Software Extensibility Strategies for Health and Demographic Systems in Low-Income Countries

Brian Hartsock, Bruce MacLeod, David Roberge and Ime Asangansi
Department of Computer Science
University of Southern Maine
Portland ME, United States
brian.hartsock@maine.edu

Abstract— **Software systems used by health research centers in low-income countries typically need to be maintained and upgraded by individuals who do not have formal Computer Science education. Without careful planning, these data management software systems can require years of technical assistance from highly trained specialists. This work proposes software design strategies to minimize the complexity of maintaining a Java based enterprise scale software application in low-income countries. In this paper we identify various extension mechanisms. The goal is to significantly reduce the complexity required for users to tailor the system to their own particular projects.**

Keywords; OpenHDS; HDSS; extensibility; extension points; modifiability; variability;

I. INTRODUCTION

There is an increasing use of open source software targeted for health centers in low-income countries. This movement has evolved through community cooperation and promotes the production of high quality software as well as being able to work cooperatively with other similarly minded people to improve open source health technologies. These health centers often employ data managers who have no formal Computer Science education. A typical profile of a data manager would have a high school education which may include a limited background in programming. Data managers often struggle with maintaining and upgrading the software system due to their lack of experience.

Health systems software needs to solve some relatively challenging problems. Security and privacy, multi-user access and communication with other government databases are among major concerns. Mobile phone support has been gaining interest since many health centers are transitioning from a paper based data collection process. Additionally, data entry is performed by inexperienced data entry clerks who can potentially make data unreliable. The data model of these software systems often requires many tables and may vary across project sites.

For many health studies, households need to be studied over a period of time. Household characteristics and member relationships can clarify the causes and consequences of mortality and illness in developing countries. Household analysis is also helpful in health interventions since the effectiveness of health services is determined by household social and behavioral factors. However, many health studies in developing countries fail to meet their objectives. Reasons for failure vary, but the most prominent among them include data management problems. Often these projects wish to integrate additional health information with demographic data which involves direct manipulation of the software system and realize that augmenting the system to manage this additional information presents challenges. Recording household data and member relationships is a complex undertaking when studied over a period of time because events of interests must be recorded and linked with other characteristics of individuals. This results in a large collection of data that becomes difficult to compile, manage and analyze [1].

Longitudinal household studies will vary across project sites and each site may have their own set of requirements on how a health and demographic surveillance (HDSS) application should behave. All sites share a common data model while some may wish to collect additional information which may include modifying the original data model. Our goal is to build a core HDSS that contains the minimum amount of fields necessary for data collection that is shared across all sites. Developing an initial data model and supporting software to implement such a system requires considerable resources and is error prone. The core HDSS must have appropriate techniques for realizing functional site-defined extensions requiring little to no custom software development. A careful, principled approach to extending the core system can provide significant benefit and can serve to better utilize available resources.

This paper identifies extension strategies that we have adopted in order to build an extensible HDSS application for health research centers. Data extensions refer to modifying the database in some form in order to collect additional data than what is provided in the core system. Behavioral extensions refer to modifying the underlying functionality of the software system. With these strategies of extensibility in place, it becomes easier for health centers to tailor the system to their particular projects.

II. BACKGROUND

We are currently developing the OpenHDS[1], an open-source web-enabled health and demographic software system. The application is currently being field tested at the Ifakara Health Institute in Tanzania and the Cross River State in Nigeria. Both sites are members of the INDEPTH Network[2], an

This project is supported by the generous contributions of the IDRC.

[1] http://openhds.rcg.usm.maine.edu/
[2] http://www.indepth-network.org/

978-0-7695-4595-0/11 $26.00 © 2011 IEEE
DOI 10.1109/GHTC.2011.56

international community-based longitudinal demographic surveillance initiative in resource constrained countries. Their mission is to provide better, empirical understanding of health and social issues and to apply this understanding to alleviate the most severe health and social challenges. Currently, INDEPTH consists of 42 HDSS sites located in 19 countries. The OpenHDS represents the next generation of HDSS applications and is regarded by INDEPTH to be the recommended software of choice for all of its members.

