Script generated by TTT

Title: Simon: Programmiersprachen (17.01.2014)

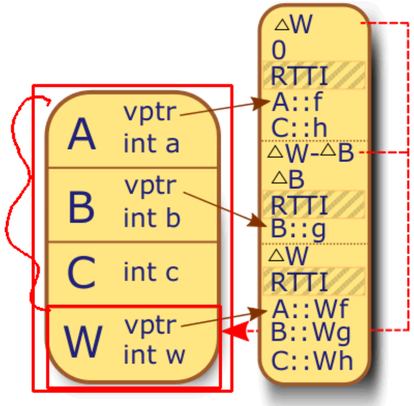
Date: Fri Jan 17 14:15:31 CET 2014

Duration: 93:37 min

Pages: 43

Common base classes

```
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);
};
...
C* pc;
pc->f(42);
((W*)pc)->h(42);
((A*)pc)->f(42);
```



- ⚠ Offsets to virtual base
- ⚠ Ambiguities
  ⇝ e.g. overwriting f in A *and* B

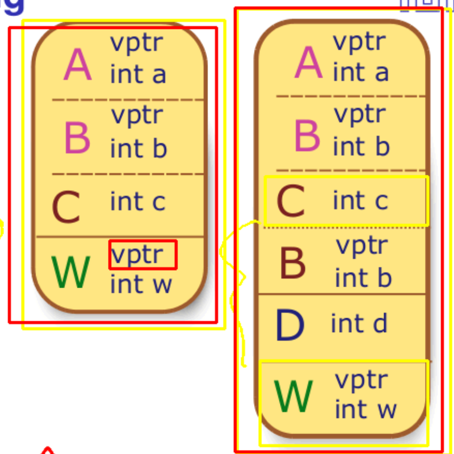
C c;
W* pw = (W*)&c;
C* pc = (C*)pw;

Dynamic vs. Static Casting

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

vs.

```
C* pc = dynamic_cast<C*>(pw);
```

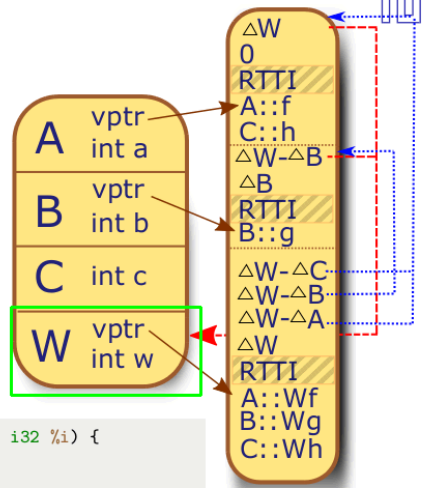


⚠ No guaranteed *constant* offsets between virtual bases and subclasses ⇝ No static casting!

⚠ *Dynamic casting* makes use of *offset-to-top*

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_B::g(%class.B* %this, i32 %i) {
  %1 = bitcast %class.B* %this to i8*
  %2 = bitcast i8* %1 to i8**
  %3 = load i8** %2                          ; load W-vtable ptr
  %4 = getelementptr i8* %3, i64 -32         ; -32 bytes is g-entry in vcalls
  %5 = bitcast i8* %4 to i64*
  %6 = load i64* %5                          ; load g's vcall offset
  %7 = getelementptr i8* %1, i64 %6          ; navigate to vcalloffset+ Wtop
  %8 = bitcast i8* %7 to %class.B*
  call void @B::g(%class.B* %8, i32 %i)
  ret void
}
```

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

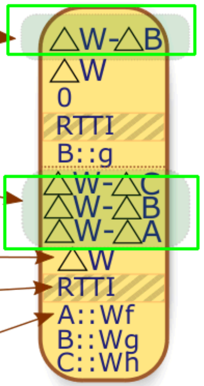
A Virtual Table for a Virtual Subclass

gets a *virtual base pointer*

A Virtual Table for a Virtual Base

consists of different parts:

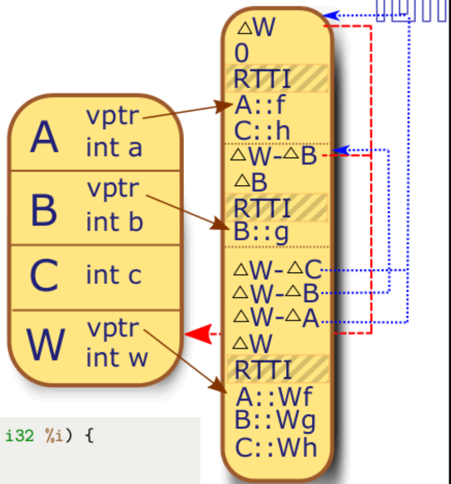
1. *virtual call offsets* per virtual function for adjusting `this` dynamically
2. *offset to top* of an enclosing objects heap representation
3. *typeinfo pointer* to an RTTI object (not relevant for us)
4. *virtual function pointers* for resolving virtual methods



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_B::g(%class.B* %this, i32 %i) {
  %1 = bitcast %class.B* %this to i8*
  %2 = bitcast i8* %1 to i8**
  %3 = load i8** %2                        ; load W-vtable ptr
  %4 = getelementptr i8* %3, i64 -32 ; -32 bytes is g-entry in vcalls
  %5 = bitcast i8* %4 to i64*
  %6 = load i64* %5                        ; load g's vcall offset
  %7 = getelementptr i8* %1, i64 %6  ; navigate to vcalloffset+ Wtop
  %8 = bitcast i8* %7 to %class.B*
  call void @B::g(%class.B* %8, i32 %i)
  ret void
}
```


