2. Literature Review

This chapter is organized as follows. Section 2.1 discusses the characteristics and classifications of BISs. Section 2.2 discusses legacy BISs. Section 2.3 discusses the deployment problem of BISs and its solutions. Section 2.4 discusses Visual Basic (VB) -like visual programming. Section 2.5 discusses the relationship between the contents of this chapter and the study presented in this paper.

2.1 Business Information Systems

A BIS can be defined as an information system, which integrates information technology, people, and business, by which data are received, stored, computed, and converted into information to suit some business purposes (Inderscience, 2007; Laudon & Laudon, 2005; Martin et al., 2004; Mosley et al., 1997). With the increasing complexity of business and the growing need for information, BISs are more important than ever (Larson et al., 2005).

2.1.1 Fundamental Business Information System Principles

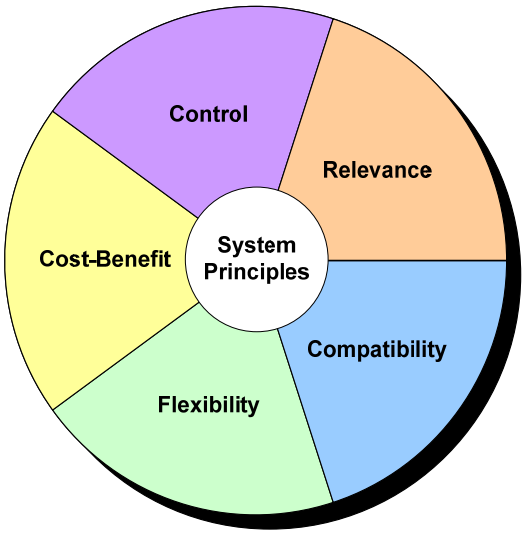

Figure 1. Business Information System Principles
Source: Larson et al. (2005)

The five basic principles of BISs are shown in figure 1 and as follows (Larson et al., 2005).

1. Control Principle – managers need to control and monitor business activities. The control principle prescribes that a BIS has internal controls. Internal controls are methods and procedures allowing managers to control and monitor business activities. They include policies to direct operations toward common goals, procedures to ensure reliable management reports, safeguards to protect company assets, and methods to achieve compliance with laws and regulations.

2. Relevance Principle – decision makers need relevant information to make informed decisions. The relevance principle prescribes that a BIS

reports useful, understandable, timely, and pertinent information for effective decision-making. The system must be designed to capture data that make a difference in decisions. To ensure this, we must consider all decision makers when identifying relevant information for disclosure.

3. Compatibility Principle – BISs must be consistent with the aims of a company. The compatibility principle prescribes that a BIS conforms to a company's activities, personnel, and structure. It also must adapt to a company's unique characteristics. The system must not be intrusive but must work in harmony with and be driven by company goals.

4. Flexibility Principle – BISs must be able to adjust to changes. The flexibility principle prescribes that a BIS be able to adapt to changes in the company, business environment, and needs of decisions makers. Technological advances, competitive pressures, consumer tastes, regulations, and company activities constantly evolve. A BIS must be designed to adapt to these changes.

5. Cost-Benefit Principle – the cost-benefit principle prescribes that the benefits from an activity in a BIS outweigh the costs of that activity. The costs and benefits of an activity such as producing a specific report will impact the decisions of both external and internal users. Decisions regarding other systems principles are also affected by the cost-benefit principle.

2.1.2 Classifications of Business Information Systems

A classification is both a process and a product. On the one hand, classification is the act or process of systematically arranging some subject matter into groups or categories according to selected criteria. On the other hand, classification is the more or less formally structured set of classes or categories which emerges (Bailey, 2005; Vessey et al., 2005; Wheaton, 1968).

Classification systems in botany, chemistry, and zoology have played an indisputable role in the development of their respective fields. First, they provide a set of unifying constructs so that the area of interest can be described systematically; more than that, however, they interpret the aspects of relevance. Second, they predict future development areas; the periodic table, for example, predicted the existence of certain elements decades before they were isolated. Third, a system of classification is a means to an end, rather than an end in itself. It can help, for example, to determine relationships between what is classified and other selected variables of interest, in which case it can be used to interpret, predict, or control some aspect of interest. In particular, it is generally agreed that a classification system should permit exhaustive classification and comprise mutually exclusive categories (Vessey et al., 2005; Wheaton, 1968).

