ACSIJ Advances in Computer Science: an International Journal, Vol. 5, Issue 2, No.20 , March 2016
ISSN : 2322-5157
www.ACSIJ.org

ACSIJ
WWW.ACSIJ.ORG

Towards An Algorithms Ontology Cluster: for Modular Code Reuse and Polyglot Programming

Karabo Selaolo[1] and Hlomani Hlomani[2]

[1] Computer Science Department, Botswana International University of Science and Technology (BIUST)
Palapye, Central District, Botswana
karabo.selaolo@studentmail.biust.ac.bw

[2] Computer Science Department, Botswana International University of Science and Technology (BIUST)
Palapye, Central District, Botswana
hlomanihb@biust.ac.bw

Abstract

Code reuse is rarely practiced and polyglot programming an informal discipline. This paper proposes the use of ontologies in tackling these issues, seeing as ontologies inherently encourage reuse and are often used as bridges between languages. This paper also reviews existing research and literature in the fields of ontologies in software engineering, code reuse and polyglot programming.

Keywords: *Code Reuse; Ontologies; Software Engineering; Polyglot Programming*

1. Introduction

As budding programmers and software developers we are often encouraged to reuse our code, some of us are even taught mantras like DRY (Don't Repeat Yourself) [1]. But more often than not we start coding from scratch, without even a second thought as to whether we're repeating ourselves. Existing code reuse tools do exist but are often poorly documented and unintuitive to use [2]. Ontologies, if applied correctly to the field of code reuse, have the potential to resolve these issues and hopefully rope developers back into practicing DRY.

Few, outside of those specializing in similar fields, have ever heard of polyglot programming. Some might have heard of it under different titles: multi-language or multi-paradigm software development [3]. But interestingly enough, most computer scientists and software engineers have practiced polyglot programming. Here lies the biggest challenge facing polyglot programming, anonymity. As a formal discipline it has yet to take off, even though it is widely practiced, especially in web development. Formal polyglot programming needs a push, and we believe ontologies could be the best tools for the job.

Ontologies are often used as tools for communication [4], bringing together knowledge for a common understanding. These features make them ideal for creating a hub of multiple programming languages, where a central knowledge base keeping track of the relationships between the languages would be essential.

This paper reviews existing research in the fields of ontologies in code reuse, polyglot programming and software engineering and then lays down the foundation for and proposes three ontologies that could be used to encourage code reuse and formalize polyglot programming.

2. Background

2.1 Code Reuse

Code reuse is the activity or practice of reusing existing code [5]. There are many ways to reuse code, ranging from the simple, but popular, copy-and-paste to buying off-the-shelf software from a vendor and customizing it to specification. This research however is not concerned with these types of code reuse but the reuse of codified algorithms; codified algorithms being methods, functions, subroutines, subprograms or code blocks, depending on what language one's using.

Reuse of codified algorithms is therefore only achievable, mainly, through user built software libraries and frameworks. Frameworks and libraries however can easily bloat which is often exacerbated by poor documentation and threadbare APIs hiding too much to be of use [2].

2.2 Polyglot Programming

As stated in the introduction, alternate titles exist for polyglot programming: multi-language or multi-paradigm programming [3]. We chose to belong to the field of polyglot programming because the first alternate title is ambiguous, referring to either multiple programming languages or multiple spoken language software

development (e.g. French, Polish etc.) and the second alternate title is too limiting in scope (we want to include the use of languages of similar paradigms too).

And so formally, polyglot programming is the development of software through the use of more than one programming language [6]. At first this may seem impractical, but then one has only to consider webpage and web application development, in which one is usually using at least 3 languages (HTML, CSS and PHP). Use of multiple languages in web development arose from the separation of an applications business logic to its interface [7]. This separation allows for the logic of an application to change, without affecting the user interface and the same goes for any change to the interface. Since the business logic and interface have been separated, it is no longer necessary to use the same language (e.g. use only HTML), one can now choose to use the best language per task, provided the environment permits this of course. And here lies the crux of polyglot programming. Provided the environment permits it, one can choose to use the best language for any given task.

However, as great as this may all sound, polyglot programming has yet to take off in any big, formal, way and is also mostly limited to web development. We believe that if polyglot programming were to be treated more formally as a field unto itself, developers would be saved many resources and coding would become a much more intuitive activity [8].

3. Related Work

This research belongs to three domains: software engineering, code reuse and polyglot programming. In particular, it encompasses the use of ontologies within these three domains.

Much research has already been done on ontologies in software engineering and code reuse, while very little to nothing has been done in polyglot programming. A select group of papers and systems were reviewed, highlighting the status quo.

3.1 Ontologies in Software Engineering

Ontologies are tools for communication [13] [2], specifically, they are tools for communicating ideas about a particular domain [15]. Pan et al [2] also believe communication to be essential in software engineering:

> "We first talk to those who need the software. We then talk to other members of software development teams. We also talk to computers to encode the elicited ideas into software. Finally, our pieces of software need to talk to each other."

