Is Multiple Inheritance the Ultimate Principle in Reusability?

**Learning outcomes**
- Identify problems of composition and decomposition
- Understand semantics of traits
- Separate function provision, object generation and class relations
- Traits and existing program languages

---

**Reusability ≡ Inheritance?**

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

**Duplication**

- Convenient implementation needs **second order types**, only available with
  - Mixins or
  - Templates

---

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

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

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

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

**SynchedSocketStream**
- `read()`
- `write()`
**Duplication**

- Convenient implementation needs *second order types*, only available with `~~ Mixins` or `~~ Templates`.
- With multiple inheritance, `read/write` Code is essentially *identical but duplicated*.

**Lack of Control**

- Common base classes are shared or duplicated at class level.

---

**Control**

- Common base classes are shared or duplicated at class level.
- Linearization overrides all identically named ancestor methods in parallel.

→ No *fine-grained specification* of duplication or sharing.
Inappropriate Hierarchies

- Implemented methods \( \text{acquireLock}/\text{releaseLock} \) to high

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

Is Implementation Inheritance even an Anti-Pattern?

Inappropriate Hierarchies

- Implemented methods \( \text{acquireLock}/\text{releaseLock} \) to high

Liskov Substitution Principle!
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 inheritance
Implementation Inheritance itself as a pattern for code reusag is often misused!
- All that is not explicitly prohibited will eventually be done!

Fragility

(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) to high
- High up specified methods turn obsolete, but there is no statically safe way to remove them
  ⚠️ Liskov Substitution Principle!
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 assembling 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 \( i \) is called trait iff \( \forall n \in \text{pre}(i) : t(n) \not\in \mathbb{N}^+ \) (i.e. without attributes)

The trait sum \( + : T \times T \rightarrow T \) is the componentwise least upper bound:

\[(c_1 + c_2)(n) = b_1 \cup b_2 = \begin{cases} 
  b_2 & \text{if } b_1 = \bot \lor n \not\in \text{pre}(c_1) \\
  b_1 & \text{if } b_2 = \bot \lor n \not\in \text{pre}(c_2) \\
  b_2 & \text{if } b_1 = b_2 \\
  \top & \text{otherwise}
\end{cases} \quad \text{with } b_i = c_i(n)\]

Trait-Expressions also comprise:

- exclusion \( \setminus : T \times N \rightarrow T \): \((t - a)(n) = \begin{cases} 
  \text{undef} & \text{if } a = n \\
  t(n) & \text{otherwise}
\end{cases}\)

- aliasing \( [\to] : T \times N \times N \rightarrow T \): \(t[a \to b](n) = \begin{cases} 
  t(n) & \text{if } n \neq a \\
  t(b) & \text{if } n = a
\end{cases}\)

Traits \( i \) can be connected to classes \( c \) by the asymmetric join:

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

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

Classes and Methods – again

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 \( \mathbb{B} \) (called bindings), consisting of:
- attribute offsets \( \in \mathbb{N}^+ \)
- method bodies \( \in \mathbb{B} \) or classes \( \in \mathcal{C} \)
- \( \bot \) (yet) undefined
- \( \top \) in conflict

and the partial order \( \bot \subseteq m \subseteq \top \) for each \( m \in \mathbb{B} \)

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

Definition (Interface and Class)
An abstract class \( c \) is called (with pre being the preimage)

- \text{interface } if \( \forall n \in \text{pre}(c) : c(n) = \bot \).
- \text{concrete } class if \( \forall n \in \text{pre}(c) : \bot \subseteq c(n) \subseteq \top \).

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 (\( \to \))
- Methods can be excluded (\( \setminus \))
- Class methods override trait methods and sort out conflicts (\( \sqcup \))
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
Traits in the Context of Modularity Problems

**Decomposition Problems**
- **Duplicated Features** ... can easily be factored out into unique traits.
- **Inappropriate Hierarchies** – Trait composition for reusable code concentrates inheritance on shaping interface relations.

**Composition Problems**
- **Conflicting Features** – Traits have no state, other conflicts resolved via exclusion, aliasing or overriding.
- **Lack of Control** – During trait composition precedence is chosen separately for each feature.
- **Dispersal of Glue Code** ... deferred to and concentrated in the final class.
- **Fragile Hierarchies** – Trait details are hideable due to missing hierarchy.

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
- especially provide definitions of *interface methods*

**Syntax:**
1. Declare a static class with definitions of static methods
2. Explicitly 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

```csharp
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 locked, 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
- 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(this Locked leftHand, Person p) {
        return p.hasKey();
    }
}
```

**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 canOpen(Person p) {
        return p.size <= 160;
    }
}

public class ShortLockedDoor implements Short, Locked, Door {
}
```

**Implementation**

...consists in adding an interface phase to invoke virtual's name resolution

**Precedence**

Still, default methods do not overwrite 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?!
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.hasMore(); }
}
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 invoke virtual's name resolution

⚠️ Precedence

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

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: #TSynchRstream uses: TSync*(TRStream&[readNext -> #next])
next
    | read |
    self acquireLock.
    read := self readNext.
    self releaseLock.
    ~ read.
Traits: So far so...

- **good**
  - Principles fully implemented
  - Concept has encouraged mainstream languages to adopt ideas

- **bad**
  - One very unconventional graphical IDE for Squeak, atak
  - ... and there is no separate compiler with command line mode!

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.

- Brian Goetz.
  Interface evolution via virtual extension methods.
  *JSR 335: Lambda Expressions for the Java Programming Language*, 2011.

- Anders Heijlsberg, Scott Wiltamuth, and Peter Golde.
  *C# Language Specification.*

- Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black.
  Traits: Composable units of behaviour.

Lessons learned

- Single inheritance, multiple Inheritance and Mixins leave room for improvement for 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
- Practically no language offers full traits in a usable manner