The Pitfalls of Meta-Systems and Business Rules

White Paper

Wolfgang Keller
c/o Generali Office Service und Consulting AG, Kratockwjlestr. 4, A1220 Wien, Austria
Email: 100655.566@compuserve.com
http://www.objectarchitects.de/

Abstract

As people find out that object-orientation and rapid development alone are often not fast enough in order to adapt software to changing environments and ever changing business processes they turn to meta-systems and try to define business rules in a declarative way. This paper discusses an array of typical design flaws and problems in such systems in order to help you avoid the pitfalls described. Most of the pitfalls described in this paper also contain possible ways to refactor a system once you have identified the problem.

Introduction

Today businesses have to react faster and faster to change in markets, organizations and business processes. Not so long ago software people would recommend you object-oriented development in order to deal with such challenges. After people have seen, that even object-oriented development and rapid development are by far not fast enough to tackle the speed of change in today's economy, people start building meta-systems[1] and using business rules in order to keep up with the speed of change.

Acolytes of business rules might tell you the following story: "If you use our business rule engine, most business logic can now be formulated as business rules – programming is replaced by writing of natural language-like business rules. Once you have installed the system, your business people will write their rules themselves. Therefore you can react to changes much faster and cheaper."

This message works especially well with senior management and has caused serious waste of money and human resources, if implemented naively. Proper use of business rules and meta-system technology may speed up your development cycle. Improper use of business rules will result in too complex and too expensive systems that roll out even later than a conventional system would roll out and that almost nobody can maintain and master.

---

[1] For a description of this category of systems see the Reflection Pattern in [Bus+96]

Copyright © 1999 by Wolfgang Keller: Work in progress – Submission for GCSE99 - Do Not Distribute or Xerox.

As there's not much practical experience around with systems that cover object-orientation, meta-systems and business rules all at the same time, this paper will help you by providing a collection of pitfalls that you can avoid, as other people before you have already shot themselves in their legs with this new silver bullet.

The paper uses a form similar to Anti Patterns[2] [Bro+98] as these provide a structured way to describe pitfalls and negative experiences, because they allow you to relate one misfortune with other similar design flaws in an organized way, and because they are fun to write – and we hope also fun to read.

Forces Driving the Design of Very Flexible Systems

As a running example for a domain that uses meta-systems we will use insurance systems. Because the insurance industry faces many of the challenges mentioned above and also because there are design examples in pattern form (e.g. [Kel98a]) that may be used to contrast the pitfalls to follow. In order to have a base for the later discussion of benefits and drawbacks of certain solutions and software designs, we need to give you a briefing on the forces that make people implement meta-systems or rule based systems and also on the counter forces that you have to deal with, when implementing such solutions.

The main forces that drive the implementation of meta-systems are ***Time to Market and Flexibility***. Companies that are able to design and market more products faster and are also able to offer individual products have a competitive advantage over companies that need years to bring a new product to the marketplace. But there are far more forces involved:

New and individual products**:** Michael Porter distinguishes between several competitive strategies [Por85]. You can become a cost leader, a quality leader or a niche player. If you want to open new market segments, or if you want to become a leader in customer perceived quality, it is helpful to be able to have many individual products for many customer groups. In times of early industrialization this has been a contradicting goal to cost effectiveness. With the advent of computers, individual products at low costs become feasible. This movement is called mass customization [Pil98]. You need to combine elementary standardized building blocks to receive individual products. Companies who manage to provide individual products have a serious competitive advantage.

Point of Service and Cost Cutting: There are several ways to react to more competition. One way is to provide service closer to the customer. Many systems e.g. in the financial industry today are still pure back office systems with a lot of paper flowing between several layers of regional offices and the sales representative. Cost cutting strategies aim at this paper flow and also aim at reducing process steps. Insurance systems are deployed closer to the customer, calling for Internet and offline processing and the sales representatives' laptop becomes closely integrated with the central transaction system.

Development Cost: Many businesses today provide pure immaterial services – no hardware – no

[2] For an explanation of Pitfalls please see [Bro+98]

Copyright © 1999 by Wolfgang Keller: Work in progress – Submission for GCSE99 - Do Not Distribute or Xerox.

product you can touch. An insurance company for example that can slash data processing costs is like a manufacturing company that reduces its labor and material costs. The result is a competitive advantage. Therefore total relative development costs for information systems need to be lowered while providing more functionality and flexibility at the same time.

Performance: Maybe the most important counter force for flexible systems is performance. The better you know what will happen in your system, the better you can optimize it for performance. It is the basic idea of flexible systems that you do not know what will exactly happen with the system at build time. So you have to take special care of performance issues.

