L

Context-oriented Programming With Only Layers

Robert Hirschfeld
Hasso-Plattner-Institute
University of Potsdam
Germany
hirschfeld@hpi.uni-potsdam.de

Hidehiko Masuhara
Mathematical and Computing Sciences
Tokyo Institute of Technology
Japan
masuhara@acm.org

Atsushi Igarashi
Graduate School of Informatics
Kyoto University
Japan
igarashi@kuis.kyoto-u.ac.jp

ABSTRACT

Most if not all extensions to object-oriented languages that allow for context-oriented programming (COP) are asymmetric in the sense that they assume a base implementation of a system to be composed into classes and a set of layers to provide behavioral variations applied to those classes at run-time. We propose L as an experimental language to further explore the design space for COP languages. In this position paper we talk about first steps towards the unification of classes and layers and with that the removal of the asymmetry in composition mechanisms of contemporary COP implementations.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming

General Terms

Languages, Design

Keywords

Context-oriented programming, modularity, layers, software composition, sideways composition

1. INTRODUCTION

There are several proposals to extend programming languages in order to support context-oriented programming (COP) [5]. While most COP extensions augment object-oriented languages, the general approach to introducing context-dependent behavioral variations by offering means for changing the computation modules already defined in a particular base language is more widely applicable.

For COP extensions such as ContextS [4], ContextJS [14], or ContextFJ [6], such modules whose behavior can be adjusted are objects (classes, instances, or both) defined in Smalltalk [3], JavaScript, or Featherweight Java [8] respectively. Here, layers mainly provide partial method definitions that are composed into classes and so affect the behavior of a system.

That incremental approach to COP is practical because it allows for existing code artifacts such as frameworks and libraries written in the original base language to not only be compatible with and so usable in systems based on such COP language extensions, but also to be affected by them.

While ContextS is used in prototypes to explore core ideas of COP in Squeak/Smalltalk [9], a more extensive ContextJS code base contributed to both kernel-level and end-user projects in Webwerkstatt [13] and Lively Kernel [10]. On the more formal side, the core calculus of ContextFJ benefits from the effort that went into Featherweight Java to provide a lightweight semantics for a sufficiently expressive COP language.

So far, language support for context-oriented programming (COP) is asymmetric in that it only adds constructs to the modularity mechanisms a particular language already provides. This leaves language designers with yet another language feature to deal with both in isolation and in interaction with all others.

While we are aware of how rich languages are perceived to provide benefits to their users, we are interested in the opposite side of the spectrum of language design, where small kernels with only a few concepts are at the core of a system.

To simplify COP, we propose to remove classes from the language and to provide *layers* as the *sole construct* for defining and composing behavior and behavioral variations. This effort is based on our assumption that the abstraction and composition mechanisms offered by layers are sufficient to subsume those provided by classes in plain object-oriented languages.

Our research prototype to explore this idea is L, an experimental programming language that is based on our previous work on ContextFJ [6, 7] to provide a minimal core calculus for COP.

In the following we will give a short introduction to L by discussing the translation of two examples taken from earlier articles on a semantics [6] and a type system for dynamic layer composition [7].

Since we are still looking at several issues still to be clarified and resolved, we will describe only some of the ideas of our ongoing work in this paper to illustrate how COP programs written in L might look like (using two different proposals for L) and what problems in its language and sys-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
COP '13, Montpellier, France
Copyright 2013 ACM 978-1-4503-2040-5 ...$15.00.

RIGHTSLINK

tem design still need to be addressed.

2. LAYERS

COP extensions that are based on object-oriented programming languages provide layers of partial method definitions to organize variations to the behavior defined in classes of a base system. Such partial methods can be run around or instead of methods defined in a base-level class or other partial methods provided by other layers.

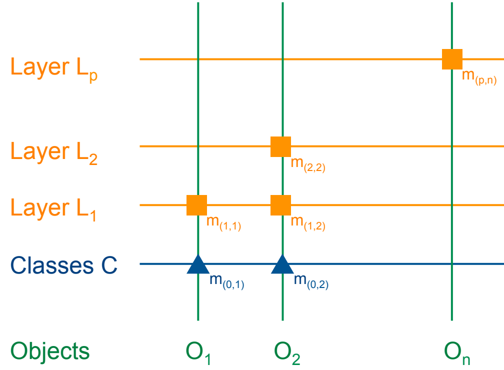

Figure 1: Layer Composition With Classes.

