Exploratory Modeling with SAP NetWeaver®

Andreas Tönne, Georg Heeg eK

WHITE PAPER

In-depth Analysis and Review

SIMPLIFICATION THROUGH INNOVATION®

Exploratory Modeling with SAP NetWeaver

Andreas Tönne, Georg Heeg eK

Table of Contents

Customization of Standard Software 1

Exploratory Modeling 2

Exploratory Modeling in Practice
with SAP NetWeaver 4

Duplicate-Analyzer – A Case Study 4

Summary 9

WHITE PAPER

In-depth Analysis and Review

About the Author

Dipl.-Inform. Andreas Tönne is general manager for projects, product development and operations at the Dortmund offices.

About Georg Heeg eK

Georg Heeg eK is the leading company for product development, consulting and training for Smalltalk™ in Germany, Switzerland and Austria. Georg Heeg eK and Cincom Systems have a close partnership in the development of new technologies and tools for Cincom Smalltalk.

Offices are located in Dortmund, Köthen (Anhalt) and Zürich.

Georg Heeg eK
www.heeg.de

A German-language article based on this white paper has been published in *OBJEKTspektrum* issue 3/07 (www.objektspektrum.de).

Exploratory Modeling with SAP NetWeaver

This document presents an introduction to "exploratory modeling," a process that supports agile software development. Exploratory modeling combines – in a suitable programming environment – the virtues of agile software development with an executable and thus immediately verifiable model. Later in this document we will show an example of exploratory modeling of heuristics for finding duplicate invoices in SAP NetWeaver with the help of Cincom Smalltalk.

Customization of Standard Software

Accurate reflection of a company's business processes is one of the main success factors for the implementation of standard software applications. Especially critical are those business processes that apply to a company's goals and unique characteristics. When its standard software does not satisfy its critical business processes, a company faces two alternatives: The company can adapt its business processes to the capabilities of the standard software or, conversely, enhance the standard software to meet its needs. Adapting business processes comes with high risks of failure. The costs for training employees as well as process errors caused by continuing past practices or misunderstanding new processes can be considerable. Careless implementation of standard software in the course of the year 2000 (Y2K phenomenon) that occurred without true business strategy illustrates these risks and costs. Adapting standard software to support a company's unique processes is thus the preferable alternative, one that all standard software vendors advocate.

SAP NetWeaver offers a clear strategy for the architecture and implementation of such enhancements. This strategy is called Enterprise Service-Oriented Architecture (E-SOA). It is based on open standards and innovative software technologies like web services. The challenge lies in establishing a precise understanding of what the critical business processes are. The company's relevant departments must establish and verify this understanding through mutual consensus.

Agile Software Processes

Innovative software technology and service-oriented architecture are vital components of excellence and risk management. But they require innovative software development processes for their realization. "Agility" is the general term for such innovative software processes that are based on a small set of problem-oriented principles; over the past few years, these agile processes have been at the forefront of software development. The central principles of agility are:

- Simplicity
- Innovation
- Communication
- Customer orientation

The key virtue of agility is the shift of project and progress control as well as risk management of software development from formalistic rules toward the project team and its intrinsic social dynamics. Control and risk management depend on communication, continuous testing and personal responsibility within the team and toward the company. In contrast, traditional "waterfall-type" software processes promise pseudo-security. They build an illusion of control and manageability through excessive formal control and documentation of all aspects of software development. However, these formalistic software processes cannot avoid the fundamental errors arising from lack of knowledge and communication clashes; they can only document these errors in detail. Compare those formalistic processes to agile development, which produces continuous proof of progress that is immediately verifiable by the developers and experts.

- Formalistic, bureaucratic and linear software processes are replaced by lightweight, iterative design and implementation cycles.
- Document-heavy, separate design and development stages are replaced by integrated cycles in which the understanding and continuous verification of current progress is the primary goal.
- Each cycle yields a working application that displays the current feature set.

Exploratory Modeling

Agile software processes have proven to be very successful for designing and implementing modern, innovative applications. They are the sum of values and principles that were developed naturally from object-oriented programming. Agility and OOP both share a paramount emphasis not on the technical act of implementation ("How do I get the machine to do what I want it to?"), but on thinking about problems and concepts ("What am I really talking about?").

