1530 E. Dundee Road, Suite 100
Palatine, Illinois 60074
(847) 963-1777
info@uniquesoft.com

Technical White Paper

Business Rules Extracted from Code

Version 2.2

1 Overview

The purpose of this white paper is to describe the essential process for extracting business rules from source code. As an integral part of a software modernization project, it is common to want to identify and extract the business rules that are implemented in the source code for a system. An important distinction is made here between the high-level, top-down business rules which may or may not have any direct implementation in the code and the concrete, bottom-up business rules which represents what the code does (which may or may not be aligned with what the business intends it to do).

While there is no single, universally applicable definition of a business rule, we can describe certain properties that the extracted business rules must have to be useful. The next sections discuss these properties, and the process of extracting the rules is described.

2 Top-Down Versus Bottom-Up Business Rules

We will focus here with the business rules that are implemented in the code. That is, we are interested in the decisions and actions that are the functionality of the applications. For example, consider the following top-down, high-level business rule:

No loans can be originated to high-risk applicants.

While this rule states a relatively simple concept, the details are too vague for it to have a direct implementation in code. What is implemented in the code would probably better summarized as the following bottom-up rule:

A loan can be originated to an applicant if the applicant has zero delinquent accounts and the loan type is home equity or mortgage, or two or fewer delinquent accounts and the loan type is auto. Otherwise, the loan is rejected.

The process of going from the code to this more detailed expression of the business rules will be discussed below.

3 Essential Rule Properties: Correctness and Understandability

For extracted business rules to be useful, independent of how one will ultimately use them, they *must* have two properties: correctness and understandability. Incorrect rules can be more detrimental than having no rules at all, and having rules that are not understandable serves no purpose. This section discusses what is required to have correctness and understandability in the extracted business rules.

3.1 Correct Rules

Three main characteristics must hold for rules to be considered correct:

1. The code from which the rules are extracted must be complete.
2. The rules that are extracted must be equivalent to the original code.
3. The context of the individual rules must be captured.

We will assume here that the original code is, in itself, correct in some sense. That is, we must capture what the code does, not what we think it should be doing. To that extent, having *all* the relevant code is critical. If the code is not complete, rules will be missed or misinterpreted. All the referenced files must be included, and there should be no missing definitions.

The behavior expressed by the extracted rules must be equivalent to the behavior of the original code. All the logic and computation of the original code must be expressed in the conditions or actions of the rules. If they are not, then the rules cannot be equivalent to the original code.

The context of the rules is equally important for correctness. Just as the statements in the code cannot be executed in an arbitrary order, a correct set of rules is not just a collection of all the decision statements in the code. The sequencing of and dependencies between the rules must be captured. For example, consider the following pseudo-code example:

```
IF retiree_years_of_service >= 5 THEN
   benefit_percent = 50
ELSE
   benefit_percent = 25
ENDIF
…
IF retiree_years_of_service >= 5 THEN
   benefit_percent = 75
ELSE
   benefit_percent = 50
ENDIF
```

As individual rules, there is clearly a contradiction as to what the benefit percent should be. However, if we look at the contexts of these decisions in the overall flow of the code, we may see that the second rule is only used when the retiree is at the vice-president level, as is shown below.

```
IF retiree_grade == vice_president THEN
   compute_compensation_b
ELSE
   compute_compensation_a
ENDIF
…

compute_compensation_a {
   …
   IF retiree_years_of_service >= 5 THEN
      benefit_percent = 50
   ELSE
      benefit_percent = 25
   ENDIF
   …
}
…

compute_compensation_b {
   …
   IF retiree_years_of_service >= 5 THEN
      benefit_percent = 75
   ELSE
      benefit_percent = 50
   ENDIF
   …
}
```

The various parts of this logic may be thousands of lines apart in the code and in different methods/paragraphs. To correctly understand a rule, one must know the context in which it is being executed. D*Code can express this context as explicit sequencings of rules, or as combined rules where possible. For example, the following rules table may be produced.

retiree_grade == vice_president	retiree_years_of_service >= 5	*ACTION*
True	True	benefit_percent = 75
True	False	benefit_percent = 50
False	True	benefit_percent = 50
False	False	benefit_percent = 25