HDSS systems are primarily concerned with monitoring the general population in a study area. The computational foundation of the software system is a relational database that resolves many of the complex data management issues associated with monitoring births, deaths, relationships, and migrations in a fixed geographical region. Fig 1 demonstrates the goal of an HDSS.

There are many rules that must be followed to ensure that data is consistent and reliable. Events tied to individuals tend to be complicated when combined with all of the other population data. For example, a birth to a woman five months after she gave birth to another child is considered an invalid event. This is a sample of the many constraints that must be enforced in order to maintain data integrity. If these constraints are not enforced then it could lead to events taking place for individuals not included in the study area or allowing births to happen for women who are underage. In addition, errors generated at one stage tend to cause additional errors in later stages. This compounding effect can quickly cause the database to become completely out of sync with the study population. An HDSS is responsible for registering new individuals as they enter the study area and appropriately managing their events over time until a death or out-migration occurs, in which the individual leaves the study area [1].

The OpenHDS provides the core functionality for a health study to be successful and also consists of variation points in which extensions can be made. Extensions can be categorized and grouped into two areas as represented by Fig 2.

III. STRATEGIES FOR EXTENSIBILITIY

Extensibility is a system design principle where the implementation takes into consideration future growth. Extensions can be made through the addition of new functionality or through modification of existing functionality. Extensible systems are designed to include mechanisms for expanding/enhancing the system with new capabilities without having to make major changes to the system infrastructure [2].

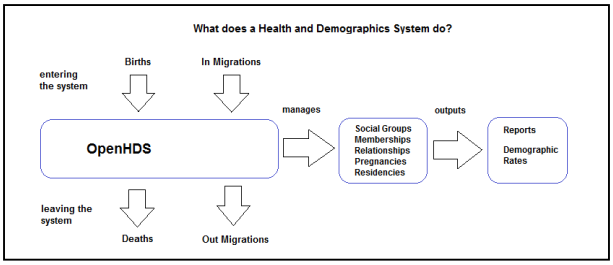

Figure 1. The goal of a Health and Demographics System.

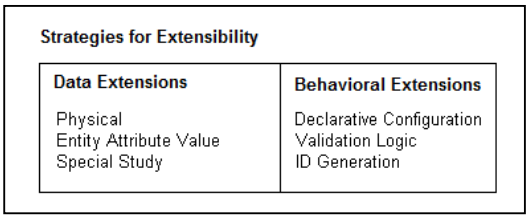

Figure 2. Grouping extensions into strategies.

A. *OpenHDS Architecture*

The architecture of the OpenHDS is built on the Spring Framework[1] which provides numerous advantages. Some of these advantages include the fact that Spring has a layered architecture and is modular. This means that parts of it can be used in isolation and the architecture remains internally consistent. It has rich support for aspect oriented programming (AOP)[2] which enables cohesive development by separating application business logic from system services such as auditing, logging and transaction management. Spring handles application configuration in a consistent way making use of Dependency Injection[3]. It's these advantages that help our extensibility goals to be met.

The OpenHDS is composed of a layered architecture which is a natural design to follow since there are many interactions between objects that take place from the user-interface to persistence. These interactions could then be categorized and placed into the appropriate layers which help to separate the system into more manageable components. Fig 3 demonstrates the process of how creating a new entity makes use of various layers within the OpenHDS architecture in a sequence diagram.

The Front Controller channels all user requests for the OpenHDS. The web forms layer is composed of multiple views for creating, editing, viewing, and listing each entity within the system. Java Server Faces[4] is the component oriented user interface framework that was selected to build the web forms layer.

