

A pattern language for BRMS development

Ian Graham

TriReme International Limited

ian@trireme.com; ian_grahams@hotmail.com

This paper presents some sample patterns from a candidate pattern language for developing business rules management systems (BRMS). In its present form, it covers patterns for requirements modelling and knowledge acquisition, patterns for discovering, writing, testing and managing rules, patterns for selecting suitable BRMS technology and a few process patterns. It does not reproduce the plethora of useful advice on writing rules given by Morgan (2002) and Ross (2003), to which the user of the language is referred. Nor is it complete in respect of software development process patterns; here I refer the user to the work of Barbara von Halle (2002) for a more complete treatment. The other reference which readers may wish to look at is Arsanjani (2000) who gives some design patterns for business rules at a lower level of abstraction from the technology than that aimed at herein..

I assume that my readers are familiar with the basic ideas of patterns and with the difference between a pattern catalogue and a pattern language. In particular, I have standardized on an 'Alexandrian' format in the hope this this will also be familiar. Most of the patterns are presented as short 'patlets' and some are summaries of patterns published elsewhere.

1 *The RulePatterns language – Part I*

The language presented here is divided into two major sections, represented by the two navigation charts of Figures 1 and 2. Within each section, the patterns are further classified according to their function.

Pattern numbering is continuous across these sections to emphasize its rather arbitrary nature. The patterns numbers have no significance. Each section starts with a map of that section of the language, which provides a high level overview of the section and provides primary navigation. The maps, although not the sections, may overlap slightly. In these maps, the patterns are classified into abstract, concrete and terminal patterns as shown by their colour coding.

Abstract patterns represent the codification of principles and are shewn in grey. There is not always a context for such a principle; its just always a useful one and informs the way downstream patterns are applied and interpreted. **Concrete** (white) patterns are patterns in the usual sense and we discuss their structure in detail below. Finally, some patterns are **terminal** with this language. Of course, abstract patterns are never terminal.

A pattern being terminal does not mean that design thinking stops with it – merely that the language considers the further design issues as beyond its scope or ambitions. The other cases where the language terminates abruptly usually concern areas of some complexity that, in my opinion, are deserving of a pattern language in their own right, as will be noted. Where such pattern languages exist, this is indicated diagrammatically by an unnumbered rounded rectangle with a dotted background and by a reference in the appropriate pattern text(s).

The simplest way to use the language is to consider pattern number one (ESTABLISH THE BUSINESS OBJECTIVES) first and then follow the links to the other patterns. It is best to have a concrete problem in mind when doing this. Eventually you will reach patterns that are terminal (represented in black on the diagrams). You should also try to construct sequences (or sublanguages) to deal with specific design problems or specific kinds of development.

Rules are made to be broken. The patterns in this chapter may be regarded as rules for successful design but it is better to think of them as providing suggestions, guidance and checklists of things not

to forget to think about. If you do find yourself treating the patterns as rules then pause. Always consider the likely effects of breaking the rules and ensure that you understand the rules that you are going to break and the justification for doing so.

Each pattern and patlet is presented using the same layout, semantic structure and typographical conventions. These are very closely based on the structure pioneered by Alexander *et al.* (1977). The pattern number and name are presented first followed, optionally, by a list of alternative names – all in a black header. The alternatives, if present, are labelled *aka* (also known as) in the same header. Next comes what many people call a **sensitizing image**: a picture or diagram concerning, supporting or illustrating the pattern. In many cases this has been omitted for brevity, especially in the case of patlets.

