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

Abstract: In recent years we have seen the rise of a new type of software called business rule management systems (BRMS). These are systems to externalize business rules and to provide a facility for centralized business rule management. This addresses an urgent need businesses do have nowadays: to change their business rules in order to adapt to a rapidly business environment, and to overcome the restricting nature of slow IT change cycles.

Early manifestations of business rule engines which have their roots in the realm of artificial intelligence and inference systems were complex, expensive to run and maintain and not very business-user friendly. Improved technology providing enhanced usability, scalability and performance, as well as less costly maintenance and better understanding of the underlying inference systems makes the current generation of business rule engines (BRE) and rules technology more usable. However, there are a number of risks and difficulties that have to be taken into account when employing a BRMS.

Another recent trend that tries to address the same problem of slow IT change cycles is agile software engineering, in particular test driven development. In this paper, we investigate how BRMSs can be used in conjunction with test driven development. The result is an approach that facilitates the authoring of business rules significantly and safeguards it by providing means for automated validation and verification.

Keywords: Rule Engines, Business Rule Management Systems, Agile Software Engineering, Test-Driven Development, Refactoring

1. Introduction

In recent years, a new type of software called business rule management systems (BRM) has emerged. BRMSs are systems to externalize business rules[1], and to manage them in different modules which are then weaved into the main application at runtime. This follows the separation of concerns principle that has been applied successfully in many other areas of software development. For instance, visualization code is separated from the main application, done by specialized people (such as web designers) using special tools (such as web design tools). The situation is similar for concerns such as security and persistency. Well defined contracts between service provider, service broker and service consumer facilitate this approach. These contracts consist of formal or informal specifications, application programming interfaces and/or test suites.

Externalizing business rules decouples the development of rules from the software engineering process which is perceived to be too slow to respond to the rapidly changing business environment manifesting itself in changing business rules. This implies that a wide range of process models (Software Development Lifecycle Models), tools and services which have been developed in order to facilitate software engineering are not longer available to support the analysis, design, implementation and deployment of business rules. This process models, tools and services must therefore be adapted to be used with business rule management systems.

There are ongoing efforts to add support to model rules to existing methodologies. UML supports certain classes of business rules that can be expressed using the Object Constraint Language (OCL). The Object Management Group is working on the Business Semantics of Business Rules, which will include a MOF (meta object facility) meta model for business rules [Ba04]. This could form the foundation for tool supported engineering of business rules within the UML framework. Wagner and Halpin [HW03] have shown how certain types of rules can be modeled using object role modeling (ORM).

In this paper, we show how certain principles which are part of agile software engineering (in particular test- driven software development) can be adapted so that they can be used to model business rules. The main advantage of this approach is that the rules can then be validated automatically against the model. We narrow the scope of our investigation and consider only a particular class of rules, derivation rules.

In chapter 2, we present a short introduction into some of the aspects of agile software development which we consider to be important for our work. In chapter 3, we investigate how the idea of test driven development can be adapted to specify and approximate the intended user model, and how this is related to the validation and verification of rules. In chapter 4, we

---

[1] We use the following definition of business rules here: "Business rules describe the operations and constraints that apply to an organization in achieving its goals" [Wi]. Rules are usually narrative description of policies, procedure and principles which are then implemented in IT system, e.g. as code fragments or as integrity constraints in database systems. In the context of this document, we focus on the class of business rules which can be expressed as derivation rules, i.e. as IF ..THEN .. statements.

try to adapt the idea of refactoring and present a short list of refactorings for sets of rules. Chapter 5 presents an implementation of the ideas developed in the paper based on the mandarax library. In the last part of this paper we summarize our work and present related work.

2. Test Driven Software Development

Test driven software development is a practice which has become extremely popular in the software engineering community in recent years. Test driven development is one of the pillars of extreme programming [Be99] and a number of other approaches to software engineering usually subsumed under the term agile software development [Ma02]. Extreme programming tries to address that fact that established process models like the Rational Unified Process (RUP) have become so complex that they started to slow down and even paralyze software engineering projects which needed to response to rapidly changing requirements. Extreme programming proposes to solve the problem by:

1. Reducing the weight of upfront design, accepting and encouraging refactoring and assuming that this will lead to good design.
2. Requiring that new features are first described by test cases. These test cases describe the semantic of the program to be implemented, and the software can be automatically validated against these test cases using unit testing tools. Test cases are an important part of the assets produced during a software engineering projects and safeguard future refactoring.
3. Extremely short release cycles, usually supported by tools. Each cycle is characterized by the following steps: 1. updating of test cases (usually adding a new test case), 2. design and implementation until all test cases succeed (automated validation), 3. releasing a new version of the software.