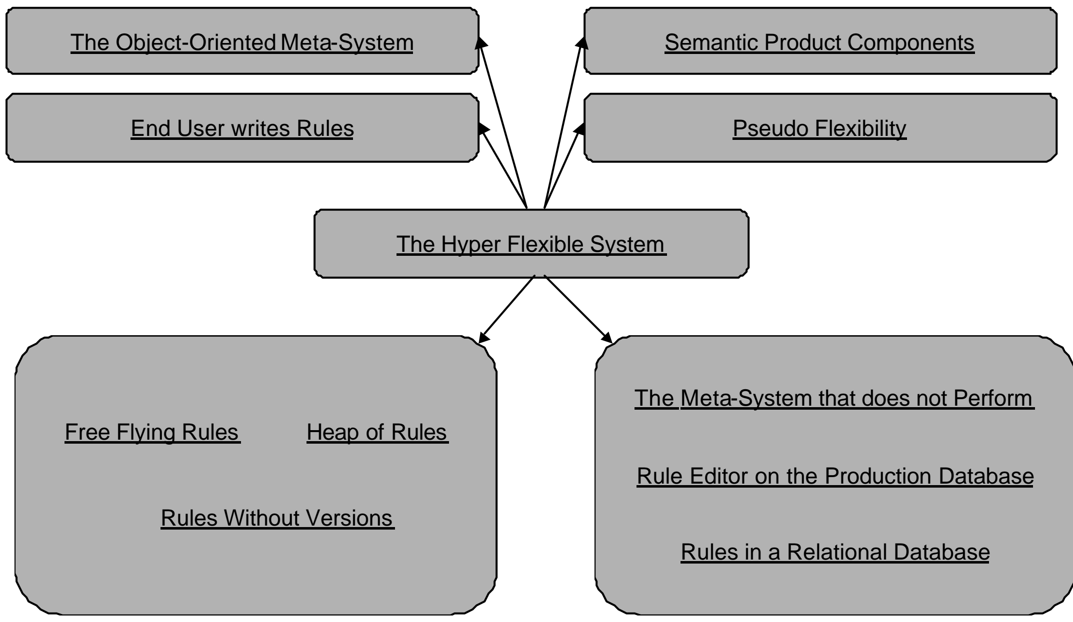
Maintainability: Another counter force is maintainability of the resulting system. This might seem irritating at first glance. You build a flexible system for shorter turnaround times – isn't that good maintainability. Maybe it is – but with a complex set of rules, it might turn out, that the problem of mastering a large set of rules, which are essentially programming statements – only without a sequential order – may choke you.

Good Software Engineering Practices: stand for good maintainability, low cost and understandability – people did not invent modularization and encapsulation for nothing. We will see that some flexible systems go too far at violating principles of good object-oriented software engineering.

A good solution will balance these forces in order to get a viable compromise. The solutions we will present in the pitfalls to follow will in most cases ignore some forces, leading to a far less than optimal solution.

Roadmap

The following is a roadmap of the pitfalls collected in this paper. Most problems stem from the quest for The Hyper Flexible System. This leads to two major blocks of problems plus a few common, but independent design flaws. The left block (Free Flying Rules etc.) is concerned with problems of manageability of a flexible system. The right block (Rules in a Relational Database etc.) deals with common performance trouble in flexible systems.



Copyright © 1999 by Wolfgang Keller: Work in progress – Submission for GCSE99 - Do Not Distribute or Xerox.

Before We Start: Some Remarks about the Known Occurrences

A Pattern is only a pattern if there are at least three independent *Known Uses*[3]. Transferred to pitfalls, this means that a pitfall is only a pitfall if there have been at least three projects falling into it independently. As people sometimes react very negatively if you publish their problems, we have only inserted references to known occurrences in our own company in order to hurt nobody. This does not mean that only the Phoenix project, which has been cited explicitly as a known occurrence from time to time did step into the pitfalls to follow. It only means that we did not name the other projects explicitly, that we have seen having similar trouble, It also means that we know our problems and are thus able to deal with them.

The Pitfalls

Pitfall: The Hyper Flexible System

Also Known As:

a variant of Overdesign,

or as a subtype of Death March

Context:

Imagine you are in an organization that thinks of a completely new software system in order to deal with flexibility requirements as outlined in the general forces section above. The organization suffers from an inadequate legacy system for years and is starting a large program in order to beat the complete competition with a new system in a single giant entrepreneurial effort.

The project is planned and everybody is bringing in his or her requirements and most important, expectations of the future, and future markets and products.

General Form:

People will then start writing very thick concepts, and specify many many features (known as Analysis Paralysis [Bro+98]). The solutions designed are far from easy to understand.

Typically there are numerous sets of multicolor slides that explain the advantages of the new system at a marketing level, but code is scarce, design concepts are hard to find. If they can be found they are incomplete and hard to understand. If you find code, it is low quality prototype code.

[3] This is called Buschmanns Law in the pattern community.

Copyright © 1999 by Wolfgang Keller: Work in progress – Submission for GCSE99 - Do Not Distribute or Xerox.

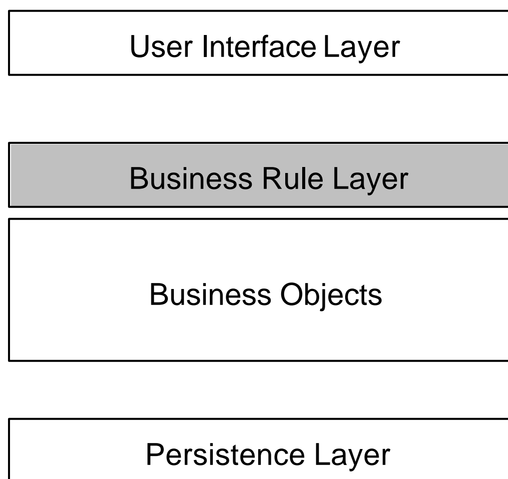


Figure 1: Typical piece work by a “multicolor slide architect”: Pictures like this should make you ask questions about the exact power and functionality in the Business Rule Layer. Often you will get answers like: We don’t know yet – but we add rules to have enough flexibility and will deal with that at customizing time. Such answers should make you very, very suspicious and cautious.

The project typically has enormous delay and is a hundred or more percent over budget. Users complain that they do not get what they did order. After the project management finally manages to get the system productive, some people check what portion of the features that provide flexibility are really used. A short calculation brings up the result, that hard coding would have been orders of magnitude cheaper and that many of those who required maximum features in the requirement phase are not using them and worse: are not even qualified to use them. The system has severe performance problems and it takes heavy redesign efforts for it to recover from that.

Typical Causes:

Typical causes for such projects are:

?? project goals are far too ambitious,

?? people try to develop an abstract and too generic software platform instead of working on a concrete solution for a single customer first – this may also be caused by Design by Committee [Bro+98].

?? no architecture, lack of a highly skilled software architect, or an architect who never wrote too much code.

?? not enough technical understanding and project experience in project management.

?? corporate politics.