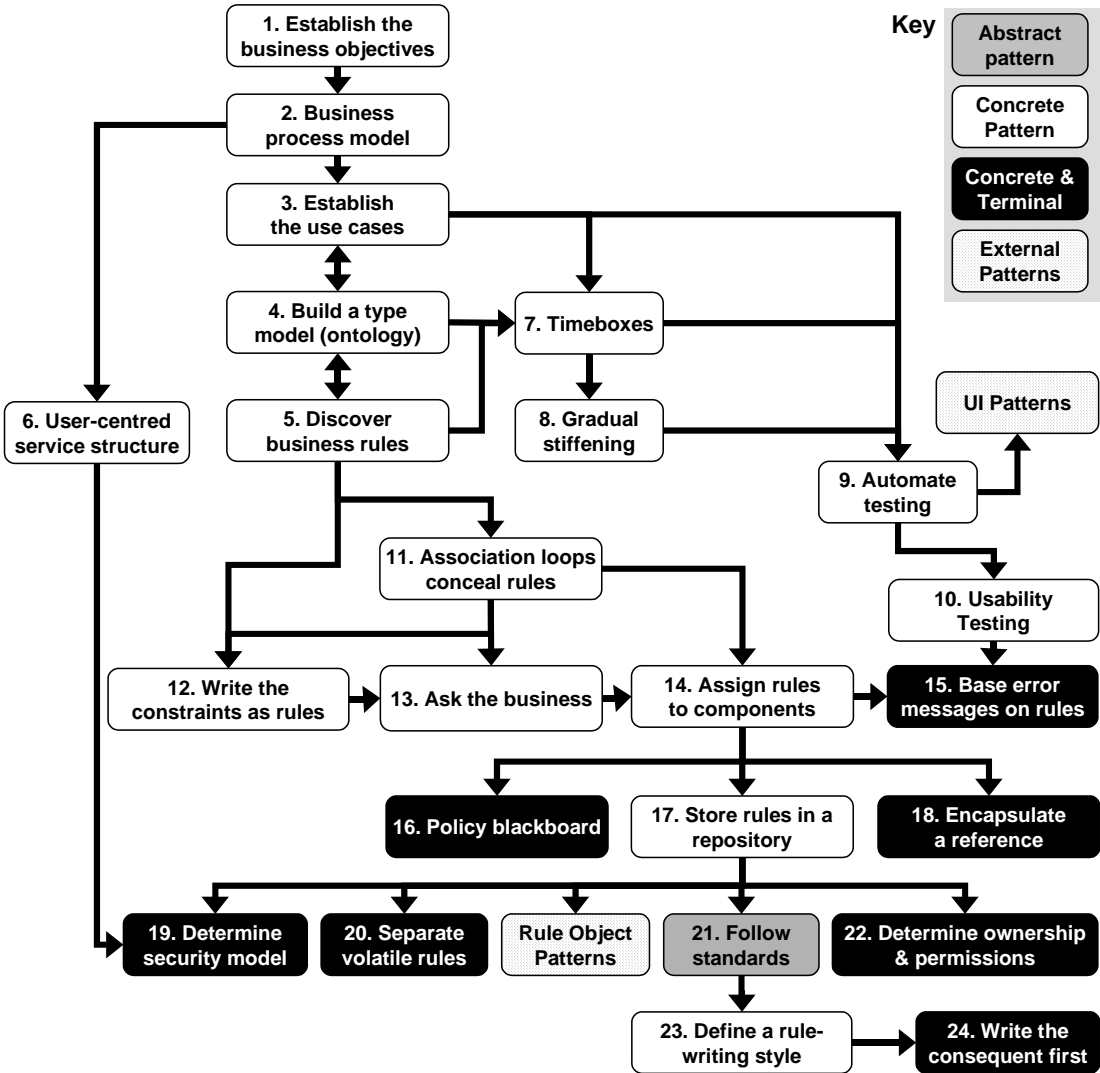


Figure 1 RulePatterns Part I

After the sensitizing image we present the **Context** in which one would normally encounter the pattern. This section usually gives the names of patterns that one has already used or considered. This is separated from the body of the pattern by three tildes, thus:

~ ~ ~

Next, the **Problem** is stated in bold text. For the discussion of the **forces** that are at work and the way the pattern deals with them we return to plain text; i.e. text of the sort you are reading in this paragraph. This section may include quite diverse types of commentary and explanations. Where

appropriate we highlight known uses of the patterns. Where this is omitted it is because the known uses are so obvious as to not need stating or because they have been intrinsic to the description of the forces and related discussion.

Once the discussion is complete, I state or summarize the recommended solution in bold text. This section is highlighted in the margin with the word **Therefore**. This completes the body of the pattern; so we again delimit it with three tildes.

The next section describes the **Resultant context** and, unless the pattern is terminal, will include the names of the patterns that one may consider applying next. This information is partly represented in Figure 1 by the arrows. Interpret these arrows as meaning ‘supplies a potential context for’.

Following Alexander again, I have classified the patterns according to my degree of confidence in them. The pattern’s ‘star rating’, shown next to its name, indicates this. Three stars means that I am totally convinced of the pattern’s efficacy, having used it or seen it used successfully on many projects. Three stars may also indicate that there is some solid theoretical justification of the pattern in the literature and folklore of the subject. If there are no stars it means that I think this is a good idea but would like people to try and see. One and two stars are interpreted on the scale between these extremes in the obvious manner.

The lengths of the patterns vary. Partly, this reflects knowledge and experience of the patterns and therefore confidence in them. However, sometimes it merely reflects the fact that they are easy to describe and understand. Some briefly described patterns are referred to as patlets. This indicates either that a longer version of the pattern has already been published or that an expanded version might be considered useful. In this text, the distinction between patterns and patlets is fairly fuzzy.

In Figures 1 and 2, rounded rectangles represent patterns and an arrow from pattern P₁ to pattern P₂ is to be interpreted as meaning ‘P₁ possibly generates a context for applying P₂ and indicates that the designer should consider applying P₂ whenever she has applied P₁’.

1.1 Patterns and patlets for requirements, process and architecture

Some of the following patterns apply equally well to projects other than BRMS projects; but they are important – and too often ignored – patterns, so I include them in the language, noting any BRMS-specific points.

Patlet 1	ESTABLISH THE BUSINESS OBJECTIVES ***
Context	You are embarking on a system development project that may or may not involve business rules.
	~ ~ ~
Problem	How can you be sure that the system will be fit for purpose and that project management can be both successful and agile in response to evolving requirements?
Forces	Many development methods encourage developers to start analysis with use case modelling or, at best, give little concrete advice on how to tie development to business goals. This can lead to dysfunctional results, unused systems or loss of focus during development. Furthermore, as requirements evolve during the project, there can be disputes over which use cases have priority for implementation before the next timeboxed delivery date. <p>When you decide to use TIMEBOXES (4) to control iterative development you can only negotiate sensibly on evolving requirements if you have consensus on the things that will <i>not</i> change during the project.</p> <p>Prioritizing objectives according to some scheme such as ‘Must have, Should have, Could have,’ often presents difficulties because stakeholders insist that their favourite objective is a ‘must’ until discrimination is lost completely and the priorities are worthless. An objective numerical ranking can be achieved by pairwise comparison of the objectives but this can be very time-consuming. It is quicker and</p>

just as practically effective to let workshop participants vote, preferably from two points of view.