Good tool support has been emerged for extreme programming in recent years, in particular build tools such as ANT which can automate many processes and can serve as universal project backbones, testing tools such as XUnit[2], refactoring browser and various fully fledged IDE which are integration platforms for these tools. In particular the Eclipse project [Ec] backed by IBM has been a driving force in this area.

Test cases are mainly about writing constraints which describe models. The following example should illustrate this. Let "List" be a Java interface describing a simple sequential list data structure. It has (at least) four methods with the following signature:

```
interface List {
        public Object get(int position) ;
        public void add(Object element) ;
        public void remove(Object element) ;
        public int size() ;
}
```

---

[2] X is replaced by a letter indicating the programming language, SUnit was the first implementation for the Smalltalk language, JUnit is the java flavor of the tool.

Every class implementing the List interface has to implement those methods, and the Java compiler will enforce this. This is only syntactical correctness, and an implementation class such as StrangeList would be accepted by the compiler:

```
class StrangeList implements List {
    public Object get(int position) {return null;}
    public void add(Object element) {// do nothing}
    public void remove(Object element) {// do nothing}
    public int size() {return 42;}
}
```

Test cases written using the JUnit framework would add constraints which specify the semantics of implementation classes. For instance, this would be a possible test case:

```
class ListTestCase1 extends TestCase {
    List toBeTested = .. ;
    public void test1() throws Exception {
        Object element = "test";
        toBeTested.add(element);
        boolean result = (1= = toBeTested.size());
        assertTrue(result);  // interface to test framework
    }
}
```

Writing these test cases is an iterative process: an initial set of test cases might not specify the intended model, but is updated frequently whenever new requirements (including reports about the malfunctioning of the software) emerge. This reflects the fact that the intended model is either not known or changes over time. Even in apparently simple cases such as the List example used here this might be the case. For instance, it is not straight forward whether the remove() method should remove only the object passed as parameter or also any other objects that equals the parameter. The later case could be expressed by the following test case added later to the initial set of test cases:

```
class ListTestCase1 extends TestCase {
    List toBeTested = .. ;
    public void test2() throws Exception {
        Clonable element = .. ; // some clonable object
        Object clone = element.clone();
        toBeTested.add(element);
        toBeTested.remove(clone);
        boolean result = (0= = toBeTested.size());
        assertTrue(result);  // interface to test framework
    }
}
```

This test case itself relies on the correct semantics of the clone() method which itself could be tested with test cases to make sure that the expressions element.clone()!=element and element.equals(element.clone()) always yield true.

Once a set of test cases is written, a testing framework such as JUnit can be employed to execute all tests and the next release of the software is only released if all tests succeed. This process can be automated (e.g., using ANT) - code is compiled, tested and released automatically. Going through this process various times usually quickly produces a good set of test cases representing the user requirements. There is a lot of anecdotal evidence that this test driven approach results in higher quality software and better conformance to user requirements, and the first empirical studies undertaking in this field support this assumption for small to mid sized projects [La04].

However, to our knowledge, there is no empirical study which proofs this and systematically compares it with the results obtained using methodologies such as the Rational Unified Process.

Test cases are not only useful to validate and but also to verify software. In software engineering, validation can be defined as "set of activities that ensure that the software that has been built is traceable to customer requirements" [Pr05] whereas verification refers to "activities that ensure that software correctly implements a specific function" [Pr05]. While the main application of test cases is validation, there is a strong verification component as well. For instance, special test cases can be written to verify database connections. All test methods in test cases are usually declared to throw exceptions. These exceptions often indicate programming errors, for instance null pointer exceptions caused by programming errors, not by lack of understanding of the user requirements. Testing frameworks catch those exceptions and mark test cases throwing exceptions as failed, therefore performing a verification function.

3. Model Refinement using Test Cases

In this chapter we adapt the idea of specifying models through test cases to rules. We set the scene by introducing the notion of the model.

