Agility in Model-Driven Software Development? Implications for Organization, Process, and Architecture

Hans Wegener

Swiss Re, Mythenquai 50/60, 8022 Zürich, Switzerland

Phone: +41 (43) 285 27 28, Fax: +41 (43) 282 27 28, E-Mail: Hans_Wegener@swissre.com

Abstract: Agile and model-driven development exhibit different constraints with respect to how products should be developed that make the two difficult to reconcile under all circumstances. Model-driven development is most affected by a lack of guaranteed congruence between model and implementation, which can be the result of an agile approach. On the other hand, the model-driven approach tends to defer or complicate feedback, which is critical to agility. If a combination of both approaches is desired, these are some measures: equal importance for forward and reverse engineering, flexible merge and diff support, lightweight modeling languages and core assets, intensive use of interpreter technology, explicit consistency management and structured handling of inconsistencies.

1. Introduction

Agile and model-driven development can be considered *en vogue*. Why not put the two together and form something even better? Reconciling both approaches requires understanding their structural differences; this position statement analyzes those differences.

Agile development is not based on a rigid execution structure but malleable values. Values are put into practice by following a set of principles that are adapted to the specific needs of the environment. The Agile Manifesto [1] values

- individuals and interactions over processes and tools,
- working software over comprehensive documentation,
- customer collaboration over contract negotiation, and
- responding to change over following a plan.

As one instantiation of an agile methodology, Extreme Programming [1] exhibits the following practices: planning game, small releases, metaphor, simple design, testing, refactoring, pair programming, collective ownership, continuous integration, sustainable pace, on-site customer, and coding standards. It sees the room for agility in "small-to-medium-sized teams developing software in the face of vague or rapidly changing requirements." Cockburn [1] defines agile as "being effective and maneuverable."

In model-driven development as exemplified by the Model Driven Architecture [13, 14] we see a different focus. Models are built, viewed, and manipulated via UML; transmitted via XMI; and stored in MOF repositories. From there, software can be generated, or the models interpreted, both either in part or in full. Modeling and generation happen in one domain, implementation and execution take place in another. We refer to them as problem and solution domains, respectively [6]. The bottom-line benefits of the self-described "CIO Problem Solver" are:

- reduced cost throughout the application life-cycle,
- reduced development time for new applications,
- improved application quality,
- increased return on technology investments, and

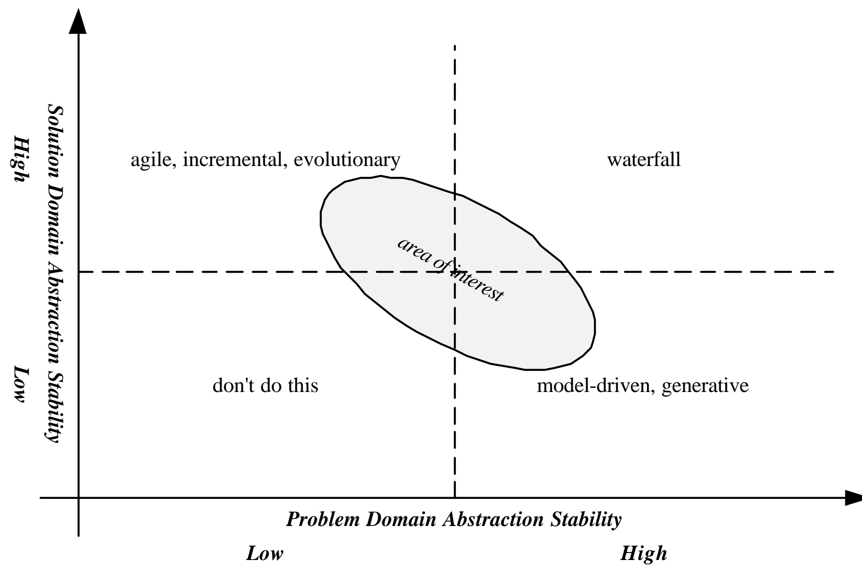

Figure 1: *Domain abstraction stability requirements of agile and model-driven development. The quadrants are not scrictly separable, the approaches overlap in their coverage of the continuum. We are specifically interested in the overlap between those located in the top left and bottom right quadrants.*

- rapid inclusion of emerging technology benefits into their existing systems.