?? User who try to stay with their old system by building up requirements impossible to fulfil – once you fulfil them, they can’t use them.

?? new technology or even worse – a whole bag of new technologies.

?? ignoring risk management practices

?? late or no feedback from end users

?? not applying the 80/20 rule, making thinks to complex and caring for too many exotic cases instead of keeping things simple.

Copyright © 1999 by Wolfgang Keller: Work in progress – Submission for GCSE99 - Do Not Distribute or Xerox.

Implications:

The implication is typically a death march project – see Yourdon's excellent book "Death March" [You97] for a description in all possible detail.

The trouble usually comes from forgetting the software engineering and project management side of forces and over emphasizing high level business aspects.

Known Occurrences:

There's plenty of examples in [You97].

Known Exceptions:

A Hyper Flexible System is o.k.

- ?? as long as the other competitors in the market are not better at software development than you are
- ?? as long as your organization is still able to benefit from the flexibility and make substantial money.

The Hyper Flexible System is also o.k. if money does not matter. There is examples of whole Death March cultures [You97], especially in the financial industry where there is evidently more money than elsewhere.

Refactored Solution:

Avoiding the problems of the Hyper Flexible System is mostly avoiding the typical causes. The Anti Patterns book contains even more. Let us emphasize a few points that are straightforward. A good medicine is having a look at eXtreme Programming methods [Bec+99]. What this tells you is stay close to code, develop the software you need, avoid too much paper and prefer code instead, communicate closely with your customer and lots of other basic concepts that make you concentrate on the essentials: software.

You might also use some of the concepts described as Meta Object Protocols or Open Implementation Guidelines [Kic+99]. These are also far from giving up any form of modularization and clean design.

Related Patterns and Pitfalls:

As you can see in the above roadmap there's a whole bunch of typical detail problems that people build into Hyper Flexible Systems. The fixes for this will be discussed with the particular problem.

Pitfall: The Object-Oriented Meta-System

Context:

You are brought into a project that has the goal to produce a very flexible system and should use the latest in technology. A few years ago this meant object-orientation. Now it could be components. Often, people who are dealing with design here are not familiar with the theoretical concepts of meta-systems. Some are using object-technology (or whatever is the new technology of the year) for the first time and some still believe in naive use of inheritance, in flexibility of object-oriented systems without effectively applying too much of theoretical concepts like encapsulation, assertions and all the like.

General Form:

You interview people from the architecture group and they are telling you, they are building a purely object-oriented system for reasons of flexibility. On the other hand they are telling you, they are using a pattern like Policy as Product Instance [Kel98a] and that they are building a rule system in order to make the thing even more flexible and adaptable.

But what they are really building is a meta-system, while thinking that they are building an object-oriented system.

Typical Causes:

A typical cause for this is lack of knowledge about theoretical background of software design, patterns and programming practices. There is no such thing as an object-oriented meta-system if you're not building an object-oriented programming system (like e.g. Smalltalk) of your own.

- You should either build an object-oriented system based on rules of good OO-design
- or you can build a meta-system, built according to rules of meta-system design. You may build this system in COBOL, C or any OO language. Does not matter too much.
- But you should not build another programming language in order to provide flexibility on top of another flexible OO-language like Smalltalk without even doing it intentionally.

Typical reasons for this are:

- programmers who are in love with technology and like to build language systems, interpreters, parsers, and tools.
- lack of theoretical background knowledge in computer science.
- an architecture group that does not deeply reflect the requirements and does not talk to users. They also often do not prevent programmers from building too complex things.

Copyright © 1999 by Wolfgang Keller: Work in progress – Submission for GCSE99 - Do Not Distribute or Xerox.

Implications:

The implications of this pattern are:

?? very complex systems, that are so complex that even your toughest cowboy programmers do not understand them any longer,

?? no real business value, as every business decision is left to later rule definition – all the user gets is an empty “knowledge definition shell”,

?? high costs and late delivery, if the system is delivered at all.

?? often combined with bad performance.

The forces out of balance here are as good as all forces, as such system are neither well engineered nor do they provide good business value.

Known Occurrences:

There’s more than one occurrence in more than one financial services company.

Known Exceptions:

None. Simply don’t do it.

Refactored Solution:

If you have a flexible OO system like Smalltalk, forget about that extra rule engine and meta stuff and do the job straight coded.

If you really need a meta-system, build a meta system. But apply rules of meta-programming in the first place and rules of OO design later. Doing some stuff in a object-oriented way is nice to have but not a necessary precondition. Have a special deeper look at performance issues, if you follow the meta path.

Related Patterns and Pitfalls:

Meta-systems are described as the Reflection pattern in [Bus+96]. The pattern often occurs in combination with the Hyper Flexible System.

Copyright © 1999 by Wolfgang Keller: Work in progress – Submission for GCSE99 - Do Not Distribute or Xerox.

Pitfall: Free Flying Rules

Also Known As:

Breaking Object Encapsulation

Context:

You come to review a team that is implementing the Hyper Flexible System. In order to provide flexibility the designers have decided to implement a rule system. On the other hand you are told that the system is implemented according to the latest technology, using an object-oriented implementation language.

General Form:

You find a rule system that allows the user to define arbitrarily complex rules that may take any methods of the object model as parameters. These rules fly free above all objects and allow the user to define arbitrary rules and business logic in the customizing phase of the system. The architect argues that this is a must have requirement to provide the flexibility the users are asking for – e.g. new products in 3 days.