Example For example, give each person red and blue stickers to the tune of two-thirds of the number of objectives and let them place the stickers next to the objectives on a flipchart. Red might represent the view of the organization, while blue represent individual (or departmental) preferences. Next one may add the results (blue and red) together and open a discussion to ensure that there is a consensus on the priorities so computed.

A longer version of this patlet can be found in (Graham, 2003a).

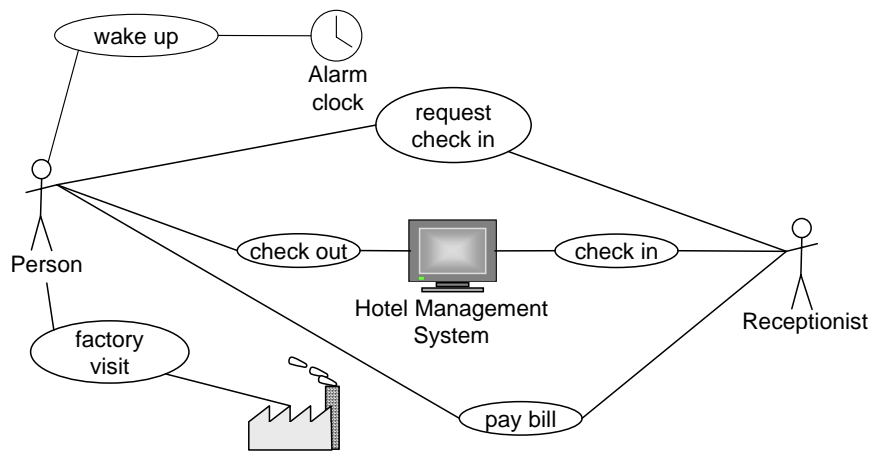
Therefore **Run a stakeholder workshop to establish the business objectives. There will typically be between 7 and 30 such objectives. Ensure each objective can be measured numerically and objectively; otherwise reject or reword it. Now assign numerical priorities to the objectives. The quickest way to do this is by voting and consensus-building discussion. Fix the objectives and priorities for the duration of the project.**

Involve as many stakeholders as possible. Make sure that potential users are represented. Find a good facilitator. Agree a mission statement to give context to the objectives.

~ ~ ~

Resultant context Once the objectives and priorities are fixed you can safely move on to construct a BUSINESS PROCESS MODEL (2). Refer to RUN A WORKSHOP (27).

Pattern 2 BUSINESS PROCESS MODEL **



Context You have ESTABLISHED THE BUSINESS OBJECTIVES (1) and fixed their priorities. You are probably also committed to a service-oriented approach.

~ ~ ~

Problem **How can you ensure that the development will take account of current or re-engineered business practices and procedures? The needs of stakeholders that are not direct users of the system must be understood – as well as those of the ‘actors’ in a conventional use case model. Do the processes contain any explicit or latent business rules?**

Forces The philosophy of service-oriented architecture emphasizes that our focus must be on the real user as well as the user who actually interacts with the system; a conventional use case model tends to focus on the latter.

There are two notational styles commonly used to represent business processes. In UML terms, we have activity diagrams or use case diagrams available. Activity diagrams are often useful but they can grow unmanageably large very quickly and they do not show ‘who does what’ very clearly; in that sense they are ‘disembodied’.

Use case diagrams tend to remain manageably concise and are ideal for emphasizing the contract-driven nature of business. Stating contracts for each conversation that occurs in a process can lead directly to statements of business rules. At a minimum, the pre- and post-conditions of the conversations (represented as use cases) always have a rule-like nature.

The image above is meant to suggest that actors in use case diagrams should be stereotyped to look like what they represent; if it is a factory, make it look like one. This helps communication in workshop situations and beyond.

This technique has been used successfully on hundreds of projects known to me around the world over thirteen years or so. A longer variant of this pattern can be found in (Graham, 2003a).

Known uses

Therefore

Draw a rich picture of the whole of each business process. Use cases can be used in this picture to represent goal-oriented conversations between actors (including users, non-users, events and artifacts). Ensure that you discuss possible changes to the process, leading to a ‘before’ and ‘after’ process model. Use stereotypes in the rich pictures that are meaningful to the business stakeholders present.

Understand first the network of agents and commitments that make up the business. Specify the conversations that take place at an appropriate level of abstraction, so that they are stereotypes for actual stories. Get people to tell these stories. Eliminate conversations that do not correspond to business objectives (or discover the missed objective). Ensure every objective is supported by a conversation.

Emphasizing the contracts that subsist in the business processes will assist in identifying business rules both now and later in the project.

~ ~ ~

Resultant context

Now ESTABLISH THE USE CASES (3) in the context of the proposed system and the business processes defined. Understanding who the real users are will provide the correct context for building a USER-CENTRED SERVICE STRUCTURE (6).

Patlet 3

ESTABLISH THE USE CASES ***

Context

You have constructed a ‘before’ and ‘after’ BUSINESS PROCESS MODEL (2).

~ ~ ~

Problem

How can you specify the behaviour of a system and the services it must provide?