It therefore stands to reason that ontologies would be highly compatible tools for communication in software engineering. Wongthongtham et al. [13] also believes this to be true and goes on to define communication as a key reason for introducing ontologies into software engineering:

> "The main purpose of the software engineering ontology is to enable communication between computer systems or software engineers in order to understand common software engineering knowledge and to perform certain types of computations."

In fact, much research has already been done advocating for the use of ontologies in software engineering [2] [13] [16] [5] [7]. However, existing research has tended to focus on all encompassing software engineering knowledge that aims to guide and manage the entirety of the software development life cycle (SDLC) [2], as opposed to focusing on any one particular phase. This apparent bias has resulted in a dearth of research around the implementation phase. Existing research has also tended towards the replacement or augmentation of model-driven software development (MDSD) approaches towards ontology driven or ontology based approaches [16] [7] [2] [6]. The emphasis, therefore, has been on communicating requirements engineering and systems design knowledge between developers.

There exist three significant research efforts relevant to our research regarding ontologies for software engineering: ontology-driven software development (ODSD), ontology-based software engineering (OBSE) and the ontology-based software environment (ODE). Below is a brief discussion on each.

3.1.1 Ontology Driven Software Development (ODSD) [9]

Ontology driven software development (ODSD) is an approach to software development that leverages ontologies as guiding tools during development. With the ODSD approach:

> "Not only are ontologies used to integrate diverse software artifacts to improve traceability, but they also guide software engineers throughout the process of the software development activities such as requirements engineering and business process modeling." [9]

The team behind the ODSD approach was commissioned by the *Marrying Ontology and Software Technology (MOST)* project to come up with something that would successfully "marry" ontologies with software engineering technologies. The goal of the project being to improve software engineering technologies through ontologies and their reasoning capabilities. The team arrived at ODSD by aiming to seamlessly integrate ontologies with model-driven software development (MDSD). MDSD is software development that is based on models, modelling and

model transformations [13]. That is to say, a model is created for the problem domain and this model is worked on and eventually transformed into the solution domain, through any number of intermediary models (e.g. ERD to relational to UML to objects). The team believed that by introducing ontologies into MDSD, software models would become cleaner, bug-proof and be delivered faster to market. ODSD can in fact be considered a type of model driven software development, in that models are based on ontologies at different levels of abstraction [13].

The ontologies created in ODSD are used to track development artifacts along the whole SDLC. This allows software designers to understand and relate activities between the different phases accurately and consistently. It also allows all members of the development team to benefit from an ontology that communicates an unambiguous understanding of all the project artifacts.

3.1.2 Ontology Based Software Engineering (OBSE) [14] [13]

OBSE advocates for software projects to not only be driven by requirements and models [14], but by an ontology that acts as a knowledge base. The ontology would therefore contain application domain knowledge, from which many new projects could be started.

Modelling is a necessary activity in all software development projects, and usually the development team has to create the models from scratch [14]. OBSE, however, derives its models from an existing ontology and the new project requirements. The idea being that the ontology has application domain knowledge from previous projects and that the new project requirements fine tune the models to be specific to the project at hand. It is easy to see the benefits of such an approach. One benefit is a reduction in development time, possible only because the developers aren't always starting from nothing. Product consistency across projects is another benefit, because projects are always started from the same knowledge base. But because ontologies are always growing and changing, software products can only ever get better in terms of quality, reliability and consistency.

OBSE, like ODSD, is an approach aiming to somehow combine ontologies with existing MDSD techniques. In fact, the two approaches are nearly identical, in that their goals are the same and their belief in ontologies for the benefit of software development is also the same. However, there are some key differences between the two. They differ in that ODSD attempts to integrate every aspect of software engineering with ontologies, in an ontology-guided development approach; whereas OBSE emphasizes moving knowledge from project to project, using an ontology as a constant source of domain knowledge. They also differ in that with ODSD, the models from MDSD are integrated or replaced with ontologies; whereas with OBSE the models from MDSD are derived from ontologies. The final difference, of importance to this research, is that with OBSE, there is an explicit, direct, relationship between the ontologies and the resultant code: a domain ontology (used in previous projects), along with the project requirements, are used to create a knowledge base (an instance of the domain ontology); the knowledge base is then used to create the models (e.g. UML classes) for the project; the models are then used to generate the code. Whereas ODSD ontologies do not relate to code in any direct, easily traceable manner.

3.1.3 Ontology-based Software Development Environment (ODE) [11]

Falbo et al. [11] have created a software engineering environment (SEE), ODE (ontology-based software development environment), which uses multiple ontologies for software tool integration. The ODE team's goal was to allow for software tools processing partially common sets of data to share an understanding of what that data means. They believed that, since ontologies were essentially hubs for communication, they could be used to integrate software tools in SEEs. The solution they came up with was ODE, an ontology-based SEE.

