On the Test-Driven Development and Validation of Business Rules

Jens Dietrich
Institute of Information Sciences & Technology
Massey University
J.B.Dietrich@massey.ac.nz

Adrian Paschke
Internet-based Information Systems
Technical University Munich
paschke@in.tum.de

Setting the Scene

Software Development Lifecycle (SDLC) considered to be inappropriate for many projects – slow, difficult to manage change, if requirements are implemented they have changed.

Different solutions proposed:

1. Agile SE (extreme programming + others): speed up development process , facilitate backtracking (redesign).
   Emerging evidence that this might work, heavily supported by industry (in particular IBM).

2. Rule-Based Systems: develop tools to empower business users to change systems – avoiding the SDLC.
   New wave of commercial tools (ILog, BlazeAdvistor, Jess, ..).

3. Can we combine 1. + 2. ?

Extreme Programming

Introduced in the late 90ties Ken Beck, Ward Cunningham, Erich Gamma and others.

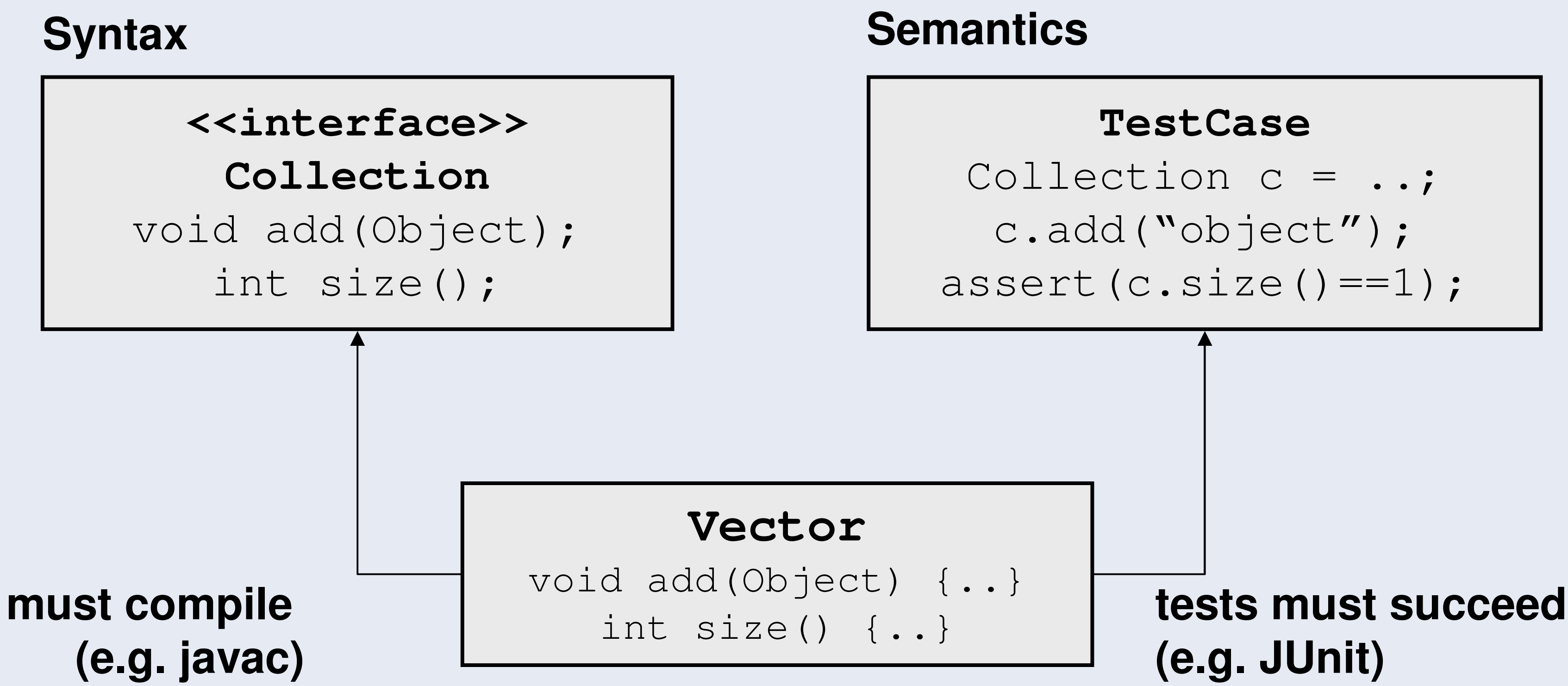
Similar approaches such as feature driven development. Umbrella term: agile software engineering.

