**Dynamic vs. Static Casting**

```cpp
class D : public C,
  public B {
  ...
  }
  ...
  ...
  ...
  C c;
  W* pw = &c;
  C* pc = (C*) pw; // Compile error
  C* pc = dynamic_cast<C*>(pw);
}
```

**Virtual thunks**

```cpp
class W { ...
  virtual void g(int);
  ...
  ...
  ...
  C c;
  W* pw = &c;
  pw->g(42);
}
```

**Common base classes**

```cpp
class W {
  int w; virtual void f(int);
  virtual void g(int);
  virtual void h(int);
};

class A : public virtual W {
  int a; void f(int);
};

class B : public virtual W {
  int b; void g(int);
};

class C : public A, public B {
  int c; void h(int);
};
```

### Notes
- **Dynamic casting** makes use of `offset-to-top`
- No guaranteed `constant` offsets between virtual bases and subclasses → No static casting!
- Offsets to virtual base
- Ambiguities → e.g. overwriting f in A and B
  
  ```cpp
  C c;
  W* pc = (A*) pc;
  ((W*)pc)->h(42);
  ((A*)pc)->f(42);
  ```

---

**Script generated by TTT**

Title: Simon: Programmiersprachen (17.01.2014)
Date: Fri Jan 17 14:15:31 CET 2014
Duration: 93:37 min
Pages: 43
Virtual Tables for Virtual Bases (≈ C++-ABI)

A Virtual Table for a Virtual Subclass gets a **virtual base pointer**

A Virtual Table for a Virtual Base consists of different parts:
1. **virtual call offsets** per virtual function for adjusting this dynamically
2. **offset to top** of an enclosing objects heap representation
3. **typeid pointer** to an RTTI object (not relevant for us)
4. **virtual function pointers** for resolving virtual methods

Virtual Base classes have **virtual thunks** which look up the offset to adjust the this pointer to the correct value in the virtual table!

---

Virtual thunks

class W {
    virtual void g(int);
};

class A : public virtual W {...};
class B : public virtual W {
    int b; void g(int i) {};
};
class C : public A, public B {...};
    C c;
    W* pW = &c;
pW->g(42);

define void @VirtualThunkTo_W@k3(\%class.H* \%this, i32 %i) {
    %1 = bitcast %class.H* \%this to i32* %2
    %2 = bitcast i32* %1 to i8* %3
    %3 = load i8* %2
    %4 = getelementptr i8* %3, i64 *-32 ; -32 bytes is g-entry in vcalls
    %5 = bitcast i8* %4 to i16* %6
    %6 = load i16* %5
    ; load g's vcall offset
    %7 = getelementptr i8* %6, i64 *1, i64 %4 ; navigate to vcalloffset+ Wtop
    %8 = bitcast i8* %7 to %class.H* %9
    call void @VirtualThunkTo_W(%class.H* \%this, i32 %i)
} ret void

---

Virtual Tables for Virtual Bases (≈ C++-ABI)

A Virtual Table for a Virtual Subclass gets a **virtual base pointer**

A Virtual Table for a Virtual Base consists of different parts:
1. **virtual call offsets** per virtual function for adjusting this dynamically
2. **offset to top** of an enclosing objects heap representation
3. **typeid pointer** to an RTTI object (not relevant for us)
4. **virtual function pointers** for resolving virtual methods

Virtual Base classes have **virtual thunks** which look up the offset to adjust the this pointer to the correct value in the virtual table!
Reusability = Inheritance?

- 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

"Is Multiple Inheritance the holy grail of 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

Streams

- FileInputStream
  - `read()`
  - `write()`
- SocketStream
  - `read()`
  - `write()`
- SynchRW
  - `acquireLock()`
  - `releaseLock()`
  - `read()`
  - `write()`

⚠️ Duplicated Wrappers
Convenient implementation needs second order types, only available with ~~Mixins or ~~Templates
Streams

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

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

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

SynchedFileStream

SynchedSocketStream

**⚠️ Duplicated Wrappers**

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

**⚠️ Duplicated Features**

With multiple inheritance, `read/write` Code is essentially *identical but duplicated*

---

**Oh my god, streams!**

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

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

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

SynchedFileStream

SynchedSocketStream

**⚠️ Inappropriate Hierarchies**

Implemented 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 holy grail?

**Fragile Hierarchies**
- Linearization overrides all identically named methods earlier in the chain in parallel — Lack of control
- super isn't enough to sufficiently qualify inherited features, while explicit qualification makes refactoring difficult, and glue code necessary — Dispersal of glue code

---

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 modellling 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

---

And Multiple Inheritance?

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

---

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
- $\perp$ undefined
- $\top$ in conflict

and the partial order $\perp \sqsubseteq m \sqsubseteq \top$ 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 $d : \mathcal{N} \to \mathcal{B}^*$, and $d^{-1}(\top) = \emptyset$

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

A trait $t \in \mathcal{T}$

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

... 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 $\sqsubseteq$ explicit disambiguation
Precedence Class methods take precedence over trait methods
Flattening Non-overridden trait methods have the same semantics as class methods

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 overwriting operator $\triangleright$

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

Composition clauses are based on

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

Traits

A trait $t \in \mathcal{T}$

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

... 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 $\sqsubseteq$ explicit disambiguation
Precedence Class methods take precedence over trait methods
Flattening Non-overridden trait methods have the same semantics as class methods
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 (>)

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.

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.
Simulating Traits in C++

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()
};

Simulating Traits in C++

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) { ... };
};

Simulating Traits in C++

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

Simulating Traits in C++

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) { ... };
};

Simulating Traits in C++

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

⚠️ What misses for full traits?
Compositional expressions are not available:
- Aliasing
- Exclusion
- Precedence of class methods
- Specifying required methods
- Fine-grained control over duplication
- ~ Type system not flexible enough

But does that matter?
Simulating Traits in C++

⚠️ What misses for full traits?
Compositional expressions are not available:
- 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
- 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”

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: TSynch+(TRStream+ (#readNext -> #next))
  next
    | read |
    self acquireLock.
    read := self readNext.
    self releaseLock.
    read.
```

“So let’s do a language with real traits!”
“So how about extension methods?”

```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 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
- Flattening not implemented
- Class-code is distributed over several class bodies

⚠️ Limited scope of extension methods causes awkward errors:

```csharp
public interface Locked {
    public bool canOpen(Person p){
    }
}
public static class DoorExtensions {
    public static bool canOpen(this Locked leftHand, Person p){
        return p.hasKey();
    }
}
```
Virtual Extension Methods (Java 8)

The upcoming Java 8 advances one pace further:

```java
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

⚠️ Flattening

Still, default methods can still not overwrite methods from abstract classes

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

Traits: So far so...

✔️ good
- Principle looks really promising
- Concept has encouraged mainstream languages to adopt ideas
- Squeak even has Aliasing and Exclusion implemented

⚠️ bad
- Especially Squeak features one of the most unconventional IDEs
- ... and there is no command line mode!

Further reading...

- Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärfi, 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 Venners.
  
  
  
  ISBN 0981531601, 9780981531601.

- Nathanael Schärfi, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black.

  "Traits: Composable units of behaviour."
  