The Tower of Babel

Concepts are of special importance. Capturing and modeling concepts correctly is the foundation of software development. By modeling we mean the critical phase of a software design, where the domain language gap between experts and developers is bridged. This is the phase where the most expensive mistakes of the entire software development process can be made. Exploratory modeling was developed to advance this process of finding and verifying concepts.

Modeling is complicated not just by differing domain languages and domain knowledge. The different levels of conceptual formalism and abstraction that developers and experts use are also a source for fundamental misunderstanding. Developers try to formalize concepts as quickly as possible and classify them in schemata of abstraction and generalization. Experts, on the other hand, use different strategies for describing concepts; they are used to thinking about them through examples, analogies and domain-specific classifications.

It's amazing that many agile and traditional software development processes try to bridge this language gap by introducing yet another technical (formal) language: UML (Unified Modeling Language). UML is a visual modeling language that satisfies the developer's needs for formalism and standardization of the model description. UML is thus handy for design documentation and implementation as well as for communication among developers. But it is not helpful for building up an understanding of model concepts on the expert side, a lack of utility that violates the agile principles of customer orientation and communication.

Experts thus continue to face the problem of understanding requirements and concepts from the formal, abstract description that the developers present to them. For many domain experts, this is a difficult task and leads to overlooked model errors that don't surface until late in the implementation cycles. The experts, with their view of the domain, might not have adequate conceptual and formal abstraction to verify the UML model.

Verifiable Models Instead of Diagrams

Our concept of exploratory modeling starts with this dilemma. We replace such abstract, formal descriptions of a model by executable models, whose properties can be grasped and verified immediately by both the expert and the developer. Model implementation involves less the drawing of detailed UML diagrams than the running of agile modeling cycles: The model is implemented in a suitable programming language and is verified, extended and further developed together with the domain experts.

This process of experimental implementation, verification and updating of the documentation is called "exploratory." An interesting introduction to modeling by implementation is found in "Domain-Driven Design".[1] The key concepts of exploratory modeling are:

- Domain terms are as close to spoken usage as possible.
- No technical distinction is made among data, entities and intangible concepts like services. Everything is represented in a uniform fashion.
- Process modeling and colloquial descriptions of use cases appear directly in the program's workflow.
- The simplest possible implementation that suffices is chosen to illustrate the processes and workflow.
- Pure model implementation is extended through discrete technical elements that facilitate experiments with the model (e.g., GUI, interfaces to existing systems and mockups).
- A continuous feedback loop runs between experiments and the current model documentation.

The last point especially is very simple in exploratory modeling, because the model implementation uses the same concepts and processes as the model documentation. Exploratory modeling does not need a technical translation between colloquial concepts and their implementation artifacts.

Exploratory modeling is one step in a software development process, but does not presuppose a particular process. It can be embedded within any existing agile process, but can also be used in non-agile "waterfall-type" processes, where it will produce greatly enhanced results by improving the depth and quality of the foundation underlying the development. We do not see it as a contradiction to reject UML for the development and communication of models, while embracing it at the same time as the standard for the documentation of models and designs between developers.

Exploratory modeling has its greatest effect at the beginning of a project, when the domain is unknown and the uncertainty is greatest. But later iterations can also benefit from the insertion of exploratory modeling phases. This is particularly true when an iteration introduces substantial new concepts or model changes. At such stages, exploratory modeling helps to confirm that the changes are well understood and accepted by the experts. To achieve this, exploratory modeling is rooted in experiments with the models. This is the best way to build a consensus between expert and developer that the model meets the needs of both parties. Such an experimental model without experiment is pointless; the experimental model must be integrated into a test context. The model is not limited to a formal representation of the expert concepts, but must be an executable program, too.

Agile Modeling Language

Exploratory modeling intentionally treats the modeling language as identical to the implementation language. Not every programming language is suitable for expressing models in the desired way; the language must support the agile principles to allow more than just prototyping. Agility for programming languages means:

- Non-technical, barrier-free
- Interactive
- Meta-programmable
- Concept-oriented (pure OO)

A non-technical language that is tailored toward the representation of concepts allows a high degree of similarity between model and program. The expert can "see" his concepts in both the model and the implementation. This allows him to conduct experiments with the implementation to verify the model.