Some XP ideas:

1. Write executable test cases first.
2. Little upfront design but evolving design. Permanent redesign supported by refactoring browsers.
3. Build often, tool supported builds, extremely short iterations.

Test Cases and Semantics

- The output of UML design is mainly a syntactical structure (classes and their APIs).
- It is cumbersome to add semantics (descriptions, OCL).
- Test cases can be used instead.



(Derivation) Rules

- Based on formal logic.
- We consider only derivation rules, but make no further assumptions about the logic (modalities, negation etc).

Rules - Syntax

Language L, fact base FB, rule base RB.
$\vdash_{\mathbf{RB}} \subseteq 2^L x L$

FB $\vdash_{RB}$ A if there exists proof using rules in RB
$Cn_{RB}(X) = \{A \mid X \vdash_{RB} A\}$

Cn usually monotonic.

Example: Resolution / unification as used in Prolog.

Generalization : replace monotony be weaker conditions (e.g. cautious monotony)

Semantics

M – class of models (e.g., true-false mappings, Kripke-models, PL models).

$\models \subseteq M \times L$

$(m,A) \in \models$ - "m is a model for A"

$Mod(x) = \{m \in M \mid m \models A \text{ for all a } \in X\}$

$Cn_{\models}(X) = \{A \in L \mid Mod(X) \subseteq Mod(\{A\})\}$

Consider only logics with $Cn_{\models} = Cn_{RB}$ (correctness and completeness) – our assumption is only logics with well understood meaning (semantics) and effective proof theory will be used to represent business rules.

Generalization to nonmonotonic logics: reasoning based on subsets of models $S(X) \subseteq Mod(X)$:

$C(X) = \{A \in L \mid S(X) \subseteq Mod(\{A\})\}$

Difficulties with Rules

It is difficult to understand the impact of changing rules.

After adding/updating/deleting a rule r, is a fact A still valid (e.g., $X \vdash_{RB} A$) ?

Problems:

- Order of rules might matter (priorities).
- Rules may contain variables.
- Rules may contain nested terms (function symbols).
- Rules may contain different connectives (strong/weak negations, deontic modalities, etc).
- Rule interaction: chaining, priorities, NAF.

Queries as Test Cases

On the other hand, simple queries can be used before modifying rules to describe the desired state of the rule base.

Simple queries means:

They are ground (no variables).

They can be flat (no functions).

They do not contain negation.

Syntax:

Q $\Rightarrow$ true // positive test case – expected outcome is true

Q $\Rightarrow$ false // negative test case – expected outcome is false

Example

V(c) – stands for “customer c uses a voucher for a purchase”

D(c) – stands for “customer c gets a special discount on a purchase”

E(c) – stands for “c is an employee”

G(c) – stands for “customer c is a Gold Customer”

FB = {G(a),E(b),G(c),V(c)} // fact base

Example (ctd) - Refinement

TC1 $\{?D(a) \Rightarrow true, ?D(c) \Rightarrow true\}$

RB1 = $\{G(x) \rightarrow D(x)\}$

MOD1 = $FB \cup \{D(a),D(c)\},\{\}$ // initial partial model

// employees also qualify for discount

TC2 $\{?D(a) \Rightarrow true, ?D(c) \Rightarrow true, \mathbf{?D(b) \Rightarrow true}\}$

