Principles, Patterns, and Techniques for Designing and Implementing Practical Fluent Interfaces in Java

Haochen Xie, Nagoya University
haochenx@acm.org

Abstract and poster available for downloading at
https://haochenxie.name/today/2017/Practical-Fluent-Interfaces-in-Java-Oct25

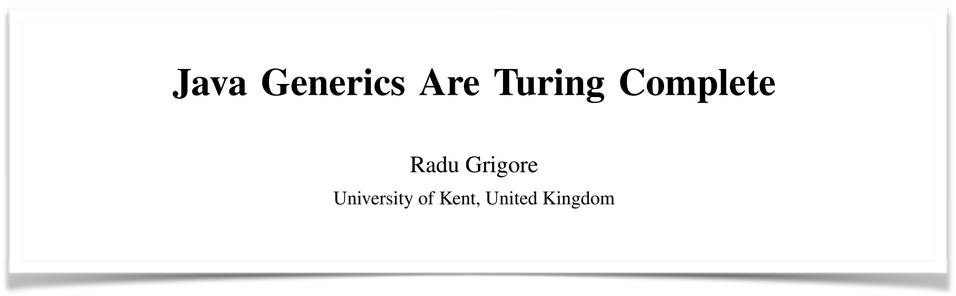
We can make internal DSLs in Java with the method chaining technique. We assume such internal DSLs are provided as Java libraries, and attribute their interfaces to be *fluent* and call such interfaces *fluent interfaces*. Fluent interface is the focus of this research.

The following code snippet using a graphics library with fluent interface generates the funny figure above.

```
// note that match of calls to newText(),
// endText() etc are enforced by the Java type checker
GroupObj example = BuilderFactory.createGroupBuilder()
    .newText().x(0).y(0).fontSize(10)
        .text("Hello, Java;")
        .horizontalAlignment(CENTER)
        .endText()
    .newGroup().x(0).y(0)
        .rotation(180)
        .newText().x(0).y(0).fontSize(9)
            .text("I can flow :D")
            .horizontalAlignment(CENTER)
            .endText()
        .endGroup()
    // the x axis goes right, and y axis goes down
    .newRectangle()
        .leftTop(-29, -12)
        .rightBottom(29, 12)
        .endRectangle()
    .buildGroup();
```

It is actually quite popular in the Java community to create fluent interfaces because, well, DSLs are beneficial in a lot of ways. There are many public resources on crafting fluent interfaces and examples in both software industry and academia could be seen.

We can use the Java type checker to impose a rigid grammar on such internal DSLs. Actually, recent researches tell us that we are so lucky that the grammar can be any Type-0 grammar (i.e. language enumerable by a Turing machine).



Java Generics Are Turing Complete

Radu Grigore
University of Kent, United Kingdom

Radu Grigore. 2017. Java Generics Are Turing Complete. POPL 2017

in this paper Grigore basically showed us how to literately run a Turing machine with the Java type checker :D

(Despite how theoretically interesting this idea seems,) yet it is evil to coerce the Java type checker to simulate a Turing machine every time we want to check the grammar of a single usage of the fluent interface. Why? well,

Firstly, unfortunately those Turing machines are very very **slow**.

Secondly, we would need an **external tool** to generate tons of code to simulate the Turing machine.

Thirdly, not to even mention that such generated code are almost **incomprehensible** and not seemingly very useful in debugging.

Not a surprise that there seems to be no library uses this technique. And unfortunately we don't yet have a comprehensive general metatheory that one could follow to guide the design and implementation of a non-trivial fluent interface. Thus comes this research.

Up to now, the results we obtained includes:

First, an **anatomy** for the conceptual establishment of fluent interfaces,

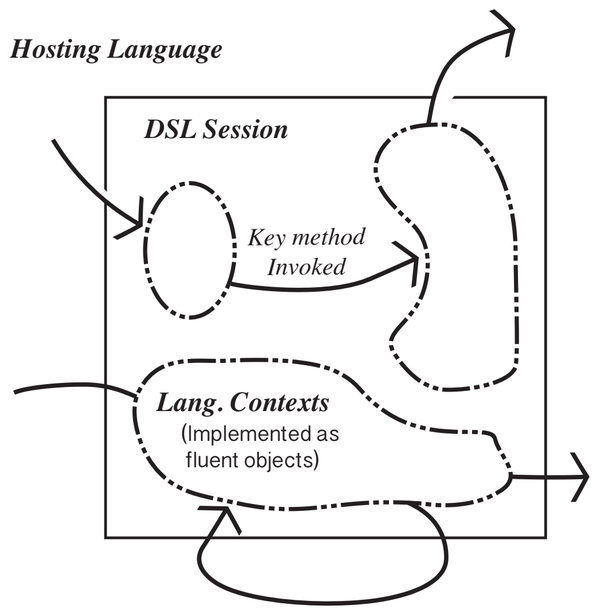
Second, six **primitives** that are essential or useful in fluent interfaces,