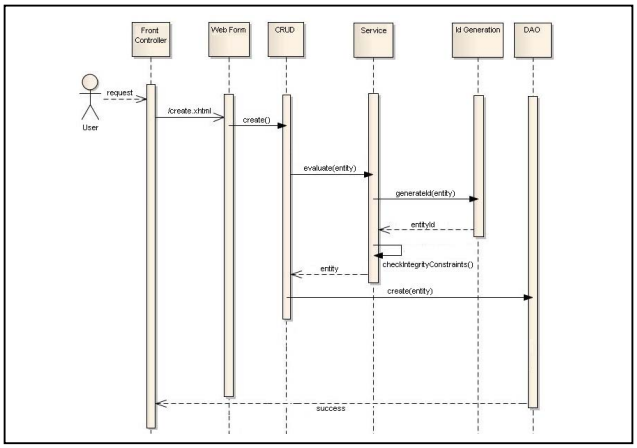

Figure 3. Sequence diagram demonstrating the use of layers in the OpenHDS architecture.

[1] http://www.springsource.org
[2] http://en.wikipedia.org/wiki/Aspect-oriented_programming
[3] http://en.wikipedia.org/wiki/Dependency_injection
[4] http://www.javaserverfaces.org/

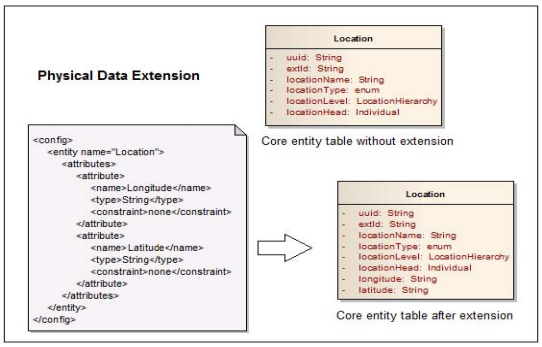

Figure 4. Physical Data Extension for the Location entity.

The Create, Read, Update, Delete (CRUD)[1] layer serves as a controller for the web forms. When a form has been submitted, the data is sent to the underlying CRUD class which handles certain functions such as navigating between pages, paging results, converting entities, messaging, and validation. The CRUD classes delegate to the Services layer to perform integrity constraints and for persisting objects with the normal create, read, update, and delete operations.

The Service layer contains unique functionally related to the underlying entity it represents since each entity may have particular integrity constraints that must be enforced. For example, before creating a new relationship for an individual, we must check that the individual doesn't have a recorded death because it doesn't make sense to create events for deceased individuals.

The Data Access Object (DAO)[2] layer consists of a mechanism to map the core entity objects to the underlying database. Its specific job is to perform any related query or transaction that the Service layer requests.

The OpenHDS architecture achieves conceptual integrity which refers to the architecture doing similar things in similar ways. This is important because it unifies the design of the system at all levels[2]. It's also important because if new functionality is to be added then it makes the job easier because there is a consistent and coherent design to the system. This includes not only how components are designed but also other factors such as coding style and variable naming. This architecture enables our extensibility strategies to be robust enough to handle site-specific requirements. The downside to a layered architecture can be seen whenever additional data is to be included such as a new field on an entity. A result of this small change often requires new additions in the CRUD, Service, and DAO layers as well.

B. Physical Data Extensions

Physical Data Extensions refer to the addition of new columns on the OpenHDS core entity tables. Extensions like this make sense if the numbers of attributes are small and if those attributes apply to a significant portion of the population. This helps to keep all data that belongs to a particular entity in one location. The OpenHDS core entity tables are mapped by Hibernate[3], an object-relational mapping tool to a corresponding relational database containing the fields that are commonly used across projects. In order to collect additional data, a project investigator could modify the Java source code for an entity by adding a new field. This isn't an ideal solution because our extensibility goals are to allow such a mechanism to be possible while minimizing the modification of source code.