Table 1. Example of the Types of Business Information Systems

Classification Criteria	Example of the Types of BISs
Organizational Level	Transaction Processing Systems Management Information Systems Decision Support Systems
Functional Area	Accounting Information Systems Production Management Systems Human Resource Management Systems
Application Scope	Personal Information Management Systems Enterprise Resource Planning Systems Interorganizational Systems
System Architecture	Centralized Systems Client/Server Systems Web-Based Systems

BISs vary widely in their functions, capabilities, performance and social consequences, as well as in their components, inputs, outputs and the users that they can support. Normally, they are classified in several ways such as: (a) organizational levels, (b) major functional areas, (c) support provided by the system (application scope), and (d) system architecture (Barron et al., 1999; Turban et al., 1996) (see Table 1).

2.2 Legacy Business Information Systems

Legacy BISs are BIS that are critical to the operation of companies, but that were developed years ago using early information technologies. These BISs have been maintained for many years by many software

engineers, and while many changes have been made to the BIS, the supporting documentation may not be current. These factors contribute to the staggering cost of maintaining legacy BISs. Consequently, there is an urgent need to find ways to make these BISs more maintainable without disrupting the operation of the company (Joiner & Tsai, 1998).

Some years ago, legacy BISs were one of the major information systems problems, but this has become less prominent recently. This is possibly because many companies have been obliged to replace old BISs to ensure Y2K (millennium) compliance. However, it is safe to predict that the problem will soon appear again, in a much more severe form (Bennett & Rajlich, 2000).

2.2.1 Tactics for Dealing with Legacy Business Information Systems

Much effort has been expended over the past twenty years in developing technology solutions to legacy BISs, of which there are several approaches (Lee & Yoo, 2000; Martin et al., 1990; Martin et al., 2004; McNurlin & Sprague, 2005; Seacord et al., 2003; Serrano et al., 2002):

1. Replace legacy BISs with purchased packages – this is a very attractive alternative when it can be used. If there is an available packaged system that satisfies the needs of the company, then it can be purchased and

installed to replace the obsolete BIS. In many instances, however, there is no packaged system that is deemed satisfactory, so this option may not be available.

2. Rewrite legacy BISs – in some cases, a legacy BIS may beyond rescue. If the code is convoluted and patched, if the technology is antiquated, or if the design is poor, it may be necessary to start from scratch. The thought of rewriting a large BIS is often discouraging because of the large amount of resources it will take.

3. Improve legacy BISs – reworking the old BIS to make it less difficult to maintain by upgrading the documentation and converting existing spaghetti code into structured code. This is a more ambitious form of improving aims to upgrade the old BIS so that it better serves the needs of the organization and is also easily changeable so that its functionality can be upgraded as the needs changes. It is often possible to upgrade an old BIS for less than half the cost of developing a new one.

2.2.2 Myths Associated with Legacy Business Information Systems

It is important to dispel some common myths about legacy BISs. Misinformation has always surrounded much of the debate regarding legacy BISs. Misinformation originates from many sources. Most of these

sources have little practical experience working with these systems or may have an agenda that is furthered by avoiding or downplaying the role of legacy BISs within the company (Ulrich, 2002). Below are common myths associated with legacy BISs (Bennett & Rajlich, 2000; Ulrich, 2002).

1. Legacy BISs provide little or limited business value – legacy BISs are the lifeblood of a company because they process critical company data. Currently, the majority of legacy BISs are written in COBOL. COBOL applications process 85 percent of all global business data. If these applications suddenly disappeared, companies would find themselves at a major loss.

2. Legacy BIS functionality is no longer valid – this is perhaps the greatest misconception about legacy BISs. Legacy functionality may be hard to decipher, hard to invoke, or redundantly defined, but legacy business logic is also very reliable. In other words, most legacy BISs contain accurate and relevant business logic, but they do not typically invoke this logic in a way that is conducive to dynamic business requirements.

