Capture accurate solution requirements the first time with exploratory modeling (xM)

How to nearly eliminate post-go-live application design failures

by Heinz Roggenkemper, Ralf Ehret, and Andreas Tönne

Heinz Roggenkemper
Executive Vice President of
Development, SAP Labs, LLC

Ralf Ehret
Development Architect,
SAP AG

Andreas Tönne
Lead Consultant,
Cincom Systems

(Full bios appear on page 98.)

Perhaps the most difficult (and most fundamental) challenge SAP project teams face is defining accurate, implementable solution requirements for the business problem they are trying to solve. Consider the following example: imagine that you need to write an algorithm to check for duplicate invoices at some point in the process, such as following data entry or during a nightly batch job. You'll quickly find yourself asking questions like: What specifically qualifies an invoice as "the same" or different from another? Should two invoices be considered the same if simply the sold-to and total quantity are identical, or do we need to check that every single field is identical? What if all of the data "matches," but the lines are out of sequence? Do these criteria differ by company, department, or user (very common!)? What other potential cases have we failed to recognize/consider?

Unfortunately, the example above is not fictitious. This was the exact solution we were trying to build on a recent project at SAP, a project that caused us to reevaluate our approach to application development. In short, we invested a large number of hours developing the solution only to find that it did not work properly and required a considerable amount of rework.[1] The cause: late discovery of a significant gap in our understanding of how users go about deciding if an invoice is "the same," and a gross underestimation of the magnitude of the challenge.

This article explores an alternative, iterative approach to requirements gathering/application design called *exploratory modeling (xM)* that we have successfully applied to recent projects at SAP. The approach seeks to bridge the chronic communication gap between developers and users

[1] In fact, the technology we had selected for the similarity analysis (TREX) was predicated on our incomplete understanding of the requirements. We subsequently had to write an entire set of search routines we had not planned on, greatly expanding the project's scope, delivery date, and cost. If we'd had an accurate view of the project up front, we might not have undertaken it.

Note!

In case you're wondering, it's not that we didn't do our due diligence. We met with users to gather requirements as we always had on past projects. Unfortunately, the traditional approach of identifying requirements in one or two meetings with business users has some fundamental flaws (for the many reasons that we'll discuss shortly).

via frequent interaction with prospective (and future) end users and shortened analysis/design cycles for immediate feedback and design changes. It's a well-established technique to use HTML mock-ups to visualize static functionality and process steps. The basic idea of xM is to build a similar functional mock-up — an implemented running model together with a simple test application — for complex or unknown requirements and to use this as a basis for discussion with business users. The challenge is to create this experimental environment with a reasonable amount of effort.

Before diving into the details of xM, we'll identify the root causes we're trying to address a bit more precisely. We'll then define our approach and discuss its implications on two projects where we used xM to overcome problems. This article is aimed at all project team members, from project managers to developers, because the techniques are universally beneficial.

Note!

Before we proceed, let's be clear. The objective of this approach is not to eradicate all errors, omissions, and surprises from your projects, nor is this possible. The objective is to substantially reduce them, e.g., by 60%-70%. The remaining amount should be considered an inevitability of human imperfection.

Clarification on the causes of application design failure

First, let's clarify the challenges we're up against. In our experience, the overwhelming majority of application design failures can be attributed to one of the following:

- **Inadequate or ill-conceived requirements:** For a requirement to succeed, it *must* be specific enough to be implementable, but must not be *so* specific that it dictates or constrains its implementation. For example, the requirement "This product has to fulfill all existing and upcoming Internet standards" is too vague to be implemented, both because future standards are unknown and the standards are not enumerated in the requirement (ask yourself: what would you be able to *do* with a requirement like that?). In contrast, a requirement such as "I need a field xyz in the database table abc" is ill-conceived, both because it expresses nothing about the actual business requirement and leaves no flexibility in terms of its technical implementation.[2] Consider the frequent case in which, once the developer begins his or her work, this specific table field can't be used for some reason, or was specified incorrectly — the developer is left completely in the dark about the actual objective. Pressed with project deadlines or the prospect of halting development to reconvene users, developers often guess — sometimes correctly, sometimes incorrectly.

Note!

This is *not* a criticism of developers! This is a process problem for which all team members are equally responsible.

[2] Some users, particularly those with a bit of technical prowess, will frequently say something like this to a developer in an attempt to be helpful, but it is essential to recognize that this communicates nothing to the developer about the business requirement you are trying to achieve.

- **Inaccurate/undetected assumptions:** Our duplicate invoice example from the beginning of the article demonstrates that it is not feasible (from an effort, cost, or runtime performance standpoint) to code for every possible business case. In practice, simplifying assumptions must always be made about what users specifically require and what the data environment will be (i.e., what cases "will happen" in production). Whether you identify them or not, these assumptions are always made, and, in the end, dictate the amount of rework required after go-live. In our experience, major problems inevitably arise when two things occur:
    - Major assumptions are not identified and validated during requirements gathering.
    - Developers lack the necessary tools to dynamically fabricate additional assumptions during development — specifically, contextual information about the target business problem, and production-quality (realistic) data for research and testing purposes.
- **Landmines:** Occasionally, a fairly straightforward requirement mushrooms into an extremely complex requirement midway into the development process. Thus, as much as quality control methodologies anticipate and search for defects to ensure the quality of production data, it is prudent to specify a target number of these potential "landmines" in advance (e.g., three to five, depending on the project's scope), and seek out potential candidates before feeling any level of confidence in the accuracy of your project schedule and application design.
- **Changing ("moving target") requirements:** Sometimes requirements change during the development process (e.g., you make a change based on what one division tells you, but another division needs you to change it back), or completely new requirements will arise because the user gets new ideas. To be effective, it's essential to recognize this pattern, and have an active protocol for identifying and dealing with these situations.

While these direct sources of design failures must be minimized, in our experience, they are really side effects of the following three organizational/process deficiencies:

1. **Insufficient advance "domain knowledge" and/or development experience:** While it sounds cliché, the importance of staffing development projects with team members possessing both application design/development experience as well as specific *functional knowledge of the application area being worked on* — a.k.a. application *domain knowledge* — cannot be over-emphasized. We've learned the hard way that lack of either, more than any other element, virtually guarantees the types of failures discussed above.

Caution!

To reiterate: domain knowledge and development experience during requirements gathering are the two single most important determinants of design success, and should therefore be considered the foundation upon which all other improvements are based. Other improvements will be negated if either is missing.

2. **Deficiencies of vision, communication, and reflection:** Earlier we mentioned that meeting once or twice to gather requirements, while the de-facto design approach of IT project teams, suffers from some severe deficiencies. Perhaps you've shared our experience that end users have a hard time adequately and accurately describing what they need. There are a few reasons for this. The first is a deficiency of vision: Most people don't know what they need until they see it, and even then they require a few weeks of hands-on time to fully realize what they need. What's more, people can only describe their own experiences, which may be somehow limited or different from others. Second is a deficiency of communication: By-and-large, technical and non-technical users have difficulty communicating effectively.

(Using our earlier example, a business clerk who has internalized the process of identifying duplicate invoices to the point of instinct often has a hard time explaining to a developer — especially one new to accounts payable — the exact algorithm for comparing two invoices, and the developer may have similar difficulties expressing his or her understanding in a way the non-technical user can understand.) The third reason is that ideas need time to mature, and users must have time and initiative to reflect on and revise answers given or ideas generated during requirements meetings. Given these factors, it's not hard to see why the traditional approach can fail to yield well-conceived, stable results, especially if the participants lack experience with the process.

