Code Generation in the World of Business Rules

Michael Rosenblum
Dulcian, Inc.
www.dulcian.com
Definitions of business rules range from:
- “A nice way to write reasonable analysis documents”
- “The system itself”

Implementation rules => Developer’s world
Analysis rules => System Architect’s world
Part 1:
Rule-Based Systems -
Repository Access Approaches
Business Rule-Based Systems

- **Flexible enough** => to allow for future modifications which may not be foreseen at the time of development

- **Scalable enough** => to handle significant data growth over time

- **Generic enough** => to survive a significant platform/front-end change (Java, JSP, JSF, XMLDB,...etc.).

- **Fast enough** => to produce specified level of performance, given limited resources
Business Rules Repository:
Real world and IT

- Developers use SQL and analysts use English.
- Business rules have to be centrally stored in the database to communicate.
Since the repository is inside of the database:

- **Workflow can be dynamically controlled.**
  - **Interpreters** => run-time access to business rules

- **Workflow could be represented as database objects.**
  - **Generators** => run-time access to generated objects
Interpreted access example

- Business rule: “The start date of an employee’s timesheet should not be later than its end date.”

- Implementation:
  - Detect validation point.
  - Find appropriate rule in the repository.
  - Translate rule from business terms to database terms.
  - Validate original object using interpreted rule.
  - Interpret the result.
  - Flag main routine regarding the success or failure of the rule and execute appropriate actions.
Pros and Cons of Interpreters

◆ Pro:
   ➢ Changes to business rules and changes to their interpretation have an immediate impact on the system.

◆ Con:
   ➢ Major performance drawback because the database must perform many operations just to compare two dates.
Business rule: “The start date of an employee’s timesheet should not be later than its end date”.

Implementation:
- The validation firing point should be the state change of the timesheet from NotSubmitted to Submitted.
- In the database, that rule could be implemented on the BEFORE UPDATE trigger.
- If a rule fails, an exception must be raised.
Example Code

CREATE OR REPLACE TRIGGER timesheet_bu
BEFORE UPDATE ON timesheet
REFERENCING NEW AS NEW OLD AS OLD
Begin
  if updating('state_cd')
  then
    if :new.state_cd='Submitted' then
      if :new.start_dt>=:new.end_dt then
        raise_application_error(-20999,
        'Rule 10 violated: start date cannot be later than end date');
      end if;
    end if;
  end if;
end if;
end;
Pros and Cons of Code Generation

◆ Pros:
  - Performance benefits

◆ Cons:
  - Any modification to the business rule now will require regeneration of the objects that reference it.
  - Significant problems in a production system because of Oracle feature of invalidating all objects referencing recompiled ones.
Common feature of both approaches

Both approaches have one major concept in common:

The implementation of the business rules is independent of their specification.

Business rule: “Start date of the employee’s timesheet should not be later than its end date”

- No mention of tables, columns, queries etc.
- System architect’s problem to determine the appropriate implementation mechanism.
Implementation differences

◆ **Interpreted approach**
  - Translate the rules into database terms and the results from the database terms by generating SQL, XML, conversion maps etc.

◆ **Compiled approach**
  - Generate database or other objects using PL/SQL, Java, XML, JSPs, etc. in order to implement the business rules.
Role of generators in rule-based system

- Generators can be tuned or changed **without touching** the business rules.
- Expected performance could be achieved **without a major re-architecting** of the system.
- If new requirements can be stored in the repository, the generators can be updated to support them.
  - Other parts of the system will not be affected by the changes.
- Generation **algorithms can be changed** depending upon the available data volume, system configuration, etc.
- Generators can be **substituted or extended** to work with other languages and platforms without altering the business rules.
Important!

No business rules-based system can be implemented without some type of code generator.
Part 2: Interpreters
Interpreters

◆ In the world of business rules:
  - “Interpreted” ~ queries are built on the fly via generic routines.

Begin
    Execute immediate "select ... from ... " into ...
End;

◆ Generic routines cover all possible tasks as broad as possible.
◆ Repository is accessed each time we need to build executable code.
Interpreters:
1. Declarative generators

- Many attempts to create a pseudo-language to allow definition of rules written in English to be easily translated into a set of database commands and conditions.
- Most attempts did not perform as promised.
- Few IT environments need to support thousands of declarative rules:
  - (Ex. “If gender is male and age is above 45, then recommend yearly heart checkup”).
  - Even in very large systems, there may only be a few hundred of them spread around the large data or process model.
Transition with the rule on it.
- If it succeeds, the object will be moved to the state “Recommend heart checkup.”

From the repository point of view, the rule should look like:

- **RuleId**: 11
- **BelongsTo**: Transition 10
- **RuleText**: gender is male and age is above 45
Solution

- Add one extra column for each implementation environment (SQL, Java etc).
- Column is populated after the first cycle of analysis by the software developer based on the text of the rule:

  ```
  ExecutableRule:
  emp.gender='Male' AND emp.age>=45
  ```
Generic validation routine

- Pass *table* (where the object is stored), *primary key*, and *column* to store the primary key into the function => allow the system to uniquely identify the desired *object*.

