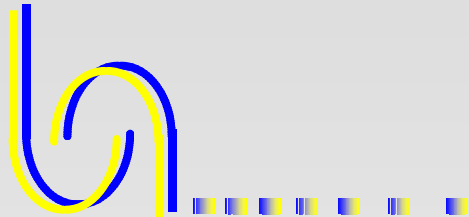


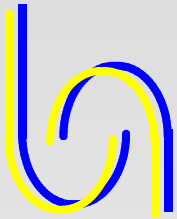
# OntoJava

Applying Mainstream Technology to  
the Semantic Web



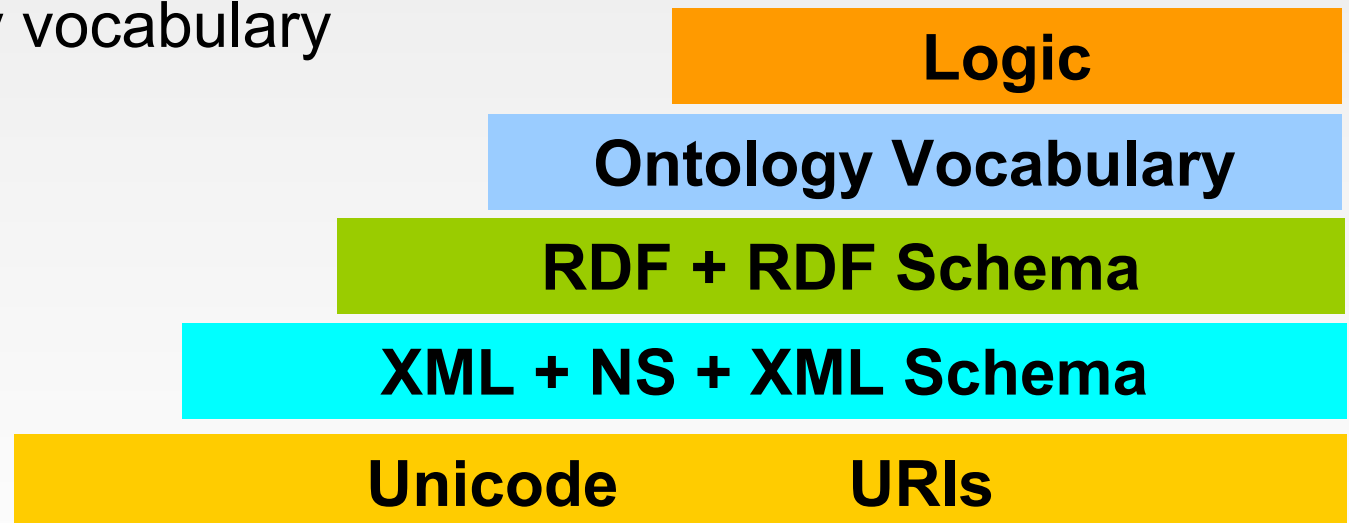
# Semantic Web Vision

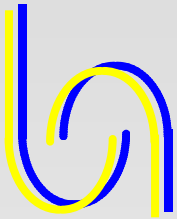
- Web Services are the current state of the art technology in B2B systems
  - WSDL, UDDI allow some dynamic discovery
  - But, the approaches are based on standardized APIs such as RosettaNet, ebXML
- Vision
  - Use (cross-linked) Ontologies to describe a domain
  - Agents only need to be aware of the Ontology in order to interact with each another



