Introduction to Rules Engines

Brian McCallister
brianm@chariotsolutions.com
http://www.chariotsolutions.com/

Objectives

- Understanding of forward-chaining rules engines
- Introduction to the Drools rules engine
- Insights into where rules engines are a good solution

Rules Engine Basics

- Forward Chaining
  - Rules match facts and execute
  - Rete and company
- Reverse Chaining
  - Answer questions based on facts
  - Prolog style

Examples are Good

Ecommerce Personalized Specials
(e.g. Product Selection in Front of Amazon)

A (Small) Set of Rules

```
<rule-set name="basics"
          xmlns="http://drools.org/rules"
          xmlns:java="http://drools.org/semantics/java"
          xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
          xs:schemaLocation="http://drools.org/rules rules.xsd
                             http://drools.org/semantics/java java.xsd">

    <import>org.skife.drools.*</import>

    <application-data identifier="specials">Specials</application-data>

    <rule name="Favorite Colors">
        <parameter identifier="person">
            <class>Person</class>
        </parameter>
        <parameter identifier="item">
            <class>Item</class>
        </parameter>
        <java:condition>
            person.getFavoriteColor().equals(item.getColor())
        </java:condition>
        <java:consequence>
            specials.increment(item);
        </java:consequence>
    </rule>
</rule-set>
```

Just a Rule...

```
<rule name="Favorite Colors">
    <parameter identifier="person">
        <class>Person</class>
    </parameter>

    <parameter identifier="item">
        <class>Item</class>
    </parameter>


    <java:condition>
     person.getFavoriteColor().equals(item.getColor())
    </java:condition>

    <java:consequence>
        specials.increment(item);
    </java:consequence>
</rule>
```

Parameters (Match Facts)

```
<rule name="Favorite Colors">
    <parameter identifier="person">
        <class>Person</class>
    </parameter>

    <parameter identifier="item">
        <class>Item</class>
    </parameter>


    <java:condition>
     person.getFavoriteColor().equals(item.getColor())
    </java:condition>

    <java:consequence>
        specials.increment(item);
    </java:consequence>
</rule>
```

Conditions (If ...)

```
<rule name="Favorite Colors">
    <parameter identifier="person">
        <class>Person</class>
    </parameter>

    <parameter identifier="item">
        <class>Item</class>
    </parameter>

    <java:condition>
     person.getFavoriteColor().equals(item.getColor())
    </java:condition>

    <java:consequence>
        specials.increment(item);
    </java:consequence>
</rule>
```

Head or LHS

```
<rule name="Favorite Colors">
    <parameter identifier="person">
        <class>Person</class>
    </parameter>

    <parameter identifier="item">
        <class>Item</class>
    </parameter>

    <java:condition>
     person.getFavoriteColor().equals(item.getColor())
    </java:condition>

    <java:consequence>
        specials.increment(item);
    </java:consequence>
</rule>
```

Head or LHS

Left Hand Side

- Parameters and Conditions
- Logically Anded Together
- Rules "Match"

Parameters

- Usually based on types
- Sometimes Type Annotation
- Product of all available facts applied to rule parameters and conditions

Conditions

- Given condition general evaluated only once, unless facts change
- Becomes graph traversal
  - Quite optimizable

Four Rules (LHS Only)