- Pass *transition* to be able to select the applicable *rules* and their translation to PL/SQL.

- Use Dynamic SQL => check *rules* against identified *object*. 
function f_validate (pin_pk_id number, pin_pk_column_cd varchar2, pin_class_cd varchar2, pin_trans_id number)
return boolean is
    cursor c1 is
        select ExecRuleTx from ste_rule where trans_id = pin_trans_id;
    begin
        open c1;
        loop
            fetch c1 into v_rule_tx;
            exit when c1%notfound or v_hasfailedrule_b=true;
            execute immediate 'select count(*) from'||pin_class_cd||'
            where'||pin_pk_column_cd||'='||pin_pk_id||
            ' and ('||v_rule_tx||')' into v_out_nr;
            if v_out_nr=0 then
                v_hasfailedrule_b:=true;
            end if;
        end loop;
        close c1;
        return v_hasfailedrule_b;
    end;
2. Event-Condition-Action Generators

- A set of UI elements:
  - button1 (named “Set Default End Date”),
  - textField1 (contains start date),
  - textField2 (contains end date)
- Elements may have events
  - button1 is associated with the event “Press.”
- Events may have conditions:
  - “End date is null.”
- If the condition is satisfied, the event has a set of actions:
  - Set end date equal to start date + one month both on the screen and in the database.
  - Disable the end date field.
Algorithm

- Notify the database about the event.
- Generate a list of appropriate actions.
- Check all corresponding rules for each action.
- Retrieve the list of actions that correspond to the event.
- Interpret list of actions at the client side into tool-specific command.
function f_checkRule (pin_action_id number) is
  cursor c1 is
  select r.condition_tx, r.executable_rule_tx, r.Class_CD
  from ar$rule r
  where action_id = pin_action_id;
  ...
begin
  open c1;
  loop
    fetch c1 into v_rec;
    exit when c1%notfound or v_hasfailedrule_b=true;
    execute immediate
      'select count(*) from '||c.Class_CD||
      ' where '||pin_pk_column_cd||'='||pin_source_object_id||
      ' and ('||v_rec.executable_rule_tx||')' into v_out_nr;
    if v_out_nr=0 then
      v_hasfailedrule_b:=true;
    end if;
  end loop;
  close c1;
  return v_hasfailedrule_b;
end;
3. Communication Interface Generators

**Requirements:**
- Front-end environment working against XML-based forms.
- Set of APIs only worked with specially formatted XML documents.

**Problems:**
- Can’t expect changes in the API
  - Need to create two-way parser
- Documents are specially formatted
  - Not possible to use internal XML parsers from the database.
Solution: Database to XML

- Map existing data into the appropriate XML tags. (discussed later)
- Generate an XML-document procedurally as a regular text document from the maps.
- Store the original CLOB in the table.
- Query CLOB from the client side.
- Convert the text into an XML-document.
- Apply the required APIs.
Solution: XML to Database

- Compare the modified XML-document with the master copy.
- Store the differences as CLOBs in the table.
- Map XML tags to real columns.
- Update identified columns with new values.
Challenges

- Manual creation of XML
  - Some characters are special for XML (">","<","&"," %")
  - Others could be special in the database (single quotes, characters from different languages).

- A converter of strings into the XML-compatible format may be useful.
Part 3: Compilers
In the world of business rules:

- “Compiled” ~ some physical objects (tables, procedures, files, etc.) will be created.

```
Begin
    Execute immediate "create or replace package ...";
End;
```

- Objects fully (or as much as possible) represent the set of business rules
- Executable code will need to access the repository fewer times.
Compilers: 1. Data Models

- Data models represent structural business rules.
- **Dynamic SQL** => all database objects (tables, views, constraints, triggers) could be generated from the repository
- **More than one way** to implement the same rule:
  "field <gender> can only have values male/female":
  - FK from the EMP table to the reference table.
  - Check constraint on EMP
  - Before-Update trigger
- **Possibility to rename/alter items in real time** => implement business rule changes on the fly.
Main task: Create Original Code

**Purpose:**
- Test versions
- Quick prototypes

**Advantage**
- Using the generator means that you always know what is going on in the database.

**Challenge**
- Limited notation of ERD => problem of implementing UML in relational database
Example

◆ Maintaining historical records on a class.