A set of rules RB ("rule base") defines an inference operation $\vdash_{RB} \subseteq 2^L \times L$ for a formal language L as follows: a $FB \vdash_{RB} A$ iff there is a proof of A from FB (the "fact base") using the rules in RB. We do not have to specify the formal language at this point. However, we can assume that a certain expressiveness is required so that L can be used to represent business rules. In particular, predicate logic constructs (functions, predicates, constant, variable and complex terms), negation or even two different negations [HJW99] and deontic modalities [Wr51] should be part of the language used[3]. The actual inference algorithm used in the proof is also not important here, resolution [Ro65] or extensions of resolution supporting a more expressive language are examples. An inference operation $\vdash_{RB}$ is usually associated with a semantics $(M, \models)$ where M is a set of so-called models and $\models \subseteq M \times L$ is a relation between

---

[3] This ensures that the language is expressive enough to represent relationships and functional dependencies between business object, negative facts (.. it is not the case that) and missing information (if it cannot be shown that ..), as well as permissions and obligations.

models and formulas of the formal language. A relationship $m \models A$ can be read as "m is a model for A" or "A is valid in m". Possible models include classical models for propositional and predicate logic, multi-valued models, the well-founded semantics [GRS91] and the stable model semantics [GL88] for classes of extended logic programs and Kripke-style possible world semantics for modal logics [Kr63]. The $\models$ relation can be easily extended to a relation $\models$ $\subseteq M \times 2^L$, $m \models X$ for a set of formulas X means that $m \models A$ for all elements $A \in X$.

The model-theoretic definition of logic consequence goes back to Tarski [Ta36] and states that A follows from a set of formula X iff each model of X is also a model of A. We use $A \in Cn_{\models}(X)$ to denote logical consequence. For most logics, completeness and correctness theorems have been proven which show the equivalence of $Cn_{\models}$ and $\vdash_{RB}$.

$$A \in Cn_{\models}(X) \text{ iff } X \vdash_{RB} A$$

Therefore, every set of facts FB and set of rules RB constraints the set of models as follows:

$$M' = \{m \in M \mid FB \vdash_{RB} A \Rightarrow (m \models FB \Rightarrow m \models A)\} \text{ with } \Rightarrow \text{ representing the implication in the meta logic used}$$

This class of models determines what is valid and not valid, true and false. The complexity of real world rule bases itself combined with complexity of the rule engine used (i.e., the complexity of $\vdash_{RB}$) can make it extremely difficult for users to understand the association between rules and facts (RB and FB) on the one hand and the model that defines valid consequences (M') on the other hand. In the rest of this chapter we want to use test cases for an alternative description of this model. This description does not use rules, and it is significantly easier to understand for end users. Having an alternative formal and executable description of the intended model has an obvious advantage: it can be used to validate the rules, i.e. to check the consistency between the model specified by test cases and the model specified by the rules. While there is not guaranty that customer requirements are translated correctly into test cases we claim that it is much more likely to translate requirements correctly into test cases than to translate them directly into rules. Test cases are inherently simple, atomic assumptions compared to rules where the end user capturing the requirements has to deal with complex matters such as rule chaining, (nested) complex terms, long lists of prerequisites, rule priorities and conflict resolution and ambiguities in the rule language (e.g., if two negations with a different semantics are supported). In most cases, test cases use only queries with ground terms and are therefore even variable free. Therefore, validation here means validation against a formal representation of the requirements that is very likely to express the requirements correctly. While test cases do not always represent requirements completely, we argue that there is a simple and tool supported process that can be used to approach completeness.

To adapt the idea of test driven development we use the following example. The rule base is built using the following unary predicates:

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"

The rules work on top of a fact base that contains facts such as V(a), E(b) etc, the letters a,b,c denote constant terms, while letters x,y,z denote variable terms. We do not use function symbols in this example. Assume that a fixed set of facts is given as follows:

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

The expert would usually start with an initial *partial* model (expressed as statements such as „good loyal customers should qualify for the special discount"):

MOD1 = {G(a),E(b),G(c),V(c),D(a),D(c)},{}

Partial means that this is not a complete model in the mathematical sense. A partial model is defined as a pair (P,N) with P⊆L and N⊆L. P contains assertions which should be valid while N contains assertions which should not be valid. This defines a class of models as follows:

M(P,N) = {m∈M | (m ⊨ A for each A∈P) and (not m ⊨ A for each A∈N) }

Another way to think of partial models is that they are constraints for models. In our example, the partial model can be described with the following initial set of test cases:

TC1 {?D(a) ⇒ true, ?D(c) ⇒ true}

A test case "?D(a) ⇒ true" should be read as follows "querying the rule engine with D(a) should yield true". Writing this test cases does not necessarily mean writing code in a programming language, rule markup or any other formal language. In our example, it could be done by using simple graphical user interfaces which would allow the end user to build a query and to attach a boolean value to the query indicating the expected outcome for this query. What is important is that the test case is executable: the query can be issued, and the result can be evaluated.