Forces

Use case modelling is a very well-known technique. However, current practice tends to produce over-complex use cases with far too much detail. Jacobsen, Fowler and Cockburn all give ‘templates’ for use cases which exacerbate this tendency to ‘over-document’ although, to be fair, I think Cockburn did not intend to encourage their (mis)use.

Theoretically, use cases are completely determined by their pre- and post-conditions. There is really no need to specify them further.

The second problem with current practice is the use of the semantically ambiguous «extends» association, which tends to over-complicate and enlarge

models. It is better to dispense with it and define exception handling by separate use cases, to which error handling messages are delegated. (See Graham (2001) for details.)

Basically, use case modelling focuses on the functionality of systems as opposed to their data structure. Rule-based methods (e.g. Date, 2002; Halle, 2002) tend to start with the data model. The danger in that approach is that the service structure becomes data-centred rather than user-centred. Much experience says that a method that starts with functionality is both sounder and easier to understand.

Use case modelling is a well-established technique for systems development and is part of most mainstream methods for object-oriented and component based development. A longer version of this patlet can be found in (Graham, 2003a).

Known uses

Therefore

Extract the use cases from the conversations in the BUSINESS PROCESS MODEL (2). Write post-conditions for each use case. Compare the vocabulary of the post-conditions to the type model. Write use cases in stimulus–response form. Do not constrain the user’s ability to perform steps in any particular sequence.

Use cases are ideal for high level specification but they must be formulated at the right level of abstraction. Very detailed use cases are an impediment to clear understanding. To avoid superfluous detail, define use cases by ONLY their pre- and post-conditions and, if necessary, remarks on what will happen if there is a non-recoverable error during execution of the task. Write separate use cases to describe how to recover from other types of error.

Write explicit rules based on the pre- and post-conditions (the use case goals).

Ensure the use cases remain cross-referenced to the business objectives and that they inherit the priorities of the latter.

~ ~ ~

Resultant context

BUILD A TYPE MODEL (4) by explicating the vocabulary needed to express the pre- and post-conditions. If you have rules, ensure that they are executable and based on the type model. Group the use cases into sets that can be implemented together; base your TIMEBOXES (14) on these prioritized sets. Use the use case model to define tests and AUTOMATE TESTING (9).

Pattern 5 DISCOVER BUSINESS RULES **

Context You are building a business rules management system. You have ESTABLISHED THE USE CASES (3). You may have already BUILT A TYPE MODEL (4).

~ ~ ~

Problem **How can you discover business rules based on the type model? How can you find rules in other ways?**

Forces The forces at work here depend on how you have arrived at this juncture. Since you have a use case model, the obvious starting point is to rewrite the use case post-conditions as rules. For example, the post-condition of a simple business process such as a sale is ‘The vendor has the money and the buyer has the goods’. The corresponding rules might be written as follows.

A sale may be recorded if both of the following are true:

- The Vendor’s stock of money has increased by the price of the Good
- The Buyer has the Good.

Therefore **Rewrite the use case post-conditions as rules.**
If there is no type model at the outset, you must use various knowledge elicitation techniques to discover rules. These are covered by the following

patterns:

- PLAN INTERVIEWS (25)
- FILLED-IN FORMS (26)
- RUN A WORKSHOP (27)
- STRUCTURED INTERVIEW (31)
- FOCUSED INTERVIEW (32)
- PROBES AND TEACHBACK (33)
- ASK FOR THE OPPOSITE (34)
- BOUNDARY OF COMPETENCE (35)

If there is a type model then we may also proceed as follows. First rewrite any cardinality constraints as rules (WRITE THE CONSTRAINTS AS RULES (12)). Then realize that ASSOCIATION LOOPS CONCEAL RULES (11).

Now BUILD A TYPE MODEL (4) based on the terms used in the new rules discovered. Next apply:

- DETERMINE INFERENCE MODEL (36) if applicable.
- DETERMINE UNCERTAINTY MODEL (37) if applicable.
- CLASSIFY YOUR APPLICATIONS (38)

~ ~ ~

Resultant context This is a link pattern that leads to the patterns listed above, in the Solution.

Patlet 6 USER-CENTRED SERVICE STRUCTURE

Context You are building a rule-based application within a service-based architecture. You have defined a BUSINESS PROCESS MODEL (2).

~ ~ ~

Problem **How can you ensure that the services and components that are provided are pitched at the right level of abstraction?**

Forces All too often, developers focus their attention on implementation concerns and thus arrive at a design mindset at far too low a level of abstraction compared to the needs of business users. They focus on technical collaborations rather than business processes, which often do not – indeed cannot – involve computers. For example, a parcel tracking system needs to understand that a real person has to collect a parcel; the IT systems really can't do that.

The user that actually operates the computer is often not the 'real' user, in the sense of the person who gains the business benefit. Designing the system around use cases (in the conventional sense of actions at the system boundary) will lead to a system that does not serve the real users and whose services are not bundled appropriately for use.