The implementation of this mechanism is being handled by Javassist[4], a Java bytecode manipulator. If a site wishes to collect more data than what is already provided (e.g. longitude, latitude for location) then it can be specified in a configuration file and those fields will be added dynamically to the associated entity tables in the underlying database as soon as the project is recompiled. When the configuration file is read, Javassist will modify the corresponding classes. These modifications include adding new fields and methods to the target entities and the supported fields can be any of the primitive types available. Fig 4 demonstrates this situation.

In this example, longitude and latitude will be added to the location entity which will be reflected in the location table in the underlying database. The constraint element can be used to restrict the values that a particular attribute can take. In this case, 'none' is specified meaning that the values of longitude and latitude can take any particular String value. In practice, projects will also wish to restrict the values a field can take to a particular set, which is discussed later in Section F.

Currently, we do not have a mechanism for linking these dynamically generated fields to the web forms; it must be performed manually. To do this, it will require modifying source code. For those that don't have the requisite skill to add new columns manually, the Javassist solution is preferred.

C. Entity Attribute Value Extensions

In many cases, health centers that conduct longitudinal monitoring of a population would like to collect cross sectional data for a subset of the population. For this reason the Entity Attribute Value model (EAV)[5] cannot be ignored because it allows attributes to be added without changing the structure of the database. The EAV is a data model that supports extensions by providing a structure for storing extended table (or entity) attributes. The basic structure of the model can be identified by name. There is a table for the entity, a table for the attributes related to that entity, and a table for storing the values for the attributes.

A site may wish to record data during one round and some other data during the next round (e.g. shared water source in a location). Attributes can be customized (e.g. shared water source) and linked to the particular entity (e.g. location) during a particular round. Fig 5 demonstrates how the EAV is implemented.

The important aspect of this design is that the location entity table does not contain an extra column to hold the value of sharedWaterSource. This could technically be done with a physical data extension but it would clutter up the table since the Attribute defines that the sharedWaterSource value is only valid for round number 1. If this were handled as a physical data extension then for any round number other than 1, sharedWaterSource would be represented as null.

[1] http://en.wikipedia.org/wiki/Create,_read,_update_and_delete

[2] http://en.wikipedia.org/wiki/Data_access_object

[3] http://www.hibernate.org/

[4] http://www.csg.is.titech.ac.jp/~chiba/javassist/

[5] http://en.wikipedia.org/wiki/Entity-attribute-value_model

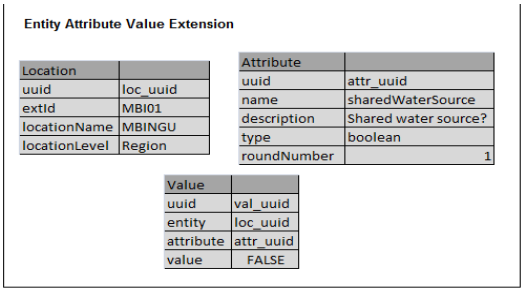

Figure 5. Entity Attribute Value Extension for collecting data on a shared water source in a Location.

Both physical extensions and EAV extensions have strengths and weaknesses. The EAV presents problems in data integrity and extraction, performance, and system comprehension. Queries across multiple tables must be performed to obtain the value for a particular entity attribute. Combined with all of the other demographic data, this can become a complex task.

D. Special Study Extensions

Special Studies often take the form of questionnaires and can be potentially 50+ fields long. These questionnaires can be specific to any of the core entities of the system and might only be collected once or a few times per year. These questionnaires could include things such as:

- What material are the walls of the household made of?
- Approximately how many meals has your family missed over the past 30 days?
- How many rooms in the household are used for sleeping?

It doesn't make sense to make these questionnaires physical data extensions since the data is to be collected a few times each year. It also doesn't make sense to put it into the EAV because normally the questions for the Special Study are asked for the entire population. It will also make data extraction difficult and it will clutter the tables in which the values are stored. It's more reasonable to handle this additional data in a separate application and maintain a separate database in which this data can be stored and accessed. There must be a mechanism for questionnaires to be modified and swapped out with ease as well.