The next step is to add rules to support the intended model. Although this is straight forward in our example, it is a creative and potentially difficult process. However, the test cases can be used to validate the correctness of the rules automatically and therefore safeguard this step.

RB1 = {G(x) → D(x)}

The test cases are much simpler than the rule: they contain only one predicate symbol, only constant terms, and no logical connectives. In particular the lack of variables means that the test cases are less abstract than the general purpose rules and therefore easier to comprehend for the end user.

A refinement of this rule base is needed when the user realizes that the model differs from the intended model – employees of the company should qualify for the discount as well. This yields model 2. Modifications made in this step are highlighted in boldface.

MOD2 = {G(a),E(b),G(c),V(c),D(a),D(c),**D(b)**},{}

This can be translated into a new test case as follows:

TC2 {?D(a) ⇒ true, ?D(c) ⇒ true, **?D(b)⇒true**}

Then the rule base is updated until all test cases succeed. In this case, there is only one simple way to do this:

RB2 = {G(x) → D(x), **E(x) → D(x)**}

In the final step, it is realised that customers using a voucher that has a discount attached to it would qualify for the discount as well. This double discount would make the sale unprofitable and therefore the D(c) has to be removed from the intended model. This is done in the final refinement step:

MOD3 = {G(a),E(b),G(c),V(c),D(a),D(b) },{**D(c)**}
TC3 {?D(a) ⇒ true, **?D(c) ⇒ false**, ?D(b)⇒true}
RB3 = {G(x),¬**V(x)** → D(x), E(x) → D(x)}

The following figure depicts the model refinement through test cases in the example given above. Test cases are represented by grey squares, a plus symbol indicates a positive test case (an assertion that should be valid), a minus symbol indicates a negative test case (an assertion that should not be valid), respectively.

.

Figure 1 Refining the model using test cases.

There is no upfront modelling, the model is specified iteratively. Each cycle consists of fours steps: updating test cases, modifying the rules to meet the new requirements, executing the test cases and therefore validating the rules, and finally the release of the rule base. A new iteration is triggered either by new requirements or feedback indicating that the current rule base (and therefore the current set of test cases) does not describe the current requirements correctly. As in the case of test driven software development test cases can also be used to *verify* the correctness of the rules. For instance, if a test case executes a certain query based on a set of

rules loaded from a database, then this test case does not only test that the inference engine computes the expected result for this query (validation), but also that the database settings are correct, the rules can be loaded and that there are sufficient resources available to execute the query (verification).

Figure 2 The Lifecycle of a rule based in the test–driven approach

In the last example we have made two assumptions: that the test cases always consist of a fully instantiated ("ground") query which can be answered with true or false, and that there is a fixed fact base. In general, we have to cater for test cases for which neither of these restrictions applies. The following examples are to support this claim:

1. Testing queries with a certain number of variables, and counting the results. This might be useful to check that certain assumptions should apply to exactly one set of variable substations or no set of substitutions at all. Example: query for the employee of the year (there should be exactly one). This tests themselves can express certain business rules such as "there should be exactly one employee of the year" and "there should be no customer who is less than 18 years old".
2. Testing for a certain result order. When querying a rule base with a query containing variables, usually multiple results will be returned. The application might associate a certain meaning with this order, for instance if the client application uses the first result (and only the first result) in a function.
3. The nature of the test itself might depend on the expressiveness of the language used.

The most general definition of a test case which accommodates all of these cases is therefore very informal: test cases are constraints on models which are simple and executable.

The fact base itself is very often variable. To express a test case about a customer who should not get a certain discount, certain base facts about this customer have to be present in the fact base. The fact base used in tests is usually not the production fact base (which is normally just a view on a database or other enterprise storage system) but a special test fact base. If the execution of the test case can change the fact base (e.g., some rule management systems

support so-called procedural attachments), then fact bases associated with test cases should be isolated and every test case usually has to have its own fact base. Therefore, the execution of a test case starts with setting up a temporary fact base. Following the execution of a test case, resources associated with this fact base should be released. This test case lifecycle corresponds to how test cases work in object oriented programming – setting up the test case (e.g., the setup() method in JUnit), performing the test method(s) and finally releasing resources (e.g., the teardown() method in JUnit).