Note!

It would be nice if requirements could be gathered verbally in two one-hour meetings. Surely this is what everyone (including the users) has in mind and is the way many teams operate. More realistically, however, even the simplest of projects involves a significant amount of work. While you don't want to exhaust your end users or lose political capital, creating an appreciation — and in turn, the right atmosphere — for the amount of work involved is essential to getting everyone committed and motivated to put in the effort required to be detail-oriented. The more detail-oriented everyone is up front, the less effort will be required later to zero in on the requirements. One way to do this is to initiate your design meeting with a brief explanation of the process and past experiences of trying to rush through it — e.g., requirements/design phase shortcuts that resulted in significant hardship on *end users* after go-live. You can finish with the truism, "Look at it this way: we're going to have to put in the work either way … a lot now or a lot more later!"

3. **Inadequate process:** A key driver behind application design failure is the lack of a process that helps you to recognize patterns in the design process (e.g., "moving target" requirements), forecast and actively search for inevitable dangers (e.g., landmines and undetected assumptions), and create a culture of buy-in, detail-orientedness, shared purpose, and ongoing collaboration among end users and the development team. All of these are required to achieve the target of eradicating 60%-70% of post-go-live surprises/failures.

The bottom line: If you directly target these three organizational deficiencies, the number and magnitude of side effects (e.g., ill-conceived requirements and landmines) will significantly decrease. Let's take a look at how xM addresses these deficiencies through a new approach to modeling that bridges the gap between developers and business users and ensures a more accurate prototype in less time.

A new approach: exploratory modeling (xM)

A real problem at the outset of a software development project — particularly in a new business domain — is a lack of knowledge and the lack of a common language between business users and developers for expressing their needs. A business clerk who knows how to identify a duplicate invoice will have a hard time explaining the concept in terms a developer can understand. He or she doesn't have development experience. On the other hand, developers often have difficulty expressing their understanding of the requirements and their needs in a way the (non-technical) user can understand. A developer can't possibly have the domain knowledge that would be needed to speak the language of every user that he or she will have.

The shortest explanation of what xM does for your software development project is this: It enables the communication between user and developer by using non-technical, business-level terms and

Where does xM fit into the development process?

You may be wondering how exploratory modeling (xM) relates to other development methodologies such as Agile, eXtreme Programming (XP), and Scrum.

- Agile is the high-level approach to application development from which the others derive.
- XP and Scrum are specific realizations of the Agile methodology for managing the overall development process.
- xM focuses on a portion of the overall development process — specifically on the exploration, definition, and management of requirements; their conversion into an application design; and the validation of that design.

Like XP and Scrum, xM adheres to the best practices outlined by Agile — in this case, using "models" as the foundation of application design/requirements discovery, and prescribing defined processes for user interaction, language, and design verification.

concepts. It provides a platform for the modeling phase of software design and the mutual reassurance between users and developers that the resulting model is right. Communication and reassurance are key values of the Agile Manifesto[3] and are the drivers behind Agile-based user- and interaction-centric software development processes, such as extreme programming (XP), Scrum,[4] and xM (see the sidebar above for more on how xM relates to these other methodologies).

Agile-based processes like xM have proven themselves highly effective for the design and implementation of modern, innovative applications. Their values and principles are a natural outcome of object-oriented programming. The key point is that it is not the technical act of implementation ("How do I tell the machine?") but the thinking about problems and concepts ("What am I talking about?") that is paramount. No formal, elaborate software development process can help you to understand the concept of "invoice duplicate." Even if you have modeled this concept and documented all development steps and specifications in exquisite detail, it does not stop your user from finding that "this is not what we intended." xM puts exploration, experimentation, and iteration in the center of the modeling phase of software development and puts prototype tools in the hands of users much earlier in the process than traditional approaches.

The challenges of modeling

Capturing and modeling concepts correctly is fundamental to software development; without understanding the way a user is thinking about something, you cannot make progress in collecting his or her requirements. Modeling is the critical phase of software design where the domain language gap between the user and developer's understanding of concepts is bridged. This is the phase where the most expensive mistakes of the whole software development can be made.

Modeling is further complicated by the various formalisms and abstractions of concepts that are

[3] See http://agilemanifesto.org/ for the complete manifesto.

[4] For more on the Scrum form of Agile development, see the *SAP Professional Journal* article "Agile development for SAP: Get into the Scrum!" (May/June 2007).

Selecting a language/format for model diagramming

While xM does define specific requirements for the modeling diagram language/format, it is very important to point out that xM does not describe or mandate any particular format (or language). Rather, xM merely prescribes the iterative process of defining/refining models, building prototypes, and validating the design through hands-on use, and defines the following requirements for whichever language/format you choose to use to express your models:

- **Non-technical/barrier-free:** Terms/concepts must be expressed using business terminology. This guarantees that users will understand what they're looking at, while giving developers a detailed model from which to build.
- **Meta-programmability:** Meta-programmable languages enhance the programming environment by framing user notions in business terms instead of forcing the developer to translate the user notions to implementation artifacts. It is important to keep the "distance" between expert notions and the experimental implementation as short as possible.
- **Interactivity:** Interactivity of the programming environment is crucial for the acceptance of xM. Larger delays in the model adaptation by time-consuming "compile-link-test" cycles break the communication flow between user and developer and also slow down the agility of the experiments.
- **Concept-oriented (object-oriented):** This means that elements within the model diagram must be business objects (such as invoices, customers, and payment types) versus technical items, such as database tables and function modules. This is critical to preventing communication problems.

used by developers and users, which are a source of fundamental misunderstandings. The developer seeks to formalize the concepts as quickly as possible, and wants to classify them in schemata of abstraction and generalization. The user, on the other hand, uses different strategies for describing the concepts, and is accustomed to thinking about them in terms of examples, analogies, and domain-specific classifications.

If communication and understanding is so important, then you would expect it to receive special treatment in software development processes. Amazingly, most popular processes try to bridge this language gap using yet another technical (formal) language: UML. This is not so amazing when you consider that UML is designed to represent technical implementation details, such as classes, object relationships, and object interactions. So, while UML satisfies developers' needs for formalism and standardization of the model description, and is thus handy for design and implementation documentation and communication between developers, it violates the fundamental Agile principle of user-orientation. The user still faces the challenge of communicating his or her understanding of the requirements and concepts in a way that can be verified against the UML model. For many domain experts, this is a difficult task and will lead to overlooked model errors, which then surface as landmines late in the implementation cycles. These are deficiencies of vision, communication, and process and, as you will see, xM provides developers with a mechanism to meet users more than halfway to address these

Currently, there are only a small number of languages/formats that meet these requirements, including Cincom Smalltalk and Ruby. For our project, we chose Smalltalk for the following reasons:

- It has a very simple syntax with little declaration and structural overhead that might confuse the clarity of the model.
- It allows us to define methods without declaring the types in signatures (this is called “duck typing”). This feature makes Smalltalk feel like interpreted language.
- “Edit-compile-link” cycles are completely unknown in Smalltalk. Programming proceeds incrementally by adding classes and methods one by one.
- Very little programming (typically 1%-5% of a comparable implementation) is needed for the first cycles.
- The program is executable after every programming increment, which allows sub-hour xM cycles or better.

