Original Article

Healthc Inform Res. 2013 December;19(4):271-277.
http://dx.doi.org/10.4258/hir.2013.19.4.271
pISSN 2093-3681 • eISSN 2093-369X

HIR
Healthcare Informatics Research

Archetype Model-Driven Development Framework for EHR Web System

Shinji Kobayashi, MD, PhD[1], **Eizen Kimura, PhD**[2], **Ken Ishihara, MD, PhD**[2]
Departments of [1]Bioreguratory Medicine and [2]Medical Informatics, Graduate School of Ehime University, Ehime, Japan

Objectives: This article describes the Web application framework for Electronic Health Records (EHRs) we have developed to reduce construction costs for EHR sytems. **Methods:** The openEHR project has developed clinical model driven architecture for future-proof interoperable EHR systems. This project provides the specifications to standardize clinical domain model implementations, upon which the ISO/CEN 13606 standards are based. The reference implementation has been formally described in Eiffel. Moreover C# and Java implementations have been developed as reference. While scripting languages had been more popular because of their higher efficiency and faster development in recent years, they had not been involved in the openEHR implementations. From 2007, we have used the Ruby language and Ruby on Rails (RoR) as an agile development platform to implement EHR systems, which is in conformity with the openEHR specifications. **Results:** We implemented almost all of the specifications, the Archetype Definition Language parser, and RoR scaffold generator from archetype. Although some problems have emerged, most of them have been resolved. **Conclusions:** We have provided an agile EHR Web framework, which can build up Web systems from archetype models using RoR. The feasibility of the archetype model to provide semantic interoperability of EHRs has been demonstrated and we have verified that that it is suitable for the construction of EHR systems.

Keywords: Electronic Health Records, Internet, Computing Methodologies, Automatic Data Processing

Submitted: Octoeber 15, 2013
Revised: December 2, 2013
Accepted: December 4, 2013

Corresponding Author
Shinji Kobayashi, MD, PhD
Department of Bioreguratory Medicine, Graduate School of Ehime University, Shitsukawa, Toon, Ehime, Japan. Tel: +81-89-960-5695, Fax: +81-89-960-5696, E-mail: skoba@moss.gr.jp

I. Introduction

The openEHR project is an open-source software, not-for-profit organization that has published a series of design specifications and an implementation of a future-proof interoperable Electronic Health Record (EHR) system [1]. The ISO/CEN 13606 standards are based on the architecture of this project, which is characterized by two-level modeling that separates clinical concerns from the computable information structure, and is the result of more than 15 years of research and development in Australia and European Union countries [1]. The original implementation was in Eiffel and C#, and a Java implementation was later provided by a Swedish team [2]. However, the TIOBE index (TIOBE Software, Eindhoven, The Netherlands), which lists the most popular computer languages, indicates that Java, C# and Eiffel only account for about 26% of software developers [3]. Additional implementations are needed to appeal to a wider range of users. Scripting languages are becoming increasingly popular among Web developers because of their inherently streamlined programming processes and dynamic behaviour. Ruby is an object-oriented scripting language [4] that was recently recognized for its efficient Web development frameworks, such as Ruby on Rails (RoR) [5]. The efficiency of RoR was proved with a demonstration video, in which a weblog system was created within 15 minutes [6]. A group of Japanese doctors who specialize in medical information system programming were inspired by the design concept behind the openEHR project, and proposed the implementation of a Ruby version as open-source software. The mission goal is a rapid development environment for creating an EHR system based on the ISO/CEN 13606 standards. The Ruby implementation project began in October 2007.

In this paper, we discuss the validity of the Ruby implementation and examine the universality of the openEHR specifications.

II. Methods

The Ruby implementation of openEHR was developed with Ruby 1.9.3 or later as a platform (2.0.0 recommended), using related libraries, such as RoR 4.0 and the Treetop parser library.

