Software Design & Programming Techniques

Domain-Specific Languages

Prof. Dr-Ing. Klaus Ostermann
Sebastian Erdweg, Msc.

Domain-Specific Languages

- 5.1 Goal of domain-specific languages (DSLs)
- 5.2 Case Study: State Machines
- 5.3 Styles of DSLs
- 5.4 SugarJ
- 5.5 Summary: DSLs

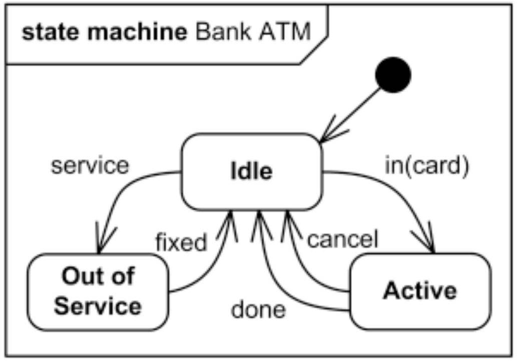
5.1 Goal of domain-specific languages (DSLs)

- Programming languages have fixed, built-in features
- These are generally useful features
- We speak of *general-purpose languages (GPL)*

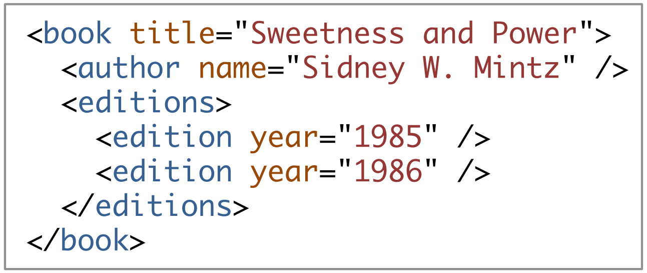
```
class MenuItem { // classes
   …
   void draw(Graphics g) { // methods
      …
   }
}

class CheckMenuItem extends MenuItem { // inheritance
   …
}

MenuItem i = new CheckMenuItem(“Activate?”); // objects
```

General-purpose vs. domain-specific

- General-purpose languages are often inadequate
- Using classes, methods, inheritance, and objects, how do you describe:
  - A *state machine* that implements an ATM?

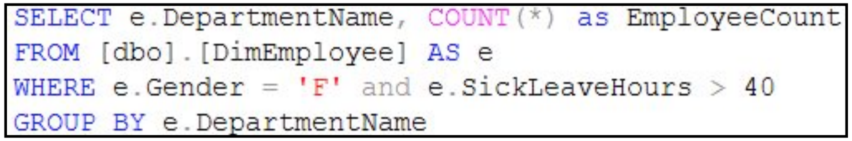


  - An *XML* document that represents books?



```
<book title="Sweetness and Power">
  <author name="Sidney W. Mintz" />
  <editions>
    <edition year="1985" />
    <edition year="1986" />
  </editions>
</book>
```

  - An *SQL query* that draws statistics on employees’ absence?



```
SELECT e.DepartmentName, COUNT(*) as EmployeeCount
FROM [dbo].[DimEmployee] AS e
WHERE e.Gender = 'F' and e.SickLeaveHours > 40
GROUP BY e.DepartmentName
```

Goal of Domain-specific languages (DSLs)

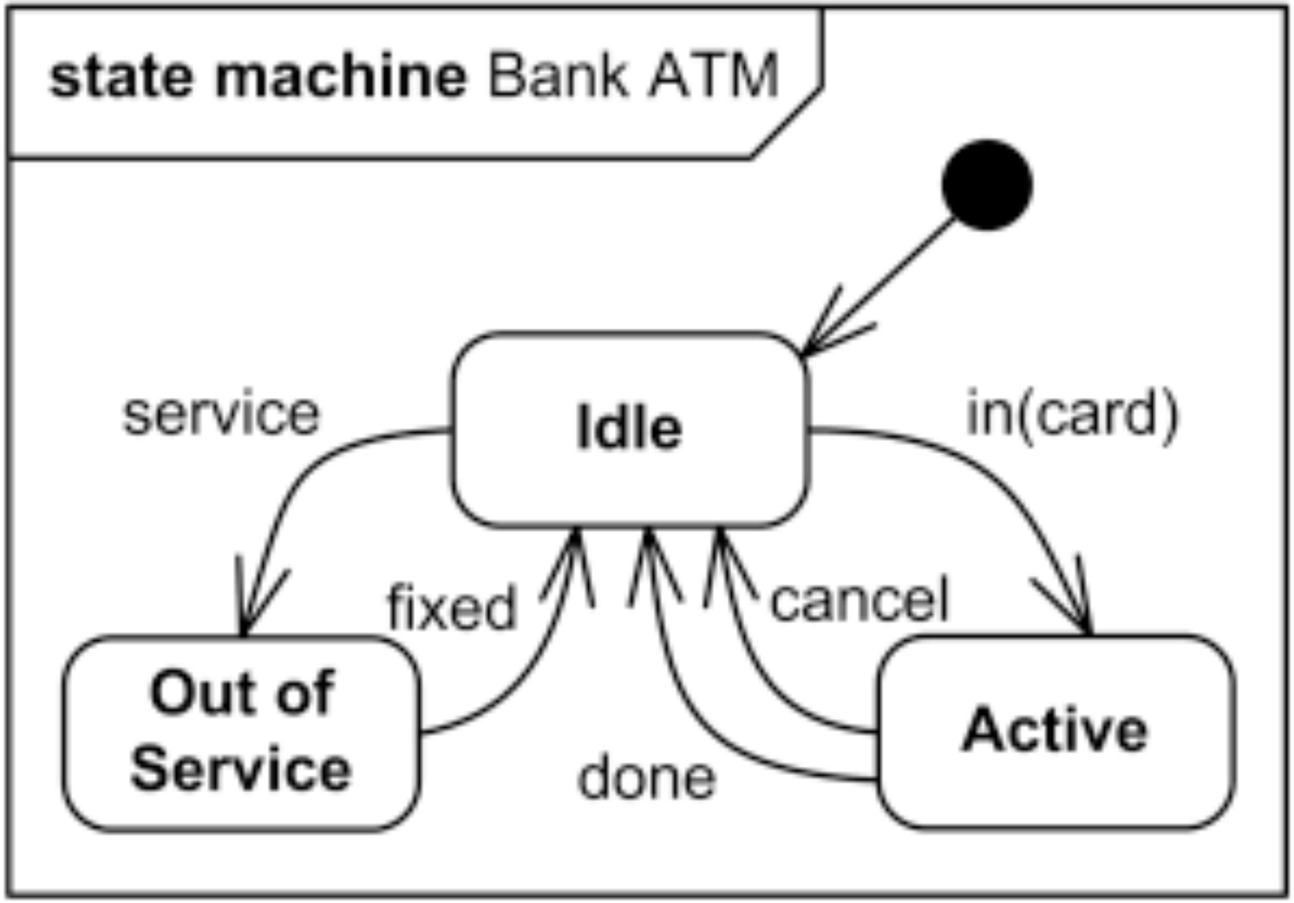
Narrow the gap between
a problem domain
and
its implementation

- Problem domains are
  - the domain an application targets (e.g., banking or telephone relaying)
  - all domains needed in the realization of the application (e.g., SQL)

- The implementation should be close to the domains to improve
  - conceptual proximity (thinking)
  - representational proximity (reading/writing)

5.2 Case Study: State Machines

- To illustrate the inadequacy of general-purpose languages, let us implement a state machine in Java



How do we do that?