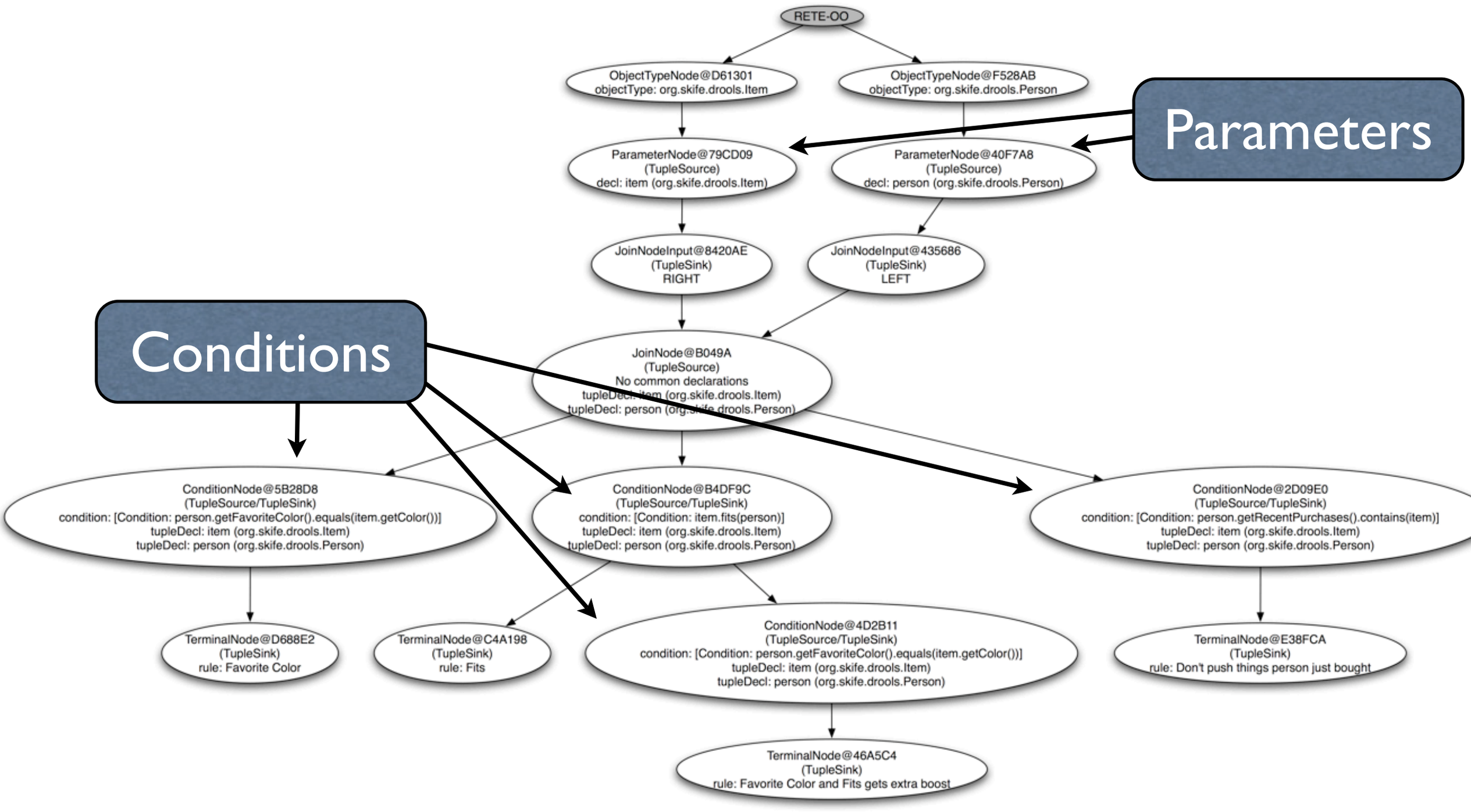
```
rule: Favorite Color
   param: Person => person
   param: Item => item
   condition: person.favoriteColor == item.color

rule: Fits
   param: Person => person
   param: Item => item
   condition: item.fits(person)

rule: Fits and Favorite Color Booster
   param: Person => person
   param: Item => item
   condition: item.fits(person)
   condition: person.favoriteColor == item.color

rule: Don't sell what they just bought
   param: Person => person
   param: Item => item
   condition: person.recentPurchases.contains(item)
```

Drools Network (Not RETE)



Right Hand Side

- The "Stuff" of the Rule
- Can manipulate facts
- What else?
  - Depends on the engine

Consequence (Does Stuff)

```
<rule name="Favorite Colors">
    <parameter identifier="person">
        <class>Person</class>
    </parameter>

    <parameter identifier="item">
        <class>Item</class>
    </parameter>

    <java:condition>
     person.getFavoriteColor().equals(item.getColor())
    </java:condition>

    <java:consequence>
        specials.increment(item);
    </java:consequence>
</rule>
```

Body or RHS

```
<rule name="Favorite Colors">
    <parameter identifier="person">
        <class>Person</class>
    </parameter>

    <parameter identifier="item">
        <class>Item</class>
    </parameter>

    <java:condition>
     person.getFavoriteColor().equals(item.getColor())
    </java:condition>

    <java:consequence>
        specials.increment(item);
    </java:consequence>
</rule>
```

Body or RHS

Using the Rules (Drools)

```
WorkingMemory wm = rule_base.newWorkingMemory();

Person brian = new Person("Brian", "blue");
wm.assertObject(brian);

Item blue_item = new BlueItem("Blue Item 1");
wm.assertObject(blue_item);

Item red_item = new RedItem("Red Item");
wm.assertObject(red_item);

Item another_blue = new BlueItem("Blue Item 2");
wm.assertObject(another_blue);

Specials specials = new Specials();
wm.setApplicationData("specials", specials);

wm.fireAllRules();

assertEquals(1, specials.scoreFor(blue_item));
assertEquals(1, specials.scoreFor(another_blue));
assertEquals(0, specials.scoreFor(red_item));
```