One of the principles of the Ruby implementation is to be faithful to the openEHR specifications while applying Ruby's unique characteristics to make the programming experience a more pleasant one. We used the Web-based redmine system for task management and git for the source code repository. More information on the Ruby implementation project and source code are available at http://openehr.jp/ref-impl-ruby. In addition, these resources are available under the openEHR open-source software license (Mozilla tri-license), as is the case for the other implementations of the openEHR project. Developers can choose the GNU General Public License (GPL), the Lesser GNU GPL (LGPL), or the Mozilla Public License (MPL), according to their needs.

Because the current Ruby implementation policy mainly involves agile programming, we made it a rule that unit tests should be written prior to the working code. At first, we used the test/unit package for unit tests, but switched to the RSpec2 package for its narrative descriptive feature. All tests are automatically performed by spork and the guard package.

The reference model of openEHR defines basic concepts, data types, data structures and support information to manage an EHR system. These models were first defined in Eiffel via contraction patterns and explicit rules of invariant assertion. The handling of invariance and checking is one of the difficulties of this implementation, and this was also noted in the Java implementation [2]. Because Ruby does not have strict typing or generics, the Design by Contract pattern is not supported in native. Retaining the Ruby look-and-feel makes the code familiar to other Ruby developers, and involves the use of conventional naming and common Ruby idioms. However, the DataValue package classes in Ruby were designed with check routines to assure the validity of assigned data, as in other static typing language implementations using dynamic typing.

Some of the original forms were considered for the nomenclature of the classes, but we decided to use camel case with the Rails convention because other projects (such as Java adopted camel case nomenclature and the Rails convention) are also written in camel case.

The Archetype model is constructed in parts to describe archetypes. It includes definition, ontology, profile, assertion and constraint rules. These on-memory archetypes are serialized to an archetype definition language, and archetypes are also generated from serialized definitions. The components also support object validation and creation on a single archetype constraint level.

The Archetype Definition Language (ADL) parser was originally developed using the racc parser library, which is a standard yacc type LALR(1) parser generator included in the Ruby standard library. This parser can parse ADL files, but it performs poorly and has a problem handling V_C_DOMAIN_TYPE. To achieve better performance, the ADL parser was re-implemented with the Treetop library, which is an implementation of the Packrat parsing algorithm [7]. Performance tests were conducted to compare the Ruby ADL parser and the Java ADL parser, whose runtime is available as open-source software. The two runtime environments were Ruby 1.9.2p290 and Diablo JDK 1.6.0 (64-bit server version), installed on a 3.3 GHz Intel Core i5 processor with 16 GBytes of RAM, running FreeBSD 8.2. The ADL parser was chosen for the performance tests because it executes not only the parser library, but also multiple related packages that generate numerous instances.

An RoR related library to utilize openEHR archetype and Web service application programming interface (API) has been development preliminary as another package.

III. Results

We implemented most of the openEHR specifications and utilities including the ADL parser to generate a Web system by RoR (Table 1). The core libraries are compact in comparison with other implementations (Table 2).

1. ADL Parser

An archetype is a formal definition of a distinct domain-level concept in the form of structured and constrained combinations of reference model classes [8].

While the latter will remain stable, the former must be flexible enough to express medical concepts that will evolve with medical practice and knowledge. The primary objective is to provide a formal expression of clinical knowledge in an interoperable and reusable way. An archetype is composed of four main parts: a header section, a description section, a definition section, and an ontology section.

The description section includes metadata information, such as audit information, life cycle status, or purpose. The definition section is a basic formal definition of the archetype, containing restrictions arranged in a tree-like structure created from the reference information model. The ontology section includes the terminological definitions and bindings that link the data structures and content to the knowledge resources.

ADL is a formal language for expressing such archetypes. It is also composed of four parts, corresponding to the structure of an archetype, and uses two main types of syntax (cADL and dADL). cADL is used to express archetype definitions, and it enables constraints on data defined by object-oriented information models to be expressed in archetypes or other knowledge definition formalisms [9]. On the other hand, dADL is used to express data appearing in the language, description, ontology, and revised_history sections. It provides a formal means of expressing instance data based on an underlying information model [9].

1) Semantics of the ADL parser

According to the two-level modeling approach, the ADL parser produces in-memory representations of Archetype Model (AM) instances built upon Reference Model (RM) components. The AM defines the semantics of an archetype, and in particular, the relationships that must hold true between the parts of an archetype in order for it to be valid as a whole [9].

