HIGHLY CONFIGURABLE SOFTWARE DEVELOPMENT - ANALYSIS OF EXISTING FRAMEWORKS AND DESIGN OF A GENERIC DEVELOPMENT FRAMEWORK

THESIS

Presented to the Graduate Council
of Texas State University - San Marcos
in Partial Fulfillment
of the Requirements

for the Degree

Master of SCIENCE

by

Narasimhan Kaliyamoorthy, B.Sc

San Marcos, Texas
August 2005

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor, Dr. Gregory Hall for his motivation and guidance. His suggestions and encouragement helped me throughout the research and writing of this thesis.

I am thankful to Dr. Carol Hazlewood and Professor Wilbon Davis for their advice and encouragement in writing this thesis. I thank them for reviewing the manuscript of this thesis and offering suggestions for improvement.

I am very grateful for the love and support of my parents Mr. Kaliyamoorthy and Mrs. Lakshmi, and my brother Dr. Sathyanarayanan.

Finally, I would like to thank the faculty of the Computer Science department at Texas State University-San Marcos for their kind support.

This manuscript was submitted on July 25, 2005.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

HIGHLY CONFIGURABLE SOFTWARE DEVELOPMENT - ANALYSIS OF EXISTING FRAMEWORKS AND DESIGN OF A GENERIC DEVELOPMENT FRAMEWORK

by

Narasimhan Kaliyamoorthy

Texas State University-San Marcos

August 2005

SUPERVISING PROFESSOR: GREGORY A. HALL

Business rules represent an organization's codified policies and decision-making practices and determine how a business operates. Typically, if the business has automated the business operation or process, the rules are embedded in the software solution within the application or database code. Separating the business rules from legacy code is one of the biggest challenges in software maintenance.

A rules-centric approach to software development facilitates defining business rules that are easily and dynamically maintained. In the thesis, a generic software engineering framework that enables development of highly configurable software was proposed. This framework will significantly reduce the number of development cycles triggered by the changing software requirements, during the software maintenance phase. This lightweight framework drives a rule-engine to incorporate dynamic changes and implement specific business logic in configurable applications. The methodology is based on the rules-centric appr-oach to software development. This research also involves a detailed study and analysis of the existing software development platforms and frameworks that support a high level of software configurability.

CHAPTER 1

1.1 Introduction

Whether they are explicit or implied, rules are an inevitable part of doing business. Business rules represent an organization's codified policies and decision-making practices, and they dictate how applications are structured and operate.

Businesses determine the way information flows through its organizational processes and the sequence of actions the organization will follow under given conditions. The set of rules that determine how a business operates are those that prevent, cause or suggest things to happen in the functions of the business.

1.1.1 Explanation of the problem

To promptly react to changing conditions, businesses must be able to adapt and implement those changes as rapidly as possible in the set of business rules. Typically, if the business has automated the business operation or process, the rules are reflected in the implemented software solution in one or more of the following locations: either within the business application code or in the database code, implemented as procedures and triggers.

Also, many business rules exist in legacy programs and may have embedded business rules that are highly intertwined with the application and data processing logic.

Separating the business rules from legacy code is one of the biggest challenges in software maintenance. Because the application code contains the business rules embedded with the control and data processing logic, adapting to change in software requirements becomes a tedious, time-consuming and costly exercise.

Moreover, every change in requirements, irrespective of the amount of change or the impact of the change, requires the application to undergo a complete software development cycle and to be rebuilt.

1.1.2 Highly configurable software overview

Software development frameworks provide the skeleton of the structure and control flow of the implemented applications. Frameworks greatly reduce the size of the solution space available to the developer. For example, The J2EE framework manages the life cycle of an application component implemented in Enterprise Java Beans (EJB) all by itself, thereby relieving the programmer from coding and maintaining the infrastructure.

This research proposes a generic framework methodology based on the rules-centric approach to highly configurable software development. This research is based on a detailed study and analysis of the existing software development platforms and frameworks that support a high level of software configurability.

The core objective of the adopted rules-centric approach for the generic framework is making the definition and implementation of business rules logically and possibly physically separate from the application data and processing logic. The first step of the rules-centric approach is to define the business rules that govern an organization. Once represented in a standard structure as a rules repository, business rules become separate, reusable definitions that can respond to changes in business requirements.

A business rule is a statement that defines or constrains some aspect of business. It is intended to assert business structure or to control or influence the behavior of the business. There are many benefits to using a business rules-centric approach to develop applications.

Because business rules are represented in an understandable way, business people are able to review the rules for accuracy and completeness. When data is analyzed from business intelligence systems, business people can quickly associate their conclusions with active business rules.

1.1.3 Organization of the thesis

This thesis is divided into six Chapters with Chapter 1 providing an overview of the problem and the methodology adopted to address the problem. Chapter 1 also gives an overview of the rules-centric approach to software development and how highly configurable development can be supported using the proposed generic framework.

Chapter 2 provides definitions for terms and explains concepts used in development of highly configurable software. The factors and elements that make software highly configurable and the benefits of a high level of configurability are discussed. Chapter 2 describes how the rules are represented in a standard format which can be used for execution by conforming rule engines and how change management is carried through in a business rules-centric approach over the software maintenance phase.

Chapter 3 analyzes several of the existing frameworks that support development of highly configurable software. This chapter identifies a set of criteria by which each framework will be analyzed. Each framework is then analyzed in detail with respect to the identified criteria. Chapter 3 also proposes rule transformation methods that can be applied to the existing frameworks to further extend software configurability provided to a reusable level.

Chapter 4 begins with a discussion on the imperative and declarative programming paradigms. Since the proposed framework will use a declarative language to represent the business rules in the rules repository, this chapter identifies some of the benefits of using a declarative language and discusses the drawbacks associated with imperative programming.

The scope for a generic framework in the rules-centric approach for software development is discussed. The shortcomings of each framework analyzed in Chapter 3 and the ways in which the proposed generic framework addresses them are discussed. In Chapter 4, the methodology adopted for the proposed generic framework is discussed and, a detailed design is presented. The adopted rule representation format for the generic framework – the Simple Rule Markup Language (SRML) - and its document structure in the rules repository is presented. The Rete algorithm is a widely used pattern matching algorithm that the generic framework uses for matching the assertions on facts with the conditions present in the rules. Chapter 4 presents a detailed discussion and examples on how the Rete algorithm was used in the implementation of the rule engine.

Chapter 4 also presents the design of the rule engine based on the JSR-94 Java Rule Engine API specification from SUN Microsystems. This chapter also discusses how the rules in the rules repository are parsed and executed in the memory model and the execution context that the generic framework employs. Chapter 5 presents the results of the analysis of the generic framework by conducting two case studies from contrasting target application domains. The case studies in effect will demonstrate the capabilities of the generic framework and how the methodology differs from conventional software development.

Each case study begins by analyzing the implementation based on the conventional approach. The sections of the application implementation where the business logic or the functional rules are embedded in code are identified.

The business rules are separated and represented in the SRML document format and the implementation based on the rules-centric approach using the proposed generic framework is then discussed for each case study in Chapter 5. Chapter 6 provides the conclusions of the research and a few directions for future research.

CHAPTER 2

2.1 Highly Configurable Software Development

This chapter provides definitions for terms and explains concepts used in development of highly configurable software. The factors and elements that make software highly configurable and the benefits of a high level of configurability are discussed. This chapter introduces the business rules-centric approach of software development, describes what configuration specifications are and what industry standards and formats are used in representation of the specifications. This chapter also describes how the rules are represented in a standard format which can be used for execution by conforming rule engines and how change-management is carried through in a business rules-centric approach over the software maintenance phase.

2.1.1 Definitions

The policies, procedures and decision-making practices of a business are codified or represented by business rules. Business rules serve as a basis for the design and structure of the business applications that implement the functional requirements.

R Ross (1997) defined business rules as follows:

"Business rules are a formal expression of knowledge or preference, guidance system for steering behavior (a transaction) in a desired direction. It is intended to assert business structure or to control or influence the behavior of the business."

D. Hay and K. Healy (2000) gave a more simplistic definition:

"A Business Rule is a statement that defines or constrains aspects of a business.“

Business rules are atomic and complete statements that enable new knowledge, and enable or disable actions based on previous knowledge. By defining business rules in a consistent manner, an organization has the opportunity to implement the business requirements in its systems in ways that was not possible in conventional software development where business rules are tied down in compiled code interwoven with data and application logic.

Once represented in a standard structure as a rules repository using formats such as the Extensible Markup Language (XML), business rules become separate, reusable definitions that can respond to changes in business requirements, technical and environmental migrations.

A *business rules repository* is a centralized store of the business rules to facilitate change and promote reuse. A *business rules server* offers dynamic access to business rules at runtime. Developers or application administrators can directly change multiple rules at a time and have those changes available to all relevant business transactions immediately, without recompiling and redeploying the application.

The most basic element of a business rule is the *rule language* used to express it. The complexity of the language adopted to represent the business rules can greatly influence the maintainability of the applications developed using the framework.

Edward J. Barkmeyer, Evan K. Wallace and Ravi Raman (2002) state the fundamental requirements for a rules language:

> "It is necessary for a rules language to support some standard mathematical reasoning systems beyond sets. These requirements arise from two areas of work: capturing the rules needed to support translation among measurement values in different units within and across unit systems and capturing structural differences between representations of the same information in different schemas when the values themselves are semantically equivalent. We cannot force the user to explicitly flatten state structures and nested interrogations and actions into conjunctions of symbols. The source rules languages must allow these things to be written in a more natural way. And software can be written to perform the transform to conjunctions of symbols."

Business rule types. Several classifications for business rules exist depending on the functional domain that the business rules are derived from. According to B. Von Halle (2001), there are four kinds of business rules:

> *"Constraint rule.* A constraint rule is a statement that expresses an unconditional circumstance that must be true or false. Example: An air travel request must have a departure airport and a destination airport.

Action enabler rule. An action enabler rule is a statement that checks conditions and upon finding them true initiates some action. Example: If no flight is found, do not look for accommodation.

Computation rule. A computation rule is a statement that checks a condition and when the result is true, provides an algorithm to calculate the value of a term. Example: If more than 2 persons travel together, the third pays only half price.

Inference rule. An inference rule is a statement that tests conditions and upon finding them true, establishes the truth of a new fact. Example: A frequent customer gets a discount of 5 %."

Business rules may contain values or even complete decision trees that might change during the life span of an application. If business rules are placed in application code, change becomes a costly exercise, resulting in high maintenance and enhancement costs. Centralizing business rules and classifying them into various types facilitates isolation for change, provides enhanced knowledge about the dependencies of rules and promotes reuse.

2.1.2 Factors that make software highly configurable

The separation of business rules from the implementation or application code is the first step towards development of highly configurable software. The next step is to adopt a consistent approach to isolate the business rules starting from the requirements elicitation phase of software development. The business rules-centric approach is one such effort towards isolating the rules and application code in an efficient and interoperable manner.

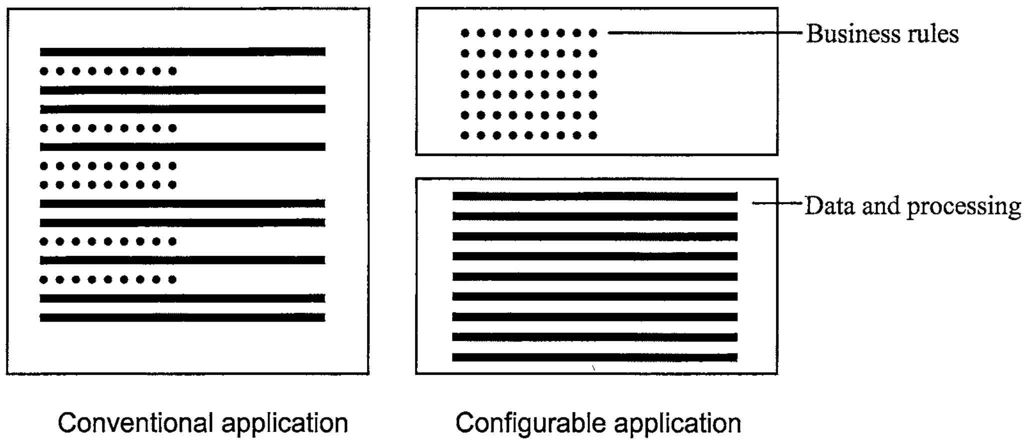


Figure 2.1 Conventional vs. Configurable application development

Any rules-centric approach requires a standard format to represent the business rules. The representation format should support easy interchange of rules between systems as well as providing efficient access to the rule engines that operate on them and use the business rules to implement the functional requirements.

Several standards have evolved to support the representation of business rules in interchangeable and portable formats such as the Rule Markup Language (RuleML), Simple Rule Markup Language (SRML), Extensible Rule Markup Language (XRML) and Relational Functional Markup Language (RFML).

The design and development of rule engines specific to the chosen representation format enables true configurability for the applications developed using the rules-centric approach. The rule engines will dynamically incorporate execution of business rules in a context-sensitive fashion.

2.1.3 Using a business rules-centric approach

A business-rules centric approach to software development frameworks facilitates defining business rules that are easily and dynamically maintained. This approach isolates the implementation of business rules both logically and physically from data and application processing code. The first step of the business rules-centric approach is to isolate and define the business rules that govern the needs of functional requirements of applications. This will allow the application development organization to gain explicit knowledge of the decision-making process itself.

The business rules-centric approach includes both a methodology by which rules of a business are captured, managed and automated as well as a technology for managing the rule automation and change process. It aims to represent business knowledge externally as an active component in the development architecture.

Relationships between rules are represented as supported by the adopted structure and language for the business rules. Decisions and inferences are evolved by grouping the rules to a point where the outcome can be clearly defined.

B. Von Halle (2001) summarizes the concepts of a rules-centric approach:

"A unique aspect of a business rules system development methodology is that it divides the systems development approach into separate, but integrally related tracks representing the workflow/process perspective which includes the user interactions and processes, but without the rules in them"

Business rules models are evolved by refining the decision-making process targeted towards a specific set of functional and operational requirements. These models capture and represent all the facts and logic required to make a decision in a given business context, allowing reuse of both atomic and complex rules.

Although several development frameworks exist in support of highly configurable software development using the business rules-centric approach with well-defined rules repositories and efficient rule engines, there is not much support for transformation of business rules from one format to another.

Transformations within a single representation format allows for accommodation of changes made to the business rules, whereas transformation between representation formats will greatly improve interoperability of heterogeneous systems and make software truly configurable for change in the environments and portability requirements.

In an attempt to enhance the configurability of software further to a higher level of reusability, this thesis proposes a rule transformation mechanism which can be applied as an extension to existing frameworks as well as integrated as part of a generic framework.

2.1.4 Benefits of the business rules-centric approach

The business rules-centric approach ensures that business logic is implemented in a standard and consistent way, narrowing the communication gap during the requirements, analysis and design phases of projects. Because of this explicit knowledge representation, there is little need to restrict change in requirements. Developers can take an iterative approach to implementation of the requirements. This results in a shorter time to code and a shorter time to deploy.

Once applications are deployed, changes to business rules are easier to implement because they reside in a centralized location. Moreover, because business rules are separated from technical implementation, it becomes much easier to accommodate changes in requirements.

There are several other benefits in using a business rules-centric approach for developing highly configurable software:

Reduced application development costs. In traditional application development, most of the application code exists solely to make the business logic apply on a specific instance of the implementation. With a rules-centric approach, application development costs are reduced by maintaining the rules dynamically and translating them into an implementation in order to adapt to changing requirements.

R Ross (1997) states the benefits of a rules-centric approach as:

"On the business side, changing business practices are no longer unnecessarily disruptive and costly to implement with technology. The separation of business logic and architecture means that software future proofing is no longer thwarted by subsequent technologies."

S. Bohner (2002) on how change is inevitable in software and how a business rules-centric approach is beneficial:

"Final requirements seldom exist for software systems since they are continually being augmented to accommodate changes in user expectations, operational environment and the like. Configurable software platforms with business rule models are more flexible to adapt for these changes. Business people become an active part in system specification as well as the analysis, design and implementation phases."

Shorter development cycles. Rules-centric frameworks shorten development cycles by developing business models that closely match the projected implementations of the application. This shortens not only the actual coding time, but the overall development cycle as well. The increased quality that results from automation shortens the quality assurance phase of the development cycle. Rules-centric systems can also generate more accurate and implementation-specific user and system documentation which shortens the documentation phase.

David Zygmont (1999) on the benefits of a rules-centric approach:

"Rule-based systems reduce application development costs by isolating the architecture in metaprograms and having the metaprograms translate into an implementation. This fixes the cost of architecture independently of the size or number of applications being developed. The bigger the application or the more applications there are, the greater the savings in time and cost. "

Easy application migration to new technologies. Advances in technology turns architectures that are just a few years old obsolete. Organizations frequen-tly discontinue support of older technology and personnel with the skills to maint-ain and enhance older technology are difficult to find. This has become increasin-gly problematic due to the ongoing trend of decreasing product life cycles.

According to David Zygmont (1999), rules-centric systems ease migration:

> "Metaprograms and rule-based systems isolate applications from the technology on which they run and are independent of any specific architecture, which makes the cost of application migration independent of the size or number of applications being migrated. A small team of developers and architects can keep pace with technology change."

A business rules-centric system can be changed easily. A business rules developer can change one or many rules at a time and have that change availa-ble to all relevant business transactions, depending on target technology. By focusing on business rules, an analyst can distinguish the absolute dependenci-es in the rules from those that are interesting from a performance or user perspe-ctive.

A business rules system can be delivered quite easily in incremental pieces. If the first increment includes a solid data foundation, incremental system releases become the delivery of upgraded or additional rule sets to an existing infrastructure.

B. Von Halle (2001) on the multiple benefits of the rules-centric approach:

> "The ultimate payback of a business rules methodology is two-fold. The

first is a system development methodology that enables the discovery of essential intellectual process flow. The second is a system specifically designed to enable more spontaneous business change. A business rules methodology leads to the delivery of a system designed to change its rules, add new ones and retire old ones. A business rules approach puts the business back in charge of its destiny."

2.2 Configuration Specification Standards and Languages

There are many reasons why applications may require configuration – incorporating bug fixes and new implementations, accommodating environmental and technological changes, adapting to changing business requirements, internationalization and localization changes. To incorporate these dynamically changing requirements, a rules-centric approach captures refined rules in standard structured formats in building the business models.

Standards organizations and industry majors have evolved several widely adopted rule representation formats and languages. RuleML and the Simple Rule Markup Language (SRML) are gaining popularity among rules-centric system developers and are considered to be viable candidates for adoption by the World Wide Consortium as a technology standard.

Harold Boley, Benjamin Grosof, Michael Sintek, Said Tabet, Gerd Wagner (2002) about the RuleML design:

"RuleML encompasses a hierarchy of rules, including reaction rules event-condition action rules), transformation rules (functional-equational rules), derivation rules (implicational-inference rules) and queries (conclusion less- derivation rules), as well as integrity constraints. "

The RuleML hierarchy of general rules branches into the two direct categories of reaction rules and transformation rules. On the next level, transformation rules specialize to the subcategory of derivation rules. Then, derivation rules have further sub subcategories, namely facts and queries. Finally, queries specialize to integrity constraints.

Gerd Wagner, Grigoris Antoniou, Said Tabet, and Harold Boley (2004) on RuleML design:

"Given the linguistic richness and the complex dynamics of business domains, it should be clear that any specific mathematical account of rules, such as classical logic Horn clauses, must be viewed as a limited descriptive theory that captures just a certain fragment of the entire conceptual space of rules, and not as the only definitive, normative account. Rather, we need a pluralistic approach to the heterogeneous conceptual space of rules. Therefore, in RuleML, a family of rule languages capturing the most important types of rules are being defined. While these languages come with a recommended formal semantics, some of their rule bases may be marked to have a variant acceptable semantics. This will accommodate various formalisms based on non-standard logics, supporting temporal, fuzzy and other forms of reasoning. "

CommonRules is a rules-centric framework promoted by IBM and provides innovative XML interoperability and prioritized conflict handling capabilities. CommonRules also helps enable non-programmer business-domain experts such as marketing managers, to easily modify the executable business rules incrementally at run-time. CommonRules defines and supports a new XML rule interchange format for rules, called Business Rules Markup Language (BRML).

Hoi Chan (2002), the CommonRules project lead on the framework:

"IBM CommonRules is a rule-based framework for developing rule-based applications with major emphasis on maximum separation of business logic and data, conflict handling, and interoperability of rules. It provides a platform that enables the rapid development of rule-based applications through its situated rule engine via dynamic and real-time connection with business objects. CommonRules can be integrated with existing applications at a specific point of interest or it can be used to create applications composed only of rules."

A typical configuration specification of a software system will consist of the business rules repository accessible via standard interfaces by the rule engine and the metadata repository for application parameters including versioning and deployment specifications. A significant number of rule representation formats and organization-specific proprietary storage techniques and domain-specific languages exist.

Several companies such as IBM, Microsoft, ILOG, and Business Rule Solutions LLC are promoting rules-centric frameworks built on proprietary technologies and rule representation formats. Because of the diverse technological difference with which these companies operate, their frameworks are tailor-made for vendor-specific application domains.

In the rules-centric approach for developing highly configurable software, the lack of standards is becoming a major issue towards developing interoperable, reusable rules repositories and rule engines.

2.3 Rule Engines and Rule Execution

A rule engine determines how rules will be applied at runtime. All rule engines are basically pattern-matching search engines, looping through the rules, deciding which to use next and then repeating the process until some end condition is reached. However, there can be major differences in how patterns are searched for, and what happens when a rule is used.

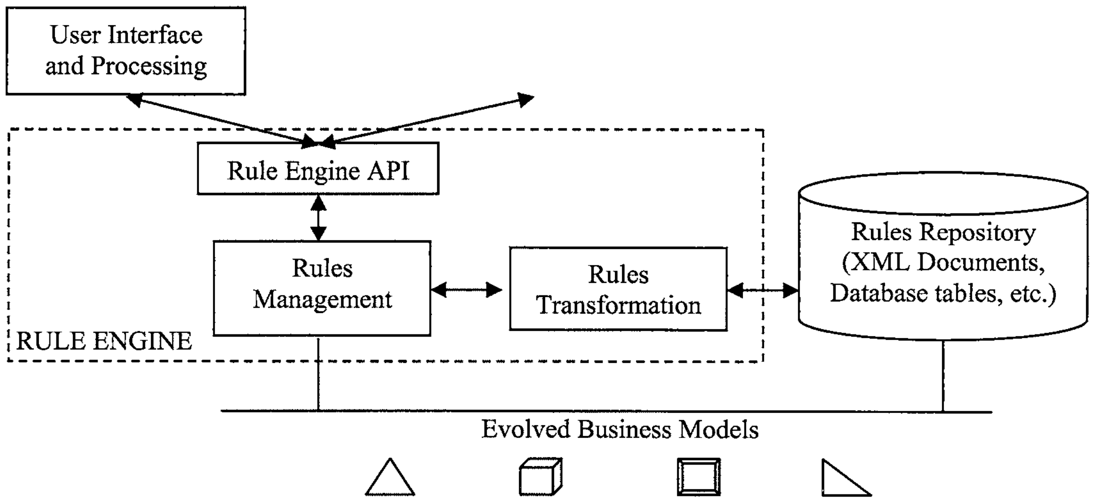


Figure 2.2 High-level architecture for a rules-centric framework

Rule engines are designed for integrating business processes and support incorporation of a knowledge base in the implementation and passing messages between processes. A general purpose rule engine will fit a wide range of proble-ms but might not fit very well for a domain-specific implementation.