ODE, like the two approaches mentioned above (ODSD and OBSE), takes an all-encompassing look at the SDLC, with a particular emphasis on software process, software quality and risk analysis knowledge. It differs, however, in that it has a development environment (ODEd), which is an implementation of its ontology-based approach to software engineering.

Of note, from the ODE teams' efforts, is that they used multiple ontologies at different levels of abstraction to support their environment, as opposed to implementing one monolithic ontology. The obvious advantages of taking this approach, among others, are a separation-of-concerns and an extensible, modular, system. In having multiple ontologies, troubleshooting and updating is made easier. If the system as a whole is experiencing some kind of error, one can identify which ontology to look into after considering the nature of the error. Having an ontology cluster makes the system as a whole more extensible. The existing ontologies have some way of communicating with one another and so if a new feature or new ontology is needed, one simply has to add it and leverage the existing communication framework. ODE makes use of three ontologies. A software processes ontology, a quality control ontology and a knowledge ontology. The software processes ontology contains knowledge about software

engineering activities. The quality control ontology houses knowledge that relates to quality control management tools, supporting product quality planning and quality evaluation. The last ontology is the most interesting and the most abstract. The knowledge ontology defines classes that describe knowledge about objects in the software process and quality control ontologies.

3.1.4 Conclusion

That most research on ontologies in software engineering has focused on SDLC activities like requirements engineering, systems design, quality control and other tasks normally associated with a project manager, as opposed to programming, implies that either ontologies are better suited for these tasks or that existing platforms for code reuse and polyglot programming are substantially superior. Hesse [14] argues that communication occurs at higher levels abstraction:

> "If systems or components are to exchange knowledge, this will happen more on the model than on the implementation level."

Which makes sense, considering the many possible ways any one system can be modelled. However it does not explain for the extreme bias towards the SDLC and the MDSD approach. This observation therefore justifies a fresh look into the status quo, and a reimagining of how things are.

3.2 Ontologies for Code Reuse

Ontologies are generally built with reuse in mind [9]. If not reuse of the ontologies themselves, then reuse of the knowledge within them. It is therefore not too surprising that some research has already been done on ontologies with code reuse in mind. However, existing research has left much to be desired with their black box like approaches to methods and fixation object transformation, as illustrated in the following brief reviews.

3.2.1 Source Code Representation Ontology (SCRO) [2]

Alnusair et al., for their paper "Effective API Navigation and Reuse" [2], created and proposed an ontology that would be part of an automatic source-code recommendation system. Their Source Code Representation Ontology (SCRO) was conceptually aware of a user's source code as well as the user's libraries. The SCRO ontology captured major object-oriented concepts and features including, but not limited to: encapsulation, inheritance, method overloading, method overriding, and method signatures.
The SCRO ontology, built using OWL-DL2, is part of a recommendation system that recommends contextually relevant code snippets that could be used by a programmer to complete specific programming tasks. The team took a semantic, web-based, approach, meaning that the ontology, not only explicitly represented the source code, but it also captured the codes metadata, allowing for more meaningful, informed, recommendations. The SCRO ontologies recommendation system uses a static code analysis technique, known as pointer analysis, to generate a list of the best possible candidates to recommend. The pointer analysis technique works by analyzing pointers and the data they point to, skipping over unlikely solutions and prioritizing the most viable paths.

The SCRO ontology team's approach was motivated by the idea that programming is effectively the chaining together of method calls: transforming some source object into some other target object (e.g. providing an argument to a method and receiving the method's return type), multiple times along a chain. And so their recommendation system receives queries of the form *source object* ⇒ *destination object* and recommends code snippets that transform the given *source object* into the desired *destination object*.

The SCRO ontology system, like most code recommendation systems, uses a graph-based representation of source code, with object types being the nodes of the graph and edges indicating object transformations performed by methods. That is to say that, an edge between any two nodes in the graph, indicates that there exists a method that takes, as a parameter, an object of one node type, and returns an object of an adjacent node type.

The SCRO team chose graph representations of source code to take advantage of graph-traversal algorithms which are powerful enough to, not only, find the best route between any given source and destination object pairs, but to do it in the shortest and least resource-intensive time.

However, unlike other recommendation systems, the SCRO system bases its graphs on ontologies, resulting in graphs that are enriched with additional data, yielding more accurate recommendations and faster graph traversal. Also unlike other recommendation systems, the SCRO ontology does not make use of a repository of sample code or need the backing of a source-code search engine (CSE) to acquire code snippets. Instead, their system constructs the code-snippet through a guided brute-force graph-traversal search, starting from the *source object* node, ending with the *destination object.* The SCRO system is also contextually aware, analyzing the users' current project code to better construct, rank and deliver code recommendations.