A model-driven architecture is a type of architecture that features two layers of distinct nature for the purpose of increased ease of evolution. Modeling is performed using the problem domain model; the abstractions found at this layer provide a very stable environment and change slowly. The solution domain is the substrate layer for actual implementation; it can (but does not have to) change quickly. Providing a mapping from problem domain to solution domain abstractions facilitates evolution of the problem domain. If the configuration of problem domain abstractions changes, the mapping takes care of it; if the solution domain abstractions themselves change, only the mapping must change, but not the problem domain configuration.

2. Comparing the Approaches

At first sight, agile development and model-driven development seem to address the same problem, namely accelerating development. However, it is not straightforward to compare the two. First, agile development concentrates on individual software products, while model-driven development is concerned with product lines, i.e. mass-produced software. We have to make explicit, which lifecycle we talk about: The lifecycle of one individual product or the lifecycle of the entire product line, including all products developed thereunder? Second, agility mostly addresses methodological aspects, while the model-driven approach is more concerned with architectural issues. We must make explicit, if and which aspect of either approach affects aspects of the other and where they are independent. Third, as it turns out agile and model-driven development address different combinations of problem domain and solution domain abstraction stability (Figure 1). However, we have to stress the fact that this picture is not black and white. Overlaps occur, and in reality one will often find a blend of both.

In our comparison we shall distinguish between the individual product's lifecycle and the lifecycle of an entire product line. This way it will be easier to tell common issues from ones only arising in a particular context. Also, we shall take a methodology-centric perspective. This is to accommodate the fact that, as will be illustrated soon, in model-driven development there exists some body of methodology [4], while architecture plays only a minor role in agile development theory [1, 2, 5]. Finally, we will identify a conflict of interests between agile and model-driven development, but are still interested in their reconciliation under premises to be named. We shall take on a black-and-white position at the start and later focus our interest on the area where both approaches can be applied.

Agile development is a way of handling so-called wicked problems [16]. These types of problems exhibit volatility predominantly in the problem domain, or the understanding of it. The approach acts along the dimensions of organization and, to a lesser degree, process. It values individuals and interactions over processes and tools. While agile development indeed promises to speed up the development process, this is secondary to being flexible in changing the course a project takes. More specifically, the "highest priority is to satisfy the customer through early and continuous delivery of [...] software." [1] The method values working software over comprehensive documentation. As such, complete specifications, proper definition of abstract syntax and behavioral semantics of an application are only a means to an end, if they are formulated at all. The Agile Manifesto claims that "the best architectures, requirements, and designs emerge from self-organizing teams." An agile project will usually not exhibit a lengthy formal requirements capture phase, even though there will be some effort spent to gather a reasonably good understanding of the problem domain. To the contrary, it is maintained that for wicked problems it is (at least) unrealistic to (at most) impossible to understand the problem domain in its entirety. In fact, some problem domains change while the specifications are written.

Given the high demand for flexibility in the problem domain, the solution domain must be tightly constrained in order to meet the requirements. The Agile Manifesto mandates that "simplicity is essential." Tools, architecture, design, and implementation are subjected to a lightweight regime: what isn't absolutely necessary is left behind. The rationale is that anything heavyweight will slow down the readjustment of the project after a shift in understanding. A corollary is that the solution domain must be very stable, i.e. a firm ground to maneuver on. If the solution domain isn't stable enough, effort is spent on mastering it, which increases the latency between formulation of the problem domain specifics and their implementation in the solution domain. However, latency reduces the feedback frequency, which diverts the project team from its most important task: mutual understanding between the customer and the development team.

In model-driven development, a number of steps have to be taken before any product can be developed. This can only pay off if the initial investment of core asset development is leveraged many times. In fact, the OMG [15] names the following scenarios for appropriate use: application reuse and retargeting, domain concept reuse, or interoperability. A critical element is the availability of core assets that were developed with all these scenarios in mind. Hence, the core assets developed during model-driven development are usually more generic than the artifacts used in agile development, which is yet another distinction.

People are not an explicit feature in model-driven development. The specification of a development problem is not discovered, it is created as the primary artifact. This claims that the problem domain can be or is already understood well enough, and that both developer and customer have exactly the same understanding. Then, the specification of the problem is modeled explicitly (e.g., in the form of UML, PIM, PSM). This is contrary to the agile development principle that favors face-to-face communication over formalization.

