Designing Flexible Business Information System for Modern-Day Business Requirement Changes

Asma Al kalbani  
Department of Information Technology, Ibri College of Applied Science, Sultanate of Oman  
asmam.ibr@cas.edu.om

Kinh Nguyen  
Department of Computer Science and Computer Engineering, La Trobe University, Australia  
kinh.nguyen@latrobe.edu.au

Abstract—In today’s competitive business environment, business requirements often experience changes at a rapid pace. Coping with requirements changes has long been recognized to be a major problem. The challenge is how to make business information systems flexible so that they can support not only current requirements but also a range of future ones. This paper presents a number of techniques to introduce such desirable built-in flexibility to information system designs. As an illustration, the techniques are applied to the design of a product catalog database with multi-language support.

Keywords—Information system; changes of requirements; stable information structure; multi-language support

I. INTRODUCTION

The lack of flexibility in business information systems to cope with changes in business requirements has long been recognized as a major problem that can incur a huge cost. The challenge is how to design information systems with inherent flexibility to enable them to best adapt to business requirements changes.

The process of adapting a system after it has been delivered and being utilized is called software maintenance [1]. One of the most difficult and costly problems of maintenance is to change or modify the system due to a change of requirements. In 1980, Lientz and Swanson conducted a survey and concluded that about 17% of maintenance was corrective, 18% adaptive and 65% perfective, as shown in figure 1. Perfective maintenance aims at improving efficiency and maintainability. It is also found that the costs of adding functionality to a system after it has been put into operation are usually much greater than providing similar functionality when software is originally developed.

Figure 1. Costs by Maintenance Types

Almost any organization changes regularly over time, complete knowledge of a system's requirements, in the traditional sense, is necessarily imperfect. Some requirements appear in the future and they are unknowable at the time the information system is designed or built. The traditional objective of system design is functional accuracy. But when designers do not look beyond the current requirements, this often results in the emergence of inflexible information systems. Unless we take special effort to deal with potential changes, our system will tend to be inflexible and cannot cope well with the required changes; and the need for perfective maintenance is increased accordingly.

Since perfective maintenance becomes a serious problem in industry, there is an urgent need for a flexible information system [3]. Our main objectives, for this paper, are as follows:

First, we aim to study the problem of inflexible information systems carefully to clarify: (a) What exactly is meant by inflexibility, and (b) To what extent can inflexibility be avoided, and how.

Second, we aim to carry out a comprehensive review of past research work, especially researches that try to deal with the problem by designing information structure with built-in flexibility.

Third, we examine in detail the existing flexible information system design approach by Bruce, Walter, Robert and Cindy (Bruce Group Approach) [9]. We apply their technique in a case study (Database-Driven product Catalog System) and carefully evaluate the resulting design. We suggest a flexible design solution for the product line of Product Catalogue that requires multi-language support.

Finally, we present our conclusion and indicate the direction where future work can be conducted.

II. ANALYSIS OF THE MAIN SOURCE OF THE PROBLEM

The essence of adaptability to change lies in information system data structure. Usually if the requirements change, the data structure design needs to be changed as well. Therefore, the source of problems lies on data structure design. The main problems (or mistakes) of information structure design that lead to inflexibility in information systems are:

A. Mistake in design identifiers
Identifiers are the names given to things, and naming conventions are the rules devised to assign names to things [2]. The serious mistake with the design of identifiers is that they are unstable. The unstable identifiers are responsible for a significant share of the maintenance burden. Identifiers are unstable when they contain information, because virtually any information about the thing identified is subject to change.

B. Weakness in design entities

Entity is defined as a thing which is recognized as being capable of an independent existence and which can be uniquely identified. Weakness in design entity occurs when the definitions of entities are too specific. It is a weakness because any change in the system, such as adding a new entity, leads to change in the information structure.

C. The weakness in design relationships

Relationships are connectivity between entities. There are three type of connectivity; one to one (1:1), one to many (1:M), and many to many (m:n). Weakness in design relationships occurs when the cardinality between the relationships is too specific. It is a weakness because specifying the cardinality between the relationships, for example, as one to one in the current requirements causes the problem if there is a need for change in future.

D. Weakness in business rules supports

Business rules are the control of the real-world. Business rules can change to reflect changing business needs and opportunities. At any given time, the model reflects only the current state of these business rules, whereas the real-world system allows the entire range, or at least a significant portion, of states. Weakness in business rule support occurs when the design can support the state according to the current business rules but fail to a range of states allowed by changes in business rules.