# Semantic Web ML Stack

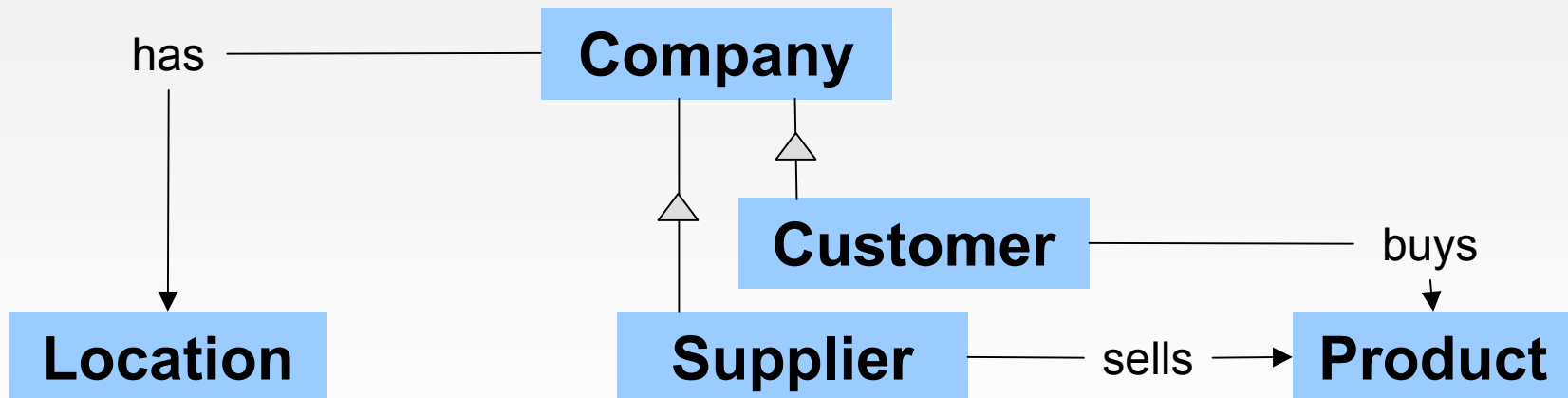
- Sharing and re-use are the key
  - Data available in RDF (Wordnet, Open Directory)
  - Initial upper level Ontologies (Open Cyc) and core building blocks (Addresses, etc) are available
  - Rules and constraints can be expressed using the ontology vocabulary