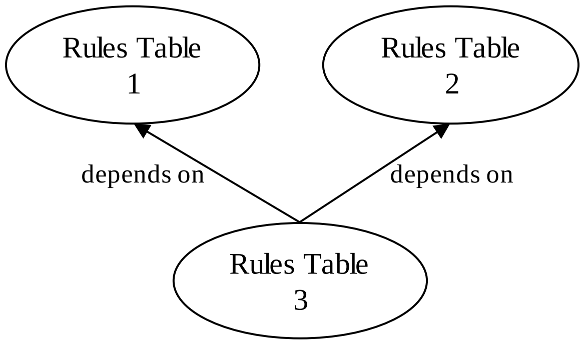
Or, in a more easily understandable English-like notation.:

Retiree Grade	Retiree Status	*ACTION*
Vice President	Fully vested	Set benefit level to 75%
Vice President	Partially vested	Set benefit level to 50%
not Vice President	Fully vested	Set benefit level to 50%
not Vice President	Partially vested	Set benefit level to 25%

Similarly, the dependencies between the rules must be fully captured. For example, consider the following pseudo-code example.

```
IF retiree_years_of_service >= 5 THEN
   benefit_multiplier = benefit_multiplier + 0.2
ENDIF
IF retiree_is_vice_president THEN
   benefit_multiplier = benefit_multiplier + 0.5
ENDIF
IF retiree_payment_choice IS annuity THEN
   benefit_percent = 75 * benefit_multiplier
ELSE
   benefit_percent = 50 * benefit_multiplier
ENDIF
```

In this case, the first two decisions can be executed in either order. However, both must be executed *before* the third rule can be executed. Capturing the dependency of the third decision on the first two, as is shown below, is critical to correctness.



Rules Table 1

Retiree Status	*ACTION*
Fully vested	Increase benefit multiplier by 0.2

Rules Table 2

Retiree Grade	*ACTION*
Vice President	Increase benefit multiplier by 0.5

Rules Table 3

Retiree Payment Choice	*ACTION*
Annuity	Set benefit level to 75% of the benefit multiplier
not Annuity	Set benefit level to 50% of the benefit multiplier

Understanding that the first two decisions are independent, while not necessary for correctness, aids understandability, which will be discussed in the next section.

3.2 Understandable Rules

While it is necessary that the extracted rules be correct, the rules must also be understandable to be useful. What is understandable will depend on the target audience, but the following three general principles apply.

1. Unwanted code should be filtered out of the rules.
2. Unnecessary complexity should be removed.
3. English descriptions of the rules should be available.

Filtering out unwanted code has two aspects. First, the business logic in the code may be separable into individual use cases. For example, the logic for creating a new insurance policy may be independent of the rules associated with the logic for processing a claim. Being able to process the business logic for each use case separately may allow smaller and more intuitive rule sets to be extracted.

Second, applications often contain code that is *not* part of the business logic. This code relates to how the application interacts with its users, what platform error handling it performs, what activities it keeps track of, etc. Code artifacts that are typically *not* business rules include:

- User input error detection / correction
- User interface design (e.g., the look and feel of the GUI)
- Program initialization and scheduling
- Database interfacing and error handling (i.e., the mechanics of database access)
- System and activity monitoring and logging
- Dead code
- Output formatting and printing
- Input data parsing
- Authentication (unless it is part of the business process itself)

In general, though, one must be careful not to remove too much code from consideration. For example, determining if a user-supplied account number is well formed is probably not business logic, but determining if the account number corresponds to an active account might be. Another consideration is the ultimate use of the extracted rules. If they will be used by a business team to review the business process or will be used as requirements for the system, then error handling, etc., should be removed. If the rules will be directly implemented in a rules engine, removing all the error handling is probably a bad idea.

As mentioned in the previous section, unwanted complexity should be removed. This complexity imposes artificial dependencies between the rules. In the example code above, we saw that the first two rules can really be executed in either order. Which one appears first in the code is based on a programmer choice and does not reflect an inherent constraint. Similarly, it may be possible to reorder the code to combine rules with the same conditions. Especially after code has been maintained for years or decades, simplifying the code though reordering and removing artificial dependencies often provides a significant benefit.

