Typing Dynamic Layer Composition

Atsushi Igarashi (Kyoto U.)

joint work with

Robert Hirschfeld (HPI)

Hidehiko Masuhara (Tokyo Tech.)

Hiroaki Inoue (Kyoto U.)

Context-Oriented Programming (COP) Language [Costanza, Hirshfeld DLS05] [Hirschfeld, Costanza, Nierstrasz JOT08]

Goal: Support for modularization of *behavioral variations* depending on the *dynamic* context of execution

Example: Mobile email app

When network is fast
inline images are shown

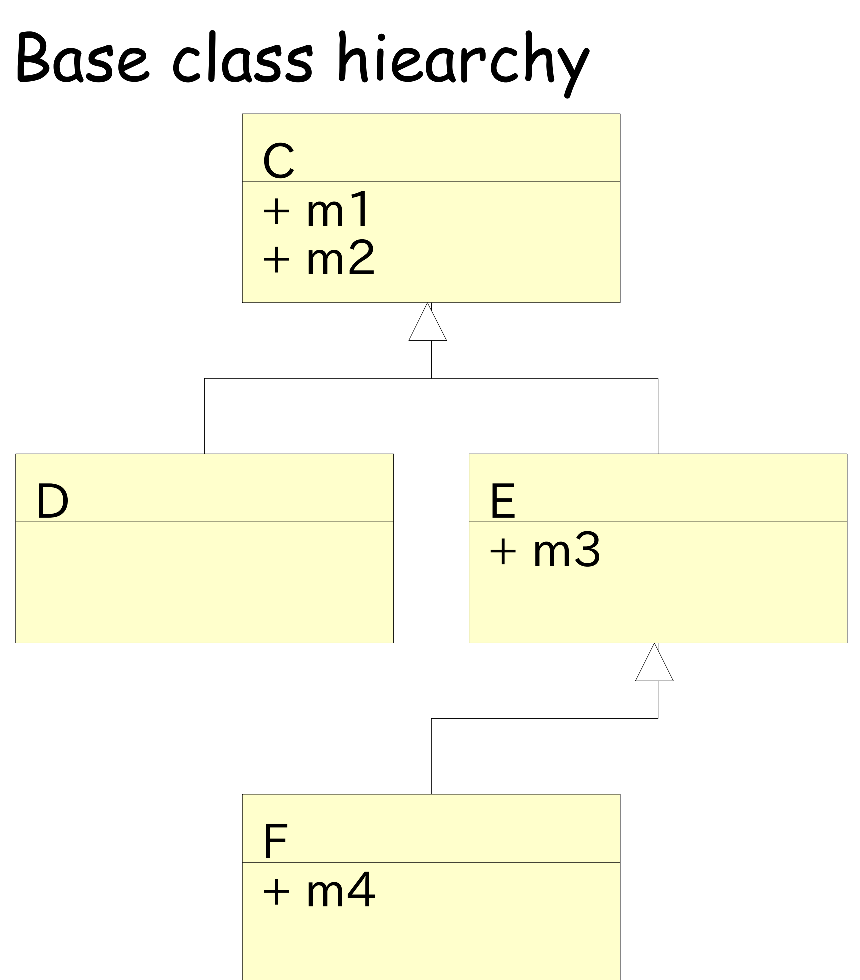
When network is slow
no images are shown