Thus, the principal roles of the semantics of ADL are to construct an AM based on underlying RM classes and to validate the numerous constraints imposed on an archetype. The validation should mainly be performed on complex objects and primitive types, in the invariant, specialization, and ontology sections. There are three types of constraints on complex objects: complex object structures, internal references, and archetype slots. The invariant section in an ADL archetype introduces assertions that relate to the entire archetype and can be used to make statements that are not possible within the block structure of the definition section. It is a type of first-order predicate logic with equality and comparison operators [9]. In the ontology section, the codes

Table 1. Packages implemented by Ruby

Package name	Brief description
AssumedTypes	Basic type to describe data
RM::Support	Support information model for ID or terminology
RM::Security	(not well defined)
RM::Integration	Item definition for composition
RM::EHR	EHR structure information
RM::Demographic	Description for personal or group data
RM::DataTypes::Basic	Base component to represent data types
RM::DataStructures	Data structure definition
RM::Composition	Data structure and rules
RM::Common	Common component to regulate EHR system, such as versioning
AM::Archetype	Archetype object validation and construction
AM::Archetype::Profile	Implementation of domain data types
Parser	Generates archetype object from ADL
Ruby on Rails plugin	Generates EHR Web system skeltons from archetype definition

RM: Reference Model, AM: Archetype Model, EHR: Electronic Health Record, ADL: Archetype Definition Language.

Table 2. Effective steps[a] of openEHR libraries

Language	AM	RM	Total
Eiffel	10,145	8,258	18,403
C#	5,472	17,488	22,960
Java	11,603	3,642	15,245
Ruby	945	3,358	4,303

EHR: Electronic Health Record, AM: Archetype Model, RM: Reference Model.

[a]Program steps were counted excluding comments or blank lines. Because each project has its own utility library extended from standard specifications, we compared core libraries under faithful conditions.

representing node IDs or bindings to terminologies should be linked to the appropriate entities.

Specialization is expressed using object-oriented inheritance relationships, but its semantics differs from that of inheritance, because of the constrained nature of archetypes [1]. Any data created via a specialized archetype must thus conform to both the archetype and its parent.

2) An implementation of the ADL parser in Ruby

The main goal of the library we have developed is to facilitate the conversion of ADL to AM objects that will be suitable building blocks for Ruby applications, including Web applications built on RoR.

Because Ruby applications are provided as gem packages, it is easy to embed archetype-enabled functionality in them. For example, when an EHR system is developed as an RoR Web application, the clinical content of the application can be expressed as archetypes. Moreover, communication between these applications should be facilitated via ADL. One part of the ADL parser that was difficult to implement was its scanners. cADL and dADL use slightly different sets of tokens, and they switch back and forth as the parsing process proceeds. The Java implementation project encountered this problem and implemented the ADL parser by LL(1) [2], using the JavaCC parser generator library. The old version of the Ruby ADL parser is also dependent upon a combinator parser library called yaparc and racc, which is a LALR(1) parser generator. This parser functions well, but has poor performance. To achieve better performance, we changed the parser algorithm from LALR(1) to parsing expression grammar (PEG)/Packrat parsing [7].

At first, this new parser failed to parse the cADL section, because cADL has a left-recursive rule that the Packrat parser does not support directly. Fortunately, a left-recursive rule can always be rewritten as an equivalent right-recursive rule [10]. After we modified the grammar of the left-recursive rule, the new parser successfully parsed the cADL section and exhibited better performance.

Regarding ADL semantics, the ADL parser must implement semantic functions for numerous validations (as described in the previous section), such as ADL specialization or assertion. We are currently investigating a design and implementation of these facilities to fit the specifications.

3) ADL parser performance test

Language performance benchmarks have proved that Ruby is slower than most other languages [11,12]; hence, the Ruby ADL parser performed more slowly than that of Java. Nevertheless, although Ruby process execution took 3.72 times more CPU time than the Java parser for 100 trials, this cost decreased as the number of trials increased (Figure 1).