The rule engine has an application program interface (API) that is used by the calling application. Dynamic business rules allow for building of expressions which are interpreted at run-time by the rule engines. This allows both develop-ers and application administrators to update and modify the business logic in an application quickly, without recompiling and redeploying the source code.

Several implementations of rule engines exist to cater to varied business models, technical environments and functional domains. Although no industry ad-opted standard specification for design of rule engines exist, industry majors and standards organizations are working on standardizing rule engine development practices.

SUN Microsystems, an industry leader for e-commerce and web based application technologies has taken initiative by providing a generic library applic-ation programming interface (API) for rule engine development.

The Java Specification Request (JSR-94) is a technology specification for Java language based rule engines targeting the J2EE and J2SE platforms. The rule engine for the implementation of a generic rules-centric framework for this thesis was developed using the JSR 94 specification of the Java Rule Engine API.

2.4 Maintainability of Rules and Configuration Specifications

Since application functionality is distributed across heterogeneous software and hardware platforms, interoperability issues are very important for impact analysis and implementation of software changes to accommodate changing requireme-nts.

Because the rules repository is a separate entity from the main application code, it can be easily updated. This results in a very flexible and responsive arch-itecture for maintenance of the rules repository.

A critical aspect of any rule engine is an API that can be called from other application components. This lets the rules repository be integrated into an app-lication context in a manner that allows the logic base, the rules repository, to be updated without requiring updates to the main application code.

Change impact analysis is the biggest challenge in maintenance of app-lications developed using the rules-centric approach. However, by capturing most of the interoperability dependencies themselves as rules, change management has been greatly improved.

S. Bohner (2002) lists the basic software change activities and impacts:

> "Activities such as understanding software with respect to the change, implementing the change within the existing system, and retesting the newly modified system has some element of impact determination. To understand the software with respect to the change, we must ascertain parts of the system that will be affected by the change and examine them for possible further impacts. Without requisite change impact analysis and

management mechanisms, software changes during maintenance can have unpredictable consequences that often delay their implementation".

The next chapter analyses several of the existing frameworks that support development of highly configurable software. Each considered framework is analyzed in detail with respect to a set of identified criteria.

CHAPTER 3

3.1 Analysis of Existing Frameworks

This chapter analyses several of the existing frameworks that support development of highly configurable software. A set of criteria based on which each framework is analyzed are identified and explained. Each considered framework is analyzed in detail with respect to the identified criteria. A few domain-specific frameworks and some of the native and proprietary rules and configuration specification formats are discussed. A research summary on the results of the analysis is provided. Finally, the scope for generalization and standardization of the frameworks are discussed. The chapter also proposes rule transformation methods that can be applied to the existing frameworks to further extend software configurability provided to a reusable level.

3.1.1 Analysis criteria for configurable software development frameworks

A software development framework is a specification or implementation that provides a general solution to some problem or aspect of applications. A toolkit or a platform is a collection of programming subroutine libraries that can be used to make development easier.

To analyze the various aspects of development frameworks, we need to identify a set of criteria that will consistently evaluate each framework from the st-andpoint of support provided for development of highly configurable software. To have these criteria as generic as possible and applicable to most of the existing frameworks, we need to isolate and avoid the criteria that are only specific to an application or functional domain. The set of identified criteria and their purpose is given below. Each development framework will be evaluated based on this set of identified criteria.

Approach used for development. This criterion evaluates the approach adopted by each framework in development of configurable software. This is an overall design strategy or paradigm used by the framework in development.

Rules representation and language. This criterion evaluates the rule lang-uage and representation format adopted by the framework. The representation format adopted can greatly influence the interoperability of the business rules.

Rule engines compatibility and support. Some frameworks have proprie-tary rule specification formats and hence have conforming rule engines that are closely tied with the framework. This criterion evaluates what rules engine a fra-mework has and with which other rule engines it is compatible.

Interoperability of business rules. This criterion evaluates the support for interoperability of business rules between applications developed using the fram-ework. Interoperability of business rules will greatly reduce the development time for new applications that closely match previously developed business rule mod-els using the framework.

Maintainability of business rules. This criterion evaluates the maintainability of the business rules repository developed using the framework. Tightly coupled rules are generally harder to maintain. The nature of how the framework captures the rules and their dependencies influences the maintainability of the rules when a change in a requirement needs to be accommodated.

Support for business rules transformation within the framework. Once a rules repository has been developed business models can be evolved based on specific set of implementation requirements. The ability to redefine the existing rules by applying certain standard transformations will facilitate reuse for each new business model development. This criterion evaluates this capability of a framework.

Support for business rules transformation outside of the framework. Though frameworks support transformation of rules within the framework's confined boundaries, there is little or no support for interchangeability or transformation of rules outside of the frameworks. This criterion evaluates a framework on this basis.

Support for remote administration of business rules. This criterion evaluates the support provided by the frameworks in administration of the business rules repository on a remote basis.

Support for change impact analysis. During the maintenance phase of software, change in requirements trigger analysis of the impact of those changes in the rules repository. This criterion evaluates the support provided by each framework in identifying and reporting the impact of changes.

Support for dynamic business rules. While most of the frameworks are designed to incorporate rules during execution from a static setup, there are frameworks which allow for dynamic incorporation of rule changes. This criterion evaluates a framework on this basis.

3.1.2 Analysis of frameworks and platforms supporting high configurability

Frameworks typically provide the skeleton of the structure and control flow of the implemented applications. Frameworks greatly reduce the size of the solution space available to the developer. For example, the J2EE framework manages the life cycle of an application component implemented in Enterprise Java Beans (EJB) all by itself, thereby relieving the programmer from coding and maintaining the infrastructure.

While most of the frameworks provide necessary support for development and maintenance of native applications, a detailed evaluation based on the identified criteria will provide comprehensive analysis on the suitability of each framework for specific implementation purposes.

The Gandiva software development system. One of the earliest adopters of the rules-centric framework design approach, Mark C. Little and Stuart M. Wheater (1996) proposed and designed a framework called the Gandiva software development system which is a software development framework that provides support for the construction of C++ software systems.

Gandiva supports development of configurable software using the delegate class mechanism for storing and maintaining control and workflow information as a separate entity.

Analysis of the Gandiva software development system

Analysis criterion	**Study/Evaluation**
Approach used for development	Design paradigm is rules-centric. Object oriented analysis approach for rule analysis.
Rules representation and language	Rules were represented as separate classes. Language used is C++.
Rule engines compatibility and support	Since rules were represented as C++ classes, they are compatible with any rule engine implemented in C++. This framework does not provide direct support for any other rule engines.
Interoperability of business rules	Since rules were represented as C++ classes, interoperability of rules is constrained to the C++ application domain. Business domains conforming to the object and data model used by Gandiva can use the rules repository for use with C++ applications and rule engines.
Maintainability of business rules	The class structure of C++ allows for easy and efficient management of the business rules in the repository. Rule maintenance can just be the usage of the correct accessors of the object model.
Support for business rules transformation within the framework	None
Support for business rules transformation outside of the framework	None
Support for remote administration of business rules	None
Support for change impact analysis	None
Support for dynamic business rules	None

Table 3.1 Analysis of Gandiva framework

This framework utilized object oriented design principles to separate the business rules from the application code. This modest framework cannot be evaluated completely because criteria that have been identified pertain to current standards and technologies. However, the generic design of the system will allow for partial evaluation.

A core part of this framework is that the binding between interface and implementation is configurable. Applications are written only in terms of interfaces, and although an application can request a specific implementation to be bound to an interface, it occurs in a way that allows this request to be changed without modifying the application.

Later, in 1998, Mark C. Little and Stuart M. Wheater migrated the Gandiva system to Java, enabling greater portability and dynamic rule management capabilities.

> "Gandiva provides a set of classes to support the construction and use of interface and implementation classes. The classes are responsible for storing and retrieving the configuration information required by an application, and the inventory, which is responsible for managing repositories of implementation classes and returning new instances to the application." Mark C. Little and Stuart M. Wheater (1998).

The Scalable and Agile Architecture for EBusiness (SAAFE) framework. SAAFE is a rules-centric software development framework that allows components to be dynamically replaced in an executing system. Thus components can be from multiple sources and the most suitable for the current requirements can be selected and used. This reconfiguration can occur on a per execution basis, thereby allowing the business process to be defined and optimized for each individual transaction.

A software system built in the SAAFE environment is described by a software template. This template provides sufficient information for the SAAFE rule engine to identify the required components and insert them into the executing application.

Analysis of SAAFE framework

Analysis criterion	**Study/Evaluation**
Approach used for development	Rules-centric approach, Component-based model development.
Rules representation and language	Software Template Language (STL) and the XML Pipeline Language.
Rule engines compatibility and support	XML Pipeline documents conforming to the XML Schema provided by SAAFE are compatible with rule engines.
Interoperability of business rules	Rules defined purely with STL are not interoperable while XML Pipelines act as an indirect way to share the rules with the outside world.
Maintainability of business rules	Highly maintainable.
Support for business rules transformation within the framework	Rule transformations are supported by the framework to a limited extent through the transformation engine built into SAAFE.
Support for business rules transformation outside of the framework	None
Support for remote administration of business rules	None
Support for change impact analysis	None
Support for dynamic business rules	None

Table 3.2 Analysis of SAAFE framework

This template models a business process as closely as possible with definition of rules. SAAFE uses Software Template Language (STL) which is based on the XML Pipeline Definition Language to describe and represent software templates.

A pipeline document describes the processing relationships between XML resources, specifying the inputs and outputs to XML processes and a pipeline.

> "The SAAFE project follows the principle of simplicity and describes a series of interactions between components in terms of the data they share. Components in the SAAFE system carry out a single task on one or more XML input documents. The schema of these documents must be one of a number of supported schemas. The SAAFE engine is capable of transforming between compatible schemas automatically." Ross Gardler , Nikolay Mehandjiev (2003).

MEDAL (UML Generic Model Transformer tool) and CASE tool extensions for model-driven software engineering. Standardization efforts in the modeling domain by the Object Management Group (OMG) have resulted in platforms and tools that support configurable software development and integration.

MEDAL is a model-driven software engineering and modeling platform that extensively uses standards such as the Unified Modeling Language (UML), the Meta Object Facility (MOF) and the XML Metadata Interchange.

Development using MEDAL covers the system, the application domain, and the requirements or the business rules from different viewpoints and levels of abstraction.

"MEDAL's component architecture handles dynamic class loading, manages the component instance life cycle, and provides a component lookup service. There is a clear separation between component specification, and component implementation" Nicolas Guelfi, Benoît Ries, Paul Sterges (2003).

Analysis of the MEDAL framework

Analysis criterion	**Study/Evaluation**
Approach used for development	Rules-centric, Object oriented.
Rules representation and language	Proprietary model template definitions
Rule engines compatibility and support	Only native rule engine
Interoperability of business rules	No interoperability support
Maintainability of business rules	Highly maintainable
Support for business rules transformation within the framework	Supports high level of rule transformations
Support for business rules transformation outside of the framework	None
Support for remote administration of business rules	None
Support for change impact analysis	Limited support
Support for dynamic business rules	None

Table 3.3 Analysis of MEDAL framework

MEDAL uses a native rule and transformation language for the business rules, represented as model template definitions, which define the rules as well as the primitive transformations.

"MEDAL's transformation language consists of the following parts:

The *transformation sequencing rules*, which define compound transformations as sequences of transformations that may either be primitive or compound transformations. The *model template application definitions*, which define, how to apply the primitive transformation in a context. The *param-*

eter definitions, which define the parameters that the compound transformation accepts." Nicolas,et al, (2003).

MEDAL supports maintenance and transformation of the business rules without having to know the details of how they are defined. Once the user has provided the parameter values, the transformations can be applied any number of times. The parameterized approach used by MEDAL's model template language is only suitable when one structural model is to be transformed into another structural model. Another limitation is that transformations can only add or modify elements but not remove them.

Compuware Corporation OptimalJ. OptimalJ is a rules-centric software development framework for applications targeted for the J2EE platform. OptimalJ implements the Object Management Group (OMG) Model Driven Architecture (MDA), offering companies enormous flexibility through vendor- and language-independent interoperability. OptimalJ supports rapid application change and on-going maintenance. OptimalJ uses a rule engine that supports pattern analysis and dynamic rule invocation.

"OptimalJ identifies rule patterns which facilitate reuse, leveraging pre-defined designs, structure and code, and capture specific knowledge about the architectures, platforms and technologies to help create reusable code and ability to redefine business rules for reuse throughout the application." Compuware OptimalJ (2004).

Analysis of the OptimalJ framework

Analysis criterion	**Study/Evaluation**
Approach used for development	Rules-centric, Object Oriented
Rules representation and language	Native format. Unified Modeling Language support.
Rule engines compatibility and support	Only native rule engine.
Interoperability of business rules	Highly interoperable format.
Maintainability of business rules	Highly maintainable.
Support for business rules transformation within the framework	None
Support for business rules transformation outside of the framework	None
Support for remote administration of business rules	Limited support.
Support for change impact analysis	None
Support for dynamic business rules	None

Table 3.4 Analysis of OptimalJ framework

SUN Java 2 Enterprise Edition (J2EE) Platform. J2EE is a framework and platform for developing configurable enterprise applications for the web. The J2EE platform is a collection of related technology specifications that describe required APIs and policies universally available for use.

"The portability required by the J2EE specification gives customers the freedom to choose technologies both at system construction time and through the application lifecycle. The combination of specification, reference implementation, and compatibility tests provide the consistency necessary for a portable, open application platform." Mark Johnson (2003).

SUN Microsystems, the provider of Java has made efforts to create a standardized rule engine API for the Java platform. In November 2003, the Java Community Process released the final version of the JSR-94 specification. JSR-

94 formalizes a basic API for working with rule engines in Java. The flexibility offered by the API allows rule engines to support development of continuously adaptive systems to changing needs.

Analysis of the J2EE framework

Analysis criterion	**Study/Evaluation**
Approach used for development	Rules-centric, Object oriented
Rules representation and language	XML
Rule engines compatibility and support	Any Java based rule engine conforming to the JSR 94 specification.
Interoperability of business rules	Highly interoperable.
Maintainability of business rules	Highly maintainable.
Support for business rules transformation within the framework	Limited XSLT support
Support for business rules transformation outside of the framework	None
Support for remote administration of business rules	Application servers support high level of remote administration
Support for change impact analysis	None
Support for dynamic business rules	Limited support

Table 3.5 Analysis of J2EE framework

Microsoft BizTalk Server. Microsoft BizTalk Server supports the goal of creating business processes that unite separate applications into a coherent whole by including a mechanism for specifying business rules and better ways to manage and monitor applications.

The Business Rule Framework within BizTalk Server represents an innovative implementation of the Service Oriented Architecture (SOA) paradigm. Every individual and combined level of functionality has been designed to be exposed, independent, and loosely coupled XML definitions, thus eliminating the need for any procedural implementation programming.

Analysis of the BizTalk Server framework

Analysis criterion	**Study/Evaluation**
Approach used for development	Rules-centric, Service Oriented Architecture
Rules representation and language	XML
Rule engines compatibility and support	Only Microsoft
Interoperability of business rules	Limited support
Maintainability of business rules	Highly maintainable
Support for business rules transformation within the framework	Limited support.
Support for business rules transformation outside of the framework	None
Support for remote administration of business rules	Full support.
Support for change impact analysis	Limited support.
Support for dynamic business rules	Limited support.

Table 3.6 Analysis of BizTalk framework

"The BizTalk process requires modeling business procedures, analyzing data communication formats, mapping formats, and properly configuring BizTalk Server. When the process flow is followed, BizTalk Server can solve business application maintenance and information exchange challenges". Rand Morimoto, Microsoft (2004).

Most modifications to a business process life cycle in BizTalk Server pertain to changes in business rules. Because conventional applications embed business rules in opaque procedural code, the rules cannot easily be accessed or modified without disrupting running processes.

IBM CommonRules Platform and the Rational Unified Process (RUP). The IBM CommonRules framework uses Java based technologies to provide support for configurable software development, interoperability, portability and dynamic rules by way of dynamic linking between any objects instances. Larger applications can be constructed by merging small modules of business logic dynamically, which enables the sharing of business logic between applications.

The framework also completely hides the programming details and exposes only the business logic as human language like syntax.

> "CommonRules is a rule-based framework for developing rule-based applications with major emphasis on maximum separation of business logic and data, conflict handling, and interoperability of rules. It is a pure Java library, and it provides a platform that enables the rapid development of rule-based applications through its situated rule engine via dynamic and real-time connection with business objects. CommonRules can be integrated with existing applications at a specific point of interest, or it can be used to create applications composed only of rules. CommonRules uses a semantically-rich rule language called Courteous Logic Program (CLP) for rule representation and provides a set of APIs for efficient application integration and data bindings based on the RuleML specification, in order to enable interoperability of different rules"
> Hoi Chan (2002).

Analysis of the CommonRules framework

Analysis criterion	**Study/Evaluation**
Approach used for development	Rules-centric, Object oriented
Rules representation and language	Native - Courteous Logic Program (CLP)
Rule engines compatibility and support	Only native
Interoperability of business rules	Limited support.
Maintainability of business rules	Highly maintainable.
Support for business rules transformation within the framework	Limited support.
Support for business rules transformation outside of the framework	None
Support for remote administration of business rules	Limited support.
Support for change impact analysis	None
Support for dynamic business rules	Limited support.

Table 3.7 Analysis of CommonRules framework

CommonRules extensively uses the Rational Unified Process(RUP), which is an iterative software design method created by the Rational Software Corporation, now a division of IBM. It describes how to deploy software effective-ly using commercially proven techniques. RUP enables remote deployment and maintenance of applications developed using CommonRules. The standards used are interoperable with several hardware and software platforms.

ILOG Business Rule Management System. ILOG's Business Rule Management System (BRMS) treats business rules as a corporate asset. Instead of simply updating rules when conditions change, ILOG BRMS lets business users manage rules throughout their entire lifecycle. Business rules are express-ed as English-syntax instead of application code.

“With ILOG BRMS, policies and practices are expressed as English-syntax business rules instead of computer code. The same rules are accessed by every organization in the enterprise, across touch points and applications. Developers, analysts, managers and administrators use powerful editing tools to manage, track and change rules. Teams employ consistent regulations and practices, ensuring prompt compliance. Business analysts change decision logic themselves, relieving overworked IT departments. New policies and regulations are implemented quickly and accurately" Jolif and Tissandier (2004).

Analysis of the ILOG Business Rule Management System framework

Analysis criterion	**Study/Evaluation**
Approach used for development	Rules-centric
Rules representation and language	Native format
Rule engines compatibility and support	Only native rule engine
Interoperability of business rules	Highly interoperable
Maintainability of business rules	Highly maintainable
Support for business rules transformation within the framework	Limited support.
Support for business rules transformation outside of the framework	Limited support.
Support for remote administration of business rules	Full support.
Support for change impact analysis	Limited support.
Support for dynamic business rules	Full support.

Table 3.8 Analysis of ILOG BRMS framework

Oracle Service Oriented Architecture (SOA) Platform and the BPEL. Oracle and the Business Process Execution Language (BPEL) is the first and foremost business orchestration technology. BPEL's major promise is that the creation of abstract and executable schemes can be defined as business processes and run in any compliant engine. The ability to specify those aspects of a shared process in a technology-neutral manner is very advantageous.

Analysis of the Oracle SOA Platform and the BPEL

Analysis criterion	**Study/Evaluation**
Approach used for development	Rules-centric and SOA
Rules representation and language	XML and BPEL
Rule engines compatibility and support	Limited support for other rule engines.
Interoperability of business rules	Highly interoperable
Maintainability of business rules	Highly maintainable
Support for business rules transformation within the framework	Full support.
Support for business rules transformation outside of the framework	Limited support.
Support for remote administration of business rules	Limited support.
Support for change impact analysis	Limited support.
Support for dynamic business rules	Full support.

Table 3.9 Analysis of Oracle BPEL framework

Because a BPEL instruction set is an XML representation of a process with a precise language and grammar structure, it provides a readable and understandable instruction set for documenting a process. Each process is independently valuable, but when combined with other processes has the potential to facilitate wholesale efficiencies and provide innovative solutions to numerous challenging problems.

3.2 Rules and Pattern Analysis on Existing Frameworks

Rules-centric frameworks validate the rules that are part of the repository for ambiguity and correctness. The validated rules repository will serve as the basis for the analysis and decision making process depending on the context in which the rules are executed.

B. Von Halle (2001) provides a set of comprehensive criteria to validate rules:

"Criteria against which to validate each rule:

Relevant/justified. Each rule must be essential to the target scope of analysis.

Atomic. Each rule must represent one thought such that an actor (human or electronic) can apply the rule in guiding behavior.

Declarative. Each rule must prescribe a decision or computation rather than dictate a procedure for performing and enforcing the decision or computation.

Intelligible/precise. The rule's intended audience must understand it such that the rule is predictable and repeatable in its usage.

Complete. Each rule must possess all intellectual properties necessary for usage.

Reliable. Each rule must originate from a source authorized to decide that the rule is as the business desires.

Authentic. As each rule is copied into various forms (natural language, templates, declarative specifications, executable code), each representation must remain faithful to the original intent of the rule"

Identifying rule patterns is a technique for conducting rules analysis. Using rule patterns, the quality of the rules repository can be greatly improved. Rules can be enhanced for completeness by revisiting the rule-enriched business data model, primarily for missing constraint rules.

Redundant rules within one rule pattern can become obvious and can be generalized. Overlapping rules, which is a subtle form of possible rule redundancy, can be detected if rules are grouped into patterns appropriately.

B. Von Halle (2001) on how the rules-centric approach supports rule analysis:

> "The business rules approach advocates the tremendous amount of business value in analyzing rules. After all, the rules of the business represent its decision-making capacity and govern how the business behaves with respect to its internal people and external partners and customers. The rules are a strong basis for business process reengineering as well as the transformation of systems from one technology to another."

The next chapter presents a discussion on the imperative and declarative programming paradigms. The methodology adopted for the proposed generic framework is discussed and a detailed design is presented.

CHAPTER 4

4.1 Methodology of Proposed Generic Framework

This chapter begins with a discussion of the imperative and declarative programming paradigms. Since the proposed framework will use a declarative language to represent the business rules in the rules repository, this chapter identifies some of the benefits of using a declarative language and discusses the drawbacks associated with imperative programming- the paradigm that is predominantly used in software development. This discussion also presents the scope for a generic framework that is available through the use of a declarative language in a rules-centric approach to software development.

In Chapter 3, some of the existing frameworks supporting development of highly configurable software were discussed and evaluated with respect to a set of identified criteria. The shortcomings of each framework analyzed in Chapter 3 and how the proposed generic framework addresses them are discussed.

The methodology adopted for the proposed generic framework is discussed and a detailed design is presented. The adopted rule representation format for the generic framework – the Simple Rule Markup Language (SRML) and its document structure in the rules repository is presented.