You can find more information on Smalltalk at http://www.cincomsmalltalk.com, and more details on Ruby are available at http://www.ruby-lang.org. For an excellent comparison of the two, see the two articles by Huw Collingbourne at http://www.sapphiresteel.com/Ruby-The-Smalltalk-Way-1 and http://www.sapphiresteel.com/Ruby-The-Smalltalk-Way-2-A.

deficiencies and to support the communication of concepts and their verification.

xM to the rescue

xM starts with the challenge of bridging the communication gap between developers and users and verifying concepts to find the right model. It replaces abstractions with executable models, whose properties can be experienced and verified immediately by both the user and the developer. Model finding does not start by drawing detailed UML diagrams, but rather by running very short modeling cycles in which you build an experimental implementation using names and concepts based in a language both developers and users can understand (see the sidebar above for more on choosing a modeling language). Then you run experiments — the user and developer together experience what the proposed model does, what it feels like, and what feels right or wrong. Necessary changes to the experimental implementation model are documented (for this purpose, in whatever style you like, including UML, because the documentation is for developers) and updated in the experimental implementation. Then the experiments start over again.[5] This iterative process of experimental implementation, verification, and updating of the documentation is the heart of xM.

[5] An interesting introduction to modeling by implementation is available in the book *Domain-Driven Design: Tackling Complexity in the Heart of Software*, by Eric Evans (Addison-Wesley, 2003, ISBN 978-0321125217).

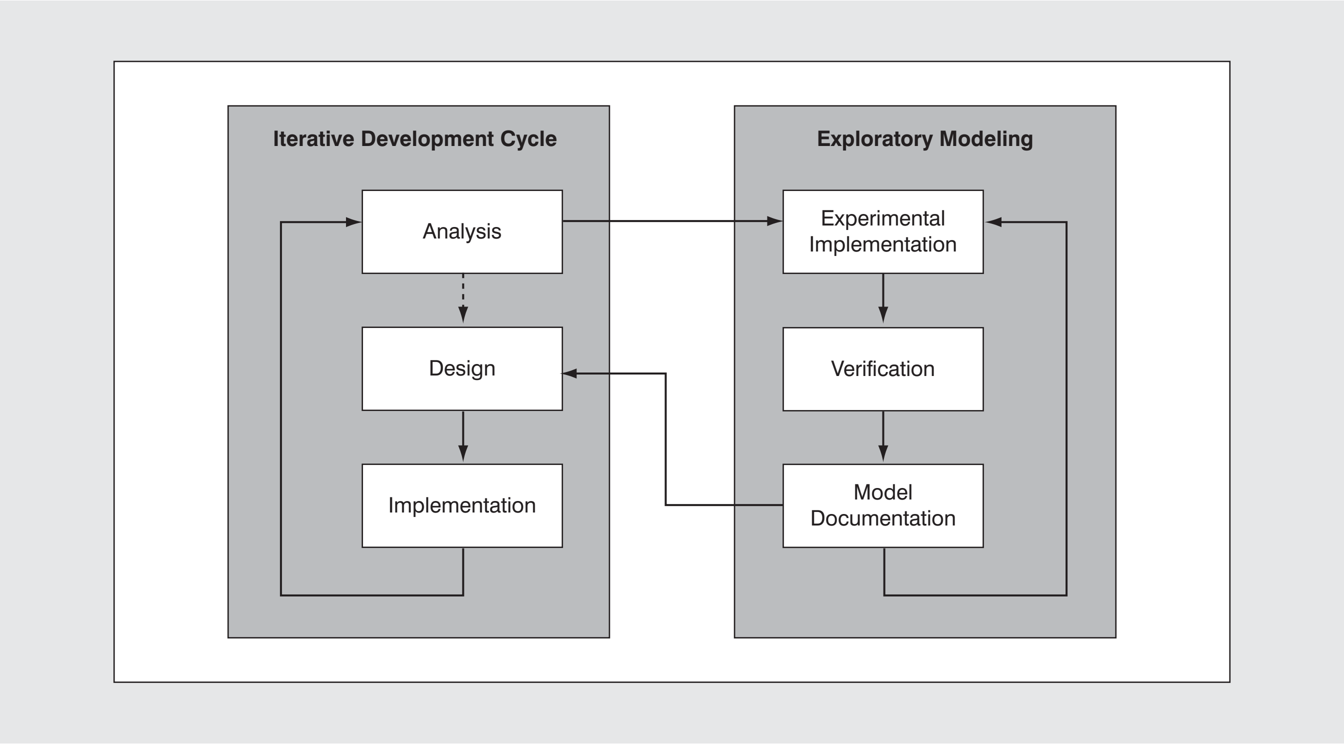


Figure 1 xM in an iterative development cycle

Figure 1 illustrates where xM fits into an Agile-based, iterative development cycle. xM has its greatest effect at the beginning of a software development project, where the domain is unknown and the uncertainty is largest, but later cycles will also benefit from injecting xM phases. This is particularly true when a cycle introduces substantial new concepts or model changes, because it helps to confirm that the changes are well understood and accepted by the user. xM can even find a place early on in non-iterative, "waterfall-type" processes, where it will produce greatly improved results by enhancing the depth and quality of the development foundation.

At this point you may be wondering what exactly is the output of xM? As shown in **Figure 1**, the xM subcycle runs outside of the traditional development process. Its output is basically improved model documentation for developers, which is what is fed back into the process. So, once the documentation is done and you're ready for the final implementation, do you throw away the experimental model implementation and start fresh? That depends on a variety of factors and the needs of the particular project. Using a modeling language (Smalltalk, in our case) as the final implementation language has its benefits, but pure xM strictly ends with production of the model documentation.

Another question you might have is to which problems you should apply xM? We use an 80/20 rule of thumb: About 80% of the development effort goes into 20% of the requirements. It does not make sense to model the whole project using xM; stay focused on the root causes of design failures described earlier (insufficient domain knowledge; lack of vision, communication, reflection, and commitment; and inadequate process), which are likely to represent the biggest threats.

Now that we've explained the theoretical side of xM, let's take a look at how we applied xM to two separate projects: the (re)development of our duplicate invoice detection application and the development of