Interactivity of the programming environment is crucial to the acceptance of exploratory modeling. Extensive delays in adapting a model because of time-consuming compile-link-test cycles interrupt the communication flow between expert and developer and also slow down the agility of the experiments.

Meta-programmable languages allow enhancement of the programming environment by new concepts, which are the foundation of expert concepts, rather than forcing the developer to translate the expert concepts to implementation artifacts. It is important to keep the "distance" between expert concepts and experimental implementation as small as possible.

The current zoo of programming languages offers a small number of languages that are suitable for exploratory modeling and have the above qualities. Common to these languages is their dynamic type system, powerful meta-programming concepts in a pure OO language and a matching programming environment. Smalltalk is closest to this ideal of an exploratory modeling language, and Ruby takes second place.

Smalltalk is ideal for exploratory modeling since it has the fewest technical barriers of all current programming languages and offers extraordinarily short development cycles. An example of such minor technical barriers is the simplicity of declarations. Smalltalk is also based on a small number of very powerful meta-constructs that allow the definition of Smalltalk itself, along with all conceivable expert concepts by means of Smalltalk. Edit-compile-link cycles are unknown in Smalltalk. In their place, one writes programs incrementally by adding empty classes and methods to classes one by one. At any time, the resulting (partial) program is executable. This process is extremely helpful for interactive modeling between developer and expert.

Exploratory Modeling in Practice with SAP NetWeaver

The key concept for exploratory modeling is experimental model implementation. Experts need examples and tangible demonstration. No model can satisfy these needs if it cannot be experienced and tried out in a live test environment.

It is important to note that these experiments do not exist in a vacuum. The models typically are a mix of new concepts, used in relation to existing, known concepts in the business environment. If we want to conduct experiments with such models, then we have to include the referenced environment in our experiment. The experimental implementation of the model can be expanded with a test environment via mockups, interfaces or facades that represent the business environment in which the experiment should run.

Experiments in SAP NetWeaver create this business environment through "Remote Function Call" (RFC) modules or web services. The programming environment must support the import and integration of these services in a simple and lightweight fashion. Cincom Smalltalk, which was used by SAP for exploratory modeling projects, has a dedicated integration concept for SAP NetWeaver that supports all agile requirements.

The integration of experimental implementation and test environment is typically symmetrical. Not only do we need to use the business environment to make our models executable, we must also integrate the experimental implementation itself into a test environment in order to run, monitor and analyze experiments with the model. This test environment might contain special GUIs, data export and import interfaces or tools for analysis. It is even possible (and has been tried) to integrate the experimental implementation in a life application and experience its behavior in the target environment. For SAP NetWeaver, this means publishing the model via RFC or web services.

Duplicate-Analyzer – A Case Study

The "duplicate-analyzer" illustrates the kind of rapid, high-quality modeling you can achieve with exploratory modeling. This example is based on a project that was performed together with the "Business Process Renovation" team at SAP Labs; the modeling language was Cincom Smalltalk. The task was to find new concepts for an existing albeit not very satisfactory solution, and to prove that these new concepts were indeed better. The modeling steps that we report here were performed over seven days by a team of two SAP developers and two consultants from Georg Heeg eK.

The job of a duplicate-analyzer is to find duplicate invoices in the financial transactions of the FI system of SAP. This task is divided between a heuristic to find potential duplicates (cases) and a business process to deal with these cases through an accounts payable clerk. The part that we created with exploratory modeling was the case builder, the heart of the heuristics, which would build cases based on similarity criteria. The existing solution did the job, but its results were not entirely satisfactory. The chosen model was impacted by the premature decision to use an existing component for similarity computations, the free-text search module TREX. This limited possible heuristics to the abilities of TREX. The desirable flexibility of the heuristics was sacrificed to an implementation decision made during model finding.

The key questions that came up early in the project were of course: How do duplicate invoices come about, and how do we recognize them? If we have answers to these questions, then we are done. How to recognize duplicate invoices is an unspecific concept that is ideal for applying exploratory modeling. The accounts payable clerk finds it simple to identify duplicates when presented with two suspect invoices: He uses his practical experience and knowledge of the company's processes (and typical errors) to make his decision. The money paid twice can be re-claimed, which is the business incentive to search for duplicate invoices in the first place.