The Rete algorithm is a widely used pattern matching algorithm that the generic framework uses for matching the assertions on facts with the conditions present in the rules. This chapter presents a detailed discussion and examples on how the Rete algorithm was used in the implementation of the rule engine.

The design of the rule engine which is based on the JSR-94 Java Rule Engine API specification from SUN Microsystems is discussed. This chapter also discusses how the rules in the rules repository are parsed and executed in the memory model and the execution context that the generic framework employs.

Finally, the support provided by the generic framework for rule extensions and rule transformations is discussed.

4.1.1 Scope for a generic framework

Imperative programming, as opposed to Declarative programming, is a programming paradigm that describes computation in terms of a program state and statements that change the program state, with a sequence of commands for the computer to execute. Imperative programming requires the programs to specify *how* things are computed as compared to Declarative programming which requires the programs to specify *what* is to be computed, without dependence on the implementation of solutions to computations.

Imperative programming and maintenance issues. Maintenance cycles in imperative programs are typically lengthier because of the complex impact analysis required to isolate sections of application code that need to be modified in order to adapt to changes in requirements. Identification and understanding of the business rules embedded in the application code becomes a tedious and error-prone process because of the underlying complexity of the implementation methodology and environment.

Anthony A. Aaby (1996) states the drawbacks of imperative programming from a software maintenance perspective:

> "Imperative constructs jeopardize many of the fundamental techniques for reasoning about business logic. The embedded logic descriptions are complex and it is this complexity that provides a strong motivation for functional and declarative programming as alternatives to the imperative programming paradigm."

The core maintenance issue with imperative programming is the nature of the paradigm to bind solutions directly with problems, which deter the clear separation of business logic from the application code.

Declarative programming and rules-centric development. Declarative programming is an approach to programming that involves the creation of a set of conditions that describes a solution space, but leaves the interpretation of the specific steps needed to arrive at that solution up to an unspecified interpreter.

Declarative programming thus takes a different approach from the traditional imperative programming which requires the programmer to provide a list of instructions to execute in a fixed manner and order for specific solutions. Declarative languages describe relationships between variables in terms of functions, inference rules or term-rewriting rules. The language executor – an interpreter or compiler - applies a fixed algorithm to these relations to produce a result.

The language executor in the case of this thesis is the rule engine, which applies a fixed algorithm for rule parsing and execution. Rules and variable relationships are represented in a declarative language highly compatible with the rule engine- the Simple Rule Markup Language (SRML). SRML supports representation of business rules in an extensible and reusable format.

Existing standards for rule engine development allow software developers to design rule engines that are compatible with one or more rule specification formats. The Java Specification Request (JSR-94) is a specification standard for rule engines based on the Java language. This specification however, does not have any native rule representation format and hence is highly interoperable. Rules represented in several formats from plain XML to standard-based formats such as SRML can be used with a rule engine developed using the JSR-94 specification.

4.2 Proposed Design for a Generic Framework

The design methodology for the proposed generic framework is based on the rules-centric approach to software development. The business rules are translated into declarative statements conforming to the SRML standard. The rule sets are then stored in a rules repository from which the rule engine can access the rules for dynamic execution.

The generic framework employs the Rete algorithm developed by Dr.

Charles L. Forgy of Carnegie Mellon University in 1979. The Rete algorithm is used in the rule engine implementation for pattern matching of the facts represe-nted in the rules repository with the attributes of the asserted runtime objects.

Design methodology. The design methodology adopted for the generic fra-mework follows the rules-centric approach to software development. The generic framework does not have any dependence on the rule representation format or any specific application domain. The rule engine implementation is based on a standard specification and is interoperable on any Java based software system. The generic framework uses a rule engine that supports pattern analysis and dynamic rule invocation. Patterns facilitate reuse of the structure and re-modeling of the captured business logic and the rules repository. A core part of the frame-work is that the binding between rules and implementation is configurable. Applications are written only in terms of interfaces to the rules repository and an application can request a specific set of rules to be executed at run-time based on the context.

The generic framework places emphasis on maximum separation of busi-ness logic and data or processing logic. The rule engine is a pure java implemen-tation and hence can have dynamic and real-time connection with the rules repo-sitory, allowing for dynamic incorporation of changes to the business rules at runtime.

Business rules repository and SRML. Business rules are complete statements that enable new knowledge, enable action or disable action based on previous knowledge. The Simple Rule Markup Language (SRML) is a XML based standard for representing rules. SRML follows a simple schema for the rule structure. Typically, a business rule represented in SRML will have the structure as shown in Figure 4.1.

```
Rule Set
        Rule Name
                Conditions
                Consequence
        Rule Name
                Conditions
                Consequence
        ....
```

Figure 4.1 SRML document structure

SRML describes a generic rule language consisting of the subset of language constructs common and useful to the popular rule engine implementations. Because it does not use constructs specific to any proprietary vendor language or engine, rules specified using SRML can easily be translated and executed on any conforming rule engine, making it useful as a standard for rule exchange format for Java-based rule engines.

The rule set is the root element of a SRML document which encloses the list of all rules defined in the document. Rules have a *condition* section and a *consequence* section, with a constraint that the condition section must have at least one condition. Conditions are composed of test expressions and can be *simple* conditions or *composite* conditions. Simple conditions can be bound to variables while composite conditions cannot. The consequence section of a rule consists of actions, which can be variable declarations and assignments, as well as the traditional assert, retract and modify statements of rule languages. Figure 4.2 shows an example of rules represented in SRML.

```
<rule-set name="Rule Set Example">
        <rule name="Rule for not disturbing a person">
        <example:condition personName="Narsi">
                <example:personIsReading/>
                <example:personIsSleeping/>           Condition Section
        </example:condition>

        <example:actions personName="Narsi">
                <example:doNotDisturb/>               Consequence Section
        </example:actions>
        </rule>
        <rule name="Rule for talking">
        <example:condition personName="Narsi">

<example:personIsAwake/><example:personIsIdle/></example:condition
>
        <example:actions personName="Narsi"><example:doTalk/>
        </example:actions>
</rule> </rule-set name>
```

Figure 4.2 Example of rules represented in SRML

This rule specifies two conditions that need to be asserted for an action to be executed. For the action *doNotDisturb* given in the first rule to be executed, both the assertions *personIsReading* and *personIsSleeping* must evaluate to be true. The language provides constructs to specify more assertions and action scenarios for accommodating complex business rules representations using logical expressions.

"Over the last few years, rule languages and engines have become increasingly popular tools for implementing business rule applications where the business logic (business rules) is very dynamic. These applications support the definition of business policies by non-technical users in business rule languages and enable users to personalize their preferences for everything from content to navigation through easily understandable ways"

Margaret Thorpe and Changhai Ke (2004).

4.2.1 Rete algorithm and usage

The Rete algorithm is an efficient pattern matching algorithm for implementing rules-centric systems. Implementation of a rules-centric system will typically require the rule engine to check each rule in the rules repository against the known facts in the execution context. The rules are executed if necessary with looping back to the first rule when finished. For even moderate sized repositories, this approach performs far too slowly.

The Rete algorithm provides the basis for a more efficient implementation of a rules-centric system. A Rete-based system builds a network of nodes, where each node except the root corresponds to a pattern occurring in the conditions section of a rule. The path from the root node to a leaf node defines a complete conditions set of a rule. Each node has a memory of facts which satisfy that pattern. As new facts are asserted or modified they propagate along the network. When a fact or combination of facts causes all of the patterns for a given rule to be satisfied, a leaf node is reached and the corresponding rule is triggered.

A Rete network can be seen as a graph through which data flows. Data is specified using tuples which express attributes about objects. For example, tuples may be used to express a person's name and his car. The tuples in the Rete network that reach the far end cause the firing of a rule.

A Rete network is comprised of two types of nodes:

1-input/1-output nodes. The 1/1 nodes are constrictive nodes that only allow matching tuples to flow through. Any tuples that do not match are discarded by the node.

2-input/1-output nodes. The 2/1 nodes simply connect the output arcs from two other nodes (either 1/1 nodes or 2/1 nodes) merging tuples from both the left and right incoming arcs into a single tuple on the outgoing arc. These types of nodes also maintain a memory of tuples for matching against future facts.

Each condition of a rule is merely a pattern for a particular tuple type. The condition describes the attributes that a tuple must have and acts as a filter. Each condition is transformed into a 1/1 node that only allows tuples matching the specified attributes to pass. An attribute value may be specified as a variable and implies that the variable must hold the same value in all occurrences. Figure 4.3 is an example representation of a Rete network.

Consider a Rule:

Rule #1: 'For any person who has a truck that is of the same brand name as that person's friend's car, perform an action'.

This could be expressed with the condition patterns of:

(1) (person name=personName friend=friendName)

(2) (person name= personName truck=brandName)

(3) (person name= friendName car= brandName)

Figure 4.3 Example Rete network

Condition pattern (1) models the friend relationship so that the rule only applies to persons who have friends. The *person* and *friend* tokens are variables that must be consistent across any set of tuples that match this rule. Condition patterns (2) and (3) serve two roles. The truck and car attributes share the same *brandName* variable and serve to identify two people who have a truck and a car with the same name.

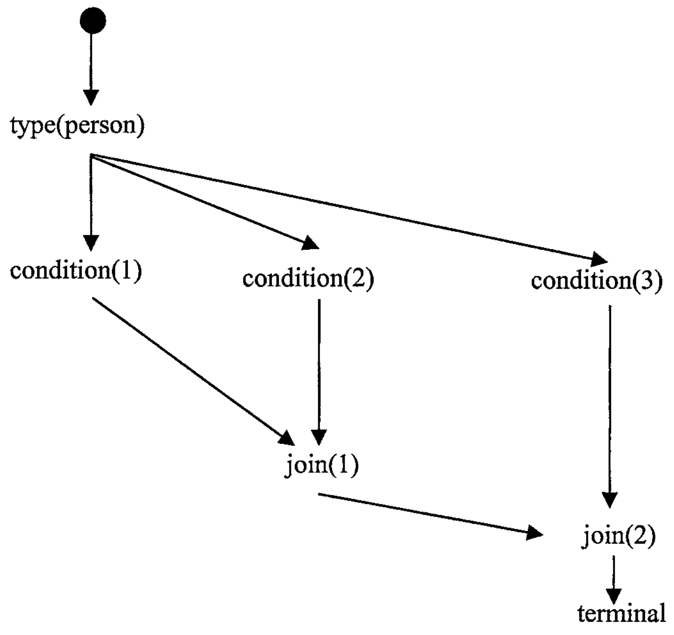


Figure 4.4 Nodes and condition evaluation in a Rete network

Type	**Person**	**Friend**	**truck**	**car**
tuple set #1				
person	narsi	naveen	ford	null
person	naveen	narsi	null	ford
tuple set #2				
person	narsi	naveen	ford	null
person	naveen	narsi	null	nissan

Table 4.1 Example tuple sets

They each contain a *name* attribute with either the variable *person* or *friend* which ties the last two conditions back to the first two. Figure 4.4 shows diagrammatically how conditions at each node are evaluated in a Rete network.

If the two tuple sets given in the example (Table 4.1), were asserted against Rule #1, then *tuple set #1* would cause a firing of the rule where *tuple set #2* would not.

In both cases the two tuples would pass node condition(1), as the nodes simply associate the *person* and *friend* variables with the appropriate values from each tuple.

The join(1) node would allow both tuples to merge and propagate past it in both the first and second case. Additionally, for both the tuple sets, the tuple with the *person* 'narsi' would pass node condition(2) and the tuple with *person* 'naveen' would pass node condition(3).

The join(2) node is where the two tuple sets differ. In the first set, nodes condition(2) and condition(3) have each associated the value of 'ford' to the *brandName* variable.

In the second set, the two nodes have different values assigned to the variable. The join(2) node only allows those tuples that have consistent associations with all variables to pass.

The Rete algorithm is designed to sacrifice memory for increased speed. In most cases, the speed increase is several orders of magnitude because Rete performance is theoretically independent of the number of rules in the system.

4.3 Rule Engine Design

A rule engine will have to possess adequate knowledge of the structure of a SRML document on which the parsing algorithms are applied. This structural information for parsing is available directly from the Document Type Definition (DTD) specification of the SRML document standard.

A DTD specifies and defines the valid structure of a document format. The complete DTD specification for an SRML document is available in Appendix A. The primary functions of a rule engine include parsing the rule sets defined in the rules repository - an SRML document - binding the rules with the execution context at runtime and executing the rules within the working memory of the execution context. The execution phase involves construction of the Rete network by adding nodes from the conditions present in the rule set to the network and evaluating each node as it propagates in the network, until leaf level where the rules are executed. The working memory of the network is designed to store intermediate values based on the assertions performed as nodes propagate in the Rete network.

Rule engines and the Java Rule Engine API (JSR-94) specification. The Java Rule Engine API defines a multi-step protocol for invoking a rule engine, adding facts to the engine, triggering rules and getting the results back. It is equivalent to a Java Database Connectivity (JDBC) API for relational database management systems access. The Java Rule Engine API specification provides ways to accomplish the goal of vendor independence by allowing developers to build a rule engine API adapter such that only the adapter will need to be rewritten should an application domain require change in the rule engine architecture.

SUN Microsystems release statement on the Java Rule Engine API specification:

> "In November, the Java Community approved the final draft of the Java Rule Engine API specification (JSR-94). This new API gives developers a standard way to access and execute rules at runtime. As implementations of this new specification ripen and are brought to the market, programming teams will be able to pull executive logic out of their applications. Instead of rushing changes in application behavior through the development cycle and hoping it comes out correct on the other end, executives will be able to change the rules, run tests in the staging environment and roll out to production as often as becomes necessary." Rupp (2005).

The JSR-94 specification is by no means complete, but it gives a unified front end to client applications for plugging into different rule engines at runtime. It also supplies a standard way for rule authors and administrators to build and deploy groups of rules in a runtime environment.

The JSR-94 supports portable rules and rule sets. This means a standard rules language with standard semantics such as the SRML should be able to switch between rule engines relatively easily with no changes to rules or the calling client code. The ability to store rules in a variety of persistent storage methods such as a Relational Database Management System (RDBMS), a Lightweight Directory Access Protocol (LDAP) server, an XML database, an

Object database or a File system allows migration and interchange of rules from one format to another to adapt for changes made to the rule engines or the environment.

Rule engines provide applications the flexibility to continuously adapt to changing requirements and the ability to spontaneously and appropriately react to the changes. Rule engines also allow developers to write Java expressions directly in the XML descriptor for a rule set which are evaluated at run-time by the rule engine.

The JSR-94 specification also has extensive native support for developm-ent based on the Unified Modeling Language (UML), by allowing rules to be spe-cified in the Object Constraint Language (OCL) and allowing relationships betwe-en objects and other UML artifacts to be constrained by the rule sets.

Rule engine architecture. The rule engine architecture proposed by the JSR-94 specification is highly scalable and generic. The specification carefully eliminates any dependence on vendor-specific standards or rule representation formats. With the JSR-94 specification for Java based rule engine development, the developer is free to choose any of the rule representation formats that may be suitable for the target application domain. The API provides the flexibility to tune the rule engine features according to the needs of the application domain where it will be used.

Although the JSR-94 specification recommends the Rete algorithm as the native parsing algorithm for pattern matching in the rules, it is possible to use any other algorithm that may be more suitable for the adopted rule representation format or target application domain. For example, a rule engine implementation operating on a rules repository represented using a standard RDBMS table with rows and columns format may use SQL queries, and other rule parsing techniqu-es applicable to the application domain, instead of building Rete networks and apply the Rete algorithm for pattern matching.

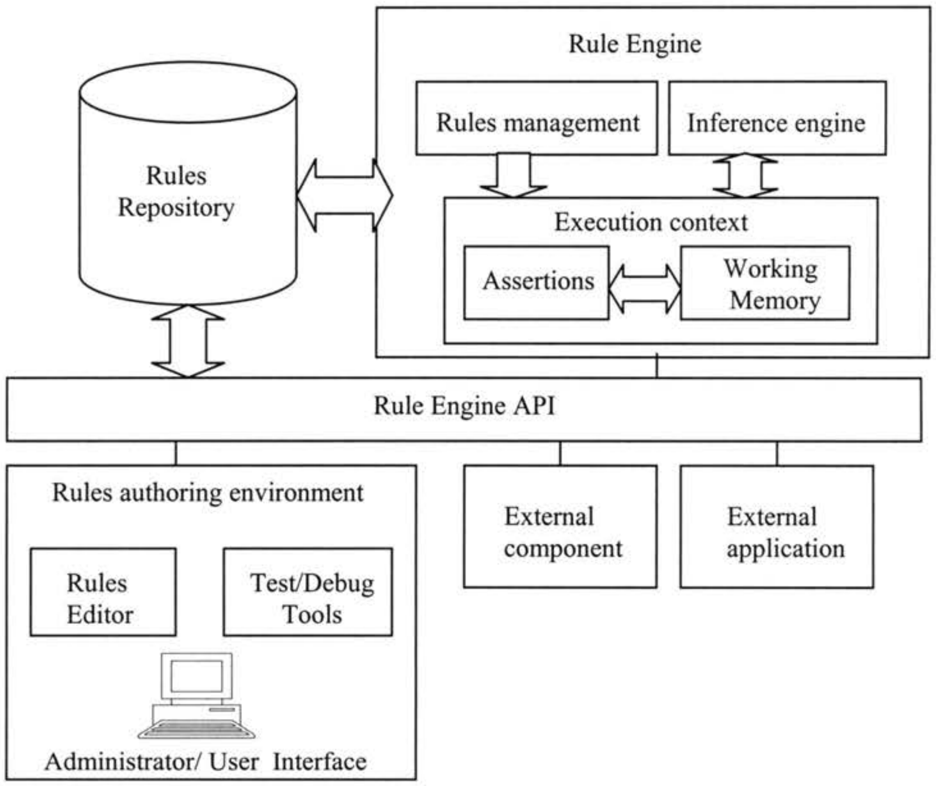


Figure 4.5 Rule engine architecture of the generic framework

The proposed generic framework uses the Rete algorithm for rule parsing because of the rich API support as well as its suitability for the SRML document structure, which is an XML based standard.

The rule engine architecture of the generic framework conforming to the JSR-94 specification (Figure 4.5) consists of the following modules:

Rules repository. The rules repository module manages the storage and retrieval of rules. The repository is accessible to the rule authoring environment through the rule engine API.

Rule engine. This module is the implementation of the Rete algorithm and rule parsing using the Java Rule Engine API.

Execution context module. This represents the runtime environment for the inference engine's execution. During the inference engine's execution cycle, an execution context will hold a grouping of objects in the working memory. More than one execution context can simultaneously exist and share the same rules-set.

Inference engine. This module performs object assertions for rules in the execution context, based on the facts in the working memory. If a rule executes, then it adds more facts into the working memory. These facts are then used to match more rules until the complete Rete network is visited and where no more rules can be matched with facts from the working memory.

Rules authoring environment. This module is the administrative user interface with which to define and maintain rules in the rules repository. Most currently available rule engines include a rules editor to compose rules in high-level rule languages that usually includes an English-like syntax. Testing and debugging functions let the user build test scenarios to simulate the effect of the rules in a real environment.

4.4 Parsing Rules - Rete Networks and the Working Memory Model

While it may be simple to create a rule engine that allows specification of business logic in a format that is suitable for understanding by business analysts as well as the rule engine efficiently, the pattern matching of the rules in the application context with the assertions based on facts is still very complex without a good algorithm.

First, the rule engine must be made aware of its environment, typically through fact assertion which consists of the program asserting facts into a rule session or Working Memory.

Working Memory. Whenever a fact is asserted, retracted or modified within the context of a rule session, the results of the intermediate assertions have to be stored and propagated with the nodes in the Rete network.

A Working Memory is a Java rule session that internally maintains and stores the Rete network in memory whenever a rule set is being evaluated for execution. In this process, many rules in the rule set may become candidates for firing, or may have become invalidated.

A simplistic approach is to reevaluate all rules against the entirety of the working memory. This method is guaranteed to be correct but is also certainly sub-optimal because any individual fact modification only affects a small number of conditions in a small number of rules.

However, the Rete algorithm allows the rule engine to maintain a memory of intermediate results from partial rule matches across time. Reevaluation of each condition is not necessary, as the engine knows which conditions might possibly change for each fact, and only those must be reevaluated.

4.4.1 Rule execution and run-time binding

The rule engine interacts with a rule engine adapter which acts as an application-specific interface to execute business rules stored in the rules repository. The rule engine API abstracts the implementation details of rule engine interaction including initialization, invocation and removal throughout the lifecycle of a rule without any dependency on the environment or specific implementation standards.

In effect, the rule engine allows the rule administrator to register a set of rules that will trigger responses when conditions specified in the rules are met. When the rule engine is passed the set of data for evaluation, it examines the set of data, finds which conditions are met and fires the rules as appropriate.

The rules are evaluated in an order established by the rule engine with the data and its current values, which also relieves rule engine developers from having to deal with the ordering issues. The run-time binding of rules in this manner with the execution context allows the rule engine to adapt to dynamic changes made to the rules repository.

4.5 Support for Rules Extension and Transformation

Once the business rules are represented in the rules repository, the rules are available for reuse with target applications. Typically, changes in business requirements will cause the rule definitions to be extended or modified allowing for the changes to be reflected in the rules repository. Representation formats such as the SRML allows for easier extension of the rule definitions because of the nature of the XML document structure.

Although many of the existing frameworks support rule maintenance through a rule authoring interface, there is little support for transforming existing rules in order to reuse the business logic in scenarios that are applicable. The rule authoring interface provided by the generic framework supports maintenance of the rules repository as an administrative task including support for rule transformation.

The importance of the support for transformations is increasingly being felt in the domains where applications change very often. Tried and tested models have good business rules repositories for specific target domains. However, to reuse rules repositories, the rules need to be transformed to the format that is understandable by applications that operate on them typically in heterogeneous environments.

The Extensible Stylesheet Language-Transformations (XSLT) is a XML based standard proposed by the Worldwide Web Consortium (W3C) for transforming XML documents between various formats. Because of the chosen rule representation format – SRML, the generic framework proposed in this thesis supports the XSLT language based transformations that can be applied to the rules in the rules repository.

The next chapter presents the results and analysis of the proposed framework by using two case studies from different application domains. These case studies demonstrate the support provided by the proposed generic framework in highly configurable software development.

CHAPTER 5

5.1 Results and Analysis

This Chapter presents the results of the analysis of the generic framework by conducting two case studies from contrasting target application domains. The case studies in effect will demonstrate the capabilities of the generic framework and how the methodology differs from conventional software development. The maintenance issues and the problems associated with embedded business logic are pointed out.

Each case study begins by analyzing the implementation based on the conventional approach. The sections of the application implementation where the business logic or the functional rules are embedded in code are identified. Sections of code of this nature are the main source for identification of business rules in a rules-centric development approach.

The implementation based on the rules-centric approach using the proposed generic framework is then discussed for each case study. The business rules are separated and represented in the SRML document format. The class mechanism of the implementation and how rules are dynamically retrieved and executed is explained. In effect, this chapter outlines how the rules-centric approach to design can neatly and clearly separate business logic from the application and data processing logic and can be maintained as an external rules repository.

The rules repository will manage the storage and retrieval of the rules through the rule engine API. The rules authoring interface for each case study is presented and a discussion of how the rules can be defined and managed through the interface is presented.