Common COP language features

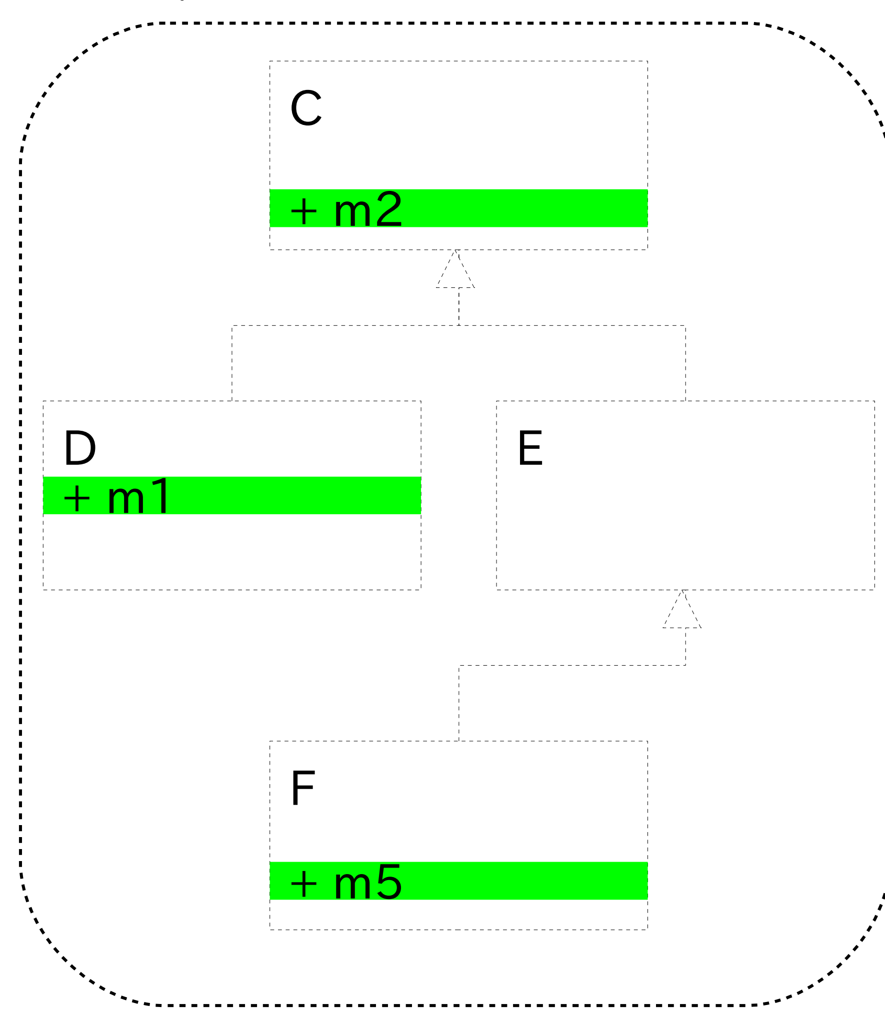
- Layer
  - A unit of behavioral variations, consisting of *partial* method definitions for multiple classes
  - (Loose) correspondence to contexts
  - A unit of cross-cutting modularity
- Dynamic layer activation
  - To change the behavior of a set of objects at the same time