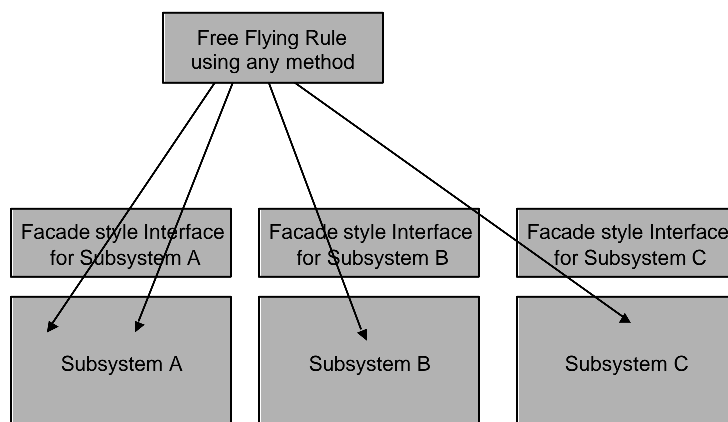


Figure 2: Free Flying Rule – The rule may use any method of the entire object model as a parameter regardless of interfaces, facades, and modularization considerations

Having a closer look at this kind of a rule system you find, that the rules need access and have access to almost all object variables of the object model. You do not find any structure of subsystems or facades [GOF95] in the system. Having facades would be impossible anyway as rules are allowed to inspect any object's state.

With a sufficient number of rules implemented you find that these rules are essentially snippets of code on top of the objects – whenever somebody changes an object's interface, he might break one or a few rules but there's no language system or compiler that raises an alarm. Whenever somebody changes a rule he may influence the behavior of all the clients and spots that are interpreting rules in the system.

What you get is exactly what people try to avoid with object-oriented systems. No encapsulation

and no modularity but everybody is allowed to peek into everybody's secrets here.

Typical Causes:

The main reasons for the occurrence of this pattern are the same as for the Hyper Flexible System.

Implications:

If the pattern occurs together with a Heap of Rules and Rules Without Versions your system might do things that nobody really understands anymore. A combinatorial explosion of possible causes for bugs will make testing expensive and insecure. Your system might become instable.

The forces violated here are the software engineering side of the equation again: Simplicity, maintainability, and good testability to name a few.

Known Occurrences:

There are commercial vendors that offer rule systems for free flying business rules and there have been projects that used them. Phoenix did use Free Flying Rules in past designs, causing a lot of trouble in testing even though the number was well below a hundred. The negative experience brought us to significantly reducing the power and use of such a rule system and to confining it to the Product Server [Kel98a].

Known Exceptions:

You should better not use rules that fly absolutely free in your system in parallel to object-oriented or procedural code. If the rules are constrained to very narrow context, like e.g. a product definition and are used more like a derivation method than like something that flies around and fires at will, this is at least better – in this case it is also good to confine rules within a subsystem in order to preserve the benefits of components, encapsulation, and facades (see Figure 3).

Refactored Solution:

The typical method to refactor a system with free flying rules is to rewrite it so that the rule system becomes obsolete. Rules will be hard coded in subsystems or commands[4] that span subsystems.

[4] Today the command pattern [GOF95] is frequently used to implement complex business transactions on top of multiple business objects – such command are typically invoked from a user interface or also within a batch run.

Copyright © 1999 by Wolfgang Keller: Work in progress – Submission for GCSE99 - Do Not Distribute or Xerox.

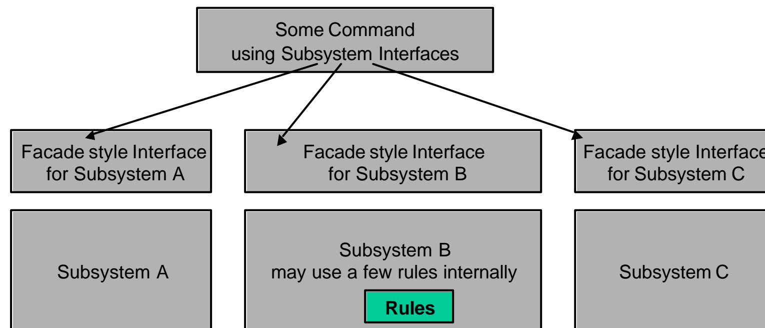


Figure 3: A rule based solution that may be better conforming to good design practices. Rules are confined to a single subsystem. If there is functionality, that needs to span more than one subsystem, it is wrapped into a command [GOF95]. The command uses well defined subsystem interfaces.

In any case rules should be confined to a single subsystem, where they may be tested below a well defined interface.

Related Patterns and Pitfalls:

The pattern often occurs together with Hyper Flexible Systems, Heaps of Rules and Rules Without Versions. It also occurs frequently together with the Object-Oriented Meta System.

Pitfall: Rules Without Versions

Context:

You come to see a project that implemented Free Flying Rules. In testing you have curious effects. Things and test cases that worked before will not work after a weekend. Your inquiries show you that no code has been changed over that weekend but somebody has inserted a few new rules. You have the Rule Editor on the Production Database.

General Form:

The problem of instability may arise if rules are not versioned like code is versioned. As we have seen in Object-Oriented Meta System, writing rules, especially Free Flying Rules is pretty much the same as writing code.

As with code, you need a very restrictive procedure to release rules, the same as you would release code. You also need to release the two in sync, as otherwise your rules might use code from a previous release that is not longer available in the current version.

The effect is worsened by the special behavior of Free Flying Rules.

Typical Causes:

The typical cause for this pattern to occur is that designers have ignored that rules are code, and that developers were told that rules are something a end-user may write and are not code. But rules ARE

code and all code needs to be versioned, at least in large scale systems.

To make things worse, designers have also forgotten about combinatorial explosion in testing needs, if you do not contain this property by introducing modules and testing them in unit tests before you start an integration test.

Implications:

The typical drawbacks of Rules without Versions are

?? instability of the system,

?? and much higher than necessary testing costs

Known Occurrences:

You can observe the pattern in many organizations who use product servers and rules. Prior designs of Phoenix suffered from the pattern.