3. New technologies will provide the ultimate answer to legacy BISs – it is seductive to think that current technology developments, such as components, middleware, enterprise computing and so on will provide the ultimate answer to legacy problem, and that once BISs are expressed in this form, there will be no more legacy BISs. Experience acquired

over the past fifty years shows this is extremely naive. It is safe to predict that in twenty years, information technology and information systems will change in ways which we cannot imagine now, and we shall have to work out how to cope with what now is the latest technology, but will become tomorrow's legacy. In other words, the legacy problem is enduring

4. Web-based BISs are rapidly displacing legacy BISs – Web-based front ends may appear to be displacing legacy BISs, but legacy online transaction volume continues to grow. The predominant online transaction processing facility is IBM's Customer Information Control System (CICS). CICS handled 20 billion transactions per day in 1998. This was more than the total number of hits per day on the WWW at that point in time. If legacy BISs were being displaced, one would expect that this number would shrink over time. Two years later, however, IBM reported that the number of daily CICS transactions jumped to 30 billion – an increase of 50 percent. The number of customers has also grown. Gray & Reuter reported 30,000 CICS systems in use in 1993. This number jumped to 50,000 CICS mainframe licenses by 1999. It is clear that legacy BISs contine to be the mainstay of the majority of business environments.

5. Organizations developing new BISs can ignore legacy BISs – studies

have shown that replacement BISs typically retain up to 80 percent or more of the functionality of the existing BISs. Even if this figure is only 40 to 50 percent, the business rules that are in legacy BISs tend to be difficult to reproduce to any degree of accuracy. Any effort to rebuild or replace a legacy BIS, in whole or in part, should do so with an understanding of the BISs being replaced. Legacy understanding is a minimal requirement to determine which portions of the legacy BIS need to be replaced.

2.3 The Deployment Problem and Its Solutions

Information system deployment is a complex process which covers all the activities that have to be carried out from the end of the development itself on developer sites to the actual installation and maintenance of the information system on end-user machines (Carzaniga et al., 1998; Hall et al., 1999). It is worth noting that until recently the research community focused on the development and evolution of information systems. Very little research work dealt with the delivery, installation and maintenance of information systems on end-user machines (Coupaye & Estublier, 2000).

On the whole, there are two ways to ease the deployment load of existing information systems. The first is to deploy information systems

using better deployment tools. Another way is to transform the architecture of information systems for the purpose of deployability improvement. Over the past few years, several research works have been devoted to the study of deployment tools, such as Dolstra et al. (2004), Hall et al. (1999), Hnetynka (2005), Taconet et al. (2003), and van der Hoek & Wolf (2003).

Dolstra et al. (2004) showed that deployment hazards are similar to pointer hazards in memory models of programming language and can be countered by imposing a memory management discipline on software deployment. Based on this analysis, they have developed a generic platform and language dependency verification; exact identification of component variants; computation of complete closures containing all components on which a component depends; maximal sharing of components between such closures; and concurrent installation of revisions and variants of components.

Hall et al. (1999) discussed how the Software Dock framework creates a distributed, agent-based deployment framework to support the ongoing cooperation and negotiation among software producers themselves and among software producers and software consumers.

Hnetynka (2005) presented Deployment Factory, a model-driven unified environment for deploying component-based applications. The Deployment Factory is based on (a) the OMG Deployment and

Configuration Specification; (b) an analysis of contemporary used component technologies; and (c) his experience from component-based development.

Taconet et al. (2003) presented a software infrastructure to support the deployment of large-scale distributed applications that they call Smart Deployment Infrastructure (SDI). SDI offers automatic deployment of multi-component applications. SDI provides a deployment solution to customize the installation of applications for mobile users and to adapt to the device's capabilities, to the user's preferences and geographical location.

van der Hoek & Wolf (2003) defined a flexible release management process and built a specialized tool to support that process in the context of distributed, component-based software development. The tool, called Software Release Manager (SRM), is based on two key notions. First, while components can be released from physically separate sites, the actual location of each component is transparent to those using the SRM. Second, dependencies among components are explicitly recorded so that they can be understood and exploited by the tool and its users. In particular, the tool helps developers automatically document and track transitive dependencies, and helps users in retrieving not just components, but also all of the dependent components.