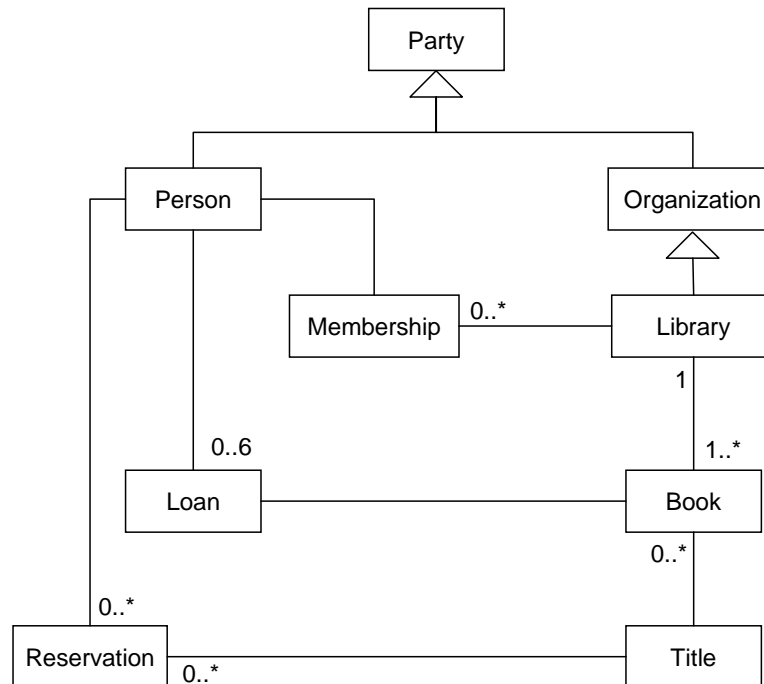
Therefore **Focus on the 'real' user and upon use cases that represent business process that may occur away from the system boundary. Focus on *what* users want to do, rather than how they want to do it. Where possible, capture this essence in the form of business rules.**

~ ~ ~

Resultant context Since you have a clear idea about who the various kinds of users are, both real and hands-on, you can now begin to DETERMINE THE SECURITY MODEL (18).

1.2 Patterns for finding, writing and organizing business rules

Pattern 11 ASSOCIATION LOOPS CONCEAL RULES **



Context. You are trying to DISCOVER BUSINESS RULES (5) and have completed part of BUILDING A TYPE MODEL (4). You know that you must WRITE THE cardinality CONSTRAINTS AS RULES (12).

~ ~ ~

Problem How can you be sure that you have not missed any rules implicit in the type model?

Example In the image above, start with a person. Do they have a loan? If yes choose one. Every loan is for a unique book that has a unique title. Does the title have an outstanding reservation against it? If yes, go back to the person you started with. Does that person have a reservation? If so, is it for the same title? Perhaps the rule is: 'A member may not reserve a title which they have already borrowed a copy of'.

Therefore. Look for cycles (loops) in the type diagrams. Start at every type in the loop, choosing a generic instance of that type, and follow the associations to another type. Ask if each route brings you to the same instance. Write down the rule that says it does.

~ ~ ~

Resultant context The rules you have written down may not be true, so now ASK THE BUSINESS (8) and then ASSIGN THE RULES TO COMPONENTS (9).

Patlet 12 WRITE THE CONSTRAINTS AS RULES *

Context. You have started to BUILD A TYPE MODEL (4) and noticed that ASSOCIATION LOOPS CONCEAL RULES (11). You may even have ASKED THE BUSINESS (13) and found that some of these rules are correct. However . . .

~ ~ ~

Problem Some rules are written as constraints in a style that does not fit into any

BRMS. It is unclear at this stage whether there is any interaction (inferencing) among the constraints. How can you clarify the situation?

Forces Writing rules in the style of constraints is useful if you want to rewrite them in OCL, as post-conditions in a language like Eiffel, using throw and catch in a language like Java or as database update constraints. On the other hand it may mean that there is a conflict of rule style with other rules elicited by other means (*c.f.* Part II of this language). Furthermore, it may be hard to see if there are inferential connexions between constraints.

Example Suppose we have the constraint ‘The pilot must be qualified to fly the type of plane assigned to the flight.’ Clear enough, but not written as a rule. Why not try this.

A pilot may be assigned to a flight if all of the following are all true:

A plane has been assigned to the flight;
The pilot is qualified to fly the plane type (of the assigned plane).

In this form it is much easier to see that inferences may be possible. Supposing we have other constraints that say, when written as rules:

A plane may be assigned to a transatlantic flight only if it is a Boeing 777.
A pilot may be hired only if she is qualified to fly Boeing 777s.

If we also know the fact ‘The flight is a transatlantic flight,’ (which may, in turn, be inferred from its origin and destination) then the original constraint may be *inferred* to be true, eliminating the need to check it directly in the database or prompt the user for information.

Therefore Rewrite the cardinality and other constraints as rules using a standard style or rule template. Look out for possible inference patterns.

~ ~ ~

Resultant context Now ASK THE BUSINESS (13) to ensure the rules are (still) correct and whether your discoveries about possible inferences are valid. If you haven’t done so already, DEFINE A RULE WRITING STYLE (23) and ensure that you have enforced it when using this pattern.

Patlet 13 ASK THE BUSINESS **

Context. You have discovered some rules, perhaps by exploiting the fact that ASSOCIATION LOOPS CONCEAL RULES (7) or by WRITING THE CONSTRAINTS AS RULES (10).

~ ~ ~

Problem. How can you be sure that the candidate rules are indeed veritable rules?

Forces. Having written a rule after much arduous analysis work, it is tempting to assume that it is true. This need not be the case. The temptation to make assumptions is very strong. For example, if air journeys have an origin and destination then one may jump to the conclusion that these have to be different. Indeed, mostly this is the case. But I remember the days when one could board Concorde at Bradford airport, fly out over the Atlantic for a rewarding sonic boom and then return to – yes, you guessed it – Bradford.