2. Archetype Model

AM functions as an instance of the semantic information model of openEHR. We implemented most of the specifications. The ADL parser generates an AM instance to manage a clinical model dynamically described by ADL.

The AM package also supports validation of the generated archetype rules based on constraint validation. Almost all specifications have been compliantly implemented with Ruby.

3. Reference Model

RM is used to describe the actual health data in openEHR. We implemented most of the specifications. The mapping between the assumed library in openEHR and the Ruby native library was perfect. Because we changed the test package from test/unit to RSpec2 while implementing the reference models, and because RSpec2 allows narrative descriptions of code behavior, the codes were refined and elevated in their readability. Eventually, the codes were tested via two other methods, which also helped to refine the codes and elevate their readability.

Our achievements cover most of the packages shown in Table 1. Because the detailed specifications of some packages are not determined, we could not implement those packages. Other projects have had the same problem. For example, RM::Support::Measurement and RM::DataTypes::TimeSpecification have not been implemented in Java or Eiffel.

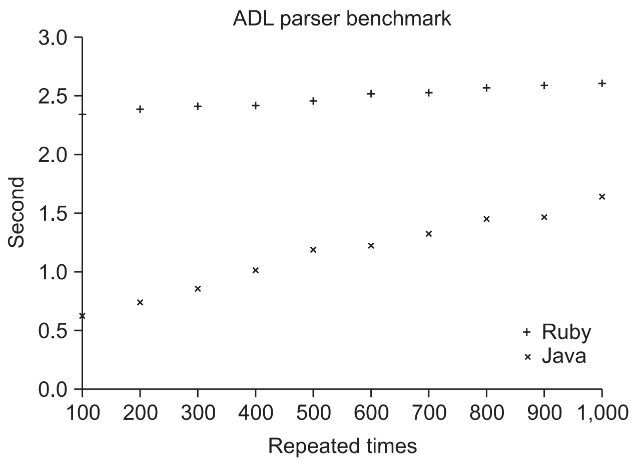

Figure 1. Archetype Definition Language (ADL) parser performance test comparing the Ruby and Java implementations. The Ruby parser requires more CPU time than the Java parser.

4. Web Framework

Clinical Knowledge Manager (CKM) is an archetype repository of the openEHR project. It provides qualified archetypes to share the development process and Web service API. We implemented the CKM access library to query archetypes via Web service.

RoR-related components have been implemented to generate Web page skeletons, database models, and controllers by RoR convention from an AM. The RoR generator works as Ruby application template from archetype. The generator makes the following artefacts from archetype definition in one step.

(1) Database schema
(2) HTML, JavaScripts and Stylesheets
(3) Controller modules
(4) Multi-lingual translation

IV. Discussion

Archetypes of openEHR specifications and the ISO 13606 standard have been discussed in terms of their feasibility for interoperability, and their suitability for EHR systems has been demonstrated [13,14]. Therefore, using archetype to design EHR systems is promising to prove its interoprability.

The openEHR specifications were first implemented in Eiffel; thus, the openEHR modeling concept is influenced by Eiffel. Table 3 shows the features of the languages used for openEHR specifications. All of the languages have object-oriented designs, but the ideas behind the designs are different, especially regarding inheritance and typing. Because the complexity of multiple inheritance often causes fatal corruption, Java and Ruby do not permit multiple inheritance of objects as a language specification. Java does permit multiple inheritance in interfaces, and Ruby allows reuse of the method of multiple modules as a 'mix-in'. For example, the DATE_TIME class in the openEHR assumes that the types library was implemented in Ruby as a class that combines the methods of the DATE module and the TIME (Figure 2) via 'mix-in'. Thus, Ruby can program multiple inheritances while maintaining its simplicity. In other languages, multiple inheritance is a point of criticism because of its complexity [15]. Using delegation instead of inheritance is an alternative idea to avoid breaks in encapsulation [15,16]. Because the openEHR specifications are not finalized, re-factoring of the classes might be necessary.

