Internal DSLs
from API design to languages

SoftLang Team, University of Koblenz-Landau

Prof. Dr. Ralf Lämmel
Msc. Johannes Härtel
Msc. Marcel Heinz
Fluent Interfaces

from https://martinfowler.com/bliki/FluentInterface.html

Fluent API
(Method Chaining + Other Beneficial Design Choices)

Pure OO-Style

(C) 2018, SoftLang Team, University of Koblenz-Landau
Fluent Interfaces
(Object-Orientation)

Joe wants API users to write concise code.

Fluent API
(Method Chaining + Other Beneficial Design Choices)

(C) 2018, SoftLang Team, University of Koblenz-Landau
Fluent Interfaces
(Software Language Engineering)

Hey, this looks like an independent format for orders.

Fluent API
(Method Chaining + Other Beneficial Design Choices)

(C) 2018, SoftLang Team, University of Koblenz-Landau
Technological Spaces


(C) 2018, SoftLang Team, University of Koblenz-Landau
Hey, this looks like an independent format for orders.
Chrestomathies @ Softlang

- **101companies** exists for demonstrating programming technologies in general by implementing a **business scenario**.
- **Metalib** exists for demonstrating programming technologies in the technical domain of implementing **domain-specific languages**.
  - Java internal DSL
  - Python internal DSL
Domain-Specific Language (DSL) – Definition

“A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.”


(C) 2018, SoftLang Team, University of Koblenz-Landau
Domain-specific vs. General purpose

- **Domain** Only DSLs have a relatively small and well-defined domain.
- **Language size** GPLs are large. DSLs are typically small.
- **Turing completeness** DSLs may not be Turing complete.
- **Lifespan** GPLs live for years to decades. DSLs may live for months only.
- **Designed by** GPLs are designed by gurus or committees. DSLs are designed by a few software engineers and domain experts.
- **Evolution** GPLs evolve slowly. The evolution of DSLs is fast-paced.
- **Deprecation/incompatible changes** This is almost impossible for GPLs; it is feasible and relatively common for DSLs.


(C) 2018, SoftLang Team, University of Koblenz-Landau
A DSL for finite state machines (FSMs)

Imagine the FSM language (FSML) to have started on the black/whiteboard a long time ago.
Concepts of FSML illustrated for the turnstile FSM

States
locked  The turnstile is locked. No passenger is allowed to pass.
unlocked The turnstile is unlocked. A passenger may pass.
exception A problem has occurred and metro personnel needs to intervene.

Events
ticket  A passenger inserts a ticket into the card reader.
pass    A passenger passes the turnstile as noticed by a sensor.
mute    Metro personnel turns off alarm after exception.
release  Metro personnel turns on normal operation again.

Actions
collect The ticket is collected by the card reader.
eject   The ticket is ejected by the card reader.
alarm   An alarm is turned on and metro personnel is requested.

Transitions
Semantics (I/O behavior) of FSML illustrated for the turnstile FSM

Input (= sequence of events)

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ticket</td>
<td>A ticket is inserted. (The turnstile is unlocked, thus.)</td>
</tr>
<tr>
<td>ticket</td>
<td>Another ticket is inserted. (The superfluous ticket is ejected.)</td>
</tr>
<tr>
<td>pass</td>
<td>Someone passes the turnstile. (This is Ok.)</td>
</tr>
<tr>
<td>pass</td>
<td>Someone passes the turnstile. (This triggers alarm.)</td>
</tr>
</tbody>
</table>

Output (= sequence of actions)

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>collect</td>
<td>The inserted ticket is collected.</td>
</tr>
<tr>
<td>eject</td>
<td>A ticket inserted in unlocked state is ejected.</td>
</tr>
<tr>
<td>alarm</td>
<td>An attempt to pass in locked state triggers alarm.</td>
</tr>
</tbody>
</table>
DSL implementation in different ‘styles’

- **External DSL:**
  Designated parser, checker, interpreter, compiler

- **Internal DSL:**
  Implementation as library using host language features

N.B.: This is a gross oversimplification. There are options or hybrids using extensible languages, extensible compilers, metaprogramming systems, and language workbenches.
We are going to do here …

