WHITEPAPER

BEHAVIOR-DRIVEN DEVELOPMENT AND DECISION MANAGEMENT

Deliver the right product to market faster with decision management and behavior-driven development (BDD)

"Test-driven development (TDD), behavior-driven development (BDD), and domain-driven development (DDD) comprise an approach to decision development that ensures predictable and productive processes."

JUSTIN HOLMES
ARCHITECT,
RED HAT CONSULTING

facebook.com/redhatinc
@RedHat
linkedin.com/company/red-hat

redhat.com

EXECUTIVE SUMMARY

Business experts and application developers in enterprise organizations need to be able to model, automate, measure, and improve their critical processes and policies. Red Hat® Decision Manager makes this possible with fully integrated business rules management, resource constraint optimization, and complex event processing (CEP).

Accommodating change and avoiding feature decay when designing and developing applications is necessary for aligning systems with the requirements brought on by competitive pressures, increased demands for regulatory compliance, and technological advancements. As leading companies strive to keep pace with emergent technologies, the fast and frequent release of software may be achieved using what some agilists call the "three amigos": test-driven development (TDD), behavior-driven development (BDD), and domain-driven design (DDD). These three concepts comprise an approach to decision development that ensures predictable and productive processes. And with a combination of software and professional services, enterprise organizations can help better keep pace with the demands of a rapid global marketplace.

By introducing these application life-cycle management frameworks and illustrating how they may be harnessed with Red Hat Decision Manager applications, Red Hat Consulting helps its clients realize business value from agile approaches coupled with proven architectures. Red Hat Consulting helps clients improve speed to market, reduce risk, and sustain quality.

This whitepaper offers an introduction to these approaches and explains how enterprise organizations can keep pace with the demands of our rapid global marketplace with a combination of software and professional services from Red Hat.

THE "THREE AMIGOS"

Gojko Adzik is the author of Specification By Example, a book explaining applications of the behavior-driven development methodology. Adzik introduced "the three amigos" in a presentation at the DDD eXchange Conference in 2010.[1] His thesis demonstrates the basis for, and advantages of, test-driven development. Test-driven development, when complimented by domain-driven design and behavior-driven development, allows a customer-centric approach to sustainable agile practice.

1 *Adzik, Godjco. "The three amigos." DDD eXchange 2010. slideshare.com. http://www.slideshare.net/skillsmatter/ddd-exchange-2010-gojko-adzic-on-ddd-tdd-bdd*

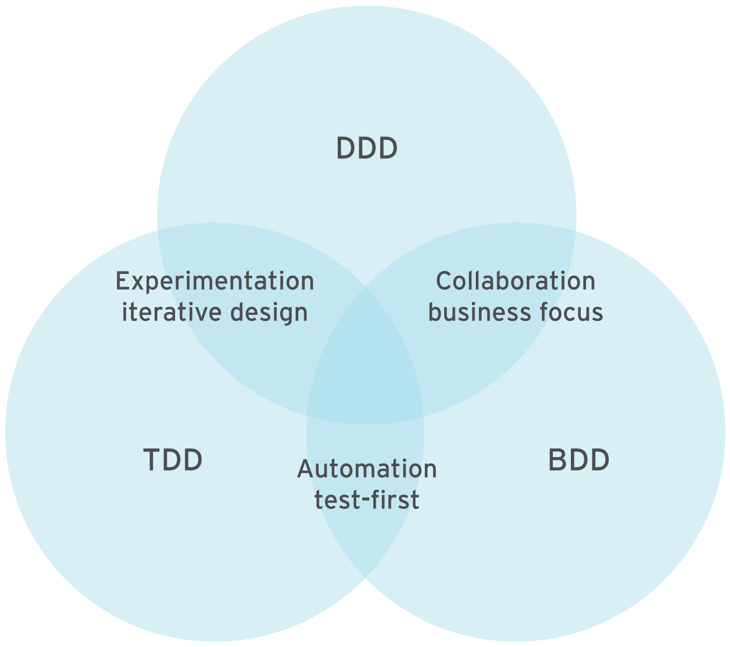

DDD
Experimentation iterative design
Collaboration business focus
TDD
Automation test-first
BDD