Third, the recognition of the important role of the **transition choreography** in the design theory for fluent interfaces, and

Fourth, a small collection of useful design or implementation **patterns** and **techniques** useful for crafting fluent interfaces

on the fourth result, come to APLAS and attend my poster for details! :D

Some part of the the *anatomy* we established could even be considered as a general contribution to the study of domain specific languages. To help illustrate the anatomy, we created a diagram and the very important concepts are explained.



- We utilize the method chaining technique to implement an internal DSL in Java, in the form of **fluent interface**.
- As usual, we call **Hosting Language** the language in which the internal DSL is implemented and used, i.e. Java in this case;
- and in this work, we differentiate each use of the internal DSL, and call each such usage a **DSL Session**.
- Each dialectic word in the DSL is simulated by a Java method, which we call them **key methods**;
- and the Java objects (who composing the fluent interface) that has one ore more key methods are called **fluent objects**.
- Method invocation chains in Java are statically type checked based on the class of the fluent objects. So within a method invocation chain, the current fluent object restricts the set of valid key methods; in other words, a fluent object implements a dialectic **Language Context**.

What we call a *primitive*, is a role that a key method could play in contributing to the fluent interface. Here are the six identified primitives. There might be more primitives but this set of six is already quite comprehensive and exerts excellent expressiveness.

Starter primitive: when such a method is called, a DSL session is **entered**.

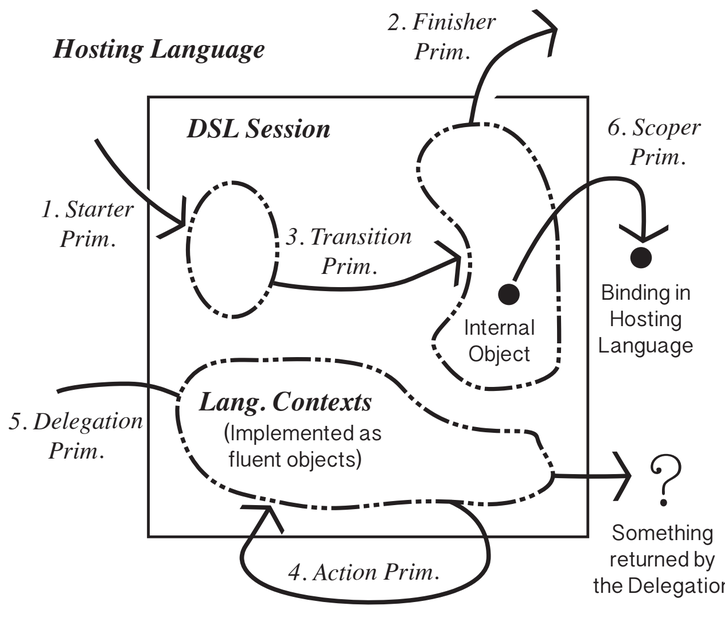
Finisher primitive: when such a method is called, a non-fluent object is returned, marking the **leave** from a DSL session.

Transition primitive: when such a method is called, another fluent object is returned, marking a **switch** to another language context.

Action primitive: when such a method is called, side effects are **triggered** based on the state of `this` and the arguments. `this` is then returned to facilitate further fluent calls.

Delegation primitive: as the name suggests, such a method receives a **delegation** and entrusts `this` to it. This primitive is especially useful when a collection of delegations is supplied.

Scoper primitive: such a method introduces a **binding** to an object internal to the DSL.



Adding those primitives to the anatomy diagram results a more exiting diagram above.

Among these six primitives, the design theory concerning the *transition primitive* is the most involved, as it is this primitive that allows us to stipulate the interplay among fluent objects and makes fluent interfaces rich in expressiveness.

We thus coined an alias for the intricate yet interesting interplay among fluent objects, intermediated by the transition primitive: the *Transition Dance* and attributed it as at the core of the design theory for fluent interface design in Java. So comes the slogan:

Transition choreography is at the core!