Translation of Use Case Scenarios to Java Code

Michał Śmiałek, Norbert Jarzębowski, Wiktor Nowakowski

Warsaw University of Technology

Krajowa Konferencja Inżynierii Oprogramowania
Kraków, Poland, September 10-13, 2012
Presentation outline

- Introduction
- Short introduction to RSL (use case scenario language)
- Side note: complexity of software systems
- Transformation rules
- Transformation program
- Validation and conclusion
The problem

- Use cases are quite imprecise
- Not possible to automate the transition from requirements to code
- Quite sparse attempts to treat use cases as “first class citizens” in software development
- Idea: bring use cases closer to code
Use Case model

Transformation Rules

RSL scenario model

Java code

MOLA program

UML design model + Java method bodies
Short introduction to RSL
General idea: separate the story from the domain
RSL = use case scenarios + domain vocabulary

Scenarios refer to notions and phrases from the domain vocabulary.

Domain vocabulary contains definitions of notions and phrases in the context of the system’s domain.
Phrases: basic building blocks in RSL

**Phrase**

- **NounPhrase**
  - * modifier 0..1
  - 1 noun
  - * verb

- **VerbPhrase**
  - 1 object
  - 0..1 verbPhrase

- **SimpleVerbPhrase**
  - 1 verb

- **ComplexVerbPhrase**
  - 0..1 preposition

**Terms**

- **Modifier**
  - name :String

- **Noun**
  - name :String

- **Verb**
  - name :String

**Example**

- register
- selected student
- for
- course
Grouping phrases within notions

- DomainElement
- Relationship
- DomainElement Multiplicity

- Notion
- Noun
  - name : String

- Phrase

- Course
  - add student to course
  - remove student from course
  - save course

- Class
  - modify class
  - check availability of class

- Course entry form
  - show course entry form
**Full RSL model - example**

### Requirements representations

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Customer wants to sign up for exercises.</td>
</tr>
<tr>
<td>2.</td>
<td>System checks availability of exercises.</td>
</tr>
<tr>
<td>3.</td>
<td>System shows time schedule.</td>
</tr>
<tr>
<td>4.</td>
<td>Customer chooses time from time schedule.</td>
</tr>
<tr>
<td>5.</td>
<td>System shows sign-up summary dialog.</td>
</tr>
<tr>
<td>7.</td>
<td>System signs up customer for exercises.</td>
</tr>
</tbody>
</table>

### Domain vocabulary

- **customer**
  - submit sign-up for exercises
- **exercise**
- **time schedule**
  - show time schedule
  - choose time from time schedule
- **sign-up summary dialog**
  - show sign-up summary dialog
Side note: complexity of software systems
Essential and accidental complexity

„The *essence* of a software entity is a construct of interlocking concepts: data sets, relationships among data items, algorithms, and invocations of functions. 
[...] such a conceptual construct is the same under many different representations.” *

„[...] *accidental* tasks arise in representing the construct in language.” *

* F.P. Brooks, ”No silver bullet: Essence and accidents of software engineering”
In most well-designed multi-tier systems, the software essence is realized by the application logic and the domain logic.

In typical development process, the essence of the system to be build is specified in details within a software requirements specification.
Transformation rules
Overview of the approach: target architecture

**Model**
- `MCourse`
  - `fetches(aCourse : XCourse)`: void
  - `getResult()`: int
  - `saves(aCourse : XCourse)`: void
  - `validates(aCourse : XCourse)`: void

- `MCourseList`
  - `builds(aCourseList : XCourseList, aTeacher : XTeacher)`: void
  - `getResult()`: int

**Presenter**
- `CAddNewCourse`
  - `_SelectsAddNewCourseOption()`: void
  - `_SelectsAddNewCourseOption(invokingUC : IInvoke)`: void
  - `SelectsOK(pCourse : XCourse)`: void
  - `SelectsOK_2()`: void

- `CShowOwnedCourseList`
  - `_SelectsShowCourseListOption()`: void
  - `SelectsOK()`: void
  - `invokeAddCourse()`: void
  - `invokeEditCourse()`: void
  - `returnInvokeResult(res : int)`: void

**View**
- `VCourseForm`
  - `displays(aCourse : XCourse)`: void

- `VErrorMessage`
  - `shows()`: void

- `VCourseListForm`
  - `shows(aCourseList : XCourseList)`: void

**Application logic**

**Domain logic**
Rule 1: generate the “presenter” classes
Rule 2: generate the “model” classes & DTOs
Rule 3: generate the “view” classes
Add new course

Rules 3 and 4: generate procedure calls

```
public void _SelectsAddNewCourseOption()
    { vCourseForm.displays(aCourse); }
```

```
public VCourseForm()
    { /* ... */
        jbnOK = new JButton("OK");
        this.add(jbnOK);
        jbnOK.addActionListener(
            /* ... */
            jbnOKActionPerformed(evt);
            /* ... */
        );
    }
```

```
private void jbnOKActionPerformed(java.awt.event.ActionEvent evt)
    { /* set aCourse from form fields*/
        cAddNewCourse.SelectsOK(aCourse);
    }
```

```
public void displays(XCourse pCourse)
    { aCourse = pCourse;
      /* set form fields from aCourse */
    }
```
Rules 5 and 6: generate “if”s and procedure calls

Add new course

3. Course manager selects OK sending course
4. System validates course
   => cond: OK
5. System saves course
   => cond: notOK
4.1.1 System shows error message
4.1.2 Course manager selects OK

CAddNewCourse

```java
public void SelectsOK(XCourse pCourse){
    int res;
    aCourse = pCourse;
    mCourse.validates(aCourse);
    res = mCourse.getResult();
    if (res == OK) {
        mCourse.saves(aCourse);
    } else if (res == notOK) {
        vErrorMessage.shows();
    }
}

public void SelectsOK_2(){
}
```

MCourse

```java
public void validates(XCourse pCourse){
}

public void saves(XCourse pCourse){
}
```
Transformation program
Transformation language: MOLA

- MOdel transformation Language
- Graphical language
- Based on metamodel-level pattern matching
- Transformation (program) describes how to transform one model into another (RSL to Code)
Example MOLA procedure
Validation and conclusion

- RSL already validated by industry partners (ReDSeeDS Project)
- RSL to Java transformation currently being validated on a banking system (REMICS project)
- Productivity gains validated through student projects
- Significant improvements in productivity, especially for less skilled programmers
- Ready code ("view" and "presenter" parts) can be used directly in production system
- Ready code skeleton available instantly for the data processing ("model") part
Thank You!

www.remics.eu

www.redseeds.eu