Figure 1: The intersection of test-driven development, domain-driven design, and behavior-driven development

TEST-DRIVEN DEVELOPMENT (TDD)

TDD is a software development process that integrates quality assurance into define-build-test teams. Known to some as "test-first" programming, TDD originated as an aspect of extreme programming (XP), an agile discipline for speeding software development. Popular agile and scrum practices now incorporate TDD into the development life cycles of many organizations. Red Hat consultants use TDD to automate unit tests and exercise knowledge engines throughout iterative development life cycles. By first writing a test and then creating modules to allow the test to pass, the development effort is driven by first codifying the requirements and then authoring the rules to meet the requirement. By addressing how to represent the business requirements first, this helps to involve business analysts up front and ensure that the decisions being modeled are going to achieve the intended outcome.

DOMAIN-DRIVEN DESIGN (DDD)

Domain-driven design (DDD) is an approach to software development that acknowledges the complexity of software development. In order to control and manage the impact of this complexity, Eric Evans, author of Domain-Driven Design, advocates the development of a domain model, which (in his words) is a model that:

"...discards the dichotomy of an analysis model and design to search out a single model that serves both purposes. Setting aside purely technical issues, each object in the design plays a conceptual role described in the model. This requires development teams to be more demanding of the chosen model since it must fulfill two quite different objectives."[2]

To this end, Evans encourages the use of a "ubiquitous language"—common vocabulary that conveys a shared understanding between subject matter experts and software developers. This language gives life to a domain model by connecting the designs of the whiteboard with the objects in an application. Domain-driven design uses a layered architecture and a set of object design patterns discussed later in this paper. Red Hat Consultants apply these techniques in a variety of enterprise development contexts to create effective decision-centric domain models. The results have been promising and provide empirical evidence of improved development and deployment of applications in complex domain contexts.

BEHAVIOR-DRIVEN DEVELOPMENT (BDD)

BDD is a software development process that combines the principles of TDD with the object-oriented analysis and design of DDD. By focusing on the creation of specific examples of domain model behavior, BDD provides developers and business analysts with a common forum to collaboratively develop "ubiquitous language." Using a variety of open source tools, Red Hat Consulting implements BDD solutions that provide automated testing and reporting using language that is a central element of both DDD and BDD. This underpinning of customer collaboration practices is central to agile frameworks and helps align delivered software with the emerging requirements of the customer.

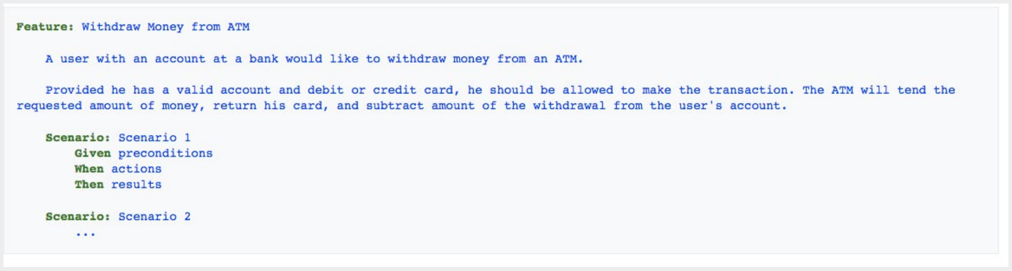
The intersection of these three processes creates an approach that facilitates empirical process control, including inspection, adaption, and transparency. By using tests to illustrate and demonstrate the empirical features of working software, stakeholders can review features with developers after each iteration. Establishing a domain vernacular that allows stakeholders and developers to innovate through business-oriented conversations improves knowledge transfer. Living documentation remains with the code to codify requirements, evolving alongside applications to ensure change is embraced rather than resisted. TDD, DDD, and BDD facilitate the inspection, adaptation, and transparency required by agile approaches. (See Figure 1.)