Carzaniga et al. (1998) and Jansen et al. (2005) characterized a variety of deployment tools to help our understanding of such deployment tools.

Similarly, several research works have focused on the architecture transformation of information systems, such as Babiker et al. (1997), Bodhuin et al. (2002), Hassan & Holt (2005), Kazman & Carriere (1999), Klusener et al. (2005), Krikhaar et al. (1999), Tahvildari et al. (2003), and Woods et al. (1999).

Babiker et al. (1997) presented a reengineering model. The goal of the model was to provide a comprehensive method to reengineer non object-oriented systems into object-oriented architecture. The model consists of three main processes: Reverse engineering, merging, and object-oriented development. Reverse engineering extracts requirements and knowledge from an existing software system and redocuments the system. In the merging process, recovered requirements and knowledge from the reverse engineering process are merged with new requirements and knowledge. The merging process removes redundancy, checks for inconsistency, and detects incompleteness. In the object-oriented development, a reengineered system is developed using an object-oriented software development method.

Bodhuin et al. (2002) presented a migration strategy whose target system was a Web-enabled architecture based on the Model-View-

Controller design pattern. By extracting all the needed information from the COBOL source code, the realized toolkit automatically generated the wrappers for the business logic and the data model and the Web user interface as Java Server Pages.

Hassan & Holt (2005) proposed an approach to migrate from one Web development framework to another, in particular they showed an example of migrating a Web application from the ASP to the NSP framework.

Kazman & Carriere (1999) presented Dali, an open, lightweight workbench that aids an analyst in extracting, manipulating, and interpreting architectural information. By assisting in the reconstruction of architecture from extracted information, Dali helps an analyst redocument architecture and discover the relationships between as-implemented and as-designed architecture.

Klusener et al. (2005) provided detailed insight into the nuts and bolts of architectural modification efforts, and delivered a road-map for computer-aided life-cycle enabling for software. Others can use this work or a variant thereof to conduct architectural modification efforts for their own deployed software systems, when malleability of these systems is not in alignment with business needs.

Krikhaar et al. (1999) described a two-phase process for software

architecture improvement, which is the synthesis of two research areas: the architecture visualization and analysis area of Philips Research, and the transformation engines and renovation factories area of the University of Amsterdam. Phase one of the process is based on Relation Partition Algebra. By lifting the information to higher levels of abstraction and calculating metrics over the system, all kinds of quality aspects can be investigated. Phase two is based on formal transformation techniques on abstract syntax trees. The software architecture improvement process allows for a fast feedback loop on results, without the need to deal with the complete software and without any interference with the normal development process.

Tahvildari et al. (2003) presented a quantitative framework that allows specific non-functional requirements (or software qualities) such as performance and maintainability to guide the reengineering process. The framework aims to address three issues: (a) the composition of a list of software transformations which relate to particular software qualities; (b) the investigation of the mutual impact these transformations have on software qualities; and (c) the design of a method to quantitatively assess the impact of a particular transformation on a particular quality in terms of metrics or quantitative software indices.

Woods et al. (1999) reflected on their experiences in reengineering

and reconstructing the architecture of complex software systems and suggest a new model for organizing the information that results from such reengineering efforts. The new model will make the exchange of information among reengineering tools more predictable and robust.

To my knowledge, however, there have been no studies on the ActiveX component-based architecture transformation of WinBISs for the purpose of deployability improvement.

Incidentally, practical and well-defined ActiveX component-based BIS architecture has not been described thus far. Therefore, I need not only develop the architecture transformation process, but also define the downloadable architecture.

2.4 Visual Basic-Like Visual Programming

The very first programmers had to work at the lowest possible level, writing programs as sequences of bits. Since then the development of programming techniques has aimed to make the programming task easier so that people with less training should be able to produce programs which work correctly, as quickly as possible. Visual programming is one current aspect of that development (Edwards, 1988).

Visual programming is a programming method that allows developers

to graphically construct software. Compared with traditional textual programming methods, visual programming provides a more efficient and easier way of producing software. Several visual programming paradigms already exist, and the features provided by different visual programming paradigms may vary greatly. The VB-like visual programming paradigm is supported by many popular software development tools (which in this paper are called VB-like visual programming tools or VB-like tools), such as VB (Microsoft, 1998), Delphi (Borland, 2005), and PowerBuilder (SyBase, 2004). In fact, the VB-like visual programming paradigm is widely used in BISs, including point of sales systems and accounting information systems.