We designed a separate application to handle Special Studies. The Special Study application is not part of the core system but is instead treated as a separate component and is only meant to be used if custom questionnaires are to be defined. This application is built using the same underlying technology that the OpenHDS is built on with a few exceptions. It's using a framework called Spring Roo[1] which has proven to be helpful because of its ability to auto-generate web forms based on customized data fields. Data fields can be defined and Roo will automatically generate code to provide a presentable web form in which that extra data can be recorded. This helps to reduce the complexity required for investigators to build their own questionnaires.

In order to create a Special Study, the entity ids must be linked to ensure referential integrity between both databases. Since the underlying database between the Special Study application and the OpenHDS core system are separate components, a Special Study service was built to perform the necessary linkages. This service has functionality to check if the entered id from the Special Study application is valid. If it is valid, then the Special Study record is created and stored in its own database. If it's not, then an appropriate error message is returned and displayed on the Special Study web form.

Special Studies help to achieve semantic coherence which refers to grouping particular areas of functionality together [2]. The goal is to ensure that the Special Study application and the core system work together without excessive reliance on one another. The extra fields from the questionnaires don't belong in the core HDSS, which is why the Special Studies are not considered to be part of the core system.

Physical extensions, entity attribute value extensions, and special study extensions are all focused on data management and help reduce the errors that investigators experience when attempting to configure the OpenHDS according to their project specific requirements. The core entity tables will no longer be polluted since each piece of data can be grouped into one of the three categories. This helps to keep the OpenHDS core system entities in tact while allowing extensions to be made. Each type of data management extension has a clear separation of responsibilities.

E. Declarative Configuration

Declarative Configuration refers to configuring and modifying overall system behavior and can be set in a configuration file which serves as a container for the global attributes of the application. The properties that can be set are often site-specific and the most common ones configured in the core system include the following:

- maleCode=M - Code for male
- femaleCode=F - Code for female
- unknownCode=UNK - Code to indicate an unknown value
- birthCode=BIR - Code to indicate a birth
- deathCode=DTH - Code to indicate a death

Codes will need to be defined since project sites will usually have their own process for how data is to be collected and stored. For example, instead of storing a gender value as Male or Female, a site may wish to store it as 'M' or 'F', or even '1' or '2' because those are the values that may be collected in the field. In practice, the data that's collected is often abbreviated for simplicity. The problem is that there is a mismatch between what is collected in the field and what the system expects for input. We call this mechanism Value Substitution which can be used to map these codes into the core system and persist them in a way that represents the project site's data collection process. There are currently over 40 codes that can be configured in the OpenHDS.

[1] http://www.springsource.org/roo

Web Form Substitution is helpful if there are multiple web form implementations. Many sites may have preferences in how data is to be entered and viewed. For example, there are currently multiple implementations of the Baseline form which captures all initial enumeration data for households. It's a complicated form and follows a strict control flow. One implementation consolidates all data entry onto one form and the other implementation delegates to many other forms. There are also multiple forms for all of the core entities which allow for creating, viewing, and editing. These can all be swapped out if the project investigator wishes to use their own forms.

There are many other miscellaneous options that can be specified in configuration as well which can adjust overall system behavior. Translated language files can be defined and swapped out for international language support. Different sites may also have a preference of which date format should be used for data entry. There are many configuration options used internally for running the OpenHDS web services, the look and feel of web forms, database support, and more.

F. Validation Logic

The OpenHDS core system contains the basic rules for ensuring that data is consistent and reliable. Extensible validation logic appears in two forms:

1) Field Validation:

These are constraints that pertain to the fields of an entity which take the form of annotations and are performed automatically by Hibernate before the data is persisted. They're helpful to ensure that fields take on the appropriate values. For example, imagine that there is a field on the location entity for specifying the locationType, which can be either RUR or URB. The field can be annotated like the following:

```
@ExtConstraint(id="locationTypeConstraint")
String locationType;
```

The constraint attribute is specified with an id of a particular configuration defined in a separate xml file which can be seen in Fig 6. Once the web form is submitted, the associated values for all fields on the entity will be validated. If a field has a `@ExtConstraint` annotation, the xml file will be checked to ensure that the field contains an appropriate value.