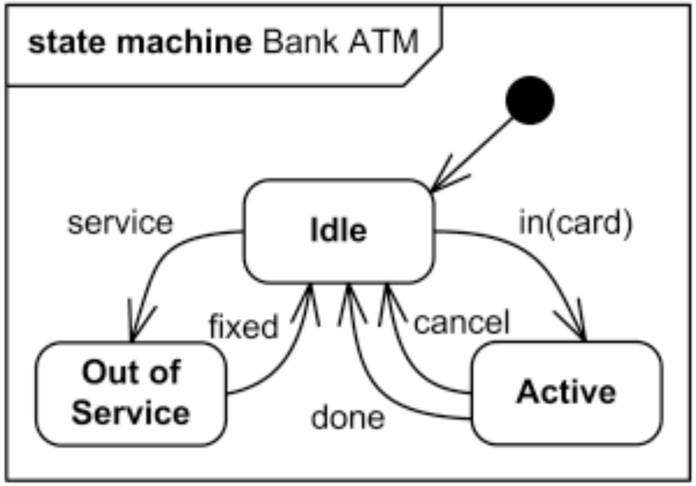
State machines in Java

- Represent the domain

```
class StateMachine {
  int current;
  // State x State -> Event[]
  String[][][] transitions;

  void step(String event) {
    for (int i = 0; i < transitions[current].length; i++)
      for (String expected : transitions[current][i])
        if (expected.equals(event)) {
          current = i;
          return;
        }
  }
}
```

State machines in Java

- Represent the domain application



```
int idle = 0;
int oos = 1;
int active = 2;

String[][][] transitions = new String[3][3];
transitions[idle][oos] = new String[] {“service”}
transitions[idle][active] = new String[] {“in-card”}
transitions[oos][idle] = new String[] {“fixed”}
transitions[active][idle] = new String[] {“cancel”, “done”}

StateMachine atm = new StateMachine(idle, transitions);
```

Why is this bad?

Evaluation

- The concepts of the state machine (states, events, transitions) are *encoded* and not directly represented:

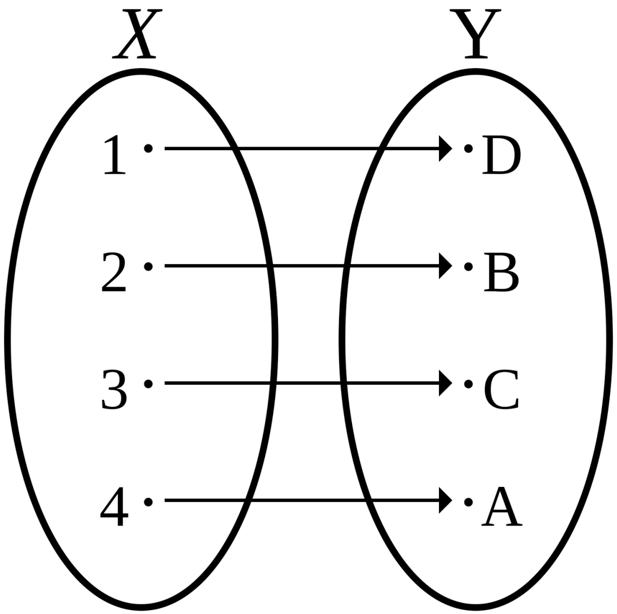
State machine	Java
state	integer
event	string
transition	lookup table

- This violates conceptual proximity (thinking)
- This also violates representational proximity (reading/writing)
  - State machines have nothing to do with array syntax, yet array syntax dominates the representation

Conceptual proximity

The concepts of a domain and their encoding should be proximal

- No big gap between domain concepts and encoding
- Domain knowledge can be directly translated into programs
- No need for adapting our mindset to think about the encoding rather than the domain concepts



Conceptual proximity

The concepts of a domain and their encoding should be proximal

- For example, in previous state machine, transitions are not proximal to their encoding within a lookup table:
  - How to figure out whether a *state* has an *outgoing transition*?

```
int state = …
for (int i = 0; i < transitions[state].length; i++)
  if (transitions[state][i] != null &&
      transitions[state][i].length > 0)
    return true;
return false;
```

  - Transitions are not directly represented
  - Complicated translation of our domain knowledge necessary

State machines in Java

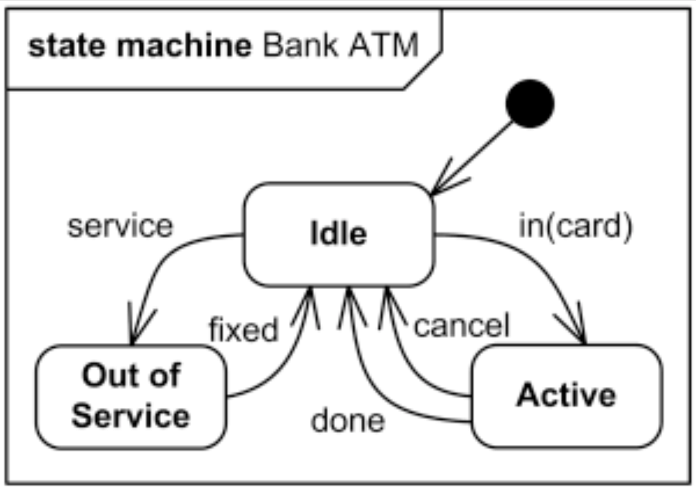
- Another try: Represent the domain

```
class StateMachine {
  State current;

  void step(String event) {
    current = current.step(event);
  }
}

class State {
  private int label;
  Map<String,State> transitions;

  State step(String event) {
    return transitions.get(event);
  }
}
```

State machines in Java

- Represent the domain application

```
State idle = new State(0);
State oos = new State(1);
State active = new State(2);

Map<String,State> idleTrans = new …
idleTrans.put(“service”, oos);
idelTrans.put(“in-card”, active);
idle.setTransitions(idleTrans);

Map<String,State> oosTrans = new …
oosTrans.put(“fixed”, idle);
oos.setTransitions(oosTrans);

Map<String,State> activeTrans = new …
activeTrans.put(“cancel”, idle);
activeTrans.put(“done”, idle);
active.setTransitions(activeTrans);

StateMachine atm = new StateMachine(idle);
```



Why is this bad?