3.2.2 Source Code Extractor Framework (SCEF) [15]

Ganapathy and Sagayaraj published a paper titled "To Generate the Ontology from Java Source Code" in the International Journal of Advanced Computer Science and Applications, proposing a framework that would extract metadata from source code and store it in an OWL-based ontology. QDox, a code generator, would extract the metadata from the source code and feed it into a semantic web framework, e.g. the Jena framework, which would then store it in the ontology; with the actual source code itself being stored in an HDFS repository. This approach, of having the metadata in an ontology and the source code in an HDFS repository, allowed for more consistent and more systematic reuse. Use of an ontology provides for semantically enriched, efficient and effective searching while the HDFS repository means that code could then be stored in a distributed environment, like Hadoop, to be accessible across multiple geographical locations.

3.2.3 PARSEWeb [16]

To help programmers address the issue of not being able to get a desired object type, Suresh Thummalapenta and Tao Xie from the North Carolina State University, in their paper "PARSEWeb: A Programmer Assistant for Reusing Open Source Code on the Web", developed an approach that would take queries of the form *source object type ⇒ destination object type* as input and then output a sequence of method calls that could get the user from their provided source object to their desired destination object. Their approach interacted with a code search engine (CSE) to gather relevant code samples and performed static code analysis over the gathered samples to determine desirable sequences. They implemented their approach using a tool called PARSEWeb, a web based tool that would act as a programming assistant, in that it could recommend sequences of method calls that would allow a user to transform objects.

3.2.4 Conclusion

Having reviewed literature on ontologies in code reuse, we can only but observe that existing biases have left much room for improvement through research into alternative perspectives on ontologies for code reuse.

The first bias being a general inclination towards Object-Oriented (OO) programming, leaving most other languages uninvestigated. This is likely because the OO paradigm is both popular and also more open to the monitoring of its concepts e.g. Java source files are easily assessable and the JVM is well documented.

The second bias observed, was a bias towards ontologies for recommendation systems. The reason for this likely being that, recommendation systems benefit the most from and are essentially the ultimate goal of code reuse research.

The final bias observed in the literature, related to how the recommendation systems received queries, specifically queries of the form *source object ⇒ destination object*. Having taken in the query of form *source object ⇒ destination object*, the system would then proceed to transform the source object into the destination object through several method calls. A system like this is therefore more concerned with the result, not necessarily the journey to the result, that is to say that these recommendation systems don't care how any given method achieves its mission, choosing to take a black box approach to the code. That is to say that, these systems would recommend a sequence of three method calls versus a sequence of five method calls, even if those three method calls take longer to execute than the five.

3.3 Ontologies Supporting Polyglot Programming

Ontologies exist supporting use by and of different spoken languages, there are also ontologies used for programming, however no ontologies promoting the use of multiple programming languages can be found. We can only assume as to why this is the case but at the very least we hope to rectify this by hopefully making a significant contribution to the field.

4. An Algorithms Ontology Cluster: The Proposal

It is at this point necessary to introduce to the reader the algorithms ontology cluster. The ontologies proposed in this research aim to address the issues raised in the related work section. For ontologies in software engineering, a bias towards knowledge of the SDLC as opposed to coding was observed, leaving room for code centric ontologies. In code reuse, ontologies have mostly been created for use by recommendation systems and OO languages, with other languages being left under investigated and a pigeonholing of the applications of code reuse. And, finally, ontologies in polyglot programming have very little, if any, published work on them. We therefore believe there to be room for a tool, a unique tool, capable of addressing all of the above-mentioned issues; the tool in question being a cluster of interconnected ontologies.

Below is a proposal for an ontology cluster composed of three ontologies: an algorithms ontology, programing languages ontology and an administrative ontology. Their

responsibilities, interactions and the processes that they will govern and be governed by, are also detailed.

4.1 Algorithms Ontology

Although the system as a whole is referred to as an algorithms ontology cluster, only one ontology actually manages the algorithms: the algorithms ontology. This ontology will contain essential algorithmic concepts like: big-Oh, recursion, parameters, loops, variables, complexity, pseudocode etc. The pseudocode concept is particularly interesting in that it will reference an actual snippet of pseudocode e.g. a Fibonacci algorithm would be an instance of an algorithm, with a pseudocode attribute referencing a pseudocode file titled "*fib.pseudo*".

Initially this ontology will be informed by Introduction to Algorithms by Lehman et al [17]. The idea being to start off with the text to get a concise, while exhaustive, list of fundamental algorithms concepts. We will be taking advantage of the fact that ontologies allow for new concepts to be added as needed, meaning we have no need to fear missing any critical concepts, because once they have been identified, they will be added.
The algorithms ontology is the most critical of the ontology cluster, because it will tackle the biggest tasks of the ontology cluster, while the other ontologies will be designed to support it. Seeing as it tackles the biggest tasks, it therefore addresses many of the issues facing related systems. For ontologies in software engineering, it specifically fills the role of an ontology for coding, of which there is a dearth of. Ontologies in code reuse don't really represent the structures that make up the code they recommend i.e. being more concerned with inputs and outputs as opposed to what goes on in between. And for polyglot programming, few ontologies like this exist.