Therefore. A competent user or domain expert must verify every rule. Check also that any inference chains among rules are valid.

~ ~ ~

Resultant context. Now that you are confident that the rules are valid, ASSIGN RULES TO COMPONENTS (9).

Pattern 16 ENCAPSULATE A REFERENCE *

Context. You want to make your components and services as reusable as possible but you also need to maintain and manage the rules centrally. You have ASSIGNED RULES TO COMPONENTS (9). Rules that apply to more than one component have been assigned to the POLICY BLACKBOARD (11).

~ ~ ~

Problem. **How can you enforce rule encapsulation and not end up with a fragmented, unmaintainable rulebase?**

Forces. Reuse implies encapsulation, although it may be hard to decide where to put the rules. Opposing to this, rule independence implies a separate rule layer. If you encapsulate you lose rule independence; if you centralize you lose the potential benefits of component reuse. It seems to be a loose-loose situation. But there is a way out. Decide where the rules *should* go, but instead of storing the rules with the components to which they have been assigned, one can store a reference to these rules in the interfaces of objects that should encapsulate them.

Therefore. **Store the actual rules in the rule repository. When you create or specify any component, ensure that any rules associated with it are both stored in the repository and referenced in the specification and implementation of that component. Perhaps implement these references as methods that invoke the rules on the server. Do this also for the POLICY BLACKBOARD (11).**

~ ~ ~

Resultant context. This pattern is terminal within this language.

Pattern 18 POLICY BLACKBOARD *

Context You are trying to ASSIGN RULES TO COMPONENTS (14). In some cases this is easy but in others you face a quandary:

~ ~ ~

Problem **If a rule applies to more than one component or service, in which component should it be encapsulated?**

Forces Let's say that there are two candidate components: A and B; and that the rule talks about both of them. If you assign the rule to A then B ceases to be fully reusable because if you reuse it, its rules may be left behind. Assigning the rule to B causes the same problem for A. How about assigning the rule to both A and B? This would mean you have two points of maintenance for this rule. If you ENCAPSULATE A REFERENCE (16) and STORE RULES IN A REPOSITORY (17) then the maintenance problem goes away; A and B only *refer* to a centrally maintained rule in their interfaces. However, there may still be a conceptual problem.

Some rules may apply to several components and, as well as this, are naturally thought of as 'policy': policy that can change as the business evolves or at the whim of regulators or lawmakers. In such a case, there is an additional complexity in that the new rules may not correspond one-to-one to the old ones.

A policy blackboard is a central component designed to encapsulate such policy statement. Rules only encapsulate a reference to these rules *in the blackboard*, which in turn should ENCAPSULATE A REFERENCE (16) and STORE its RULES IN A REPOSITORY (17). In addition to this, each component sets up an OBSERVER to the policy blackboard. There are two versions of this. Either the publisher (the

blackboard) broadcasts all changes to rules in which interest has been registered to the subscribed objects, allowing them to update their interfaces or stored rules accordingly (if this can be done) or it merely broadcasts an ‘I have changed’ message, leaving it to the subscribers to decide whether to ask for more information and, indeed, what action to take. To distinguish these alternative architectures we may think of them as subpatterns: PUSH POLICY BLACKBOARD and PULL POLICY BLACKBOARD.

It is sometimes useful, when the rules are grouped into rulesets in a complex way for example, to segment the policy blackboard into pigeonholes that contain different kinds of knowledge. Subscribing components can register interest in whichever pigeonholes they need to know about.

Such an approach not only makes maintenance changes easier to understand at the business level and implement at the technical level, it also support a model of complex, coöperative decision making. For example, services implemented as intelligent agents can collaborate in applying the rules to a problem they face, sharing knowledge through the policy blackboard. In that case, the blackboard component may also need methods for handling a problem-solving agenda – which, in turn, may be rule-based.

The alternative to pigeonholes is to divide the rules among several policy blackboards according to the provenance of the rules; e.g. accounting rules, stock control rules, rules of engagement, etc. This approach makes the blackboards themselves more reusable (shareable) but may introduce too much complexity or overhead in agent-based applications.

‘The pilot must be qualified to fly the type of plane assigned to the flight,’ is a structural constraint. It is almost inconceivable that any policy change would reverse it – except perhaps in Alan Sillitoe’s (1971) fictional country, Nihilon. It is not, therefore, an obvious candidate for the policy blackboard, although it could just about conceivably be part of a ‘safety rules’ blackboard.

‘We may not fly more than twenty flights a week out of Bangkok.’ ‘Our share of transatlantic flights must not exceed 20% of total transatlantic flights.’ These rules look more like policy.

Example This pattern is a specialization of Buschmann, *et al*’s (1996) BLACKBOARD, which is, in turn, an architectural generalization of PUBLISHER-SUBSCRIBER (the GoF OBSERVER pattern).

Therefore **When rules refer to more than one object, consider encapsulating them (or references to them) in one or more policy blackboard component. This is especially indicated when there is a stated distinction between rules that are ‘policy’ and those which merely describe the structural relationships between objects.**

Resultant context ~ ~ ~ This pattern is formally terminal within this language, although it may be related to SEPARATE VOLATILE RULES (20).

Patlet 24 WRITE THE CONSEQUENT FIRST *

Context You are trying to DEFINE A RULE WRITING STYLE (23).