Dynamic Layer Activation in COP

Base class hiearchy

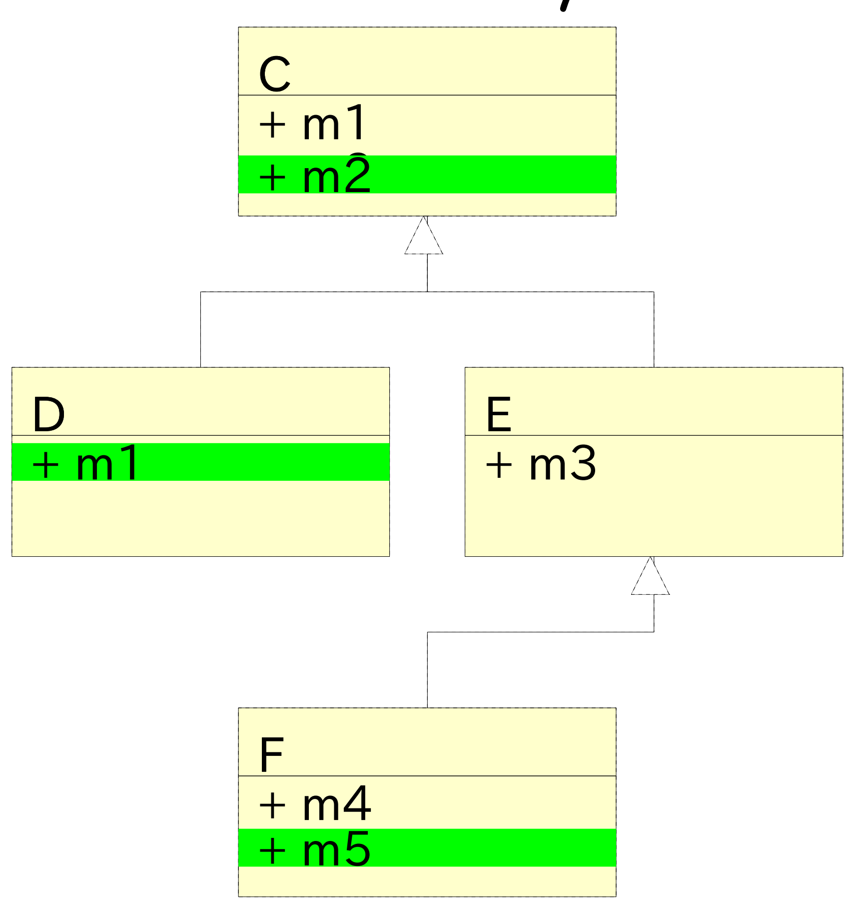


Layer of partial methods



Dynamic Layer Activation in COP

Base class hiearchy



- Layer activation changes behavior of objects *that have been already instantiated*
- Partial methods can call the original behavior by **`proceed()`**

This Talk

- Quick tour on JCop [Appeltauer+], a specific implementation of COP on top of Java
  - With a more concrete example
  - (Comparison with AOP using pointcut/advice)
- Foundations of COPL

This Talk

- Quick tour on JCop [Appeltauer+], a specific implementation of COP on top of Java
  - With a more concrete example
  - (Comparison with AOP using pointcut/advice)
- Foundations of COPL

Example: Telecom simulation
(adapted from AOP example)

- Class `Conn` to represent connection between two `Customers`
  - `complete()` when a connection has been established
  - `drop()` when the customers are disconnected
- Behavioral variations to consider
  - Recording the lengths of conversations
  - Billing

Base Program

```
class Conn { // Connection
  Conn(Customer a, Customer b) { … }
  void complete() { … }
  void drop() { … }
    // details are not important ...
}
```

```
static Conn simulate() {
  Customer robert = …, hidehiko = …;
  Conn c = new Conn(robert, hidehiko);
                  // Robert calls Hidehiko
  c.complete();   // Hidehiko accepts
  c.drop();       // and hangs up
  return c;
}
```

Layer for Measuring Time

```
layer Timing {
  Timer timer = …;
  void Conn.complete(){ proceed(); timer.start();}
  void Conn.drop(){ timer.stop(); proceed(); }
  int Conn.getTime() { return timer.getTime(); }
}
```

- The two methods in `Conn` are modified by *partial* method definitions to operate the timer
  - The original behavior is represented by **`proceed()`**
- **`getTime()`** is newly introduced
  - but also called "partial" method

Layer Activation with `with`

```
with (new Timing()) {  // layer activation!
   Conn c = simulate();
   System.out.println(c.getTime());
}
```

- `with` block to activate a layer
- Activation is effective even in methods invoked inside the block
- A layer *instance* has to be created
  - Layer instances are also first-class objects

Layer for Billing

```
layer Billing {
  void Conn.drop() { proceed(); charge(); }
  void Conn.charge() { … getTime(); … }
}
```

```
with (new Timing()) {
  with (new Billing()) {
    Connection c = simulate();
} }
```

- Recently activated layer has priority
  - `drop()` will stop the timer, hang the call, and charge

Not in this example, but...

