“Is Multiple Inheritance the holy grail of reusability?“

Learning outcomes

1. Identify problems of composition and decomposition
2. Understand semantics of traits
3. Separate function provision, object generation and class relations
4. Traits and existing program languages

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

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

SynchedFileStream
- SynchedSocketStream

⚠️ Duplicated Wrappers
Multiple Inheritance is not applicable as super-References are statically bound
(→ Alternative: Mixins)

Streams modified

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

SynchedFileStream
- SynchedSocketStream

⚠️ Duplicated Features
read/write Code is essentially identical but duplicated

Oh my god, streams!

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

SynchedFileStream
- SynchedSocketStream

⚠️ Inappropriate Hierarchies
Implement methods (acquireLock/releaseLock) to high
Decomposition problems

All the problems of
- duplicated Wrappers
- duplicated Features
- inappropriate Hierarchies
are centered around the question

“How do I distribute functionality over a hierarchy?”

~ functional decomposition

Are Mixins the solution?

- Rectangular
  - ToString()
- Rectangular + Color
  - ToString()
- Rectangular + Border
  - ToString()
- MyRectangle
  - ToString()

Fragile Hierarchies
- Linearization overrides identically named methods earlier in the chain
- super is not enough to sufficiently qualify inherited features, while explicit qualification makes refactoring difficult

Are Mixins the solution?

Lack of Control and Dispersal of Glue Code
- Overriding methods always happens in parallel
- Glue code penetrates the whole hierarchy

And Multiple Inheritance?

SpyCamera
- shoot()
MountablePlane
- MountablePlane
- unload(Ammunition)
- Visible
- equipment
- CameraPlane
- download():pics
- PoliceDrone
- parking
- PrecisionGun
- shoot()

Conflicting Features
- Common base classes are shared or duplicated at class level
- No fine-grained specification of duplication or sharing
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

A trait $t \in T$

- is a function $t : \mathcal{N} \rightarrow \mathcal{B}^*$
- has conflicts : $T \rightarrow 2^\mathcal{N}$ with conflicts$(t) = \{ l \mid t(l) = \top \}$
- provides : $T \rightarrow 2^\mathcal{N}$ with provides$(t) = t^{-1}(\mathcal{B})$
- selfSends : $\mathcal{B} \rightarrow 2^\mathcal{N}$, the set of method names used in self-sends
- requires : $T \rightarrow 2^\mathcal{N}$ with requires$(t) = \bigcup_{b \in \text{Binds}((\mathcal{N}))} \text{selfSends}(b) \setminus \text{provides}(t)$

... and differs from Mixins

- Traits are applied to a class in parallel, Mixins incrementally
- Trait composition is unordered, avoiding linearization problems
- Traits do not contain attributes, avoiding state conflicts
- With traits glue code is concentrated in particular classes

Trait composition principles

Flat ordering: All traits have the same precedence — explicit disambiguation

Precedence: Class methods take precedence over trait methods

Flattening: Non-overridden trait methods have the same semantics as class methods

Classes and methods

We will construct our model from the primitive sets of

- a countable set of method names $\mathcal{N}$
- a countable set of method bodies $\mathcal{B}$
- a countable set of attribute names $\mathcal{A}$

Values of method bodies $\mathcal{B}$ are extended to a flat lattice $\mathcal{B}^*$, with elements

- concrete implementations
- ⊥ undefined
- ⊤ in conflict

and the partial order $\bot \sqsubseteq m \sqsubseteq \top$ for each $m \in \mathcal{B}$

Definition (Method)
Partial function, mapping a name to a body

Definition (Method Dictionary $d \in \mathcal{D}$)
Total function $d : \mathcal{N} \rightarrow \mathcal{B}^*$, and $d^{-1}(\top) = \emptyset$

Definition (Class $c \in \mathcal{C}$)
Either $\text{nil}$ or $(\alpha, d) \cdot c'$ with $\alpha \in \mathcal{A}, d \in \mathcal{D}, c' \in \mathcal{C}$

Trait composition

Composing Classes from Traits:

$\langle \alpha, d \triangleright t \rangle \cdot c'$ with $\langle \alpha, d \rangle \cdot c'$ a class, $t$ a composition clause

with the overriding operator $\triangleright$:

$$(d \triangleright t)(l) = \begin{cases} t(l) & d(l) = \bot \\ d(l) & \text{otherwise} \end{cases}$$

Composition clauses are based on

- trait sum: $$(t_1 + t_2)(l) = t_1(l) \cup t_2(l)$$
- exclusion: $$(t - a)(l) = \begin{cases} \bot & \text{if } a = l \\ t(l) & \text{otherwise} \end{cases}$$
- aliasing: $$t[a \rightarrow b](l) = \begin{cases} t(l) & \text{if } l \neq a \\ t(b) & \text{if } l = a \land t(a) = \bot \\ \top & \text{otherwise} \end{cases}$$
### Trait handling

⚠️ **Conflicts**
Conflicts arise if composed traits possess methods with identical signatures.