Up to now we have used test cases to validate the correctness of an intended model defined by its rule sets. A business rule set is considered valid if it correctly solves all test cases, and there is evidence that it will correctly solve any further added test case. However, while this is a required property of a high-quality business rule set, it is rarely a sufficient one (except in very small business domains). In most cases business rule engines in particular with very expressive formal rule languages are facing high computational tasks, where the run-time efficiency is essential for the acceptance. Here another validation dimension must be added, namely the measuring of the maximum efficiency of the computation process. It is essential to equip a validation tool with an automated procedure, which can recognize problematic relations between rules. Test cases can be used to measure the runtime performance for different outcomes of a rule set given a certain test fact base as input. By this certain points of attention, e.g., test case with long computations, loops or deeply nested derivation trees, can be identified and a refactoring of the rule base optimizing the rule sequences (e.g. by reordering rules, narrowing rules, deleting rules etc. – see next chapter "Refactoring Rule Bases") according to these attention points can be attempted in order to reach the maximum efficiency of the computation process. We call this dynamic validation in opposite to functional validation of business rules. Dynamic validation can be best applied after the rule base has undergone the functional testing and all detected anomalies have been removed from the knowledge base. Dynamic test cases with maximum time values (~ time constraints) for answering certain queries can be defined to assure that a business rule set solves its intended tasks in an acceptable time frame, i.e. ,that with any further added business rule the computational time stays below this limit. If the test case fails it becomes an attention point for refactoring the rule base. A dynamic test case might look like this:

TC4 {?D(a) < 1000 ms}

stating that the query "?D(a)" on the rule engine should succeed in less than 1000 milliseconds. To do dynamic validation with test cases we have one prerequisite: That the tested fact base and the used test cases must be an adequate benchmark for the later run-time environment and that we can predict how well a rule set would do on a different compiler, computer or data set. Although this might be unsatisfactory because dynamic test cases are therefore very domain specific or because we do not want to give a rule set a special order we argue that in many applications where business rules are applied we need at least a clue if our rule set is able to solve its purpose in an adequate time and we need methods to do this benchmarking already in the development phase of business rules. Test cases constraining the computation time might help to carry out this estimation. Further, when the business rule set went into action we can adjust the upper computational limits to the particular application domain which must not be exceeded even if new rules are added to model to assure that the rule set fits into the application.

Another interesting question is how the approximation of the intended model by test cases can be measured. In software engineering test coverage metrics have been studied for years [WGM85] and tools like Clover [Cl] measuring test coverage have become very popular in recent years. The representations of models as sets seems to indicate that a pseudo metric (the symmetric difference between sets) can be used to measure the distance to the intended model. This is made difficult by the fact that the intended model is usually unknown. What is more, it changes over time as it reflects a changing environment. We could still try to measure how the sequence of approximations (test cases) stabilised, similar to approximation of real numbers by Cauchy sequences of rational numbers. We could for instance measure the ratio between the number of modified (new, updated, deleted) test cases and the number of unchanged test cases and interpret low values as an indicator for the stability of the model. A detailed analysis of possible metrics is an area of further research.

If the iterative, test driven approach to develop rules is used test cases will become an important asset for the organisation. As test cases and rules should be consistent, both should be stored and versioned together. In object-oriented software engineering, test cases and code are usually managed in the same source code repository and separated only when the software is deployed into the production environment. New technologies like the meta data APIs for Java will probably lead to an even closer integration of program code and test cases. To achieve such a high level of integration between tests and rules, existing formats such as RuleML [BTW01] used to make rule bases persistent and exchange rules would have to be extended in order to accommodate test cases as well.

4. Refactoring Rule Bases

The existence of test cases only safeguards the authoring of rules, but the task of editing the rules remains challenging. The question is how this process can be supported by tools. Again, there is a counterpart in agile software engineering where refactoring browsers are used to support changing the software until the test cases succeed. Those refactorings are formalised, catalogued [Fo99] and supported by tools such as Eclipse [Ec] or RefactorIt [Re]. Refactorings in software engineering are program transformations to improve the overall quality of software (in particular source code). They address problems called code smells. Smells represent structures in source code that can be improved by the application of refactoring. The definition of smells is generally very informal, although for many smells a more formal definition can be given (for instance, by means of [anti-] patterns or metrics). We give three examples of smells which represent defects in rule bases which can be addressed by refactoring these rules.

Redundancy – There are redundant rules or rule fragments (for instance, shared subsets of prerequisites). This may result in inconsistencies, poor runtime performance of the rule engine and makes the rule base more difficult to manage.