Evaluation

- The concepts of the state machine (states, events, transitions) are encoded directly:

State machine	Java
state	object of class State
event	string
transition	maps event to state

- This conforms to conceptual proximity (thinking)
  - How to figure out whether a *state* has an *outgoing transition*?

```
State state = …
return !state.transitions.isEmpty()
```

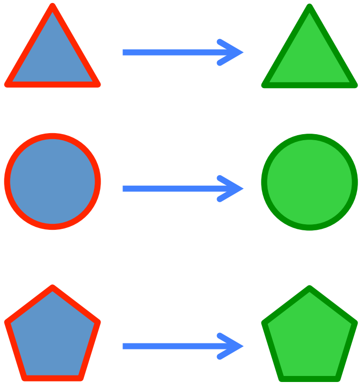
- But it violates representational proximity (reading/writing)

Representational proximity

The concepts of a domain and their representation should be proximal

- No big gap between domain representation and program representation
- No indirect translation of domain representation

- Domain knowledge can be directly represented in code (write)
- Code can be directly read as domain knowledge (read)



Representational proximity

The concepts of a domain and their representation should be proximal

- The first state machine violates representational proximity:
  - Array syntax dominates the representation of the state machine
  - A state and its transformations are separated

```
int idle = 0;
int oos = 1;
int active = 2;

String[][][] transitions = new String[3][3];
transitions[idle][oos] = new String[] {“service”}
transitions[idle][active] = new String[] {“in-card”}
transitions[oos][idle] = new String[] {“fixed”}
transitions[active][idle] = new String[] {“cancel”, “done”}

StateMachine atm = new StateMachine(idle, transitions);
```

Representational proximity

The concepts of a domain and their representation should be proximal

- The second state machine violates representational proximity:
  - Collection syntax for Map dominates the representation
  - A state and its transformations are separated

```
State idle = new State(0);
State oos = new State(1);
State active = new State(2);

Map<String,State> idleTrans = new …
idleTrans.put(“service”, oos);
idelTrans.put(“in-card”, active);
idle.setTransitions(idleTrans);


…


StateMachine atm = new StateMachine(idle);
```

Goal of Domain-specific languages (DSLs)

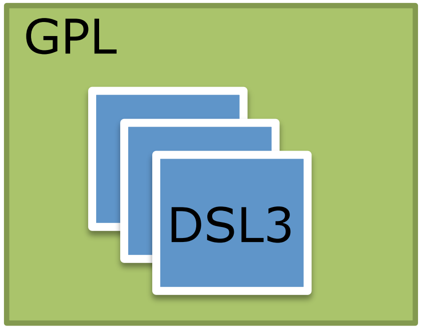
Narrow the gap between
a problem domain
and
its implementation

- The implementation should be close to the domains to improve
  - conceptual proximity (thinking)
  - representational proximity (reading/writing)

5.3 Styles of DSLs

- DSLs come in different flavors
- Internal/external to a general-purpose language
  - External DSLs come with their own interpreter/compiler
    - Standalone implementation
    - Independent of GPL
  - Hard to use multiple external DSLs together
    - only sequential composition

  - Internal DSLs are implemented as part of a GPL
  - Applying multiple internal DSLs corresponds to using different parts of a GPL
    - deep integration of DSLs possible



- We focus on internal DSLs

Internal DSL by pure embedding

- The state-machine DSL from before is an internal DSL
- Implemented as a *library* in the GPL
- This form of implementation is called *pure embedding*

- In fact, many DSLs are implemented as libraries or APIs
  - SQL: API in `java.sql`
  - XML: JDOM encoding in `org.jdom2`
  - regular expressions: library `java.util.Regex`
  - ...

- Conversely, many libraries represent DSLs
  - `java.net.HttpURLConnection` implements HTTP DSL
  - `java.io.*` implements File I/O DSL
  - ...

Pure embedding

- Implement DSLs as *libraries* in the GPL
  - Pro: No special language support needed
  - Cons: Bound to syntax, static analysis, and IDE support of GPL

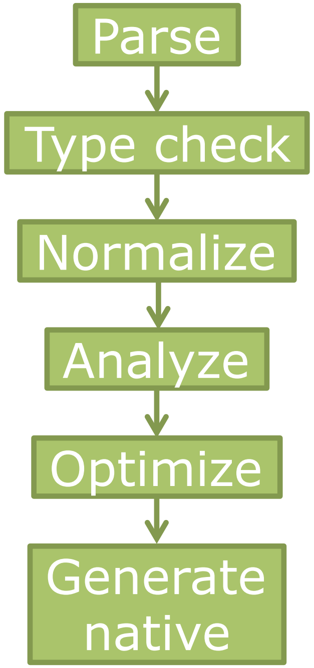
  - Example: only Java compiler needed, but Java syntax dominates DSL

```
State idle = new State(0);
State oos = new State(1);
State active = new State(2);

Map<String,State> idleTrans = new …
idleTrans.put(“service”, oos);
idelTrans.put(“in-card”, active);
idle.setTransitions(idleTrans);

Map<String,State> oosTrans = new …
oosTrans.put(“fixed”, idle);
oos.setTransitions(oosTrans);

Map<String,State> activeTrans = new …
activeTrans.put(“cancel”, idle);
```

Besides pure embedding

- Alternatives:
  - Compiler extension
  - Preprocessor
- Free to change the language
  - syntax
  - static analysis
  - semantics (to some degree)

- But: hard to develop, maintain, use, and compose
  - Require specific infrastructure
  - Developers cannot use standard compiler
    - need build scripts
  - Developers cannot use standard IDE