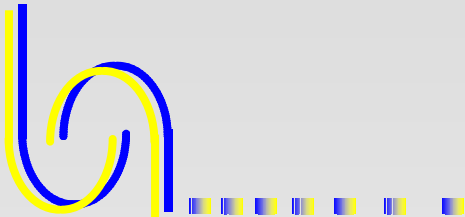




# Example: Choosing a Supplier

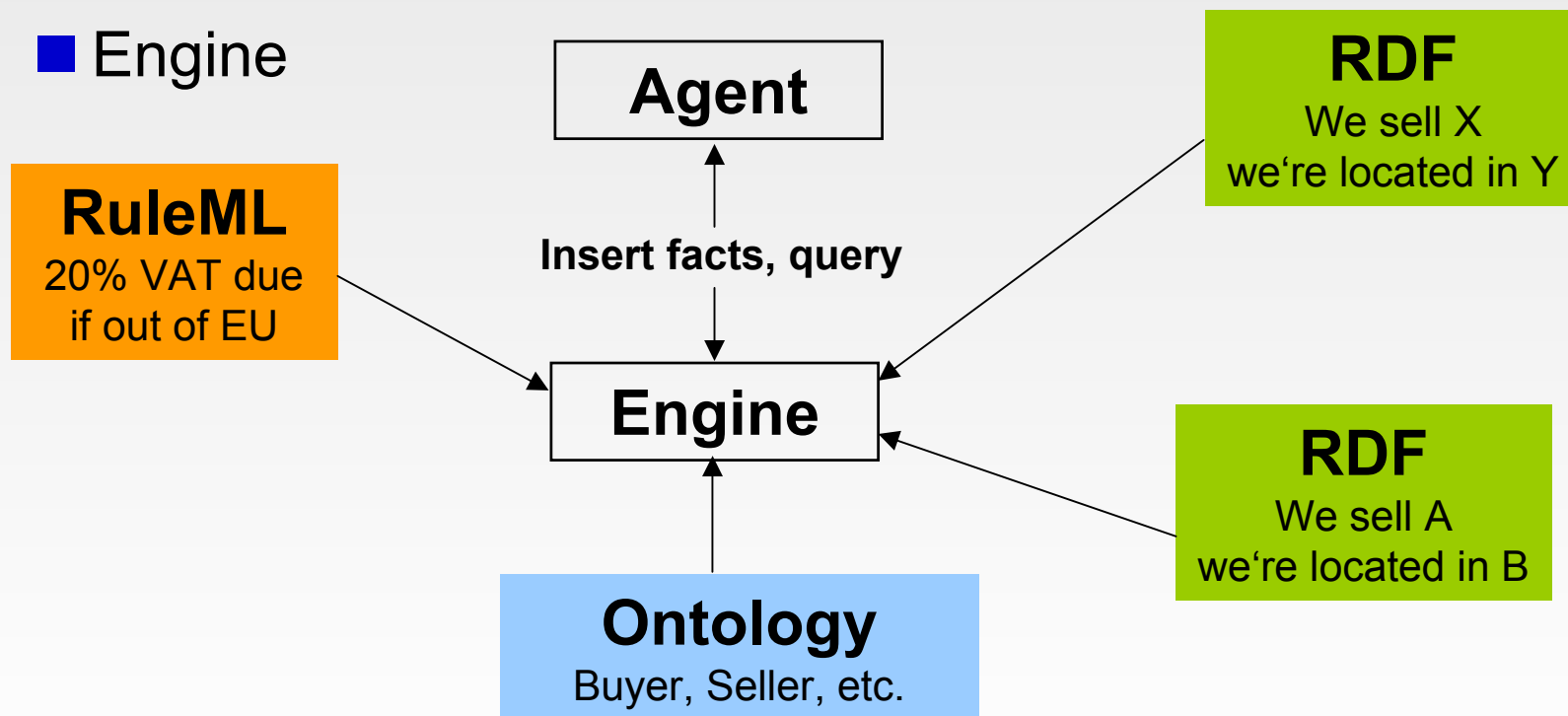
- RDF data: global geography database
- Ontology: agents use generic B2B ontology
- Logic: rules to select a supplier:
  - Compute taxes, shipping costs
  - Determine the reliability of a supplier