Problem ~ ~ ~ **How can you make executable rules easier for users and business analysts to understand?**

Forces The obvious and most general way to write rules is in if ... then ... form. This form translates directly into machine understandable languages and is friendly to

developers. It also facilitates the observation of inference patterns among the rules.

However, experience has shown that business users find it easier to articulate and read rules written with the outcome preceding the conditions. Inference patterns may still be spotted easily and there is less likelihood of getting mixed up about ANDs and ORs in antecedent clauses: a common problem in developing rule systems.

This approach also suggests a useful knowledge elicitation method. Ask 'What outcomes are possible?' Then, for each outcome, ask 'Under what conditions does that happen?' This is usually much more effective than asking 'OK then, what are the rules?'

Therefore **Prefer the 'Consequent if some/all of the following antecedents' style over the 'If Antecedents then Consequent(s)' style.**

Resultant context ~ ~ ~
This pattern is terminal within this language.

2 The RulePatterns language – Part II

The second part of RulePatterns is shown in Figure 2. It may be divided into patterns for knowledge elicitation and those for product selection and application development. For brevity, I only give two small sample patlets here.

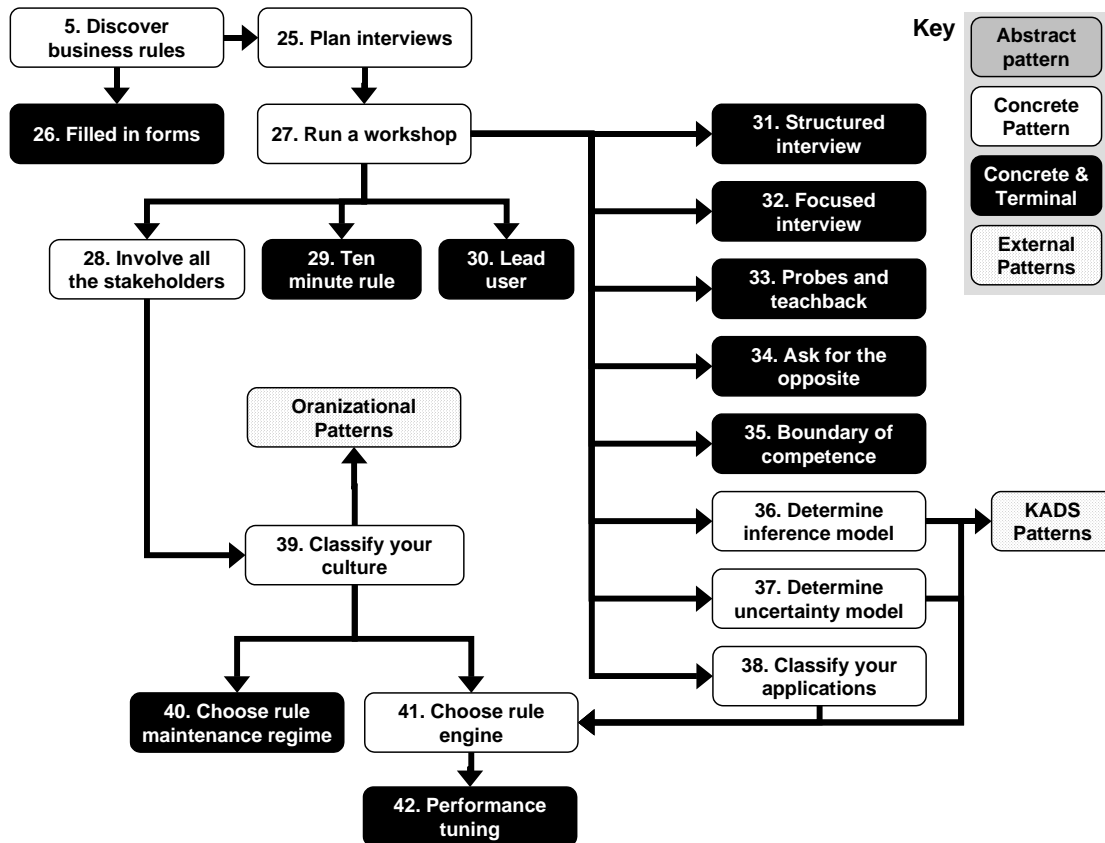


Figure 2 RulePatterns Part II

Patlet 29	TEN MINUTE RULE ***
Context	You are starting to run a workshop (27). You need to ensure that discussion is focused while not wanting to restrict it so much that information is lost.
Problem	How can you shut up vociferous (and possibly senior) bores whilst ensuring that people with valuable information to add are given free rein?
Forces	Junior or diffident individuals sometimes find it difficult to contribute and senior or aggressive ones can easily dominate the conversation even though they may have less to contribute. You really don't want long off-topic exegeses in a tightly run workshop. However, so-called 'war stories' sometimes conceal valuable gems of information and sometimes it needs quite a long presentation to delve into the business rules and processes deeply enough. So exactly how can you, as facilitator, tell the MD to shut the f*** up without causing offence?
Therefore	At the start of the workshop, announce the ground rules. These may vary, depending on circumstances. They might include a rule that forbids critical remarks (typical of brainstorming workshops). But <i>always</i> announce a ten

minute rule: no one may speak on a topic for more than ten minutes. Then, as facilitator, listen to the debate and ignore this rule totally when you feel that useful information is being added by the speaker. Only invoke it when Mr Bigmouth wants to show how clever and interesting he is. Invoke the rule and offer to ‘take his issues offline’.