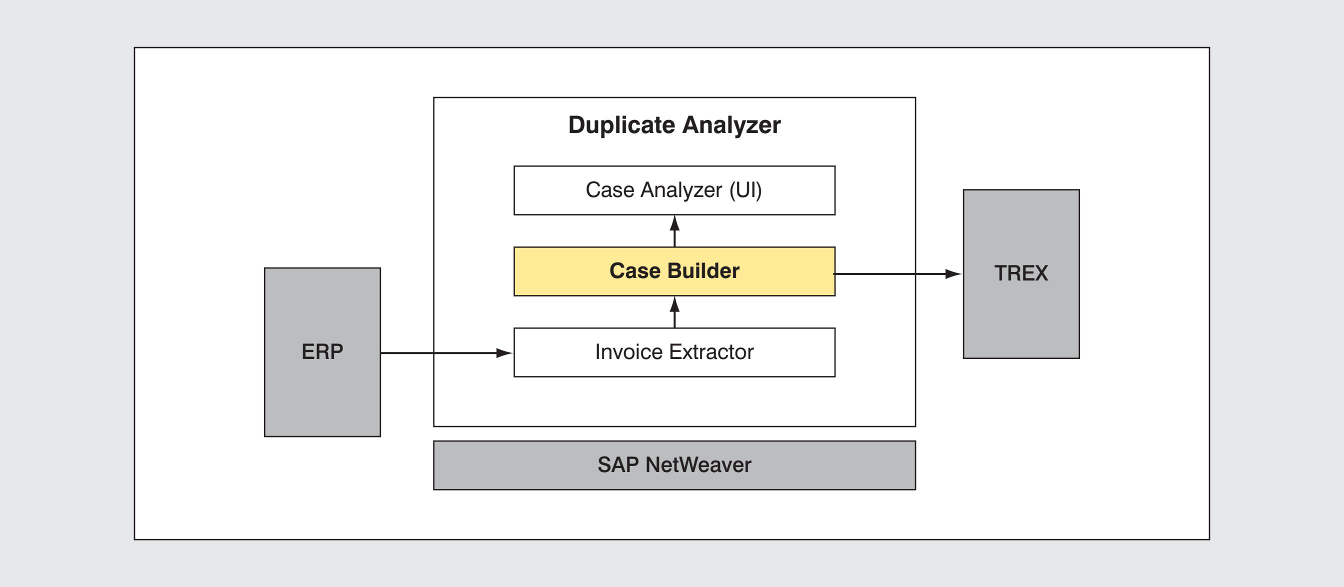


Figure 2 Architectural overview of the Duplicate Analyzer

a shipping service integration solution. For each case study, we applied xM concepts by following these four[6] steps:

1. Talk to the business users.
2. Build the initial model.
3. Build the experimental environment (the test application).
4. Verify the model with business users by performing the experiment and refine the model as needed.

In the following case studies, you'll see how using xM by following these steps helps you to avoid the root causes of design failures.

Case study #1: The Duplicate Analyzer

The Duplicate Analyzer application, which we mentioned earlier in the article, is an application that identifies potential duplicate invoices to avoid or reclaim double payments. The prototype consists of three main components, shown in **Figure 2**: an Invoice Extractor, a Case Builder that encapsulates the similarity algorithm for identifying potential duplicate invoices, and a Case Analyzer UI that allows an accounts payable clerk to compare identified potential duplicates.

The similarity algorithm is based on a set of invoice attributes, such as invoice number, supplier name, amount, and invoice date. For the central comparison engine, we reused the TREX search and classification engine of SAP NetWeaver. We quickly had a working prototype that analyzed a large number of invoices and ranked possible duplicates. Our pilot users were happy, but we were not satisfied.

We immediately recognized that there were a tremendous number of false positives — potential pairs of invoices that were identified as a duplicate case but were not, in fact, duplicates. This was a sign that we did not really understand how accounts payable clerks detect duplicate invoices because they were not able to give us precise rules. We did not understand the domain well enough and therefore we

[6] Steps 2 and 3 can be done in parallel.

did not model it accurately. We just used the existing search capabilities of TREX, which are not adequate for semantically comparing invoices. Understanding this issue, we looked for an appropriate similarity algorithm, but we were restricted by the capabilities of the comparison tool (TREX). At least we identified our problem, however: the similarity algorithm inside the Case Builder.

To solve the problem and determine the appropriate similarity algorithm, we decided to try xM with the Case Builder component of the application. Our first step was to start the implementation of the Case Builder from scratch by creating duplicate detection rules, but this time using rules that could be discussed in the language of business users.

We chose Smalltalk as our implementation language (see the sidebar on page 84 for details on this choice). Because our team did not have any Smalltalk experience, we started the project with one week of training on the basics of Smalltalk.

> ***Note!***
>
> It is worth mentioning that we would not have been able to do the job on our own with only one week of Smalltalk experience! Smalltalk and xM alone aren't "silver bullets." Achieving improvements like the ones discussed in this article requires a rigorous rethinking of traditional development methodologies. We were able to deliver rapid results largely because we had experienced mentors.

After the training, our small team — two colleagues from SAP and two experienced Smalltalkers — was ready to start building a model for the Case Builder. But first, we needed to understand how the accounts payable clerks identify duplicates.

Step 1: Talk to the business users

We talked to the domain experts — the accounts payable clerks of our finance department responsible for recording and verifying invoices. Our goal was to understand the concepts they used when finding duplicates and to identify the associations between these concepts. We asked them simple questions, such as, "How did you identify a duplicate check before?" It sounds trivial, but the answers were revealing and were key to ensuring an accurate model. They talked about "invoices" and about "rules" based on some invoice attributes — e.g., "If the amount of two invoices is exactly the same, then these two invoices belong to a suspicious 'case,' except if they are marked as recurrent invoices." We listened carefully to them and didn't ask too many questions, because we didn't want to affect their answers with the model we had in mind. Through these discussions, we started to think about the model from the perspective of a business user, using business user concepts and terminology. After about half a day, we were ready to sketch our first model for the Case Builder.

Step 2: Build the initial model

The first implementation of the executable model took us roughly two hours. The UML diagram in **Figure 3** shows the result of the initial model based on the concepts we learned from the business users — specifically, the "invoice," the "case," and the "rule." In addition, we added some technical classes to the model representing the functionality we knew we would need: a CaseBuilder class, an InvoiceRepository class, and a CaseRepository class. You can also see that we named the classes, as much as possible, with terminology a business user would use. The only *technical* classes were the InvoiceRepository and the CaseRepository, but the interfaces to these are described in plain language in our model.

As you can see, the base functionality is really simple and the model is clear and easy for anyone to understand: The CaseBuilder reads Invoices stored in the InvoiceRepository and compares them by applying a Rule. If the result of applying the Rule is a

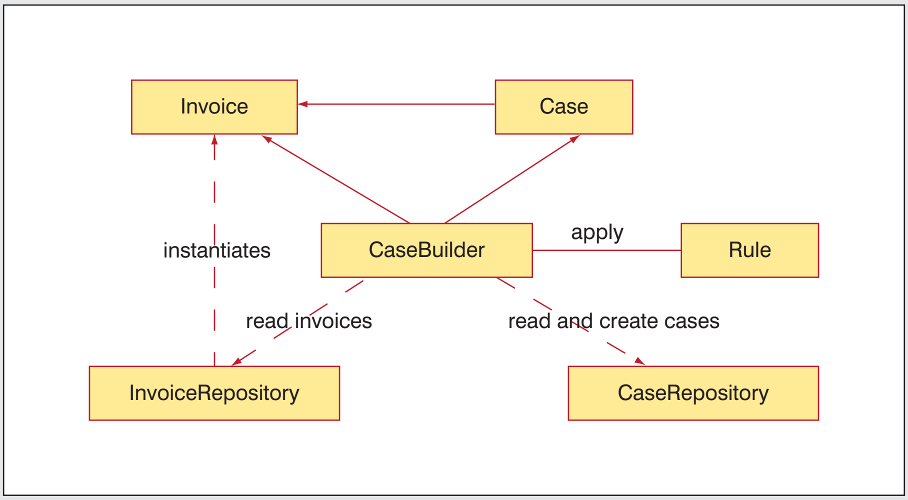


Figure 3 First domain model of the Case Builder

Note!