III. RELATED WORKS

Many professionals in the IT field recognize that there are problems with inflexible information systems. Unfortunately it appears that research devoted to this issue has been rare and insufficient. Generally, people do not discuss the problem in specific and productive way. Most would describe various aspects of software evolution and leave it at that.

The work by (those we refer to as) Bruce Group is an exception to this “rule”. According to this group, “The flexible system is defined here as an automated system that can be resynchronized with changes in the real world system through user-based data value modifications, without information restructuring or program code modification” [3]

Information structure can become unstable (i.e. undergoing numerous changes) when the system requirement changes. A number of factors can affect the stability of the information structure. For example, the relationship between the entity and their constraints may change. These kinds of changes are related to the problems presented in Section II.

Bruce Group suggests the solution for these problems by what can be called Flexible Domain Representation [25]. Essentially, the Flexible Domain Representation approach takes the form of information structures composed of generic entity types and m:n relationships (i.e. least constrained relationship and the least specialized entities).

This form of information structure can support diverse data manipulation processes without modification (and the approach has been successfully employed for commercial systems [2]). Because an entity is a person, place, thing, concept, etc., the idea of generic entity is to recognize that such things come in subtypes. For example, a person entity can have subtypes of employee, student, dependent, client, etc. the new subtype is added to the generic entity without a need for change in the information structure. Similarly, wherever we have an m:n relationship, it can accommodate a corresponding m:n or 1:n or 1:1 relationship [3].

In addition, Bruce Group proposes a flexible approach to support regulating business rules, which can be called General Regulation of Business Rules.

General regulation of business rules include two main general techniques: (a) Identifying entities and rules and potential changes, (b) Validation rules as data, and (c) Stable identifiers.

Identifying Entities and Rules and Potential Changes.

The simple example below (from [2]) illustrates how an analysis can be carried out to help us identify potential system entities and business rules potential changes. Suppose we have the following explanation about the admission process at a university:

An applicant submits an application, indicating the term and major. We review the applicant’s test scores, high school transcript, etc. If the applicant meets the criteria for that major, we send an offer letter. If not, we send a denial letter, suggesting alternatives.

In the above description, the nouns and adjectives are underscored. Where adjectives are encountered, the potential for entity subtypes exists. For example, we note that the description distinguishes offer letters from denial letters. Therefore, we should explore the possibility of a generic data structure for letters, which provided for diverse types of letters. As another example, the term “high school transcript” was specified, this leads us to consider what other type of transcript might exist [2].

Validation Rules as Data. The developer of a flexible system should think in terms of a design that allows a new logical table to be added to the system without a change in information structure. The term “validation rule” is used to illustrate a basic form of business rule that verifies whether or not a value entered into a field is valid. Such rules vary from validation of a single value against a list of values to validation of complex combinations of values and multiple files and operators.

For simple validation of a value against a list of values, the addition of a table storing valid values is a simple solution may work well. However, the developer of flexible
information system should provide a design that allows the user to logically create new table for validation purpose. For example, instead of having separate tables for state and country, we should have only one table with a column “type” that can take value such as “state” or “country”. Then if we need to validate ethnic, the user can simply enter a number of rows, one for each valid value of ethnic.

**Stable identifiers.** The inclination to use identifiers that carry information is probably wired into our brains. But this can be a source of mistakes that leads to unstable identifiers, which can be a major factor that leads to instability in the information structure [4]. For example, a company uniquely identifies one of its products as P6DHIL225, which is encoded with information about product.

P6 = production plant = Peoria
DH = product type = heavy-duty detergent
L = material type = liquid
225 = sequence number within product type

If the production of the item was moved to the Kansas City plant, the product ID would be subject to change. In contrast, if the product is uniquely identified as, 273758, with no encoded information, then nothing could change that would require the ID to be changed.

**CASE STUDY: ILLUSTRATION AND EVALUATION**

In this section, we present a case study, which was originally published in [5], together with our analysis and discussion. In particular, we will carefully examine the case study from the viewpoint of Bruce Group. We will also suggest a flexible general solution for the product line of catalogue systems that require multi-language support.