ITERATIVE SOFTWARE DEVELOPMENT AND TESTING USING BDD

Clients striving to keep up with the pace of change are turning to agile frameworks to organize and sustain ongoing "design-build-test teams." These teams use iterative development practices to avoid the delays associated with waterfall and big-bang development approaches. Through TDD, agile teams release high quality software achieved through rigorous unit testing and increased code coverage. These approaches are integrated into an automated build in order to practice of continuous integration. BDD takes this a step further by establishing a ubiquitous domain language in the vernacular of the customer.

2 *Evans, Eric. Domain-Driven Design. O'Reilly Media. 2003. p.49*

Through specification by example, requirements are captured in scenarios that document the success criteria found in user stories and other artifacts. These scenarios, in a "given-when-then" format, may be automated through software frameworks like Cucumber, JBehave, and xUnit. Tests are then stored and managed within source code control environments, such as Git, and provide living documentation that evolves with the accompanying applications' source code. Tests are executed using open source continuous integration tools like Jenkins. Test results may be reported and distributed to help identify and remedy defects throughout the development life cycle. This is unlike serial testing, which often doesn't occur until late in a waterfall development process.



```
Feature: Withdraw Money from ATM

    A user with an account at a bank would like to withdraw money from an ATM.

    Provided he has a valid account and debit or credit card, he should be allowed to make the transaction. The ATM will tend the
requested amount of money, return his card, and subtract amount of the withdrawal from the user's account.

    Scenario: Scenario 1
        Given preconditions
        When actions
        Then results

    Scenario: Scenario 2
        ...
```

Figure 2: An example scenario that may be automated with Cucumber using BDD[3]

By capturing requirements in a domain language familiar to business stakeholders, working software is demonstrated and reviewed in an iterative delivery process. Revisions to requirements can be captured and maintained with the application source code. This customer-driven technique yields a level of knowledge transfer and customer collaboration that allows for ongoing innovation. Work may be prioritized by business value, risk, complexity, and organizational enablement due to the close contact and ongoing interaction with subject matter experts and stakeholders.

3 *"Cucumber (software)." en.wikipedia.org, https://en.wikipedia.org/wiki/Cucumber_(software)*