Inconsistency – Different, inconsistent results are supported by the same rule set.

Incompleteness – Certain queries can not be answered by a rule set.

The concepts of inconsistency and incompleteness are not the concepts used in formal logic. For instance, a user might consider a rule base as inconsistent if it can compute two different variable substitutions x/aDiscount1 and x/aDiscount2 for a query such as discount(aCustomer,?x). This rule base would still be consistent according to the strict logical definition where inconsistency is defined as $X \vdash_{RB} A$ for all formulas $A \in L$ [4].

For each smell there are refactorings which can be used to address this problem. We do not claim here that a simple application of a refactoring always solves the problem; very often a series of refactorings or a combination of various refactorings is required.

We use a simple template to describe the refactoring. It consists of the name and the description of the refactoring, a representation of the rule base (or a rule base fragment) before and after the refactoring has been performed, and the smells that can be addressed by applying the respective refactoring.

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

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

Name:	Introducing a Default Rule
Description:	There are gaps in the rule set, i.e. there is no result for certain queries. Example: "all customers get a discount of 0%". A default rule is introduced to address this problem. If priorities are supported, the default rules have the lowest priority. Often default rules do not have prerequisites and are actually facts. If the

---

[4] This is a generalisation of the classical definition of inconsistency using negation. In most logics with negation the following is true: $\{A, \neg A\} \vdash B$ for an arbitrary formula B. These logics are also called *explosive*.

	language supports predicates, the conclusion of the default rule has a predicate that already occurs in other rule heads in the rule base.
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

We use the term narrowing for the second example since the refactored rule set will result in deeper but narrower derivation trees with more rule chaining. Tools supporting narrowing could prompt the user to name the newly defined A (usually the predicate), and this might help the end user to define new business concepts. For instance, if narrowing is applied to rules with the shared set of prerequisites

"turnover of customer last year > 1000 $ and customer has company Visa Card"

then narrowing could result into introducing a new business concepts such as "Platinum Customer".

There might be some refactorings which depend on the rule engine used. For instance, some rule engines might want to remove disjunction in the prerequisite list (replacing $A \lor B \rightarrow C$ by two new rules $A \rightarrow C$ and $B \rightarrow C$), remove function symbols from rule heads etc. However, all refactorings should have the following properties in common:

Integrity – is defined by the test cases which succeed. All test cases which succeeded before applying the refactoring still succeed after the refactoring has been completed. If TC_{before} and TC_{after} denote the set of test cases which succeeded before/after the refactoring, we have: $TC_{before} \subseteq TC_{after}$.

Refactorings such as narrowing optimise the internal structure of the rule base, and support the conceptualisation of the problem domain. These refactorings do usually not change the set of supported test cases and we have $TC_{before} = TC_{after}$. On the other hand, refactorings such as exception to the rule do address new requirements, and therefore we can expect $TC_{before} \subset TC_{after}$. There are refactoring which might change the test cases as well. In particular, renamings (for instance, renaming predicates) fall into this category.

A tool environment supporting refactorings should furthermore support the following two features:

Transactionality – refactoring runs as a transaction. If it fails, the rule base is reversed to its old status ("rollback"), and only versions satisfying the integrity condition are made persistent ("commit").

Support for Auditing – information about the refactoring is attached to the respective rules (as meta information) and can be used for auditing purposes.

5. Example: Implementing Test based validation in Mandarax

In this chapter we present a prototype[5] which shows how the test case based validation can be integrated into a business rule management system. The system used is the open source mandarax system [Ma][Di03][Di04]. Mandarax is completely implemented in Java and uses Java classes and primitives as type system. Rules are interpreted by the mandarax inference engine which uses a modified version of the resolution algorithm [Ro65]. Rules and facts are managed in container objects called knowledge bases along with queries and references to external sources of rules and facts called clause sets. This supports a smooth integration of facts and rules managed by external systems (databases, ERP systems, systems accesses using web services) and avoids replication –references are resolved at query time. Mandarax supports persistent knowledge bases; XML is used for this purpose. This includes support for the emerging RuleML standard [BTW01], [Ru].

The test framework used to add validation support to mandarax makes use of the de facto standard library for testing java code, JUnit by K. Beck and E. Gamma [GB04]. The main test class subclasses the JUnit class TestCase and adds references to the rule base to be tested (the "kb" attribute), the fact base (the "assumptions" attribute) and the inference engine used in the test (the "ie" attribute).

