The Perfect Couple: Domain Models & Behavior-Driven Development

Dan Elbaum
dan@danelbaum.com

Carlin Scott
carlin.q.scott@gmail.com

Abstract

Behavior-Driven Development entails a unified set of practices for expressing requirements in business-readable language, and binding them to automated test cases. Unfortunately, the Agile community has been so focused on the test automation aspects of the methodology that they have overlooked the challenges involved in the initial step of BDD: expressing requirements in business-readable language.

While business-readable language simplifies communication with business stakeholders, it also complicates the process of specifying requirements for the implementation team. The informality of business language makes it difficult to write requirements that are accurate, consistent, and unambiguous. How can practitioners specify technical requirements with business-readable language?

This paper explores how domain models help to reconcile the looseness of business language with the exacting requirements of engineering. By systematically representing key concepts in the business domain with visual diagrams and operational definitions, we can construct a domain model that allows us to unambiguously specify implementation requirements in business-readable language.

Biography

Dan Elbaum is a Senior IT Business Analyst at Con-Way Enterprise Services in Portland, Oregon. Over the past 6 years, he has focused on the development of web-based information systems. Dan holds an M.S. in Management from the University of Florida and is an IEEE Certified Software Development Professional, as well as an INCOSE Associate Systems Engineering Professional.

Carlin Scott is a Software Development Engineer in Test at Intel Corporation and previously worked for Hewlett-Packard. He has worked in QA as an SDET for the past 4 years and holds a BS in Computer Engineering from Oregon State University.

1 Introduction & Motivation

Requirements hand-off from business stakeholders to implementation teams is a notoriously weak point in commercial software development due to overlapping technical and social factors. At the technical level, the intrinsic complexity of software makes it difficult to conceptualize and communicate requirements. That technical layer of difficulty is compounded by the organizational context of software development, where requirements must be coordinated with multiple stakeholders who come from different disciplinary backgrounds, think about software at different levels of abstraction, and express their ideas in different terminology. Commentators often suggest that this “communication gap” is the main source of our requirements problems.

Behavior-Driven Development aspires to bridge the communication gap by framing requirements at a level of abstraction that is mutually understandable to business and technical stakeholders alike. Unlike traditional imperative specifications that describe how code operates procedurally, BDD specifications describe an application’s intended behavior from the perspective of a domain expert, using domain-specific terminology (also referred to as “ubiquitous language”). There is wide consensus that the vocabulary of a ubiquitous language should be derived from a model of the application domain[1], yet few practitioners bother to create that domain model. This is problematic. To skip domain modeling is to miss the point of domain-specific language: it is intended to shift our focus to a different level of abstraction, not to relax our level of precision. Domain modeling deserves broader consideration from Agile practitioners because it is a powerful technique that allows us to structure our conceptions of an application domain, disambiguate ubiquitous language, and ultimately bridge the communication gap.

The purpose of this paper is to explain how domain models augment ubiquitous language, and demonstrate a general method for how to apply them in practice. The goal of this method is to enable Agile teams to comprehend specifications faster and more accurately. The method uses visual diagrams and operational definitions to illustrate and define domain-specific concepts. The result is a richer ubiquitous language that allows us to communicate application requirements with greater clarity and consistency.

This paper is divided into five sections. The first section describes how BDD carries requirements throughout the application development lifecycle. Section two evaluates the strengths and weaknesses of ubiquitous language in different areas of requirements engineering. Section three explains how domain models enhance ubiquitous language. Section four presents our method for building a domain model, deriving a ubiquitous language from it, and using those terms as a controlled vocabulary to write scenarios. Section five addresses anticipated criticisms of domain modeling as anti-Agile.