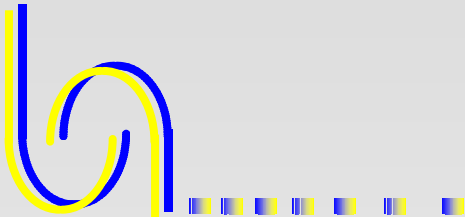




# Architecture

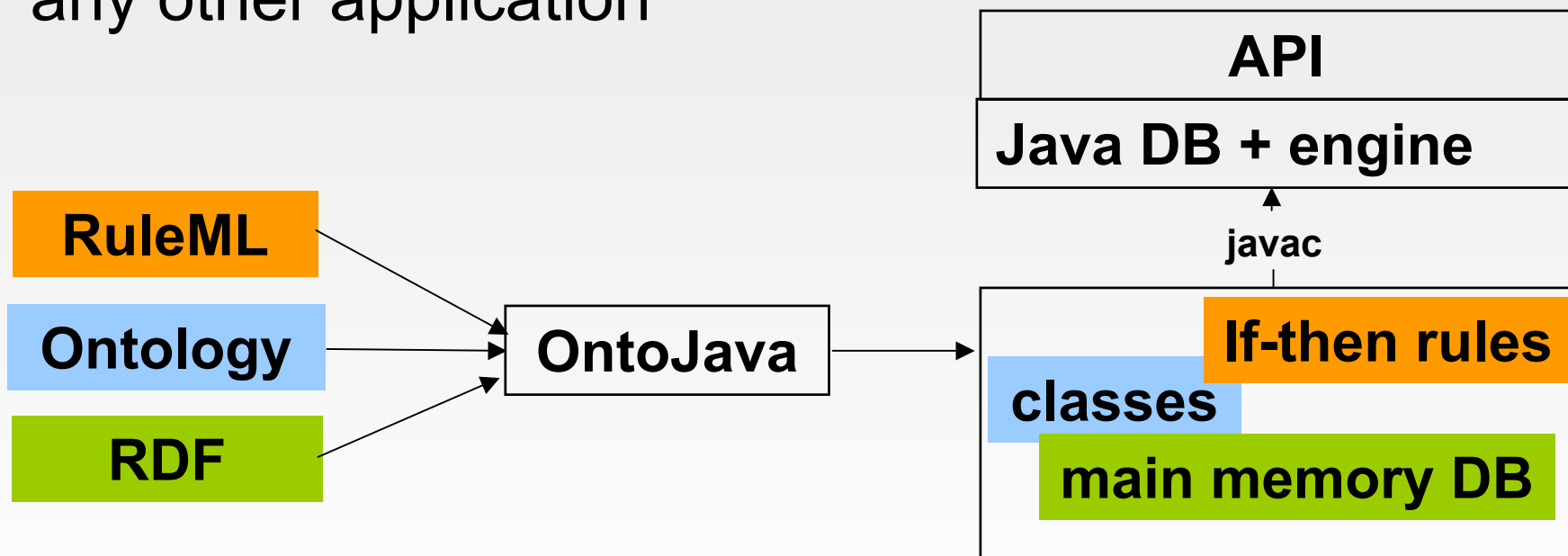
- The Semantic Web Language independent of platform or software used
  - Need Query language
  - Engine

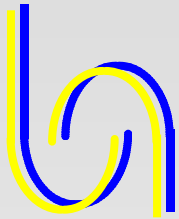




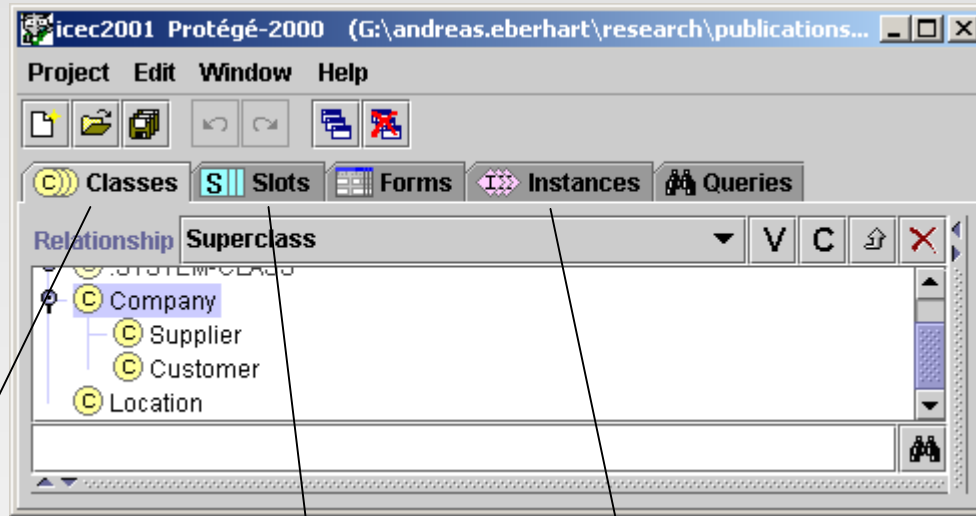
# Java Mapping

- OntoJava compiles RDFS, RDF, and RuleML into a main memory DB and inference engine
- Data can be inserted or queried via the API by any other application