Being able to review the rules in an English-like notation removes the reliance on understanding the programming language and on the algorithms used to implement the logic. For some programming languages, such as COBOL, the use of abbreviations and cryptic identifier names in the code is common, which makes understanding the rules even more difficult. When the rules will be reviewed by a business team, making these rules more English-like may be a necessary step. For example, consider the pseudo-code example.

```
IF comb_score <= 450 THEN
   ln_cnt = 0
   ln_tot = 0
   FOR ln FROM 0 TO max_ln DO
      ln_cnt = ln_cnt + 1
      ln_tot = ln_tot + lns[ln]
   ENDFOR
ENDIF
```

Extracting this rule would result in a row in the table such as the following:

comb_score <= 450	*ACTION*

True	`ln_cnt = 0`
	`ln_tot = 0`
	`FOR ln FROM 0 TO max_ln DO`

	`   ln_cnt = ln_cnt + 1`
	`   ln_tot = ln_tot +`
	`lns[ln]`
	`ENDFOR`

With some idea of the domain of the code and the way the programmer named the identifiers, this code is not that hard to figure out assuming a basic understanding of coding. Even so, *every* reviewer would have to figure it out. It would be better to have the rule in the table shown as the following:

Combined Credit Score	*ACTION*
Poor	Calculate the number and total amount of all outstanding loans

Rules such as these are also easily written in a textual form, as in the example below:

given Combined Credit Score is Poor
do Calculate the number and total amount of all outstanding loans

4 <u>Business Rules Extraction with D*Code</u>

UniqueSoft has an automated tool, called D*Code, for extracting business rules. D*Code has all the correctness and understandability characteristics described above and supports various aspects of legacy modernization:

- The *visualization* component provides both a high-level qualitative overview of the legacy code as well as detailed data and metrics, including the completeness of the code. The information gleaned from visualization is often used to perform a quality analysis of the legacy code to understand where are the highest areas of maintenance risks or to understand the architecture of the legacy system, its behavioral use cases, and its relationship to external resources.
- The *discovery* component allows features to be located that share certain characteristics within the legacy code. These features may be the code realizing business use cases, architectural elements, cross-cutting concerns such as error handling, and so on.
- The *rule extraction* component identifies business-relevant conditions in the legacy code and refactors the legacy code into a set of rules and the dependencies between the rules. A rule is comprised of a set of business-relevant conditions and a code fragment (action), such that if the conditions all evaluate to true, then the action is executed. The rules, when evaluated in any sequence allowed by the dependencies, exhibit the same behavior as the original legacy code and document its complete behavior.
- The *translation* component translates the legacy code from its original language into a target language, preserving its behavior. The translated code is similar to code written by hand to facilitate maintenance.
- The *migration* component removes code from the legacy system that is specific to its original computing environment (e.g., a mainframe) and realizes this functionality in terms of the target computing environment (e.g., a Linux cluster) leveraging a chosen set of libraries. Code that is not relevant for the target environment will be removed, and code will be added which realizes functionality which was implicit in the legacy environment.
- The *test generation* component derives a test suite from a high-level model of the legacy system such that the test suite is the smallest set of tests that will achieve branch coverage of the system model.

The process of extracting business rules with D*Code is described in the rest of this section.

4.1 Rule Extraction Overview

D*Code aids the analyst in expressing a business-level understanding of the code which is not in itself present in the code. The output of D*Code business rule extraction is a set of decision tables which express the behavior of the system as independent actions that are triggered by business-relevant conditions having certain values. In addition to the tables, the dependencies between the tables is captured in a navigable, hierarchical representation.

A *rule* is code that is executed when a set of conditions is true. There are, of course, numerous conditions throughout the code. A business rule is a rule which contains only business-relevant conditions. Such rules summarize the legacy code in terms of its business requirements. What makes a condition business-relevant depends on what the original implementers intended. Business rule extraction thus is focused on separating business-relevant conditions from technical decisions that are necessarily part of the implementation, but not part of the business requirements. D*Code provides various techniques to identify the business-relevant conditions.

Business rule extraction proceeds in the following steps for each program in the legacy system:

- Identify use cases in the program
- Identify the business logic in the program
- Identify the business-relevant conditions
- Identify fragments considered as independent rule sets
- Convert each fragment to a set of business rules

This process is iterative. If the actions performed by extracted business rules are considered too large or contain conditions which upon closer inspection appear to be business-relevant, one can easily iterate in D*Code by identifying additional (or fewer) business-relevant conditions. This ability to easily iterate the extraction process also aids correctness and understandability. The reality of modernization projects is that the legacy code may be changing during extraction, and the business analysts may change their minds on what constitutes business decisions. If the extracted business rules are not updated to reflect the new code or the new business conditions, they will be incorrect or difficult to align with the business processes. Either way, redoing the rules extraction without automated support can be prohibitively expensive and time consuming, and when the extraction cannot be easily iterated, the initial extraction, for good or bad, will be used.