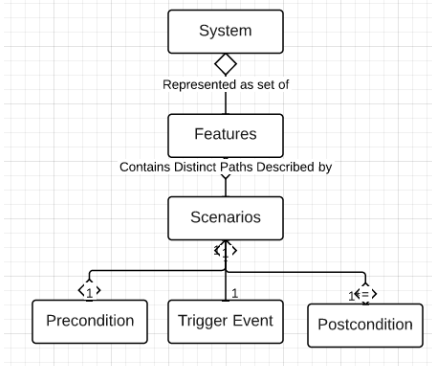
2 The Life of a Requirement in Behavior-Driven Development

Behavior-Driven Development is an emerging agile methodology that carries requirements throughout the entire value stream, from analysis, through to development and testing. The methodology was first introduced by Dan North, who defines it to mean "implementing an application by describing its behaviour from the perspective of its stakeholders”.[2] In what follows below, we provide an overview of how requirements are formulated and carried throughout the development lifecycle in BDD.

Feature-centric. As a starting point, BDD organizes development by partitioning an application into modular features that can be independently built. Requirements are organized around individual features.

Use Case Scenarios. Each feature’s intended behavior is described with a type of use case scenario called as a Given-When-Then scenario, a one sentence narrative that describes the set of possible interactions that a user can have with the feature. The general format is: Given the application’s initial state, When a trigger event occurs, Then the application should exhibit some behavior in response.

Below are representations of use case scenarios. The meta-model is on the left, the traditional tabular format is in the middle, and the BDD format is on the right.



Overview	
Title	[Title of the basic flow use case]
Description	[Short description of the basic flow]
Actors and Interfaces	[Identifies the Actors and Interfaces to components and services that participate in the use case]
Initial Status and Preconditions	[A pre-condition (of a use case) is the state of the system that must be present prior to a use case being performed]
Basic Flow	
STEP 1: …	
STEP 2: …	
Post Condition	
[A post-condition (of a use case) is a list of possible states the system can be in immediately after a use case has finished]	
Alternative Flow(s)	
[Alternative flows are described here if needed]	

Given Precondition

When Trigger Event

Then Postcondition

Ubiquitous Language. We write use case scenarios in terms of domain-specific abstractions elicited from domain experts. Collectively, these terms constitute a "Ubiquitous Language". Framing requirements at this level of abstraction enables domain experts to participate in requirements validation, and confirm that a specification satisfies their requirements. The table below illustrates how different levels of abstraction could be used to frame a scenario where a user logs into Netflix and sees on their landing page a set of recommended movies.

Level of Abstraction	Use Case Scenario
Domain	Given that I am a Netflix Member When I submit correct login credentials Then I should see my Movie Recommendations
User Interface	Given that I am on the Netflix login page And I have typed my email address into the username field And I have typed my password into the password field When I click on the login button Then I should see my homepage
Application	Given that the text input posted contains no dangerous characters When the user's email address matches a row in the customer table And they successfully authenticate Then their session should be initiated And the movies in the recommendation table ranked 1-8 should be retrieved

In the example above, at the domain-specific level of abstraction, "Movie Recommendations" would be considered a domain-specific concept. Suppose it refers to the output of a learning algorithm which aggregates members' viewing history to construct a model of their preferences and recommends films they are likely to enjoy. That term would be a formal construct within the application domain. In BDD, all participants in system development are supposed to adopt the same terminology for referring to such abstractions when communicating with one another about the application, making a single set of terminology effectively "ubiquitous". The hypothesis behind this approach is that cross-functional teams can communicate more effectively when they speak a common language.

Use-Case Testing. To verify that a feature satisfies its requirements, we implement each of its Given-When-Then scenarios as executable tests. Each step in a GWT scenario is parsed into an executable block of code that directly exercises the system-under-test. When run, the Given step sets up the precondition, the When step executes the trigger event, and the Then step checks to see if the post-condition has occurred. If so, then the entire test passes. This testing technique is known as use-case testing because the tests are intended to simulate how a real user would interact with the system.

Because GWT scenarios serve a dual-purpose in BDD as both a functional requirements specification and a test case, these are often referred to as “executable specifications.”