**Internal DSL style**

with **Java** and **Python** libraries

N.B.: If we were using C++, Scheme, Haskell, or others for internal DSL implementation, additional or different techniques could or should be leveraged, e.g., operator overloading, macros, or templates.
turnstile = new Fsm();
State s = new State();
s.setStateid("locked");
s.setInitial(true);
turnstile.getStates().add(s);
s = new State();
s.setStateid("unlocked");
turnstile.getStates().add(s);
s = new State();
s.setStateid("exception");
turnstile.getStates().add(s);
Transition t = new Transition();
t.setSource("locked");
t.setEvent("ticket");
t.setAction("collect");
t.setTarget("unlocked");
turnstile.getTransitions().add(t);
t = new Transition();
... add more transitions ...
Java API with functional constructors

turnstile = new Fsm();
turnstile.getStates().add(new State("locked", true));
turnstile.getStates().add(new State("unlocked"));
turnstile.getStates().add(new State("exception"));
turnstile.getTransitions().add(new Transition("locked", "ticket", "collect", "unlocked"));
turnstile.getTransitions().add(new Transition("locked", "pass", "alarm", "exception"));
... add more transitions ...

N.B.: Functional constructors have been used by C++ and Java et al. programmers for a long time, but they are insufficient to avoid repetitive code and to hide the internal representation.
Basic Java object model
including functional constructors

```java
public class Fsm {
    private List<State> states = new LinkedList<>();
    private List<Transition> transitions = new LinkedList<>();
    public List<State> getStates() { return states; }
    public List<Transition> getTransitions() { return transitions; }
}

public class State {
    private String id;
    private boolean initial;
    public String getStateid() { return id; }
    public void setStateid(String state) { this.id = state; }
    public boolean isInitial() { return initial; }
    public void setInitial(boolean initial) { this.initial = initial; }
    public State() {
    }
    public State(String id) { this.id = id; }
    public State(String id, boolean initial) { this.id = id; this.initial = initial; }
}

public class Transition {
    private String source;
    private String event;
    private String action;
    // Constructor and methods...
}
```

It’s easy, but not what we want.
Use of a fluent API in Java

Fsm turnstile =
  fsm()
  .addState("locked")
  .addTransition("ticket", "collect", "unlocked")
  .addTransition("pass", "alarm", "exception")
  .addState("unlocked")
  .addTransition("ticket", "eject", "unlocked")
  .addTransition("pass", null, "locked")
  .addState("exception")
  .addTransition("ticket", "eject", "exception")
  .addTransition("pass", null, "exception")
  .addTransition("mute", null, "exception")
  .addTransition("release", null, "locked");

Leveraged techniques:
- Factory methods
- Method chaining
- Implicit parameters
- Conventions (defaults)

N.B.: The current state is maintained along the way.
The state declared first is assumed to be the initial one.
The representation is not revealed—no constructors are used.
Use of a fluent API in Python

```python
turnstile = Fsm()
    .addState("locked")
        .addTransition("ticket", "collect", "unlocked")
        .addTransition("pass", "alarm", "exception")
        .addState("unlocked")
        .addTransition("ticket", "eject", "unlocked")
        .addTransition("pass", None, "locked")
    .addState("exception")
        .addTransition("ticket", "eject", "exception")
        .addTransition("pass", None, "exception")
        .addTransition("mute", None, "exception")
        .addTransition("release", None, "locked")
```

Leveraged techniques:
- Factory methods
- Method chaining
- Implicit parameters
- Conventions (defaults)

N.B.: If we were using C++, Scheme, Haskell, or others for internal DSL implementation, additional or different techniques could or should be leveraged, e.g., operator overloading, macros, or templates.