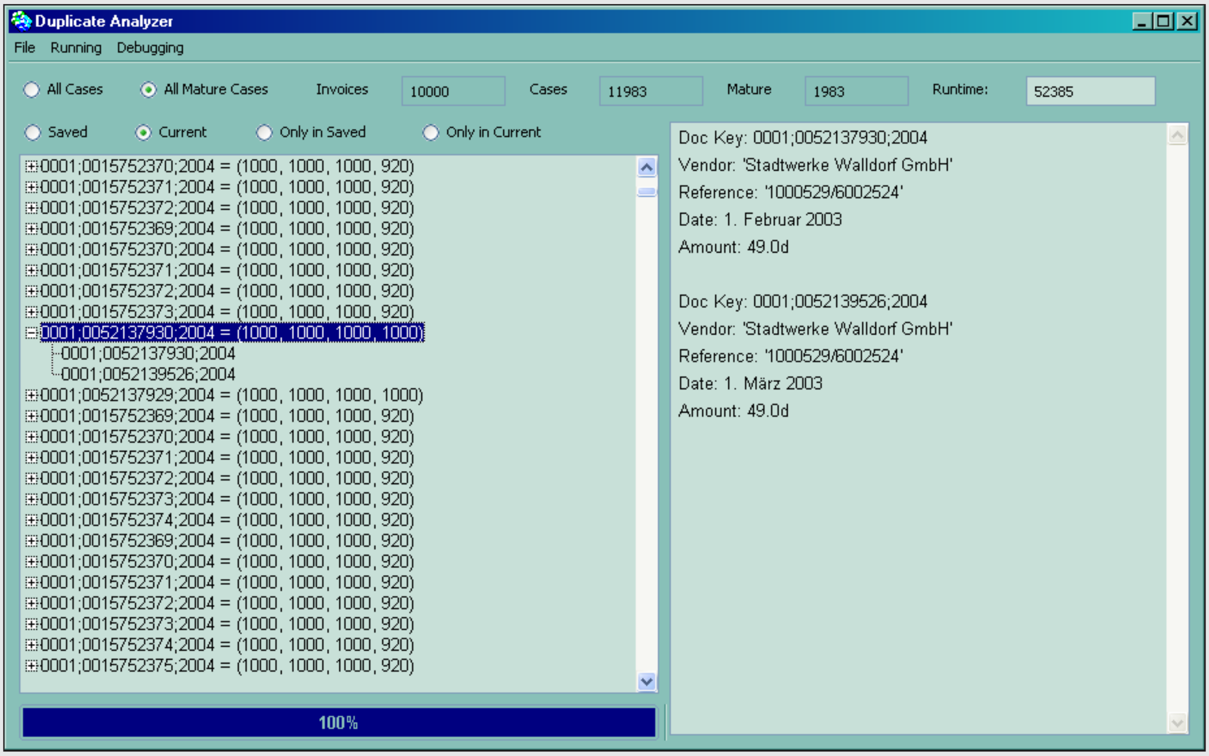
During the implementation of the algorithm, we realized that we had to introduce some more terms into the model, such as SimilarityMeasure, a design class responsible for calculating the similarity of two invoices. While none of the business users actually used this term, it represented a very important but unspoken fact — all accounts payable clerks have a similarity measure in mind when they compare two invoices and have a "feeling" how likely it is that they are duplicates. This feeling is based on their own experiences, however, and it is very hard for them to describe this feeling to someone else. On the other hand, it is very hard to implement a "feeling" — we needed an exact and reproducible measure (a SimilarityMeasure design class). This is a typical example of a hidden assumption.

suspicious pair of invoices, the CaseBuilder creates a Case and stores it in the CaseRepository.

The next step was to implement the duplicate identification rules. As a starting point, we decided to use the same similarity algorithm used in the TREX engine. We then analyzed the existing known duplicates identified by the previous version of the application, such as invoices with similar but different vendor names (like SAP AG and SAP) and duplicates with different external reference numbers, and used the information we gathered to implement the duplicate identification rules and refine the comparison algorithm.

Step 3: Build the experimental environment (the test application)

It's typical in xM to separate completely the experiment from the existing production environment, so while we were building the initial Case Builder model, we also built a simple test application to serve as the experimental environment for the Case Builder.



○ All Cases ● All Mature Cases Invoices 10000 Cases 11983 Mature 1983 Runtime: 52385

○ Saved ● Current ○ Only in Saved ○ Only in Current

```
⊞0001;0015752370;2004 = (1000, 1000, 1000, 920)
⊞0001;0015752371;2004 = (1000, 1000, 1000, 920)
⊞0001;0015752372;2004 = (1000, 1000, 1000, 920)
⊞0001;0015752369;2004 = (1000, 1000, 1000, 920)
⊞0001;0015752370;2004 = (1000, 1000, 1000, 920)
⊞0001;0015752371;2004 = (1000, 1000, 1000, 920)
⊞0001;0015752372;2004 = (1000, 1000, 1000, 920)
⊞0001;0015752373;2004 = (1000, 1000, 1000, 920)
⊟0001;0052137930;2004 = (1000, 1000, 1000, 1000)
   ├0001;0052137930;2004
   └0001;0052139526;2004
⊞0001;0052137929;2004 = (1000, 1000, 1000, 1000)
⊞0001;0015752369;2004 = (1000, 1000, 1000, 920)
⊞0001;0015752370;2004 = (1000, 1000, 1000, 920)
⊞0001;0015752371;2004 = (1000, 1000, 1000, 920)
⊞0001;0015752372;2004 = (1000, 1000, 1000, 920)
⊞0001;0015752373;2004 = (1000, 1000, 1000, 920)
⊞0001;0015752374;2004 = (1000, 1000, 1000, 920)
⊞0001;0015752369;2004 = (1000, 1000, 1000, 920)
⊞0001;0015752370;2004 = (1000, 1000, 1000, 920)
⊞0001;0015752371;2004 = (1000, 1000, 1000, 920)
⊞0001;0015752372;2004 = (1000, 1000, 1000, 920)
⊞0001;0015752373;2004 = (1000, 1000, 1000, 920)
⊞0001;0015752374;2004 = (1000, 1000, 1000, 920)
⊞0001;0015752375;2004 = (1000, 1000, 1000, 920)
```

```
Doc Key: 0001;0052137930;2004
Vendor: 'Stadtwerke Walldorf GmbH'
Reference: '1000529/6002524'
Date: 1. Februar 2003
Amount: 49.0d

Doc Key: 0001;0052139526;2004
Vendor: 'Stadtwerke Walldorf GmbH'
Reference: '1000529/6002524'
Date: 1. März 2003
Amount: 49.0d
```

Figure 4 Case Analyzer UI created for the experimental environment

In order for the business users to evaluate and verify the Case Builder, we needed a simple application where we could inspect the results of a duplicate analysis: a ranked list of cases with potential duplicate invoices. So the task was to load and analyze some invoices and to present the potential cases in a very simple form.

To create the test application, we built an InvoiceRepository (to serve as a source of invoices) and a simple Case Analyzer UI (to display the results to the business users). Without these components, we would not be able to test our model Case Builder. The Case Analyzer UI included a simple tool to analyze duplicate cases, which gave us some statistics and allowed us to customize our Case Builder in a simple way by switching on and off some similarity calculation algorithms of attributes and selecting different case building strategies.

