Using enactable models to enhance use case descriptions

‘Banging on about enactment again’

Keith Phalp
Presentation to the ESERG Workshop, July 2003
Bournemouth University, UK
Supporting Use Cases
Our (group) context

- Elicitation. Process models, Use Cases and interfaces.
- Writing: Using writing rules, guidelines or templates.
- Assessing Quality.
- Comprehension: Questions and interrogation
- Validation and evolution
  - Dependencies and enaction. TOOL SUPPORT.
- Moving towards design.
  - Teasing out (hidden) issues.
  - Use case driven processes. Construction & validation
Research Rationale / agenda

• Use Case Descriptions do not have good tool support.
• Validation of descriptions has always been less easy than UML suggest.
• Enaction provides an excellent opportunity to validate descriptions.
• Enaction also enables consideration of later design issues.
Two sporting use cases

1. The match reached full-time
2. The referee blew his/her whistle
3. The ball crossed the goal-line
4. The goal was not given

Alternatives
4. The goal was given

1. The match reached full-time
2. The referee blew his/her whistle
3. The ball crossed the goal-line
4. The goal was given

Alternatives
4. The goal was not given

Validation & Context. Someone who ‘knows the the game’.
Real agenda

• With many process models (say with RolEnact) users are able to play with behaviour.
  – Lots of arguments about increased understanding, validation etc...

• Wouldn’t that be handy for specifications (as use cases)
  – it’s the old executable spec argument again (its so 80s).

• So my analogy is that of RolEnact, which I’ve talked about lots before,
  – (so will show examples - where I’m coming from).
Divisional Director
- New project approved
- Start new project manager
  - Agree TOR for project

Project Manager
- Start new designer
  - Write TOR for designer
  - Agree TOR and delegate
  - Prepare an estimate
  - Choose a method
    - Obtain estimate
    - Prepare a plan
      - Give plan to designer
        - Produce design
          - Carry out design quality check
            - Design OK?
              - Yes: Produce project debrief report
              - No: Deliver design
            - No: Produce project debrief report

Designer
- Start new designer
  - Write TOR for designer
  - Agree TOR and delegate
  - Prepare an estimate
  - Choose a method
    - Obtain estimate
    - Prepare a plan
      - Give plan to designer
        - Produce design
          - Carry out design quality check
            - Design OK?
              - Yes: Produce project debrief report
              - No: Deliver design
            - No: Produce project debrief report

Role
Activity Diagram (standard)
Interaction Role1.Interaction
  Me(before1 → after1)
  Role2(before2 → after2)
End

Example RolEnact code

Interaction Designer.deliver_design
  me(accepted_design → design_sent)
  Project_Manager(plan_sent → design_received)
End
An example enaction?
Experiences with Enaction

• Student experience:
  – Can write RolEnact equivalent to use case description and validate with enactment
    • helps tease out issues..
  – Role Activity Diagrams, RolEnact, Use Cases as part of a method
    • strong combination as a requirements validation mechanism

• Industrial experience:
  – Programming to enact each Use Case seen an unwelcome overhead. Not feasible for industrial application.
Use Case Enaction Tool(s)

• The prototype includes:
  – Pre- and post-conditions for each Use Case
  – Text editing capability for standalone Use Case
  – Default dependency capability and Branch dependencies (alternative / exceptions)
  – Enaction of the Use Case
  – Scenario generation of the path selected during enaction
  – Grammar enforcement capability
Previous version: Use Case Editor

<table>
<thead>
<tr>
<th>ID</th>
<th>Actor</th>
<th>Action</th>
<th>Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Borrower</td>
<td>presents item to borrow</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Librarian</td>
<td>requests borrowers' card</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Borrower</td>
<td>presents membership card</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>Librarian</td>
<td>validates membership details</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>Librarian</td>
<td>issues item requested</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>Borrower</td>
<td>receives item and leaves</td>
<td>5</td>
</tr>
</tbody>
</table>
Example Enaction

- The Customer inserts a credit card
- The System prompts the customer to enter PIN number
- The Customer enters his PIN number
- The System states that the PIN number is correct
- The System displays a menu
- The System prompts the customer to choose a menu option
- The Customer selects withdraw money option
- The System prompts the user to select amount
- The Customer selects £200
- The System prompts the user for a reject
Problems

• Abbreviated dependency mechanism only makes sense at system level / single actor.
• Strength (point) of enaction lost.
• Not helpful for considering AND, where two precondition on two or more actors.
  – Note AND implicit in an interaction.
• Currently revising interface.
Revised interface plan

<table>
<thead>
<tr>
<th>Actor name</th>
<th>Event</th>
<th>pre</th>
<th>post</th>
<th>Actor name</th>
<th>pre</th>
<th>post</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keith</td>
<td>gives pen</td>
<td>has pen</td>
<td>no pen</td>
<td>Mathenge</td>
<td>no pen</td>
<td>has pen</td>
</tr>
<tr>
<td>Mathenge</td>
<td>gives pen</td>
<td>has pen</td>
<td>no pen</td>
<td>Keith</td>
<td>no pen</td>
<td>has pen</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Actor name</th>
<th>Event</th>
<th>pre</th>
<th>post</th>
<th>Actor name</th>
<th>pre</th>
<th>post</th>
</tr>
</thead>
<tbody>
<tr>
<td>Me</td>
<td></td>
<td></td>
<td></td>
<td>Me</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Driver</td>
<td>drives to ticket machine</td>
<td>initial</td>
<td>at machine</td>
<td>Driver</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Driver</td>
<td>presses the ticket button</td>
<td>at machine</td>
<td>ticket requested</td>
<td>Ticket Machine</td>
<td></td>
<td>ticket requested</td>
</tr>
<tr>
<td>Ticket Machine</td>
<td>dispenses ticket</td>
<td>ticket requested</td>
<td>ticket dispensed</td>
<td>Ticket Machine</td>
<td></td>
<td>ticket dispensed</td>
</tr>
<tr>
<td>Driver</td>
<td>takes ticket</td>
<td>ticket requested</td>
<td>ticket taken</td>
<td>Ticket Machine</td>
<td></td>
<td>ticket taken</td>
</tr>
</tbody>
</table>

- See example?
Also for future Construction

- Levels of Usage
  - Advanced usage (detailed dependency selection) versus basic user.

- Multiple use cases
  - Depicting dependencies and enaction across use cases (via include and extend relationships)

- Further flexibility in editing the description
  - e.g., ability to re-order events simply.
Advantages of Tool Support: well here’s hoping

• Use Cases dependency examination offers insights into:
  – the problem domain, the requirements and later in subsequent design
  – and is important to requirements validation.

• Enaction thus provides this dependency scrutiny at ‘minimum’ effort for clients.
Some Issues for tool support

• Does the increased capability offered by dependencies enhance or overcomplicate descriptions?
• Will the inclusion of use case writing guidelines restrict the flexibility offered by enaction?
• Does the template approach to structuring use cases fit more naturally with tool support?
• Will requirements volatility make dependency mapping unmanageable?
• Do users really require models that consider dependencies across use cases, or does the restriction to consideration within a use case provide a partitioning of understanding?