A Type System for Dynamic Layer Composition

Atsushi Igarashi
(Kyoto Univ.)

Joint work with
Robert Hirschfeld (HPI)
Hidehiko Masuhara (Univ. of Tokyo)

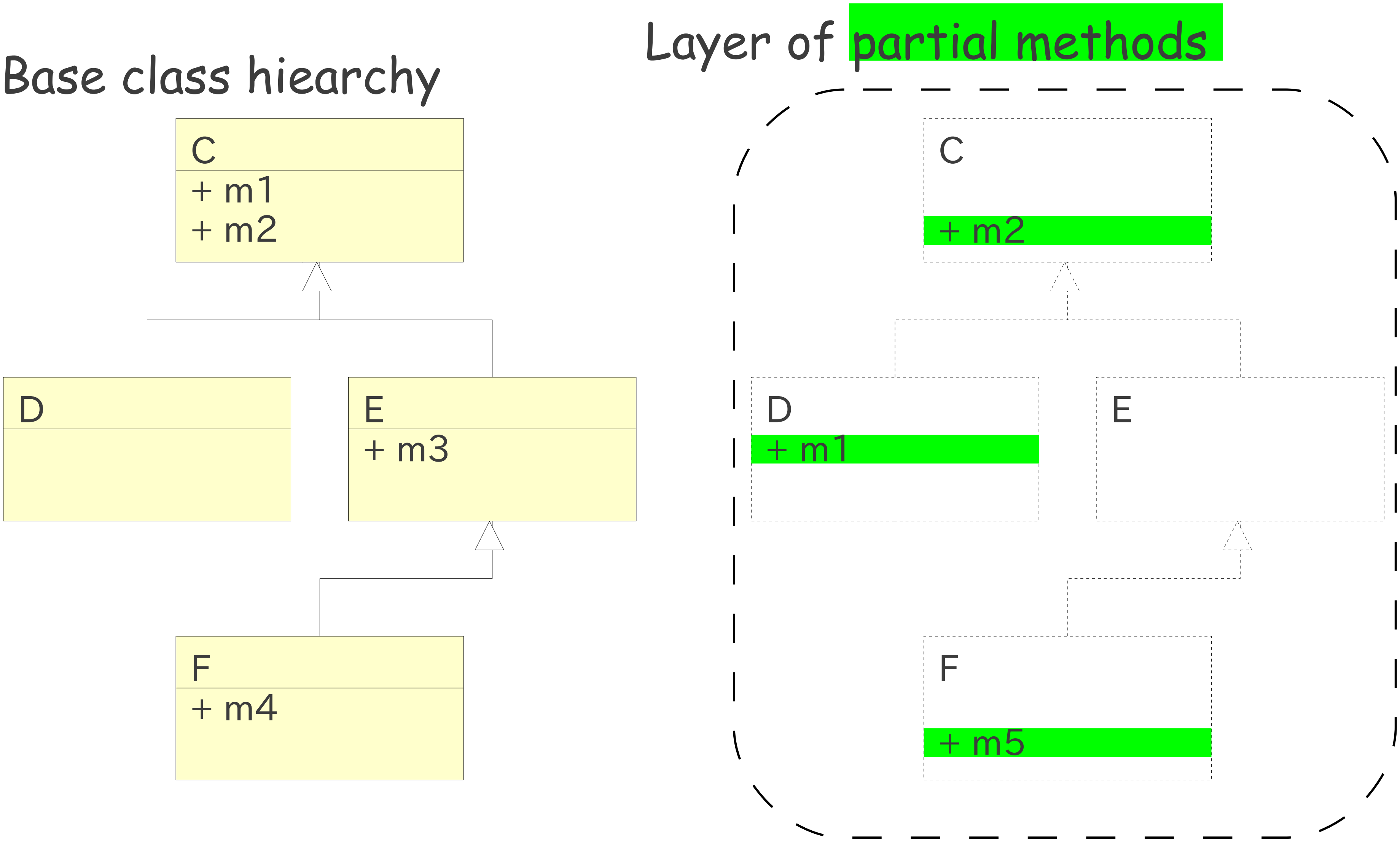
Background

Context-oriented Programming (COP)