Since the rules authoring interface is a web-based application, rules can be maintained remotely while the target application can dynamically adapt to changes made to the rules repository.

5.1.1 Case Study 1: Conway's Game of Life

This case study is the analysis of an implementation of the popular mathematical game - Conway's Game of Life. The game was chosen as a candidate for case study because of the concise and small set of rules the game follows. With this small rule set, the differences between a conventional development approach and a rules-centric approach using the generic framework can be better explained as compared to a complex application example where the emphasis may be shifted to underlying application complexity rather than the development approach.

Conway's Game of Life. The Game of Life is not a typical computer game. It is a 'cell automaton' of the cells in a grid in accordance to the rules of the game. This game was invented by Cambridge mathematician John Conway and became widely known when it was mentioned in an article published by Scientific

American in 1970.

It consists of a collection of cells in a grid which, based on a few mathematical rules, can live, die or multiply. Depending on the initial conditions, the cells form various patterns throughout the course of the game. It has often been claimed that since 1970 more computer time world-wide has been devoted to the Game of Life than any other single activity.

Martin Gardner (1970) wrote:

"The game made Conway instantly famous, but it also opened up a whole new field of mathematical research, the field of cellular automata. Because of Life's analogies with the rise, fall and alterations of a society of living organisms, it belongs to a growing class of what are called 'simulation games' - games that resemble real-life processes."

The basic idea of the game is to start with a simple configuration of living cells which are placed on a 2D grid by various methods. This constitutes the first generation. Conway's 'genetic laws' for births, deaths and survivals - the four rules of the game - are then applied to the pattern and the next generation pattern is placed accordingly.

The rules of the game. The rules of the game are simple and elegant:

Kill the Lonely - Any live cell with less than two neighbors dies of loneliness.

Kill the Overcrowded - Any live cell with more than three neighbors dies of crowding.

Give Birth - Any dead cell with exactly three neighbors comes to life.

Surviving Cells - Any live cell with two or three neighbors lives, unchanged, to the next generation.

All births and deaths occur simultaneously. Together they constitute a single generation of the initial configuration. This game is a zero-player game, meaning that its evolution is determined by its initial state, needing no input from human players. It runs on a grid of square cells which stretches to infinity in all directions. Each cell has eight neighbors, which are the cells adjacent to it (including diagonally). Each cell can be in one of two states: it is either *alive* or *dead*.

The state of the grid evolves in discrete time steps. The states of all of the cells at one time are taken into account to calculate the states of the cells one time step later. All of the cells are then updated simultaneously. The transitions depend only on the number of live neighbors.

Conway chose his rules carefully, after a long period of experimentation, so that the rules should be such as to make the behavior of the population both interesting and unpredictable.

Implementation using conventional approach. This case study uses the implementation of Conway's Game of Life using the conventional approach by Edwin Martin (2002). In the conventional approach, the Java application provides the basic graphical user interface with the grid for cell placement. The events for marking a cell as living or dead are provided and event listeners generate the layout of the cells in the grid. The interface also provides templates for generating fixed starting cell patterns for the first generation.

The rules of the game are implemented using a standard decision structure within the application code whenever the even for the next generation of the cells are initiated. The rules are applied one at a time in the specified order within the application code.

Identifying business rules. The rules of Conway's Game of Life are embedded in the application code specifically with the event listener routine which is responsible for producing the next generation of the cells in the current grid layout.

This section of code is where the game's 'business rules' are completely interlinked with the application's event processing, data processing and control flow logic. This section of code will be the source for identifying the business rules used for implementing the requirements of the game.

The rules however are not directly understandable from the application code because of the underlying complexity of the programming language used. This is one of the core maintenance issues with conventional applications.

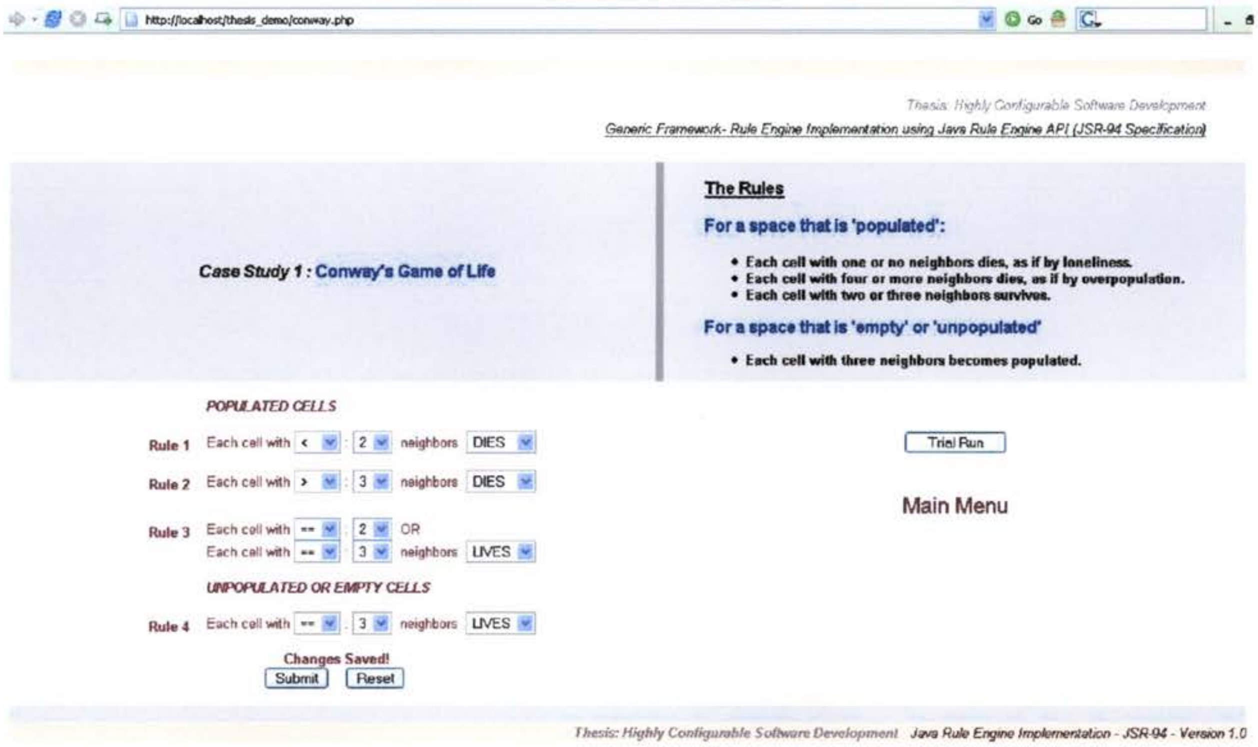
Rules representation. For use with the rules-centric development approach, the rules are separated out and are available in an English-like syntax. These rules can be represented in the SRML language using the simple document structure of the standard. Figure 5.1 shows the representation of all the rules of the game in the SRML format.

```
<rule-set name="conway">
  <rule name="Kill The Overcrowded">
   <conway:condition>
              <conway:cell name="cell"> <conway:
              liveNeighborCountGreaterThan>3</conway:liveNeighbor
       CountGreaterThan></conway:cell>
    </conway:condition>
    <conway:actions>
      <conway:cell name="cell">
        <conway:queueState>dead</conway:queueState>
      </conway:cell> </conway:actions>
  </rule>
  <rule name="Kill The Lonely">
    <conway:condition>
      <conway:cell name="cell">
                     <conway:liveNeighborCountLessThan>2
                            </conway:liveNeighborCountLessThan>
      </conway:cell>
    </conway:condition> <conway:actions>
      <conway:cell name="cell">
        <conway:queueState>dead</conway:queueState>
      </conway:cell>
    </conway:actions>
  </rule>
  <rule name="Give Birth">
    <conway:condition>
      <conway:cell name="cell">
                     <conway:liveNeighborCountEquals>3
                            </conway:liveNeighborCountEquals>
      </conway:cell></conway:condition>
    <conway:actions> conway:cell name="cell">
        <conway:queueState>live</conway:queueState>
      </conway:cell></conway:actions>
  </rule>
</rule-set>
```

Figure 5.1 SRML representation of rules for Conway's Game of Life

Implementation using rules-centric approach. The implementation using the rules-centric approach involves the authoring of the rules represented in the SRML format in the rules repository. This can be done through the rule authoring environment provided by the generic framework.

The rule authoring environment for this game allows for modification of the rules of the game through a web based interface. This environment will allow for rewriting of the rules represented in the rules repository as SRML documents. The web based interface is shown in Figure 5.2. The screen shows the initial configuration of the rules of the game.


http://localhost/thesis_demo/conway.php

Thesis: Highly Configurable Software Development

Generic Framework- Rule Engine Implementation using Java Rule Engine API (JSR-94 Specification)

Case Study 1 : **Conway's Game of Life**

The Rules

For a space that is 'populated':

- **Each cell with one or no neighbors dies, as if by loneliness.**
- **Each cell with four or more neighbors dies, as if by overpopulation.**
- **Each cell with two or three neighbors survives.**

For a space that is 'empty' or 'unpopulated'

- **Each cell with three neighbors becomes populated.**

POPULATED CELLS

Rule 1	Each cell with	<	2	neighbors	DIES
Rule 2	Each cell with	>	3	neighbors	DIES
Rule 3	Each cell with	==	2	OR	
	Each cell with	==	3	neighbors	LIVES

UNPOPULATED OR EMPTY CELLS

Rule 4	Each cell with	==	3	neighbors	LIVES

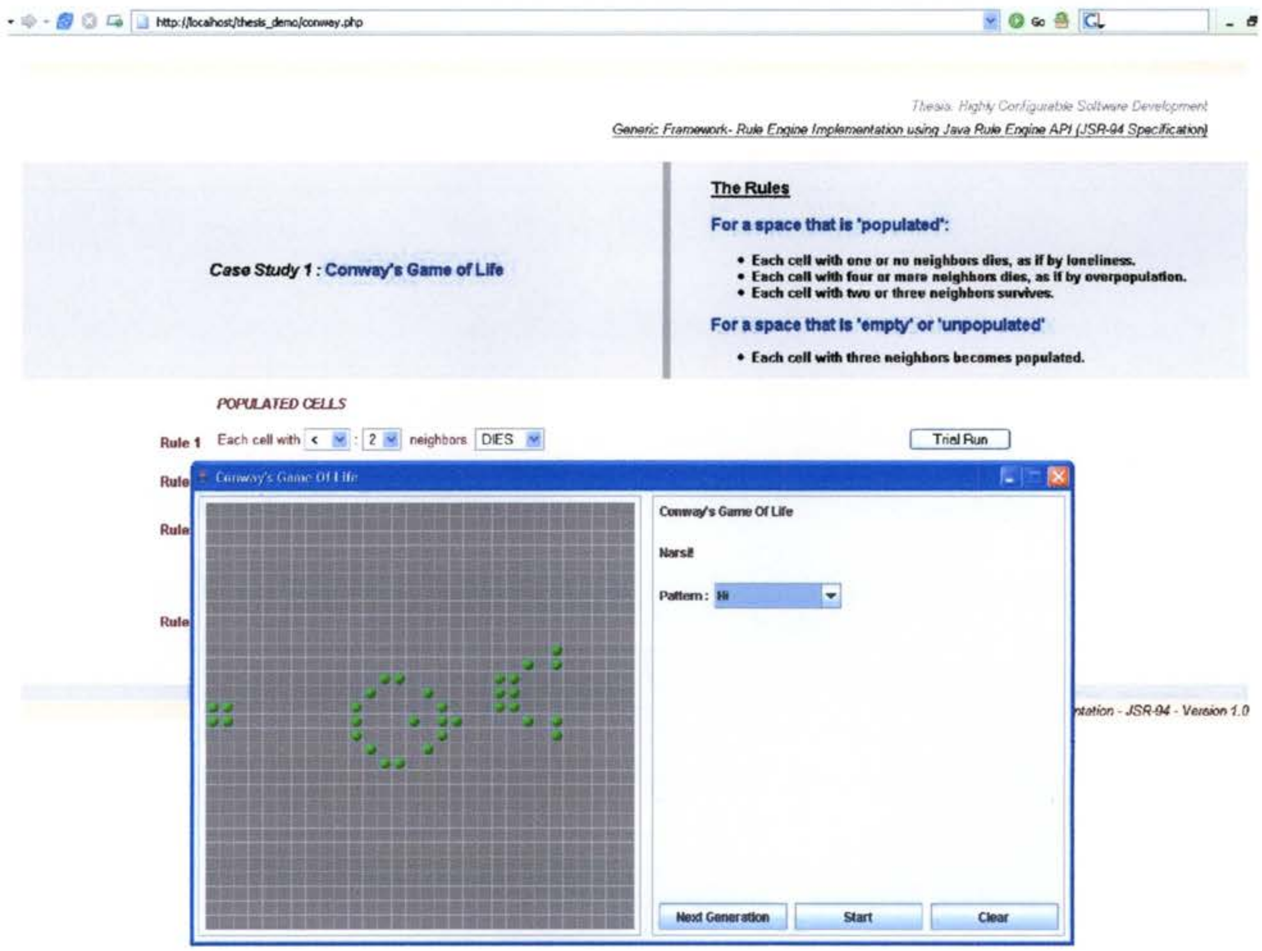
Changes Saved!

Thesis: Highly Configurable Software Development *Java Rule Engine Implementation - JSR-94 - Version 1.0*

Figure 5.2 Initial rule configuration of the Conway Game of Life

The environment employs the Java rule engine API to store the rules in the rules repository accessible by the rule engine for retrieval and execution. To execute the rules in the application's execution context, application code that is free of any business logic incorporation is used as a 'skeleton' which specifies only the control flow and the event processing logic of the application.

The main graphical interface for the game was developed using the Java Abstract Window Toolkit (AWT) API. The interface will provide options to form an initial configuration of the grid. Next and subsequent generations are evolved by a timer based operation that applies the rules of the game to the cell configuration in the grid. Figure 5.3 shows the application screen running with an initial configuration of cells.


http://localhost/thesis_demo/conwey.php

Thesis: Highly Configurable Software Development

Generic Framework- Rule Engine Implementation using Java Rule Engine API (JSR-94 Specification)

Case Study 1 : Conway's Game of Life

The Rules

For a space that is 'populated':

- Each cell with one or no neighbors dies, as if by loneliness.
- Each cell with four or more neighbors dies, as if by overpopulation.
- Each cell with two or three neighbors survives.

For a space that is 'empty' or 'unpopulated'

- Each cell with three neighbors becomes populated.

POPULATED CELLS

Rule 1 Each cell with < : 2 neighbors DIES

Conway's Game Of Life

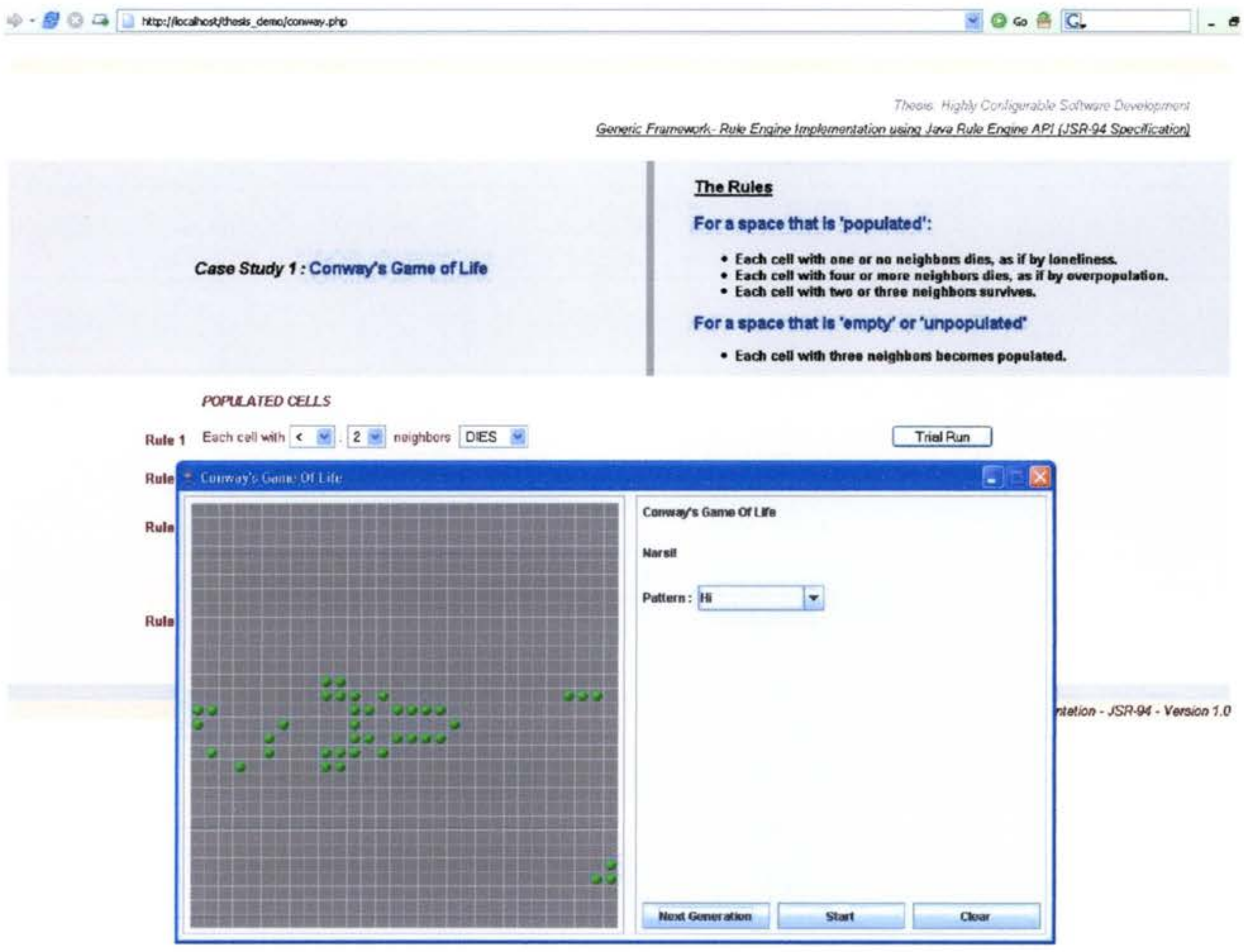
Narsil

Pattern: Hi

ntation - JSR-94 - Version 1.0

Figure 5.3 Initial cell configuration of the Conway grid

Figure 5.4 shows the cell configuration after 50 generations. This configuration was evolved from the initial configuration of cells shown in Figure 5.3. The rules of the game applied after every generation is based on the current cell configuration of the grid.



http://localhost/thesis_demo/conway.php

Thesis: Highly Configurable Software Development
Generic Framework- Rule Engine Implementation using Java Rule Engine API (JSR-94 Specification)

Case Study 1 : Conway's Game of Life

The Rules

For a space that is 'populated':

- **Each cell with one or no neighbors dies, as if by loneliness.**
- **Each cell with four or more neighbors dies, as if by overpopulation.**
- **Each cell with two or three neighbors survives.**

For a space that is 'empty' or 'unpopulated'

- **Each cell with three neighbors becomes populated.**

POPULATED CELLS

Rule 1 Each cell with < . 2 neighbors DIES

Conway's Game Of Life

Narsil

Pattern : Hi

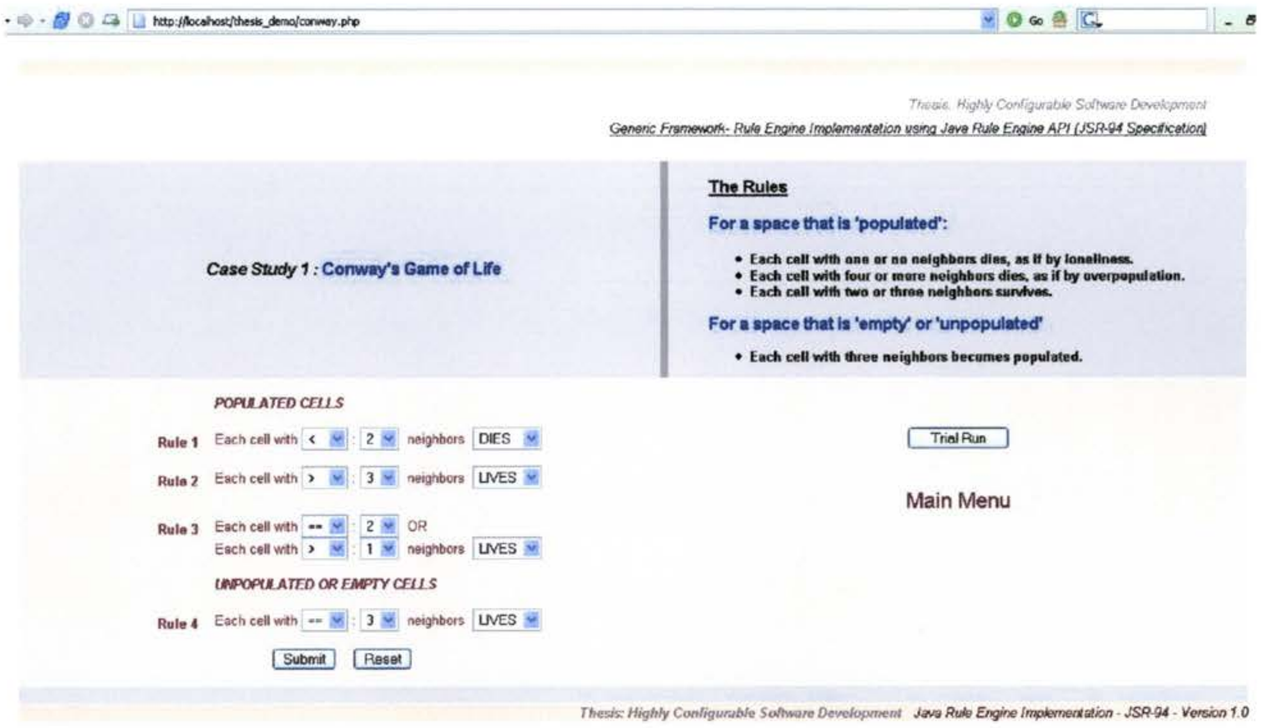
Figure 5.4 Cell configuration of the Conway grid after 50 generations

The Java class mechanism allows for abstracting the rule engine API interactions. The SRML rule document serves as the input for a class implementation of a 'Rule Factory' which provides the rules at run-time for invocation at the object's execution context. Figure 5.5 shows the partial implementation of this class. The full implementation is available in Appendix A.

```
public class RuleBaseFactory {
            //Instantiate a new rule set object
             private static RuleBaseFactory ourInstance      = new RuleBaseFactory( );
             private static final String    DEFAULT_RULE_FILE = "conway.java.srml";
             private RuleBase ruleBase;
            //Get the Rule Execution Context
             public static RuleBaseFactory getInstance()    {
                return ourInstance;    }
             private RuleBaseFactory() {
                try {
                            //Get the SRML file name from the rules repository
                            String       conwaySRMLFile       =        System.getProperty(
                  "conway.java.srml " );
                            if ( conwaySRMLFile == null ) {
                                      System.err.println(  "Rule   file   system   property   not
          specified. using
                            default: " +   DEFAULT_DRL_FILE );
                                      conwaySRMLFile = DEFAULT_RULE_FILE;
                  }
                            System.out.println( "loading drl file: " + conwaySRMLFile );
                            URL resource = CellGrid.class.getResource( conwaySRMLFile );
                            //Create the working memory for the execution context
                            ruleBase = RuleBaseLoader.loadFromUrl( resource );
                  }
                catch ( Exception e ) {
                   throw new RuntimeException( "Unable To Initialise RuleBaseFactory:\n"
          +
                            e.getMessage() );       }  }
             public static RuleBase getRuleBase() {
                return ourInstance.ruleBase;   }
}
```

Figure 5.5 Class mechanism for Rule Factory

As the configuration of the rules are changed using the rule authoring interface, the changes are reflected in the rules repository immediately. The next session of the application will incorporate the new rules from the repository. Figure 5.6 shows a rule configuration different from the original rules of the game.


http://localhost/thesis_demo/conway.php

Thesis: Highly Configurable Software Development

Generic Framework- Rule Engine Implementation using Java Rule Engine API (JSR-94 Specification)

Case Study 1 : Conway's Game of Life

The Rules

For a space that is 'populated':

- Each cell with one or no neighbors dies, as if by loneliness.
- Each cell with four or more neighbors dies, as if by overpopulation.
- Each cell with two or three neighbors survives.