While the OMG says nothing about the development process, model-driven development exhibits a preference. For this purpose, we distinguish three different types of model-driven development:

1. Conceptualization: An object-oriented domain model is designed in UML and tossed later; the software itself is implemented by hand.
2. Blueprint: An object-oriented domain model is designed in UML; a blueprint (or framework) of the software is generated from that, the remaining pieces are programmed by hand.
3. Specification: An object-oriented domain model and the execution specification are designed in UML; the software is directly generated from there with no manual programming involved. Available core assets [4] are selected using predefined types, possibly using tagged values and stereotypes from profiles, c.f. [17].

Going from top to bottom the list exhibits an increasing degree of rigidity. While it is perfectly possible in a conceptualization scenario to implement the solution before the modeling artifact is created, this is already compromised in the blueprint scenario, because the remaining pieces cannot be created independent of the framework reference. This goes so far as to rule out any implementation work in the specification scenario. Obviously, this constrains a developer's (or client's) ability in terms of when he or she will do what. The permitted task sequence is restricted ranging from moderately to absolutely.

To conclude, model-driven and agile development address different dimensions of the spectrum (Table 1). It is important to remark here that, of course, this is a black-and-white scenario. However, it shows that the two

	Agile Development	*Model-Driven Development*
People	Have highest priority; interactions between customer and developer are facilitated.	No explicit role; problem domain model takes role of discussion platform for customer and developer.
Process	Has medium priority; ensures consistency of results of interactions between people; incremental, evolutionary.	No explicit role; strong tendency towards waterfall, less incremental processes.
Technology	Has lowest priority; only a means to an end; must be as simple as possible.	Is at center of approach; problem model (e.g., PIM) is manipulated, stored in and exchanged between repositories. Can be used to generate software (e.g., using a PSM).
Model	Secondary artifact; only produced up-front when absolutely needed.	Primary artifact; source of generated implementation.
Software	Primary artifact; sole measure of progress.	Secondary artifact; depending on solution domain, provides or adds aspects not covered by specification.

Table 1: *Distinguishing general aspects of agile and model-driven development.*

exhibit different structural characteristics. It is therefore indicated to examine the chances of reconciling both and examine the consequences.

3. Reconciling the Approaches

3.1 Interests Addressed

Agile development claims that when the problem domain is volatile or not well understood, the interaction between customer and developer must be at the center of the approach. In accordance with Dzida and Freitag [1], we understand the problem of validating analysis and design as an incremental, evolutionary process of negotiation and finding mutual agreement on the basis of the realized software. The *artifact in vivo* is the only concrete object based upon which customer and developer can establish congruence and validate their understanding of both the problem and the solution.

Agile development also maintains that the process comes second to people and interactions. This explicitly includes the ability to change the process when the necessity grows from experience. The task sequence and workflow required to develop the software cannot be predicted and may not be restricted in advance. What is required is that the technology and the process allow for such changes. The corresponding principle is *controllability* [9], which demands that "the user is able to initiate and control the direction [...] of the interaction until [...] the goal has been met." Controllability places full control of the task sequence and workflow in the hands of the individual trying to achieve the goal.

Agile development is the constant attempt to maintain simplicity. This is to react flexibly to changing demands. Anything in excess of what is absolutely necessary would be considered a burden. We need *suitability for the task* [9], i.e. to support "the user in the effective and efficient completion of the task." An object-oriented model is only used if it can be used as a good representation of the problem domain. Domains that exhibit other characteristics, e.g. relational, logical or functional, are used when more appropriate. A design is *refactored* [8] as soon as it no longer adequately represents the understanding of the problem, specifically when the same things are done in different places.

