person.size < 210
  end
end

module ShortLocked
  include Short
  include Locked
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

p = Person.new
# d = Door.new  # This line is commented out
# d.extend ShortLocked
puts d.canPass(p)

---

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

Is Inheritance the Ultimate Principle in Reusability?
Lack of Control

⚠️ Control
- Common base classes are shared or duplicated at class level

Inappropriate Hierarchies

⚠️ Inappropriate Hierarchies
- High up specified methods are obsolescent, 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..."

⚠ Misuse of Implementation Inheritance

Implementation Inheritance itself as a pattern for code reuse is often
misused!

~ All that is not explicitly 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 ∈ T)

An abstract class t is called trait iff \( \forall n \in \text{pre}(t) \cdot t(n) \notin \mathbb{N}^+ \) (i.e. without attributes)

The trait sum + : T × T → T is the componentwise least upper bound:

\[
(c_1 + c_2)(n) = b_1 \cup b_2 = \begin{cases} 
    b_1 & \text{if } b_1 = \bot \vee n \notin \text{pre}(c_1) \\
    c_1(n) & \text{otherwise}
\end{cases} 
\]

\[
(b_2 & \text{if } b_2 = \bot \vee n \notin \text{pre}(c_2) \\
    b_2 & \text{if } b_1 = b_2 \\
    b_2 & \text{otherwise}
\]

Trait Expressions also comprise:

- exclusion \(-\) : T × N → T:

  \[
  (t - a)(n) = \begin{cases} 
    \text{undef} & \text{if } a = n \\
    t(n) & \text{otherwise}
  \end{cases}
  \]

- aliasing \((-\rightarrow\) : T × N × N → T:

  \[
  t[a → b](n) = \begin{cases} 
    t(n) & \text{if } n \neq a \\
    t(b) & \text{if } n = a
  \end{cases}
  \]

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

\[
c \sqcup t(n) = \begin{cases} 
    c(n) & \text{if } n \in \text{pre}(c) \\
    t(n) & \text{otherwise}
  \end{cases}
\]

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

Traits – Concepts

Trait composition principles

Flat ordering: All traits have the same precedence under +

~ explicit disambiguation with aliasing and exclusion

Precedence: Under asymmetric join \(\sqcup\), class methods take precedence
over trait methods

Flattening: After asymmetric join \(\sqcup\): 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 \(\sqcup\)
Can we augment classical languages by traits?

public class Person
{
    public int size = 160;
    public bool hasKey() { return true; }
}

class Short
{
    public static bool canOpen(Person p)
    { return p.hasKey(); }
    public static bool canPass(Person p)
    { return p.size < 160; }
}

class ShortLockedDoor : Locked, Short
{
    public static void Main()
    {
        ShortLockedDoor d = new ShortLockedDoor();
        Console.WriteLine(d.canOpen(new Person()));
    }
}

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:**
- Declare a static class with definitions of static methods
- Explicitly declare first parameter as receiver with modifier this
- Import the carrier class into scope (if needed)
- Call extension method in infix form with emphasis on the receiver

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:

```csharp
public interface Locked
{
    public bool canOpen(Person p);
}

public static class DoorExtensions
{
    public static bool canOpen(Person p)
    { return p.hasKey(); }
}
```
Extension Methods as Traits

- 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:

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

⚠️ Extension methods cannot overwrite abstract signatures

Traits as General Composition Mechanism

⚠️ Central Idea

Separate class generation from hierarchy specification and functional modelling

- model hierarchical relations with interfaces
- compose functionality with traits
- adapt functionality to interfaces and add state via glue code in classes

Simplified multiple Inheritance without adverse effects

Virtual Extension Methods (Java 8)

Java 8 advances one step further:

```java
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 {
}
```

⚠️ Precedence

Still, default methods do not overwrite methods from abstract classes when composed

So let’s do the language with real traits?!
Traits in Squeak

Trait named: #TStream
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: #TSynchRStream
uses: TSync
next
  | read |
  self acquireLock.
  read := self readNext.
  self releaseLock.
  ~ read.

Disambiguation

Traits vs. Mixins vs. Class-Inheritance

- 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

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...

- Gilead Bracha and William Cook. Mixin-based inheritance. European conference on object-oriented programming on Object oriented programming systems, languages, and applications (OOPSLA/ECOOP), 1999
- Stéphane Ducasse, Oscar Nierstrasz, Nathanaël Schöll, Rod Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. ACM Transactions on Programming Languages and Systems (TOPLAS), 2006.
- Nathanaël Schöll, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behaviour. European Conference on Object-Oriented Programming (ECOOP), 2003.