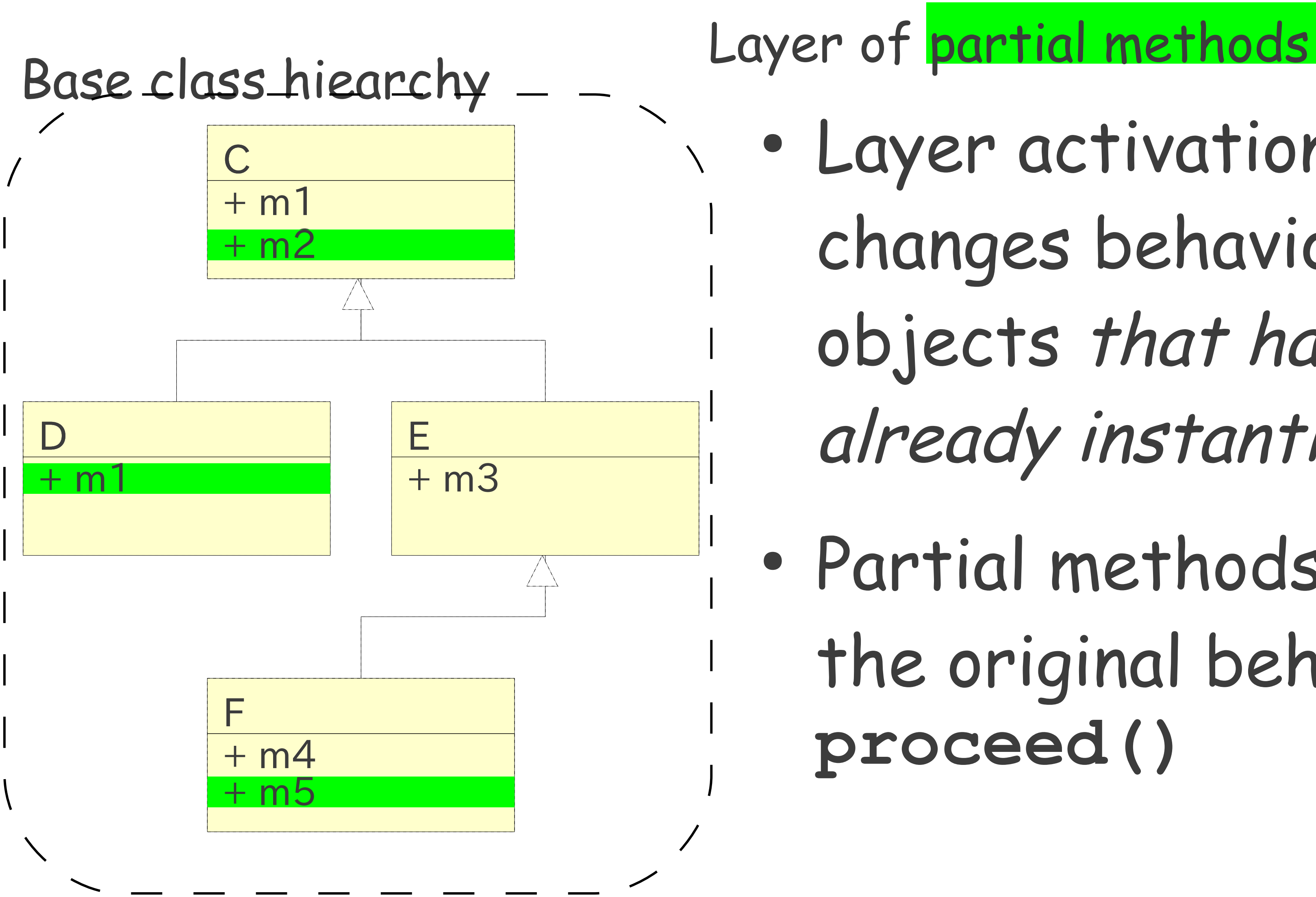
[Hirschfeld, Costanza, Nierstrasz JOT08]

