Introducing take

take – a rule compiler for Java

Jens Dietrich, feb 2008

background

- rules part of AI vision
- expert systems (production rules), Prolog (derivation rules)
- successful commercial “rule engines”
- idea: rules support easy specification of models, intelligent computer program can be used to find intended models

the comeback of the rules

- business rules: managing them as data shortcuts SDLC – useful to deal with dynamic business processes
- web: rules are useful to reason about resources (e.g. mail filters), establish trust (eCommerce)

rules and OO

- rules got divorced from OO in the 90ties
- rule engines try to fix this
- most engines based on production rules (button up or forward reasoning, usually using RETE)
- open source: Drools / JBoss rules, JESS
- Commercial: ILOG, Fair Isaacs
- API: JSR-94 (very weak)

why backward reasoning

- clear semantics (Herbrand model, based on PL1)
- can record the derivation tree (explains what is does)
- does not have to cache resources
- fits well into pull based system landscape (RDBMS queries, HTTP requests [end user, REST])
- not state changing – side effect free

mandarax

- open source lib for derivation rules
- popular, commercial applications
- very flexible, and very complex
- Java classes are types
- methods can be wrapped as functions and predicates
- integration of external knowledge sources as iterators (ClauseSets)
- scalability problems

new in Java (J2SE 6)

- compiler API: official API to access the compiler (JSR199)
- adhoc solutions has been used before (e.g., to run JSPs)
- Scripting API: integrate scripting languages (JSR233)

compiling rules at runtime

- JSP like
- generate source code (tmp folder, memory buffer)
- generate API
- compile (Java6 API or ANT for older JDK compatibility)
- load classes (and unload old classes if necessary)

using take

SCENARIO 1 - STATIC

- compile rules into classes
- integrate classes into project code
- deploy project

SCENARIO 2 - DYNAMIC

- compile rules into interfaces
- integrate interfaces into project code
- deploy project
- Generate, compile and deploy classes at runtime

scripting

- scripts can be used to instantiate rules
- Eclipse plugin provides script editor
- features: annotations, queries, rules, comments, external data sources
- scripting based on JSP-EL, provides easy access to Java objects and their properties:
  - if employee.salary>80000 then ...

scripting - example

global annotations (meta info)