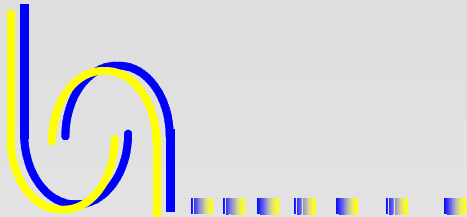
# Editing Ontology and Facts with Protégé



```
<rdfs:Class rdf:about="&pre;Supplier">  
  <rdfs:subClassOf rdf:resource="&pre;Company"/>  
</rdfs:Class>
```

```
<rdf:Property rdf:about="&pre;has"  
  <rdfs:domain rdf:resource="&pre;Company"/>  
  <rdfs:range rdf:resource="&pre;Location"/>  
</rdf:Property>
```

```
<geo:Location rdf:about="&pre;ITALY"/>
```

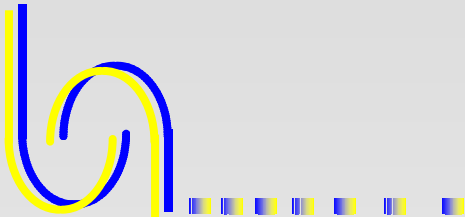


# Mapping the Class Hierarchy

- The ontology's classes and their hierarchy are mapped to Java classes

```
<rdfs:Class rdf:about="&pre;Supplier">  
  <rdfs:subClassOf rdf:resource="&pre;Company"/>  
</rdfs:Class>
```

```
public class Supplier extends Company
```

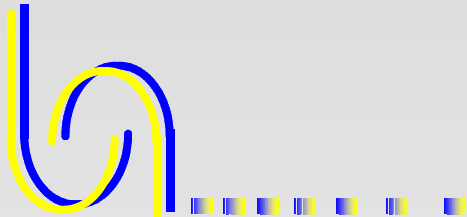


# Mapping the Slots

- The slots are converted to instance variables
  - Primitive datatypes are straight forward
  - Relations are mapped with a HashSet and appropriate accessor methods
  - Compiler enforces correct datatypes
  - Cardinality constraints can be mapped in an ADT

```
<rdf:Property rdf:about="&pre;has"  
  <rdfs:domain  
    rdf:resource="&pre;Company"/>  
  <rdfs:range  
    rdf:resource="&pre;Location"/>  
</rdf:Property>
```

```
private HashSet has = new HashSet();  
public void puthas(Location s) {  
    has.add(s);  
}  
public boolean gethas(Location s) {  
    return has.contains(s);  
}
```



# Mapping Facts

## ■ Facts such as

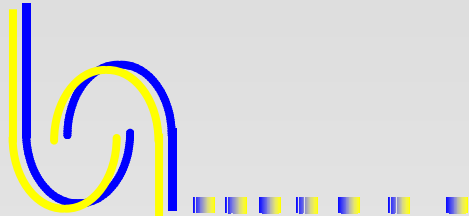
```
<geo:Location rdf:about="&pre;ITALY"/>
```

are mapped to global variables that can be referenced from rules

```
static Location ITALY = new Location();
```

## ■ A global collection of facts is also maintained

```
static HashSet object = new HashSet();  
object.add(ITALY);
```



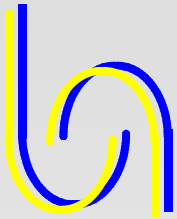
# Mapping the Rules

## ■ Algorithm

- One method per rule
- Check combinations of free variables
- Check the condition

## ■ Sample rule:

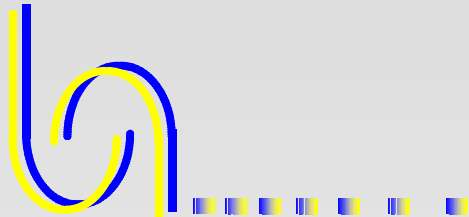
- If you're a customer in Italy, then the VAT tax applies