- One layer can contain partial methods belonging to different classes
  - c.f. Mixin layers [Smaragdakis&Batory 98]
- `super()` is also supported
- Layer inheritance/subtyping

Layer Inheritance/Subtyping

- Implementation of different billing policies, switched by run-time conditions

```
abstract layer AbsBilling {
  void Conn.drop();
  void Conn.charge();
}
layer Billing1 extends AbsBilling { … }
layer Billing2 extends AbsBilling { … }
```

```
AbsBilling b =
  some_cond ? new Billing1():new Billing2();

with(b) { … }
```

Very rough Comparison with Aspect/J-style AOP

	COP	AOP
Unit of behavior	partial meth.	advice
Oblivious?	No	Yes
Join points	Meth. exec.	Many kinds
Pointcut	cflow + execution	Many kinds

Some Foundational Questions

- What is the semantics of method invocations?
  - What happens when the same layer is activated more than once?
  - How do `proceed`, `super`, and `with` interact with each other?
- How can types prevent `NoSuchMethodError`?
  - Object interface can change dynamically!
  - Only overriding partial methods can `proceed`

This Talk

- Quick tour on COP language features
  - With a more concrete example
- Foundations of COPL

A core calculus of COP: ContextFJ

[Hirschfeld, I., Masuhara FOAL'11]

ContextFJ = Featherweight Java [I.,Pierce,Wadler'99]

+ partial methods

+ proceed(), super()

+ with expressions

- layers are global and second-class

- no layer inheritance

ContextFJ<:

[Inoue&I. APLAS'15]

ContextFJ<: = Featherweight Java

+ partial methods
+ proceed(), super()
+ with expressions

+ first-class layers (w/o fields)

+ layer inheritance

+ layer subtyping

Semantics of Method Dispatch w/o Layer Inheritance

```
with (new L1()) {
  with (new L2()) {
    c3.m(…);
} }
```

L1

L2

C1

9

C2

6

C3

3

C1

8

C2

5

C3

2

C1

7

C2

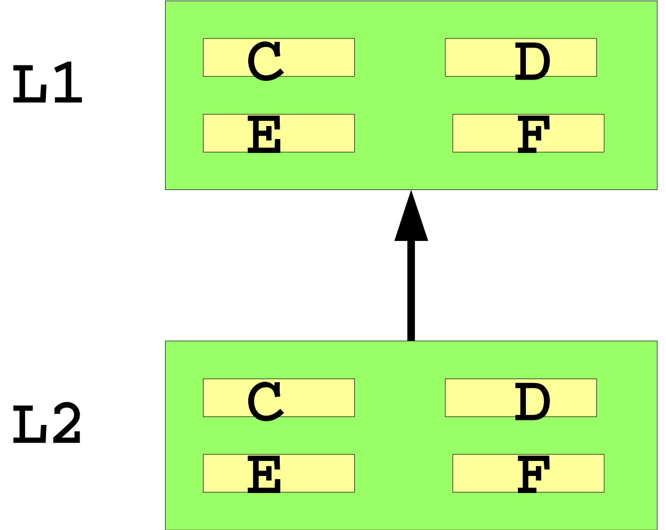
4

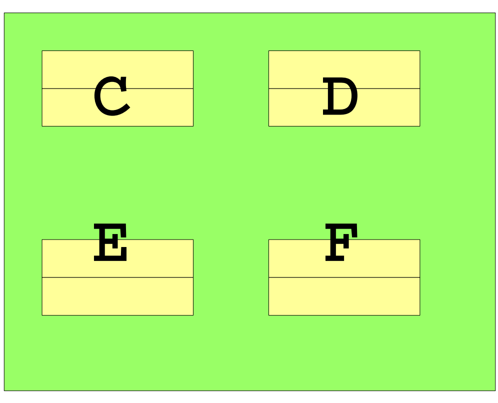
C3

1

Semantics with Layer Inheritance

- "3D" dispatching
- Each layer can be thought of as the result of (possibly overriding) composition of superlayers