Known Exceptions:

The pattern may be acceptable for systems that are written and maintained by a single person. It is acceptable for systems that are so small that you would not use version control anyway.

Refactored Solution:

There are two possible ways to refactor the solution

?? *Hard Coding*: Write your rules in your programming language and make them subject to the same versioning mechanisms as all your other code.

?? *Release Management for Rules*: Install a release management process for rules. This implies you need a separate version control repository for rules. You also need a development sphere, a test sphere plus a production sphere for rules. The transition from sphere to sphere needs to be defined and controlled. You need extensive testing, before releasing rules into production.

Related Patterns and Pitfalls:

Rules Without Version frequently occurs with Rule Editor on the Production Database and Rules in a Relational Database as versioning and organizational procedures to provide different spheres for programming, testing and production are some of the tougher topics to implement and people often ignore them when under pressure to finish a Hyper Flexible System that tends to be late and over budget anyway

Copyright © 1999 by Wolfgang Keller: Work in progress – Submission for GCSE99 - Do Not Distribute or Xerox.

Pitfall: Rules in a Relational Database

Also Known As:

Storing Trees in a Relational Database

Context:

You come to see a project that implements a Hyper Flexible System, or simply a meta-system. There are always to sorts of data in such a system

?? data that define the way processing is done: So called meta-data.

?? and the operational date that describe the live object instances of the system, e.g. policies, agreements, claims and all the like.

Rules are typically part of the meta-data. As the organization requires to put all data in a relational database, designers will also put all meta-data into the operational relational runtime database.

General Form:

Rules or any other form of tree like entities like product trees, product structures, CAD data are stored in a relational database. The best indicator for the pattern is serious performance trouble.

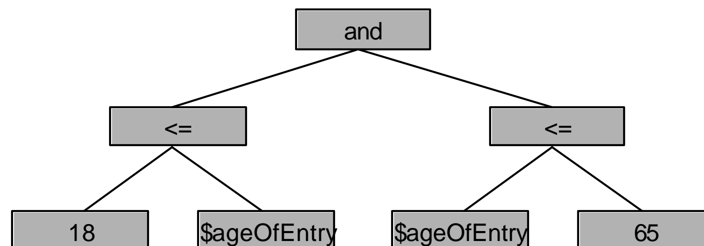


Figure 4: Example of a very simple acceptance rule for a life insurance policy. The ageOfEntry should be above or equal 18 and should not be greater than 65 years. This simple rule has seven tree nodes and leaves if you treat it as a composite – if you add navigation paths and the like you will have far more nodes.

A closer look will show you that trees (and rules are best modeled as trees – see Figure 4) are typically modeled using a composite pattern. Reading the tree from a relational database means roughly one database read operation per node or per leaf of the tree – this is simply too much as even simple rules come easily with at least around 10 nodes and leaves. Once you

Copyright © 1999 by Wolfgang Keller: Work in progress – Submission for GCSE99 - Do Not Distribute or Xerox.

need to read 50 rules you have 500 database accesses, resulting in a read time of about 5 seconds if you do not optimize for performance[5].

There are a few ways to improve this but the general performance behavior will remain unacceptable.

Typical Causes:

The typical cause for this pattern to occur is that people are using an object/relational database access layer or a relational database without bothering too much how data are physically stored in the database and without thinking enough about runtime performance.

Tree like structures are generally something that should be put under special scrutiny when you intend to store them in a relational database.

Implications:

The one and only implication of this is performance trouble. People who had implemented this pattern wondered about response times that were measured in minutes instead of in milliseconds.

This has resulted in more than one crashed project in the past because initial performance was too bad and investors ran out of money, patience, or both.

Known Occurrences:

There are quite a few adopters of well known insurance architectures who started off with the pattern. Phoenix used to be one of them. Same things have occurred, when people tried to implement CASE tools on top of relational databases or when people try to implement CAD systems on top of relational databases. This accounted for a few very prominent project crashes. All of these comprise extensive use of tree-like structures and all of these are endangered with suffering from performance trouble.

Known Exceptions:

You may use a relational database as long as you're only implementing prototypes of your future software and as long as performance does not matter in those prototypes. But before you use a relational database to prototype an object-oriented system it is more likely you will use an object database (OODB) for prototyping purposes.

---

5 Database experts might argue that you can refactor this to a single multiple read operation per rule by inserting something like a rule identifier and by retrieving all rule elements by issuing a statement like "select * from ruleTable where ruleTable.ruleId = <theRuleID>". This is true but still too expensive for large numbers of rules. You could also always read all the rule elements, regardless of whether you need them or not. Also works under some conditions. But working with the other refactored solutions is usually cheaper.

Copyright © 1999 by Wolfgang Keller: Work in progress – Submission for GCSE99 - Do Not Distribute or Xerox.

Refactored Solution:

There are at least three ways to refactor in this situation:

- ?? The cheapest one is to keep on defining rules on top of a relational database. The performance of the rule editor tool will be cruel – but as rule editing is not a matter of day to day work this may be bearable. Then use an object filer to file out all rules to a flat file (or BLOB in a relational database) and load all the rules into your main memory (by filing them in) before you start your processing. The result of this is that you will not need to access your relational database at runtime and your performance trouble at runtime of the system will be solved. The drawback of his is you use more memory.

- ?? The second way is you use some other formats than an object filer – e.g. a special byte code (see the Product Server pattern [Kel98a] for a description of this solution) that is executed by a runtime system.

- ?? The third way is you use an object-oriented database for rules and similar data. The drawback of this is the trouble you have from operating yet another database system in your organization and OODBs are not the easiest thing to deal with.

If you want to stay with your relational database for a while, you may use database performance optimization patterns (see [Kel98b] for a collection of patterns and more references) but be warned that this is only a short term solution that will only seldom yield satisfactory performance on the long term.