---

[5] The source code can be downloaded from the public CVS server cvs.sourceforge.net, project name: mandarax, module name: mandarax.
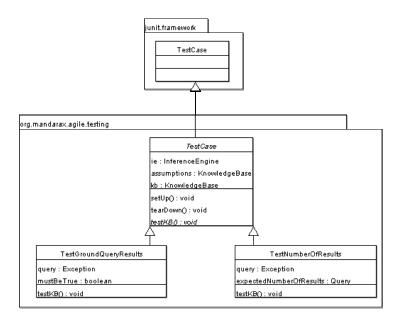
Figure 3 UML Class Diagram for the Mandarax Test Integration Layer

The abstract test case class implements the lifecycle methods – facts are temporary added to / removed from the knowledge base. Removing assumptions is a potentially dangerous operation as facts, rules and other objects will be removed if they are equal to (not identical to) the parameter passed in the remove method. This problem can be circumvented by using a special factory[6] to create facts which are then automatically flagged as facts for test cases.

```
// Sets up the test case.
public void setUp() throws Exception {
    super.setUp();
    for (int i=0;i<assumptions.length;i++) this.kb.add(assumptions[i]);
}
// Releases the test case.
public void tearDown() throws Exception {
    for (int i=0;i<assumptions.length;i++) this.kb.remove(assumptions[i]);
    super.tearDown();
}
```

The constructor informs the test framework that the actual test method is "testKB", the JUnit framework uses this information to invoke the respective method at runtime using reflection. It is the responsibility of the subclasses to implement this method.

```
// testKB implementation from TestGroundQueryResults
public void testKB() throws Exception {
```

_____

[6] Mandarax uses the AbstractFactory design pattern [Ga94] to create facts and rules and to separate implementation and specification layer.

```
                // issue query
                ResultSet rs =
ie.query(query,kb,InferenceEngine.ALL,InferenceEngine.BUBBLE_EXCEPTIONS);
                // result ok if result set is not empty (Mandarax uses a JDBC like API)
                boolean result = mustBeTrue==rs.next();
                // release resources
                rs.close();
                // call method inherited from junit.framework.TestCase
                assertTrue(result);
        }
```

6. Conclusion and Related Work

In this paper, we have tried to establish a link between two flourishing areas in modern software engineering: business rule management systems on the one hand and test driven software engineering on the other hand. We have shown how practises used in test driven software engineering can be used to facilitate the application of business rules, and to add support for automated validation and verification to rules. We have related the main concepts from test driven software engineering such as test case and refactoring to concepts known in formal logic. Most of the considerations did not depend on a particular logic (language, model theory or proof theory) but referred to logic in a very general sense in the spirit of Tarski.

By using test cases to validate and verify rules we obtain the following advantages:
- Users without not trained in formal logic can be guided by test cases.
- The maintenance and development of large, complex and distributed rule sets is safeguarded.
- Error analysis is facilitated, and fixing errors is supported through versioning of rule sets consistent with test cases.
- Test cases allow analysing the quality of rule sets.
- The performance of rule sets increase through the application of refactorings improving the quality of the rules.

There are numerous points we have left open. In particular, the lists of refactorings, smells are still very small and only serve as examples.

We are aware of a couple of related approaches. LIBRT [Li] has a range of products to support the implementation of business rules. The VALENS validation component supports validation with test cases. According to the web page, support to generate test cases from rules is currently (February 2005) under development. This product is design to work with other BRMSs, the integration point is an XML based representation of rules based on the schemas LIBRT publishes.

There is a strong connection between refactoring of rule bases and logic program transformations. Transformations are defined as transformations preserving the semantic value [PP96] of the (logic) program. We have a weaker requirement for refactorings – the transformed rule base does not necessarily have the same model (for instance, Herbrand model) as the original rule base but the same or a better approximation of the model expressed by the

test cases. For many combinations or rule engines, languages and refactorings results obtained in the field of logic program transformations can be adapted and used within our framework. For instance, the refactoring called narrowing is similar to the "Predicate Tupling Strategy" presented in [PP96].

We are only aware of one attempt to add support for test driven development to Prolog, a project called Test Driven Development for Prolog [Td].