MOD2 = $FB \cup \{D(a),D(c),\mathbf{D(b)}\},\{\}$

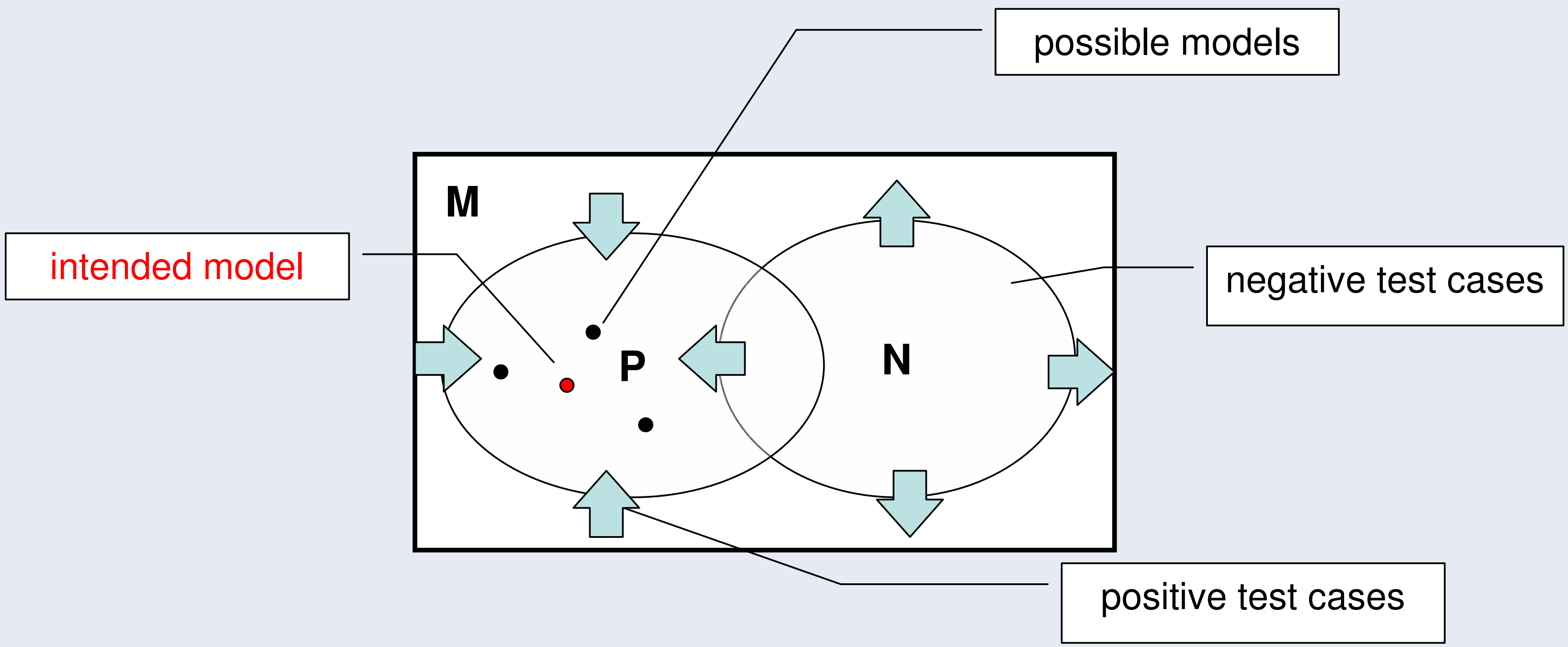
RB2 = $\{G(x) \rightarrow D(x), \mathbf{E(x) \rightarrow D(x)}\}$

Test Cases as Partial Models

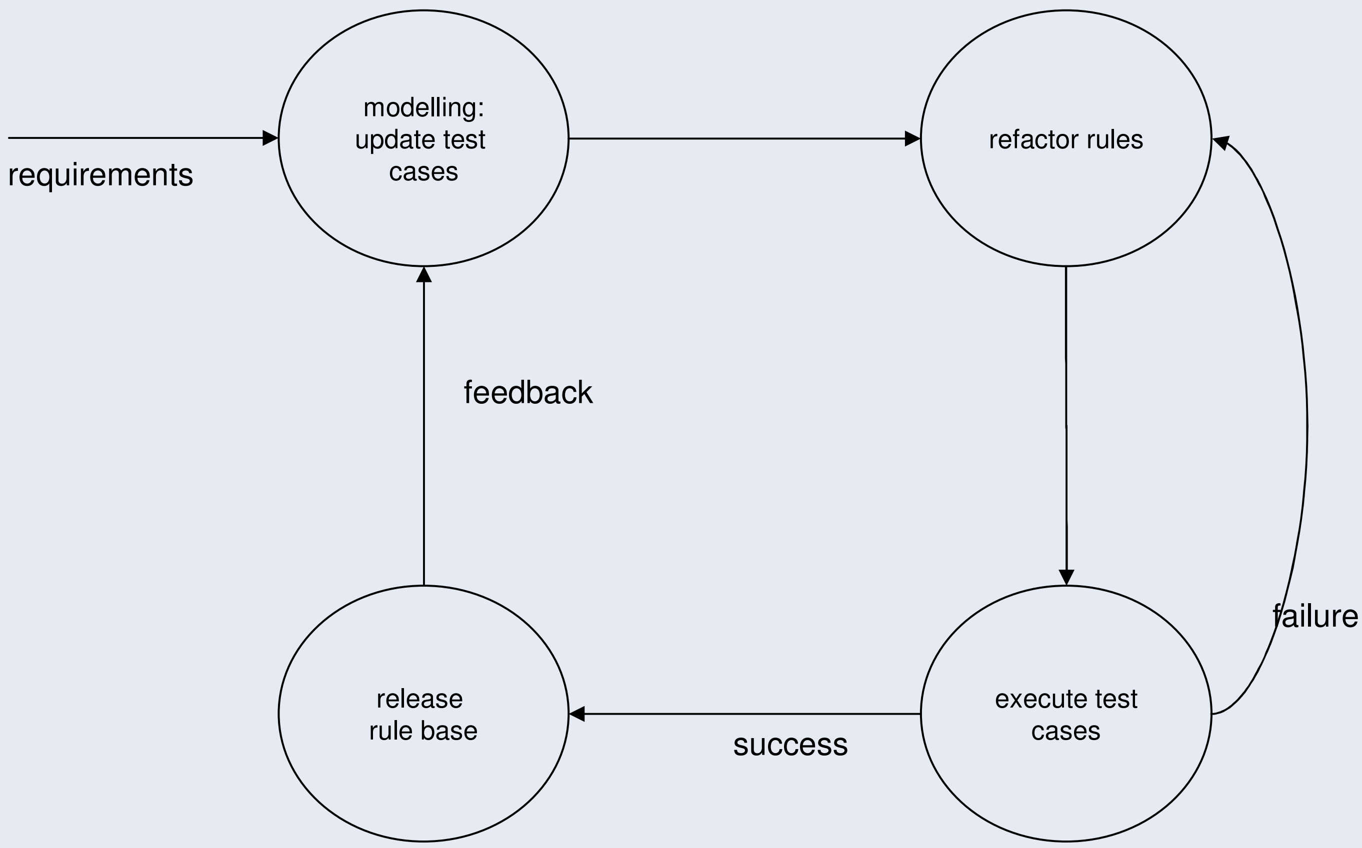
Set of positive/negative test cases P,N and class of models M.