4.2 Programming Languages Ontology

The second ontology, in the algorithms ontology cluster, is the programming languages ontology. As the title implies this ontology relates programming language concepts to one another and to algorithms in the algorithms ontology e.g. paradigm, interpreted, procedural, OO, binding, type casting, run time environment, requirements etc. The idea behind this ontology is to maintain a comprehensive list of different programming language concepts, with enough detail on them to allow for codification of pseudocode into any one specific, known, language, working hand in hand with the algorithms ontology.
Initially this ontology will be informed solely by Concepts of Programming languages by Sebesta [18] but will grow from there, not unlike in the way the algorithms ontology will grow.
This ontology is particularly unique, in that we have as yet to come across any ontology that does what it does. We have come across an ontology, SCEF [15], which contains Java metadata used for searching through stored code and of course ontologies for spoken language software development [4], but none strictly for programming language concept.

4.3 Administrative Ontology

The final ontology in the algorithms ontology cluster is the administrative ontology, so titled because it's responsible for handling miscellaneous activities like: interface interaction, code storage, class management etc. It's most important responsibilities being that it references instances of actual source code and that it interacts with the interface to the agents on behalf of the other two ontologies.
This ontology will be composed using ideas from existing ontologies like SCRO [2] and SCEF [15]. Ideally we would integrate these ontologies to form the administrative ontology, however these ontologies are no longer easily available and the process of integration would necessitate the incorporation of many unneeded concepts, creating unnecessary bloat and room for semantic errors like ambiguity and redundancy.

4.4 Processes

Having described the three ontologies, it is now time to define their processes and interactions. The ontology cluster will be used by agents, both human and software alike. The agents will use the ontologies to assist them in their general programming activities. In the case of human agents, a query-based interface or recommendation system could be implemented to make use of the ontologies. Software agents would interact with the ontologies in a more direct way but in a way similar to search interaction.
There are two main system phases: a down-phase and an up-phase. In the down-phase, an agent is extracting a given algorithm and in the up-phase some code is given to the ontologies and is then summarily reverse engineered into pseudocode. Below is a more detailed explanation.

4.4.1 Down-Phase

The down-phase, is the code dissemination phase. It is the phase in which code is requested by an agent and then produced by the ontologies. The process first begins with a request from an agent to the interface. The interface then relays the request to the administrative ontology that then processes the request, determining the language, environment, algorithm and constraints. The administrative ontology then checks to see if the code exists, if the code doesn't exist it checks the algorithms ontology for the

pseudocode. If the algorithm pseudocode exists, the programming language will be queried for in the programming languages ontology. Having confirmed that the algorithm exists and that the language is known, source code will then be generated, through generative methods, and then outputted to the agent. If approved by the agent, this new source code will be integrated into the ontology cluster. If the code is dissatisfactory to the agent, it will not be integrated into the ontology cluster. Approval and disapproval of the source code generated by the ontology cluster will be used as reinforcement learning to improve the system's ability to generate accurate code.

4.4.2 Up-Phase

This phase can be thought of as the knowledge acquisition phase. The user submits to the ontology cluster a snippet of code (e.g. a method, code block, subroutine etc.) with sufficient details to reverse engineer it into its pseudocode, disseminating information to the three ontologies throughout the process. If the language is new to the ontologies, the user will have to incorporate the new language into the programming language ontology, with certain basic constructs being automatically picked up.

5. Methodology

Going about creating the three ontologies would necessitate the use of a very particular kind of methodology. A methodology specific to ontology creation that also conformed to standard research practices for computer science research would be ideal, because the research is about more than just the creation of ontologies. Having reviewed several methodologies, both for ontology creation and computer science research, we found none that could accomplish the task at hand with satisfaction. However, we did find a methodology for computer science research that did part of what we wanted to do and an ontology specific methodology that also did part of what we wanted to do; the methodologies in question being Nunamaker's systems development methodology and the Methontology methodology respectively. It was at this point that we decided to hybridize the two methodologies giving rise to a research driven, ontology development methodology.

5.1 Nunamaker's Systems Development Methodology [19]

Nunamaker's systems development methodology was chosen because it explicitly made room for the necessary research contributions that one would expect to come out of comprehensive research. It also provided room for repetition and iteration, repetition and iteration being important because they reduce the burden of researcher inexperience, allowing for mistakes to be made and for said mistakes to be corrected in future iterations. Another reason for the iteration being important is because it would allow for the consistent, formal, refinement of the ontologies, a necessary and essential task.

Fig. 1, is a diagrammatic representation of Nunamaker's system's development methodology, the foundation upon which this research's approach was built upon.