Iteration 1

However, as simple as this sounds, it turned out to be difficult to understand how the clerk actually uses his experience and knowledge. So we started out simple and asked the key question for the first iteration: How do we classify suspiciously similar invoices that could form a duplicate case? Exploratory modeling tells you to write down what you know and create a first model implementation. We quickly identified the interfaces for data import and export that the clerk was not taking into account, but that are needed for experimentation. The resulting model implementation was rather blunt and considered each pair of invoices as similar. Nevertheless, we established a basis for creating cases and running experiments.

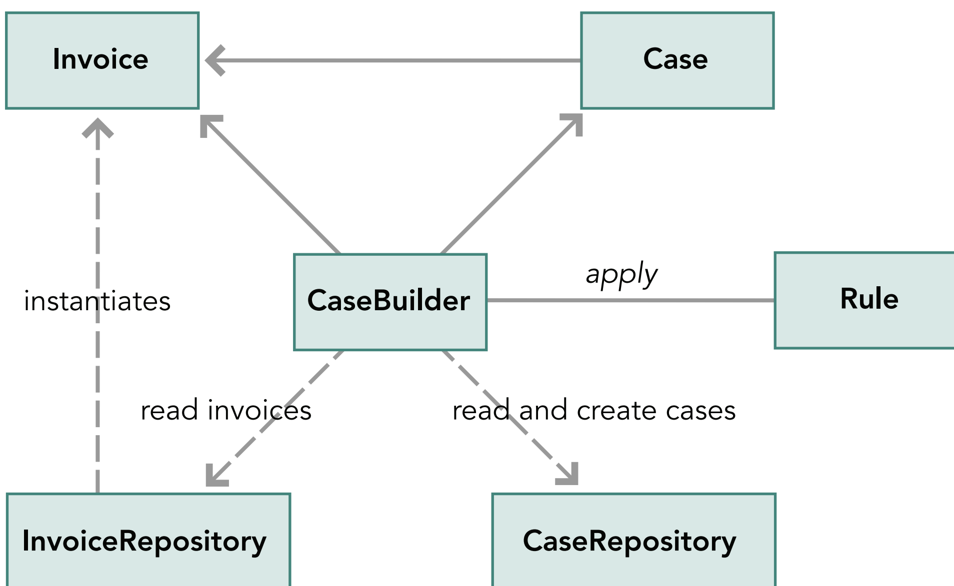


Figure 1

The invoices of this model are facades of the real FI system data that is received through the invoice repository interface. The attributes of invoices that we used in the model are reference key, invoice date, name of the account and amount. The definition of similarity was modeled in the rule with respect to these attributes. This was the basis for the first experiments.

We extended this model with a few obvious rules that were comparable to the existing application of TREX. One rule, for example, looked for typographical errors in names and percentage deviation in amounts. The first experiment was run with real financial data of 120,000 invoices.

Over the course of four days, we created a running model and experimentation environment that covered our knowledge of invoice duplicates and that produced results that were comparable to the output of the existing application. A full import of the 120,000 invoices took roughly an hour and produced what we had expected: far too many duplicate cases that proved to be false positives. Obviously, the rules for similarities and the heuristics of the case-building were not robust enough. After all, this was the reason for redoing the existing application. But the experiments and in particular the analysis of the cases found showed us that we had failed to consider the practical reasons that duplicates occur.

Iteration 2

This insight into the failure of our simple heuristics started an extended discussion of alternative rules and causes for duplicates. Invoices can occur twice if they are sent multiple times, e.g., via both e-mail and postal delivery; or if different clerks input variable data, e.g., a different spelling of the customer name or other typos. Swapped day and month in invoice dates can also be quite common in international business. Or imagine receiving a reminder for an already entered/paid invoice that differs in amount by a small penalty fee. These are examples of duplicates that cannot be readily found by simply comparing invoice details for an exact match.

Thus, a discussion of similarity was also needed. Do we compare amounts by percentage deviation or amount figures by typos? Is it sensible to combine both similarity criteria in one comparison or does it worsen the results? As a result, we added more flexibility to the model and improved the configuration options to allow more variants in the experiment.