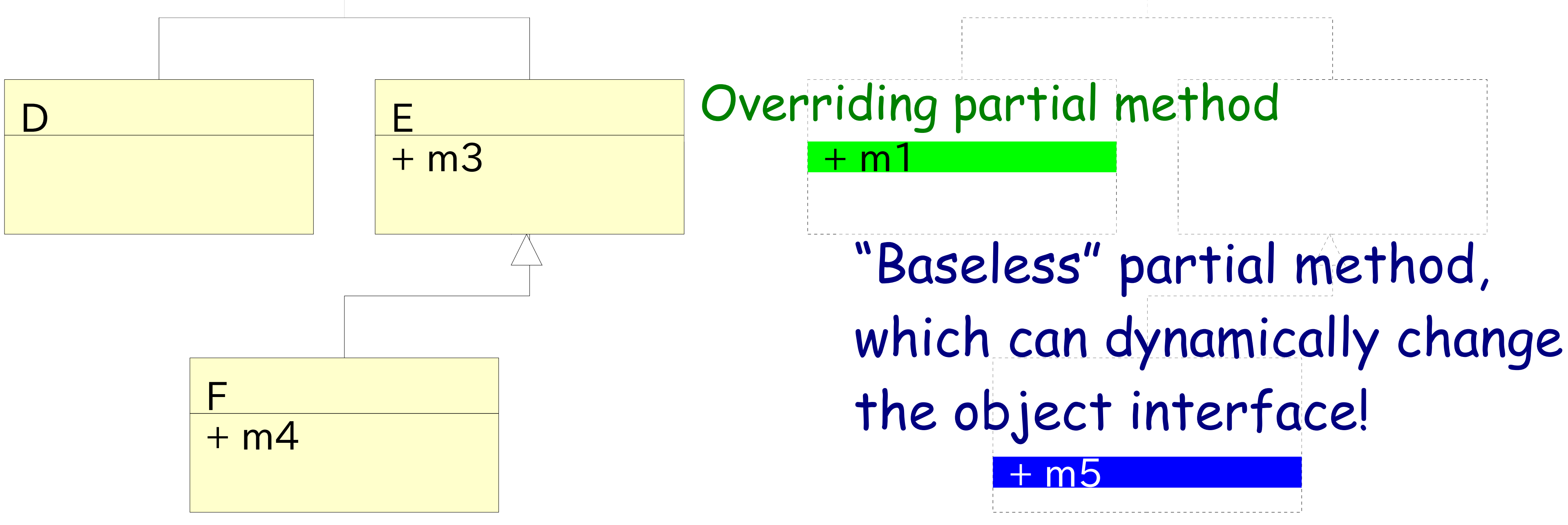


This Talk

- Quick tour on COP language features
  - With a more concrete example
- Foundations of COPL
  - (Operational) Semantics
  - Type System
    - To prevent "NoSuchMethodError" including dangling `proceed` calls

"Sounds like an old problem. What is a challenge?"

- Object interfaces can change as layers are (de)activated!



Key Ideas (1/2)

Approximating activated layers at each program point

- With the help of explicit "**`requires`**" declarations to specify inter-layer dependency
  - (Static analysis could dispense with such explicit declarations)

Key Ideas (2/2)

Two kinds of substitutability for layers

- When one layer L1 `requires` layer L2, does a sublayer of L2 can satisfy L1's requirement?
- When is it safe to pass an instance of a layer to where a supertype is expected?

should be distinguished

Telecom example, revisited

```
class Conn {
  Conn(Custome
  void complet
  void drop()
}
```

```
layer Timing {
  Timer Conn.tim
  void Conn.comp
  void Conn.drop
  int Conn.getTi
}
```

```
layer Billing                    {



  void Conn.drop(){ proceed(); cl

  void Conn.charge(){ … getTime()
}
```

- For **charge()** in **Billing** to work, baseless partial method **getTime()** defined in **Timing** should be active beforehand