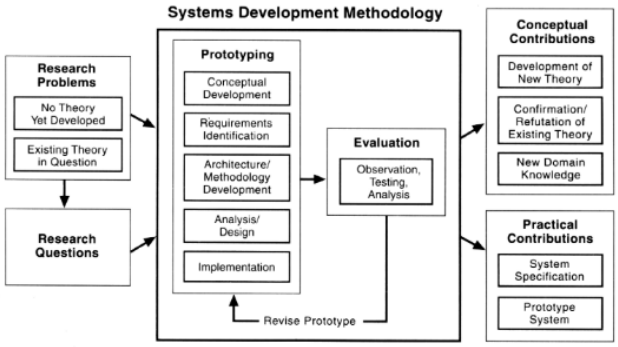

Fig. 1 Nunamaker's System Development Methodology

5.2 Methontology

Several ontology development methodologies were reviewed before Methontology [20] was chosen to conduct part of this research. Methontology was chosen because it proved itself general enough to be considered an all-purpose, fundamental, ontology development methodology and because it was straight forward and concise when creating an ontology "from scratch" [20]. Most popular ontology development methodologies were too specific to a language or an environment to be truly considered a methodology; they were more akin to tutorials in that they provided a set of instructions to produce a very specific instance of an ontology, meaning they could prove restrictive if the ontology to be developed varied greatly from their template or implementation. The simplicity of Methontology also allowed for easy integration with Nunamaker's systems development methodology and also remaining easy to apply, irrespective of the ontology being developed.

Methontology has 7 phases for ontology development: Specification, Knowledge Acquisition, Conceptualization, Integration, Implementation, Evaluation and Documentation. The 7 phases will be discussed in more detail in the next section on the final hybrid methodology, the research oriented ontology development methodology.

5.3 The Research-oriented Ontology Development Methodology

In order to arrive at a suitable methodology that could create an ontology for use in research, Nunamaker's Systems Development Methodology and Methontology were combined to give the Research-oriented Ontology Development Methodology (RODM), as seen in figure 2.

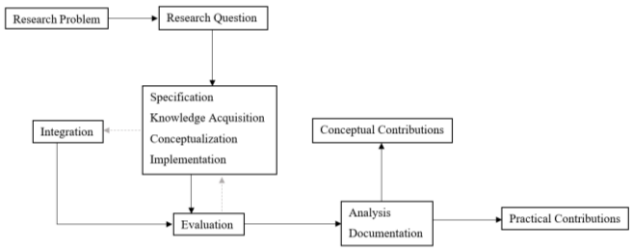

Fig. 2 The Research-oriented Development Methodology

Following is a step-by-step description of each key activity in the research oriented ontology development methodology.

5.3.1 Research Problem

In this first phase, a researcher identifies a gap that has as yet to be addressed. Naturally said gap has to be substantial enough to warrant formal research of some kind. In our case we had observed underutilized and underperforming tools in both code reuse and polyglot programming.

5.3.2 Research Question

Having identified a potential problem area it is now time to propose a theory as to how one could address it. This is usually in the form of a hypothesis, a problem statement, a research question or all three. Depending on the nature of the research, the researchers will choose accordingly. It is also at this point that the researcher identifies ontologies as a viable tool for addressing the research problem.

5.3.3 Specification

The specification phase is the phase in which the researcher has officially decided that an ontology is in fact the best way to address the research problem. It is at this point that the researcher produces an informal, semi-formal or formal ontology specification document. An ontology specification document is effectively a broad description of the ontology to be created: its title, purpose, goal, the kinds of concepts it intends to represent, environment etc. are all outlined in the specification document. This document is used to keep the researcher within scope and also as a referral document for any other's whom are interested in the ontology.

We chose to go with a semi-formal specification document, to keep the document readable by non-ontology experts while also being able to use concise technical terms to keep the document short and to the point.

5.3.4 Knowledge Acquisition

Having specified the kind ontology to be built, it is now time to identify sources of concepts, classes and relationships. The researcher must choose appropriate sources that allow for a fully functioning, complete, ontology.
In our case we had three initial sources since there are three ontologies. For the Algorithms ontology we chose to go with Introduction to Algorithms by Cormen et al [17]. The programming languages ontology was informed by Concepts of Programming Languages textbook by Sebesta [18]. The third and final ontology was mostly informed instinctively, since its responsibilities were mainly administrative. However, some aspects of it will be from the integration with other, existing, ontologies (this will be elaborated upon in the Integration phase).

5.3.5 Conceptualization

In this phase, all domain knowledge acquired from the knowledge acquisition phase must be formed into some conceptual model before implementation into an ontology. Popular ontology conceptual models include UML Class diagrams and Mind-maps. The idea behind this activity is to certify that the logic of the ontology is in order before anything is committed to code.