Such layer composition with classes is shown in Fig. 1. There are two methods ($m_{(0,1)}$ and $m_{(0,2)}$) implemented by classes and four partial methods ($m_{(1,1)}$, $m_{(1,2)}$, $m_{(2,2)}$, and $m_{(p,n)}$) that belong to layers. Here, $m_{(1,1)}$ refines $m_{(0,1)}$, and $m_{(2,2)}$ refines $m_{(1,2)}$, which itself is a refinement of $m_{(0,2)}$. Partial method $m_{(p,n)}$ is special in that it does not refine any other base-level or partial definition but introduces new signatures instead.

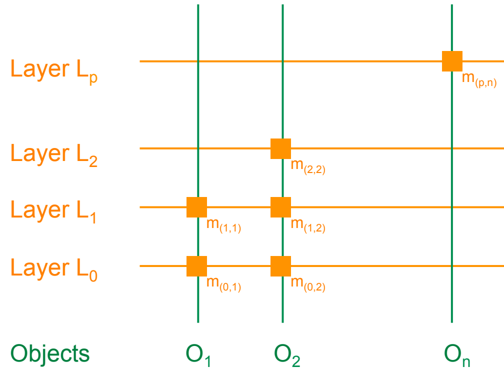

Figure 2: L-style Layer Composition.

In L there are *no classes*. All behavior is described using layer-based partial method definitions. Base-level method definitions that would have been associated with classes in most other COP systems are now associated with just another layer.

Fig. 2 illustrates L-style layer composition. The main difference to the composition shown in Fig. 1 is that methods $m_{(0,1)}$ and $m_{(0,2)}$ are now provided via Layer L_0, which we introduced as a replacement for all class-based method definitions previously associated with classes from C.

We will now translate a ContextFJ example [6] to L. Commentary will be informal.

```
object Person {
  String name, residence, employer;
  Person(String _name, String _residence,
         String _employer) {
    name = _name;
    residence = _residence;
    employer = _employer;
  }
}

object Student extends Person {
  String major;
  Person(String _name, String _residence,
         String _employer, String _major) {
    super(_name, _residence, _employer);
    major = _major;
  }
}
```

Listing 1: Objects Defining State.

Object `Person` in Lst. 1 defines three fields named `name`, `residence`, and `employer`. All fields will be initialized during instance creation. Note that with describing the fields and the constructor our object definition is complete since methods are defined elsewhere (in layers). Object `Student` then *refines* `Person` by defining yet another field (`major`) and a constructor to assist its initialization.

```
layer LPerson {
  object Person {
    String toString() {
      return "Name: " + name;
    }
  }
}

layer LContact {
  object Person {
    String toString() {
      return proceed()
             + "; Residence: "
             + residence;
    }
  }
}

layer LEmployment {
  object Person {
    String toString() {
      return proceed()
             + "; Affil.: "
             + employer;
    }
  }
}
```

Listing 2: Layer Definitions.

We now define the behavior of `Person` (Lst. 2). In our example, we have three layers `LPerson`, `LContact`, and `LEmployment` and all of them provide a partial implementation of method `toString()`. `LPerson`'s variant of `toString()` defines the basic behavior to simply return a `Person`'s `name`. `LContact`'s and `LEmployment`'s variants add to other partial definitions of `toString()` by first calling such definitions via `proceed()` and then appending either the `Person`'s `residence` or `employer` respectively.

RIGHTSLINK Copyright Clearance Center

Instantiating objects and interacting with them works the same way as in our class-based ContextFJ example [6]. A sample transcript is depicted in Lst. 3.

We start <1> by creating a new instance of `Person` and print out its string representation using an instance of a `Printer` object. Since there is no other but the `LCommon` layer active at this point in time, the result of the `println(..)` message is `"an Instance"`. After activating the `LPerson` layer <2>, `println(..)` leads to running the partial method definition provided there, which prints `"Name: Atsushi"`.

```
// <1>
Person atsushi =
  new Person("Atsushi", "Kyoto", "Kyodai");
new Printer().println(atsushi);
==> "an Instance"

// <2>
with LPerson {
  new Printer().println(atsushi);
}
==> "Name: Atsushi"

// <3>
with LPerson {
  with LContact {
    new Printer().println(atsushi);
  }
}
==> "Name: Atsushi; \
     Residence: Kyoto"

// <4>
with LPerson {
  with LEmployment {
    with LContact {
      new Printer().println(atsushi);
    }
  }
}
==> "Name: Atsushi; \
     Affil.: Kyodai; \
     Residence: Kyoto"

// <5>
with LPerson {
  with LContact {
    with LEmployment {
      new Printer().println(atsushi);
    }
  }
}
==> "Name: Atsushi; \
     Residence: Kyoto; \
     Affil.: Kyodai"

// <6>
with LPerson {
  with LContact {
    without LContact {
      new Printer().println(atsushi);
    }
  }
}
==> "Name: Atsushi"
```

Listing 3: Instance Creation and Interaction.