For example, the ARCHETYPE class depends on the ARCHETYPE_ONTOLOGY class, and the ARCHETYPE_ONTOLOGY class depends on the ARCHETYPE class. Figure 3 shows a simple diagram illustrating this relationship. In the openEHR mailing list, Thomas Beale suggested using closure to resolve this issue, but Java does not yet have closure. However, Ruby does have closure in its syntax, which may solve the problem in this implementation. In addition, the Ruby statement 'require' loads another library only once (Figure 4). Therefore, the Ruby implementation is not affected by this problem. On the other hand, circular import is a significant problem for Java, suggesting that this specification might be re-factored.

Typing of languages is a controversial theme with a long history of discussion [17]. Whereas the other implementations of the openEHR specifications are based on strong

Table 3. Specifications of the languages used in openEHR implementations

Name	Typing	Inheritance	Runtime
Eiffel	Strong	Multiple	Compiler
C#	Strong	Single	Compiler/CLI
Java	Strong	Single	Compiler/VMs
Ruby	Weak	Single	Interpreter

EHR: Electronic Health Record, CLI: Common Language Infrastructure, VM: virtual machine.

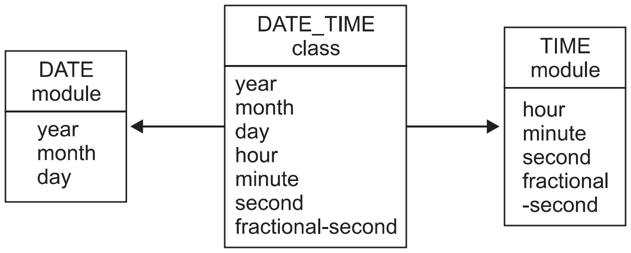

Figure 2. Example of a Ruby mix-in of the DATE_TIME class in the Assumed Types Library.

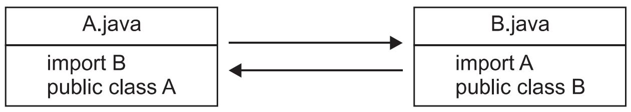

Figure 3. Sample circular import code in Java. Class A imports Class B and Class B imports Class A circularly.

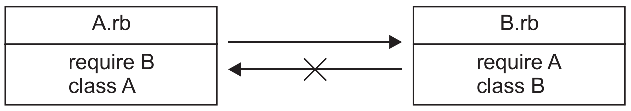

Figure 4. Sample noncircular import code in Ruby. After class A requires class B, both classes are loaded in memory. In this situation, even if class A is required by class B, Ruby does not load class A again.

typing languages, our implementation was accomplished with Ruby, a weak (duck) typing language. Duck typing was especially useful for implementing the data value packages.

As noted above, Ruby has a smart and simple programming syntax. Therefore, we were able to implement a core library with fewer steps than the other languages (Table 2). While program steps do not directly indicate programming efficiency, fewer code steps do reflect reduced programmer effort. This implementation of the openEHR specifications suggests that Ruby might provide an efficient development environment for an EHR system. Moreover, Web generators can provide many artefacts instantly, which should be generated by many steps by hand as usual. This suggests this implementation may reduce EHR Web system development costs, too.

Unfortunately, however, Ruby is also a slow language, and it has been proven so by benchmark tests [11,12]. These tests have demonstrated that Ruby process execution takes from 2 to 1,000 times as much CPU time as Java. The ADL parser performance test also highlighted this disadvantage of the Ruby implementation (Figure 1), but the time factor was only 3.72, which is much smaller than those of the benchmark tests. Moreover, as the number of trials increased, the difference in execution time decreased. One possible reason for this is that the Packrat parser algorithm used in the Ruby parser performs better than LL(1), the JavaCC parsing algorithm. Although execution speed sometimes critically influences system performance, Ruby is used for enterprise systems because of its high development efficiency. When a Web vendor adopts Ruby to launch a new service quickly, a clinical information system sometimes needs to launch a new service in a short time to meet clinical demands. This Ruby implementation would be suitable for such a situation.