5.3.6 Integration

Ontologies are designed to grow, not only through the introduction of new concepts but also through other ontologies. Why create an ontology from scratch when you can re-appropriate parts of an existing one?
As mentioned earlier, the third ontology will not be built strictly from scratch but from one or two ontologies, SCRO [2] and SCEF [15] in particular. The SCRO ontology team demonstrated strengths in conceptual knowledge representation of code, even though they were bias towards OO. The SCEF [15] ontology team used an HDFS repository in conjunction with their ontology to reference snippets of code, a technique that will be of particular use when it comes to referencing our snippets of code and pseudocode.

5.3.7 Implementation

In this phase, the ontology will finally be implemented. Usually this is the act of transforming some conceptual model into an ontology.

In our case, we chose to implement the ontologies using description logic (OWL DL), to curb the possibility of redundancy and ambiguity. Having completed the DL axioms the ontologies will be prototyped in Protégé.

5.3.8 Evaluation

Once the ontologies have been built, it will then be time to evaluate whether or not they can do what was set out for them in the ontology specification document. Depending on what the ontology is meant to be used for an appropriate testing and evaluation method will be chosen.
In the case of the Algorithms Ontology Cluster, agents will be designed to make use of the ontologies. The ontologies will therefore be evaluated on how well the agents can perform their duties. If the performance is dissatisfactory or something is found to be lacking, we can return back to the specification phase to re-specify some aspect of the ontologies.

5.3.9 Analysis

Having completed the ontology and having also evaluated it, it is now necessary to analyze its performance with respect to the ontology specification document and the research problem. It is at this point that we step out of the development of an ontology and enter a research contributions phase. That is to say that, it is now time contextualize your ontology and consider what effect it has on the greater ontology community.

5.3.10 Documentation

Several documents can be generated coming from the analysis phase. The most important document, however, would have to be the research report, which would contain the analysis of the ontology. The research report can be in the form of a long-form journal paper or even a dissertation, if the contributions are significant enough. Other documentation produced in this phase can include the final draft of the ontology specification document, interesting sections of the research published at conferences and even technical documents for reuse of the ontology.

5.3.11 Conceptual Contributions

Having completed the research, one can now consider what they've contributed to the ontology development community. Several conceptual contributions are possible e.g. new domain knowledge, a new theory, a re-imagination of an existing approach etc.

At the very least the research should contribute new domain knowledge or extend existing domain knowledge, if it didn't do this, then it isn't research.

5.3.12 Practical Contributions

Practical contributions include but are not limited to the ontology itself and any documents relating to it like the specification document, user manual, reuse manual etc. These contributions are essential for the research to continue after completion. Ideally, one wants their ontology to be used and reused by as many people as possible. This provides a platform for improvement and constant evaluation, resulting in a more refined, more practical ontology.

6. Research Contributions

6.1 Research-oriented Ontology Development Methodology (RODM)

It is our hope that this methodology is adopted by the ontology building community as a viable option for developing ontologies for research. It was our observation that many tutorials are available on how to build an ontology, but very few could suffice as research methodologies. And those methodologies that incorporate ontology building as part of their activities, do not look at the research from beyond the ontology perspective.

6.2 The Algorithms Ontology Cluster

Seeing as this ontology is quite unique in what it attempts to accomplish, we hope it to be a significant contribution to the three domains of software engineering, code reuse and polyglot programming, which it falls under. Polyglot programming in particular stands to gain the most from the ontologies, seeing as nothing like this has ever been done before.

7. Conclusion and Future Work

It is our belief that the Algorithms Ontology Cluster will be of great use to those that practice code reuse and polyglot programming. At the very least, we know that it is a truly unique alternative that provides for a different way to get things done.

Because ontologies are intelligent they are unlikely to face bloat like libraries or frameworks. The three ontologies are designed, from the ground up, to be without redundancy and ambiguity (through the use of description logic). The ontologies have a single interface that handles input and output, effectively making this a black box system. The agent only ever knows whether it has something or not by querying this one interface.

As for polyglotism, ontologies have as yet to be released that offer similar features. Some ontologies do exist for specific programming languages (with a bias for OO), but none attempt to bridge languages for use within a single project.

The ontology cluster is designed for use by agents, both human and software alike, however the relationship goes even further when it comes to software agents. There are many tasks that could be done by agents that will initially be done by simple scripts. Integration with agents was considered out of the scope of this initial building of the system and thus has been left for future visits into this project.

In conclusion, we believe the ontology cluster proposed in this paper, will be a viable alternative to current code reuse tools and polyglot programming and look forward to completing this research in the coming months.