For a space that is 'empty' or 'unpopulated'

- Each cell with three neighbors becomes populated.

POPULATED CELLS

Rule 1 Each cell with < : 2 neighbors DIES

Rule 2 Each cell with > : 3 neighbors LIVES

Rule 3 Each cell with == : 2 OR
Each cell with > : 1 neighbors LIVES

UNPOPULATED OR EMPTY CELLS

Rule 4 Each cell with == : 3 neighbors LIVES

Thesis: Highly Configurable Software Development Java Rule Engine Implementation - JSR-94 - Version 1.0

Figure 5.6 A configuration different from the rules of the game

Since this configuration alters the original rules of the game, the behavior of the cells during each generation is changed by the new rule configuration. The rule engine retrieves the new rules for the repository and asserts the objects from the working memory against conditions in each rule set.

The new rule configuration specifies that a cell will live in the next generation if it has more than 3 neighbors. This change will result in the increase of living cells at each generation.

Figure 5.7. shows the cell configuration after 50 generations using the new rules.

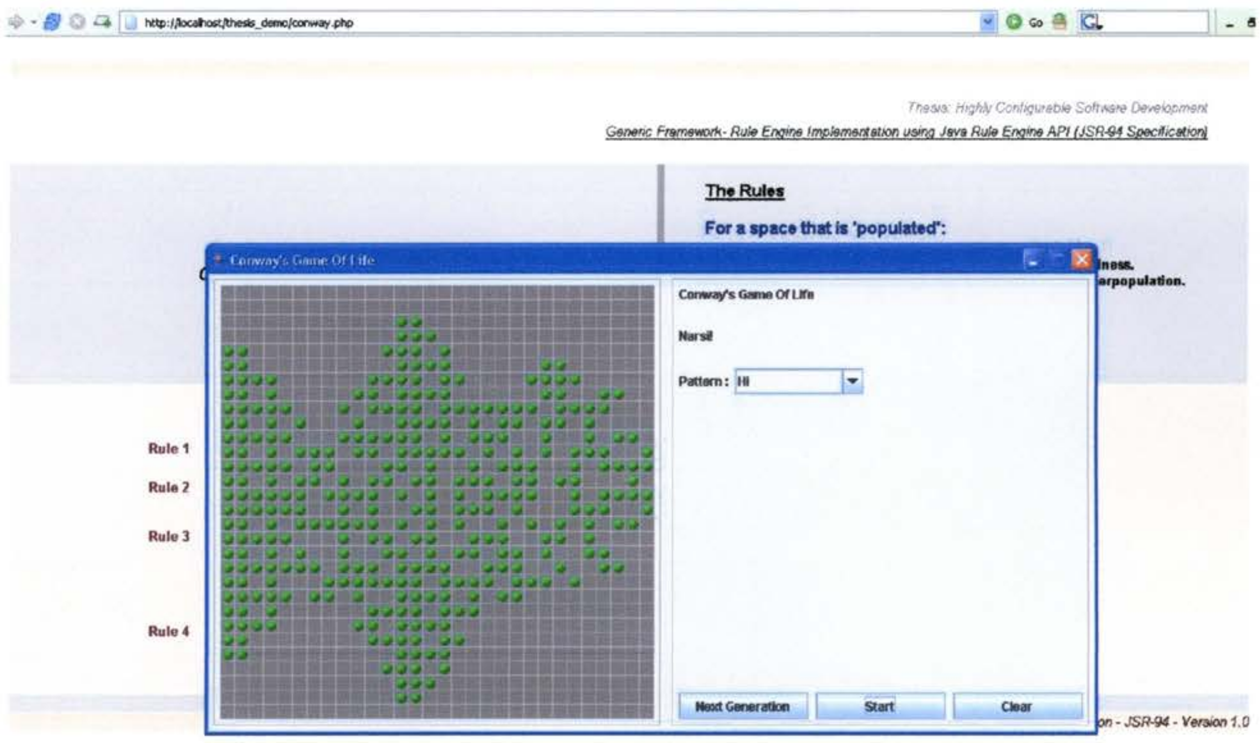

http://localhost/thesis_demo/conway.php
Thesis: Highly Configurable Software Development
Generic Framework- Rule Engine Implementation using Java Rule Engine API (JSR-94 Specification)
The Rules
For a space that is 'populated':
Rule 1
Rule 2
Rule 3
Rule 4
Conway's Game Of Life
Conway's Game Of Life
Narsil
Pattern : Hi
Next Generation
Start
Clear
on - JSR-94 - Version 1.0

Figure 5.7 Cell configuration after 50 generations based on new rules

This case study demonstrated the ability of the generic framework to adapt for the changes in rules dynamically without the need for recompilation of the application code. The next case study will demonstrate the dynamic rule change behavior in more a complex business scenario.

5.1.2 Case Study 2: Java Pet store Application

The Java Pet Store Application is a popular J2EE example application in the J2EE Blue-prints series, created by Sun Microsystems. It models an e-commerce application where customers can purchase pets online using a Web browser. The purpose of this application is to demonstrate the capabilities of the J2EE platform and is written for learning purposes.

Java Pet store Application. This application was chosen as a case study candidate for this thesis because it represents a more realistic example scenario for demonstrating how business applications can use the proposed generic framework to have a clear separation of business logic from application control and data processing code.

The rules of the order process. The Pet Store example implements the following rules.

Free Fish Food Sample -	Add free fish food sample based on the shopping cart contents (if it has at least one fish).
Suggest Fish Tank -	Suggest the buyer to buy a fish tank if there are 5 or more Gold fish in the shopping cart.
Apply Discounts -	Apply discounts based on the gross total of the shopping cart reaching a certain eligible amount.

Free Fish Food Sample. If the user has at least one fish and has not bought any fish food the application checks if a free sample has already been given and if not, adds a free fish food sample to the cart.

Suggest Fish Tank. If the user has bought at least 5 Gold Fish and does not already have a Fish Tank, ask the user if they would like a fish tank. If they do, add one to the cart.

Apply Discounts. The rules currently apply two discounts 5% and 10%. Both check if the Gross Cost is between certain levels and that the discount has not already been applied. If the user currently qualifies for a 5% discount and the system prompts them for a Fish Tank and if it is added, the discount rules are checked again and if appropriate, the 10% discount rule is applied.

Implementation using conventional approach. The example uses the Java Swing API to provide a GUI for the order process and the shopping cart. In the conventional approach, the rules are embedded in the application code and are applied to the shopping cart whenever the application generates the event for the order check out.

Implementation using rules-centric approach. To get the integration between the GUI and the Rules Engine a callback mechanism was used. When the user clicks the 'Checkout' button it calls the callback function, passing it a reference to the current JFrame and also the list of chosen items. These items are added to a Shopping Cart object which is then asserted into the Working Memory.

Identifying business rules . The Pet store application has complex interrelated rules that are applied to the Order process. The rules can be identified from the section of code that applies the rules to the shopping cart. However, separating the rules of the application in this manner is tedious and error prone, which is a problem faced when legacy applications following conventional development approach are considered for migration into a rules-centric development approach. For this case study, since we have the rules in a English-like syntax already, we can represent them in the SRML format (Figure 5.8).

Rules representation

```
<rule-set name="PetStore Rules">
        <!-- Initiate the shopping cart -->
        <rule name="Explode Cart">
                <parameter identifier="cart">
                        <class>petstore.ShoppingCart</class>
                </parameter>
                <condition>
                        cart.getState("Exploded") == false
                </condition>
                <consequence>
                        cart.setState( "Exploded",true)
                </consequence>
        </rule>
        <!-- Free Fish Food sample - when buying a Gold Fish, If
                haven't already bought Fish Food and Don't already
                have a Fish Food Sample        -->
        <rule name="Free Fish Food Sample">
                <parameter identifier="cart">
                        <class>petstore.ShoppingCart</class>
                </parameter>
                <parameter identifier="item">
                        <class>CartItem</class>
                </parameter>
                <condition>
                        cart.getItems( "Fish Food Sample" ).size() == 0
                </condition>
                <condition>
                        cart.getItems( "Fish Food" ).size() == 0
```

```
          </condition>
          <condition>
                    item.getName().equals( "Gold Fish")</condition>
          <consequence>
                    cart.addItem( new CartItem(
                    "Fish Food Sample", 0.00 ) )     </consequence>
</rule>
<!-- Suggest a tank if we have bought more than 5 gold
          fish and dont already have one          -->
<rule name="Suggest Tank">
          <parameter identifier="cart">
                    <class>petstore.ShoppingCart</class>
          </parameter>
          <condition>
                    cart.getState( "Suggested Fish Tank" ) == false
          </condition>
          <condition>
                    cart.getItems( "Gold Fish" ).size() &gt;= 5
          </condition>
          <condition>
                    cart.getItems( "Fish Tank" ).size() == 0
          </condition>
          <consequence>
                    cart.setState( "Suggested Fish Tank", true )
          </consequence>
</rule>
<!-- Give 5% discount if gross cost is more than 20.00 -->
<rule name="Apply 5% Discount">
          <parameter identifier="cart">
                    <class>petstore.ShoppingCart</class>
          </parameter>
          <condition>
                    cart.getGrossCost() &gt;= 10.00</condition>
          <condition>cart.getGrossCost() &lt; 19.9</condition>
          <condition>cart.getDiscount() &lt; 0.05</condition>
          <consequence>
                    cart.setDiscount( 0.05 )
```

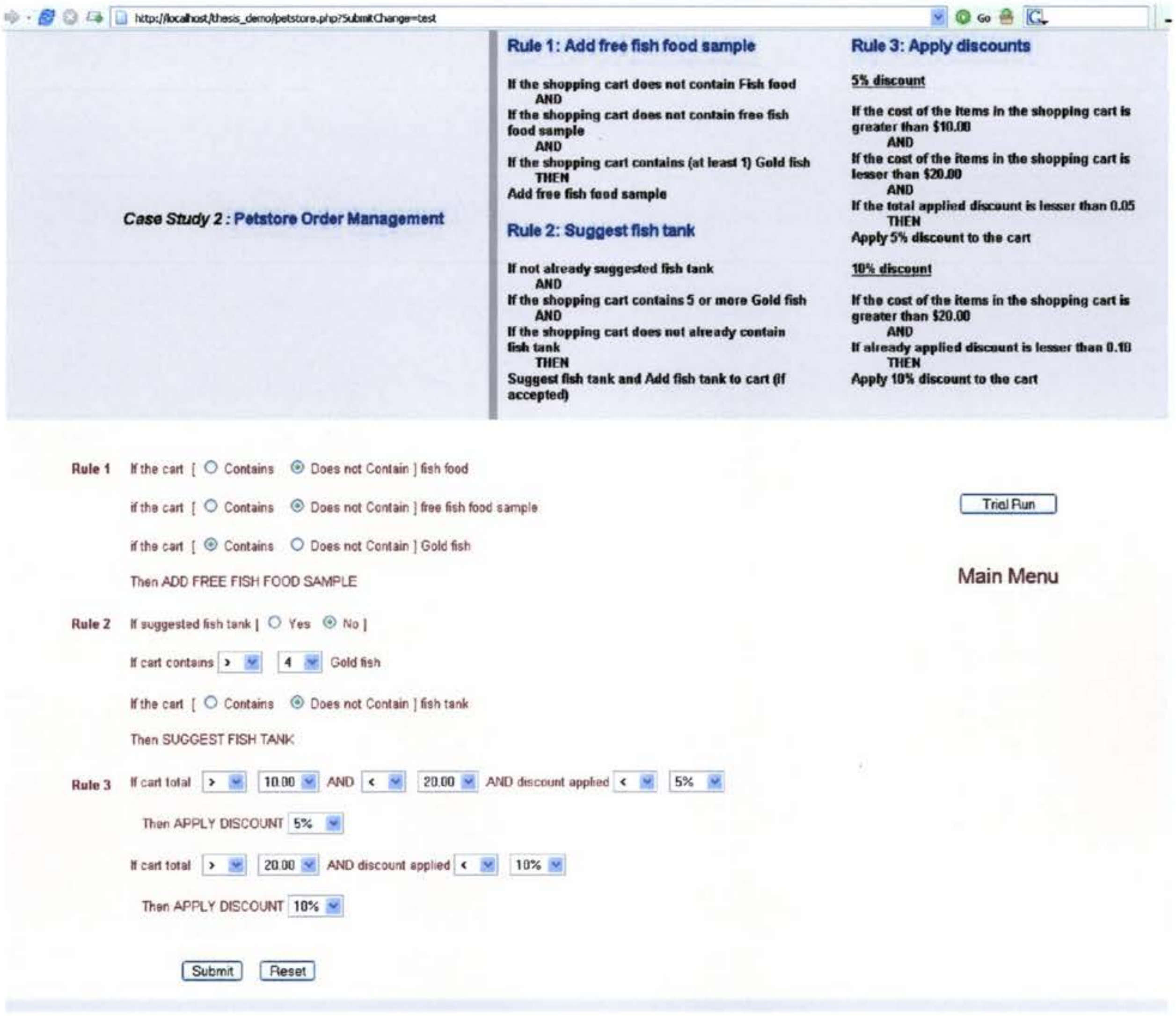
```
                </consequence>
        </rule>
        <!-- Give 10% discount if gross cost is more than 20.00 -->
        <rule name="Apply 10% Discount">
                <parameter identifier="cart">
                        <class>petstore.ShoppingCart</class>
                </parameter>
                <condition> cart.getGrossCost() &gt;= 20.00
                        </condition>
                <condition>cart.getDiscount() &lt; 0.10
                        </condition>
                <consequence>cart.setDiscount( 0.10 )
                        </consequence>
        </rule>
</rule-set>
```

Figure 5.8 SRML representation of rules for Java Petstore Application

The rule authoring interface for this case study is shown in Figure 5.9. The initial rule configuration is set to be the same as the business rules used in the conventional approach.

The application implements the rules defined through the rules authoring environment. Figure 5.10 shows the running application with the shopping cart contents based on user purchase.

The rules will be applied to the shopping cart contents when the check out process is initiated by clicking the check out button. The run-time environment provided by the rule engine will execute the rules retrieved from the rules repository.



Case Study 2 : **Petstore Order Management**

Rule 1: Add free fish food sample

If the shopping cart does not contain Fish food
AND
If the shopping cart does not contain free fish food sample
AND
If the shopping cart contains (at least 1) Gold fish
THEN
Add free fish food sample

Rule 2: Suggest fish tank

If not already suggested fish tank
AND
If the shopping cart contains 5 or more Gold fish
AND
If the shopping cart does not already contain fish tank
THEN
Suggest fish tank and Add fish tank to cart (if accepted)

Rule 3: Apply discounts

5% discount

If the cost of the items in the shopping cart is greater than $10.00
AND
If the cost of the items in the shopping cart is lesser than $20.00
AND
If the total applied discount is lesser than 0.05
THEN
Apply 5% discount to the cart

10% discount

If the cost of the items in the shopping cart is greater than $20.00
AND
If already applied discount is lesser than 0.10
THEN
Apply 10% discount to the cart

Rule 1 If the cart [Contains [✓ Does not Contain]] fish food

if the cart [Contains [✓ Does not Contain]] free fish food sample

if the cart [[✓ Contains] Does not Contain] Gold fish

Then ADD FREE FISH FOOD SAMPLE

Rule 2 If suggested fish tank | Yes [✓ No] |

If cart contains > 4 Gold fish

If the cart [Contains [✓ Does not Contain]] fish tank

Then SUGGEST FISH TANK

Rule 3 If cart total > 10.00 AND < 20.00 AND discount applied < 5%

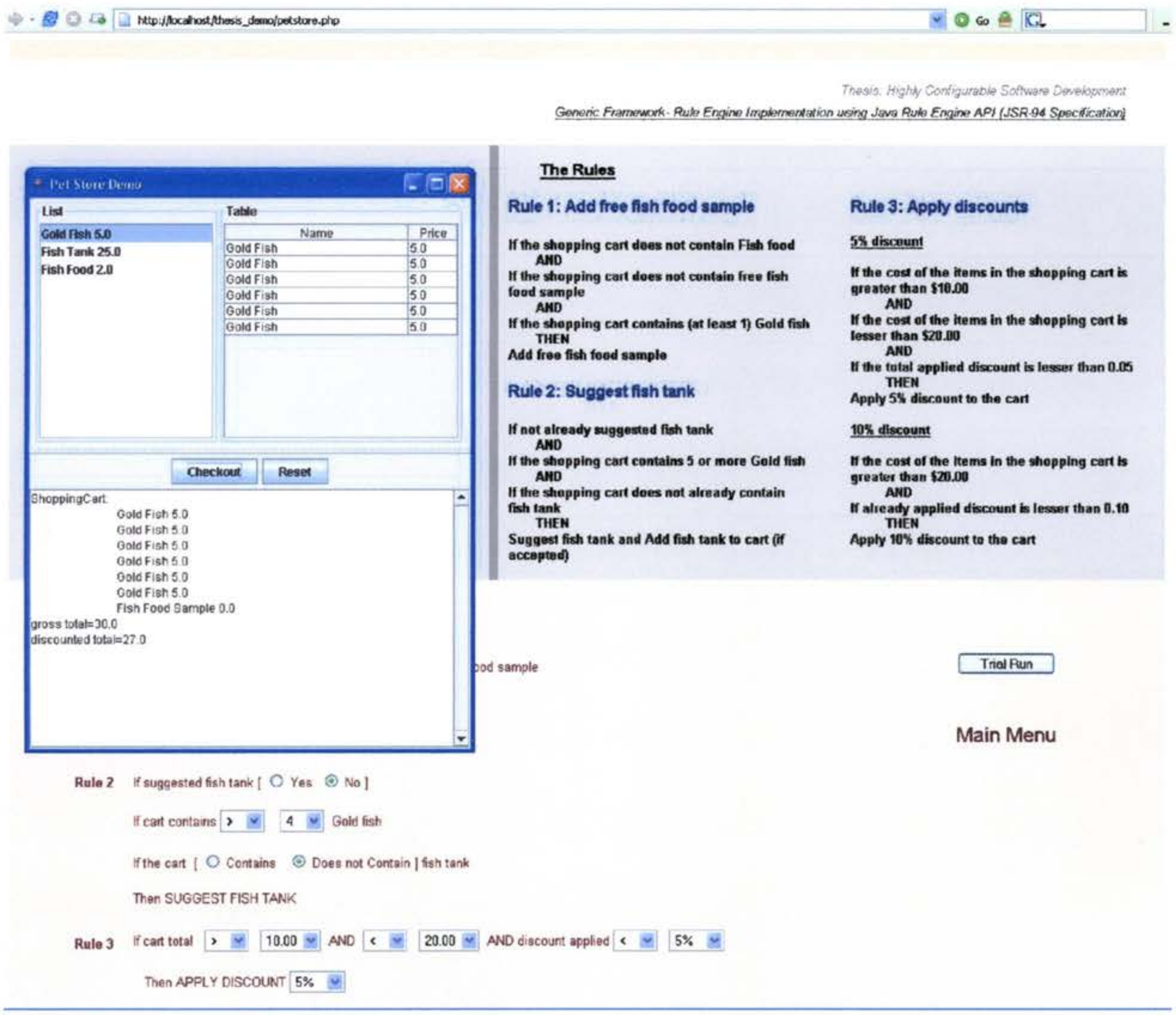
Then APPLY DISCOUNT 5%

If cart total > 20.00 AND discount applied < 10%

Then APPLY DISCOUNT 10%

Figure 5.9 Rules authoring environment for the petstore case study

According to the initial rule configuration, the application adds the free fish food sample to the shopping cart based on the number of Gold fish purchased. Figure 5.11 shows a rule configuration that is different from the initial configuration. This rule set will be saved through the rules authoring environment and the application behavior will be studied based on the new rules.



http://localhost/thesis_demo/petstore.php

Thesis: Highly Configurable Software Development

Generic Framework - Rule Engine Implementation using Java Rule Engine API (JSR-94 Specification)

Pet Store Demo

List

Gold Fish 5.0
Fish Tank 25.0
Fish Food 2.0

Table

Name	Price
Gold Fish	5.0
Gold Fish	5.0
Gold Fish	5.0
Gold Fish	5.0
Gold Fish	5.0
Gold Fish	5.0

```
ShoppingCart:
            Gold Fish 5.0
            Gold Fish 5.0
            Gold Fish 5.0
            Gold Fish 5.0
            Gold Fish 5.0
            Gold Fish 5.0
            Fish Food Sample 0.0
gross total=30.0
discounted total=27.0
```

The Rules

Rule 1: Add free fish food sample

If the shopping cart does not contain Fish food
AND
If the shopping cart does not contain free fish food sample
AND
If the shopping cart contains (at least 1) Gold fish
THEN
Add free fish food sample

Rule 2: Suggest fish tank

If not already suggested fish tank
AND
If the shopping cart contains 5 or more Gold fish
AND
If the shopping cart does not already contain fish tank
THEN
Suggest fish tank and Add fish tank to cart (if accepted)

Rule 3: Apply discounts

5% discount

If the cost of the items in the shopping cart is greater than $10.00
AND
If the cost of the items in the shopping cart is lesser than $20.00
AND
If the total applied discount is lesser than 0.05
THEN
Apply 5% discount to the cart

10% discount

If the cost of the items in the shopping cart is greater than $20.00
AND
If already applied discount is lesser than 0.10
THEN
Apply 10% discount to the cart

ood sample

Rule 2 If suggested fish tank [Yes [✓ No]]

If cart contains > 4 Gold fish

If the cart [Contains [✓ Does not Contain]] fish tank

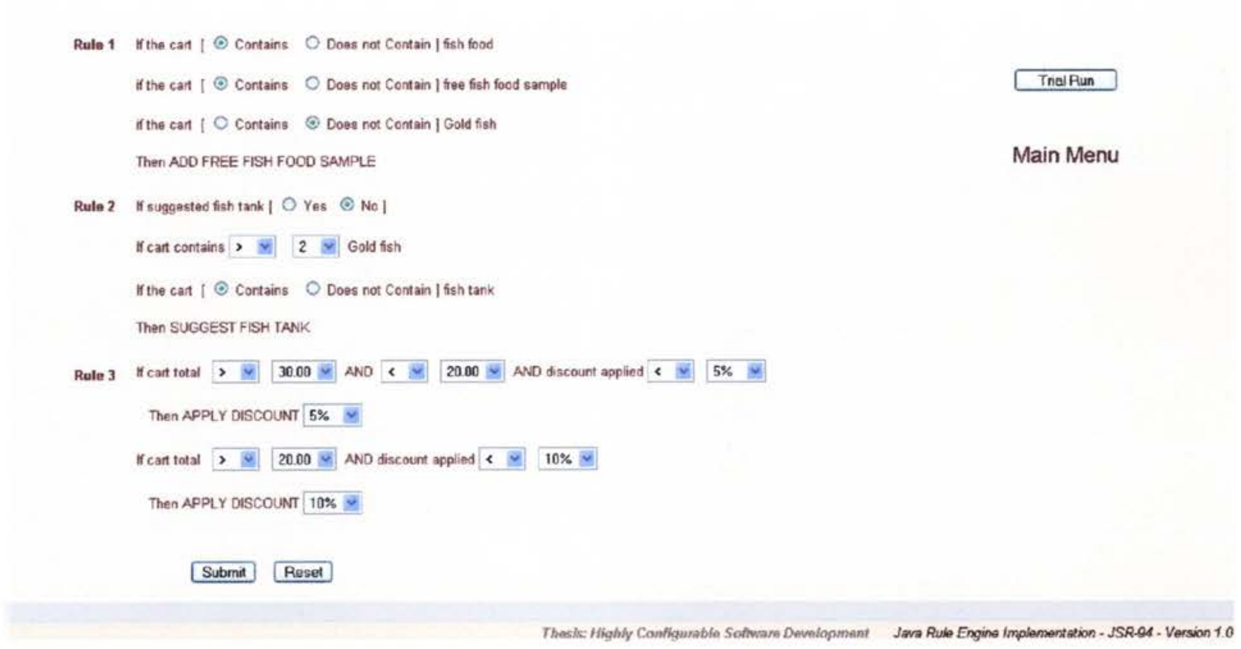
Then SUGGEST FISH TANK

Rule 3 If cart total > 10.00 AND < 20.00 AND discount applied < 5%

Then APPLY DISCOUNT 5%

Figure 5.10 Petstore case study - initial rule configuration

The new rules will be retreived by the rule engine from the rules repository and the run-time environment provided by the rule engine will execute the rules retrieved from the rules repository.