Test-Driven Development: These “executable specifications” are the tests that drive BDD’s test-driven development lifecycle. The focus of development is to code all features so that they pass all of their Given-When-Then tests. Once those tests pass, development of a feature is considered complete. The philosophy behind this approach is that writing tests before development helps to build-in quality from the outset, which is more effective than “inspecting it in” through post-development quality assurance activities.

Thus, every step in the development process is glued together by Given-When-Then scenarios written in domain-specific terminology. Domain-specific terms are elicited from stakeholders, and used to write Given-When-Then scenarios that specify the application’s intended behavior. Then, these scenarios are translated into test cases which guide implementation in a test-driven lifecycle.

3 Ubiquitous Language: Strengths & Weaknesses

We have seen that domain-specific scenarios are the singular artifact that BDD uses to carry requirements throughout the entire development lifecycle. A domain-specific level of abstraction is used to elicit requirements from upstream business stakeholders, and to specify and verify requirements with the downstream implementation team. Unfortunately, ubiquitous language is not equally effective for all of these requirements engineering activities. Ubiquitous language is convenient for requirements elicitation because it is informal. However, this informality makes it less suitable for requirements analysis and specification. In this section, we review ubiquitous language’s strengths and weaknesses for different requirements engineering activities.

3.1 Strengths of Ubiquitous Language: Requirements Elicitation & Validation

The primary strength of ubiquitous language is that it simplifies requirements elicitation and validation. It provides a common language that enables business and technical stakeholders to communicate more effectively. Speaking in terms of the application domain enables stakeholders to focus on their business requirements, and avoid conflating them with implementation details. The black-box perspective of ubiquitous language is beneficial in that it helps to separate “the specification of requirements from design, simplifying the model and making the requirements model easier to construct, review, and formally analyze.”

A secondary strength of ubiquitous language is its ability to provide the development teams with cognitive benefits that enable them to deliver better-working software, faster. In using ubiquitous language to collaborate with businesspeople, development teams gain insight into the underlying purpose that an application is intended to fulfill, and that helps them to comprehend business requirements. Comprehending complex requirements specifications has been shown to depend crucially on understanding the rationale behind them.[3] Nancy Leveson, a software researcher specializing in software specifications, has extensively documented the phenomena whereby software engineers have difficulty comprehending complex specifications if the underlying rationale and intent is not explained. Another principle which would suggest that ubiquitous language may have a beneficial psychological impact on implementation teams’ understanding of an application is the Sapir-Whorf hypothesis. The Sapir-Whorf hypothesis posits that the language that we speak influences the way that we conceptualize the world. As applied to software development, adopting the terminology of the business domain may enable the development team to better understand the perspective of business stakeholders.

3.2 Weaknesses of Ubiquitous Language: Requirements Analysis & Specification

Ubiquitous language lacks the degree of exactness required to analyze requirements and specify scenarios clearly. Because scenarios directly influence implementation in BDD, precision is critical, and errors are costly. In contrast to user stories, which can be written loosely because they only convey the general purpose of a feature, BDD scenarios must be expressed more precisely because they prescribe how software is intended to function, and serve as the basis for implementation. Heightened attention to accuracy is necessary, because errors in ubiquitous language can directly introduce defects into code and, as everyone knows, defects introduced in the requirements stage are often the most difficult and expensive to correct. Consider, for example, how inexact usage of ubiquitous language might impact the development of a college course registration application. Suppose that we're building a test suite which contains scenarios that refer to the instructor of a course in various ways such as a lecturer, professor, and teacher. It may be unclear whether these terms are synonyms, or different domain-level concepts that should be implemented as separate classes. If such a misunderstanding weren't detected early upstream, it would directly introduce cascading defects into the product. Such a possibility is not at all unlikely because technologists often perceive non-technical sounding business terms as fluffy, vague, and not worth paying attention to.