References

[1] T. Davis, (2013, June 6), "What's the use of code reuse", [Online], Available: https://www.simple-talk.com/blogs/2013/06/06/whats-the-use-of-code-reuse/
[2] A. Alnusair, T. Zhao and E. Bodden, "Effective API Navigation and Reuse", IEEE International Conference on Information Reuse and Integration, Vol. 11, 2010, pp. 7-12.
[3] M. Toele, "Multi-language Software Development in DLX", M.S. thesis, Faculty of Natural Sciences, University of Amsterdam, Amsterdam, Netherlands, 2007.
[4] P. Wongthongtham, E. Chang, T. Dillon and I. Sommerville, "Development of a Software Engineering Ontology for Multi-site Software Development", IEEE Transactions on Knowledge and Data Engineering, 2008, Vol. 21, No. 8, pp. 1205-1217.
[5] W. B. Frakes and K. Kang, "Software reuse research: status and future". IEEE Transactions on Software Engineering, Vol. 31, No. 7, 2005. pp. 529-536.
[6] H. C. Fjeldberg, "Polyglot Programming: A Business Perspective". M.S. Thesis, Department of Computer and Information Science, Norwegian University of Science and Technology, Trondheim, Norway, 2008.
[7] J. Harmanen, "Polyglot Programming in Web Development", M.S. Thesis, Faculty of Computing and Electrical Engineering, Tampere University of Technology, Tampere, Finland, 2013.
[8] D. Intersimone, (3 Nov. 2009), "Polyglot programming -- development in multiple languages" [Onine] Available at: http://www.computerworld.com/article/2467812/internet/polyglot-programming----development-in-multiple-languages.html
[9] J. Z. Pan, S. Staab, U. Aßmann, J. Ebert and Y. Zhao, "Ontology-Driven Software Development", Berlin Heidelberg: Springer-Verlag, 2013.
[10] A. Bachmann, W. Hesse, A. Russ, C. Kop H. C. Mayr, and J. Vöhringer, "OBSE – an approach to Ontology-based Software Engineering in the practice", Enterprise Modelling and Information Systems Architectures, 2007, pp. 129-142.
[11] R. de A. Falbo, A. C. C. Natali, P. G. Mian, G. Bertello and F. B. Ruy, "ODE: Ontology-based software Development Environment", Congresso de Ciencas De la Computation, 2003, pp. 1124-1135.
[12] H. Happel and S. Seedorf, "Applications of Ontologies in Software Engineering", Semantic Web Enabled Software Engineering, 2006, pp. 1-14.
[13] J. B. Alonso, "Ontology-based Software Engineering: Engineering Support for Autonomous Systems", Integrating Cognition + Emotion + Autonomy, 2006, pp. 8-35.
[14] W. Hesse, "Ontologies in the Software Engineering process", Proceedings of the Workshop on Enterprise Application Integration, 2005.
[15] G. Ganapathy and S. Sagayaraj, "To Generate the Ontology from Java Source Code". International Journal of Advanced Computer Science and Applications, Vol. 2, No.2, 2011, pp. 111-116.
[16] S. Thummalapenta and T. Xie, "PARSEWeb: A Programmer Assistant for Reusing Open Source Code on the Web". IEEE/ACM International Conference on Automated Software Engineering, Vol. 22, 2007, pp. 204-213.
[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, "Introduction to Algorithms," USA: The MIT Press, 2009.
[18] R. W. Sebesta, "Concepts of Programming Languages," New Jersey: Pearson, 2012.
[19] Morrison, J. and George, J. E. Exploring the Software Engineering Component in MIS Research. Communications of the ACM July 1995 Volume 38, No. 7, pp. 80-91.
[20] Fernández, M., Gómez-Pérez, A. and Juristo, N., "Methontology: From Ontological Art towards Ontological Engineering," AAAI Technical Report SS-97-06, 1997, pp. 33-40.

Karabo Selaolo B.Sc. Computer Science (2014) University of Botswana, M.Sc. student Botswana International University of Science and Technology. Consultant ExceQ Services Pty. (Ltd.) (2014), Teaching Assistant Centre for Management, Entrepreneurship and General Studies, Department of General Studies and Teaching Assistant College of Science, Department of Computer Science. Currently my research interests lie in Cognitive Sciences and Neuromorphic Technologies, hopefully a Ph.D. will be born!

Hlomani Hlomani received his bachelor's degree in Information Technology from the Cape Peninsula University of Technology, South Africa in 2005 and both his MSc and PhD degrees at the University of Guelph, Canada in 2009 and 2014, respectively. Currently, he is a Lecturer and Researcher in the College of ICT at the Botswana International University of Science and Technology. He also holds an Adjunct position in the School of Computer Science at the University of Guelph. His research interests span several disciplines within the computer science domain including knowledge engineering, artificial intelligence, distributed systems and software engineering. He basically seeks solutions to computing problems using a blend of different technologies. Recent research problems include those in data integration, modeling, knowledge representation, prediction, clustering, classification and machine learning. He has received a best paper award at the Knowledge Engineering and Ontology Development (KEOD) conference in 2014. He has published several papers including journal articles, conference papers, and book chapters.