- Goal: modularizing behavioral variations depending on the dynamic context of execution
  - e.g., editor key binding depending on buffer modes
- Several COP extensions of existing (OOP) langs
- Common language features
  - Layers of partial methods
  - Dynamic layer activation mechanism

Dynamic Layer Activation in COP



Dynamic Layer Activation in COP



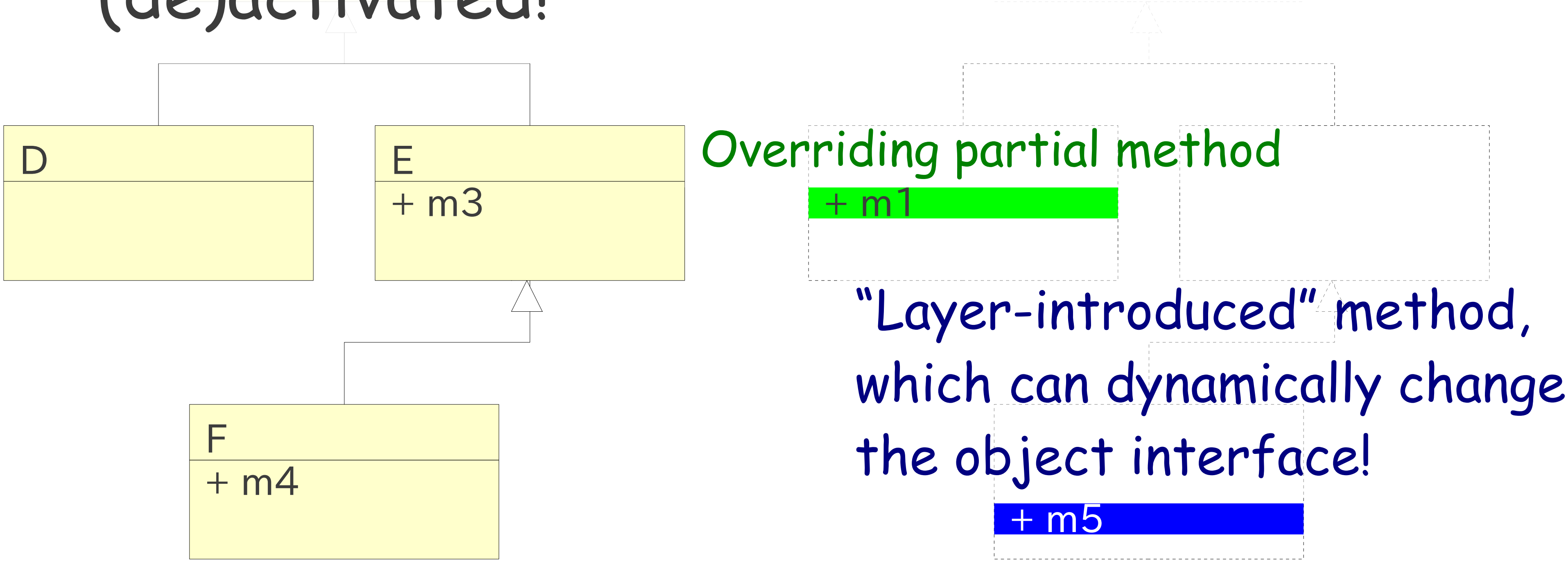
- Layer activation changes behavior of objects *that have been already instantiated*
- Partial methods can call the original behavior by **`proceed()`**

This Work

Type system to prevent “NoSuchMethod” incl. dangling `proceed` calls

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

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



This Work

Type system to prevent "NoSuchMethod" incl. dangling `proceed` calls

- By keeping track of which layers are activated at each program point
  - Using declaration of dependency between layers

Restriction:

- Activation/deactivation constructs are (slightly) different from the original

Technical Contributions

- Formal type system for (a variant of) ContextFJ [Hirschfeld, I., Masuhara @FOAL'11]
  - FJ-style calculus modeling COP features
- Proof of type soundness

Plan of the Talk

- COP Language Constructs
- Type System
- Discussion
- Conclusion and Future Work