This mechanism can also be applied to physical data extensions. If locationType was going to be added in a dynamic way similarly to Fig 4, all that would be required is to add 'locationTypeConstraint' as the constraint element value. This has advantages because it allows a project to define their own fields for particular entities but also to restrict their values so that the data collected is reliable.

```
<value-constraints>
    <constraint id="locationTypeConstraint">
        <value description="Rural">RUR</value>
        <value description="Urban">URB</value>
    </constraint>
    ...
</value-constraints>
```

Figure 6. Constraint to ensure locationType takes on an appropriate value.

Field level validation follows the JSR 303 Bean Validation[1] specification.

2) Integrity Validation:

This type of validation is specific for the entity that it represents and requires a database query while the other type of validation doesn't. For example, when defining a new household, the household head must not be a deceased individual. Nothing would stop the user from entering malformed data so it's our responsibility to ensure that this doesn't happen. If both field level and integrity validation rules are successful, the entity is ready for persistence.

Validation Logic is directly tied to Declarative Configuration since custom behavior can be defined for the constraints that are in place from within a configuration file. Project sites may place various restrictions on the data that's collected. For example, one site may wish to only allow marriages if both individuals are over 16 in age while another may only allow marriages if both individuals are older than 12. The most common rules that can be adjusted include the following:

- The earliest date in which an individual can be enumerated into the system
- Minimum age of parenthood
- Minimum age of household head
- Minimum age for marital relationship

There are currently over 15 validation rules that can be configured and enforced dynamically in the OpenHDS.

G. ID Generation

Certain parts of the application need configurability support that cannot be completely specified in advance. The Identification codes (Ids) for the key entities is one example. These are central to the health center operations and there are many different variations with different levels of logic that are required. To solve this extensibility challenge, the OpenHDS uses an interface for the characteristics of Ids and various implementations can be used to satisfy this interface. The implementation is handled by the Template Pattern[2] and Open-Closed Principle[3]. It's common that different sites will have their own ways of how ids are formed and it's more beneficial to the user if they are formed in a manner that's predictable and have some meaning rather than just a string of random alphanumeric characters.

The OpenHDS has a pluggable id mechanism and can be defined from a configuration file. These ids refer to the core entities of the system: location, individual, social group, and visit. There are many ways to customize these ids depending on the entity. The basic forms of an id are as follows:

[Prefix][FieldNames ...] [Increment Digit][Check Char]

[1] http://www.jcp.org/en/jsr/summary

[2] http://en.wikipedia.org/wiki/Template_method_pattern

[3] http://en.wikipedia.org/wiki/Open/closed_principle

The prefix can take the form of any String containing alpha characters. FieldNames represent actual fields from the entity and are usually dependent on the particular entity that it's for. The Increment Digit is a required part of the id because it ensures that each id is unique. This cannot be set as a parameter and is used internally by the system when forming the id. The Check Char is represented by an alphanumeric character that is appended to the end of an id which is generated by Luhn's Mod N Algorithm[1]. The algorithm ensures that a set of characters will have a particular check character at the very end which helps to eliminate the possibility that a user has mistyped an id. A site is not constrained to use the built in id generation templates and could enter them manually if they choose.

An example id for generating a new location entity could be something such as the following: LOCMBI1Q. In this example, 'LOC' is defined as a prefix. The string 'MBI' refers to a field name on the location entity table. The '1' refers to the increment digit. In this case, it's the first location being created with a prefix of 'LOCMBI'. The 'Q' represents the check character that is generated from the string 'LOCMBI1'.