Model-driven development is based on the design of a software system in UML. The modeling language provides assets from the problem domain to be composed and further parameterized. Generative techniques map them onto the solution domain, possibly making use of platform-specific peculiarities. Thus we achieve *congruence* between problem domain (design) and solution domain (implementation). The *artifact in vitro* is subjected to validation, not the generated product. Mutual understanding between customer and developer is

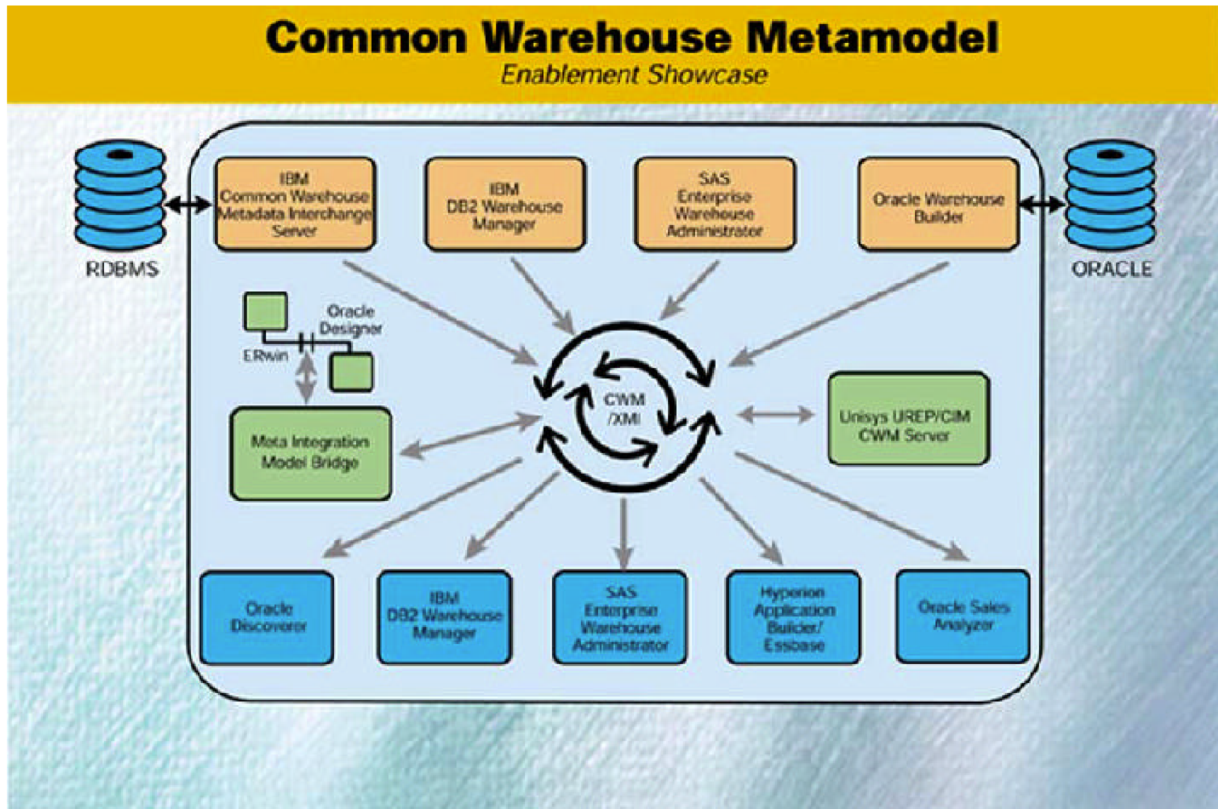

Figure 2: *The OMG vision of the MDA, illustrated with the CWM. Source: OMG.*

established at the design level. Furthermore, the tasks are *sequentialized.* The production process is designed up-front. First comes understanding, then realization, not vice versa.

What the MDA envisions is depicted in Figure 2: Software is designed in UML (or CWM, for that purpose) and stored in a MOF repository, which acts as a hub. Implementation code is generated from there. Metadata is fully exchangeable between tools from different vendors. Most important of all, the solution domain is not in the picture as a source. Reverse engineering, other than forward engineering, plays no prominent role in the development vision of the OMG.

When agile and model-driven development are compared, it becomes clear why they are difficult to reconcile. Complementing and extending earlier results [18] and building on our own experience with the CWM [19], the problems are elaborated in the following section.

3.2 Issues Arising Across a Single Product Lifecycle

In the single product lifecycle scenario, a product is developed on the basis of an already established product line. We assume that a good understanding of the problem domain has been established, though it is still volatile (to a certain extent). The core assets have already been developed. Based on these premises, the following issues arise:

- Unless the software is fully generated from a UML specification, manual work will have to be performed at the solution domain level. If the development process is expected to be controllable, it may not be rigid. That, however, will eventually result in *incongruences* between model and implementation.
- The solution domain may not permit the implementation of certain models. Especially when COTS components are used in the solution domain, they are often not as open as they should be [11]. There exists a *reverse influence* from the solution domain to the problem domain, something that is not part of the MDA model. It is assumed that learning does no longer take place.