Another disadvantage of ubiquitous language is that it cannot express all aspects of a domain that must be understood to analyze elicited requirements, and form a specification sufficiently detailed to serve as the basis for development. Much of the information that software engineers need to build a system is difficult to convey in non-technical terms. If we use only business-readable language in requirements specifications, how are software engineers supposed to get the information they need? This is a problem which must be resolved in order for domain-specific language to effectively carry requirements throughout specification and verification.

4 Domain Models Improve Analysis & Specification

Domain models can ameliorate the shortcomings of domain-specific language that we have just described in regard to requirements analysis and specification. In the context of software development, a domain model is an abstract representation of the elements of the business domain that the software-under-development is intended to embody. It "collects all the information about the problem domain that must be known and understood to allow capturing requirements for the system, specifying them, implementing and verifying the system."[4]

The general purpose of domain models is to organize domain-specific concepts into a coherent representation. This allows us to make technical inferences about the domain without creating a separate set of "technical specifications". Structuring information about the domain into a formal or semi-formal model allows us to analyze requirements for completeness, consistency, and correctness, as well as to communicate requirements to the implementation team more effectively. In this regard, domain modeling aligns with our overarching goal in requirements specification: to support the implementation team's ability to understand and reason about the application domain effectively and efficiently. In this section, we provide an overview of the benefits conferred by the two key parts of our domain models, visual diagrams and operational definitions.

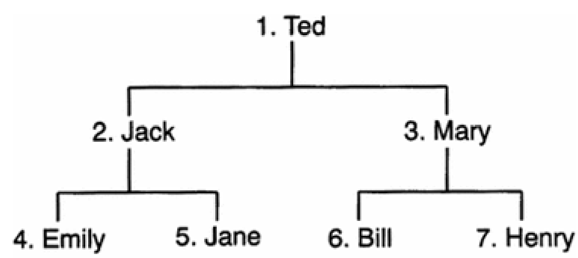
4.1 Visual Diagrams Provide Cognitive Benefits

Visual diagrams are useful in domain modeling because they present important cognitive benefits that enable people to learn certain types of information more effectively. Text and pictures can represent equivalent information, but pictures can be processed faster in many instances. While textual requirements are useful in some phases of software development, visual representations are an effective tool for aiding developers' understanding of complex specifications. A large body of cognitive psychology research has proven that supplementing or replacing text-based representations of information with visual diagrams can help learners to process information more effectively, with less time and mental effort.[5] This directly supports Agile’s aim to deliver working software rapidly. It attacks the limiting factor on the speed of software delivery: the rate at which developers can process information about the system-to-be.

One important psychological advantage that graphical diagrams confer is known as the “gestalt effect”. The gestalt effect states that humans perceive objects by trying to grasp them as a whole. When equivalent information is represented in both text and diagrams, people can gain a high-level overview faster from the diagram. Depicting a high-level overview of a system helps development teams to contextualize and comprehend its lower-level details.

Another important psychological advantage of visual diagrams is that they support alternate search strategies, or ways of finding information. Viewers can zoom into or out of different sections in a diagram, or focus on the relationship between two parts of a diagram. Text provides must generally be read in a linear order. Visually displayed of information can be perceived in a single glance.

The below diagram of a family tree is from a classic cognitive psychology paper that describes the advantages of visual representations of information over text.[6] Both the text and the diagram contain equivalent information. Try using the text to infer cousin and sibling relationships, and then try it again using the diagram. Which is easier to use?

TEXT-BASED REPRESENTATION	**DIAGRAMMATIC REPRESENTATION**
1. Ted is Jack’s parent. 2. Ted is Mary’s Parent. 3. Jack is Emily’s parent. 4. Jack is Jane’s Parent 5. Mary is Bill’s Parent 6. Mary is Henry’s Parent	

In section 5, we will review some diagrammatic notation which can be useful for domain modeling.