References

[Ba04]   Baisley, D.: A Metamodel for Business Vocabulary and Rules: Object-Oriented Meets Fact-Oriented. *Business Rules Journal*, Vol. 5, No. 7 (July 2004).
           URL: http://www.BRCommunity.com/a2004/b197.html
[Be99]   Beck, K: Extreme Programming Explained: Embracing Change. Reading, MA, Addison-Wesley, 1999.
[BG04]   Beck, K,; Gamma, E.: The JUnit Cookbook.
           http://junit.sourceforge.net/doc/cookbook/cookbook.htm
[BTW01]   Boley, H.; Tabet, S.; Wagner, G.: Design Rationale of RuleML: A Markup Language for Semantic Web Rules, Proc. SWWS'01, Stanford, 2001.
[Cl]   Clover Product Homepage. http://www.cenqua.com/clover/
[Di03]   Dietrich, J.; Kozlenkov, A.; Schroeder, M.; Wagner,G. : Rule-Based Agents for the Semantic Web, Journal on Electronic Commerce Research Applications, 2003.
[Di04]   Dietrich, J.: A Rule-Based System for eCommerce Applications. Proceedings KES 2004, LNAI 3213 p 455. Springer, 2004.
[Ec]   The Eclipse Project. http://www.eclipse.org
[Fo99]   Fowler, M.: Refactoring, Improving the design of existing code, Reading, MA, Addison-Wesley, 1999.
[Ga94]   Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns: Elements of Reusable Object Oriented Software. Addison-Wesley, 1994.
[GRS91]   van Gelder, A.; Ross, K.; Schlipf, J.: The well-founded semantics for general logic programs. JACM, vol. 38(3):620-650, 1991.
[GL88]   Gelfond, M.; Lifschitz, V.: The stable model semantics for logic programming. in Kowalski,R.; Bowen,K.A. (ed.): 5th International Conference on Logic Programming, Proceedings, pp. 1070-1080, MIT Press, 1988.
[HW03]   Halpin, T.; Wagner,G.: Modeling Reactive Behavior in ORM. In: Conceptual Modeling - ER 2003, 22nd International Conference on Conceptual Modeling, Chicago, IL, USA, October 13-16, 2003, Proceedings, Springer LNCS 2813, 2003.
[HJW99]   Herre, H., Jaspars, J., Wagner, G.: Partial Logics with Two Kinds of Negation as a Foundation for Knowledge-Based Reasoning, in D.M. Gabbay and H. Wansing (Eds.), What is Negation ?, Kluwer Academic Publishers, 1999.
[Kr63]   Kripke, S. A.: Semantical analysis of modal logic. I. Normal modal propositional calculi. Zeitschrift fur Mathematische Logik und Grundlagen der Mathematik, 9:67—96, 1963.
[La04]   Layman, L.: "Empirical Investigation of the Impact of Extreme Programming Practices on Software Projects," Companion to the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04), pp. 328-329, 2004.
[Li]   LIBRT web site. http://www.librt.com/

[Ma]   The Mandarax Project. http://www.mandarax.org.
[Ma02]   Martin, R.: Agile Software Development, Principles, Patterns, and Practises. Prentice
          Hall, 2002.
[Pr05]   Pressman, R.S.: Software Engineering. 6th International Edition, McGraw-Hill, Boston,
          2005.
[PP96]   Pettorossi, A.; Proietti, M: Rules and Strategies for Transforming Functional and
          Logic Programs. ACM Computing Surveys, 28 (2):360-414, 1996.
[Re]   RefactorIt web site. http://www.refactorit.com.
[Ru]   The RuleML project. http://www.ruleml.org.
[Ro65]   Robinson, J. A.: A Machine-Oriented Logic Based on the Resolution Principle." J.
          Assoc. Comput. Mach. 12, 23-41, 1965.
[Ta36]   Tarski, A., Ueber den Begriff der logischen Folgerung, Actes du congrès international
          de philosophie scientifique, 1936. In English: On the concept of logical consequence,
          in: Logic, Semantics, Metamathematics. Papers from 1923 to 1938 by Alfred Tarski,
          Oxford: Clarendon Press, 1956.
[Td]   TDD for Prolog. Project Web site.
          http://courses.cs.vt.edu/~cs3304/Spring03/tddplg.php
[WGM85]   Weiser, M.D.; Gannon, J.D.; McMullin. P.R.: "Comparison of Structural Test
          Coverage Metrics", IEEE Software, Vol 2, No 2, pp 80-85, 1985.
[Wi]   Wikipedia Definition of Business Rule. http://en.wikipedia.org/wiki/Business_rules.
[Wr51]   von Wright, G.H.: Deontic Logic, Mind, N.S., 60, 1-50, 1951.