Integrating Model-Driven Engineering Techniques in the Personal Software Process

João Pascoal Faria
Faculty of Engineering, University of Porto, Portugal
(jpf@fe.up.pt)
Index

- Background and motivation: PSP, UML, MDE
- A lightweight MDE approach
- Key features and benefits
- Refinements to PSP scripts
- Lessons learned from case studies
- Conclusions and future work
The PSP provides a set of specification templates for completely and precisely recording reviewable software designs covering 4 important design views:

- **External**
  - Operational ST
  - Functional ST
- **Internal**
  - State ST
  - Logic ST

- **Dynamic**
- **Static**
  - Functional ST
Background and motivation: PSP & UML

- UML is a standard visual notation for representing OO software designs, and is supported by many tools.
- Especially when enriched with:
  - contract specifications (pre/post conditions and invariants) in OCL
  - algorithm descriptions in a UML compliant action language
  - documentation notes and properties of relevant model elements

UML diagrams provide a convenient and familiar means for recording essentially the same info as the PSP templates.

<table>
<thead>
<tr>
<th>Dynamic</th>
<th>Static</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>External</strong></td>
<td>Class diagrams (+OCL or API doc)</td>
</tr>
<tr>
<td>Uses cases and sequence diagrams</td>
<td></td>
</tr>
<tr>
<td><strong>Internal</strong></td>
<td>Activity diagrams (flowcharts) or action specifications</td>
</tr>
<tr>
<td>Statemachne diagrams</td>
<td></td>
</tr>
</tbody>
</table>
Background and motivation: MDE (1)

- Although PSP is agnostic about the usage given to design specs/models: as documentation or compilable artifacts ...

- ...UML practitioners have concluded that building detailed design models for documentation only has several problems
  - is time consuming
  - the resulting models are often wrong
    - lack of static analysis, compilation, execution, etc., to spot problems
  - the resulting models soon become outdated and are not maintained

- Recent Model-Driven Engineering (MDE) approaches aim at avoiding such problems by generating code from models
  - If not production code (MDD), at least test code (MBT)
In fact, with that MDE approach (code generation from models):
- the time invested in building design models can be recovered
- the quality of the models can be checked
- there are higher chances that models are kept up to date

This is also more in line with the agile values
- (value more) *Working software over comprehensive documentation*

This will also help solving problems we found when introducing PSP training in academia, using UML as the design notation:
- Instructors time for grading and feedback is exacerbated when UML models are required for documentation only, because students don’t have a reliable means to check by themselves if the models are right
- Students see the cost of creating design models, but practically no short term benefits
Unfortunately, the level of detail of behavioral models needed to generate complete apps is often too high or only effective for specific domains (with domain specific languages).

So, we propose a lightweight MDE approach:

- develop structural models, from which parts of the application can be generated (e.g., class skeletons) (MDD)
- develop partial behavioral models, not sufficient for app generation, but adequate for test generation (MBT)

This is also inline with some agile practices (your tests are your specs, or vive-versa)
A lightweight MDE approach (2)

(Partial) Behavioral Model (= Test Model)

Structural Model

Production Code Skeletons

Test Code

Completed Production Code

static analysis (consistency & completeness)

assert

m()

m()

x=1;

}
Behavior modeling and testing
(at all levels: unit, integration, system)

- Actor (client app or user)
- Things in the system
- Things not yet implemented
- Example values for parameters

- Generated Test Code
  - (Driver) Generate inputs as in spec and check responses against spec
  - (Monitor) Trace execution and check against spec
  - (Stub) Generate the responses as in spec
  - Exercise the scenario for each example

- Behavioral Model/Spec
  - external interactions
  - internal interactions
  - intractions with things not yet implemented

Example values for parameters:
- Things not yet implemented
- Things in the system
- Actor (client app or user)
Process & tools

**DLD** (incl. test spec.)

**DLDR** (incl. static analysis)

**CODE**

**UT**

**CODE**

**CR**

**UT**

---

1. Model application structure & behavior

2. Check model consistency & completeness (UMLChecker)

3. Generate code
   - 3a. Generate production code skeletons from structural model
   - 3b. Generate test code from behavioral model (Test Generator)

4. Execute tests & see them fail

5. Complete production code (method bodies)

6. Execute tests & see them pass

---

**Artifacts**

- **Model**
  - Structural model (UML class diag.)
  - Behavioral model (UML seq. diag.)