4.2 Operational Definitions Disambiguate Domain-Specific Terms

Operational definitions are the second key part of domain models. Domain-specific concepts must be precise in order to be useful in specifying an application’s behavior. In order to get the necessary level of precision, we must formalize the semantics of domain-specific terms by operationally defining each one. The collection of operational definitions serves as a controlled vocabulary which we use to describe the application’s intended behavior in Given-When-Then scenarios. This technique, known as Vocabulary Management, is a way to “improve the effectiveness of information storage and retrieval systems. The primary purpose of vocabulary control is to achieve consistency in the description of content objects and to facilitate retrieval”.[7] These definitions are crucial because they disambiguate domain terminology and reduce opportunities for misunderstanding. Teams working on complex projects may need to re-review the specifications multiple times throughout the development cycle in order to refresh their memory or clarify a term. A glossary of operational definitions serves as an authoritative source that people can refer back to at any time.

5 Building Domain Models for Usage in BDD

This section describes our general method for creating ubiquitous language from domain models. The approach consists of three steps: 1) graphically model the business domain, 2) operationally define its constituent elements, and 3) use those definitions as a controlled vocabulary for writing BDD Scenarios.

5.1 Step 1: Build a Visual Diagram of the Domain

The first step in our approach is to visually diagram two key aspects of the business domain: static structure, and dynamic behavior. The static structure of a domain is a point-in-time snapshot of the key elements in the domain. Often referred to as the 4 W's of Who, What, Where, When, this is simply the key people, things, places, and times involved in the domain. The dynamic behavior is the set of processes, events or transactions that occur over time which effect or involve its static parts listed above.

Our recommendation is to start by modeling the static aspects before the dynamic aspects. The logic behind this decision can be illustrated through analogy. Just as the motion of a mechanical system is easier to understand after examining the system at rest and seeing how its parts fit together, the behavior of an information system is also easier to learn after we understand its structure.

5.1.1 Capturing Key Domain Concepts in a Conceptual Data Model

In developing information systems, as most of us in commercial application development are, the static elements of the domain are represented within the application as persistently stored data. The field of information science has already developed mature data modeling techniques that are useful not only for implementing databases, but also for illustrating the structural organization of abstract information. That is our purpose here. We will depict the static aspects of the application domain using a "conceptual data model". The Data Management Body of Knowledge defines a conceptual data model to mean "a visual, high-level perspective on a subject area of importance to the business. It contains only the basic and critical business entities within a given realm and function, with a description of each entity and the relationships between entities. Conceptual data models are intended to convey a conceptual overview of domain concepts, not to specify the implementation of a database. We omit implementation-related details that are not relevant to a conceptual overview. For example, if our diagram of choice for a conceptual data model were an entity-relationship diagram, we would not normalize the tables or enforce referential integrity.

One proposed approach to creating a conceptual data model within the BDD context is to start with an unstructured "mind map" of key domain-specific terms, and transform it into a UML class diagram.[8] Whether the notation of choice is a class diagram or an extended entity-relationship diagram, the following concepts can help to organize the structure of domain concepts.

5.1.1.1 Entities & Attributes

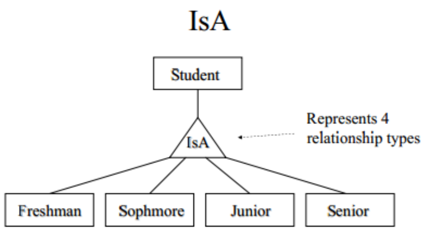
At the most basic level, our conceptual data model must capture the key people, places, and things as atomic entities, and indicate their key properties as attributes.

5.1.1.2 Associations & Their Multiplicities

The relationships between the entities identified above should be noted, and characterized in terms of their "multiplicity". Multiplicity is the number of instances of some entity that can exist for each instance of a related entity. To illustrate the importance of multiplicities, consider the following issue that could arise in the context of a college class registration application. Suppose that a lazy student attempts to enroll in two classes held during the same time period because he plans to skip all lectures anyways, and only come to class on exam days. Whether the student is permitted to double-enroll for a timeslot is not merely an implementation detail, it's a business rule. This information can be concisely expressed using the notation for multiplicities.