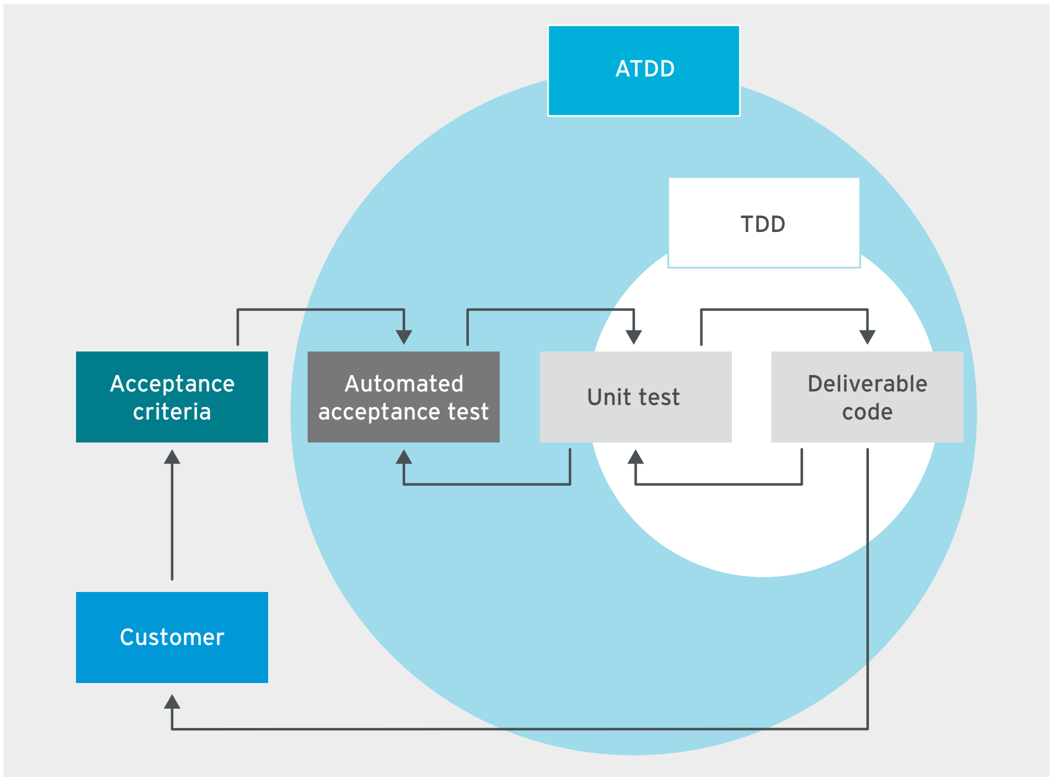


Figure 3: An iterative approach with integrated testing

DOMAIN-DRIVEN DESIGN AND THE DECISION MANAGER KNOWLEDGE BASE

The domain model developed via a DDD process represents a rigorously structured executable model of the business architecture, in which "...each object in the design plays a conceptual role described in the model."[4] In other words, a proper domain model should exhibit the following two traits:

- Subject matter experts agree that the domain model represents the knowledge needed to automate their business architecture
- Software engineers agree that the domain model alone is responsible for the business behavior exhibited by the overall system.

When viewed through this lens, a proper domain model should be considered a formal knowledge base as described by Ronald Brachman and Hector Levesque, the coauthors of Knowledge Representation and Reasoning. The domain model can be said to adhere to what is called the knowledge representation hypothesis by the philosopher Brian Smith.

4 *Tirelli, Edson, "BRMS Best and Worst Practices And Real World Examples." dmcommunity.files.wordpress.com. 2016. https://dmcommunity.files.wordpress.com/2016/01/rulesfest2011-101_edsontirelli__brmsbestandworstpracticesandrealworldexamples.pdf*

"Any mechanically embodied intelligent process will be comprised of structural ingredients that a) we as external observers naturally take to represent a propositional account of the knowledge that the overall process exhibits, and b) independent of such external semantic attribution, play a formal but causal and essential role in engendering the behaviour that manifests that knowledge."[5]

THE RED HAT APPROACH

Knowledge representation and reasoning is considered by many the core subject of artificial intelligence, and it is in this rich field of study that hybrid reasoning systems such as Red Hat Decision Manager take root. Connecting the theoretical underpinnings of both the technologies and methodologies used to deliver solutions is a fundamental component of the Red Hat Consulting approach.

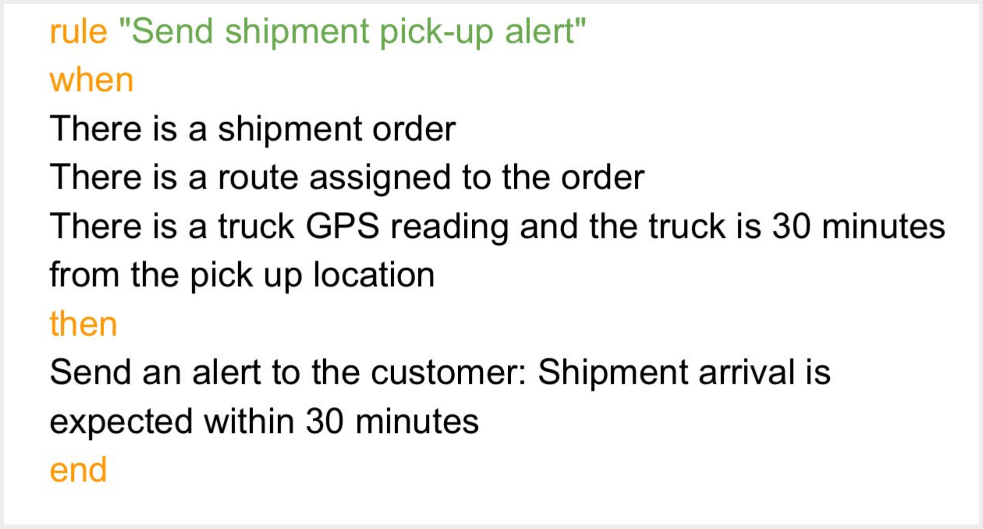
To provide software developers with building blocks to create domain models, Eric Evans offers a handful of immensely powerful object-oriented design patterns. Developers familiar with open source frameworks, such as Hibernate and Spring Data, will recognize many of these terms, as they have become common conceptual components of modern software development.