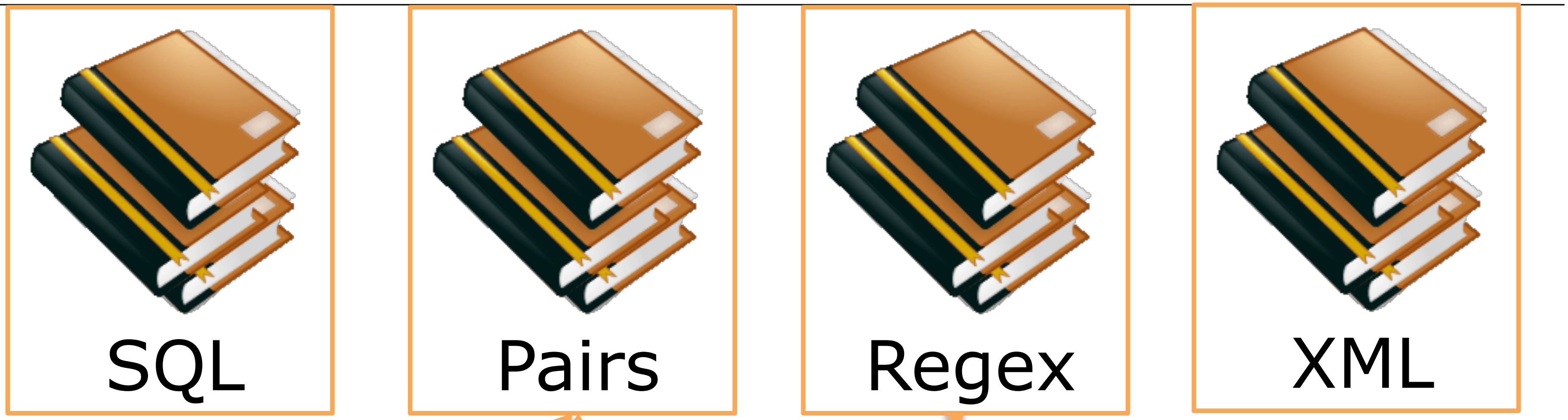
5.4 SugarJ

- We want the advantages of pure embedding
- And the freedom of compiler extensions

  - No external tools or build scripts
  - Easy to use
  - Customizable syntax
  - Customizable static analysis
  - Customizable IDE support
  - Composable

Libraries

Languages in Libraries

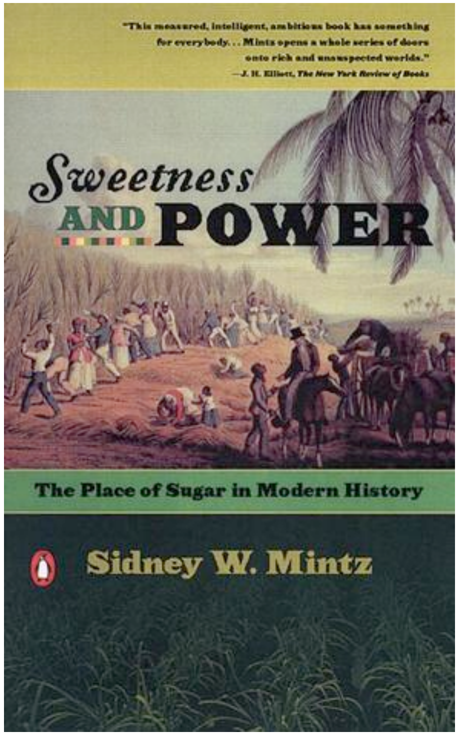


```
import Pa
import Re

public cl
  private
    ("/Users/seba", "/Users/seba".matches(/^\/[a-zA-Z\/]*$/));
}
```

```
import Pairs;

public class Test {
  private (String, Integer) p = ("12", 34);
}
```

Data serialization with XML

Task: serialize information about books using XML


"This measured, intelligent, ambitious book has something for everybody. . . Mintz opens a whole series of doors onto rich and unsuspected worlds."
—J. H. Elliott, The New York Review of Books
Sweetness AND POWER
The Place of Sugar in Modern History
Sidney W. Mintz

serialize

```
<book title="Sweetness and Power">
  <author name="Sidney W. Mintz" />
  <editions>
    <edition year="1985"
             publisher="Viking Press" />
    <edition year="1986"
             publisher="Penguin Books" />
  </editions>
</book>
```

Example: XML serialization

In Java using SAX

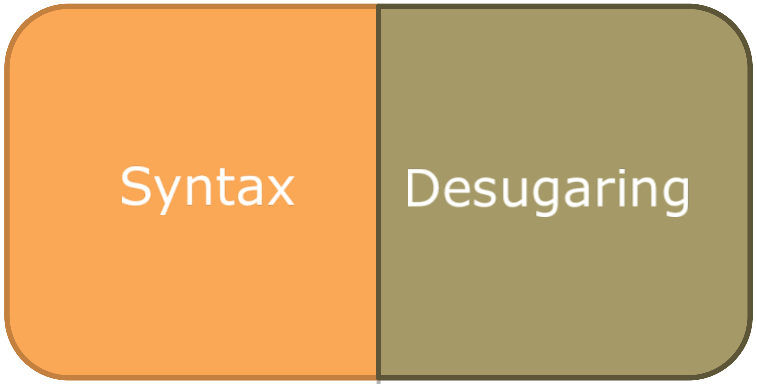
- No representational proximity

```
public void appendBook(ContentHandler ch) {
  String title = "Sweetness and Power";
  ch.startDocument();
  AttributesImpl bookAttrs = new AttributesImpl();
  bookAttrs.addAttribute("", "title", "title", "CDATA", title);
  ch.startElement("", "book", "book", bookAttrs);
  AttributesImpl authorAttrs = new AttributesImpl();
  authorAttrs.addAttribute("", "name", "name", "CDATA", "Sidney W. Mintz”);
  ch.startElement("", "author", "author", authorAttrs);
  ch.endElement("", "author", "author");
  ch.startElement("", "editions", "editions", new AttributesImpl());
  AttributesImpl edition1Attrs = new AttributesImpl();
  edition1Attrs.addAttribute("", "year", "year", "CDATA", "1985");
  edition1Attrs.addAttribute("", "publisher", "publisher", "CDATA", "Viking");
  ch.startElement("", "edition", "edition", edition1Attrs);
  ch.endElement("", "edition", "edition");
  ch.endElement("", "editions", "editions");
  ch.endElement("", "book", "book");
  ch.endDocument();
}
```

XML in SugarJ

```
import XML;

public void appendBook(ContentHandler ch) {
  String title = "Sweetness and Power";

  ch.<book title="{title}">
       <author name="Sidney W. Mintz" />
       <editions>
         <edition year="1985" publisher="Viking Press" />
         <edition year="1986" publisher="Penguin Books" />
       </editions>
     </book>;
}
```

Sugar libraries



```
ch.<book title="Sweetness and P
      <author name="Sidney W. Mi
      <editions>
        <edition year="1985" pub
        <edition year="1986" pub
      </editions>
    </book>
```

```
ch.startDocument();
AttributesImpl bookAttrs = new AttributesI
bookAttrs.addAttribute("", "title", "title
ch.startElement("", "book", "book", bookAt
AttributesImpl authorAttrs = new Attribute
authorAttrs.addAttribute("", "name", "name
ch.startElement("", "author", "author", au
ch.endElement("", "author", "author");
ch.startElement("", "editions", "editions"
AttributesImpl edition1Attrs = new Attribu
edition1Attrs.addAttribute("", "year", "ye
edition1Attrs.addAttribute("", "publisher"
public sugar Pairs {

  context-free syntax
    "(" JavaExpr "," JavaExpr ")" -> JavaExpr

  rules
    pair-desu
      |[ (~e1

  desugarings
    pair-desugaring
}
```

```
import Pairs;

public class Test {
  private (String, Integer) p = ("12", 34);
}
```

```
private (String, Integer) p = ("12", 34);
```