IV. FUTURE WORK

Module Extensions are currently in active research and represent the different components of the OpenHDS which are not required in order for the application to function. We envision the OpenHDS as the core system consisting of many optional modules and in our research; we have been exploring Spring Dynamic Modules for OSGi platform[2]. It's a module system and service platform that implements a complete and dynamic component model. It follows a runtime registration strategy where components can be remotely installed, started, stopped, updated, and installed without requiring having to reboot [2]. A modular solution would provide better support for Special Studies, especially with providing the necessary linkage for referencing ids between databases.

We envision an HDSS application that is divided into several pieces with the connections between them controlled and well defined. This not only helps to organize and group functionality, but it's also important in the overall system architecture in achieving conceptual integrity.

Continued investigation is required to implement the manual steps that must be performed when applying Physical Data Extensions. We will also be developing support for integrating data from mobile data collection systems based on xForms[3] into the OpenHDS. This is a high priority since health centers are transitioning from paper based data collection to mobile phone support.

V. CONCLUSION

Our goal was to build a core HDSS that is general enough so that extensions can be made which allow sites to tailor the system to their own project requirements. The strategies of extensions identified were grouped into two categories: data and behavioral extensions.

Data extensions refer to the different mechanisms in which data can be stored and retrieved. There are three variations of data extensions including physical extensions, entity attribute value extensions and special studies. Physical data extensions are used to dynamically add additional fields into the core entity tables and are only recommended if the data applies to a significant portion of the population. Entity attribute value extensions are another mechanism to support data that is only collected for a subset of the population. Special Studies often take the form of questionnaires that can be potentially 50+ fields long and are collected a few times each year. A distinct Special Study application was created to separate the additional data from the core demographic data into two managed databases.

Behavioral extensions refer to altering overall system behavior in some way and three main areas were explored. Declarative Configuration is a mechanism to define all site specific parameters. It can also be used to modify overall system behavior such as the validation rules that are applied to entities, the look and feel of web forms, international language support, web form substitution, and more. Validation logic plays a role in keeping data consistent and reliable which is enforced dynamically from a configuration file. A site may also have a preferred format for how their ids are to be generated.

With these strategies identified and currently being tested in the development of the OpenHDS, we believe that the process of adapting and maintaining the system will be reduced. We have developed various mechanisms for extending the system so that different sites will be able to configure the system to their own project specific requirements.

REFERENCES

[1] B. MacLeod, J. Phillips and B. Pence, "The Household Registration System: Computer Software for the Rapid Dissemination of Demographic Surveillence Systems," Max-Planck-Gesellschaft, Demographic Research vol. 2, A6 June 2000.

[2] L. Bass, P. Clements and R. Kazman, Software Architecture in Practice: Second Edition. Pearson Education, MA: Carnegie Mellon Software Engineering Institute, 2003.

[3] K. Henttonen, M. Matinlassi, E. Niemela and T. Kanstren, "Integratibility and Extensibility Evaluation from Software Architectural Models – A Case Study," Bentham Science Publishers Ltd, The Open Software Engineering Journal, 1, 1-20, 2007.

[4] M. Fowler, Patterns of Enterprise Architecture. Pearson Education, MA, 2003.

[5] P. Clements, R.Kazman, M. Klein, D. Devesh, S. Reddy, and P. Verma, "The duties, skills, and knowledge of software architects", in The Working IEEE/IFIP Conference on Software Architecture (WICSA), 2007.

[6] F. Bachmann and L. Bass, "Introduction to the Attribute Driven Design Method," in the International Conference on Software Engineering (ICSE), 2001.

[7] I. Crnkovic, "Component-based Software Engineering – New Challenges in Software Development", in the Information Technology Interfaces (ITI), 2003.

[8] T. Kelly, "Using Software architecture Techniques to Support the Modular Certification of Safety-Critical Systems," in the Australian Workshop on Safety-Related Programmable Systems (SCS), 2006.

[1] http://en.wikipedia.org/wiki/Luhn_mod_N_algorithm

[2] http://www.springsource.org/osgi

[3] http://www.w3.org/MarkUp/Forms/