~ ~ ~

Resultant context This pattern is terminal within this language.

Patlet 30 LEAD USER ***

Context You are starting to run a workshop (27). You know that disputes and impasses will inevitably arise.

~ ~ ~

Problem **How can you resolve such disputes quickly in order to move on to the next topic without bureaucratically curtailing the discussion or upsetting anyone?**

Forces In workshops, when a difficult technical issue arises and there is a significant silence – as people think about it – one often sees eyes turning to one person in the room. This person may or may not be the most senior individual but is clearly the focus of respect in the group. He (or she) could be a domain expert or a user or a skilled artisan; there are no rules for this. But he is the guy whose shoulder is cried upon when technical or practical issues need to be resolved. A good facilitator keeps a weather eyes open for such people. We call them **lead users** or **lead experts**.

In a workshop, the facilitator may decide to make the identification of the lead expert explicit, saying something like ‘Can we agree that Emily is going to resolve issues like this one when we can’t agree or just get stuck – at least for this week?’ Or it may be more prudent to keep quiet and just make sure that the lead expert is assigned to the teams given open issues to resolve and consulted by the project team regularly throughout the project. This is the right strategy when there is a danger of jealousy arising.

A lead user or lead expert should be appointed as early as possible to resolve disputes. It will be too late to get consensus on who the lead expert is to be after the dispute has arisen. This person will be one respected by other users/experts/colleagues and need not always be the most heavily involved in the project in terms of time spent. The lead user corresponds somewhat to what DSDM calls an *ambassador* user. Many users will only be consulted on an *ad hoc* basis and these then correspond to the *adviser* users of DSDM.

The lead user can help resolve another force: that of confidentiality. The group should be assured that any tapes made will be confidential to the project team and that they will be destroyed after use or, if required, returned to the lead user for destruction.

Therefore **Identify a lead expert or lead user early in any workshop or in any project. Such a person will resolve disputes and open issues and may also act as the conscience and guardian of the group. It is a matter of discretion (usually the facilitator’s) whether the lead user is publicly acknowledged as such. Consult the lead user regularly throughout the project.**

~ ~ ~

Resultant context This pattern is terminal within this language.

3 Related patterns and pattern languages

The RulePatterns language refers to three external sets of patterns that may contain useful guidance to the specifier or builder of a business rules management system.

3.1 Arsanjani's Rule Object patterns

The only other pattern language concerned specifically with business rules that I have been able to discover was developed by Ali Arsanjani (2000). Rule Object 2001 is a pattern language with 20 or so patterns for the architecture and design of business rules management systems. Some of these patterns overlap (or even contradict) those in RulePatterns but others follow on neatly from this language at STORE RULES IN A REPOSITORY (17).

3.2 KADS patterns

The guidance to be found in published knowledge acquisition methods, such as KADS (Gardner *et al.*, 1998), may be useful in conjunction with RulePatterns. In the case of KADS, its 'patterns' are quite different from what normally pass for patterns; they are largely chunks of system building advice presented as process flowcharts. However, KADS does contain a well thought out classification of different inference types and this may be useful after this language's possibilities have been exhausted.

3.3 Organizational patterns

Coplien and Harrison (2005) present well researched and tested organizational patterns in the form of two languages. These patterns will be as much use to BRMS developers as to those working on any other kind of project. Indeed they quote patterns 1, 2, and 8 from this language.

4 References

- Alexander, C., Ishikawa, S. and Silverstein, M. (1977) *A Pattern Language*, Oxford: Oxford University Press
- Arsanjani, A. (2000) Rule Object: A Pattern Language for Pluggable and Adaptive Business Rule Construction; in *Proceedings of PLoP2000*. Technical Report #wucs-00-29, Dept. of Computer Science, Washington University Department of Computer Science, October.
- Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. (1996) *Pattern-oriented Software Architecture: A System of Patterns*, Chichester, England: Wiley
- Coplien, J.O. and Harrison (2005) *Organizational Patterns of Agile Software Development*, Upper Saddle River NJ: Prentice Hall
- Date, C.J. (2000) *What Not How: The Business Rules Approach to Application Development*, Reading MA: Addison-Wesley
- Gardner, K., Rush, A., Crist, M., Konitzer, R. and Teegarden, B. (1998) *Cognitive Patterns*, Cambridge: University Press
- Graham, I. (2001) *Object-Oriented Methods: Principles & Practice – Third Edition*, Harlow, England: Addison-Wesley
- Graham, I.(2003a) *A Pattern Language for Web Usability*, Harlow, England: Addison-Wesley
- Graham, I.(2003b) Four web usability patterns from the *wu* language, in O'Callaghan, A., Eckstein, J. and Schwanninger, C. (Eds) *Proc. EuroPLoP '02*, UVK Universitätsverlag Konstanz, 159-177
- Halle, B. von (2002) *Business Rules Applied*, New York: Wiley
- Morgan, A. (2002) *Business Rules and Information Systems: Aligning IT with Business Goals*, Boston MA: Addison-Wesley
- Ross, R.G. (2003) *Principles of the Business Rules Approach*, Boston MA: Addison-Wesley
- Sillitoe, A. (1971) *Travels in Nihilon*, London: W.H. Allen & Co.