Desugar

```
private Pair<String, Integer> p = new Pair("12", 34);
```

State machines in SugarJ

- Another try: Represent the domain

Syntactic representation

```
sugar SMSugar {
  context-free syntax
    …

  rules
    …

  desugarings
    …
}
```

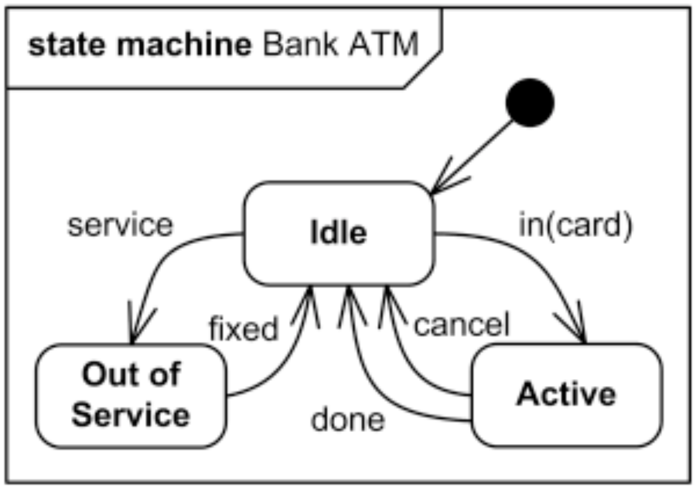
Semantic encoding

```
class StateMachine {
  State current;

  void step(String event) {
    current = current.step(event);
  }
}

class State {
  private int label;
  Map<String,State> transitions;

  State step(String event) {
    return transitions.get(event);
  }
}
```

State machines in Java

- Represent the domain application

```
import SMSugar;

statemachine atm {
  idle {
    service -> oos
    in-card -> active
  }

  oos {
    fixed -> idle
  }

  active {
    cancle -> idle
    done -> idle
  }
}
```



Language composition

We want DSLs for all problem domains

- the domain an application targets (e.g., banking or telephone relaying)
- all domains needed in the realization of the application (e.g., SQL)

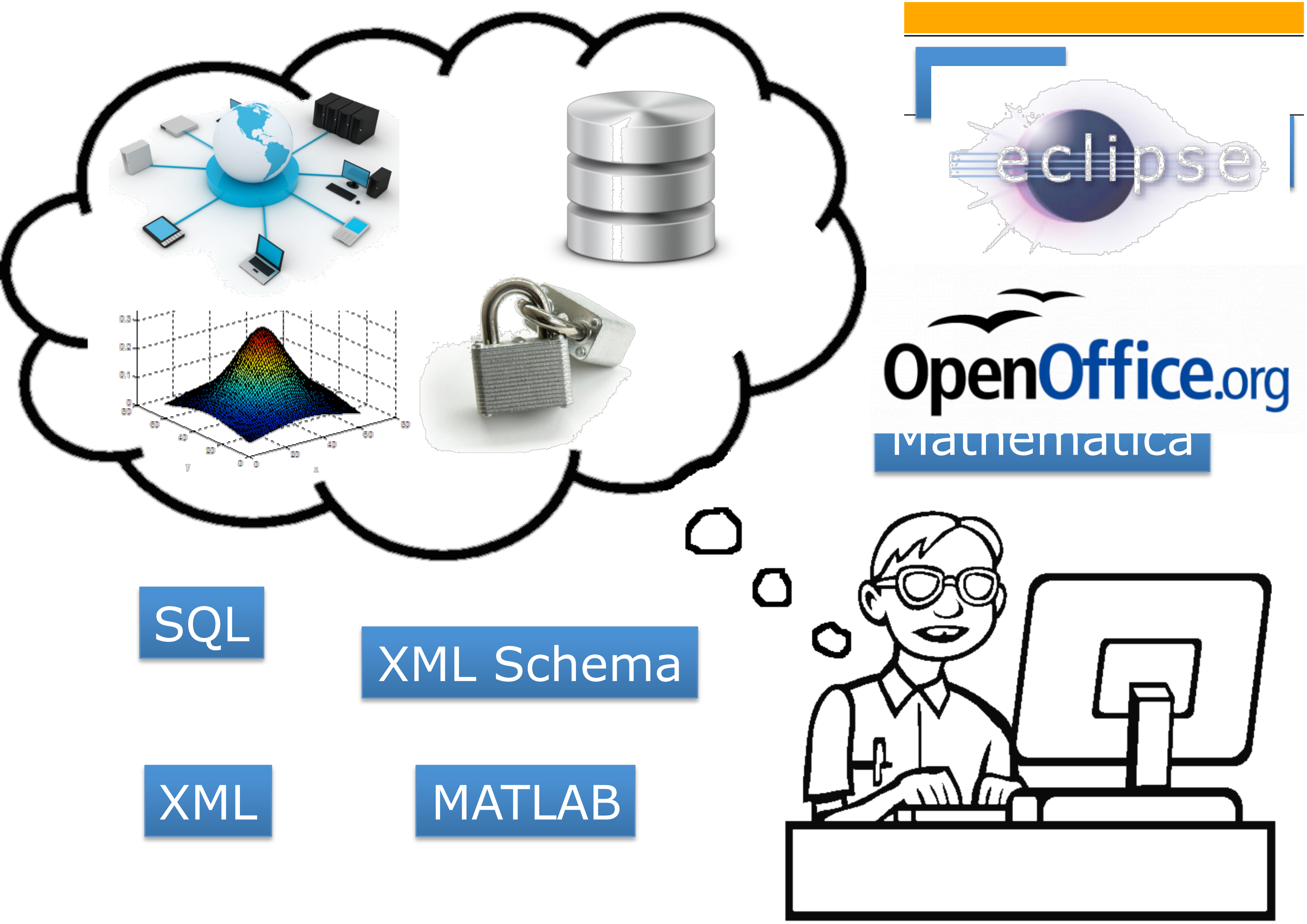
- Many domains are involved in realistic software projects
- Need support for composing DSLs