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

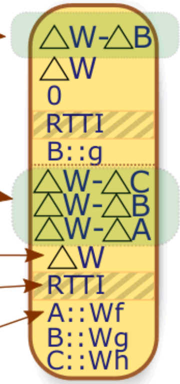
A Virtual Table for a Virtual Subclass

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. *typeinfo 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!

Multiple Inheritance | Implementation of Multiple inheritance | Common base classes 31 / 35

5inheritance.pdf — Programming Languages

File Edit View Go Bookmarks Help

Previous Next 31 (38 of 42) Fit Page Width

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

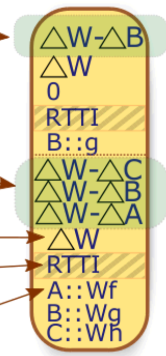
A Virtual Table for a Virtual Subclass

gets a *virtual base pointer*

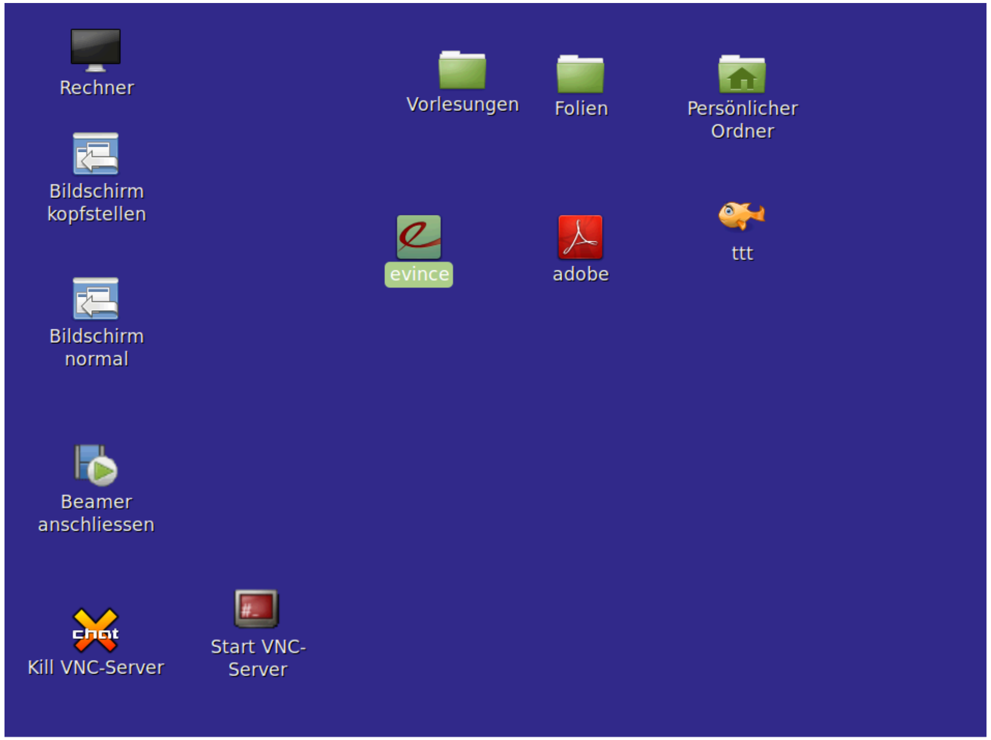
A Virtual Table for a Virtual Base

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. *typeinfo 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!



"Is Multiple Inheritance the holy grail of reusability?"