The case study is called Database-Driven Product Catalog system (DDPCS). It is a web-based product catalog that enables companies to extend their service to customers and other businesses over the Internet. Customers expect to be able to browse product catalogs. Catalogs come in many forms and present a variety of products in different formats. A special requirement is that the database should support all products regardless of what attributes the products may have. The attributes may include media objects such as pictures, video clips, and audio samples.

In [5], a database design is proposed for storing product information. We will evaluate this design, identifying its strengths and weaknesses, and propose a new design. We will be concerned primarily with the logical design of the database, and temporarily ignore other issues such as performance.

In this database, we have *products* such as shoes, cameras, and equipment, and each product belongs to one or more *categories*. For example, products CCC3 and CCC4 may belong to two categories Nike and shoes, while product CCC5 may belong to one category Dunlop. Each product maintains a number of *product attributes* such as product Id and product name, color, size, material. Some of the product attributes may be of *media types* such as pictures and video. In addition, each category has a number of *category attributes* such as category Id, category name, logo, contact address, staff and supplier name.

**A. A Logical Model for DDPCS**

In [5], a logical model for the database of the DDPCS, shown in figure 2, is first given.

**Figure 2. DDPCS Logical Model**

The model presents five entity types and four relationship types. *Category* represents the categories (e.g. shoe). *Category Information* represents category attributes. *Product* represents products (e.g. CCC1, which is a pair of shoes). *Product Feature* represents product attributes that are not media type (e.g. product name). And *Product Media* represents product attributes that are of media type (pictures, audio scripts, etc.).

**B. A Physical Model**

Next, in [5], a physical model derived from the logical model is given.

The first part of database design is the *Category* table. This table specifically identifies each category separate from the category details.

The second table is the *Category Features* table (corresponding to entity type Category Information). This is where the features (attributes) of each category are stored. A category feature is shared by all products belonging to the category.

The third table is the *Product Features* table, which stores the non-media attributes that make up each of the products. It is crucial to note that an attribute, such as ‘size’ and ’10’, can be used by many different products. In other words, *this attribute is never repeated*. Any new product that needs an attribute of ’size’ and ’10’, can just refer to this attribute by the unique attribute identifier.

The last table is the *Product Media* table, which stores media attributes of the products. This table contains all of the individual media data for the product. It uses a different model to that of the *Product Features* table. Each row contains information about a media object relevant to a product. The type of media may include pictures, audio, video, or other animation. The media data column is where the media object is stored. Since the raw data is being stored for the media object, a short description, called “caption”, is provided.

Details about the products (e.g. product id, product name) and their links to categories and product attributes (features and media) are implemented with the clustering
technique. With this technique, if we have the same primary key for different tables, the primary key of the whole table is specified as the cluster key. This key will be the one that groups the part of tables together. Physically, this key is stored only once, and connected to it will be all the part records that are associated with it [6]. In the product database, the cluster key will be the product key.

Applying this technique, two clusters are created. The first cluster is Category List table, which contains product ID (cluster key) and category references that links a product to the categories it belongs to. The second cluster is the Product Parts table created to hold all the parts of the product. This table contains product ID (cluster key), part ID and parts type (from media and features table).

Finally, we have the Product table, which basically contains two clustered tables (Category List and Product Parts) and a primary key.

C. Issues Regarding the DDPCS Logical Model

The main issue for DDPCS is that the logical design model is unclear and it is difficult to understand. There is no example given in the logical model. The design presents each entity without giving any details or attributes. Therefore, it is an incomplete logical model and the user goes straight to the physical model and gives detailed reference to the physical model. Moreover, the other issue in this design model is the relationship between attribute of product and category. In fact, there is no relationship between them. The idea behind relationship between product attributes and category is unclear in the original paper. In addition, even the original paper proposed that the data redundancy is reduced and no restructuring is required, since the attributes are never repeated. However, data in the original paper is not consistent and is sometimes reuse and sometimes not. For example, attributes color ‘Black’ in Product Features. To further clarify, we will present the table here again and the repeating attributes will be highlighted with shading.

Also, The DDPCS does not have the product Name for each product. Each product just has a product ID and product feature which is unclear as well because each product should have a name.

The DDPCS logical model is well-designed (as will be seen) but not well-explained. Most readers would have much difficulty in understanding it. The reader need examples to understand the logical model diagram but unfortunately the example is not given in the logical model. The entire examples are given in terms of the physical model (which is very hard to follow), not in terms of the logical model. The entire examples are given in physical model. As a result we have to examine the physical model in detail and examine the sample data given in terms of the physical mode in order to understand the logical model.