5.1.1.3 Type Hierarchies

Type hierarchies are a generic type of relationship between entities that we can express precisely with diagrammatic notation. Identifying subtypes of a superclass, or supertypes of a subclass, is useful for illustrating classification-related aspects of information organization. The following diagram contains an example of a type hierarchy which illustrates the four subtypes of high school student: Freshman, Sophomore, Junior, Senior.



5.1.1.4 Aggregation

Aggregation is another useful concept for illustrating the organization of information. For example, suppose we're looking at population by geographical areas. The population of a continent is the aggregated population of the countries it contains, which is the aggregate of its state populations, etc.

5.1.1.5 Composition

Composition is another generic concept that illustrates how a higher-level object emerges from the composition of its constituent parts, i.e., humans are more than the sum of their anatomical parts. UML's class diagrams provide support for expressing this type of relationship.

5.1.2 Diagramming the Dynamic Processes

Next, we must identify the processes that the application supports, such as ordering a pizza or executing financial transactions. BDD conceptualizes processes as state machines, where a trigger event causes the system to transition from an initial state to a final state. Along these lines, BDD scenarios can be represented as a graph, where the initial state is a node, actions required to transition to another state is an edge, and the final state is another node. Each scenario specifies a possible state transition. There has been some work published that provides guidance on how BDD scenarios map to constructs in Business Process Modeling Language.[9] We provide a graphical depiction of a simple BDD scenario related to grocery store checkout at the end of this section.

5.2 Step Two: Create a Controlled Vocabulary of Operational Definitions

The second step in our approach is to label and operationally define key elements of the domain. An operational definition is “a definition that identifies a concept as a variable that can be used in a hypothesis”.[10] These terms can be thought of as operational definitions in the sense that the BDD scenarios are testable hypotheses about how the system should behave. Together, these terms serve as a controlled vocabulary for writing BDD statements.

5.3 Step Three: Apply the Controlled Vocabulary to Write Specifications

The idea here is to use the controlled vocabulary derived from our domain model to write our Given-When-Then scenarios. Given that this specifies exact terminology for key roles, objects, transactions, etc, we can express most interactions with the system these terms. Consistent terminology here will enable the software delivery team to look up the definition, or refer back to the graphical models to refresh their memory. Below we provide a sample scenario for a section of a feature involving grocery store self check-out. It includes a process diagram and operational definitions for the relevant ubiquitous language.

Operational Definitions

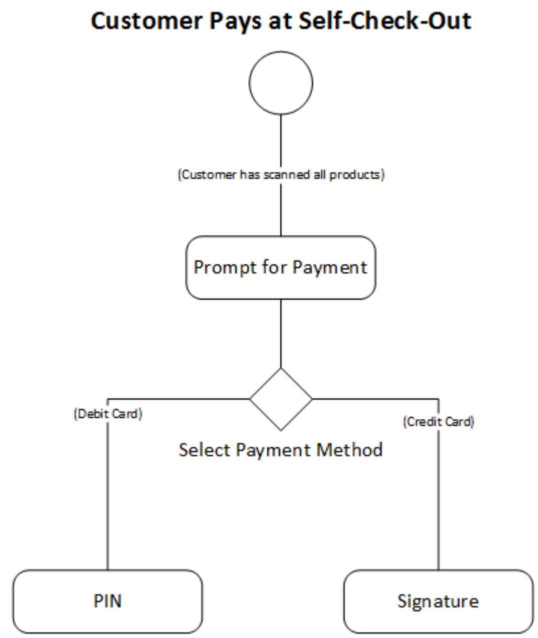
Customer – Person Purchasing of Product(s).

Product – Store inventory item available for purchase

Form of Authorization – Information Required to Complete Purchase with non-cash payment method.

```
Feature: Self-Checkout

 Scenario: Payment Authorization

 Given Customer has been prompted for payment

 When Customer pays with <Payment Method>

 Then Customer should be prompted for <Form of Authorization>
```

Payment Method	Form of Authorization
Debit Card	PIN
Credit Card	Signature



6 Does Domain Modeling Violate Agile Principles?

The approach suggested in this paper may meet some resistance from practitioners who perceive domain modeling as anti-Agile. Does domain modeling enhance or detract from the fast delivery of working software?