When we go on and start nesting layer activations as in <3>, <4>, or <5>, some of the partial method difinitions `proceed()` to the next layer and lead to augmenting the results of subsequent activations of those partial methods. Note from <4> and <5> that the activation order of layers directly corresponds to the execution sequence of partial method definitions. In <6> we can see that by using `without` layers can be retracted from a composition.

For bootstrapping this and the next example, we made the following assumptions.

To ensure that every object exhibits a set of base behaviors, we introduce a `Base` object, which is our root of the object hierarchy (Lst. 4). If an object does not explicitly extend another object, it implicitly extends `Base`. Behavior for `Base` and with that common behavior for all objects is defined in Layer `LCommon`—here a default implementation of `toString()`.

```
object Base {}

layer LCommon {
  object Base {
    String toString () {
      return "an Instance";
    }
  }
}

with LCommon {
  // [[read-eval-print|main]]
}
```

Listing 4: Common Definitions for Base.

To ensure that the behavior defined in `LCommon` is available to all objects, this layer is deployed via `with` around a location central to the execution of a script or program such as the `read-eval-print` loop of the interpreter or the entry points (such as main) into the system.

```
object Printer {}

layer LCommon {
  object Printer {
    void println(Base o) {
      // PRIMITIVE_println(o.toString());
    }
  }
}
```

Listing 5: Common Definitions for Printer.

We also made `Printer`'s `println(..)` available via the `LCommon` layer (Lst. 5). So far, `LCommon` and the objects it directly applies to can be considered to be part of L's runtime.

3. STATE DECLARATION

Allowing partial class definitions to introduce object fields is of great interest to us since a feature like that would help decreasing dependencies between such definitions by reducing centers [1] (such as those for state declarations) and so allow for layers that are more robust.

In our next example, based on ContextFJ code from our 2012 paper on a type system for dynamic layer composition [7], we describe one of several approaches to decentralized state declaration we are investigating for future versions

RIGHTSLINK

of L. (Note that the version of L used here is slightly different from that in the previous example: State definitions of an object can be distributed across several layers and there are only no-arg constructors.)

The basic idea is that *fields of an object can be declared redundantly*. While every partial definition of an object can repeat the declaration of a field, all the declarations refer to the same field.

```
layer LCustomer {
  object Customer {
    String name;
    void setName(String _name) {
      name = _name;
    }
  }
}

layer LConnection {
  object Connection {
    Customer from, to; // <1>
    void setFrom(Customer _from) {
      from = _from;
    }
    void setTo(Customer _to) {
      to = _to;
    }
    void complete() { ... }
    void drop() { ... }
  }
}

layer LTiming {
  object Timer {
    void start() { ... }
    void stop() { ... }
    int getTime() { ... }
  }
  object Connection {
    Timer timer; // <2>
    void setTimer(Timer _timer) {
      timer = _timer;
    }
    void complete() {
      proceed();
      timer.start();
    }
    void drop() {
      timer.stop();
      proceed();
    }
    int getTime() {
      return timer.getTime();
    }
  }
}

layer LBilling { // requires LTiming
  object Connection {
    Timer timer; // <3>
    void setTimer(Timer _timer) {
      timer = _timer;
    }
    void charge() {
      // cost =  ...getTime()...;
      // ...charge cost on caller...
    }
    void drop() {
      proceed();
      charge();
    }
  }
}
```

Listing 6: Redundant Field Declarations.

In this example, partial layers, classes, and methods are defined similarly to the previous ones (Sec. 2). The difference we can see here (Lst. 6) is that fields, such as those of `Connection`, can be put in several of its partial definitions (like `from`, `to`, and `timer` in <1>, <2>, and <3>), and that some of them, while denoting the same field, can appear redundantly (like `timer` in <2> and <3>).

```
// for convenience, available from somewhere
Connection simulate() {
  Customer a = new Customer();
  a.setName("Atsushi");
  Customer h = new Customer();
  h.setName("Hidehiko");
  Connection c = new Connection();
  c.setFrom(a);
  c.setTo(h);
  c.complete();
  c.drop();
  return c;
}

with LTiming {
  with LBilling {
    Connection c = simulate();
    new Printer().println(c.getTime());
  }
}
==> "712" // seconds <sigh>
```

Listing 7: Instance Creation and Interaction.

The workspace for composing and using our objects and layers looks similar to that of the previous example (Lst. 7). `simulate()` is there only for convenience and can be located elsewhere.

4. RELATED WORK

Here we would like to mention a few other approaches to language design, which also aim at limiting the core concepts of the language and also asymmetries in the modularity and composition mechanisms provided.