Further experiments on the basis of this new model were performed with a variety of case-building strategies and rules. This was very much like a brainstorming session with very short cycles (minutes). The change of the model for the second iteration together with the experiments took two days. As a result, we had a much better model together with a clearer understanding of the problem domain, which led to a practical insight: Any further progress in the heuristics can be achieved only through reducing and accelerating efforts for devising new rules and configuring strategies. We used to use too much "copy and paste" programming for new rules. Another insight, which came after consulting with SAP's financial experts, was that SAP installations would necessarily require very different heuristics, because the reasons for invoice duplicates depend on the business processes and environment of a particular company.

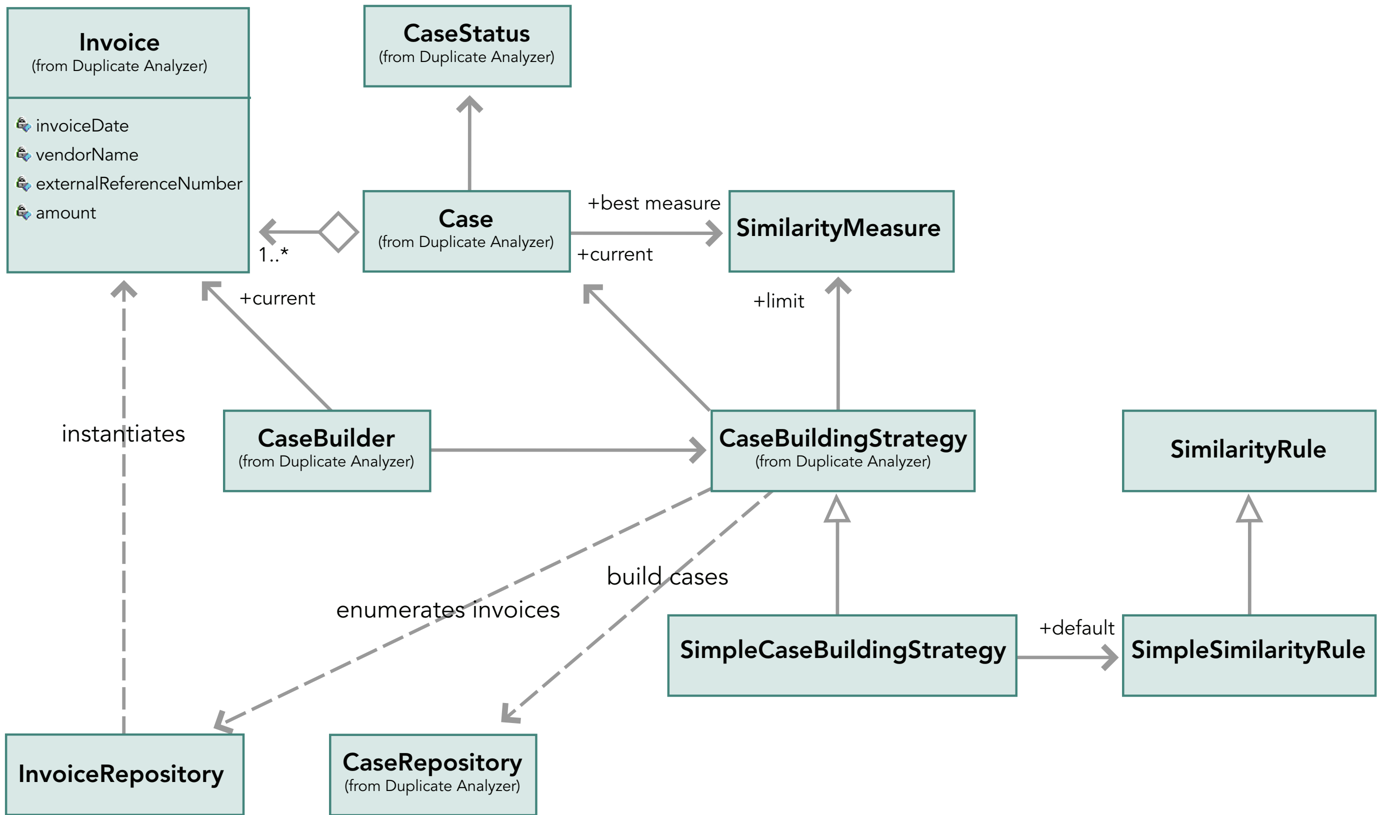


Figure 2

Iteration 3