SQL

Pairs

Regex

XML



Mathematica

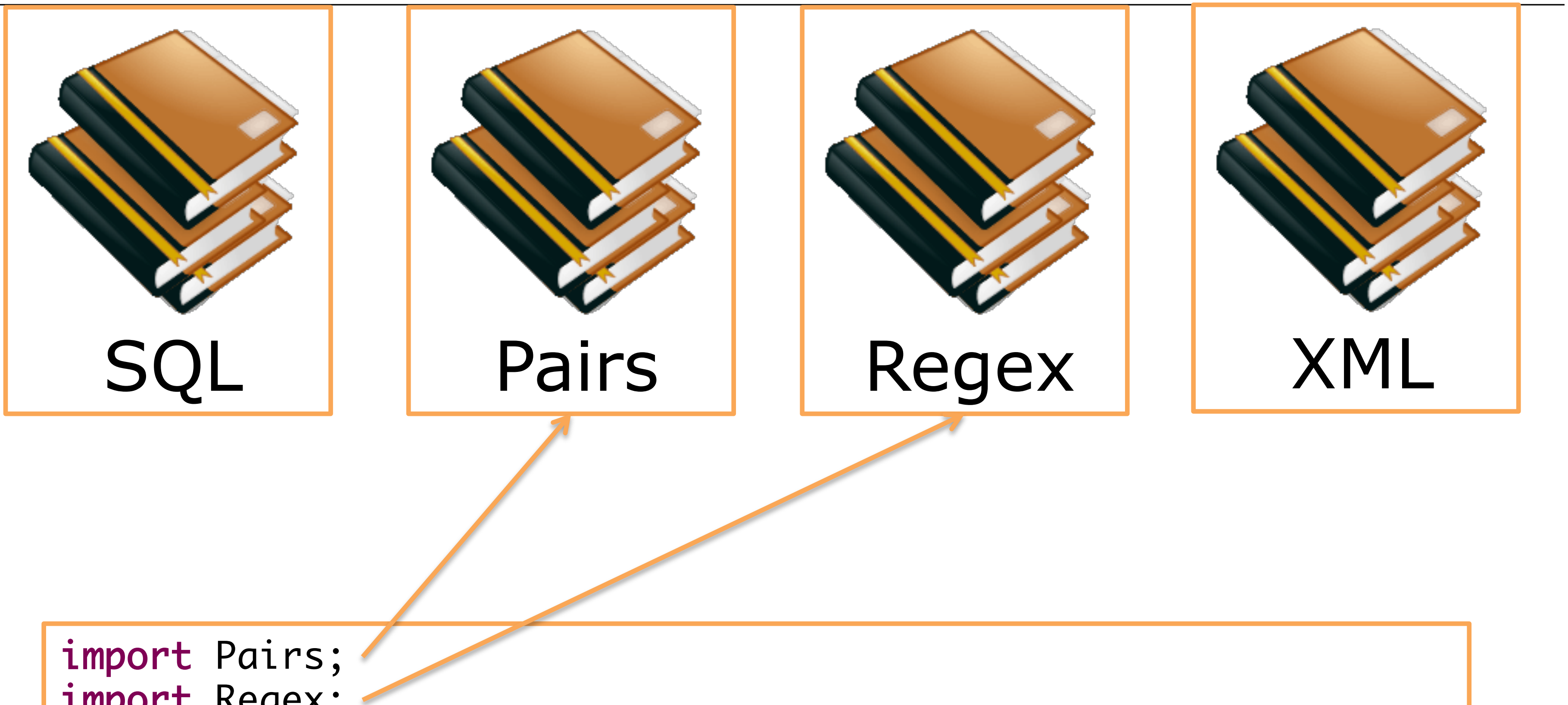
SQL

XML Schema

XML

MATLAB

Languages in Libraries



SQL

Pairs

Regex

XML

```
import Pairs;
import Regex;

public class Test {
  private (String, Boolean) homeDir =
    ("/Users/seba", "/Users/seba".matches(/^\/[a-zA-Z\/]*$/));
}
```