The work on hyperspaces and multi-dimensional separation of concerns [16], which lead to Hyper/J [15], tried to address the dominant decomposition problem by allowing features of a system to be organized along *dimensions* and implemented independently of each other as so-called *hyperslices*. Hyper/J systems do not start out from a particular set of base classes, but allow for the implementation of individual concerns that can be reasoned about and understood in isolation. Compared to L and other COP languages, Hyper/J's compositon of slices happens at compile-time.

Early versions of AspectJ [11]—back then one of the primary languages for research on aspect-oriented programming (AOP [12])—did not distinguish between classes and aspects. There were only aspects. This unification however has been subsequently given up. We do not know for sure about the reason, but assume that by having classes to which aspects can be added, the impact of tools like AspectJ can be much higher since they can be applied to existing systems written in the primary language the AOP extension is provided for.

Self [17] and Newspeak [2] are two other languages that inspired us—Self for its simplicity gained from removing

RIGHTSLINK

classes from Smalltalk (prototypes unify classes and their instances) and its uniform message-based access to slots and Newspeak for its consistent use of nested classes and late binding for modularity.

5. OUTLOOK

L is our experimental language for exploring COP modularity mechanisms. In this position paper we describe our first attempt to unify classes and layers and so remove the asymmetry of module constructs observed in other COP systems, where a base system is factored along static class hierarchies and dynamic variations of that are provided via layer composition.

In Sec. 2 we illustrate how a COP program, which relies only on layers to define its computation, might look like. We also describe a small runtime environment that provides common functionality (here the printing of objects) and a way to make its functionality available.

Using a slightly different version of L in Sec. 3, we sketch our idea on how to make state declaration more robust by allowing field declarations of an object to be distributed across several partial definitions and to be redundant.

We are working on several issues for consolidating L and making it more practical including object initialization, stateful layers, object initialization, subtractive compositions via `without`, and the interaction between state extension and layer refinement.

L is in an early stage. There are many open questions and to many of them we still do not have good answers yet. But hopefully L will converge to something that might be more helpful in investigating interesting modularity and langauge design problems.

6. REFERENCES

[1] C. Alexander. *The Nature of Order: An Essay on the Art of Building and the Nature of the Universe.* Routledge, 2004.

[2] G. Bracha, P. von der Ahé, V. Bykov, Y. Kashai, W. Maddox, and E. Miranda. Modules as Objects in Newspeak. In *Proceedings of ECOOP'10*, volume 6183 of *Lecture Notes in Computer Science*, pages 405–428. Springer, 2010.

[3] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[4] R. Hirschfeld, P. Costanza, and M. Haupt. An Introduction to Context-oriented Programming with ContextS. In *Generative and Transformational Techniques in Software Engineering II*, volume 5235 of *Lecture Notes in Computer Science*, pages 396–407. Springer, 2008.

[5] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented Programming. *Journal of Object Technology*, 7(3):125–151, 2008.

[6] R. Hirschfeld, A. Igarashi, and H. Masuhara. ContextFJ: A Minimal Core Calculus for Context-oriented Programming. In *Proceedings of FOAL'11.* ACM, 2011.

[7] A. Igarashi, R. Hirschfeld, and H. Masuhara. A type system for dynamic layer composition. In *Proceedings of FOOL'12.* ACM, 2012.

[8] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001.

[9] D. H. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the Future: the Story of Squeak, a Practical Smalltalk Written in Itself. In *Proceedings of OOPSLA'97*, pages 318–326. ACM, 1997.

[10] D. H. Ingalls, K. Palacz, S. Uhler, A. Taivalsaari, and T. Mikkonen. The Lively Kernel A Self-supporting System on a Web Page. In *Self-Sustaining Systems*, volume 5146 of *Lecture Notes in Computer Science*, pages 31–50. Springer, 2008.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of ECOOP'01*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of ECOOP'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[13] J. Lincke. Webwerkstatt. `http://lively-kernel.org/repository/webwerkstatt/webwerkstatt.xhtml`. Accessed: 2013-04-04.

[14] J. Lincke, M. Appeltauer, B. Steinert, and R. Hirschfeld. An Open Implementation for Context-oriented Layer Composition in ContextJS. *Elsevier Journal on Science of Computer Programming, Special Issue on Software Evolution*, 2011.

[15] H. Ossher and P. L. Tarr. . *CACM*, 44(10):43–50, Oct. 2001.

[16] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. S. Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of ICSE'99*, pages 107–119. IEEE Computer, 1999.

[17] D. Ungar and R. B. Smith. Self: The Power of Simplicity. In *Proceedings of OOPSLA'87*, pages 227–242. ACM, 1987.

RIGHTSLINK