Knowledge Base

- Like a Class
- Contains/Reads/Parses Rules
- “Heavy”

Working Memory

- Instance of Knowledge Base
- Contains Facts
- "Lightweight"

Knowledge Base & Working Memory

```
URL rules = new URL("http://rules.local/basic.drl");

RuleBase rule_base = RuleBaseLoader.loadFromUrl(rules);

WorkingMemory wm = rule_base.newWorkingMemory();
```

Asserting Facts

```
Person brian = new Person("Brian", "blue");
wm.assertObject(brian);

Item blue_item = new BlueItem("Blue Item 1");
wm.assertObject(blue_item);

Item red_item = new RedItem("Red Item");
wm.assertObject(red_item);

Item another_blue = new BlueItem("Blue Item 2");
wm.assertObject(another_blue);

Specials specials = new Specials();
wm.setApplicationData("specials", specials);

wm.fireAllRules();
```

Firing the Rules

```
Person brian = new Person("Brian", "blue");
wm.assertObject(brian);

Item blue_item = new BlueItem("Blue Item 1");
wm.assertObject(blue_item);

Item red_item = new RedItem("Red Item");
wm.assertObject(red_item);

Item another_blue = new BlueItem("Blue Item 2");
wm.assertObject(another_blue);

Specials specials = new Specials();
wm.setApplicationData("specials", specials);

wm.fireAllRules();
```

They Work!

```
        Item another_blue = new BlueItem("Blue Item 2");
        wm.assertObject(another_blue);

        Specials specials = new Specials();
        wm.setApplicationData("specials", specials);

        wm.fireAllRules();

        assertEquals(1, specials.scoreFor(blue_item));
        assertEquals(1, specials.scoreFor(another_blue));
        assertEquals(0, specials.scoreFor(red_item));
```

Working Memories

- Facts + Facts == more Facts!
  - or less, or sometimes just different
- Execution of rules has handle on the working memory
- Alter, retract, modify facts

Modifying Knowledge

```
<rule name="10% Discount at end of month" no-loop="true">
    <parameter identifier="today">
        <class>Calendar</class>
    </parameter>
    <parameter identifier="item">
        <class>Item</class>
    </parameter>
    <java:condition>
       today.get(Calendar.DAY_OF_MONTH) > 20
    </java:condition>
    <java:consequence>
      item.setPrice(item.getPrice()
                          .multiply(new BigDecimal("0.90")));

      drools.modifyObject(item);
    </java:consequence>
</rule>
```

Modifying Knowledge

```
<rule name="10% Discount at end of month" no-loop="true">
    <parameter identifier="today">
        <class>Calendar</class>
    </parameter>
    <parameter identifier="item">
        <class>Item</class>
    </parameter>
    <java:condition>
       today.get(Calendar.DAY_OF_MONTH) > 20
    </java:condition>
    <java:consequence>
      item.setPrice(item.getPrice()
                        .multiply(new BigDecimal("0.90")));

      drools.modifyObject(item);
    </java:consequence>
</rule>
```

With Drools...

- Currently doesn't use actual RETE
  - Probably slightly slower algorithm for really large rule sets
- Built In "Languages"
  - Java
  - Jython
  - Groovy
  - BeanShell
- Strong DSL capabilities

Okay, So What?

- So, it's a bunch of fast ands...
- Classics:
  - Financial and Insurance
    - Highly regulated
      - Rules (from the gov't) change often
- In General:
  - Rules change often
  - Very complex behaviors
  - Correlations

Some Real Examples

No real code, though, I don't own that, unfortunately =(

Inventory Reduction

Ads the Smart Way

Inventory Reduction

- Online retailer wants to lower inventory
- Could be used to control supply chain shape as well

Facts

- Items to be made available as specials
- All known marketing data on the user
  - Demographic
  - Past purchases
  - etc
- Contents of Shopping Cart
- Trail (though this didn't work so well)

Rules

- First Try
  - Multiple regression analysis of mass trends
  - Good in theory, too complicated
- Second Try
  - Best Guesses
  - “Common Sense”

Remember These?

```
rule: Favorite Color
   param: Person => person
   param: Item => item
   condition: person.favoriteColor == item.color

rule: Fits
   param: Person => person
   param: Item => item
   condition: item.fits(person)

rule: Fits and Favorite Color Booster
   param: Person => person
   param: Item => item
   condition: item.fits(person)
   condition: person.favoriteColor == item.color

rule: Don't sell what they just bought
   param: Person => person
   param: Item => item
   condition: person.recentPurchases.contains(item)
```