Rule 1 If the cart [[✓ Contains] Does not Contain] fish food

if the cart [[✓ Contains] Does not Contain] free fish food sample

if the cart [Contains [✓ Does not Contain]] Gold fish

Then ADD FREE FISH FOOD SAMPLE

Rule 2 If suggested fish tank [Yes [✓ No]]

If cart contains > 2 Gold fish

If the cart [[✓ Contains] Does not Contain] fish tank

Then SUGGEST FISH TANK

Rule 3 If cart total > 30.00 AND < 20.00 AND discount applied < 5%

Then APPLY DISCOUNT 5%

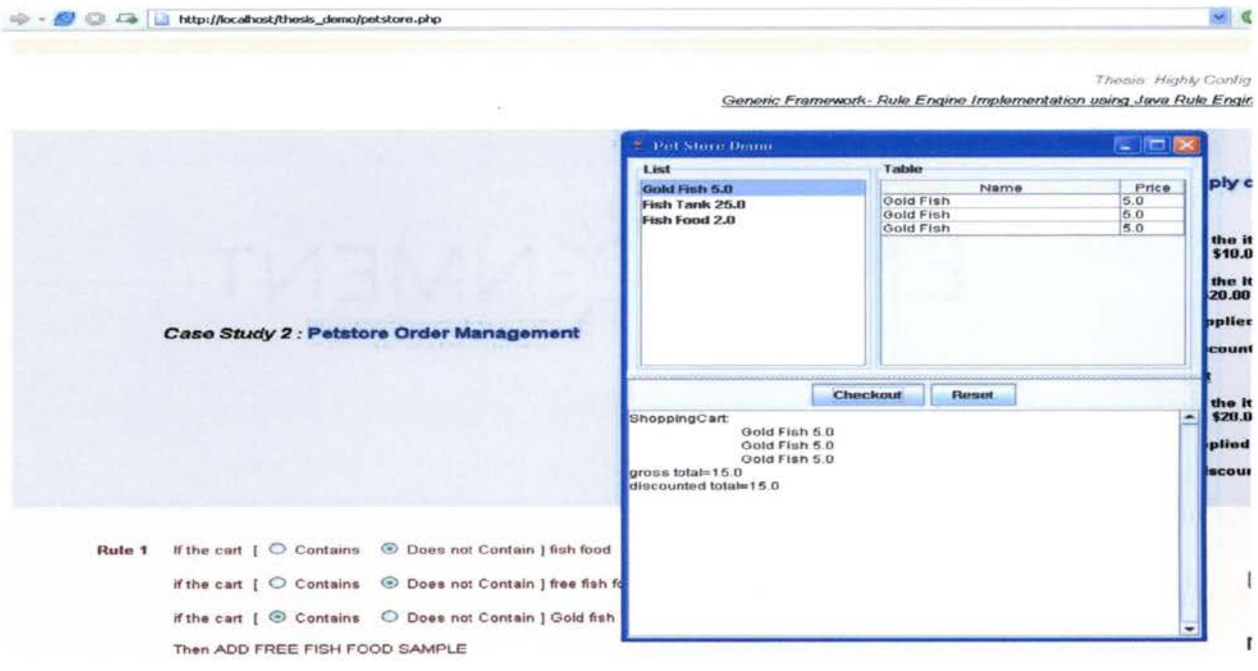
If cart total > 20.00 AND discount applied < 10%

Then APPLY DISCOUNT 10%

Trial Run
Main Menu
Submit
Reset
Thesis: Highly Configurable Software Development
Java Rule Engine Implementation - JSR-94 - Version 1.0

Figure 5.11 Petstore case study – A new rule configuration

Figure 5.12 shows how the changes to the order process are reflected in the next application restart.


http://localhost/thesis_demo/petstore.php
Thesis: Highly Config
Generic Framework- Rule Engine Implementation using Java Rule Engir
Case Study 2 : Petstore Order Management
Pet Store Demo
List
Table
Checkout
Reset

List:

Gold Fish 5.0
Fish Tank 25.0
Fish Food 2.0

Name	Price
Gold Fish	5.0
Gold Fish	5.0
Gold Fish	5.0

```
ShoppingCart:
                    Gold Fish 5.0
                    Gold Fish 5.0
                    Gold Fish 5.0
gross total=15.0
discounted total=15.0
```

Rule 1 If the cart [Contains [✓ Does not Contain]] fish food

if the cart [Contains [✓ Does not Contain]] free fish f

if the cart [[✓ Contains] Does not Contain] Gold fish

Then ADD FREE FISH FOOD SAMPLE

Figure 5.12 Petstore case study – Application behavior with new rule configuration

This Chapter showed how business rules are separated and represented in the SRML document format and the implementation based on the rules-centric approach using the proposed generic framework is performed. The next chapter provides the conclusions on the research and a few directions for future research.

CHAPTER 6

6.1 Conclusions

This study has shown how the rules-centric approach for software development can be highly beneficial and why a high level of software configuration support is required. It is essential that technology be developed for effectively navigating change in requirements to software over time. This research addresses this concern by proposing a generic framework that follows a rules-centric approach and is highly interoperable because of the adopted standards and methodology the design was based on.

The proposed framework allows applications constructed to cope with many different types of changes in the requirements. The features of the proposed generic framework that facilitate a high level of software configurability have been explained in detail.

Static business rules lead to inflexible implementations within the application, which can lead to problems when business rules change. The proposed generic approach ensures that business logic is implemented in a standard way, narrowing the communication gap during the requirements, analysis and design phases of projects.

Interaction and development stages between these phases can be more flexible and there is little need to freeze requirements. This means that project teams can take an iterative approach to development. The result is a shorter time to code and a shorter time to deploy.

Once applications are deployed, changes to business rules are easier to implement because they reside in a centralized location. Moreover, because business rules are separated from technical implementation, it becomes much easier to accommodate technology innovations such as service oriented architectures.

The benefits of the rules-centric approach are realized fully when business rules are maintained and monitored apart from the rest of the system by centralizing business rules in a repository and providing access to them dynamically.

Dynamic business rules let end users deal with changes in business rules at deployment time without recompiling and redeploying the application, which is one of the primary objectives for a rules-centric development framework. This allows both developers and application administrators to update and modify the business logic in an application quickly, without regenerating, recompiling and redeploying the source code.

To realize this vision, an organization must build its systems with tools that are powerful enough to capture their business rules, publish them to a centralized repository, automate the rules in a consistent manner, easily maintain the rules to accommodate business change, and provide business people with the ability to monitor the execution of those rules.

This requires a platform with powerful new functionality - the kind of functionality that is proposed in the generic framework and more.

Finally the capabilities of the proposed framework in support of construction of highly configurable software development was demonstrated using two contrasting case studies from a simple game example to a complex real-life business application scenario.

6.2 Directions for Future Research

Software change is more and more influenced by middleware and Commercial-Off-The-Shelf (COTS) components. Since software functionality is routinely distributed across heterogeneous software and hardware platforms, complex int-eroperability issues necessarily govern any analysis of software changes. For example, when a custom software component is replaced by a COTS comp-onent, the internal complexity of the component is replaced by a focus on interface and interoperability complexities. Web services in today's distributed applications provide an excellent research platform to explore these shifts in complexity.

Though rules-centric software development platforms are beginning to address these expanding problems with features and power to develop highly configurable and adaptive software, there is still a long way to go in realizing the dream of true configurability in various application domains including the e-commerce and the web applications scenarios.

New standards and features such as in the Java platform offer further possibilities for configuration support. JDK 1.1 introduced classes to support reflection and introspection, making it possible to query the capabilities of a class at run-time and determine, for example, what methods it provides, what parameters they take, and what exceptions they raise.

Using this reflection API an application can use code it had no prior knowledge of simply by invoking the class and its methods through their names. Implementations need not conform to interfaces to be able to use them. In addition, JDK 1.1 introduced the concept of object serialization: the complete state of an object, including any objects it refers to, can be written to an output stream, and this stream can be used to recreate that object at a later time.

Therefore, once an application has been configured, the configuration could be 'frozen' using this mechanism, and automatically recreated later, without a need for the original object name. Innovations like these have led to the development of standard specifications that can serve as a reference point for implementing Java based rule engines, such as the JSR-94 specification which was used in this research.

Having all business rules in one place provides a holistic view of a business process and further leverages the power of rule management. A true business logic management repository should provide a centralized storage of a full range of business logic regardless of the execution environment.

The repository should have the ability to translate and distribute rules to other environments for execution and avoid the maintenance costs due to duplication of business rules in parallel locations, such as the same data validation rule used in web, telephone and email interfaces.

By extending the analysis and research to this growing area, we can enable software engineers, architects, and project managers to make better decisions about software changes, preserve the software quality, and increase the life of systems while lowering the total costs over their operational life.

APPENDIX A

DTD Specification for an SRML Document

```
<!-- SRML (Simple Rule Markup Language) DTD -->
<?xml version="1.0" encoding="ISO-8859-1"?>
<!ELEMENT ruleset (rule*)><!ATTLIST ruleset
  name NMTOKEN #IMPLIED>
<!ELEMENT rule ( priority?, conditionPart, actionPart ) >
<!ATTLIST rule name NMTOKEN #REQUIRED>
<!ELEMENT priority (%expression;)>
<!ELEMENT conditionPart (%condition;)+>
<!ELEMENT actionPart (%action;)* >
<!ENTITY % condition "(simpleCondition | notCondition)">
<!ENTITY % action "(assignment | bind | assert | assertobj
             | modify | retract)">
<!ELEMENT simpleCondition (%expression;)*>
<!ATTLIST simpleCondition
  className CDATA #REQUIRED
  objectVariable NMTOKEN #IMPLIED>
<!ELEMENT notCondition (%expression;)*>
<!ATTLIST notCondition
  className CDATA #REQUIRED>
<!ELEMENT assert (assignment | bind)* >
<!ATTLIST assert  className CDATA #REQUIRED>
```

```
<!ELEMENT assertobj (%expression;)>
<!ELEMENT retract (variable)>
<!ELEMENT modify (variable, (assignment | bind)+)>
<!ENTITY % expression
  "(%assignable; | constant | unaryExp | binaryExp | naryExp)">
<!ELEMENT unaryExp (%expression;)>
<!ATTLIST unaryExp operator (plus | minus | not) #REQUIRED>
<!ELEMENT binaryExp (%expression;,%expression;)>
<!ATTLIST binaryExp
  operator (eq | neq | lt | lte | gt | gte) #REQUIRED>
<!ELEMENT naryExp (%expression;)+>
<!ATTLIST naryExp operator (add | subtract | multiply | divide |
          remainder | and | or) #REQUIRED>
<!ELEMENT assignment (%assignable;,%expression;)>
<!ELEMENT bind (%expression;) >
<!ATTLIST bind name NMTOKEN #REQUIRED>
<!ELEMENT constant EMPTY>
<!ATTLIST constant
  type (string | boolean | byte | short | char | long
       | int | float | double | null) #REQUIRED
  value CDATA #REQUIRED>
<!ENTITY % assignable "(variable | field)">
<!ELEMENT variable EMPTY>
<!ATTLIST variable
  name NMTOKEN #REQUIRED>
<!ELEMENT field (%expression;)?>
<!ATTLIST field  name NMTOKEN #REQUIRED>
```