© 2017 Ralf Lämmel, Software Languages Team, http://www.splang.org, University of Koblenz-Landau. All rights reserved.
Definition of fluent API in Java

```java
public interface Fsm {
    public Fsm addState(String state);
    public Fsm addTransition(String event, String action, String target);
    public String getInitial();
    public ActionStatePair makeTransition(String state, String event);
}

public class ActionStatePair {
    public String action;
    public String state;
}
```

N.B.: This interface does not expose the internal representation. The interface does not just cover fluent construction; it also covers ‘observation’ of the opaque representation.
Implementation of fluent API in Java

```java
public class FsmImpl implements Fsm {
    private String initial; // the initial state
    private String current; // the "current" state
    // A cascaded map for maintaining states and transitions
    private HashMap<String, HashMap<String, ActionStatePair>> fsm =
        new HashMap<>();
    private FsmImpl() { }
    // Construct FSM object
    public static Fsm fsm() { return new FsmImpl(); }
    // Add state and set it as current state
    public Fsm addState(String id) {
        // First state is initial state
        if (initial == null) initial = id;
        // Remember state for subsequent transitions
        this.current = id;
        if (fsm.containsKey(id)) throw new FsmIDistinctIdsException();
        fsm.put(id, new HashMap<String, ActionStatePair>());
        return this;
    }
    // Add transition for current state
    public Fsm addTransition(String event, String action, String target) {
        // Code...
    }
}
```
if (fsm.containsKey(id)) throw new FsmlDistinctIdsException();
fsm.put(id, new HashMap<String, ActionStatePair>());
return this;
}

// Add transition for current state
public Fsm addTransition(String event, String action, String target) {
    if (fsm.get(current).containsKey(event)) throw new FsmlDeterminismException();
    ActionStatePair pair = new ActionStatePair();
    pair.action = action;
    pair.state = target;
    fsm.get(current).put(event, pair);
    return this;
}

// Getter for initial state
public String getInitial() {
    return initial;
}

// Make transition
public ActionStatePair makeTransition(String state, String event) {
    if (!fsm.containsKey(state)) throw new FsmlResolutionException();
    if (!fsm.get(state).containsKey(event)) throw new FsmlInfeasibleEventException();
    return fsm.get(state).get(event);
}
A *JUnit* test case for simulation

```java
public class FluentTest {

    private static final String[] input = {
            "ticket", "ticket", "pass", "pass", "ticket", "mute", "release"};

    private static final String[] output = {
            "collect", "eject", "alarm", "eject"};

    @Test
    public void runSample() {
        assertArrayEquals(output, run(Sample.turnstile, input));
    }
}
```

N.B.: This is how a Java programmer (a DSL user) would document a use case of a specific FSM (and validate intuitions).
public class FsmlInterpreter {
    public static String[] run(Fsm fsm, String[] input) {
        ArrayList<String> output = new ArrayList<>();
        String state = fsm.getInitial();
        for (String event : input) {
            ActionStatePair pair = fsm.makeTransition(state, event);
            if (pair.action != null) output.add(pair.action);
            state = pair.state;
        }
        return output.toArray(new String[output.size()]);
    }
}

N.B.: The interpreter essentially models the dynamic semantics of FSML. This is a non-interactive interpreter. In practice, an interactive DSL implementation may be required.
Implementation of fluent API in Python

class Fsm():
    def __init__(self):
        self.fsm = defaultdict(list)
        self.current = None
    def addState(self, id):
        return self.addStateNoDefault(self.current is None, id)
    def addStateNoDefault(self, initial, id):
        if id in self.fsm[id]: raise FsmDistinctIdsException;
        self.stateObject = dict()
        self.stateObject['transitions'] = defaultdict(list)
        self.stateObject['initial'] = initial
        self.fsm[id] += [self.stateObject]
        self.current = id
        return self
    def addTransition(self, event, action, target):
        if event in self.stateObject['transitions']: raise FsmDeteminismException;
        self.stateObject['transitions'][event] += 
        [(action, self.current if target is None else target)]
        return self

N.B.: no high-level API is provided for ‘observation’, one would access the dictionary directly.
def run(fsm, input):
    # Determine initial state
    for id, [decl] in fsm.iteritems():
        if decl["initial"]:
            current = decl
            break
    # Consume input; produce output
    output = []
    while input:
        event = input.pop(0)
        if event not in current["transitions"]: raise FsmInfeasibleEventException
        else:
            [(action, target)] = current["transitions"][event]
            if action is not None: output.append(action)
            if target not in fsm: raise FsmResolutionException
            [current] = fsm[target]
    return output