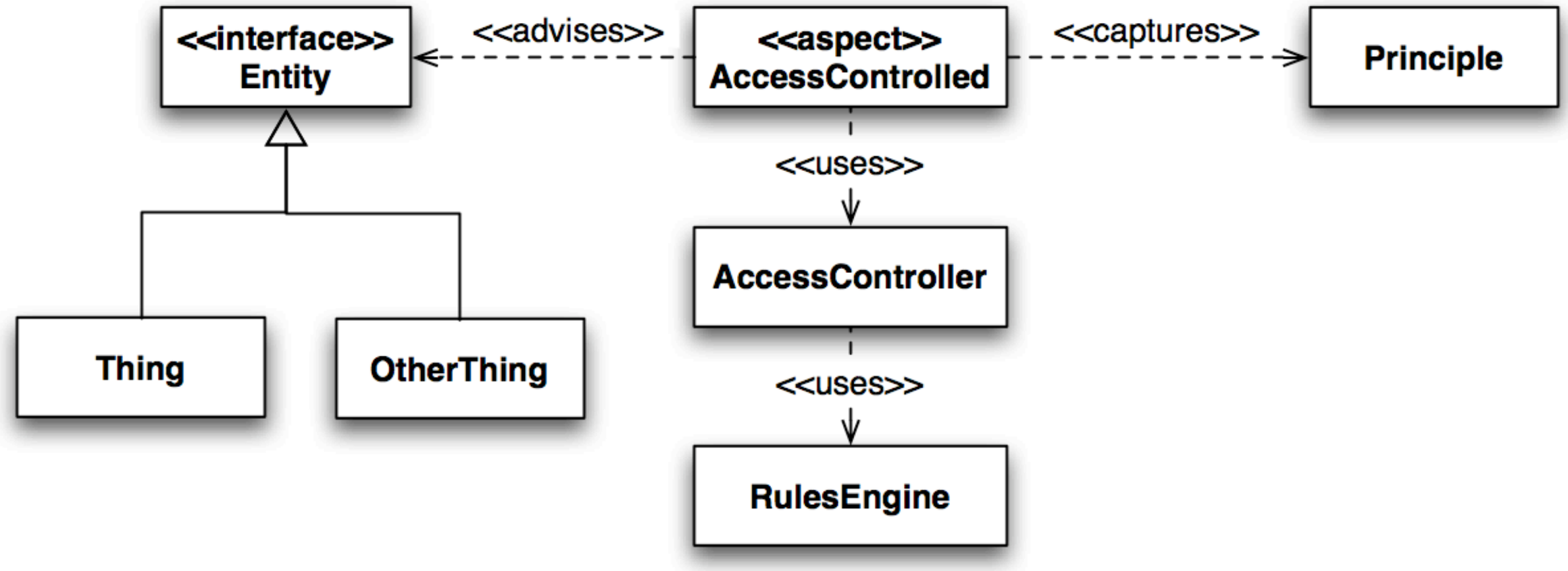
Design

- Rules system is a queryable service
- Lots of facts, small number of rules
  - Tiny by scale of traditional rule systems
- Rules implement core logic of the service

Instance Level Security

With lots of complex silly stuff

1000m View



Authorization Based On...

- Role
- Entity Properties
- Organizational Affiliation
- Differential Data Expiration
- Locality
  - EU Data Protection Laws

(Blame Marketing)

Facts

- Subject (user)
- Instance to check access for
- User Roles
  - Obtainable from principal, but more efficient to just assert
- User's organizational affiliation
- User's home country
- Today's Date

For Example

```
rule: Acme Parts Managers See All
   param: Organization => org
   param: Role => role

   condition: role == Role.MANAGER
   condition: org.name.equals(“Acme Parts”)

   consequence: token.sufficient()


rule: Acme Parts Data Expires in Two Weeks
   param: Organization => org
   param: Date => today
   param: Entity => it

   condition: org.name.equals(“Acme Parts”)
   condition: today > (it.expiresOn + (Dates.WEEK * 2))

   consequence: assertFact(Facts.STALE_ENTITY)
```

Design

- Rules modify behavior of core domain model
- Implemented as a concern
- Queryable as a service
  - This wasn't used except in testing though
- Lots of rules, few facts

Where Else?

- Application Customization
- Directed Questioning
- Complex Validation
- Business Automation
- Workflow / Orchestration
- "Routing" problems

Questions, Discussion, Commentary?

Thank you!

Brian McCallister
http://www.chariotsolutions.com/
brianm@chariotsolutions.com

Resources

- Drools
  - http://drools.org/
- CLIPS
  - http://www.ghg.net/clips/CLIPS.html
- JavaRules
  - http://www.javarules.org/
- JSR-94
  - http://www.jcp.org/en/jsr/detail?id=94