- Add two new columns to the table:
  start_dt and end_dt
- Add three new columns to the view:
  start_dt, end_dt and active_yn
- If end_dt is populated => object is inactivated.
- If end_dt is set to Null => object is reactivated.
- Add Before-(Insert, Update, Delete) to the view to prevent any activity on the object if it is inactivated except for update of end_dt
Main task: Maintain changes

◆ Purpose:
  - Implementation of changes to the rules repository

◆ Challenge
  - Need to maintain a strict one-to-one relationship between the definition of a business rule and its implementation.
Example

◆ Modify name of the class

```sql
procedure setTableName(in_class_id number, 
in_oldclasscode_tx varchar2, 
in_newclasscode_tx varchar2) is

v_ddl_tx varchar2(2000);
begin
   v_ddl_tx:='rename '||in_oldclasscode_tx||
      ' to '||in_newclasscode_tx;
   execute immediate v_ddl_tx;

   v_ddl_tx:='alter table '||in_oldclasscode_tx||
      ' rename column '||in_oldclasscode_tx||'_'OID ||
      ' to '||in_newclasscode_tx||'_OID';
   execute immediate v_ddl_tx;
end;
```
A workflow can (and sometimes should) be represented as generated code.

Major drawback of implementing advanced process flows is the large number of repository requests.

If all communications between different states of the flows could be generated, not much else is required.
Dulcian STE notation

- Extension of UML activity diagram:
  - Classes can have workflows consisting of states.
  - States can have events.
  - States are connected by transitions.
  - Transitions are initiated by special kinds of events.
  - Transitions can have rules. If a rule fails, then navigation via the transition is impossible.
  - Events can have rules. If a rule fails, then the event is aborted.
  - Events and transitions can have corresponding tasks.
Sample state with transitions

**State #1**
SELECT TYPE OF HEALTH CHECKUP
BeforeOpen Event

**State #2**
REQUEST YEARLY HEART CHECKUP
Trans #10
Rule #1: Males over 45

**State #3**
REQUEST YEARLY MAMMOGRAM
Trans #20
Rule #2: Females over 45

**State #4**
SKIP CHECKUP
Trans #30
procedure p_auto_1(SelfOID in Number) is
Begin
/*BeforeOpen*/
emp.setHealthValidationDt(sysdate);
/:HealthValidationDt:=sysdate*/

if (emp.getAge(selfOID)>=45 and emp.getGender(selfOID)='Male') then
/*(:Age >= 45 and :Gender='Male')*/
  ste$pkg.setState(SelfOID,2,10); -- object, state, transaction
elsif (emp.getAge(selfOID)>=45 and emp.getGender(selfOID)='Female') then
/*(:Age >= 45 and :Gender='Female')*/
  ste$pkg.setState(SelfOID,3,20); -- object, state, transaction
elsif 1=1 then
/*No Rule*/
  ste$pkg.setState(SelfOID,4,30);
end if;
End;
Advantages of extended UML notation

- Higher level of abstraction in the definition of a state (about 1 high-level state for every 60 in a regular flowchart)
- Smaller number of logical structures involved (IF..THEN, LOOP, etc.) => simplify the code
- Predefined and limited number of events => precisely identify the elements to be generated
- Any event (with rules and tasks) can be represented as a set of commands in any procedural language.
- Any transition (with rules and tasks) can be represented as a set of commands inside of the initiating event.
Compilers: 3. Data mappings

- Problem working with XML-based forms:
  - Architecturally, forms were exact copies of the paper forms
  - Data model was significantly different from what was shown on the screen.

- Challenge:
  - Map stored data into XML tags.
  - Mandatory to decrease workload on the client machines.
Solution

- Full two-way conversion of stored data into the precise data representation required for the client code.
- Special mapping repository was created to carry out the conversion.
Solution (continued)

- Originally DB → XML and XML → DB maps were used.
- Later this architecture was extended to support DB ↔ DB maps => extremely powerful migration utility
  - Generic definition of the source and target => migration maps between completely different data models of any level of complexity.
Conclusions

So what?
Advantages of using generators

- **Build systems “better”:**
  - Improve flexibility, performance, maintainability, scalability etc.

- **Build systems “faster”:**
  - 90% of the code is generated => less time to create, less time to implement the change.

- **Build systems “cheaper”:**
  - Shorter development cycle.
  - One top-level developer is still cheaper than 10 low-level.
  - Less chance for human mistakes, miscommunication etc.
Guidelines for using code generators

- Keep them in mind from the very beginning of the project.
  - Flexible data sources => generate queries
  - Logical processing => generate workflows
  - Process a lot of data => generate batch workflows
  - Future extensions of the system => generate data model
  - Data transformations => generate maps
  - Communication with other language systems => create converters.
Dulcian’s BRIM® Environment

- Full business rules-based development environment
- For Demo
  - Write “BRIM” on business card
- Includes:
  - Working Use Case system
  - “Application” and “Validation Rules” Engines
Contact Information

- Michael Rosenblum mrosenblum@dulcian.com
- Code examples from these slides are available in the accompanying paper on the Dulcian website
  www.dulcian.com
- See Conference Papers and Presentations/Presentations by Conference/ODTUG 2005