**Conflict treatment**
- Methods can be aliased (↦)
- Methods can be excluded
- Class methods override trait methods and sort out conflicts (↦)

### Composition

✔️ **Conflicting Features**
Traits cannot have conflicting states, and offer conflict resolving measures like exclusion, aliasing or overriding.

✔️ **Lack of Control and Dispersal of Glue Code**
The composition entity can individually choose for each feature, which trait has precedence or how composition is done. Glue code can be kept completely within the composed entity.

✔️ **Fragile Hierarchies**
Conflicts can be resolved in the glue code. Navigational glue code is avoided.

### Decomposition

✔️ **Duplicated Features**
... can easily be factored out into unique traits.

✔️ **Inappropriate Hierarchies**
Trait composition as means for reusable code frees inheritance to model hierarchical relations.

✔️ **Duplicated Wrappers**
Generic Wrappers can be directly modeled as traits.

### Simulating Traits in C++

```cpp
template <class Super>
class SyncRW : virtual 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()
};```

---

Traits | Traits against the identified problems | Decomposition | 16 / 30
---|---|---|---
Traits | Traits against the identified problems | Decomposition | 16 / 30
Traits | Traits against the identified problems | Decomposition | 16 / 30
Traits | Traits against the identified problems | Decomposition | 16 / 30
Traits | Traits against the identified problems | Decomposition | 16 / 30
Simulating Traits in C++

```cpp
template <class Super>
class LogOpenClose : virtual public Super {
    public: virtual void open(){
        Super::open();
        log("opened");
    }
    virtual void close(){
        Super::close();
        log("closed");
    }
    protected: virtual void log(char* s) { ... };
};

template <class Super>
class LogAndSync :
    virtual public LogOpenClose<Super>,
    virtual public SyncRW<Super> {
};
```

Simulating Traits in C++

What misses for full traits?

- Aliasing
- Exclusion
- Precedence of class methods
- Specifying required methods
- Fine-grained control over duplication
- ~ Type system not flexible enough

But does that matter?

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”
Traits in PHP

trait Rectangular {
    private $l=3, $w=4;
    public function printInfo() { echo 'rectangular $l x $w'; } }

trait Colored {
    public $color = "red";
    public function printInfo() { echo 'color $this->color'; } }

class ColoredRect {
    use Colored, Rectangular;
    public function printInfo(){
        Rectangular::printInfo();
        echo ' with ';
        Colored::printInfo();
    }
}$c = new ColoredRect();$c->printInfo();

Aliasing Traits in PHP

trait Rectangular {
    private $l=3, $w=4;
    public function printInfo() { echo 'rectangular $l x $w'; } }

trait Colored {
    public $color = "red";
    public function printInfo() { echo 'color $this->color'; } }

class ColoredRect {
    use Colored, Rectangular {
        Rectangular::printInfo as printShapeInfo; 
        Colored::printInfo as printColorInfo; 
    }
    public function printInfo(){ ...
    }
}$c = new ColoredRect();$c->printColorInfo();

Alasing Drones as Traits in PHP

trait MountablePlane {
    abstract function store($equip);
    abstract function retrieve();
    public function mount($equip){ $this->store($equip); } 
    public function shoot() { $this->retrieve()->fire(); } }

trait CameraPlane { use MountablePlane; }
trait CombatPlane { use MountablePlane; }

class PoliceDrone { use CameraPlane, CombatPlane {
    CameraPlane::mount as mountCam;
    CombatPlane::mount as mountGun;
    CameraPlane::store as storeCamera;
    CombatPlane::store as storeGun;
    CameraPlane::shoot as shootPhoto;
    CombatPlane::shoot as shootTerrorist;
    CameraPlane::retrieve as retrieveCamera;
    CombatPlane::retrieve as retrieveGun;
}
private $cam, $gun;
...
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: #TSyncRStream uses: TSync+(TRStream(#readNext -> #next))
  next
    | read |
    self acquireLock.
    read := self readNext.
    self releaseLock.
    "read.

Lessons learned

Lessons Learned

- Single inheritance, multiple Inheritance and Mixins reveal shortcomings in real world problems
- 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

so far so...

- Syntax looks really promising
- Aliasing and Exclusion is implemented

- Especially Squeak features one of the most unconventional IDEs
- ...and there is no 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.

- Martin Odersky, Lex Spoon, and Bill Venner.
  ISBN 0981531601, 9780981531601.

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