APPENDIX B

```
//*****************************************************************************
*******
//Case Study 1 - Conway's Game of Life
//*****************************************************************************
*******


//*******
//Class- ConwayPattern
//*******

package org.thesis.examples.conway.patterns;

public class Border   implements   ConwayPattern
{

    private boolean[][] grid = {{true, true, true, true, true, true, true,
true, true, true, true, true, true, true, true, true, true, true, true, true,
true, true, true, true, true, true, true, true, true, true, true, true, true,
true, true, true, true, true, true, true, true}};

    public boolean[][] getPattern()
    {
        return grid;
    }

    /**
     * @return the name of this pattern
     */
    public String getPatternName()
    {
        return "Border";
    }

    public String toString()
    {
        return getPatternName( );
    }
}


//*******
//Class- Cell
//*******

package org.thesis.examples.conway;

import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;


public class Cell
{

    private Set neighbors         = new HashSet( );

    private CellState state       = CellState.DEAD;

    private CellState queuedState = null;

    /**
     * @return the number of neighbors that this cell has
     * @see #getNumberOfLiveNeighbors()
     */
    public int getNumberOfNeighboringCells()
    {
        return neighbors.size( );
    }

    /**
     * @return the number of live neighbors that this cell has
     * @see #getNumberOfNeighboringCells()
     */
    public int getNumberOfLiveNeighbors()
    {
        int numberOfLiveNeighbors = 0;
        Iterator it = neighbors.iterator( );
        Cell cell = null;
        while ( it.hasNext( ) )
        {
            cell = (Cell) it.next( );
            if (cell.getCellState( ) == CellState.LIVE)
            {
                numberOfLiveNeighbors++;
            }
        }
        return numberOfLiveNeighbors;
    }

    /**
     * ads a new neighbor to this neighbor
     *
     * @param neighbor
     *            new neighbor
     */
    public void addNeighbor(Cell neighbor)
    {
        neighbors.add( neighbor );
        neighbor.neighbors.add( this );
    }

    /**
     * tell this cell to queue its next live state. this is the state that this
     * cell will be in after the cell is transitioned (after the next
     * iteration). This transition state is necessary because of the 2 phase
     * process involved in evolution.
     *
     * @param nextLiveState
     *            this cell's next live state
     * @see CellState
     * @see #getCellState()
     * @see #transitionState()
     */
    public void queueNextCellState(CellState nextLiveState)
    {
        if ( nextLiveState != state )
        {
            queuedState = nextLiveState;
        }
    }

    /**
     * Transitions this cell to its next state of evolution
     *
     * @return <code>true</code> if the state changed, otherwise false
     * @see #queueNextCellState(CellState)
     */
    public boolean transitionState()
    {
        boolean stateChanged = false;
        if ( queuedState != null )
        {
            state = queuedState;
            queuedState = null;
            stateChanged = true;
        }
        return stateChanged;
    }

    /**
     * @return this cell's current life state
     * @see #queueNextCellState(org.thesis.examples.conway.CellState)
     * @see CellState
     */
    public CellState getCellState()
    {
        return state;
    }

    /**
     * Sets this cells state
     *
     * @param newState
     *            new state for this cell
     * @see CellState
     */
    public void setCellState(CellState newState)
    {
        state = newState;
    }
}


//*******
//Class- CellGrid
//*******

package org.thesis.examples.conway;

import org.thesis.RuleBase;
import org.thesis.WorkingMemory;
import org.thesis.examples.conway.patterns.ConwayPattern;
import org.thesis.examples.conway.rules.RuleBaseFactory;
```

```
public class CellGrid
{

    private final Cell[][] cells;

    /**
     * Constructs a CellGrid
     *
     * @param rows
     *            number of rows in the grid
     * @param columns
     *            number of columns in the grid
     */
    public CellGrid(int rows,
                    int columns)
    {
        cells = new Cell[rows][columns];

        // populate the array of Cells and hook each
        // cell up with its neighbors...
        for ( int row = 0; row < rows; row++ )
        {
            for ( int column = 0; column < columns; column++ )
            {
                Cell newCell = new Cell( );
                cells[row][column] = newCell;
                if ( row > 0 )
                {
                    // neighbor to the north
                    newCell.addNeighbor( cells[row - 1][column] );
                    if ( column <= (columns - 2) )
                    {
                        // neighbor to the northeast
                        newCell.addNeighbor( cells[row - 1][column + 1] );
                    }
                }
                if ( column > 0 )
                {
                    // neighbor to the west
                    newCell.addNeighbor( cells[row][column - 1] );
                    if ( row > 0 )
                    {
                        // neighbor to the northwest
                        newCell.addNeighbor( cells[row - 1][column - 1] );
                    }
                }
            }
        }
    }

    /**
     * @param row
     *            row of the requested cell
     * @param column
     *            column of the requested cell
     * @return the cell at the specified coordinates
     * @see Cell
     */
    public Cell getCellAt(int row,
                          int column)
    {
        return cells[row][column];
    }
```

```
/**
 * @return the number of rows in this grid
 * @see #getNumberOfColumns()
 */
public int getNumberOfRows()
{
    return cells.length;
}

/**
 * @return the number of columns in this grid
 * @see #getNumberOfRows()
 */
public int getNumberOfColumns()
{
    return cells[0].length;
}

/**
 * Moves this grid to its next generation
 *
 * @return <code>true</code> if the state changed, otherwise false
 * @see #transitionState()
 */
public boolean nextGeneration()
{
    boolean didStateChange = false;
    try
    {
        RuleBase ruleBase = RuleBaseFactory.getRuleBase( );
        WorkingMemory workingMemory = ruleBase.newWorkingMemory( );
        // for (Cell[] rowOfCells : cells) {
        Cell[] rowOfCells = null;
        Cell cell = null;
        for ( int i = 0; i < cells.length; i++ )
        {
            rowOfCells = cells[i];
            for ( int j = 0; j < rowOfCells.length; j++ )
            {
                cell = rowOfCells[j];
                workingMemory.assertObject( cell );
            }
        }
        workingMemory.fireAllRules( );
        didStateChange = transitionState( );
    }
    catch ( Exception e )
    {
        e.printStackTrace( );
    }
    return didStateChange;
}

/**
 * @return the number of cells in the grid that are alive
 * @see CellState
 */
public int getNumberOfLiveCells()
{
    int number = 0;
    Cell[] rowOfCells = null;
    Cell cell = null;
    for ( int i = 0; i < cells.length; i++ )
    {
        rowOfCells = cells[i];
        // for (Cell cell : rowOfCells) {
        for ( int j = 0; j < rowOfCells.length; j++ )
        {
            cell = rowOfCells[j];
            if ( cell.getCellState( ) == CellState.LIVE )
            {
                number++;
            }
        }
    }
    return number;
}

/**
 * kills all cells in the grid
 */
public void killAll()
{
    Cell[] rowOfCells = null;
    Cell cell = null;
    for ( int i = 0; i < cells.length; i++ )
    {
        rowOfCells = cells[i];
        // for (Cell cell : rowOfCells) {
        for ( int j = 0; j < rowOfCells.length; j++ )
        {
            cell = rowOfCells[j];
            cell.setCellState( CellState.DEAD );
        }
    }
}

/**
 * Transitions this grid to its next state of evolution
 *
 * @return <code>true</code> if the state changed, otherwise false
 * @see #nextGeneration()
 */
public boolean transitionState()
{
    boolean stateChanged = false;
    Cell[] rowOfCells = null;
    Cell cell = null;
    for ( int i = 0; i < cells.length; i++ )
    {
        rowOfCells = cells[i];
        // for (Cell cell : rowOfCells) {
        for ( int j = 0; j < rowOfCells.length; j++ )
        {
            cell = rowOfCells[j];
            stateChanged |= cell.transitionState( );
        }
    }
    return stateChanged;
}

/**
 * Populates the grid with a <code>ConwayPattern</code>
 *
 * @param pattern
```

```
     *            pattern to populate the grid with
     * @see ConwayPattern
     */
    public void setPattern(ConwayPattern pattern)
    {
        boolean[][] gridData = pattern.getPattern( );
        int gridWidth = gridData[0].length;
        int gridHeight = gridData.length;

        int columnOffset = 0;
        int rowOffset = 0;

        if ( gridWidth > getNumberOfColumns( ) )
        {
            gridWidth = getNumberOfColumns( );
        }
        else
        {
            columnOffset = (getNumberOfColumns( ) - gridWidth) / 2;
        }

        if ( gridHeight > getNumberOfRows( ) )
        {
            gridHeight = getNumberOfRows( );
        }
        else
        {
            rowOffset = (getNumberOfRows( ) - gridHeight) / 2;
        }

        killAll( );
        for ( int column = 0; column < gridWidth; column++ )
        {
            for ( int row = 0; row < gridHeight; row++ )
            {
                if ( gridData[row][column] )
                {
                    Cell cell = getCellAt( row + rowOffset,
                                           column + columnOffset );
                    cell.setCellState( CellState.LIVE );
                }
            }
        }
    }
}

//*******
//Class- CellGridCanvas
//*******

package org.thesis.examples.conway.ui;


import javax.swing.*;

import org.thesis.examples.conway.Cell;
import org.thesis.examples.conway.CellGrid;
import org.thesis.examples.conway.CellState;

import java.awt.*;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionAdapter;
```

```
public class CellGridCanvas extends Canvas
{
    private Image              offScreenImage;
    private Image              backgroundImage;
    private final int          cellSize;
    private final CellGrid     cellGrid;
    private final Image        liveCellImage    = new ImageIcon(
CellGridCanvas.class.getResource( "liveCellImage.gif" ) ).getImage( );

    private static final Color BACKGROUND_COLOR = Color.gray;
    private static final Color GRID_COLOR       = BACKGROUND_COLOR.brighter( );

    /**
     * Constructs a CellGridCanvas.
     *
     * @param cellGrid
     *            the GoL cellgrid
     */
    public CellGridCanvas(CellGrid cellGrid)
    {
        this.cellGrid = cellGrid;
        this.cellSize = liveCellImage.getWidth( this );

        setBackground( GRID_COLOR );

        addMouseListener( new MouseAdapter( ) {
            /**
             * Invoked when a mouse button has been pressed on a component.
             */
            public void mousePressed(MouseEvent e)
            {
                toggleCellAt( e.getX( ),
                              e.getY( ) );
            }
        } );

        addMouseMotionListener( new MouseMotionAdapter( ) {

            public void mouseDragged(MouseEvent e)
            {
                Cell cell = getCellAtPoint( e.getX( ),
                                            e.getY( ) );
                if ( cell != null )
                {
                    cell.setCellState( CellState.LIVE );
                    repaint( );
                }
            }
        } );
    }

    private void toggleCellAt(int x,
                              int y)
    {
        Cell cell = getCellAtPoint( x,
                                    y );
        if ( cell != null )
        {
            if ( cell.getCellState( ) == CellState.LIVE )
            {
                cell.setCellState( CellState.DEAD );
            }
            else
            {
                cell.setCellState( CellState.LIVE );
            }
            repaint( );
        }
    }

    private Cell getCellAtPoint(int x,
                                int y)
    {
        Cell cell = null;

        int column = x / cellSize;
        int row = y / cellSize;
        final int numberOfColumns = cellGrid.getNumberOfColumns( );
        final int numberOfRows = cellGrid.getNumberOfRows( );

        if ( column >= 0 && column < numberOfColumns && row >= 0 && row <
numberOfRows )
        {
            cell = cellGrid.getCellAt( row,
                                       column );
        }

        return cell;
    }

    /**
     * Use double buffering.
     *
     * @see java.awt.Component#update(java.awt.Graphics)
     */
    public void update(Graphics g)
    {
        Dimension d = getSize( );
        if ( (offScreenImage == null) )
        {
            offScreenImage = createImage( d.width,
                                          d.height );
        }
        paint( offScreenImage.getGraphics( ) );
        g.drawImage( offScreenImage,
                     0,
                     0,
                     null );
    }

    /**
     * Draw this generation.
     *
     * @see java.awt.Component#paint(java.awt.Graphics)
     */
    public void paint(Graphics g)
    {
        // Draw grid on background image, which is faster
        final int numberOfColumns = cellGrid.getNumberOfColumns( );
        final int numberOfRows = cellGrid.getNumberOfRows( );
        if ( backgroundImage == null )
        {
            Dimension d = getSize( );
            backgroundImage = createImage( d.width,
                                           d.height );
            Graphics backgroundImageGraphics = backgroundImage.getGraphics( );
            // draw background (MSIE doesn't do that)
            backgroundImageGraphics.setColor( getBackground( ) );
            backgroundImageGraphics.fillRect( 0,
                                              0,
                                              d.width,
                                              d.height );
            backgroundImageGraphics.setColor( BACKGROUND_COLOR );
            backgroundImageGraphics.fillRect( 0,
                                              0,
                                              cellSize * numberOfColumns - 1,
                                              cellSize * numberOfRows - 1 );
            backgroundImageGraphics.setColor( GRID_COLOR );
            for ( int x = 1; x < numberOfColumns; x++ )
            {
                backgroundImageGraphics.drawLine( x * cellSize - 1,
                                                  0,
                                                  x * cellSize - 1,
                                                  cellSize * numberOfRows - 1
);
            }
            for ( int y = 1; y < numberOfRows; y++ )
            {
                backgroundImageGraphics.drawLine( 0,
                                                  y * cellSize - 1,
                                                  cellSize * numberOfColumns -
1,

                                                  y * cellSize - 1 );
            }
        }
        g.drawImage( backgroundImage,
                     0,
                     0,
                     null );

        // draw populated cells
        for ( int row = 0; row < numberOfRows; row++ )
        {
            for ( int column = 0; column < numberOfColumns; column++ )
            {
                Cell cell = cellGrid.getCellAt( row,
                                                column );
                if ( cell.getCellState( ) == CellState.LIVE )
                {
                    g.drawImage( liveCellImage,
                                 column * cellSize,
                                 row * cellSize,
                                 this );
                }
            }
        }
    }

    /**
     * This is the preferred size.
     *
     * @see java.awt.Component#getPreferredSize()
     */
    public Dimension getPreferredSize()
    {
        final int numberOfColumns = cellGrid.getNumberOfColumns( );
        final int numberOfRows = cellGrid.getNumberOfRows( );
        return new Dimension( cellSize * numberOfColumns,
                                  cellSize * numberOfRows );
    }

    /**
     * This is the minimum size (size of one cell).
     *
     * @see java.awt.Component#getMinimumSize()
     */
    public Dimension getMinimumSize()
    {
        return new Dimension( cellSize,
                              cellSize );
    }

}

//*******
//Class- CellState
//*******


package org.thesis.examples.conway;



public class CellState
{

    public static final CellState LIVE = new CellState("LIVE");
    public static final CellState DEAD = new CellState("DEAD");

    private final String name;

    private CellState(String name)
    {
        this.name = name;
    }

    public String toString()
    {
        return "CellState: " + name;
    }
}


//*******
//Class- ConwayApplicationProperties
//*******

package org.thesis.examples.conway;

import java.util.ResourceBundle;


public class ConwayApplicationProperties
{
    private static ConwayApplicationProperties ourInstance = new
ConwayApplicationProperties( );

    public static ConwayApplicationProperties getInstance()
    {
        return ourInstance;
    }
    private final ResourceBundle resources;

    private ConwayApplicationProperties()
    {
        resources = ResourceBundle.getBundle( "conway" );
    }

    public static String getProperty(String propertyName)
    {
        return ourInstance.resources.getString( propertyName );
    }
}


//*******
//Class- ConwayConditionFactory
//*******

package org.thesis.examples.conway.rules.dsl;

import org.thesis.examples.conway.Cell;
import org.thesis.rule.Declaration;
import org.thesis.rule.InvalidRuleException;
import org.thesis.rule.Rule;
import org.thesis.semantics.base.ClassObjectType;
import org.thesis.smf.ConditionFactory;
import org.thesis.smf.Configuration;
import org.thesis.smf.FactoryException;
import org.thesis.spi.Condition;
import org.thesis.spi.RuleBaseContext;

import java.util.ArrayList;
import java.util.List;

public class ConwayConditionFactory
        implements
        ConditionFactory
{

    public Condition[] newCondition(Rule rule,
                                    RuleBaseContext context,
                                    final Configuration config) throws
FactoryException
    {
        Configuration[] configurations = config.getChildren();
        List conditions = new ArrayList();
        final String cellName = config.getAttribute( "cellName" );
        final Declaration cellDeclaration = getDeclaration( rule,
                                                            Cell.class,
                                                            cellName );
        for ( int i = 0; i < configurations.length; i++ )
        {
            Configuration childConfig1 = null;
            childConfig1 = configurations[i];


                        if ( childConfig1.getName().equals( "cellIsAlive" ) )
                        {
                            conditions.add( new IsCellAliveCondition(
cellDeclaration ) );
                        } else if ( childConfig1.getName().equals( "cellIsDead" ) )
                        {
```

```
                        conditions.add( new IsCellDeadCondition(
cellDeclaration ) );
                    } else if ( childConfig1.getName().equals(
"cellIsOverCrowded" ) )
                    {
                        conditions.add( new OvercrowdedCondition(
cellDeclaration) );
                    } else if ( childConfig1.getName().equals( "cellIsLonely" )
)
                    {
                        conditions.add( new LonelyCondition( cellDeclaration )
);
                    } else if ( childConfig1.getName().equals(
"cellIsRipeForBirth" ) )
                    {
                        conditions.add( new RipeForBirthCondition(
cellDeclaration) );
                    }
        }

        return (Condition[]) conditions.toArray( new
Condition[conditions.size()] );
    }

    private Declaration getDeclaration(Rule rule,
                                       Class clazz,
                                       String identifier) throws
FactoryException
    {
        Declaration declaration = rule.getParameterDeclaration( identifier );
        if ( declaration == null )
        {
            ClassObjectType type = new ClassObjectType( clazz );
            try
            {
                declaration = rule.addParameterDeclaration( identifier,
                                                            type );
            }
            catch ( InvalidRuleException e )
            {
                final FactoryException factoryException = new FactoryException(
"Error occurred establishing parameter." );
                factoryException.initCause( e );
                throw factoryException;
            }
        }
        return declaration;
    }


}


//*******
//Class- ConwayConsequenceFactory
//*******


package org.thesis.examples.conway.rules.dsl;

import org.thesis.examples.conway.Cell;
import org.thesis.rule.Declaration;
import org.thesis.rule.InvalidRuleException;
import org.thesis.rule.Rule;
import org.thesis.semantics.base.ClassObjectType;
import org.thesis.smf.Configuration;
import org.thesis.smf.ConsequenceFactory;
import org.thesis.smf.FactoryException;
import org.thesis.spi.Consequence;
import org.thesis.spi.RuleBaseContext;

public class ConwayConsequenceFactory
        implements
        ConsequenceFactory
{
    public Consequence newConsequence(Rule rule,
                                      RuleBaseContext context,
                                      Configuration config) throws
FactoryException
    {

        Configuration childConfig = null;
        Configuration[] configurations = config.getChildren();
        Consequence consequence = null;
        final String cellName = config.getAttribute( "cellName" );
        final Declaration cellDeclaration = getDeclaration( rule,
                                                            Cell.class,
                                                            cellName );
        for ( int i = 0; i < configurations.length; i++ )
        {
            childConfig = configurations[i];

            if ( childConfig.getName().equals( "giveBirthToCell" ) )
            {
                consequence = new GiveBirthConsequence( cellDeclaration );
            } else if ( childConfig.getName().equals( "killCell" ) )
            {
                consequence = new KillCellConsequence( cellDeclaration );
            }
        }


        return consequence;
    }

    private Declaration getDeclaration(Rule rule,
                                       Class clazz,
                                       String identifier) throws
FactoryException
    {
        Declaration declaration = rule.getParameterDeclaration( identifier );
        if ( declaration == null )
        {
            ClassObjectType type = new ClassObjectType( clazz );
            try
            {
                declaration = rule.addParameterDeclaration( identifier,
                                                            type );
            }
            catch ( InvalidRuleException e )
            {
                final FactoryException factoryException = new FactoryException(
"Error occurred establishing parameter." );
                factoryException.initCause( e );
                throw factoryException;
            }
        }
        return declaration;
    }

}


//*******
//Class- ConwayGUI
//*******

package org.thesis.examples.conway.ui;

import com.jgoodies.forms.builder.PanelBuilder;
import com.jgoodies.forms.factories.ButtonBarFactory;
import com.jgoodies.forms.layout.CellConstraints;
import com.jgoodies.forms.layout.FormLayout;
import foxtrot.Job;
import foxtrot.Worker;

import javax.swing.*;
import javax.swing.border.Border;
import javax.swing.border.EtchedBorder;

import org.thesis.examples.conway.CellGrid;
import org.thesis.examples.conway.ConwayApplicationProperties;
import org.thesis.examples.conway.patterns.ConwayPattern;

import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.StringTokenizer;

public class ConwayGUI extends JPanel
{
    private final JButton   nextGenerationButton;
    private final JButton   startStopButton;
    private final JButton   clearButton;
    private final JComboBox patternSelector = new JComboBox( );
    private final Timer     timer;

    public ConwayGUI()
    {
        super( new BorderLayout( ) );
        final String nextGenerationLabel =
ConwayApplicationProperties.getProperty( "next.generation.label" );
        nextGenerationButton = new JButton( nextGenerationLabel );
        final String startLabel = ConwayApplicationProperties.getProperty(
"start.label" );
        startStopButton = new JButton( startLabel );
        final String clearLabel = ConwayApplicationProperties.getProperty(
"clear.label" );
        clearButton = new JButton( clearLabel );
        final CellGrid grid = new CellGrid( 30,
                                            30 );
        final CellGridCanvas canvas = new CellGridCanvas( grid );
        JPanel panel = new JPanel( new BorderLayout( ) );
        panel.add( BorderLayout.CENTER,
                   canvas );
        Border etchedBorder = BorderFactory.createEtchedBorder(
EtchedBorder.LOWERED );
        Border outerBlankBorder = BorderFactory.createEmptyBorder( 5,
                                                                    5,
                                                                    5,
                                                                    5 );
        Border innerBlankBorder = BorderFactory.createEmptyBorder( 5,
                                                                    5,
                                                                    5,
                                                                    5 );

        Border border = BorderFactory.createCompoundBorder(
BorderFactory.createCompoundBorder( outerBlankBorder,

etchedBorder ),
                                                             innerBlankBorder );
        panel.setBorder( border );
        add( BorderLayout.CENTER,
             panel );
        add( BorderLayout.EAST,
             createControlPanel( ) );
        nextGenerationButton.addActionListener( new ActionListener( ) {
            public void actionPerformed(ActionEvent e)
            {
                Worker.post( new Job( ) {
                    public Object run()
                    {
                        grid.nextGeneration( );
                        return null;
                    }
                } );
                canvas.repaint( );
            }
        } );
        clearButton.addActionListener( new ActionListener( ) {
            public void actionPerformed(ActionEvent e)
            {
                Worker.post( new Job( ) {
                    public Object run()
                    {
                        grid.killAll( );
                        return null;
                    }
                } );
                canvas.repaint( );
            }
        } );

        ActionListener timerAction = new ActionListener( ) {
            public void actionPerformed(ActionEvent ae)
            {
                Worker.post( new Job( ) {
                    public Object run()
                    {
                        if ( !grid.nextGeneration( ) )
                        {
                            stopTimer( );
                        }
                        return null;
                    }
                } );
                canvas.repaint( );
            }
        };
        timer = new Timer( 500,
                           timerAction );
        startStopButton.addActionListener( new ActionListener( ) {
            public void actionPerformed(ActionEvent e)
            {
                if ( timer.isRunning( ) )
                {
                    stopTimer( );
                }
                else
                {
                    startTimer( );
                }
            }
        } );

        populatePatternSelector( );

        patternSelector.addActionListener( new ActionListener( ) {
            public void actionPerformed(ActionEvent e)
            {
                ConwayPattern pattern = (ConwayPattern)
patternSelector.getSelectedItem( );
                if ( pattern != null )
                {
                    grid.setPattern( pattern );
                    canvas.repaint( );
                }
            }
        } );

        patternSelector.setSelectedIndex( -1 );
    }

    private void populatePatternSelector()
    {
        String patternClassNames = ConwayApplicationProperties.getProperty(
"conway.pattern.classnames" );
        StringTokenizer tokenizer = new StringTokenizer( patternClassNames );

        String className = null;
        while ( tokenizer.hasMoreTokens( ) )
        {
            className = tokenizer.nextToken( ).trim( );
            try
            {
                Class clazz = Class.forName( className );
                if ( ConwayPattern.class.isAssignableFrom( clazz ) )
                {
                    patternSelector.addItem( clazz.newInstance( ) );
                }
                else
                {
                    System.err.println( "Invalid pattern class name: " +
className );
                }
            }
            catch ( Exception e )
            {
                System.err.println( "An error occurred populating patterns: "
);
                e.printStackTrace( );
            }
        }
    }

    private void startTimer()
    {
        final String stopLabel = ConwayApplicationProperties.getProperty(
"stop.label" );
        startStopButton.setText( stopLabel );
        nextGenerationButton.setEnabled( false );
        clearButton.setEnabled( false );
        patternSelector.setEnabled( false );
        timer.start( );
    }

    private void stopTimer()
    {
        timer.stop( );
        final String startLabel = ConwayApplicationProperties.getProperty(
"start.label" );
        startStopButton.setText( startLabel );
        nextGenerationButton.setEnabled( true );
        clearButton.setEnabled( true );
        patternSelector.setEnabled( true );
    }

    private JPanel createControlPanel()
    {
        FormLayout layout = new FormLayout( "pref, 3dlu, pref, 3dlu:grow",
                                            "pref, 15dlu, pref, 15dlu, pref,
3dlu:grow, pref" );
        PanelBuilder builder = new PanelBuilder( layout );
        CellConstraints cc = new CellConstraints( );

        String title = ConwayApplicationProperties.getProperty( "app.title" );
        builder.addLabel( title,
                          cc.xywh( 1,
                                   1,
                                   layout.getColumnCount( ),
                                   1 ) );

        String info = ConwayApplicationProperties.getProperty(
"app.description" );
        builder.addLabel( info,
                          cc.xywh( 1,
                                   3,
                                   layout.getColumnCount( ),
                                   1 ) );

        final String patternLabel = ConwayApplicationProperties.getProperty(
"pattern.label" );
        builder.addLabel( patternLabel,
                          cc.xy( 1,
                                 5 ) );

        builder.add( patternSelector,
                     cc.xy( 3,
                            5 ) );
        JPanel buttonPanel = ButtonBarFactory.buildLeftAlignedBar(
nextGenerationButton,

startStopButton,
                                                                     clearButton
);
        builder.add( buttonPanel,
                     cc.xywh( 1,
                              7,
                              layout.getColumnCount( ),
                                1 ) );

        Border etchedBorder = BorderFactory.createEtchedBorder(
EtchedBorder.LOWERED );
        Border outerBlankBorder = BorderFactory.createEmptyBorder( 5,
                                                                   5,
                                                                   5,
                                                                   5 );
        Border innerBlankBorder = BorderFactory.createEmptyBorder( 5,
                                                                   5,
                                                                   5,
                                                                   5 );
        Border border = BorderFactory.createCompoundBorder(
BorderFactory.createCompoundBorder( outerBlankBorder,

etchedBorder ),
                                                            innerBlankBorder );
        builder.setBorder( border );
        return builder.getPanel( );
    }

    public static void main(String[] args)
    {
        if ( args.length != 1 )
        {
            System.out.println( "Usage: " + ConwayGUI.class.getName( ) + " [drl
file]" );
            return;
        }
        System.out.println( "Using drl: " + args[0] );

        System.setProperty( "conway.drl.file",
                            args[0] );

        final String appTitle = ConwayApplicationProperties.getProperty(
"app.title" );
        JFrame f = new JFrame( appTitle );
        f.setResizable( false );
        f.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        f.getContentPane( ).add( BorderLayout.CENTER,
                                 new ConwayGUI( ) );
        f.pack( );
        f.setVisible( true );
    }
}


//*******
//Class- ConwayPattern
//*******

package org.thesis.examples.conway.patterns;

import java.io.Serializable;

public interface ConwayPattern
    extends
    Serializable
{

    /**
     * This method should return a 2 dimensional array of boolean that
represent
```

```
     * a conway grid, with <code>true</code> values in the positions where
     * cells are alive
     *
     * @return array representing a conway grid
     */
    public boolean[][] getPattern();

    /**
     * @return the name of this pattern
     */
    public String getPatternName();
}


//*******
//Class- GiveBirthConsequence
//*******


package org.thesis.examples.conway.rules.dsl;

import org.thesis.spi.Consequence;
import org.thesis.spi.Tuple;
import org.thesis.rule.Declaration;
import org.thesis.examples.conway.Cell;
import org.thesis.examples.conway.CellState;

public class GiveBirthConsequence implements Consequence
{
    private final Declaration cellDeclaration;

    public GiveBirthConsequence(Declaration cellDeclaration)
    {
        this.cellDeclaration = cellDeclaration;
    }

    /**
     * Execute the consequence for the supplied matching <code>Tuple</code>.
     *
     * @param tuple
     *           The matching tuple.
     */
    public void invoke(Tuple tuple)
    {
        Cell cell = (Cell) tuple.get( cellDeclaration );
        cell.queueNextCellState( CellState.LIVE );
    }
}



//*******
//Class- IsCellAliveCondition
//*******

package org.thesis.examples.conway.rules.dsl;

import org.thesis.examples.conway.Cell;
import org.thesis.examples.conway.CellState;
import org.thesis.rule.Declaration;
import org.thesis.spi.Tuple;
import org.thesis.spi.Condition;
```

```
public class IsCellAliveCondition implements Condition
{
    protected final Declaration cellDeclaration;


    public IsCellAliveCondition( Declaration cellDeclaration )
    {
        this.cellDeclaration = cellDeclaration;
    }

    /**
     * Determine if the supplied <code>Tuple</code> is allowed by this
     * condition.
     *
     * @param tuple
     *           The <code>Tuple</code> to test.
     *
     * @return <code>true</code> if the <code>Tuple</code> passes this
     *         condition, else <code>false</code>.
     *
     */
    public boolean isAllowed(Tuple tuple)
    {
        Cell cell = (Cell) tuple.get( cellDeclaration );
        return cell.getCellState() == CellState.LIVE;
    }

    /**
     * Retrieve the array of <code>Declaration</code> s required by this
     * condition to perform its duties.
     *
     * @return The array of <code>Declarations</code> expected on incoming
     *         <code>Tuples</code>.
     */
    public Declaration[] getRequiredTupleMembers()
    {
        return new Declaration[]{cellDeclaration};
    }
}


//*******
//Class- IsCellDeadCondition
//*******

package org.thesis.examples.conway.rules.dsl;

import org.thesis.examples.conway.Cell;
import org.thesis.examples.conway.CellState;
import org.thesis.rule.Declaration;
import org.thesis.spi.Tuple;
import org.thesis.spi.Condition;

public class IsCellDeadCondition implements Condition
{
    protected final Declaration cellDeclaration;

    public IsCellDeadCondition(Declaration cellDeclaration)
    {
        this.cellDeclaration = cellDeclaration;
    }

    /**
```

```
     * Determine if the supplied <code>Tuple</code> is allowed by this
     * condition.
     *
     * @param tuple
     *           The <code>Tuple</code> to test.
     *
     * @return <code>true</code> if the <code>Tuple</code> passes this
     *         condition, else <code>false</code>.
     *
     */
    public boolean isAllowed(Tuple tuple)
    {
        Cell cell = (Cell) tuple.get( cellDeclaration );
        return cell.getCellState() == CellState.DEAD;
    }

    /**
     * Retrieve the array of <code>Declaration</code> s required by this
     * condition to perform its duties.
     *
     * @return The array of <code>Declarations</code> expected on incoming
     *         <code>Tuples</code>.
     */
    public Declaration[] getRequiredTupleMembers()
    {
        return new Declaration[]{cellDeclaration};
    }
}


//*******
//Class- KillCellConsequence
//*******


package org.thesis.examples.conway.rules.dsl;

import org.thesis.spi.Consequence;
import org.thesis.spi.Tuple;
import org.thesis.rule.Declaration;
import org.thesis.examples.conway.Cell;
import org.thesis.examples.conway.CellState;

public class KillCellConsequence implements Consequence
{
    private final Declaration cellDeclaration;

    public KillCellConsequence(Declaration cellDeclaration)
    {
        this.cellDeclaration = cellDeclaration;
    }


    /**
     * Execute the consequence for the supplied matching <code>Tuple</code>.
     *
     * @param tuple
     *           The matching tuple.
     */
    public void invoke(Tuple tuple)
    {
        Cell cell = (Cell) tuple.get( cellDeclaration );
        cell.queueNextCellState( CellState.DEAD );
    }
}


//*******
//Class- LonelyCondition
//*******


package org.thesis.examples.conway.rules.dsl;

import org.thesis.examples.conway.Cell;
import org.thesis.rule.Declaration;
import org.thesis.spi.Tuple;
import org.thesis.spi.Condition;

public class LonelyCondition implements Condition
{
    protected final Declaration cellDeclaration;

    public LonelyCondition(Declaration cellDeclaration)
    {
        this.cellDeclaration = cellDeclaration;
    }

    /**
     * Determine if the supplied <code>Tuple</code> is allowed by this
     * condition.
     *
     * @param tuple
     *            The <code>Tuple</code> to test.
     *
     * @return <code>true</code> if the <code>Tuple</code> passes this
     *         condition, else <code>false</code>.
     *
     */
    public boolean isAllowed(Tuple tuple)
    {
        Cell cell = (Cell) tuple.get( cellDeclaration );
        int numberOfLiveNeighborsCellHas = cell.getNumberOfLiveNeighbors();
        boolean isAllowed = ( numberOfLiveNeighborsCellHas < 2 );
        return isAllowed;
    }

    /**
     * Retrieve the array of <code>Declaration</code> s required by this
     * condition to perform its duties.
     *
     * @return The array of <code>Declarations</code> expected on incoming
     *         <code>Tuples</code>.
     */
    public Declaration[] getRequiredTupleMembers()
    {
        return new Declaration[]{cellDeclaration};
    }
}


//*******
//Class- OvercrowdedCondition
//*******
```

```
package org.thesis.examples.conway.rules.dsl;

import org.thesis.examples.conway.Cell;
import org.thesis.rule.Declaration;
import org.thesis.spi.Condition;
import org.thesis.spi.Tuple;

public class OvercrowdedCondition implements Condition
{
    protected final Declaration cellDeclaration;

    public OvercrowdedCondition(Declaration cellDeclaration)
    {
        this.cellDeclaration = cellDeclaration;
    }

    /**
     * Determine if the supplied <code>Tuple</code> is allowed by this
     * condition.
     *
     * @param tuple
     *            The <code>Tuple</code> to test.
     *
     * @return <code>true</code> if the <code>Tuple</code> passes this
     *         condition, else <code>false</code>.
     *
     */
    public boolean isAllowed(Tuple tuple)
    {
        Cell cell = (Cell) tuple.get( cellDeclaration );
        int numberOfLiveNeighborsCellHas = cell.getNumberOfLiveNeighbors();
        boolean isAllowed = ( numberOfLiveNeighborsCellHas > 3 );
        return isAllowed;
    }

    /**
     * Retrieve the array of <code>Declaration</code> s required by this
     * condition to perform its duties.
     *
     * @return The array of <code>Declarations</code> expected on incoming
     *         <code>Tuples</code>.
     */
    public Declaration[] getRequiredTupleMembers()
    {
        return new Declaration[]{cellDeclaration};
    }
}


//*******
//Class- RipeForBirthCondition
//*******

package org.thesis.examples.conway.rules.dsl;

import org.thesis.examples.conway.Cell;
import org.thesis.rule.Declaration;
import org.thesis.spi.Tuple;
import org.thesis.spi.Condition;

public class RipeForBirthCondition implements Condition
{
    protected final Declaration cellDeclaration;
    public RipeForBirthCondition( Declaration cellDeclaration)
    {
        this.cellDeclaration = cellDeclaration;
    }

    /**
     * Determine if the supplied <code>Tuple</code> is allowed by this
     * condition.
     *
     * @param tuple
     *            The <code>Tuple</code> to test.
     *
     * @return <code>true</code> if the <code>Tuple</code> passes this
     *         condition, else <code>false</code>.
     *
     */
    public boolean isAllowed(Tuple tuple)
    {
        Cell cell = (Cell) tuple.get( cellDeclaration );
        int numberOfLiveNeighborsCellHas = cell.getNumberOfLiveNeighbors();
        boolean isAllowed = ( numberOfLiveNeighborsCellHas == 3 );
        return isAllowed;
    }

    /**
     * Retrieve the array of <code>Declaration</code> s required by this
     * condition to perform its duties.
     *
     * @return The array of <code>Declarations</code> expected on incoming
     *         <code>Tuples</code>.
     */
    public Declaration[] getRequiredTupleMembers()
    {
        return new Declaration[]{cellDeclaration};
    }
}


//*******
//Class- RuleBaseFactory
//*******

package org.thesis.examples.conway.rules;

import org.thesis.RuleBase;
import org.thesis.examples.conway.CellGrid;
import org.thesis.io.RuleBaseLoader;

import java.net.URL;

public class RuleBaseFactory
{

    private static RuleBaseFactory ourInstance      = new RuleBaseFactory( );
    private static final String    DEFAULT_DRL_FILE = "conway.java.drl";

    private RuleBase               ruleBase;

    public static RuleBaseFactory getInstance()
    {
        return ourInstance;
    }
    private RuleBaseFactory()
    {
        try
        {
            String conwayDrlFile = System.getProperty( "conway.drl.file" );
            if ( conwayDrlFile == null )
            {
                System.err.println( "conway.drl.file system property not
specified. using default: " + DEFAULT_DRL_FILE );
                conwayDrlFile = DEFAULT_DRL_FILE;
            }
            System.out.println( "loading drl file: " + conwayDrlFile );
            URL resource = CellGrid.class.getResource( conwayDrlFile );
            ruleBase = RuleBaseLoader.loadFromUrl( resource );
        }
        catch ( Exception e )
        {
            throw new RuntimeException( "Unable To Initialise
RuleBaseFactory:\n" + e.getMessage() );
        }
    }

    public static RuleBase getRuleBase()
    {
        return ourInstance.ruleBase;
    }
}



//*******
//Case Study 2 - PetStore Application
//*******



//*******
//Class- PetStore
//*******


package org.thesis.examples.petstore;


import java.net.URL;
import java.util.Vector;

import org.thesis.RuleBase;
import org.thesis.io.RuleBaseLoader;

public class PetStore
{
    public static void main(String[] args)
    {
        if ( args.length != 1 )
        {
            System.out.println( "Usage: " + PetStore.class.getName( )
                                + " [drl file]" );
            return;
        }
        System.out.println( "Using drl: " + args[0] );
        try
        {
            URL url = PetStore.class.getResource( args[0] );
            RuleBase ruleBase = RuleBaseLoader.loadFromUrl( url );

            Vector stock = new Vector( );
            stock.add( new CartItem( "Gold Fish", 5 ) );
            stock.add( new CartItem( "Fish Tank", 25 ) );
            stock.add( new CartItem( "Fish Food", 2 ) );

            //The callback is responsible for populating working memory and
            // fireing all rules
            PetStoreUI ui = new PetStoreUI( stock,
                                            new CheckoutCallback( ruleBase ) );
            ui.createAndShowGUI( );
        }
        catch ( Exception e )
        {
            e.printStackTrace( );
        }
    }

}


//*******
//Class- PetStoreUI
//*******

package org.thesis.examples.petstore;


import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.GridLayout;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.util.ArrayList;
import java.util.List;
import java.util.Vector;

import javax.swing.AbstractButton;
import javax.swing.BorderFactory;
import javax.swing.BoxLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JSplitPane;
import javax.swing.JTable;
import javax.swing.JTextArea;
import javax.swing.ListSelectionModel;
import javax.swing.ScrollPaneConstants;
import javax.swing.table.AbstractTableModel;
import javax.swing.table.DefaultTableCellRenderer;
import javax.swing.table.TableColumnModel;

public class PetStoreUI extends JPanel
{
    private JTextArea        output;

    private TableModel       tableModel;
    private CheckoutCallback callback;

    /**
     * Build UI using specified items and using the given callback to pass the
     * items and jframe reference to the DRL application
     *
     * @param listData
     * @param callback
     */
    public PetStoreUI(Vector items, CheckoutCallback callback)
    {
        super( new BorderLayout( ) );
        this.callback = callback;

        //Create main vertical split panel
        JSplitPane splitPane = new JSplitPane( JSplitPane.VERTICAL_SPLIT );
        add( splitPane, BorderLayout.CENTER );

        //create top half of split panel and add to parent
        JPanel topHalf = new JPanel( );
        topHalf.setLayout( new BoxLayout( topHalf, BoxLayout.X_AXIS ) );
        topHalf.setBorder( BorderFactory.createEmptyBorder( 5, 5, 0, 5 ) );
        topHalf.setMinimumSize( new Dimension( 400, 50 ) );
        topHalf.setPreferredSize( new Dimension( 450, 250 ) );
        splitPane.add( topHalf );

        //create bottom top half of split panel and add to parent
        JPanel bottomHalf = new JPanel( new BorderLayout( ) );
        bottomHalf.setMinimumSize( new Dimension( 400, 50 ) );
        bottomHalf.setPreferredSize( new Dimension( 450, 300 ) );
        splitPane.add( bottomHalf );

        //Container that list container that shows available store items
        JPanel listContainer = new JPanel( new GridLayout( 1, 1 ) );
        listContainer.setBorder( BorderFactory.createTitledBorder( "List" ) );
        topHalf.add( listContainer );

        //Create JList for items, add to scroll pane and then add to parent
        // container
        JList list = new JList( items );
        ListSelectionModel listSelectionModel = list.getSelectionModel( );
        listSelectionModel
                              .setSelectionMode(
ListSelectionModel.SINGLE_SELECTION );
        //handler adds item to shopping cart
        list.addMouseListener( new ListSelectionHandler( ) );
        JScrollPane listPane = new JScrollPane( list );
        listContainer.add( listPane );

        JPanel tableContainer = new JPanel( new GridLayout( 1, 1 ) );
        tableContainer.setBorder( BorderFactory.createTitledBorder( "Table" )
);
        topHalf.add( tableContainer );

        //Container that displays table showing items in cart
        tableModel = new TableModel( );
        JTable table = new JTable( tableModel );
        //handler removes item to shopping cart
        table.addMouseListener( new TableSelectionHandler( ) );
        ListSelectionModel tableSelectionModel = table.getSelectionModel( );
        tableSelectionModel
```

```
                            .setSelectionMode(
ListSelectionModel.SINGLE_SELECTION );
        TableColumnModel tableColumnModel = table.getColumnModel( );
        //notice we have a custom renderer for each column as both columns
        // point to the same underlying object
        tableColumnModel.getColumn( 0 ).setCellRenderer( new NameRenderer( ) );
        tableColumnModel.getColumn( 1 ).setCellRenderer( new PriceRenderer( )
);
        tableColumnModel.getColumn( 1 ).setMaxWidth( 50 );

        JScrollPane tablePane = new JScrollPane( table );
        tablePane.setPreferredSize( new Dimension( 150, 100 ) );
        tableContainer.add( tablePane );

        //Create panel for checkout button and add to bottomHalf parent
        JPanel checkoutPane = new JPanel( );
        JButton button = new JButton( "Checkout" );
        button.setVerticalTextPosition( AbstractButton.CENTER );
        button.setHorizontalTextPosition( AbstractButton.LEADING );
        //attach handler to assert items into working memory
        button.addMouseListener( new CheckoutButtonHandler( ) );
        button.setActionCommand( "checkout" );
        checkoutPane.add( button );
        bottomHalf.add( checkoutPane, BorderLayout.NORTH );

        button = new JButton( "Reset" );
        button.setVerticalTextPosition( AbstractButton.CENTER );
        button.setHorizontalTextPosition( AbstractButton.TRAILING );
        //attach handler to assert items into working memory
        button.addMouseListener( new ResetButtonHandler( ) );
        button.setActionCommand( "reset" );
        checkoutPane.add( button );
        bottomHalf.add( checkoutPane, BorderLayout.NORTH );

        //Create output area, imbed in scroll area an add to bottomHalf parent
        //Scope is at instance level so it can be easily referenced from other
        // methods
        output = new JTextArea( 1, 10 );
        output.setEditable( false );
        JScrollPane outputPane = new JScrollPane(
                                                  output,

ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,

ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED );
        bottomHalf.add( outputPane, BorderLayout.CENTER );
    }

    /**
     * Create and show the GUI
     *
     */
    public void createAndShowGUI()
    {
        //Create and set up the window.
        JFrame frame = new JFrame( "Pet Store Demo" );
        frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );

        setOpaque( true );
        frame.setContentPane( this );

        //Display the window.
        frame.pack( );
        frame.setVisible( true );
    }

    /**
     * Adds the selected item to the table
     */
    private class ListSelectionHandler extends MouseAdapter
    {
        public void mouseReleased(MouseEvent e)
        {
            JList jlist = ( JList ) e.getSource( );
            tableModel.addItem( ( CartItem ) jlist.getSelectedValue( ) );
        }
    }

    /**
     * Removes the selected item from the table
     */
    private class TableSelectionHandler extends MouseAdapter
    {
        public void mouseReleased(MouseEvent e)
        {
            JTable jtable = ( JTable ) e.getSource( );
            TableModel tableModel = ( TableModel ) jtable.getModel( );
            tableModel.removeItem( jtable.getSelectedRow( ) );
        }
    }

    /**
     * Calls the referenced callback, passing a the jrame and selected items.
     *
     */
    private class CheckoutButtonHandler extends MouseAdapter
    {
        public void mouseReleased(MouseEvent e)
        {
            JButton button = ( JButton ) e.getComponent( );
            try
            {
                output
                      .append( callback
                                      .checkout(
                                                ( JFrame ) button

.getTopLevelAncestor( ),
                                                tableModel.getItems( ) ) );
            }
            catch ( org.thesis.FactException fe )
            {
                fe.printStackTrace( );
            }
        }
    }

    /**
     * Resets the shopping cart, allowing the user to begin again.
     *
     */
    private class ResetButtonHandler extends MouseAdapter
    {
        public void mouseReleased(MouseEvent e)
        {
            JButton button = ( JButton ) e.getComponent( );
            output.setText( null );
            tableModel.clear( );
            System.out.println( "------ Reset ------" );
        }
    }

    /**
     * Used to render the name column in the table
     */
    private class NameRenderer extends DefaultTableCellRenderer
    {
        public NameRenderer()
        {
            super( );
        }

        public void setValue(Object object)
        {
            CartItem item = ( CartItem ) object;
            setText( item.getName( ) );
        }
    }

    /**
     * Used to render the price column in the table
     */
    private class PriceRenderer extends DefaultTableCellRenderer
    {
        public PriceRenderer()
        {
            super( );
        }

        public void setValue(Object object)
        {
            CartItem item = ( CartItem ) object;
            setText( Double.toString( item.getCost( ) ) );
        }
    }

    /**
     * This is the table model used to represent the users shopping cart While
     * we have two colums, both columns point to the same object. We user a
     * different renderer to display the different information abou the object

     * name and price.
     */
    private class TableModel extends AbstractTableModel
    {
        private String[]  columnNames = {"Name", "Price"};

        private ArrayList items;

        public TableModel()
        {
            super( );
            items = new ArrayList( );
        }

        public int getColumnCount()
        {
            return columnNames.length;
        }
        public int getRowCount()
        {
            return items.size( );
        }

        public String getColumnName(int col)
        {
            return columnNames[col];
        }

        public Object getValueAt(int row, int col)
        {
            return items.get( row );
        }

        public Class getColumnClass(int c)
        {
            return CartItem.class;
        }

        public void addItem(CartItem item)
        {
            items.add( item );
            fireTableRowsInserted( items.size( ), items.size( ) );
        }

        public void removeItem(int row)
        {
            items.remove( row );
            fireTableRowsDeleted( row, row );
        }

        public List getItems()
        {
            return items;
        }

        public void clear()
        {
            int lastRow = items.size( );
            items.clear( );
            fireTableRowsDeleted( 0, lastRow );
        }
    }

}

//*******
//Class- ShoppingCart
//*******


package org.thesis.examples.petstore;


import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;

public class ShoppingCart
```

```
{
    private List          items;

    private double        discount;

    private Map           states;

    private static String newline = System.getProperty( "line.separator" );

    public ShoppingCart()
    {
        states = new HashMap( );
        this.items = new ArrayList( );
        this.discount = 0;
    }

    public boolean getState(String state)
    {
        if ( states.containsKey( state ) )
        {
            return ( ( Boolean ) states.get( state ) ).booleanValue( );
        }
        else
        {
            return false;
        }
    }

    public void setState(String state, boolean value)
    {
        states.put( state, new Boolean( value ) );
    }

    public void setDiscount(double discount)
    {
        this.discount = discount;
    }

    public double getDiscount()
    {
        return this.discount;
    }

    public void addItem(CartItem item)
    {
        this.items.add( item );
    }

    public List getItems()
    {
        return this.items;
    }

    public List getItems(String name)
    {
        ArrayList matching = new ArrayList( );

        Iterator itemIter = getItems( ).iterator( );
        CartItem eachItem = null;

        while ( itemIter.hasNext( ) )
        {
            eachItem = ( CartItem ) itemIter.next( );
            if ( eachItem.getName( ).equals( name ) )
            {
                matching.add( eachItem );
            }
        }

        return matching;
    }

    public double getGrossCost()
    {
        Iterator itemIter = getItems( ).iterator( );
        CartItem eachItem = null;

        double cost = 0.00;

        while ( itemIter.hasNext( ) )
        {
            eachItem = ( CartItem ) itemIter.next( );

            cost += eachItem.getCost( );
        }

        return cost;
    }

    public double getDiscountedCost()
    {
        double cost = getGrossCost( );
        double discount = getDiscount( );

        double discountedCost = cost * ( 1 - discount );

        return discountedCost;
    }

    public String toString()
    {
        StringBuffer buf = new StringBuffer( );

        buf.append( "ShoppingCart:" + newline );

        Iterator itemIter = getItems( ).iterator( );

        while ( itemIter.hasNext( ) )
        {
            buf.append( "\t" + itemIter.next( ) + newline );
        }

        buf.append( "gross total=" + getGrossCost( ) + newline );
        buf.append( "discounted total=" + getDiscountedCost( ) + newline );

        return buf.toString( );
    }
}


//*******
//Class- CartItem
//*******
```

```
package org.thesis.examples.petstore;


public class CartItem
{
    private String name;

    private double cost;

    public CartItem(String name, double cost)
    {
        this.name = name;
        this.cost = cost;
    }

    public String getName()
    {
        return this.name;
    }

    public double getCost()
    {
        return this.cost;
    }

    public String toString()
    {
        return name + " " + this.cost;
    }

}


//*******
//Class- CheckoutCallback
//*******

package org.thesis.examples.petstore;

import java.util.List;

import javax.swing.JFrame;

import org.thesis.FactException;
import org.thesis.RuleBase;
import org.thesis.WorkingMemory;


public class CheckoutCallback
{
    RuleBase ruleBase;

    public CheckoutCallback(RuleBase ruleBase)
    {
        this.ruleBase = ruleBase;
    }

    /**
     * Populate the cart and assert into working memory Pass Jframe reference
     * for user interaction
     *
     * @param frame
     * @param items
     * @return cart.toString();
     */
    public String checkout(JFrame frame, List items) throws FactException
    {
        ShoppingCart cart = new ShoppingCart( );

        //Iterate through list and add to cart
        for ( int i = 0; i < items.size( ); i++ )
        {
            cart.addItem( ( CartItem ) items.get( i ) );
        }

        //add the JFrame to the ApplicationData to allow for user interaction
        WorkingMemory workingMemory = ruleBase.newWorkingMemory( );
        workingMemory.setApplicationData( "frame", frame );
        workingMemory.assertObject( cart );
        workingMemory.fireAllRules( );

        //returns the state of the cart
        return cart.toString( );
    }
}
```