- UML and CWM are *not suitable* for several modeling problems. In particular, the abstractions provided are sometimes inadequate, too fine grained or exhibit a wrong level of abstraction [19]. Extensions and adaptations may be required, causing additional effort.

The first two issues can be arranged around the way *feedback* is dealt with. In model-driven development plus agile development, two different layers are the subject of manipulation. If the feedback from the artifact in vivo conflicts with the assumptions made during modeling, we have to manage them. This, however, is either not part of the model-driven approach (dealing with reverse influences) or actually challenges basic assumptions (congruence between model and implementation). The last issue is concerned with the adequacy of UML-based languages themselves.

3.3 Issues Arising Across the Entire Product Line Lifecycle

Model-driven development revolves around flexibility based on properties of the software architecture, which are established in a series of steps. Clements and Northrop cite the following tasks as essential: core asset development, product development, and management [4]. The core assets (product and production constraints, product line scope, production strategy, etc.) are developed "to establish a production capability." In this scenario we start from scratch to establish this capability and, based on it, develop at least one product. We assume that no understanding of the problem domain has been established yet and that it is volatile (to a certain extent). The core assets have not been developed, and finally, we assume that the cost of identifying the right core assets outweighs the cost of developing the first product quickly.

- Understanding the problem domain takes time. That means that there is *latency* between the beginning of the development effort and the specification of the target UML model. It defers feedback in vitro, which is crucial for the learning process.

Again, feedback is an issue. While in the previous scenario the interaction between problem and solution domain played the dominant role, we here are concerned with the timing. In a situation where the problem domain is not yet understood and partially volatile, the critical ingredient in developing is attaining feedback rapidly. This, however, is thwarted by the initial effort required to set up production capability.

4. Implications for Organization, Process, and Architecture

The reconciliation of agile and model-driven development is not straightforward. How should this affect organization, process, and architecture? In order to become more agile, the challenges to be mastered by model-driven development include the expectation to maintain congruence, the sequentiality of task arrangement, long(er) latencies, and the reliance on standardized, general-purpose metamodels. We think these issues can be overcome:

- First and foremost, requirements are not developed, they are discovered. This means that design (problem domain) and implementation (solution domain) will be out of sync until right before deployment. This highlights the prominence of continuous exchange between the two levels. *Forward and reverse engineering attain the same importance*.
- *Flexible merge and difference tools* for managing the consistency gap are a must. However, since both problem and solution domain may evolve at the same time, this integration problem takes on a new level of complexity. Particularly, reverse engineering and merging will often fall together. It will become necessary to flexibly associate solution domain artifacts with problem domain artifacts and track changes on both levels. Different levels of granularity must be offered (c.f. [10], and the next item).
- The integration of problem and solution domain artifacts is more difficult to master with a general-purpose modeling language. In order to correctly interpret changes in either domain in light of the other, we need an *understanding of concurrent modifications in both domains.* We must know which modifications typically happen at one domain and which corresponding changes in the other they represent. This understanding, when turned into a model, will be semantically rich and very domain-, possibly even project-specific.
- *Consistency must be managed explicitly.* During development, the assumption that problem and solution domain (understanding) are aligned with each other, will no longer hold true. Architecture and development

process must be able to *handle inconsistency in a structured fashion*. For Nuseibeh at al. [12], inconsistency management is risk-driven and has a process and a technology component. To ensure convergence of problem and solution domain, consistency management has to be a continuously managed part of the development process. Divergence between problem and solution domain artifacts must be averted, which again calls for proper diff tool support and an understanding of typical changes. Nuseibeh at al. even go as far as to say that "some inconsistencies never get fixed." As a result, *traceability may be limited.*