Related Patterns and Pitfalls:

Rules in a Relational Database can frequently be seen together with Rule Editor on the Production Database and the Meta-System that does not Perform.

Pitfall: Heap of Rules

Also Known As:

Large Scale Rule Based System

Context:

Again you come to see a team that implements a Hyper Flexible System. They want to make it *really* flexible so they come to the conclusion that it is hardly possible to implement any application logic with straight code. Instead they decide to put everything into rules, arguing that this is the only way to implement all the flexibility needs their users have.

Copyright © 1999 by Wolfgang Keller: Work in progress – Submission for GCSE99 - Do Not Distribute or Xerox.

General Form:

In systems like this everything is left to rules but nobody will be able to explain you how this will exactly work. In an early project phase you might be confronted with a view of the system like shown in Figure 5.

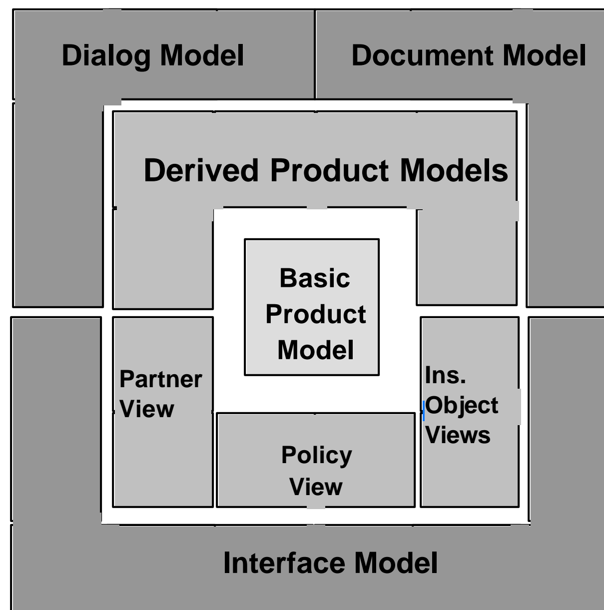


Figure 5: Typical example of a rules for everything. Model means “set of rules modeling X”. Everything depends on rules: the dialogs, the printed documents, products and also the behavior of other large components like the business partner, a policy system or an insured object system. Very efficient for sales events – but a nightmare if you try to implement it.

Even if you should ever be able to implement a system like this, the number of rules would get you. After first experiences with Prolog in e.g. systems for the assessment of medical risk in life insurance, people came to learn about the downsides of large scale rule oriented systems. Maintainability and testability. As one new rule may make a whole model, consisting of hundreds or thousands of rules, inconsistent, it is very tough to write, maintain and test such systems. Especially in testing you have problems with combinatorial complexity without the opportunity to contain it by modularization.

Typical Causes:

The main reasons are the same as for the Hyper Flexible System pattern.

Copyright © 1999 by Wolfgang Keller: Work in progress – Submission for GCSE99 - Do Not Distribute or Xerox.

Implications:

Typical downsides of an implementation like this are project delay and exceeded budgets due to:

- ?? bad maintainability
- ?? high testing costs and behavior that comes close to non-determinism, because ordinary people come to a point where they are no longer able to play blind chess against a rule interpreter algorithm and considerable number of rules.
- ?? instability due to lack of methods to cope with rule sets that range from a few hundreds to a few thousands
- ?? and bad performance due to the fact that the whole heap needs to be evaluated in order to detect the rules which are allowed to fire.

Known Occurrences:

You might understand that we will not cite examples we know outside of our own project.

Known Exceptions:

None – You simply better don't build a system like that.

Refactored Solution:

Refactoring such a system means throwing it away and replacing it with a well coded, modularized system that uses a few rules at some predefined and well testable spots. See Figure 3, page 12.

Pitfall: Semantic Product Components

Context:

You are reviewing a project that implements software to define insurance products. The project could as well deal with financial or other products. From a first glance at affairs, everything looks fine. Project members tell you they have used the Policy as a Product Instance Pattern [Kel98a]. They tell you their product server also implements a Composite Pattern [GOF95]. But after having seen the inheritance hierarchy of product components you feel somewhat discomforted. It has a depth of more than 10.

General Form:

What you see is a inheritance tree of specialized product building blocks. You find a elementary product component that is derived from a product component. There are things like sellable products, like special classes for death risk insurance, annuities, dread disease etc all modeled in a

Copyright © 1999 by Wolfgang Keller: Work in progress – Submission for GCSE99 - Do Not Distribute or Xerox.

giant inheritance tree.

Typical Causes:

Typical causes of this design are:

?? designers who are new to object-oriented concepts and think that using much inheritance is a good thing to do. See also Webster's account on the misuse of inheritance [Web95],

?? lack of knowledge of other concepts for product servers.

Implications:

If you use semantic classes to describe your product building blocks you can combine new products exactly from the classes you have implemented so far. If you need something you did not implement yet, you need to code a new semantic class. This is another case of Pseudo Flexibility.

A deep inheritance tree stands for poor maintainability due to the broken base class problem. Implementation inheritance is also seen very critical today as it will cause massive reuse problems.

Known Occurrences:

An older product server design at Generali uses semantic classes. We have seen other projects in the industry, that have similar designs.

Known Exceptions:

Using the pattern may be acceptable, if you are restricted in your flexibility by another part of the system anyway. In life insurance systems you will often find so called mathematical subsystem. You can only combine products from their basic component types anyway. So the restrictions imposed by the semantic product component classes (as in the above mentioned Known Occurrence at Generali) are not a real factor, if the corresponding mathematical subsystem needs programming anyway.

Refactored Solution:

You may use the Product Tree pattern instead [Kel98a]. If you use it, your product modeling system will be free of too many assumptions about the kind of products you are modeling. Drawback is, you need much experience in order to avoid effects as described in End User writes Rules.