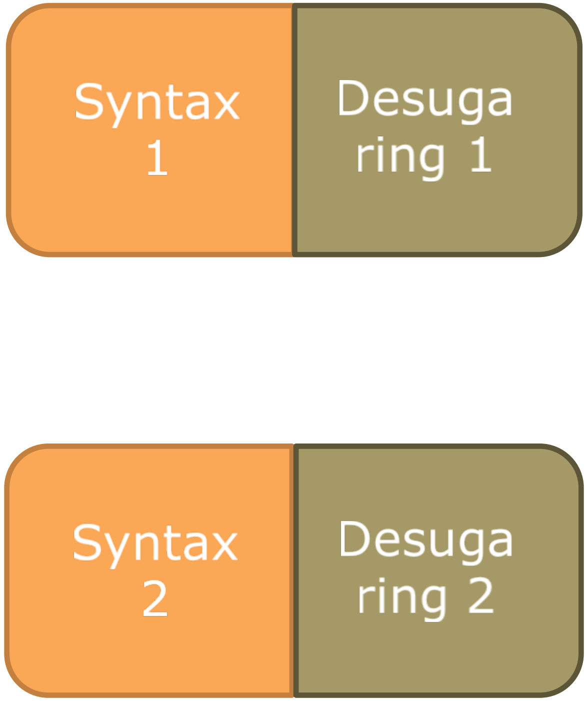
Language composition in SugarJ

SDF

- scannerless parsing
- generalized: full CFG
- grammar composition

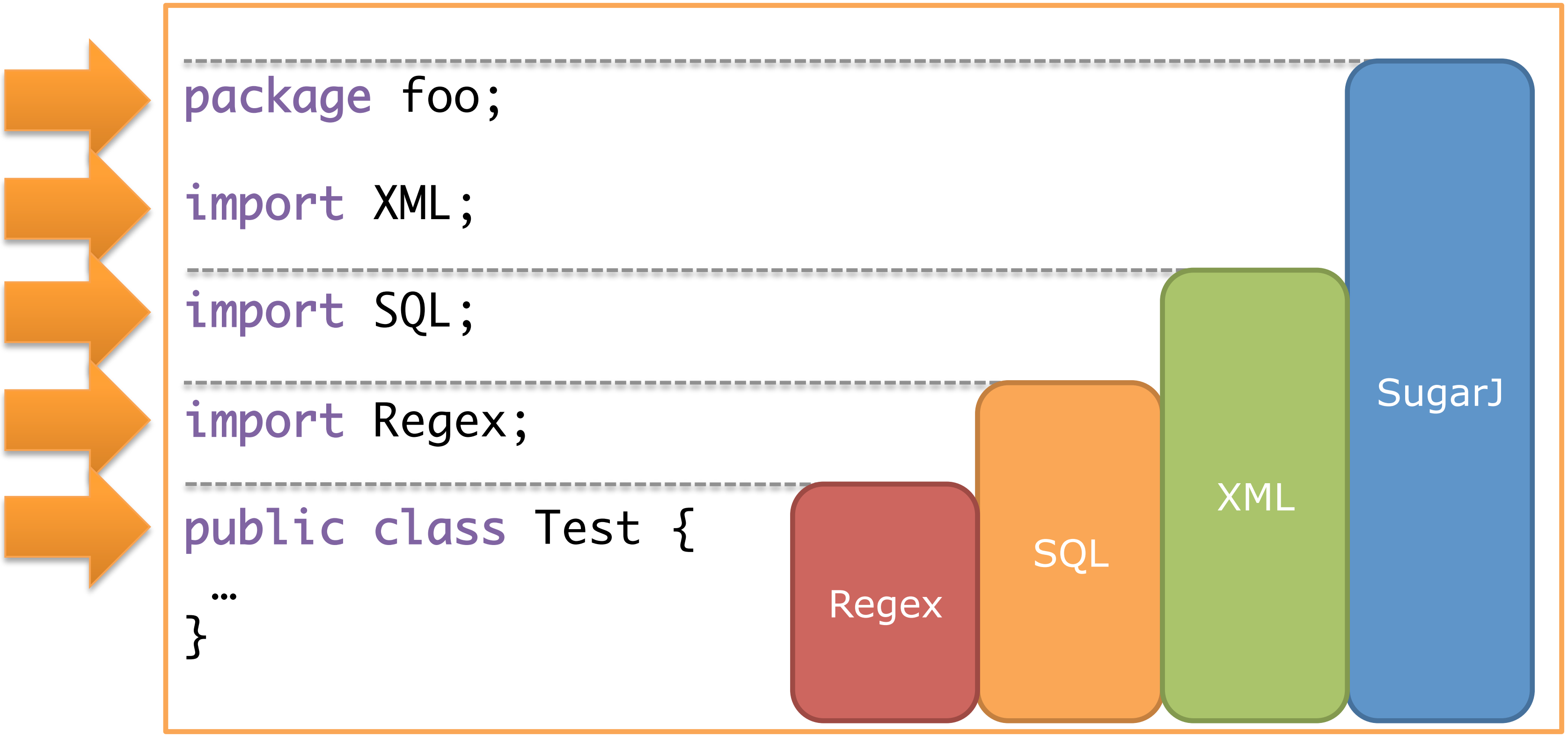
Stratego

- term rewriting
- higher-order rules
- rule composition



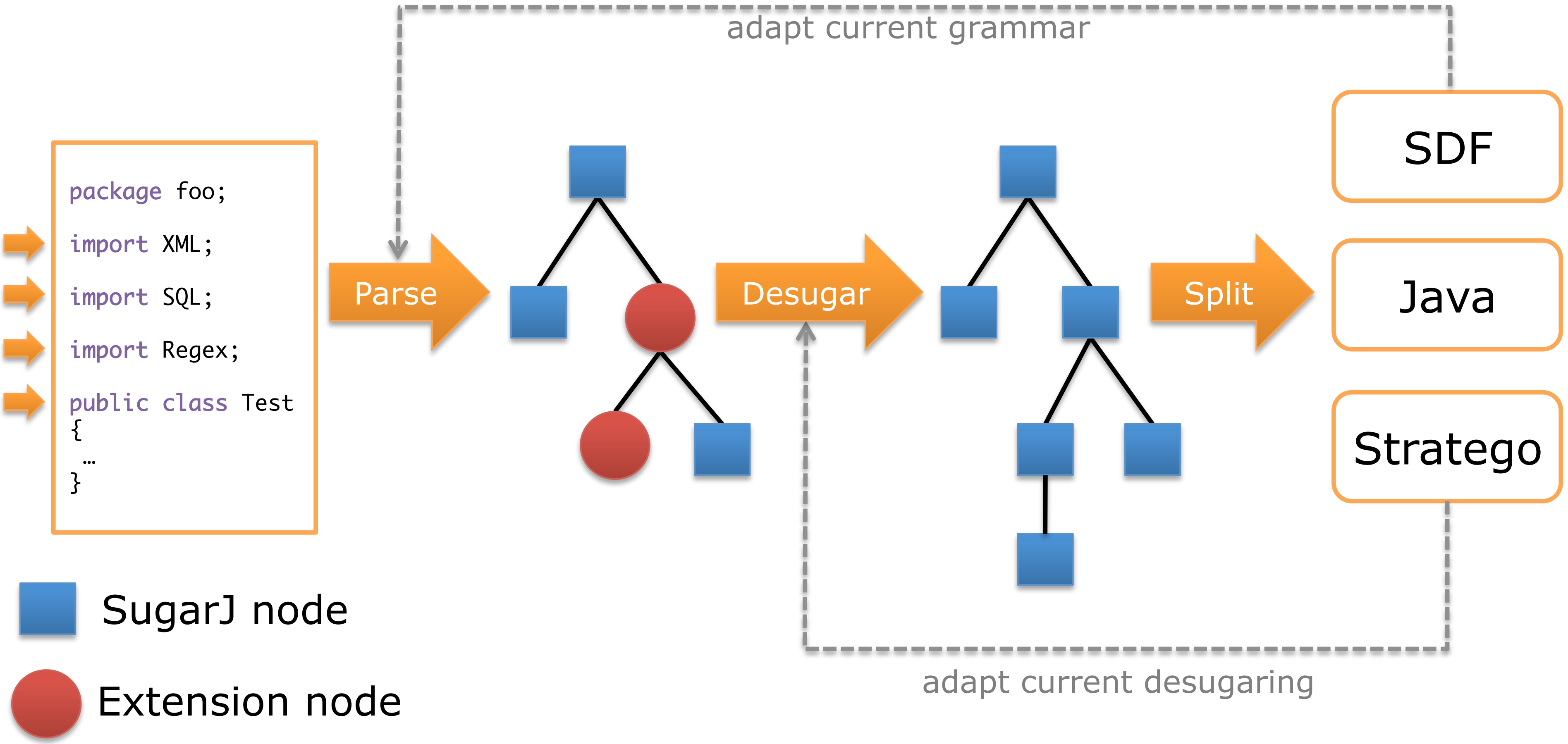
Sugar library composition

incremental parsing and grammar adaption



```
package foo;

import XML;
import SQL;
import Regex;
public class Test {
  …
}
```

SugarJ internals

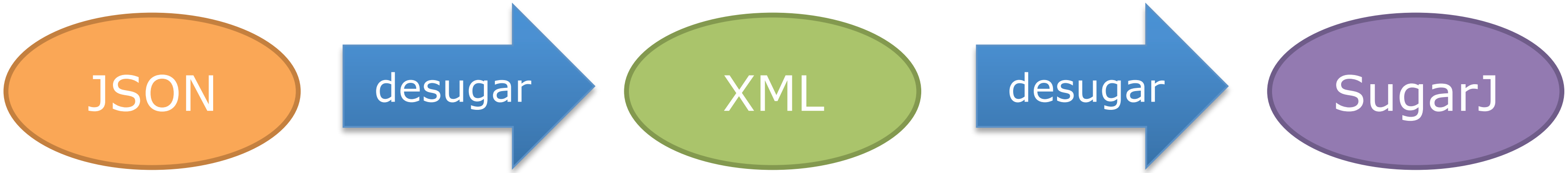


```
package foo;

import XML;

import SQL;

import Regex;

public class Test
{
  …
}
```