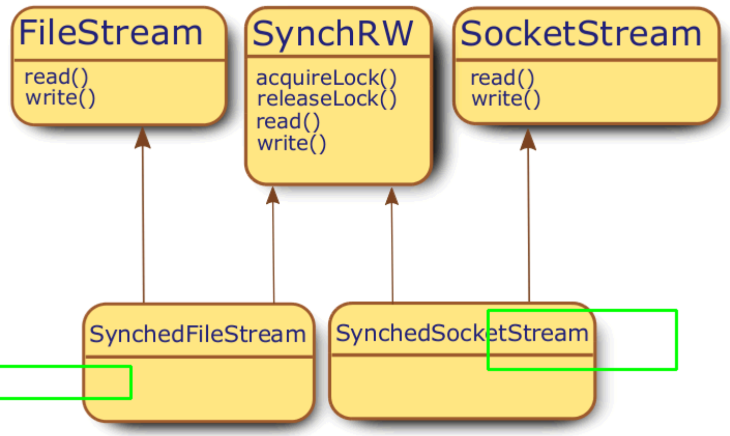
Learning outcomes

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

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*

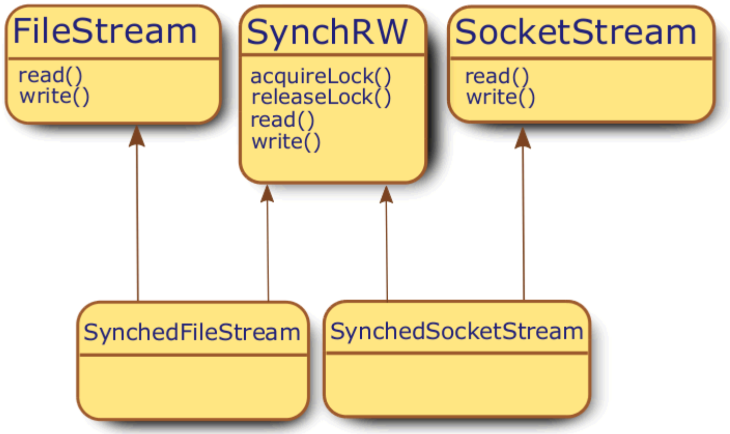
Streams



⚠ Duplicated Wrappers

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

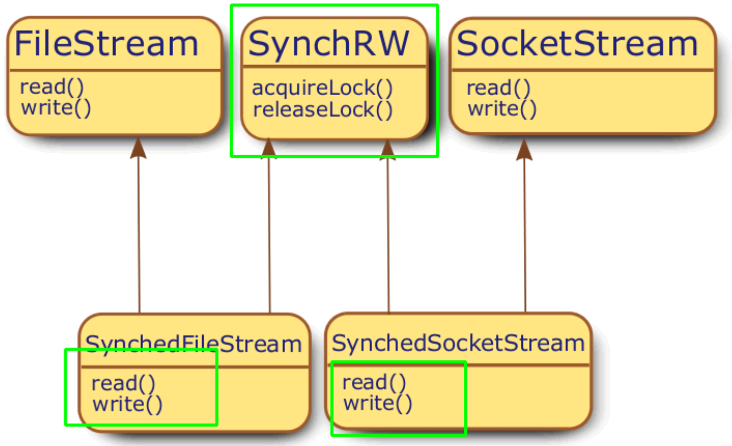
Streams



⚠ **Duplicated Wrappers**

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

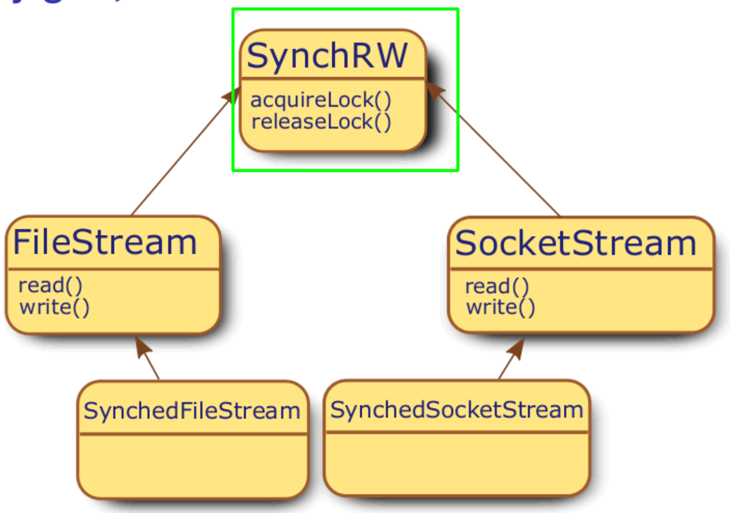
Streams modified



⚠ **Duplicated Features**

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

Oh my god, streams!