COP Language Constructs Considered in This Work

- Partial methods
  - Smallest unit to describe behavioral variations
  - Comparable to advice in AOP
- Layers
  - A bunch of partial methods
  - Unit of modularity/cross-cutting concerns
- Block-structured *global* layer activation
  - **`ensure`** statements (a variant of **`with`**)

Example: Telecom simulation

```
class Connection {
  Connection(Customer a, Customer b) { … }
  void complete() { … }
  void drop() { … }
}
```

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

Example: Telecom simulation

```
class Connection {
  Connection(Customer a, Customer b) { ... }

}
```

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

```
ensure (Timing) {  // layer activation!
  Connection c = simulate();
  System.out.println(c.getTime());
}
```

- Layer activation is valid inside `simulate()`

Example: Telecom simulation

```
layer Billing {
  class Connection {
    void drop() { proceed(); charge(); }
    void charge() { … getTime(); … }
} }
```

```
class Co
  Connec
}
```

```
layer
  clas
    Ti
    vo                                              ; }
    void drop() { timer.stop(); proceed(); }
    int getTime() { return timer.getTime(); }
} }
```

```
ensure (Timing) {                      :ivation!
  ensure (Billing) {
    Connection c = simulate();()));
} }
```

- Layer activation is valid inside `simulate()`
- Recently activated layer has priority

Plan of the Talk

- COP Language Constructs
- Type System
- Discussion
- Conclusion and Future Work

Main Problem and Key Ideas

Main Problem:

- Layer activation can change object interface

Key Ideas:

- Approximation of activated layers at each program point
  - With the help of explicit "requires" declarations to specify inter-layer dependency

Telecom example, revisited

```
class Connecti
  Connection(C
  void complet
  void drop()
}
```

```
layer Timing {
  class Connecti
    Timer timer;
    void complet
    void drop()
    int getTime(
} }
```

```
layer Billing                    {
  class Connection {


    void drop() { proceed(); cha

    void charge() { … getTime();
} }
```

- For **`charge()`** in **`Billing`** to work, layer-introduced base method **`getTime()`** defined in **`Timing`** should be active beforehand

Telecom example, revisited

```
class Connecti
  Connection(C
  void complet
  void drop()
}
```

```
layer Timing {
  class Connecti
    Timer timer;
    void complet
    void drop()
    int getTime(
} }
```

```
layer Billing  requires Timing {
  class Connection {


    void drop() { proceed(); cha

    void charge() { … getTime();
} }
```

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

Meaning of `requires`

When layer L `requires` $L_1, ..., L_n$

- Partial method in L can invoke methods defined in any of $L_1, ..., L_n$ (or base)
- Partial method m in L can `proceed` when m is defined in any of $L_1, ..., L_n$ (or base)
- All of $L_1, ..., L_n$ must have been already activated before activating L

A Bit of Formalism

ContextFJ calculus [Hirschfeld, I., Masuhara @FOAL'11]

```
L ::= class C extends C { ~C ~f; ~M }
M ::= C m(~C ~x) { return e; }
e ::= x | e.f | e.m(~e) | new C(~e)
  | ensure L e | proceed(~e) | super.m(~e)
```

- A ContextFJ program is (*CT*, *PT*, e), where
  - Class table: *CT*(*C*) = *L*
  - Partial method table: *PT*(*m*,C,*L*) = *M*

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

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

- {}; c: `Conn.` $\vdash$ `c.getTime()` : `int`
- {`Timing`}; c: `Conn.` $\vdash$ `c.getTime()` : `int`
- {}; c: `Conn.` $\vdash$ `ensure Timing c.getTime()` : `int`
- {}; c: `Conn.` $\vdash$ `ensure Billing c.drop()` : `void`
- {`Timing`}; c: `Conn.`
  $\vdash$ `ensure Billing c.drop()` : `void`

Main Typing Rules

- Typing rule for method invocation

$$\frac{\Lambda; \Gamma \vdash e_0 : C_0 \qquad mtype(m, C_0, \Lambda) = \sim D \rightarrow C \qquad \Lambda; \Gamma \vdash \sim e : \sim C \qquad \sim C <: \sim D}{\Lambda; \Gamma \vdash e_0 . m (\sim e) : C}$$