variables and objects references in rules

local annotations used to define names in code generated for queries

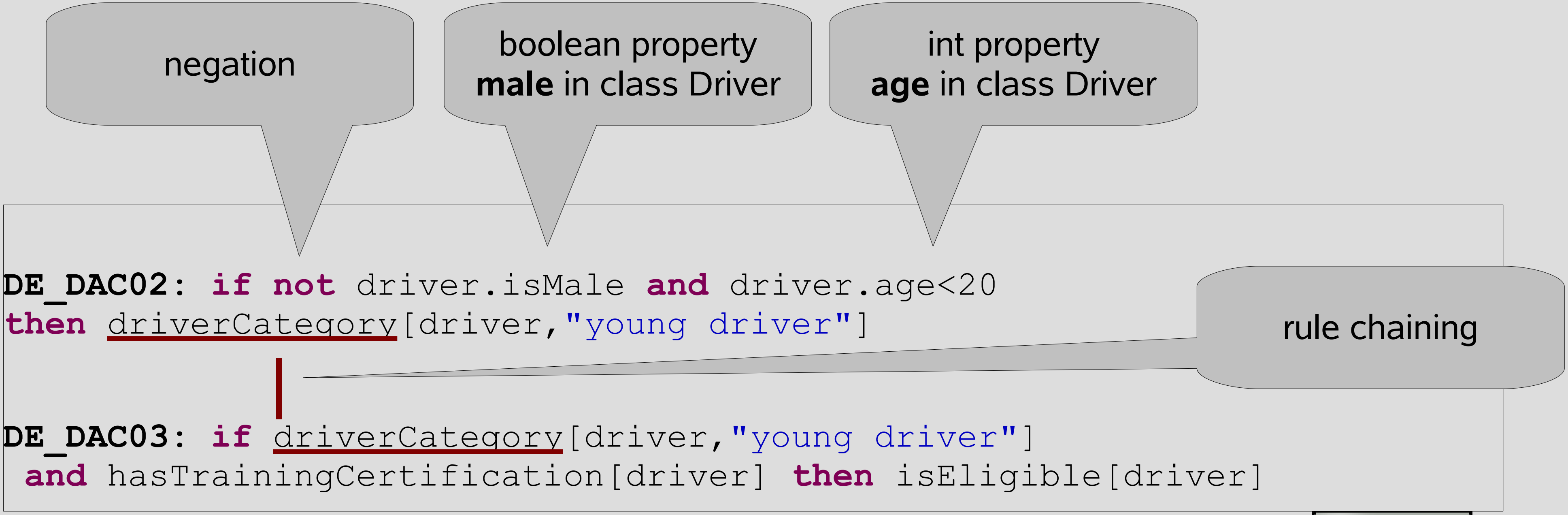
query

rules

```
Userv Business Rule Scenario

@@dc:creator=Jens Dietrich
@@dc:date=2007-09-12
@@special=true
import example.nz.org.take.compiler.userv.domainmodel.*;
var Car car
var Driver driver,client

// queries
@take.compilerhint.class=PotentialTheftRating
@take.compilerhint.slots=car,rating
@take.compilerhint.method=getPotenialTheftRating
query potentialTheftRating[in,out]

@category=Auto Eligibility Rule Set
@description=If the car is a convertible, then the car's potential
   theft rating is high.

AE_PTC01: if car.isConvertible then potentialTheftRating[car,"high"]
```

expressiveness

- methods can be used as functions in complex terms
- methods returning a boolean can be used as predicates
- properties can be used as predicates
- full use of Java arithmetic
- JavaScript like



negation

boolean property
male in class Driver

int property
age in class Driver

rule chaining

```
DE_DAC02: if not driver.isMale and driver.age<20
then driverCategory[driver,"young driver"]

DE_DAC03: if driverCategory[driver,"young driver"]
 and hasTrainingCertification[driver] then isEligible[driver]
```

static compilation

- Location defines src/bin locations
- NameGenerator defines names of methods and classes to be generated
- class **com.example.userv.rules.UservRules** will contain query methods

```
BasicConfigurator.configure();
nz.org.take.compiler.Compiler compiler = new DefaultCompiler();
compiler.setNameGenerator(new DefaultNameGenerator());
compiler.setLocation(new DefaultLocation());
InputStream script =..; // script location
ScriptKnowledgeSource ksource = new ScriptKnowledgeSource(script);
compiler.setPackageName("com.example.userv.rules");
compiler.setClassName("UservRules");
compiler.compile(ksource.getKnowledgeBase());
```

generated code

```
query potentialTheftRating[in,out]
```

```
public class PotentialTheftRating {
    public PotentialTheftRating() {
        super();
    }
    public Car car;
    public String rating;
}
public class UservRules {

    public ResultSet<PotentialTheftRating> getPotenialTheftRating(
        final Car car) {
        ...
        return _result;
    }
...
```

generated code (ctd)

- large parts are template based (velocity) and can be customized
- uses iterator pattern, with empty, singleton, nested and chained iterators (inspired by Apache commons collection library)
- Iterator pattern extended by close() method to ResourceIterator – can release associated resources
- Userv example: 69 rules -> 111 source code files, 354 classes, parsed and compiled in < 3s on average 2007 PC

querying the rules

- queries are plain methods
- methods return result sets
- Result sets are iterators (extend java.util.Iterator)
- have additional close() method
- have additional method to access the derivation
- class generated for query predicate is generic parameter

```
Car car = ...;
UservRules kb = new UservRules(); // generated
ResultSet<PotentialTheftRating> result = kb.getPotenialTheftRating(car);
String rating = result.next().rating;
```

dynamic compilation

- compiler creates only interfaces
- at runtime, create implementing classes into tmp folder with generated version name
- load with special classloader – can be later used to undeploy
- Simple generic KnowledgeBaseManager facilitates this

```
KnowledgeBaseManager<UServ> kbm = new KnowledgeBaseManager<UServ>();
Map<String,Object> bindings = new HashMap<String,Object>();
Userv kb = kbm.getKnowledgeBase(
    Userv.class,
    new ScriptKnowledgeSource(Tests.class.getResourceAsStream(
    "/userv.take")),
    bindings
);
```

semantic reflection

- generated code has reference to rule meta data
- can be used to query specs used: which rules have been used for this decision, who has entered this rule and when?
- improves trust in systems – overcomes limitations of blackbox approach

```
ResultSet<PotentialTheftRating> rs = .. kb.isAvailable10(c);
IsAvailable result = rs.next();
List<DerivationLogEntry> log = rs.getDerivationLog();
for (DerivationLogEntry e:log) {
  Map<String,String> annotations =
    kb.getAnnotations(e.getName());
  System.out.println("rule last modified on: " +
    annotations.get("date"));
  System.out.println("rule last modified by: " +
    annotations.get("author"));
}
```

external fact stores

- facts supplied by databases, web services etc
- compiler creates simple interface, programmer must supply implementation
- example: wrap an SQL query
- before KB is queried, an instance of the implementation class must be registered by the id used for the fact store in the script

external fact stores (ctd)

```
external DRCx:   hasBeenConvictedOfaDUI[Driver]
```

```
public interface ExternalFactStore4hasBeenConvictedOfaDUI {
    public ResourceIterator<hasBeenConvictedOfaDUI> fetch(Driver slot1);
}

 public class hasBeenConvictedOfaDUI {
    public Driver slot1;
    public hasBeenConvictedOfaDUI() {
        super();
    }
 }
```