⚠ **Inappropriate Hierarchies**

Implemented methods (`acquireLock`/`releaseLock`) *to high*

Decomposition problems

All the problems of

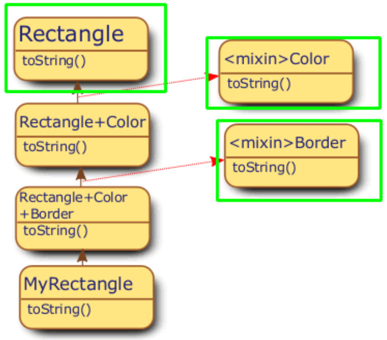
- duplicated Wrappers
- duplicated Features
- inappropriate Hierarchies

are centered around the question

"How do I distribute functionality over a hierarchy"

⇝ *functional decomposition*

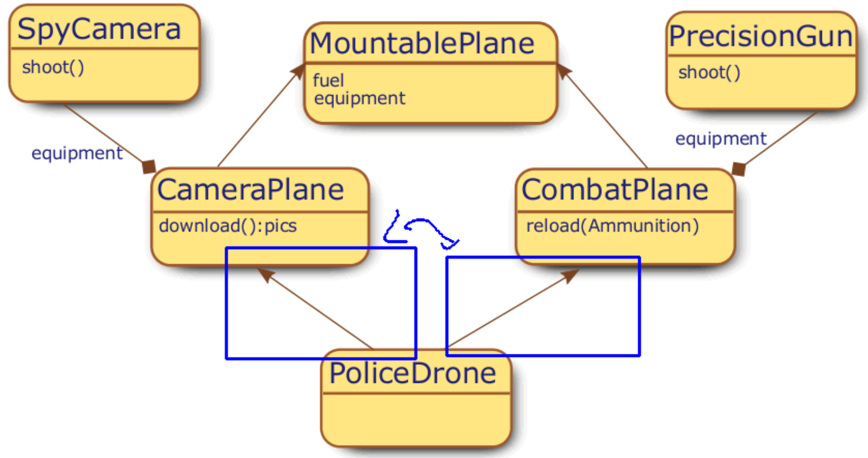
Are Mixins the holy grail?



⚠ Fragile Hierarchies

- Linearization overrides all identically named methods earlier in the chain in parallel ⇝ *Lack of control*
- `super` is not enough to sufficiently qualify inherited features, while explicit qualification makes refactoring difficult, and glue code necessary ⇝ *Dispersal of glue code*

And Multiple Inheritance?



⚠ 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

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* $\mathcal{B}^\star$, with elements

- concrete implementations
- $\bot$ undefined
- $\top$ in conflict

and the partial order $\bot \sqsubset m \sqsubset \top$ 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 $d : \mathcal{N} \mapsto \mathcal{B}^\star$, and $d^{-1}(\top) = \emptyset$

Definition (Class $c \in \mathcal{C}$)

Either nil or $\langle \alpha, d \rangle \cdot c'$ with $\alpha \in \mathcal{A}, d \in \mathcal{D}, c' \in \mathcal{C}$

Traits

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

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

... 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 $\rightsquigarrow$ 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:

$$\langle \alpha, d \rhd t \rangle \cdot c' \qquad \text{with } \langle \alpha, d \rangle \cdot c' \text{ a class, } t \text{ a composition clause}$$

with the overwriting operator $\rhd$:

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

Composition clauses are based on

- trait sum: $(t_1 + t_2)(l) = t_1(l) \sqcup t_2(l)$
- exclusion: $(t - a)(l) = \begin{cases} \bot & \text{if } a = l \\ t(l) & \text{otherwise} \end{cases}$
- aliasing: $t[a \to b](l) = \begin{cases} t(l) & \text{if } l \neq a \\ t(b) & \text{if } l = a \wedge t(a) = \bot \\ \top & \text{otherwise} \end{cases}$

Traits

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

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

... 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 $\rightsquigarrow$ explicit disambiguation

Precedence Class methods take precedence over trait methods

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

Traits

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

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

... 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 $\rightsquigarrow$ 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 posses methods with identical signatures

Conflict traitment

- ✓ Methods can be aliased ($\rightarrow$)
- ✓ Methods can be excluded
- ✓ Class Methods override trait methods and sort out conflicts ($\triangleright$)

Trait handling

⚠ **Conflicts**

Conflicts arise if composed traits posses methods with identical signatures

Conflict traitment

- ✓ Methods can be aliased ($\rightarrow$)
- ✓ Methods can be excluded
- ✓ Class Methods override trait methods and sort out conflicts ($\triangleright$)

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);
    relaseLock();
  };
  // ... 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) { ... };
};

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


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?

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) { ... };
};

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

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

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 a language with real traits!"

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

“So how about extension methods?”

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. Explicitely 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 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:

```
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:

```
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*

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!

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

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 Venners.
  Programming in Scala: A Comprehensive Step-by-step Guide.
  Artima Incorporation, USA, 1st edition, 2008.
  ISBN 0981531601, 9780981531601.
- Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black.
  Traits: Composable units of behaviour.
  European Conference on Object-Oriented Programming (ECOOP), 2003.