We used the last iteration to apply these insights to the model. We added even more choices for the rules and also created experimentation tools that supported the configuration and execution of heuristics. The key extension of the experimental model shifted the similarity algorithms from the rules to the individual invoice attributes. A particular rule is then created by configuring the desired similarity algorithms for each invoice attribute involved. We no longer needed to create new rule classes for minor changes in the similarity of one attribute, which in turn greatly reduced the number of "copy and paste" operations.

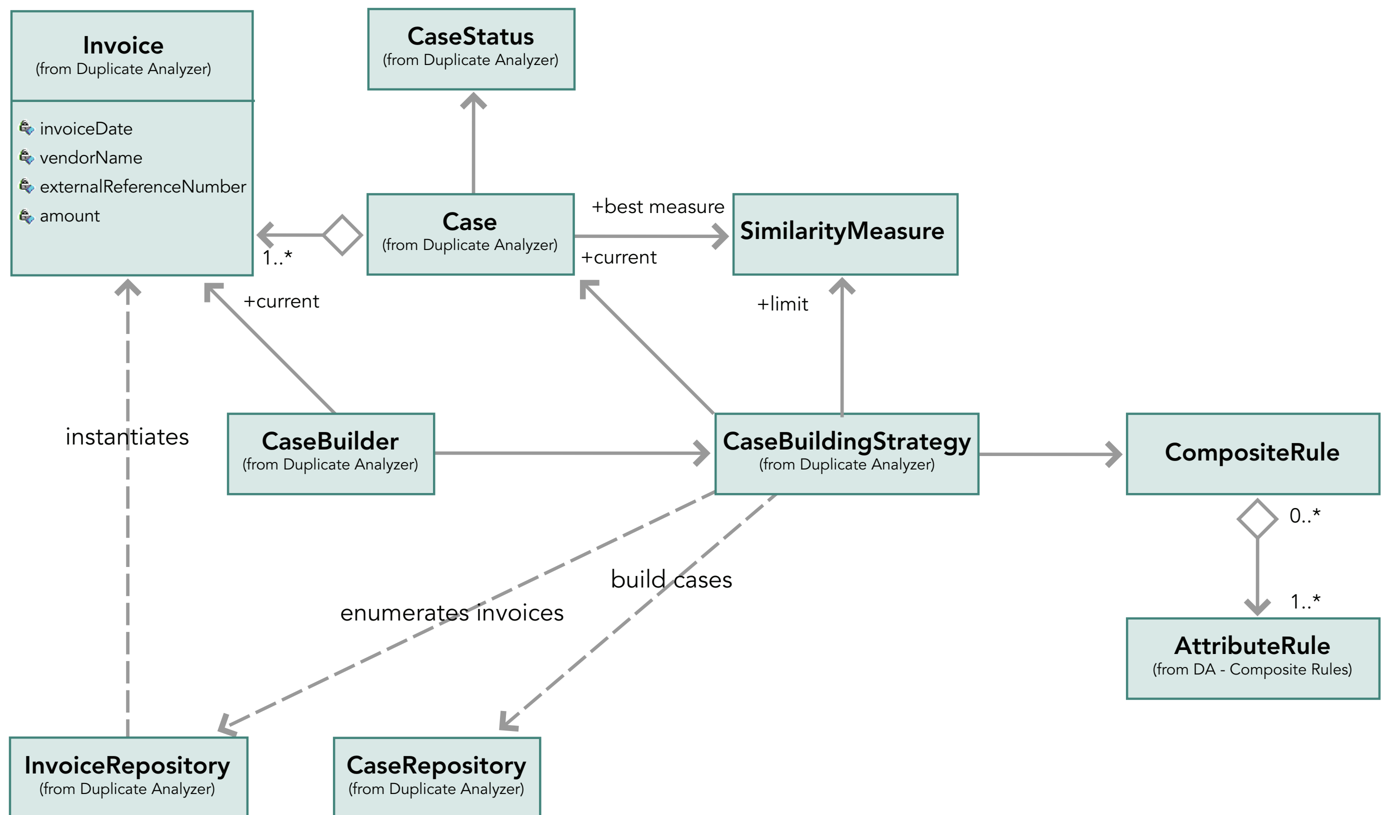


Figure 3

We then created a large number of experimental attribute rules and verified their utility with respect to the test invoices. For example, we used variants of the similarity of customer names by substring or typo, and amounts were compared by deviation in percentage, deviation in absolute numbers or typos.