Related Patterns and Pitfalls:

The pattern is yet another form of Pseudo Flexibility. You need to code whenever you want to implement a radical new product or concept. The pattern is also often related to the following Pitfalls of Object-Oriented Development [Web95]: *Using Inheritance Badly*, *Confusing is-a,*

Copyright © 1999 by Wolfgang Keller: Work in progress – Submission for GCSE99 - Do Not Distribute or Xerox.

has-a and is-implemented-using Relationships, as well as *Confusing interface Inheritance with Implementation Inheritance*.

Pitfall: Pseudo Flexibility

Context:

Somebody is presenting you his or her new system. The system is said to be very flexible because you have access to all business objects and you are able to formulate rules using all accessor-, get-methods, and whatever methods.

General Form:

What you find is a rule system (for example written in Smalltalk or Java) that derives values (expressions) from methods of the business objects as parameters.

Only problem is ... If you want to write a totally new rule and if you do not have the appropriate get-method to provide the rule with parameters, you end up programming get-methods if you want to define a new rule.

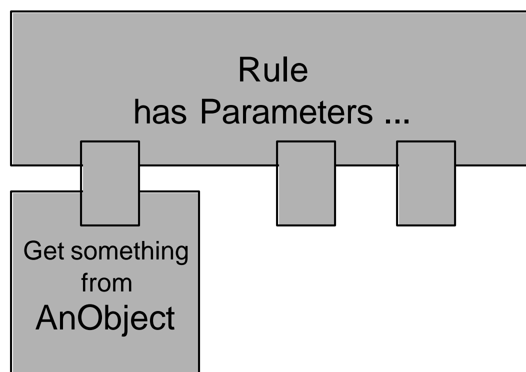


Figure 6: A Free Flying Rule is defined using get-methods as parameters. If you want to do something you need new get-methods for you will not be able to benefit from the flexibility of a rule system – you have pseudo flexibility as you have to program anyway.

But you wrote the rule to allow domain experts to write rules without programming. BINGO! You have to program to write such a rule and the whole flexibility is pseudo flexibility. You could as well have hard coded the solution.

The pattern is more a question of marketing. If you announce that you can handle 80% of rule definitions without programming and only the last 20% complex cases are left to programmers, this is something like honest marketing. If you announce that your users will be able to define even the most complex business logic in terms of rules, you will be confronted with providing pseudo flexibility.

Typical Causes:

?? Aggressive marketing for a Hyper Flexible System in order to justify cost,

?? belief in high tech solutions instead of keeping things simple.

Copyright © 1999 by Wolfgang Keller: Work in progress – Submission for GCSE99 - Do Not Distribute or Xerox.

Implications:

The one and only implication is, that you paid a lot of money for something, that does not have too much benefit over hard coding especially in Smalltalk. This is the best case – see Free Flying Rules for another category of drawbacks.

Known Occurrences:

Early versions of Phoenix Life were announced as absolutely flexible. After some time people found that they needed programmers anyway. But don't laugh at Phoenix – there's plenty of other examples of similar systems in the financial industry..

Known Exceptions:

Proper announcement of the kind of flexibility that you can achieve will prevent users from being disappointed.

Refactored Solution:

Use straight coding for all or the 20% of unexpected rules. If this is not flexible enough, you need to generate lots and lots of get methods resulting in a bss of object encapsulation, opening all your objects' interior open to the public.

Related Patterns and Pitfalls:

The pattern often comes together with Free Flying Rules. If it is allowed to use any methods in any rule (even methods with side effects) this makes things even nastier as you loose any control and understanding of your system.

Pitfall: End User writes Rules

Context:

After years you have shipped a Hyper Flexible System. You have done all your best and implemented Free Flying Rules and also implemented them as Rules without Versions. Then your users write their first rules and you wonder why your system is producing errors in production.

General Form:

Deeper analysis yields that your domain experts have trouble expressing their knowledge in rules. A closer look shows you that writing rules is pretty much like writing code. And even worse: When you give the same task of modeling product X to two people at end user department, they will typically come up with two very different product models, neither of them maintainable.

The problem behind this is that most rule systems offer you a shell and some syntax. But people who build them will only in very rare cases invest the time to provide enough demo product models and

modeling patterns that allow users of their rule system to write reasonable product or other models. There are only very few published examples of modeling patterns for product systems so far. Object Event Indemnity [Kel98a] is a widely used one, but there are far more that are yet waiting to be mined.

Some domain experts do not have concepts to express in rules, what they used to discuss with a programmer. And some do have a feeling that what they do is like programming under harder conditions, because large sets of rules are much harder to check than programs. And there is provably no algorithm to check rules automatically.

Typical Causes:

Typical causes are the same misunderstandings that lead to the Object-Oriented Meta-System. People want to provide systems that can be changed without coding and are only replacing one language with another one. Usually the new one is less tested, less documented and there has been by far less money flowing in the "rule system" than in the underlying programming language. Compare Smalltalk with some custom build rule editor – see the difference.

The pattern may also be credited in parts to the *Not Invented Here* syndrome. Why use Smalltalk if I can write my own language and my own programming tools instead of talking to users of my systems. Building tools is so much cooler.

Implications:

The typical downside is that the IS department will end up coding the rules anyway – the end user department will seldom be able to do it.

So the result is a waste of time and money in most cases. Time because the IS people could have straight coded the system an order of magnitude faster while they were writing their new languages and tools. Money, because all this translates into money.

Known Occurrences:

The pattern occurs with most of today's product definition and rule systems. Even the better ones. But vendors are beginning to understand that a product server or rule system needs to supported with training and brain ware in order to be really successful.

Known Exceptions:

If the end users are computer experts as well they could write their own tool for their own work. But if the end users are not proficient in writing programs, there are hardly any exceptions.

Refactored Solution:

As a first solution let the IS people write the rules together with the end users in a pair programming effort.