The development of the ADL parser library still has some issues to be resolved. First, the current parser produces coarse-grained AM objects, in that almost all validations are missing in the current parser. Second, the current ADL parser is capable of handling only the openEHR version of the reference model. Third, there is no Ruby implementation of the ADL serializer, which will convert AM to ADL, and the archetype template mechanism has not been implemented at this stage of the development.

We are currently developing an EHR system using these libraries and RoR. The CKM access module is one of them to build Web applications on the archetype clinical model. At the present time, most EHR developers are not familiar with Ruby, and the healthcare industry has had less experience with Ruby than with other languages. However, the number of Ruby developers has been increasing worldwide. Therefore, our Ruby implementation of openEHR has the potential to be a next-generation e-Health platform, and we are undertaking a new project to construct an EHR system using this library and RoR.

Conflict of Interest

No potential conflict of interest relevant to this article was reported.

Acknowledgments

We thank the openEHR Foundation and community. Financial support for this study was provided by JSPS Grant-in-Aid for Scientific Research (C) number 24590614.

References

1. Beale T. Archetypes: constraint-based domain models for future-proof information systems. In: Proceedings of the 11th OOPSLA Workshop on Behavioral Semantics: Serving the Customer; 2002 Nov 4; Seattle, WA. p. 16-32.
2. Chen R, Klein G. The openEHR Java reference implementation project. Stud Health Technol Inform 2007;129(Pt 1):58-62.
3. TIOBE Software. TIOBE Programming Community Index [Internet]. Eindhoven, The Netherlands: TIOBE Software; c2013 [cited at 2013 Oct 2]; Available from: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.
4. Matsumoto Y. Ruby programming language [Internet]. [place unknown: publisher unknown]; c1993 [cited at 2013 Oct 2]. Available from: https://www.ruby-lang.org/en/.
5. Thomas D, Hansson DH, Breedt L. Agile Web development with Rails. Raleigh (NC): Pragmatic Bookshelf; 2005.
6. Hansson DH. How to build a blog engine in 15 minutes with Ruby on Rails [Internet]. [place unknown: publisher unknown]; 2005 [cited at 2013 Oct 1]. Available from: http://www.youtube.com/watch?v=Gzj723LkRJY.
7. Bryan Ford B. Packrat parsing: simple, powerful, lazy, linear time, functional pearl. ACM SIGPLAN Not 2002;37(9):36-47.
8. International Organization for Standardization. Health Informatics: electronic health record communication. Part I. Reference model. Geneva, Switzerland: International Organization for Standardization; 2008. (ISO

13606-1:2008).
9. Beale T, Heard S. Archetype Definition Language [Internet]. London: The openEHR Foundation; 2007 [cited at 2013 Sep 10]. Available from: http://www.openehr.org/releases/1.0.1/architecture/am/adl.pdf.
10. Aho AV, Sethi R, Ullman JD. Compilers, principles, techniques, and tools. Reading (MA): Addison-Wesley; 1986.
11. Corlan AD. Programming language benchmarks [Internet]. [place unknown: publisher unknown]; c2013 [cited 2013 Oct 3]. Available from: http://dan.corlan.net/bench.html.
12. Fulgham B. The computer language benchmarks game [Internet]. [place unknown: publisher unknown]; c2013 [cited 2013 Oct 3]. Available from: http://shootout.alioth.debian.org/.
13. Costa CM, Menarguez-Tortosa M, Fernandez-Breis JT. Clinical data interoperability based on archetype transformation. J Biomed Inform 2011;44(5):869-80.
14. Marcos M, Maldonado JA, Martinez-Salvador B, Bosca D, Robles M. Interoperability of clinical decision-support systems and electronic health records using archetypes: a case study in clinical trial eligibility. J Biomed Inform 2013;46(4):676-89.
15. Truyen E, Joosen W, Jorgensen BN, Verbaeten P. A generalization and solution to the common ancestor dilemma problem in delegation-based object systems. In: Proceedings of the 2004 Dynamic Aspects Workshop; 2004 Mar 23; Lancaster, UK. p. 103-19.
16. Bloch J. Effective Java 2nd ed. Reading (MA): Addison-Wesley; 2008.
17. Pierce BC. Types and programming languages. Cambridge (MA): MIT Press; 2002.