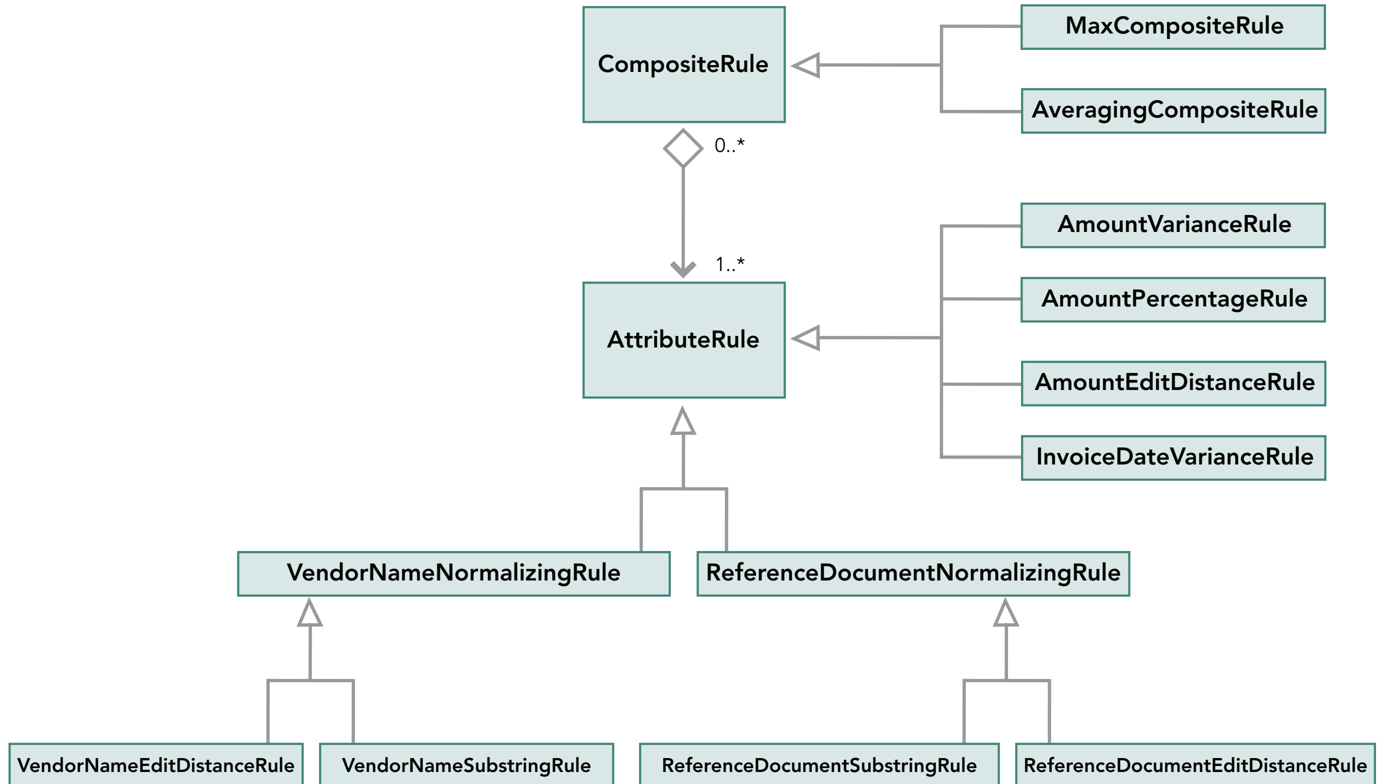


Figure 4

The experiments were supported by a dedicated tool that allowed customers to run "What if …?" experiments easily. The tool enables users to configure rules and rule parameters for one heuristic and also compare the results of runs with different heuristics. We managed to define heuristics that resulted in much improved findings of duplicate cases, both in terms of rejecting false positives and finding new true duplicates. The computing time was doubled to two hours for all 120,000 invoices.