- Typing rule for layer activation

$$\frac{L \ \mathbf{req} \ \Lambda' \qquad \Lambda' \subseteq \Lambda \qquad \Lambda \cup \{L\}; \Gamma \vdash e : C}{\Lambda; \Gamma \vdash \mathbf{ensure} \ L \ e : C}$$

Main Typing

Method lookup takes activated layers into account

- Typing rule for method invocation

$$\frac{\Lambda; \Gamma \vdash e_0 : C_0 \qquad mtype(m, C_0, \Lambda) = \ \sim D \rightarrow C \qquad \Lambda; \Gamma \vdash \sim e : \sim C \qquad \sim C <: \sim D}{\Lambda; \Gamma \vdash e_0 . m(\sim e) : C}$$

- Typing rule for layer activation

$$\frac{L \ \mathtt{req} \ \Lambda' \qquad \Lambda' \subseteq \Lambda \qquad \Lambda \cup \{L\}; \Gamma \vdash e : C}{\Lambda; \Gamma \vdash \mathtt{ensure} \ L \ e : C}$$

Main Typing

- Typing rule for method invoc

$$\frac{\Lambda; \Gamma \vdash e_0 : C_0 \qquad mtype(m, C_0, \Lambda) = \sim D \rightarrow C \qquad \vdash \sim e : \sim C}{\Gamma \vdash e_0 . m (}$$

Method lookup takes activated layers into account

Layers that L requires are already activated

Body of **ensure** is typed under additional assumption

- Typing rule for layer activation

$$\frac{L \ \mathbf{req} \ \Lambda' \qquad \Lambda' \subseteq \Lambda \qquad \Lambda \cup \{L\}; \Gamma \vdash e : C}{\Lambda; \Gamma \vdash \mathbf{ensure} \ L \ e : C}$$

Type Soundness

Sequence of active layers

- Reduction: $L_1;\ldots;L_n \vdash e \rightarrow e'$
- Thm. (Subject Reduction):
  - If $\{L_1,\ldots,L_n\}; \Gamma \vdash e : C$ and $L_1;\ldots;L_n \vdash e \rightarrow e'$ and $L_1;\ldots;L_n$ is well formed, then $\exists D. \{L_1,\ldots,L_n\}; \Gamma \vdash e' : D$ and $D <: C$
- Thm. (Progress)
  - If $\{L_1,\ldots,L_n\}; \cdot \vdash e : C$, then e is a value or $\exists e'. e \rightarrow e'$

Type So

Sequence of active layers

- Reduction: $L_1;\ldots;L_n \vdash e$
- Thm. (Subject Reduction)
  - If $\{L_1,\ldots,L_n\};\ \Gamma \vdash e : C$ and $L_1;\ldots;L_n \vdash e \rightarrow e'$
    and $L_1;\ldots;L_n$ is well formed,
    then $\exists D.\ \{L_1,\ldots,L_n\};\ \Gamma \vdash e' : D$ and $D <: C$
- Thm. (Progress)
  - If $\{L_1,\ldots,L_n\};\ \cdot \vdash e : C$,
    then e is a value or $\exists e'.\ e \rightarrow e'$

- *Empty seq.* is wf.
- $L_1;\ldots;L_n$ is wf.
  if $L_1;\ldots;L_{n-1}$ is wf.
  and L_n (**req**; $\subseteq$) $\{L_1,\ldots,L_{n-1}\}$

Plan of the Talk

- COP Language Constructs
- Type System
- Discussion
  - Activation constructs
  - Related work
- Conclusion and Future Work

Constructs for (de)activation

The original version of ContextFJ, as well as implementation (ContextJ, JCOP) has different constructs for (de)activation