N.B.: When compared to the Java-based interpreter, we access directly the presentation.
‘Minimum’ DSL implementation

✓ Syntax (fluent API for internal DSL)
✓ (Dynamic) semantics (e.g., by means on an interpreter)
  - Well-formedness / -typedness (aka static semantics)

N.B.: Just like the interpreter, we implement a ‘well-formedness checker’ as functionality on top of (the API for) the internal DSL representation. (We could use a constraint language such as OCL.)
Well-formedness of FSMs

distinctStateIds  The state ids of the state declarations must be distinct.
singleInitialState  An FSM must have exactly one initial state.
deterministicTransitions  The events must be distinct per state.
resolvableTargetStates  The target state of each transition must be declared.
reachableStates  All states must be reachable from the initial state.

```
resolutionNotOk = \n  Fsm() \n  .addState("stateA") \n  .addTransition("event1", "action1", "stateB") \n  .addTransition("event2", "action2", "stateC") \n  .addState("stateB")
```

N.B.: This sample violates `resolvableTargetStates`.

N.B.: a violated `resolvableTargetStates` can (should) be detected even before running an FSM on a specific input.
def ok(fsm):
    for fun in [
        distinctStateIds,
        singleInitialState,
        deterministicTransitions,
        resolvableTargetStates,
        reachableStates ]: fun(fsm)

N.B.: Violations of distinctStateIds and deterministicTransitions can be detected during construction, but we may need explicit checks if we also accommodate ‘serialization’.

def distinctStateIds(fsm):
    for state, decls in fsm.iteritems():
        if not len(decls) == 1: raise FsmlDistinctIdsException()

def singleInitialState(fsm):
    initials = [initial for initial, [decl] in fsm.iteritems() if decl["initial"]]
    if not len(initials) == 1: raise FsmlSingleInitialException()

def deterministicTransitions(fsm):
    for state, [decl] in fsm.iteritems():
        for event, transitions in decl["transitions"].iteritems():
            if not len(transitions) == 1: raise FsmlDeterminismException()

def resolvableTargetStates(fsm): ...
def singleInitialState(fsm):
    initials = [initial for initial, [decl] in fsm.iteritems() if decl["initial"]]
    if not len(initials) == 1: raise FsmlSingleInitialStateException()

def deterministicTransitions(fsm):
    for state, [decl] in fsm.iteritems():
        for event, transitions in decl["transitions"].iteritems():
            if not len(transitions) == 1: raise FsmlDeterminismException()

def resolvableTargetStates(fsm):
    for _, [decl] in fsm.iteritems():
        for _, transitions in decl["transitions"].iteritems():
            for (_, target) in transitions:
                if not target in fsm: raise FsmlResolutionException()

def reachableStates(fsm):
    for initial, [decl] in fsm.iteritems():
        if decl["initial"]: reachables = set([initial])
        chaseStates(initial, fsm, reachables)
        if not reachables == set(fsm.keys()): raise FsmlReachabilityException()

# Helper for recursive closure of reachable states
def chaseStates(source, fsm, states): ...
Online resources

YAS’ GitHub repository contains all code.
YAS (Yet Another SLR (Software Language Repository))
http://www.softlang.org/yas
See here specifically:
https://github.com/softlang/yas/tree/master/languages/FSML
Subdirectories Java and Python

The Software Languages Book
http://www.softlang.org/book
The topic is covered in Chapter 2.
DSL Implementation

Language implementation

Syntax
- Abstract syntax
- Concrete syntax
  - Textual syntax
  - Graphical syntax
- Parsing

Semantics
- Dynamic semantics
- Static semantics
- Translation semantics
Metalib

- https://softlang.github.io/metalib/

(C) 2018, SoftLang Team, University of Koblenz-Landau
Outlook

- Simplified: Internal DSLs make use of an existing programming language environment.
- How to implement FSML as an own independent language?
  - Parsing
  - Code generation

(C) 2018, SoftLang Team, University of Koblenz-Landau