Figure 4 shows the simple test Case Analyzer UI. As you can see, we did not spend too much effort on it, and there are two very good reasons for this. First, you lose momentum on the project when you spend too much time creating highly sophisticated, well-designed UIs for test purposes. Keep in mind that the user does not have to work with this test UI, it's just a medium to start communication. You will be sitting next to the users when they are using the test interface, and you can explain the application, so stay focused on the important part — the modeling! The second reason is that if you provide a stylish UI at this stage, the users will expect this same UI to be the real UI of the application, even if it is not possible in practice.

With the experimental environment in place, we were able to perform some test runs of the comparison algorithm we implemented for the initial Case Builder model. We were not able to find any new duplicates or reduce the number of false positives, but we did not expect to because all that we had done was rebuild the old Case Builder. The real accomplishment was building the complete environment for our experiment, including a first implementation of our model. So, after roughly three days, we were ready to begin our experiment and start evaluating cases together with the business end users.

Step 4: Verify the model with business users by performing the experiment and refine the model as needed

An important moment came: the first verification of our model with our business users.

At first, our discussions were similar to typical requirements-finding sessions — we (the developers) would ask, "How do you identify a duplicate, by looking at the attribute amount?" and the accounts payable clerks would answer, "It depends on the process: where does the invoice come from, which contract do we have with the vendor, etc." Then, we showed them our test application and the defined rules we used to produce the potential duplicate cases. Looking at the duplicate cases in which we changed the rules was the ice-breaker. Suddenly the accounts payable clerks had a direct connection to their specific domain and a very good discussion about "invoice processing" started — and we recognized that business users think in terms of processes rather than thinking in terms of attributes, like developers do.

After a while we realized that each duplicate detection rule represented a gap in the process of invoice management itself.[7] And now it was clear why our original implementation of the Duplicate Analyzer did not result in improving the process. We were only looking at the symptoms — we did not try to find the reasons. From the beginning of the project, we had asked the business users the wrong question. Rather than asking *how* they identified duplicate invoices, we needed to ask *why* duplicates occur. By using xM to close the communication gap between the domain experts and us (the developers), we were able to identify the root cause of the problem, and the domain experts were able to provide answers that would improve the process significantly.

To refine the model, together with the business users, we identified potential gaps in the process that could lead to a duplicate invoice and we created new duplicate detection rules based on these gaps. By asking the right questions, we learned that similarity for the string "external invoice number" is different from similarity for the string "vendor name." The vendors "SAP" and "SAP AG" are obviously the same. But from the point of view of a string similarity algorithm, they look quite different. The old Case Builder solved this problem by normalizing the vendor names — by changing letters to uppercase and removing common fill words like "AG," "ltd," etc. On the other hand, the "external invoice number" similarity is quite different. It's a kind of number with additional semantic meaning: parts of the number could describe the year, separated by some special characters like "-" or "/," while other parts could represent an ongoing order number, including some characters for the external specialist (e.g., his or her initials). Last, but not least, the similarity interpretation of the attribute "amount" could be based on number or string similarity. Sometimes a bill was not paid on time and a reminder was sent with an additional dunning. So two amounts could be similar because of a number similarity: -$1234 and -$1239.50, in which case the second number contains a dunning of $5.50. On the other hand, a typo could occur in an amount of an invoice, causing the accounts payable clerk to request a corrected invoice — in this case, a potential duplicate is based on a string similarity, e.g., -$1234 and -$1324. As you can see, one attribute could have different rules for similarity.

[7] Let's say you are an accounts payable clerk and you find a typo in an invoice that is already in the FI system. You request a corrected invoice from the vendor, but you forget to note this, and the next day you are on vacation. Will your colleague know to cancel the old one?

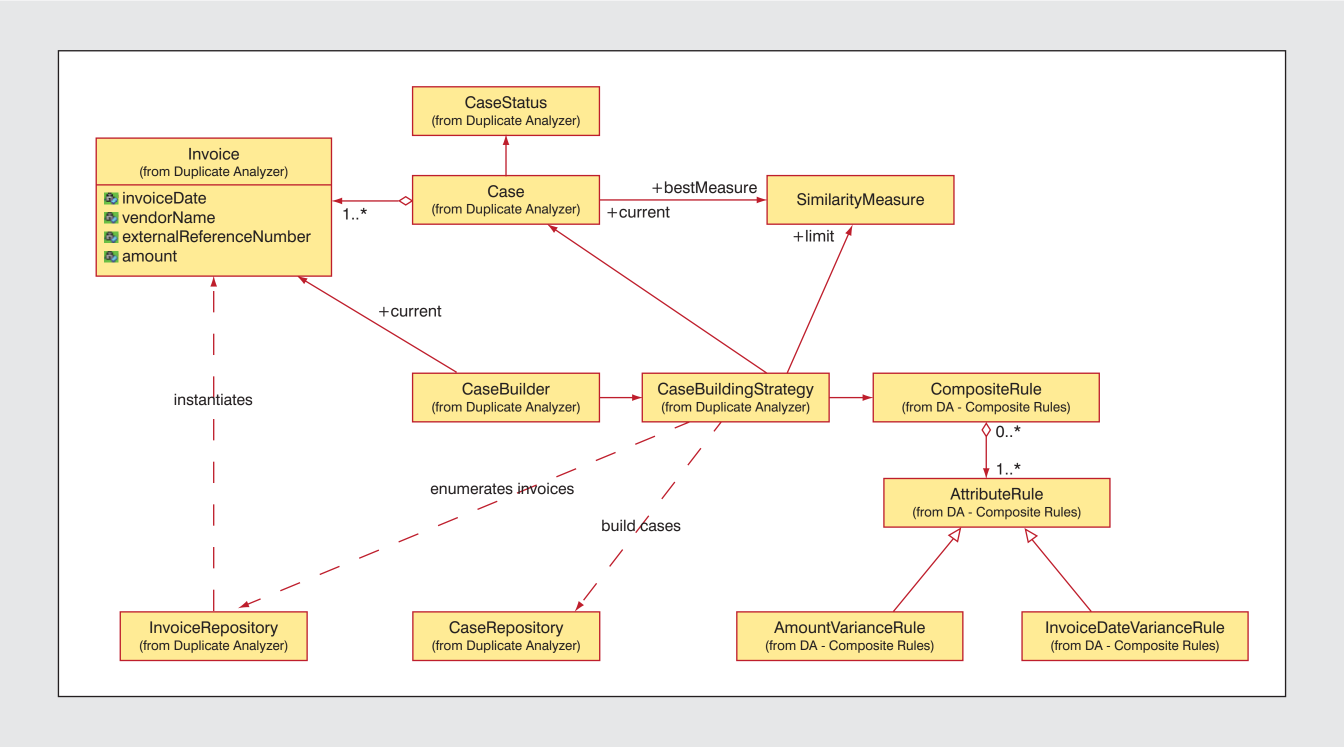


Figure 5 Final model for the Case Builder

By factoring in such potential gaps, in the end, we arrived at a complex but very flexible model, which is shown in **Figure 5**. If you compare this UML diagram with the first domain model in **Figure 3**, you can see that the overall structure is still the same, but with much more detail added. The Case class now has a reference to the SimilarityMeasure class mentioned earlier. We also needed a CaseStatus class for selecting cases for the UI. Another area we refined completely is the Rule class, which we changed frequently during our experiment. We learned that a single rule returning a similarity measure between 0 and 1000 would not be adequate. We needed a variety of rules to compare single attributes of the invoices, such as the similarity measure AmountVarianceRule. We also needed rules to combine similarity measures from single attribute comparisons, which we called CompositeRules. Last, but not least, we introduced a CaseBuildingStrategy responsible for evaluating the CompositeRules and classifying a pair of invoices as duplicates or not.