# A translated rule

```
public static void rule6(Customer customer)
{
    Tax VAT = DB.VAT;
    Location ITALY = DB.ITALY;

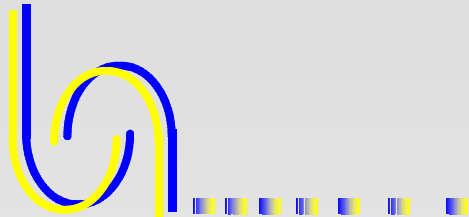
    boolean customerflag = (customer == null);
    Iterator customerIterator;
    if (customer == null)
        customerIterator = DB.student.iterator();
    else
        customerIterator = new PseudoIterator(customer);
    while (customerIterator.hasNext())
    {
        customer = (Customer)customerIterator.next();
        if (customer.gethas(ITALY))
        {
            Trace.print("rule 6 fired");
            customer.puttaxApplies(VAT);
        }
    }
}
```



# Executing the Rules

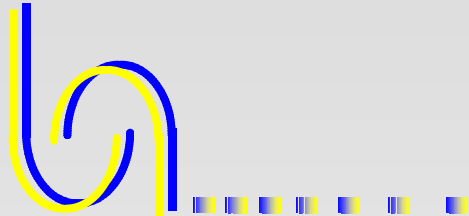
- The rules are executed in an incremental way
  - If predicate  $p$  on object  $O$  is changed, run all rules that have  $p$  on the right side
  - Firing rules can assert new facts and trigger new rules – run only if something changes!

```
private HashSet has = new HashSet();
public void puthas(Location s) {
    if (!(has.add(s)))
        return;
    Rule.rule4(this);
    Rule.rule6(null, this);
    Rule.rule9(this, null);
}
public boolean gethas(Location s) {
    return has.contains(s);
}
```



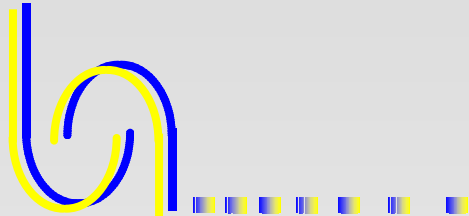
# Limitations of the Approach

- Only forward reasoning supported
  - Outcome might vary depending on the order in which the rules are executed
- Multiple inheritance currently not supported
  - Use interfaces or C#
- Currently only binary predicates possible
- No ad-hoc querying possible
  - Create new rule and recompile



# Why Java?

- Obviously has certain shortcomings
- But many advantages
  - Can use Java tools (IDEs, Debugger, javadoc)
  - Easy integration into J2EE applications
  - Simple ontology browser using reflection API
  - Ontologies can be compiled into a java package and distributed in jar files
  - Inferencing is customizable (e.g. Using probabilities)



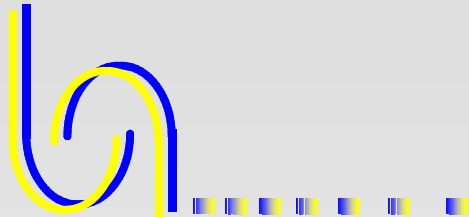
# Possible RuleML Extensions

## ■ Embed java commands in the rules

```
<head>  
    <% Util.sendEmail(R, B) %>  
</head>
```

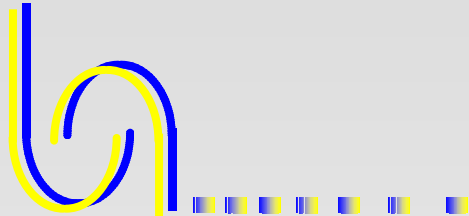
## ■ Access properties of the variables

```
<atom>  
    <_opr>&lt;</_opr>  
    <var>A.age</var>  
    <var>B.age</var>  
</atom>
```



# Future Work

- Object Relational Mapping
  - Borrow some ideas from EJB, container managed persistence
  - Determine rule and constraint patterns that can be mapped to SQL constraints, triggers, views



# Summary

- **OntoJava takes a very simple approach**
- **Many limitations**
- **But**
  - **Easy to use**
  - **Easy to integrate**