REFERENCES

R. Ross. The *Business Rule Book: Classifying, Defining, and Modeling Rules*, 2nd Ed., Database Research Group,1997.

D. Hay and K. Healy. *Defining business rules - what are they really, GUIDE Business Rule Report.* http://www.businessrulesgroup.org/first_paper/br01c1.htm, 2000.

Edward J. Barkmeyer, Evan K. Wallace and Ravi Raman. NIST "Position Paper" for the W3C Workshop on *Rule Languages for Interoperability*, 2002.

B. Von Halle. *Business Rules Applied: Building Better Systems using the Business Rules Approach*, Wiley, 2001.

David Zygmont. *New Automation Solution - Significant Advantage in Development of Enterprise Applications*, Wiley, 1999.

Hoi Chan. CommonRules http://www.alphaworks.ibm.com/tech/commonrules 2004.

S. Bohner. *Extending Software Change Impact Analysis into COTS Components*, IEEE/NASA Software Engineering Workshop, December 2002.

Harold Boley, Benjamin Grosof, Michael Sintek, Said Tabet, Gerd Wagner. *RuleML Design,* 2002. http://www.ruleml.org/indesign.html

Harold Boley, Said Tabet, Gerd Wagner. *Design Rationale of RuleML: A Markup Language for Semantic Web Rules*, 2001.

Jae Kyu Lee and Mye M. Sohn. *The eXtensible Rule Markup Language, Communications of the ACM*, Volume 46, Issue 5, pp. 59-64,. May 2003.

Gerd Wagner, Grigoris Antoniou, Said Tabet, and Harold Boley. *The Abstract Syntax of RuleML – Towards a General Web Rule Language Framework.*

Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence, 2004.

Mark C. Little and Stuart M. Wheater. *Building Configurable Applications in Java,* Proceedings of the 4th IEEE International Conference on Configurable Distributed Systems, May 1998.

Mark C. Little and Stuart M. Wheater. *The Design and Implementation of a Framework for Configurable Software,* 3rd International Conference on

Configurable Distributed Systems, 1996.

Ross Gardler, Nikolay Mehandjiev. *Scalable and Agile Architectures for Ebusiness.* http://research.saafe.org, July 31, 2003.

Mark Johnson. J2EE Blueprints group at Sun Microsystems, 2002.

Nicolas Guelfi, Benoît Ries, Paul Sterges. *Proceedings of the IEEE International Conference on Software-Science, Technology & Engineering*, 2003.

Emmanuel Tissandier, Christophe Jolif. *SVG support in ILOG JViews Component Suite,* 2004.

Anthony A. Aaby. *Introduction to Programming Languages*, Walla Walla College, 1996.

Margaret Thorpe and Changhai Ke. *Simple Rule Markup Language (SRML): A General XML Rule Representation for Forward-chaining Rules.* , ILOG, S.A, 2004.

N. Alex Rupp. *JSR-94 Lead, SUN Microsystems*, 2005.

Martin Gardner. *"Mathematical Games" column in Scientific American*, October 1970 issue.

Edwin Martin. *John Conway's Game of Life – Java based implementation*, 2002 http://www.bitstorm.org/gameoflife/

VITA

Narasimhan Kaliyamoorthy was born in Chennai, India. He attended University of Madras, receiving his degrees in Mathematics and Computer applications. Upon leaving graduate school, Narasimhan worked for Ramco Systems, a global vendor for enterprise software products. His duties are to develop and maintain enterprise resource planning products for global customers. After 2 years of service, Narasimhan pursued his master's degree in Software Engineering from Texas State University-San Marcos. He graduated from the graduate college at Texas State in August 2005.

Starting September 2005, Narasimhan is employed as Software Engineer at Advanced Micro Devices, a semiconductor manufacturing company in Austin, Texas.

E-mail: narsi.txstate@gmail.com

This thesis was typed by Narasimhan Kaliyamoorthy.