- **Entities.** Objects defined by their continuity and identity, not by the value of their attributes
- **Value objects.** Immutable objects used to describe the attributes of other objects
- **Aggregates.** Entities that manage the life cycle of the objects they own
- **Factories.** Objects that handle the creation of complex entities and aggregates
- **Repositories.** Objects that abstract the access of pre-existing aggregates and are interfaced with the application
- **Services.** Objects that encapsulate complex logic that doesn't naturally belong in a single object

Of particular interest to Decision Manager applications is the concept of the domain service. At the core of Decision Manager is an implementation of sophisticated pattern-matching algorithms which support complex behavior expressed by the confluence of several objects. Take for example the rule found in Figure 4 below.

To notify a customer that their delivery is 30 minutes away, several objects in the model must be inspected and each one must conform to a certain pattern. However, implementing this behavior in different objects would unnaturally fracture the single concept of alerting a customer of their impending delivery. By moving this concept to a domain service implemented with Decision Manager, complex relationships between numerous objects can be captured in a single location within the model using a technology optimized for the behavior.

5 *Brachman, Ronald and Levesque, Hector. Knowledge Representation and Reasoning. O'Reilly Media. 2004. pp. 5-6*



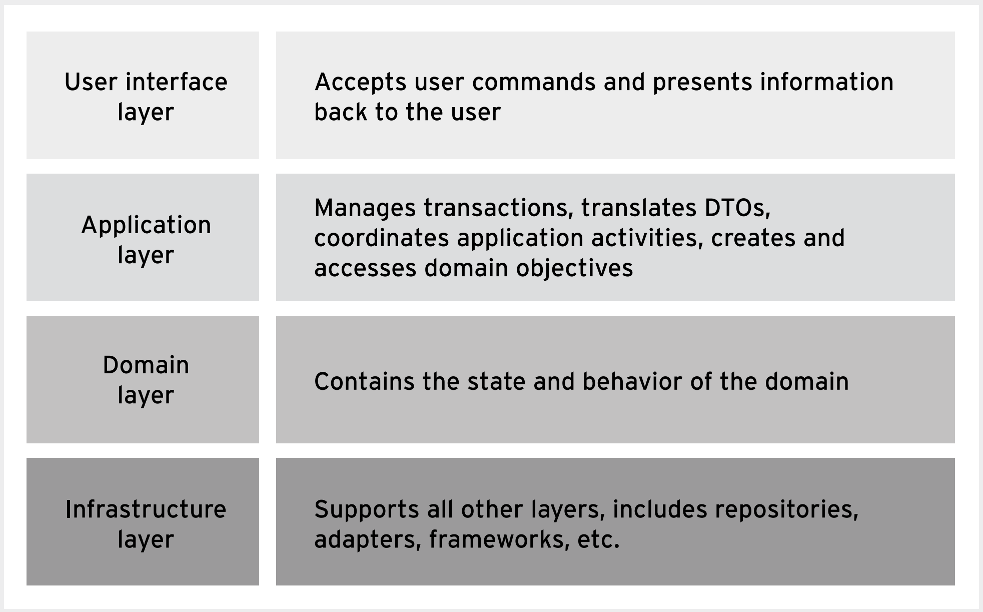
```
rule "Send shipment pick-up alert"
when
There is a shipment order
There is a route assigned to the order
There is a truck GPS reading and the truck is 30 minutes
from the pick up location
then
Send an alert to the customer: Shipment arrival is
expected within 30 minutes
end
```