Telecom example, revisited

```
class Conn {
  Conn(Custome
  void complet
  void drop()
}
```

```
layer Timing {
  Timer Conn.tim
  void Conn.comp
  void Conn.drop
  int Conn.getTi
}
```

```
layer Billing requires Timing  {


  void Conn.drop(){ proceed(); c

  void Conn.charge(){ … getTime(
}
```

- For **charge()** in **Billing** to work, baseless partial method **getTime()** defined in **Timing** should be active beforehand
- In other words, **Billing requires Timing**

Meaning of `requires`

When layer L **`requires`** $L_1, \ldots, L_n$

- All of $L_1, \ldots, L_n$ (or their sublayers) must have been already activated (in any order) before activating L
- Partial method in L can invoke methods defined in any of $L_1, \ldots, L_n$
- Partial method m in L can **`proceed`** when any of $L_1, \ldots, L_n$ (or base class) defines m

Type Judgment $\Lambda; \Gamma \vdash e : T$

Coeffect system?

"Under set Λ of activated layers and type env. Γ, exp e is given type C"

- {}; c: `Conn` ⊢ `c.getTime()` : `int`
- {`Timing`}; c: `Conn` ⊢ `c.getTime()` : `int`
- {}; c: `Conn` ⊢ `with (new Timing())c.getTime()` : `int`
- {}; c: `Conn` ⊢ `with (new Billing())c.drop()` : `void`
- {`Timing`}; c: `Conn`
  ⊢ `with (new Billing()) c.drop()` : `void`

Inheritance, subtyping and `requires`

- Sublayer can't require fewer layers than its parent
  - Otherwise, requirement by inherited partial methods may be invalidated
- It seems natural to allow a sublayer to require more layers ...

…Or, maybe not!

```
AbsBilling b =
  some_condition ? new Billing1():new Billing2();

with(b) { … }
```

- The type system seems to always allow `with(b)` (if `AbsBilling` requires no layer)
- But, what if `Billing2` requires more layers than `AbsBilling`?
  - At run time, dependecy is broken!!

Our Solution: Two subtyping rels for layer types

- Weak subtyping (reflexive transitive closure of `extends`) for checking `requires` at `with`

```
// L1 extends L2,  L3 requires L2
with(new L1())
  with(new L3()) { … }
```

- Normal subtyping (reflexive transitive closure of `extends` *with invariant* *`requires`*) for ordinary subsumption

For more details

- ContextFJ [Hirschfeld, I., Masuhara; FOAL'11]
    - Operational semantics
    - Simple type system disallowing baseless methods
- Type system for baseless methods [I., Hirschfeld, Masuhara; FOOL12]
    - (Slightly different activation semantics)
- Layer inheritance & first-class layers [Inoue&I.; APLAS'15]

Related Work

- Type System for COP [Clarke & Sergey; COP'09]
  - ContextFJ
    - proposed independently of us
    - no inheritance, subtly different semantics
  - Set of method signatures as method-wise dependency information
    - Finer-grained specification
  - No proof of soundness
    - In fact, the type system turns out to be flawed (personal communication), due to `without`

Related Work, contd.

- Type Systems for Mixins [Bono et al., Flatt et al., Kamina&Tamai, etc.]
  - Interfaces of classes to be composed
    - Structural type information
  - Composition is fixed once an object is instantiated
  - A similar idea works (to some extent ;-) also for more dynamic composition as in COP
- Types for FOP, DOP

Related Work, contd.^2

- Typestate checking [Strom&Yemini'86, etc.]
  - Checking state transition for computational resources (such as files and sockets)
  - Layer configuration can be considered a state

Conclusion

- Dynamic layer composition for describing context-dependent behavioral change concisely and modularly
- Inter-layer dependency (`requires`) works for dynamic composition (as well as static)
- Two kinds of subtyping relations

Future work:

- Type-sound *deactivation*