Lessons learned

By applying xM to this project, we learned a lot about both the domain and the process. The result was a very complex, but flexible and easy-to-understand rule system. For each process gap we identified together with the domain experts, we created a new rule. And at the end, we were able to find *new* duplicates!

As in most projects, we did not meet the timeline. The project was scheduled for two weeks (10 working days). But it was — at least for us — the first time that we finished a project *before* the scheduled end! After only seven days we had:

- A clear view of the core concepts
- A flexible implementation of similarity algorithms
- A support for case building strategies
- A better feeling concerning our knowledge of the data

- A much better understanding of the problem

So overall, we had surprising progress only a little over a week into the project! Even though the initial prototype did nothing more than replicate the rules that were already in place, by putting the model in the language of end users, we laid the foundation for a very important realization: The problem wasn't how duplicate invoices were uncovered, but how duplicates were created in the first place. The most gifted developers could have spent months building the most sophisticated Duplicate Analyzer in the world, but without framing the model in terms that the end users could understand, the root cause of the issue may never have been identified and the Duplicate Analyzer itself may not have resulted in any real business value. By bridging communication gaps and involving end users in the early phases of the model development (in other words, by putting ourselves, the developers, in the end users' shoes), we leveraged the classic tenets of xM to cut to the heart of the issue and deliver results that created real value.

Stimulated by the success of our internal Duplicate Analyzer project, we wanted to see if we could improve our requirement analysis with customers. We decided to apply what we learned to another application where we suspected that our pilot customers did not really know what they wanted: the xCarrier project.

Case study #2: The xCarrier project

The xCarrier project was an integration solution that we developed for parcel shippers and carriers. It is part of a complete logistics execution process. Previously, it had been very hard for shippers to connect to a carrier, largely because it was a point-to-point integration where you needed to map the data on the shipper side as well as on the carrier side. We proposed a flexible and extensible multi-carrier solution based on a service discovery model. In this giant project, we also needed a tiny little component: a rule engine to decide which carrier service product[8] should be used for a certain shipment based on the parcel's weight, volume, ship from/ship to address, and additional attributes of the shipping organization, such as the responsible sales organization and customer classification data (is the customer a "gold customer"?). Before the xCarrier project, it had been necessary to "hard-code" the rules in customer implementations. It's no surprise that some customers simply decided not to use very many different carrier services. Other customers developed some monster routines with more than 1,000 lines of coding containing too many IF statements. In the end, most customers simply couldn't tell us what rules they really needed. Our challenge was to develop a flexible solution using a rule engine to decide which parcel will be sent by which carrier and which service has to be used.

At the beginning, we didn't pay much attention to the rule engine requirement, because we knew that we had several rule engines or components at SAP. Some of them were too specific to the area where they were used, but there were at least six that possibly could fit. After a while we got nervous. What happened?

We spent a lot of time evaluating all of the options, but we still didn't know the exact requirements of our end users. Too often, the requirements seemed to be driven by the constraints of the evaluated rule engines, not the actual business needs. And, as is all too often the case, if we simply asked our customers, "Do you need feature A of rule engine Z?" the answer was always, "Of course!" At the end, we looked at the requirements for our rule engine and realized that no rule engine at SAP could fulfill all of the requirements, and furthermore, the poor shipping manager responsible for maintaining the rules would need a lot of knowledge in the area of Boolean algebra — very unusual for a forwarding merchant!

[8] A carrier service product is a service that a carrier such as UPS offers (e.g., to send a parcel). You know this service from using Amazon, for example, when at the end of an order, you select how the book you just purchased will be sent: by UPS Standard or Express.

We had a situation where requirements were difficult to discern, our end customers (the shipping managers) had developed unstated rules, and constraints in existing technology had started to drive the requirements. In other words, we had a perfect situation to try out xM.

We decided to set up a project with one SAP resource and one experienced Smalltalker to build a rule engine and a mock-up application where we would be able to simulate a shipping request and evaluate some rules to find a suitable carrier service. At the same time, another SAP colleague started to evaluate possibilities for connecting our prototype rule engine to an existing SAP NetWeaver system.

We followed our xM approach from the Duplicate Analyzer project.

Step 1: Talk to the business users

First, we talked to the domain experts to try and gain an understanding of the concepts and processes they used. Based on what we learned, we identified the shipping attributes as the best candidates to serve as the rule attributes.

Step 2: Build the initial model

With an understanding of the domain experts' perspective, we were ready to build the initial model. With our two resources — the SAP resource and the experienced Smalltalker — we developed a first prototype of a rule engine tailored to the requirements we knew at this point in time.

Step 3: Build the experimental environment (the test application)

In addition to the rule engine, we needed a simple rule editor to define the rules and a UI to input a shipping request and inspect the results of the rule engine evaluation. Based on what we determined from our discussions with the domain experts, the rule definitions would use the shipping request attributes.

For example:

```
IF
    Weight is less than 9 kg and
    Volume is less than 5 l and
    Sales Organization is 1000, 1010 or 1020 and
    Ship to address is in Europe and
    Customer is “gold Customer”
THEN
    Use UPS 2nd day air with insurance
```

After five working days, we had our first iteration of the rule engine, including a simple application with a rule editor UI and the capability to input a shipping request for processing. We were ready for the first experiments with business users.

Step 4: Verify the model with business users by performing the experiment and refine the model as needed

The first “victims” were our solution manager colleagues, who had already spent a good deal of time with the end customers during several traditional requirements gathering sessions for the rule engine. After some hard discussions and several iterations, we were ready to go to the customer site — after only 10 working days of development!

We installed our prototype on the laptop of our solution manager and did two customer site visits. In the first visit, we talked to the IT department only. We processed some shipping requests with some example rules they gave us. One example required a new attribute. While our solution manager was talking, we implemented it within 2 minutes![9] They were very impressed and wanted some other attributes. Within an hour, we developed the rules they had in mind without disrupting the ongoing discussion.

Now we were ready to visit another customer,

[9] To be honest, we expected to do this and we were well prepared!

where the meeting included both the IT department and business users. We started the meeting with a demo of the HTML mock-up of our xCarrier shipping manager application. The IT people were not very interested, but they asked a lot of technical questions concerning the integration into different systems and so on. This was the part that the business users didn't like. As soon as we came to the demo of our rule processing engine, we asked again for some example rules — and there was no feedback. The shipping managers told us they did not need any rules! After some time, the IT people told us that up to now, it's been too expensive for them to build rules, because they had to write an ABAP function module. So the business users had worked for years without rules, because the IT department was not able to write ABAP coding.

We started to show both parties our prototype — and suddenly a shipping manager asked: "Oh, you can do this with your rules? We could use this for our shipments to Spain." And one IT person answered, "But you need the attribute xyz!" We quickly implemented the attribute and showed them the new rule. With the prototype, they were able to experiment with the rules and to find *their* rules. It was the first time that we were able to target the IT and the business people at the same time and a good example of how xM breaks down barriers of communication between developers and end users.