libraries are self-applicable

Self-applicability

DSLs can build on other DSLs

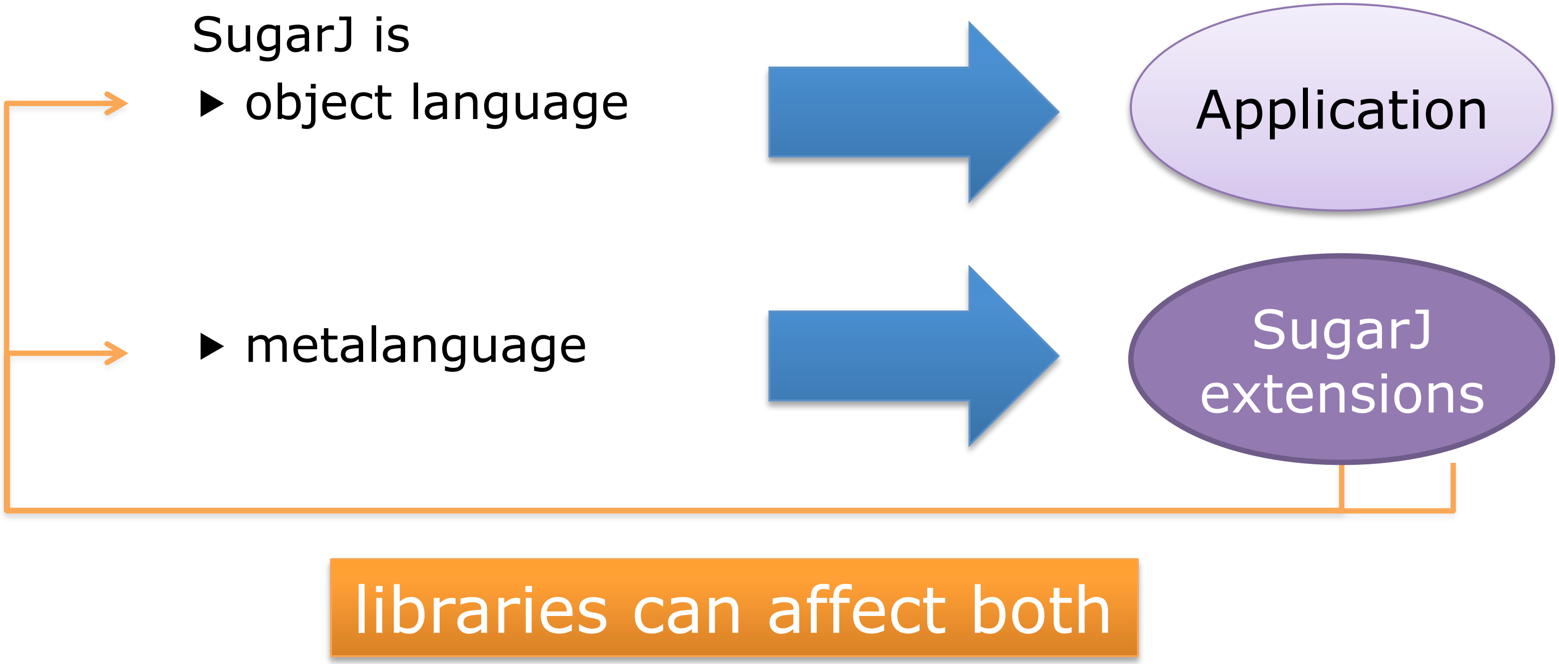


```
{
  book : {
    title : "Sweetness"
    author : {
      name : "Sidney"
    }
    editions : { … }
  }
}
```

```
<book
  title="Sweetness">
  <author
    name="Sidney" />
  <editions> …
  </editions>
</book>
```

```
ch.startDocument();
AttributesImpl bookAttrs = new Attributes
bookAttrs.addAttribute("", "title", "tit
ch.startElement("", "book", "book", bookA
AttributesImpl authorAttrs = new Attribu
authorAttrs.addAttribute("", "name", "na
ch.startElement("", "author", "author", 
ch.endElement("", "author", "author");
ch.startElement("", "editions", "edition
AttributesImpl edition1Attrs = new Attri
edition1Attrs.addAttribute("", "year", "
edition1Attrs.addAttribute("", "publishe
ch.startElement("", "edition", "edition"
ch.endElement("", "edition", "edition");
ch.endElement("", "editions", "editions"
ch.endElement("", "book", "book");
ch.endDocument();
```

Metalevels and SugarJ



SugarJ is

- object language
- metalanguage

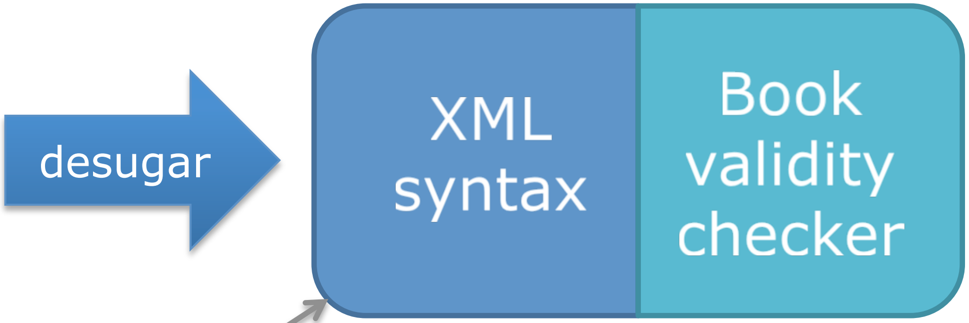
Application

SugarJ extensions

libraries can affect both

XML Schema

```
<xsd:schema targetNamespace="lib">
  <xsd:element name="book" type="B
  <xsd:complexType name="Book">
    <xsd:choice maxOccurs="unbound
      <xsd:element name="author" t
      <xsd:element name="editions"
    </xsd:choice>
    <xsd:attribute name="title" ty
  </xsd:complexType>
</xsd:schema>
```



!

```
import BookSchema;

ch.<book title="{title}">
     <author name="Sidney W. Mintz" />
     <editions>
       <edition year="1985" publisher="Viking Press" />
       <edit    year="1986" publisher="Penguin Books" />
     </editions>
   </book>;
```

!

5.5 Summary: DSLs

**Narrow the gap between
a problem domain
and
its implementation**

- The implementation should be close to the domains to improve
  - conceptual proximity (thinking)
  - representational proximity (reading/writing)
  - language composition to support multiple domains

Further reading

- Pure embedding of DSLs
  - Hudak: Modular domain specific languages and tools
  - We discuss this paper next week on Wednesday

- SugarJ: Library-based Syntactic Language Extensibility
  - Paper and further documentation available online http://sugarj.org
  - Try it out: Eclipse update from http://update.sugarj.org

- Interested in a thesis topic?
  - Come talk to us!