4.2 Identify Use Cases

If a program contains multiple use cases, business rules are typically more intuitive when extracted separately for each use case. These use cases can be identified in D*Code though domain analysis or through inspection of the program flow graphs. Identifying the use cases through domain analysis requires user input, but it provides a finer-grained control over what code is in which use case. This is especially helpful when the use cases are intertwined in the code.

Identifying the use cases from the flow graphs can be done more quickly than through domain analysis, but it requires some understanding of the structure of the program (e.g., that CICS is being used for user input and output). Both techniques provide the information needed by D*Code to separate the business rules into logically cohesive groups.

4.3 Identify Business Logic

In this step, we need to separate the code that performs the business purpose of the program from code that merely performs the implementation of technical detail. A developer has only two means of documenting the code's intent: comments can be inserted into the code clarifying the meaning of a fragment of code, and the identifiers and literals in the code can be chosen to illustrate their meaning. In our experience, comments are not a reliable indicator as they are often not maintained and may thus be misleading.

D*Code first focuses on the identifiers and literals, and uses their distribution throughout the code and their relationships to analyze which identifiers relate to which features of the code. D*Code provides various tools to highlight such relationships, such as substring cluster analysis, substring sequence analysis, and substring analysis. This analysis highlights important relationships which determine features of the program. D*Code makes it simple for the user to get a quick understanding of the purpose of such identifiers or literals by navigating to definitions and all uses of each identifier or literal throughout the legacy code base. This information is used to automatically remove the related code.

If business rules are to be extracted from the code for review by a business team (as opposed to being extracted for implementation in a rules engine), D*Code can remove all code not associated with business logic. This code is unlikely to reflect business-relevant concerns, and its removal will make the extracted rules smaller and easier to understand.

For example, a substantial portion of most programs is devoted to dealing with failure (“rainy day”) scenarios. Missing input, lack of authorization to access a data base, lost connections, and many more, all need to be accounted for in a robust and reliable application. Often, the logic devoted to dealing with these scenarios is much larger than the logic devoted to implementing the business purpose of a program, both in terms of code size and in terms of complexity. This logic dealing with error situations is often distracting and tends to obscure the business logic itself. It is, therefore, helpful to focus the rule extraction on the “sunny-day” scenario, where failure scenarios are assumed not to occur.

For this purpose, we define features using D*Code discovery to identify code that is not part of the business logic. D*Code can then remove all code that has been annotated as being part of these features. The applied transformation removes all code that would be executed uniquely along a path that leads to one of these features. In other words, after this transformation, the remaining code assumes that this unwanted behavior never occurs.

4.4 Identify Business-Relevant Conditions

For business rule extraction, we need to determine which of the remaining decisions relate to business concepts and which, albeit still present in the business logic, relate to technical concepts. D*Code domain analysis depicts all conditions, condition expressions, and condition literals used in decisions throughout the program, along with links to where they are used in the code, allowing the user to determine the purpose of the decisions.

Often, many of the variables used in conditions can be traced, directly or indirectly, to inputs or to data extracted from database tables. Either the values of these variables were input or obtained from database tables directly or assigned from such variables, or the variables expressed that some other data was successfully extracted from a database table or obtained from input. Such conditions also tend to reflect information that is business-relevant.

4.5 Identify Fragments Considered as Independent Rule Sets

Either by visual inspection, from the flow graph, or from the complexity table, we identify paragraphs or sections that encompass fragments of high complexity, as these will contain many decisions. These fragments will benefit from being extracted into separate rule sets. Similarly, fragments that seem to indicate independent computations, as indicated by use case analysis or by the flow graph, will be treated as independent rule sets.

4.6 Extract Business Rules

D*Code then transforms the legacy code into rule sets for each identified fragment. The extracted rule sets have the same computational result as the original code under the following interpretation: A rule set consists of rule groups. These must be executed in order of their data dependencies. When all conditions of a rule are true, then the action of the rule is performed. Within a rule group, the rules either have mutually exclusive conditions so only one rule will eventually apply, or all matching rules may be performed in arbitrary order. The conditions of the rules in a rule group can be evaluated in any order or in parallel.