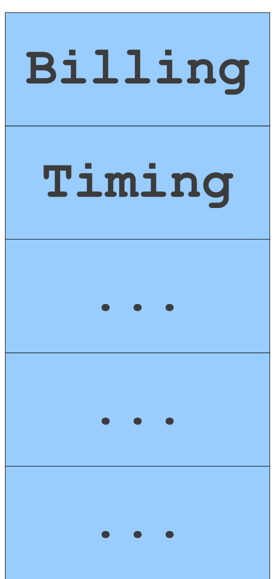
- `with` for activation
- `without` for (block-structured) deactivation

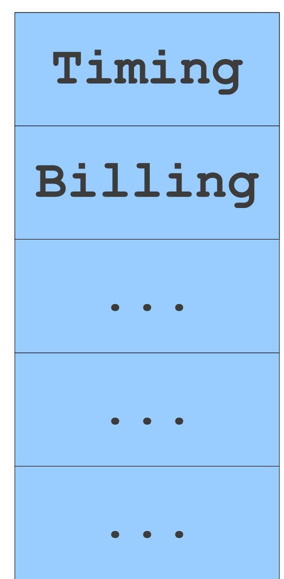
Comparing **ensure** and **with**

- Difference emerges when the same layer is to be activated twice

```
ensure Timing {
  ensure Billing {
    ensure Timing {
```

```
with Timing {
  with Billing {
    with Timing {
```



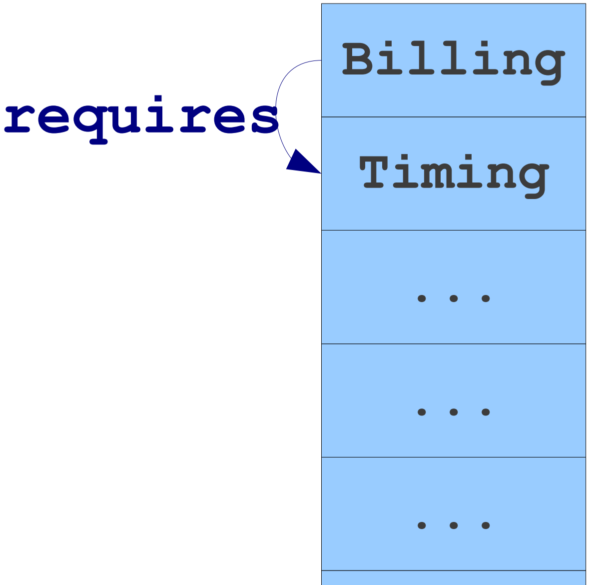


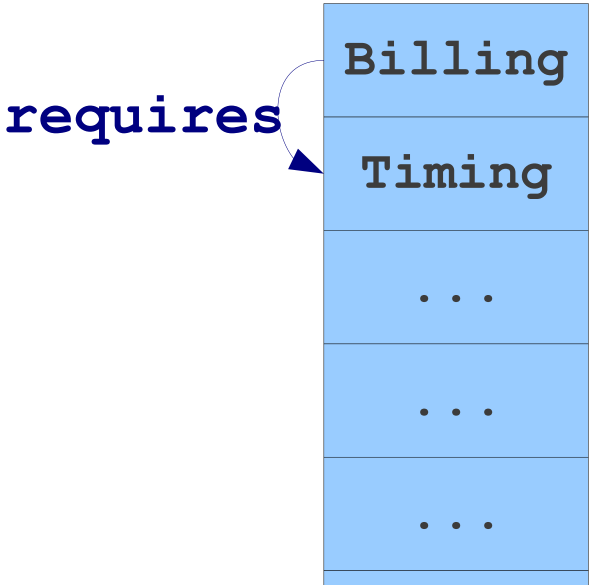
Why not **with** and **without**?

- Because they break the invariant (i.e. well-formedness) enforced by the type system!

```
ensure Timing {
  ensure Billing {
    without Timing {
```

```
with Timing {
  with Billing {
    with Timing {
```





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
    - But it's global and unique

Conclusion

Type system for dynamic layer composition

- Estimation of (globally) activated layers at each program point
- Explicit `requires` clauses to help typechecking
- Soundness proof

Future work: Dealing with the originally proposed layer activation mechanism (`with` and `without`)