- *Interpreter technology speeds understanding.* Riehle at al. [17] point out that the latency between formulation of a requirement and its implementation can make exploration of the possibilities awkward if not impossible, thereby slowing down the learning process. Their solution is interpreter technology based on a virtual machine for a domain-specific language (in their case, though, based on UML), which allows for more rapid prototyping.
- The development process must become more iterative. This means that artifacts must be generated from problem domain models that do not reflect the entire solution, but parts of it. Comprehensive, complex core assets with many dependencies to other assets are less suited. To a certain degree, this also means dismissing comprehensive domain modeling. Things will work better with *thin, lightweight core assets*. The preceding means also that complex, general-purpose modeling languages are less suited than *lightweight modeling languages.*
- The *organization should start co-located, non-hierarchical, and exhibit shared ownership.* Of the organizational alternatives discussed by Bosch [3], especially the development department is the soundest. It realizes the ideal of co-location, shared ownership, and exhibits a lack of up-front, comprehensive domain modeling (c.f. XP's practices collective code-ownership, pair-programming, and customer on-site), all in the interest of accelerating the learning process. Hierarchical and snowflake-type organizations should be avoided. Scaling and evolving the organization takes it through the state of a business unit with mixed responsibilities, finally arriving at assigned asset responsibilities.

It is important to note that model-driven development isn't the only approach with interests different from agile development. Any software development method based on the idea of problem domain models will fall prey to the problem of mutual understanding between customer and developer once things become complicated or volatile. However, the question is to what degree that matters. There are problem domains that are so simple or so well understood that it is indeed viable (if not outright imperative) to apply model-driven development. On the other hand, some problem domains are so complex or so volatile that an agile development project would be able to finish before one could even understand the entire problem domain in the first place. Finally, it should have become obvious that a software solution addressing all of the above concerns will be very complex (thick on architecture, thin on modeling, could be the motto). This underlines the fact that it is crucially important to understand the characteristics of problem and solution domain, and to make an informed decision as to which course to follow.

5. References

1. Agile Alliance: The Agile Manifesto. Available from http://www.agilemanifesto.org
2. Kent Beck: Extreme Programming Explained. Reading 2000 (Addison-Wesley)
3. Jan Bosch: Software Product Lines: Organizational Alternatives. In Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001),
4. Paul Clements, Linda M. Northrop: Software Product Lines. Practices and Patterns. Reading 2002 (Addison Wesley)
5. Alistair Cockburn: Agile Software Development. Reading 2001 (Addison-Wesley)
6. Krzysztof Czarnecki, Ulrich Eisenecker: Generative Programming. Reading 2000 (Addison-Wesley)
7. Wolfgang Dzida, Regine Freitag: Making Use of Scenarios for Validating Analysis and Design. IEEE Transactions on Software Engineering 24(12):1182-1196, December 1998
8. Martin Fowler: Refactoring. Improving the Design of Existing Code. Reading 2001 (Addison-Wesley)

9. International Standardization Organization: ISO 9241, Part 10. Ergonomic Requirements for Office Work with Visual Display Terminals (VDTs). Dialogue Principles, 1996
10. Christine M. Neuwirth, Ravinder Chandhok, David S. Kaufer, Paul Erion, James Morris, Dale Miller: Flexible Diff-ing in a Collaborative Writing System. In Proceedings of the 1992 ACM Conference on Computer Supported Collaborative Work (CSCW 1992), 147-154
11. Linda M. Northrop: SEI's Software Product Line Tenets. IEEE Software, July/August 2002, 32-40
12. Bashar Nuseibeh, Steve Easterbrook and Alessandra Russo: Making Inconsistency Respectable in Software Development. Journal of Systems and Software 56(11), November 2001
13. Object Management Group: Model Driven Architecture. November 2000. Available from http://www.omg.org/mda
14. Object Management Group: Model Driven Architecture. July 2001. Available from http://www.omg.org/mda
15. Object Management Group: Executive Overview. Model Driven Architecture. Available from http://www.omg.org/mda
16. Mary Poppendieck: Wicked Problems. Software Development Magazine, May 2002
17. Dirk Riehle, Steven Fraleigh, Dirk Bucka-Lassen, and Nosa Omorogbe: The Architecture of a UML Virtual Machine. In Proceedings of the 2001 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2001), 327-341
18. Hans Wegener: Generative Programming and Incompleteness. OOPSLA 2001 Workshop on Generative Programming. Available from http://www.generative-programming.org/oopsla01-workshop.html
19. Hans Wegener: Adoption of the Common Warehouse Metamodel at Credit Suisse. 4th Workshop of the Competency Center Data Warehousing 2, University of St. Gallen. Hamburg 2001