The extracted business rules can be represented in a variety of ways, such as decision tables captured in a spreadsheet or informal text. D*Code can aid in this process by, for example, automatically migrating the extracted rules to a rules engine or by translating the extracted rules to a modern language such as Java.

Once the rules are extracted, the conditions and actions can be summarized in English. The names used in the original program are often cryptic and inconsistent, and the actions are expressed in the implementations of the algorithms in the code. While this is often the format desired when reimplementing the rules in a rules engine, it is not the easiest for a human to review. D*Code provides a mechanism for the decision names, decision results, and actions to be summarized in English, and this English form can be browsed and reviewed in the same way that the original rules can. However, the original code-based rules are also still available so that there is traceability from the English rules back to the code.

5 **Summary**

There are multiple purposes for rules extraction, including creating system documentation, understanding the overall business logic, and providing a basis for reimplementing the system on a new platform. Whatever the purpose, correctness and understandability are paramount. The code from which the rules are extracted must be complete, the rules that are extracted must be equivalent to the original code, the context of the individual rules must be captured, the unwanted code should be filtered out of the rules, unnecessary complexity should be removed, and English descriptions of the rules should be available.

When rules are extracted from code, most tools produce rules as sets of conditions and an action, obtained from the code. However, more than just code snippets are needed. For example, all the following are important:

- Differentiating between business relevant conditions and merely technical conditions. The conditions of business rules should only reflect what matters to the business, the remainder of the conditions should remain hidden in the actions or be removed.
- Separating the business logic from the technical logic. While it is not always desired that the technical details be removed, it can provide a focus on the business logic for a review team or for creating requirements.
- Removing the "accidental ordering" from the code. Programming languages force an ordering of the statements, and this sequencing may not reflect the required sequencing of the business rules based on their dependencies.
- Understanding the context and structure of the conditions in the code. A decision based on a condition in one part of the code may mean something different than the same condition in another part of the code.
- Providing intuitive aliases for both the conditions and the actions. The vocabulary of the code is often cryptic and inconsistent, and providing understandable aliases can greatly ease the review and documentation processes.
- Providing multiple representations of the resulting rules. Different teams and different uses require different formats for the output. For example, a project may need tabular rules, an interactive graphical format, English-like summaries, gherkin output, etc.
- Extracting the conditions and decisions from the code itself, which show what the code actually does, as opposed to what the developers or business analysis thought it did.

One common element shared by all rule extraction tools is the ability to extract decision statements and conditions from the code. The differentiators between the tools are in how well the above concerns are supported, how much manual effort is required to extract the rules, how suitable the output is for the intended purpose, and how easy it is to iterate the process.

UniqueSoft's D*Code tool is the only one that provides all the capabilities listed above as well as a higher degree of automation to the rules extraction process. Manual extraction is tedious and brings opportunities for error, so automation is a critical component. For example, some commercial rules extraction tools allow the user to provide a rule summary description in English, However, the rule itself is still legacy code that simply includes the entire conditional statement this summary was based on. Other commercial tools simply extract every condition in the code as is. When entire IF-THEN-ELSE statements are captured as a single rule, this can cause confusion when the two branches represent completely different behaviors. In addition, when looking only at a single condition, the context of how the execution of the program arrived at this condition is lost. This context, however, is an essential part of the true business rules.

To extract true business rules, one needs to consider not just individual conditions, but also understand how conditions are nested in sequences of other decisions and calls. These inter-rule dependencies and sequencings are vital for the correct functioning of the system as a whole. D*Code separates each branch of the conditionals into different rules, building rule sets that capture not only the individual rules, but also the sequencing within the rules and among the rule sets. A graphical interface is provided which interactively shows the hierarchical flow of rule dependencies as well as details of each individual rule. Intuitive names can also be systematically applied to both the conditions and the actions of the rules, which aids in the review and documentation process.

D*Code has additional capabilities that are often useful for rules extraction projects. For example, D*Code can be customized to deliver the rules in a wide range of desired formats, standards, and languages, and it can automatically generate a set of test cases to help validate the functionality of the new system. Please contact UniqueSoft for more information.