VB-like visual programming reveals six distinguishing characteristics (Cheng et al., 2007). Firstly, VB-like visual programming supports primarily the development of C/S, data-centric BISs. The database runs on top of a shrink-wrapped RDBMS (Relational Database Management System) package. In contrast, the client programs are either custom-built window programs or ASP.NET-style Web programs, both developed using VB-like visual programming.

Secondly, VB-like visual programming-developed programs consist of form modules, each form comprising COTS (Commercial-Of-The-Shelf) components. Moreover, there are six very common types of forms: splash

form, main form, about form, function-specific primary form, function-specific secondary form, and function-specific design-time-only form. Typically, a VB-like visual programming-developed program is almost completely made up of these common forms.

Thirdly, the heart of VB-like visual programming consists of component libraries and code generators. Through the reuse of COTS components, VB-like visual programming enables developers to dramatically reduce the amount of time and code required to write a program. In addition, VB-like visual programming provides WYSIWYG (What You See Is What You Get) code generators such as form editor and report editor, where the developers merely fill in forms, drag and drop icons, or click buttons to automatically generate most code (structured code), and thus developers only need to hand-write some code (unstructured code).

Fourthly, a component library designed especially for VB-like visual programming is a special object-oriented class library that follows a certain component specification, such as OMG CCM component specification, Microsoft .NET component specification, or Borland VCL component specification. Any component library must be installed on a VB-like tool, and the installed components will appear on the component palette systematically. Of course, developers are not limited to using the

components that ship with a VB-like tool. In fact, developers always add certain customized or third-party components to the VB-like tool for particular reasons. Currently, Microsoft .NET Framework Class Library and Borland VCL Library are the most popular component libraries. Both libraries provide components and functionalities that allow developers to build forms, make reports, and to access databases.

Fifthly, VB-like visual programming supports unstructured code through the event-driven writing method. In event-driven writing the developers identify the events (such as a user action or a change in focus) that the program must handle, and write event-handlers to respond to the events.

Finally, in VB-like visual programming the developers always write some non-event-handling code. Typical reasons given include reducing code redundancy, simplifying complex event-handlers, decreasing memory requirements, simplifying data passing between forms and facilitating code reuse in the future.

It should be concluded, based on the characteristics outlined above that: (a) VB-like visual programming is a form-oriented, COTS components-based, and event-driven programming method; and (b) VB-like visual programming is an iterative and incremental process that is organized around five primary activities:

1. Creating a new form.
2. Adding a component to a form.
3. Setting a component property.
4. Writing an event-handler.
5. Writing some non-event-handling code.

2.5 Summary of Literature Review

1. Legacy BIS management is an important issue in information systems (see section 2.2). The study presented in this paper proposes and demonstrates a solution to make the maintenance of numerous legacy WinBISs easier, and is thus a contribution to the information systems field.

2. The study presented in this paper is a unique and innovative study designed to improve the deployability of legacy WinBISs using ActiveX components. To my knowledge, no studies have used a similar design to address the same problem (see section 2.3).

3. The architecture transformation process presented in this paper is suitable for the VB-like tool-implemented legacy (existing) WinBISs – a type of BISs in the classification based on system architecture. An architecture transformation process is independent of organizational

level, functional area, application scope, and other such non-architectural classification criteria. We have no need to develop a more specialized architecture transformation process to meet a particular organizational level, functional area, or application scope (see section 2.1).

4. The architecture transformation process presented in this paper is suitable for the VB-like tool-implemented legacy WinBISs. An architecture transformation process is dependent on the information system's architecture and implementation method. It is impossible to develop a more general yet practical architecture transformation process, which is independent of the information system's architecture and implementation method (see section 2.1).

5. Almost all WinBISs are implemented using VB-like visual programming. Moreover, VB-like visual programming is also the heart of the architecture transformation process presented in this paper. Section 2.4 discussed the essential features of VB-like visual programming.