$M(P,N) = \{m \in M \mid (m \models A \text{ for each } A \in P) \text{ and } (\text{not } m \models A \text{ for each } A \in N)\}$

M(P,N) constraints possible models and is an approximation of the **intended model**.



Testcases and the Lifecycle of Rules



“Proof of Concept” Implementation

- Based on Mandarax
- Test cases are part of the knowledge base.
- Test cases are persistent (e.g., XML).
- Test runner based on JUnit tool by K.Beck and E. Gamma.
- Supports test case lifecycle – assertions can be added before tests are executed and are removed after the tests.

Open Questions

- Assist user to find rules which are consistent with test cases.
- Measure the quality of test cases (similar to test coverage metrics used in SE).
- Generalizing this approach to cover non-monotonic logics.

Refactoring

- Tool supported redesign of rule sets.
- Invariants: test cases (after refactoring, the test cases should still succeed).
- Inspired by Refactoring Browsers (Smalltalk, RefactorIt) and Refactoring catalogues (M. Fowler: Refactoring).
- Smells – structures which need to be improved.

Smells

- Redundancy – There are redundant rules or rule fragments (for instance, shared subsets of prerequisites).
- Inconsistency – Different, inconsistent results are supported by the same rule set.
- Incompleteness – Certain queries can not be answered by a rule set.

Refactoring: “Exception to the Rule”

Name:	Exception to the Rule
Description:	A rule R does not apply in a particular situation. This situation can be described by a fact EXC.
Rule base before refactoring:	.. $A_1,..,A_N \rightarrow B$ ..
Rule base after refactoring:	.. $A_1,..,A_N,\neg EXC \rightarrow B$ ..
Addresses:	Inconsistency

Refactoring: "Narrowing"

Name:	Narrowing
Description:	Multiple rules share the same set of prerequisites.
Rule base before refactoring:	.. $A_1,..,A_{N,}A_{N+1}..\rightarrow B$ $A_1,..,A_{N,}A_{N+1}..\rightarrow C$ ..
Rule base after refactoring:	.. $A_1,.., A_{N,}A_{N+1}..\rightarrow A$ $A_{,}A_{N+1}.._{,..}\rightarrow B$ $A_{,}A_{N+1}.._{,..}\rightarrow C$ ....
Addresses:	Redundancy

Related to: Transformations of Logic Programs - Pettorossi, Proietti 96

Refactoring: “Narrowing”

Name:	Introducing a Default Rule
Description:	There are gaps in the rule set, i.e. there is no result for certain queries. A default rule is introduced to address this problem.
Rule base before refactoring:	.. $A_1,..,A_{N,..} \rightarrow B$ $A'_1,..,A'_{N,..} \rightarrow B$ ..
Rule base after refactoring:	.. $A_1,..,A_{N,..} \rightarrow B$ $A'_1,..,A'_{N,..} \rightarrow B$ $\rightarrow B$ ..
Addresses:	Incompleteness

Conclusion

Combination of Rule-based systems and principles from agile software engineering is promising.

TODOs:

- generate rules from test cases
- apply to special logics
- comprehensive list of refactorings

- Read the full paper:

ISTA 2005 proceedings http://www.gi-ev.de/LNI/