A rigid, predefined algorithm was turned into a highly flexible model of a configurable rule system for comparing invoices. Customers can now determine experimentally the heuristics that are best for their specific invoice data.

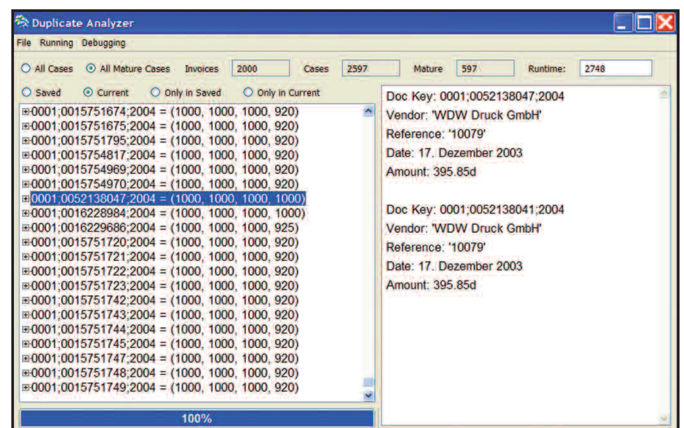


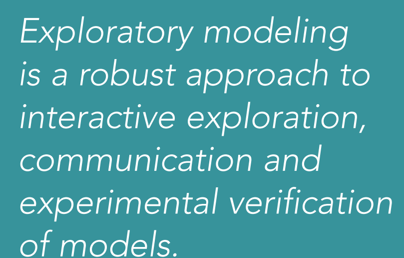
Figure 5

Summary

Exploratory modeling is a robust approach to interactive exploration, communication and experimental verification of models. It is based on experimental implementations of model concepts in a special programming language and on systematic experimentation with these executable models.

Exploratory modeling combines the principles of agile software development with immediately verifiable results that create a consensus between expert and developer. The necessary formal rigor of the model is achieved much more effectively through experimental implementation and its verification than through static UML diagrams. The resulting models can be used in multiple software development processes.

The example of the duplicate analyzer showed that exploratory modeling is particularly fruitful for unclear, informal or changing requirements and yields excellent, dependable results in a very short time.



Exploratory modeling is a robust approach to interactive exploration, communication and experimental verification of models.

Bibliography:

[1] "Domain-Driven Design", Eric Evans, Addison-Wesley, ISBN 978-0321125217

Cincom, the Quadrant Logo, Cincom Smalltalk and Simplification Through Innovation are trademarks or registered trademarks of Cincom Systems, Inc. SAP NetWeaver is a registered trademark of SAP AG. All other trademarks belong to their respective companies.

© 2007 Cincom Systems, Inc.
FORM CS070214-1-A4 4/07
All Rights Reserved.

World Headquarters • Cincinnati, OH USA • US 1-800-2CINCOM
Fax 1-513-612-2000 • International 1-513-612-2769
E-mail info@cincom.com • http://www.cincom.com

Contact our European offices:

Vienna, Austria
+43-(0)1-24027 524
infode@cincom.com

Brussels, Belgium
+32-(0)2-679 68 11
marketingbelux@cincom.com

Paris, France
+33-(0)1-53 61 70 00
marketingfrance@cincom.com

Schwalbach, Germany
+49-(0)6196-9003 0
infode@cincom.com

Torino, Italy
+39-011-5154 711
cincomitalia@cincom.com

Monaco
+377-93-10 01 20
cincommonaco@cincom.com

Vianen, The Netherlands
+31-347-358 458
info_europenorth@cincom.com

Madrid, Spain
+34-91-524 9820
cincomiberia@cincom.com

Stockholm, Sweden
+46-8-594 605 00
info_europenorth@cincom.com

Geneva, Switzerland
+41-(0)22-747 75 18
infode@cincom.com

Maidenhead, United Kingdom
+44-(0)1628-542 300
info_europenorth@cincom.com