- **Code**
  - Production code (OOP)
  - Test code (xUnit)

- **Reusable Libraries**
  - Standard libraries
  - Tracing library (AOP)

---

* J. Faria, A. Paiva, Z. Yang, Test Generation from UML Sequence Diagrams, Proc. of the 8th Int. Conf. on the Quality of Information and Communication Technologies (QUATIC 2012), IEEE CPS, 2012
Key features and benefits (1)

**Feature**

- Support the modeling & automatic testing of
  - External interactions with users (UI)
  - External interactions with client applications (API)
  - Internal interactions among objects in the program

**Benefits**

- Covers the 4 design views (w/ structural model)
- Assures higher conformance with spec
- Improves fault localization
- Accelerates test phase

<table>
<thead>
<tr>
<th>Dynamic</th>
<th>Static</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ext.</td>
<td>Class diagrams (public/external interfaces)</td>
</tr>
<tr>
<td>Int.</td>
<td>Class diagrams (private/internal interfaces)</td>
</tr>
<tr>
<td>Sequence diagrams (external interactions)</td>
<td>Sequence diagrams (internal interactions)</td>
</tr>
</tbody>
</table>
### Key features and benefits (2)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameterization</td>
<td>Keep behavioral specs as generic as desired</td>
</tr>
<tr>
<td>Combined fragments (alt, opt, loop, par)</td>
<td>Keep behavioral specs as simple as desired (focus on relevant interactions)</td>
</tr>
<tr>
<td>Loose conformance checking</td>
<td>Verifiable completeness criteria</td>
</tr>
<tr>
<td>additional or intermediate calls are allowed in implementation</td>
<td>Higher quality assurance</td>
</tr>
<tr>
<td>Automatic checking of model consistency &amp; completeness</td>
<td>Iterative implementation &amp; testing</td>
</tr>
<tr>
<td>“Stubs” inject the specified response messages for things marked as not yet implemented</td>
<td>Independence of external components</td>
</tr>
</tbody>
</table>
### PSP2.1 Development Script

<table>
<thead>
<tr>
<th>Step</th>
<th>Activities</th>
<th>Description</th>
</tr>
</thead>
</table>
| 1    | Design     | - Review the requirements and produce an external specification to meet them.  
- Complete Functional and Operational Specification templates to record this specification.  
- Develop a design model to describe externally visible system structure and behavior. (*)  
- Produce a design to meet this specification.  
- Record the design in Functional, Operational, State, and Logic Specification templates.  
- Refine the design model to describe internal system structure and behavior. (*)  
- (...) |

(*) Guidelines about diagrams, templates, completeness criteria, etc., in Design standard
## PSP2.1 Development Script (cont.)

<table>
<thead>
<tr>
<th>Step</th>
<th>Activities</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Design Review</td>
<td>- Follow the Design Review script and checklist and review the design.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Check the design model with a static analysis tool.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- ....</td>
</tr>
<tr>
<td>3</td>
<td>Code</td>
<td>- Generate initial production code from the design model.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- ...</td>
</tr>
<tr>
<td>4</td>
<td>Code Review</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Compile</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Test</td>
<td>- Generate initial test code from the design model.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- ...</td>
</tr>
</tbody>
</table>

### Exit Criteria
- A thoroughly tested program that conforms to the Coding standard
- Completed Design templates
- Completed design model consistent with the code
- ...
Lessons learned from case studies

- We validated the approach viability on a set of case studies
- Size metrics and savings are promising, as in a typical example:

<table>
<thead>
<tr>
<th>Item</th>
<th>Size unit</th>
<th>Manual</th>
<th>Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structural model</td>
<td>model</td>
<td>42</td>
<td>0</td>
</tr>
<tr>
<td>Behavioral model</td>
<td>elements</td>
<td>56</td>
<td>0</td>
</tr>
<tr>
<td>Subtotal</td>
<td></td>
<td>98</td>
<td>0</td>
</tr>
<tr>
<td>Production code</td>
<td>LOC</td>
<td>174</td>
<td>81</td>
</tr>
<tr>
<td>Test code</td>
<td></td>
<td>0</td>
<td>82</td>
</tr>
<tr>
<td>Subtotal</td>
<td></td>
<td>174</td>
<td>163</td>
</tr>
</tbody>
</table>

- We also found some manageable issues
  - Compilable models still need some doc. notes for human readability
  - More details fixed in design than usual
  - Very small iterations are problematic (same as for metrics collection)
Conclusions

- Presented a lightweight MDE approach
  - Based on lightweight behavioral and structural models
  - (Partial) production code and (full) test code generation from models

- That is “PSP friendly”
  - Covers the 4 design views (in a sense of “internal”)
  - Promotes complete (in a sense), precise and reviewable designs
  - Implies minimal changes to design scripts
  - Embeds test specification in the design phase (as behavior specs)
  - Is designed to bring short term productivity and quality benefits

- And “agile friendly”
  - Compilable models are not mere documentation
  - TDD/BDD [create a test = create an (external + internal) behavior spec]
Future work

- Conduct more extensive experiments, using the PSP measurement framework, to quantify the productivity & quality gains and better understand the contexts of applicability
- Devise a simplified way to specify exceptional behavior
- Extend the approach and tools to broaden its applicability
  - other target languages (now only Java)
  - other modeling tools (now only Enterprise Architect)
  - GUI testing (now, only command line interface testing), particularly for system testing
  - testing of time constrained, concurrent and distributed systems, particularly for integration testing
Thank You!

Questions?

Suggestions?