Figure 4: A sample of rules used within a knowledge engine

INCREASING SPEED TO MARKET WITH SIMPLIFIED ARCHITECTURE DESIGN AND DECISION ENGINES

Simplifying architecture is a way to decrease the time it takes to initially develop an application as well as reduce the time and cost to maintain it. Conventional model view controller (MVC) application architectures have traditionally been employed for online applications that do not require enterprise scalability. The services associated with data access, RESTful interfaces, and multitenant environments are critical to larger enterprise applications.

This hybrid architecture looks to create a layered architecture that aligns with microservices or service-oriented-architecture (SOA). DDD presents an architecture that puts the domain at the center of these common architectures. It takes advantage of Decision Manager's unique ability to be embedded as Java™ archive (JAR) files in a simple Java application deployed as a microservice or web application archives (WAR).



User interface layer	Accepts user commands and presents information back to the user
Application layer	Manages transactions, translates DTOs, coordinates application activities, creates and accesses domain objectives
Domain layer	Contains the state and behavior of the domain
Infrastructure layer	Supports all other layers, includes repositories, adapters, frameworks, etc.

Figure 5: A layered architecture based on a domain model approach

USER INTERFACE LAYER

The user interface layer makes up the view component of the layered architecture, which can be implemented with any modern JavaScript framework such as React, Angular, or Backbone. This layer is just focused on how information is displayed to users and how they can interact with it.

APPLICATION LAYER

The application layer makes up the controller components of a layered architecture. The web service API, web application controllers, integration technologies, and batch frameworks might be included at this level. This layer defines the purpose of the application and also the various services to be exposed across the enterprise to promote re-use and utilization of standardized protocols such as REST.

DOMAIN LAYER

The domain layer makes up the model component of a layered architecture and is home to the domain model. This layer should generally be developed as its own JAR and can take advantage of Decision Manager's unique capability to be embedded as a small subset of Java dependencies or use Decision Manager's out-of-the-box KIE server, which provides a simple way to expose decisions as REST or JMS services.

INFRASTRUCTURE LAYER

The infrastructure layer provides the technical details of connections to external data sources. Most enterprise-class systems require connections to multiple data sources, so it can be helpful to provide repository interfaces in the domain layer and repository implementations in the infrastructure layer. Object relational model (ORM) technologies such as Hibernate, a community project from Red Hat, can be used to model data sources as objects and ensure loose coupling to specific database technologies.

This layered approach simplifies the architecture of the applications and builds on the simplicity and elegance of the domain model, facilitated through the DDD mapping mentioned previously. Instead of complex systems with too many moving parts, Decision Manager's engine may be deployed with only Plain Old Java Objects (POJOs) and the common elements of a J2EE application.