As a last step, we integrated the Smalltalk rule engine with our pilot customer environment.[10] Finally we had our prototype running together with an SAP ERP system. Because we used Web services, it was not suitable for high-volume shipping, but it was more than sufficient for our pilot customer scenarios. During the complete customer evaluation, we had no major changes in the rule engine — apart from a lot of new rule attributes and some bug reports. There were no big surprises, no landmines, and the customer got what they expected, thanks to the experimental, iterative approach of xM.

Lessons learned

As in the first case study, we had really good results in an amazingly short period of time — it took much longer to organize the customer visits than it did to implement our model. By using xM:

- We were able to verify the requirements for the rule engine on the customer site *before* we implemented the solution.
- We got more precise requirements for our rule engine.
- We were able to choose the correct rule engine.
- Our pilot customers got what they wanted — although they were not aware of what they really needed.
- Because the customers got what they wanted, we had no major changes in the rule engine during the pilot phase.

Tips for effective model building

Traditional "waterfall" development approaches, where developers convene a couple of one-hour meetings with end users to identify requirements and then hole themselves up in their cubicles for a months-long development cycle, often result in expensive "landmine" change requests in the implementation phase resulting from poorly qualified requirements. Barriers to communication between domain experts and developers cause expensive, frustrating changes late in the development cycle that could have been averted with the more iterative, interactive approach

[10] This was the most time-consuming part — the technical connection between a Smalltalk system and an SAP system. But, based on our lessons learned, the vendor of the Smalltalk system decided to support the connection via a tool.

suggested by xM. By putting developers in the shoes of the end users, and forcing them to build models using the plain-language terms their end customers use, critical gaps in requirements, and root causes of the problems with existing applications, can be revealed early on. xM exposes users to prototypes much earlier in the process than traditional approaches. This creates an engagement and a rapport that helps to defeat some of the most frustrating aspects of application development.

In our Duplicate Analyzer case study, we showed how by getting developers and end users on the same page — getting developers to speak the end user language — we identified the root cause of the fundamental problem. In our second case study, we brought a prototype implementation to the customer site within days of beginning the project. We put IT and end users in the same room, and used our prototype to show them how easily they could overcome previously insurmountable obstacles and to create a valuable dialog. Even if our prototype leveraged components that wouldn't ultimately be scalable, and its integration with SAP NetWeaver was at the time only a prototype, we nonetheless demonstrated an approach to model development that could deliver meaningful business value at the end of the implementation cycle.

There are a few rules that make xM more than just simple prototyping and greatly add to its value:

- Use business terms that domain experts understand when creating domain terms to avoid miscommunications.

- Don't bother end users with technical concepts like entities and services in the early phases of model development because the users cannot relate to these distinctions.

- Model processes and use cases in colloquial language. The program model should make sense when read aloud.

- Choose the simplest possible prototype implementation to illustrate the processes and control flow, resisting the temptation of efficiency considerations or geeky frameworks that can alienate end users and lead to missed assumptions.

- Extend the pure model implementation with clearly separated additions that facilitate experiments with the model, like GUIs, interfaces to existing systems, and mock-ups.

- Run a continuous feedback loop between experiments and the current model documentation. Remember the output of a cycle is a model document, not the actual implementation.

Following these simple rules helps to reduce the level of formality and abstraction that is too often used by developers, and keeps the model documentation in terms and concepts the end user can understand, facilitating an iterative approach that surfaces missed assumptions before they become costly changes. There is no "breaking" of the end user's concepts for the representation of the model in documentation and code. You can still embrace UML for the documentation of models and designs between developers, but use plain business language when finding and communicting about models with end users.

Conclusion

Exploratory modeling is a powerful methodology for interactive finding, communication, and the experimental verification of software designs. It's based on the experimental implementation of a model in a non-technical, barrier-free, metaprogrammable language (Smalltalk, in our case) that allows fast iterations and refactoring.

Using xM, we were able to gain more precise requirements from the unspoken experience of the functional experts. It helped us to get a better understanding of the domain as well as of the

Some frequently asked questions we've encountered about xM

- **Does this really work for you?**

  Yes it does!

- **Does the fact that Smalltalk requires its own environment bother you?**
- No it doesn't! We were actually able to go to our customers with executable prototypes. Try that with a complete ABAP installation!

- **Do you see an overall improvement even when you re-implement a component for productization?**

  Yes of course! Look at the xCarrier example. Although we had several rule engines, we were not able to choose one, because we didn't know the exact requirements of our customers. In the end, we found that limitations in the existing components, not the needs of the customer, were driving the requirements. We spent only two weeks of implementation with two resources, and we were able to quickly identify the true requirements of our customers and choose the right rule engine.

- **Will you use this for all projects?**

  No, we won't, but wherever we have unclear and complex requirements in well-separated components, we will use it.

- **Will other teams at SAP do the same?**

  Unfortunately we can't yet answer this question. We talked to some colleagues and they were interested, but they probably need some more time to be convinced. Maybe this article is the first step!

requirements. We fixed some key problems and were able to improve and accelerate our work process. It was the first time that we were able to verify the requirements *before* the testing phase and *together with* domain experts as well as our pilot users. And our accounts payable clerks were satisfied too — we found a new duplicate invoice.

For our group, it is a new kind of problem-solving technique for certain kinds of application problems. So we will not use it to build interactive HTML mock-ups for a data entry application. We will use it whenever we have a complex requirement like the invoice similarity comparison or the shipping request rule engine and we don't feel that we're getting exact requirements from the functional experts.

Although we don't plan to deliver our Smalltalk prototypes in a real product, our group will continue to use Smalltalk as a tool for xM to learn about new domains more quickly and to reduce the risk of failure for critical components.

Heinz Roggenkemper is an Executive Vice President of Development at SAP Labs, LLC, and is responsible for the Business Process Renovation team, which strives to improve business processes by renewing them — making changes to something that already exists, leaving its essence intact, and giving it new vigor. Heinz joined SAP AG in 1982 as a systems consultant. From 1986 to 1987, he worked at SAP International in Switzerland, followed by two years in the United States as the first president of SAP America. In 1990, he returned to SAP Walldorf's development organization and was responsible for the development of application link enabling (ALE) and Internet applications until 1996. Most recently (until the end of 2003), Heinz served as the managing director of SAP Labs North America, where he was instrumental in growing SAP's development activities in North America.

Ralf Ehret is a development architect at SAP. He joined SAP in April 1998 as a developer working with HR and Workforce Management software before moving on to the Business Process Renovation Team. One of his first challenges was to evaluate Smalltalk as a tool for rapid prototyping. Together with consultants from Georg Heeg eK, he has successfully completed two Smalltalk projects at SAP. At the end of these two projects, Ralf became the main SAP contact during Cincom Smalltalk's VisualWorks-to-SAP integration project.

Andreas Tönne is Lead Consultant at Cincom Systems, where he is responsible for the Cincom Smalltalk-related service business as well as partner development in the European market. Andreas coined the name "Exploratory Modeling" and wrote the first articles about this discovery as an answer to the question, "Why is Smalltalk so successful in projects like at SAP?" He was a consultant and co-developer in the two xM projects at SAP that are described in this article. His work on this article was mainly done while working with Georg Heeg eK.