Ideally, the logical should have included attributes for each of the five entity types. And then examples should be given in terms of the logical model, completed with attributes, for example as shown in figure 3.

Category

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAA1</td>
<td>Nike</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Category Information ID</th>
<th>Category ID</th>
<th>Attributes Description</th>
<th>Attributes value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BBB3</td>
<td>AAA1</td>
<td>Logo</td>
<td>Swoosh</td>
</tr>
</tbody>
</table>

Figure 3. The Logical Model with Sample Data

D. A Clearer Description of the Logical Design for DDPCS

The clearer logical model for the database of DDPCS, with attributes, is given in figure 4. From the diagram, we can easily see the entity and the relationships.

The model presents the following entity types:

- **Category**: Contains categories information including category ID and category name.
- **Category Attribute**: We have in this entity generic attributes such as attribute description (specific information description) and attribute value (specific information value) and unique ID for each unique attribute (Main ID). The attribute is a unique value so it is never repeated. No extra columns or tables are created to account for different attributes. When a new category attributes needs to be entered a new row is created and the information is entered.
- **Product**: Contains product ID as the primary key and product name. Also, we can add some generic information if necessary.
- **Product Attribute**: We have in this entity generic attribute such as attribute description (specific information description) and attribute value (specific information value) and unique ID for each unique attribute (Attribute ID). The attribute is a unique value so it is never repeated. No extra columns or tables are created to account for different attributes. When new product attributes need to be entered a new row is created and the information is entered.
- **Media Attribute**: We have in this entity generic Media Attributes such as attribute description (specific information description) and attribute value (specific information value) and unique ID for each unique attribute (Media ID). The attribute is a unique value so it is never repeated. No extra columns or tables are created to account for different attributes. When new product media attributes need to be entered a new row is created and the information is entered.
Figure 4. A Clearer Model for DDPCS

Also, as shown in the diagram, the logical model presents four relationships, of all which are many-to-many relationships.

Note on a simple relational physical model. For the purpose of validation, the design should be implemented. The implementation provides an executable version that we can run and test. The preferred physical model is one that is simple. This means we should choose the straightforward relational schema – without interfering features such as clustering, for example (note that we are not interested in performance at this stage). Such a straightforward set of tables can be easily derived. We would have one table for each entity type. As for relationships, because all the relationships are many to many, we would have a table for each of the relationship, and they are as follows:

- Category/Category Attribute: Includes category ID and Main ID (category Attribute ID).
- Category/Product: Includes category ID and Product ID.
- Product/Attribute: Includes product ID and Media ID.

E. Flexibility of the Design Model

It is clear that the proposed model is very flexible. With the given information structure, we can add or delete categories, category attributes, product and product attributes – without any change to be made to the underlying structure. This is consistent with the suggestions covered in Sections II and III.

IV. V. ADDING MULTI-LANGUAGE SUPPORT

A. Multi-Language Support Model 1

To add multi-language support, [5] produces the model shown in figure 5. We will refer to it as MLS Model 1.

This model has a number of serious defects. First of all, it is not clear what is meant by Language Support. What are the attributes of this entity type? Second, there are no indications for multiplicities. This adds to the difficulty of interpreting the model.

As for the case of the original model without multi-language support, no examples are given for this model in terms of the logical model. Consequently, one needs to examine, in much detail, the physical model and the examples given in terms of this physical model to make sense of the logical model.

After such work, the logical model can be interpreted as shown in the following sample. Suppose in the English-only model has this sample data:

<table>
<thead>
<tr>
<th>Att_ID</th>
<th>Att Desc</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX1</td>
<td>Colour</td>
<td>Black</td>
</tr>
</tbody>
</table>

Then in the Multi-Language Model I, we would have the sample data:

- **(Multi-Language Support Model I)**

<table>
<thead>
<tr>
<th>Att_ID</th>
<th>Att Desc</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX1</td>
<td>MULT120</td>
<td>MULT125</td>
</tr>
</tbody>
</table>

(Langage Support table)

<table>
<thead>
<tr>
<th>Attribute_ID</th>
<th>Language</th>
<th>Attribute_Value</th>
</tr>
</thead>
</table>
The sample data shows clearly that Attribute_ID is not an attribute ID at all but it can be an Att_Desc (attribute description, e.g. MULT120) or a value (an attribute value, e.g. MULT125. Thus, things such as MULT120 and MULT125 are simply identifiers of language term. For example, the language term MULT120 in English would be “Colour”, in Italian “Colore”. Given that nature, it would be much better to call the column “Term_ID” instead of “Attribute_ID”.