BEHAVIOR-DRIVEN DEVELOPMENT IN AN ENTERPRISE AGILE ENVIRONMENT

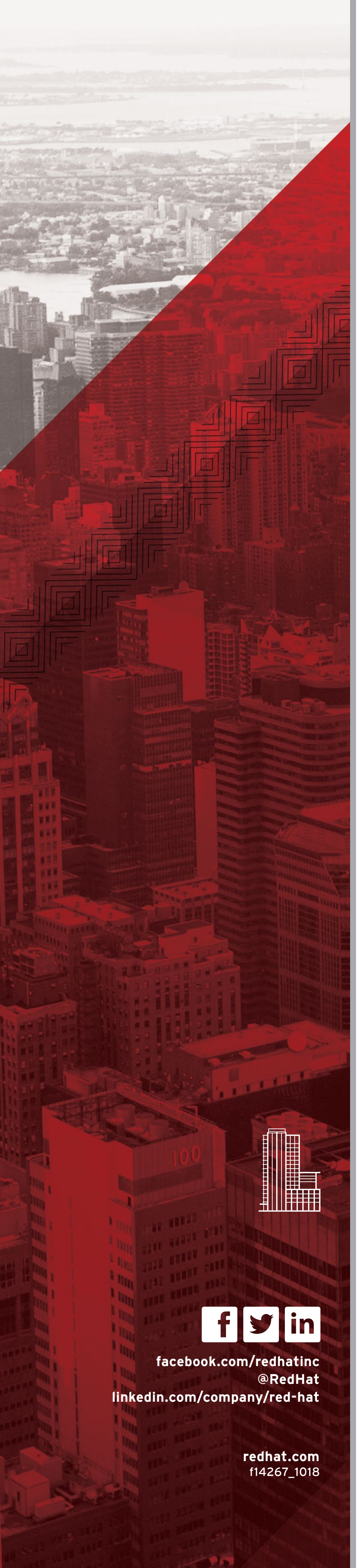
The simplified architectures in this paper allow the crafting of a knowledge engine within the domain layer to speed development and implement object-oriented approaches. Since agile development teams typically have only 5 to 9 members, there are usually multiple teams operating in parallel. The delivery of each team's component at the end of a potentially shippable increment (PSI) may by synchronized through lightweight agile frameworks such as the scaled agile framework (SAFe).

If rules engine development is used as a kanban swimlane, a team devoted to its design, development, and testing might be organized within the overall program. This team may operate autonomously and independently through the use of specification by example and BDD tools, such as Cucumber or JBehave.

Instead of having to wait for components from other teams to exercise the knowledge sessions, dependencies may be reduced and blockers or impediments may be avoided through the BDD approach. Through the repository interface, test implementations of a repository can be written to provide data to a knowledge session using the "given" statements that set up the context of a scenario. "When" statements can be used to specify a rule flow group or agenda group to exercise based on a certain event. This connects the BDD scenarios to the feature Decision Manager implements and helps manage large knowledge bases. "Then" statements can be implemented using Drools queries to request information from the knowledge session and Assert calls to verify the state of said information. An entire rules application can use the automated scenario to function without the user interface or access to external data sources.

When this approach is employed, productivity increases through the elimination of dependencies. When the rules engines are integrated with modules and components produced by other teams and, after integration, the anticipated results are not achieved, then the BDD tests may be used to isolate integration errors and quickly remedy defects at the time of integration.

Additionally, in agile environments, the BDD approach aligns stakeholders with the decision engine far in advance of integration with other components. Errors in specification, improper representation of requirements, elaboration of outcomes, and comprehensive review of use cases may all occur early in the application life cycle. Innovation often occurs through these review processes because as initial functionality and use cases are demonstrated, customer requirements may be further refined. Since the domain model is mapped through DDD, changes in the rules engine are easily incorporated into the model so that affected components are also updated.



CONCLUSION

Agile organizations must develop applications that yield high returns at a reduced cost. This means these organizations must find ways to support more productive teams. Organizations recognize that quality software has to be testable and supportable in enterprise-level production environments. Success requires accurate, understandable requirements and an integrated approach to quality assurance (QA).

Red Hat Consulting uses BDD and Decision Manager to improve our clients' success rates. Our technologies and methods allow companies to close the gap between business analysts (BAs), developers, and QA analysts by having them work in close alignment. We help our clients take advantage of the complementary skills found within cross-functional design-build-test teams. Our results are high-quality applications that deliver business value quickly at reduced cost.

facebook.com/redhatinc
@RedHat
linkedin.com/company/red-hat

redhat.com
f14267_1018

NORTH AMERICA	EUROPE, MIDDLE EAST, AND AFRICA	ASIA PACIFIC	LATIN AMERICA
1 888 REDHAT1	00800 7334 2835	+65 6490 4200	+54 11 4329 7300
	europe@redhat.com	apac@redhat.com	info-latam@redhat.com