If the pattern occurs with other pitfalls like e.g. Heap of Rules or Free Flying Rules, consider a

redesign and hard coding of the rules.

Pitfall: Rule Editor on the Production Database

Context:

A project uses a rule system. They are telling you that they are very proud of their rule system that provides maximum flexibility and short reaction time to change because the rules are directly edited in the production database.

General Form:

This one is hardly to be distinguished from Rules without Versions and Rules in a Relational Database. The specific property of this pattern is that rules are edited in the production database. If the production database contains something like a development, a testing and a production space, at least one source of trouble is absent.

If the team also uses an object database to edit the rules, another source of trouble is not present – performance trouble.

But you often find Rules without Versions and Rules in a Relational Database combined in a single system which causes us to name it a separate pitfall.

Typical Causes:

The typical causes for this are a combination of the typical causes for Rules without Versions and Rules in a Relational Database.

Implications:

So are the typical drawbacks. Instability, plus bad maintainability, plus bad performance. Plus you are playing with a sharp knife in an open body. Every change may be lethal if production is hit.

Known Occurrences:

Prior versions of Phoenix suffered from this problem. Again – don't laugh at Phoenix – there's far more examples of projects who have not even mentioned yet that they will have the problem in a year so.

Known Exceptions:

No way.

Refactored Solution:

See the refactored solutions for Rules in a Relational Database plus for Rules Without Versions.

Copyright © 1999 by Wolfgang Keller: Work in progress – Submission for GCSE99 - Do Not Distribute or Xerox.

Related Patterns and Pitfalls:

The pattern often occurs with Rules in a Relational Database.

Pitfall: The Meta System that does not Perform

Context

You review a project that is said to have bad performance. You find out that people were trying to build a very flexible system (need not be a Hyper Flexible System but could also be something more reasonable). If they are good they will tell you that they were trying to build a meta-system.

General Form:

Most meta-systems are somewhat slower than their conventional, less flexible counterparts. But closer analysis shows you that the project uses a relational database, because a relational database has been prescribed by the IS operations department.

You also find that both meta-data and production data are held together in one relational database. You also find out that the structure of meta-data and the structure of operational data is pretty much tree-like (for an example see the pattern Policy as Product Instance [Kel98a] implemented in a way that does not take care of performance). We did already discuss this kind of problem with the Storing Rules in a Relational Database pitfall.

Typical flaws are too many joins or reads needed to build something like a policy or an agreement or also use of an object/relational access layer that stores one attribute per database row (I'm not kidding).

Typical Causes:

The typical causes are pretty similar to the problems we have seen in the Storing Rules in a Relational Database pattern. In both cases people store tree-like data in a relational database without a closer look at performance issues.

Another reason is use of an improper object/relational access layer like one who stores everything in a fixed number of tables with e.g. one attribute value per database row. Luckily these are dying out.

Implications:

As the pattern name indicates: performance that may well make a project fail.

Copyright © 1999 by Wolfgang Keller: Work in progress – Submission for GCSE99 - Do Not Distribute or Xerox.

Known Occurrences:

There's plenty of conventional systems that suffer from performance trouble in the beginning. Driving database normalization to an extreme can do much harm to performance. Most meta-systems that are implemented using a relational database suffer from the problem.

Known Exceptions:

A system that does not perform properly may be acceptable as a functional prototype for a limited period of time.

Refactored Solution:

As this pattern is mostly the final result of a combination of storing tree-like structures in a relational database (see Storing Rules in a Relational Database), or using a Heap of Rules the cures for this can be derived from a closer analysis using the above pitfalls.

Related Patterns and Pitfalls:

For a far from complete collection of good practices that will prevent this pattern from occurring see the patterns on building insurance systems [Kel98a] and patterns on object/relational access layers [Kel98b] which will provide you with pointers to more patterns and performance fixes.

Copyright © 1999 by Wolfgang Keller: Work in progress – Submission for GCSE99 - Do Not Distribute or Xerox.

References

[Bec+99] Kent Beck et al.: eXtreme Programming. For the latest scan the Wiki Web http://c2.com/cgi/wiki?ExtremeProgramming or Ron Jeffries' web site http://www.armaties.com/Practices/PracLeadin.htm

[Bro+98] **William J. Brown, Raphael C. Malveau, Hays W. McCormick III, Thomas J. Mowbray:** Antio-Patterns*; Refactoring Software, Architectures and Projects in Crisis*; Wiley 1997.

[Bus+96] **Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal:** *Pattern Oriented Software Architecture - A System of Patterns*, Wiley 1996.

[GOF95] **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides**: *Design Patterns, Elements of Reusable Object-oriented Software*, Addison-Wesley 1995.

[Kic+99] **Gregor Kiczales et al.:** *Open Implementation Design Guidelines*; see http://www.parc.xerox.com/oi/ .

[Kel98a] **Wolfgang Keller**: *Some Patterns for Insurance Systems*; Proceedings – PloP 1998; Washington State University; Technical Report TR #WUCS-98-25; see http://jerry.cs.uiuc.edu/~plop/plop98/final_submissions/; 1998

[Kel98b] **Wolfgang Keller**: *Object-Relational Access Layers* – Proceedings EuroPLoP 1998; see also http://www.coldewey.com/europlop98/Program/writers.htm, 1998

[Pil98] **Frank Piller:** *Kundenindividuelle Massenproduktion - Mass Customization: Die Wettbewerbsstrategie der Zukunft, mit einer Einführung von B. Joseph Pine II*, München/Wien: Carl Hanser Verlag 1998

[Por85] **Michael E. Porter:** *Competitive Advantage*; The Free Press 1985.

[Web95] **Bruce Webster**: *Pitfalls of Object-Oriented Development*; M&T Books 1995

[You97] **Ed Yourdon***:* *Death March, The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects*; Prentice Hall, 1997.