That is one problem. The second problem, which is more serious, is that the former English-only model is no longer a proper part of this new model. To see this point, consider the sample data above. The Product Feature table in the English-only version and the Multi-Language version have the same structure (column name), but the data is totally different. While in the first version we have a row with “Colour” and “Black”, in the second version we have “MULT120” and “MULT125.”

B. Multi-Language Support Model II

To overcome the problems of Model I, we propose a new model, shown in figure 6. We will refer to it as Model II.

With this model, the previous sample data would be as follows:

<table>
<thead>
<tr>
<th>Pro_Att_ID</th>
<th>Att_Desc</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX1</td>
<td>Colour</td>
<td>Black</td>
</tr>
</tbody>
</table>

Figure 6. Multi-Language Support Model II

The last table above corresponds to the association between Language Support and Product Attribute (shown in the diagram).

We can observe that Model II is a much clearer model in that we do not have to introduce new arbitrary identifiers for language term such as MULT120, MULT125 that we have seen in Model I.

Another very feature of this Model II is that the English-Only Model is a proper subset of this new model. For example, the table Product Attribute, in the new model, remains exactly the same (same columns, same row values). By retaining the English-only model, we have designated English as privilege language. Thus, Model II consists of

- Tables in the English-only model
- Language Support table. This table has one row for each non-English language supported by the database
- Five tables representing the five associations between Language Support entity type and the five entity types in the English-only model (see figure 6)

Due to it flexible and rich features, we propose this Model II as a general solution for the product line of Product Catalog.

V. FURTHER PROPOSAL TO ENHANCE FLEXIBILITY

Section III proposes a number of techniques to introduce flexibility to the design. In this section, we propose another technique to enhance flexibility. This feature is analogous to the technique Validation Rules as Data, which allows the user to enter a new type of validation with its associated valid data. The technique proposed in this section would allow the user to enter a relationship type by entering some appropriate data.

To do that, we will create a table that contains the following attributes: relation ID and relation degree. Relational ID is a unique ID for each relationship type and degree is a degree of the relationship (2 for binary, 3 for ternary).

We will then create another table that has the following attributes: Relation ID, role, entity type, minimum bound and maximum bound. Entity type is the entity that plays the role in the relationship, and minimum and maximum bounds specify in multiplicities associated with the role in sense defined by UML.

Those two tables allow the user to define a new relationship type. To store relationship instances, we create
a third table, which has three attributes: relation instance ID, relation ID (the same one as in the first and second table), role (as in second table), and entity ID (id of the instance playing the role).

This technique, we can observe, is an extension of the technique to design generic structure for the organization of an enterprise so that it can accommodate changes to the enterprise’s hierarchical structure.

VI. CONCLUSION & FUTURE WORK

In this paper, we have presented a critical survey regarding the cost of inflexible information system and how to design information systems with built-in flexibility to cope with a range of changes.

We noted that though many researchers describe the problem and examine the costs that inflexible design can incur, very few propose practical techniques to deal with the problem. One exception, we noted, is the work by the group that we refer to as the Bruce Group. We covered design problems/mistakes and techniques to overcome them in Sections II and III. We proposed a further technique in Section VI.

We considered the Product Catalog case study presented in [5], and observed that the design for the English-only version is flexible and exhibits features consistent with the ideas proposed in Sections II and III. Thus, this case study has illustrated the usefulness of flexible design, in particular they use of generic entity and relations.

In contrast, we showed that the Multi-Language Support model presented in [5] has a number of undesirable features; and we suggested a better alternative model. This catalogue case study is important in it is own right, and the proposed Multi-Language Support Model II can serve as a general model for the product line of Product Catalog that requires multi-language support.

For future work, we plan to study: (a) How business rules can be related to use case and how to exploit this relationship in designing and maintaining flexible information systems; (b) How refactoring technique can be applied for the same purpose; and (c) How we may combine the flexible design techniques in this paper with the results obtained the study of (a) and (b).

ACKNOWLEDGEMENT

The first author is grateful to her family and would like to express her thanks to her uncle Saud and her aunt Dr. Salma who have so freely given their time in support and advices.

REFERENCES