As a starting point, we must note the obvious fact that a domain model takes time to create. Thus, the domain model will delay delivery unless it accelerates other activities in the development process. Luckily, there is reason to believe that domain models may actually accelerate delivery. Domain models could allow the team to understand the requirements faster, and more accurately. Learning the specification faster produces a small gain in speed. The major gain in delivery speed would be related to understanding the specification more accurately. This would enable the team to correctly implement quality code from the outset, reducing the need for re-work downstream.

We suspect that the benefit of domain models is directly proportional to the complexity of the application at hand. As the application domain grows in complexity and becomes less intuitive, the quality of specifications plays a larger role in enabling a quality implementation. As a result, domain modeling may not be a valuable technique to apply to simple applications. However, we believe it's an extremely worthwhile approach for more complex domains such as health informatics, financial systems, or supply chain management.

A secondary anticipated criticism of domain modeling is its alignment with Agile's aim to reduce comprehensive documentation. Domain modeling is not a comprehensive form of documentation. A domain model can be continuously iterated to provide just-in-time detail, and it does not require a "big-design up front" approach where the complete application needs to be modeled in its entirety before development begins.

Given that domain modeling employs lightweight models to accelerate delivery of working software, we argue that it advances the Agile principles.

7 Conclusion

Domain models enable better analysis and specification of requirements written in domain-specific terms. Constructing a domain model helps us to decompose stakeholders' descriptions of the domain into a set of primitive semantic constructs (the vocabulary of the ubiquitous language) that we can then combine to compose systematic descriptions of how the domain behaves (using Given-When-Then scenarios). Domain modeling also precludes the need for separate "technical specifications" to some extent by illustrating the structural organization of domain-level abstractions with diagrammatic notation. These diagrams exploit powerful psychological principles to enable people to process specifications more effectively. The approach to domain modeling presented in this paper is intended to serve as a basic explanation of a general technique. Our hope is that this paper has made readers more aware of the importance of domain modeling to BDD.

8 References

[1] Solís, Carlos and Wang, Xiaofeng. 2011. "A Study of the Characteristics of Behaviour-Driven Development." *Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications.*

[2] North, Dan 2006. "Introducing Behaviour-Driven Development" http://dannorth.net/introducingbdd. (Accessed July 1st, 2013)

[3] Leveson, Nancy G. 2000. "Intent Specifications: An Approach to Building Human-Centered Specifications", *IEEE Transactions on Software Engineering*.

[4] Broy, Manfred. 2013. "Domain Modeling and Domain Engineering: Key Tasks in Requirements Engineering". *Perspectives on the Future of Software Engineering*.

[5] Larkin, Jill and Simon, Herbert. 1987. "Why a Diagram is (Sometimes) Worth a Thousand Words", *Cognitive Science* 11, 65-99.

[6] Winn, William, Li, Tian-Zhu,Schill, Donna. Diagrams as Aids to Problem Solving: Their Role in Facilitating Search and Computation, *Educational Technology Research and Development*. 1991, Volume 39, Issue 1, pp 17-29

[7] ANSI/NISO Z.39.19.2005. "Guidelines for Construction Format, Mgmt of Monolingual Controlled Vocabularies"

[8] Wanderley, Fernando, and Silva De Silveria, Denis. 2012. "A Framework to Diminish the Gap between the Business Specialist and the Software Designer." *Eighth International Conference on the Quality of Information and Communications Technology.*

[9] Carvalho, Silva, and Manhaes. "Mapping Business Process Modeling Constructs to Behavior Driven Development Ubiquitous Language." (arXiv:1006.4892)

[10] Svenonius, Elaine. 2000. *The Intellectual Foundation of Information Organization.* Cambridge, MA: MIT Press