Core Value 2 — A Project Chronicle: Tools and Techniques for Successful Client-Server Implementations

As a reader of newspaper columns, I have often been frustrated with the de facto requirement that an article must exclaim a product or technology either wonderful or terrible. As a reader, what do you believe? As a business professional, how do you pick technologies and products that maximize your success?

The answer to both questions is to learn from the past. Over the past 20 to 30 years, four basic principles evolved and remained constant. The names and the implementation tools for these principles keep changing, but the concepts remain the same.

1. Good systems have solid underlying architectures; they do not just evolve.
2. Good applications are designed; they are not just thrown together.
3. Successful implementations are planned; planning is the insurance policy of systems implementations.
4. Most technologies are evolutions, not revolutions.

I use history as a guide to analyze critical success factors for successful project implementations. To begin, can client-server technologies with hundreds of on-line users be successfully used for mission critical large-scale projects?

Just a few short years ago, the promises of client-server systems covered the front pages of many industry periodicals. Client-server systems were highlighted as slashing the costs of computing. Today, these same front pages are covered with articles describing the failures of client-server systems and declaring they are not ready for prime time systems. What went wrong?

In general, two different groups of technologists were asked to build new client-server systems. The first group consists of seasoned professionals with many years of data processing experience who are accustomed to development cycles that extend over many years. The classical developers focus on applications, systems, networks, or databases. In the client-server world where a developer needs to know something about each of these disciplines, the classical developer feels lost. Frequently, due to long development cycles, the business needs have changed by the time the application is completed.

The second group of developers grew up with personal computer tools such as dBase and Basic. These PC developers successfully built single user and small departmental systems with 2 to 20 users. The developers succeeded by using a "prototyping" methodology where they meet with the user, write a code, run a test, review the application, and begin the cycle over again. This methodology is acceptable for small systems with limited users. Likewise, applying this approach to large-scale systems with hundreds or thousands of users, many interrelated functions, and complex databases, often leads to failure. If both of these approaches are wrong for today's systems, how do we proceed?

At Metamor, we have deployed systems with hundreds of relational database tables, hundreds of on-line users, and very complex requirements, by combining the best of traditional software development methodologies with current tools and techniques. Simply stated, our formula is as follows.

First, we develop a custom framework for our client/server systems. The framework consists of complementary standards and methods that provide a common mechanism for communication, as well as an ensured consistency over the project life cycle. Next, we break down all projects so no single phase is longer than 12 months. Finally, we use a modified full development methodology, which consists of the following items:

- Business Function Specification (BFS)
- Functional Design Specification (FDS)
- Detail Design Specification (DDS)
- Acceptance Test Plan (ATP)
- Deployment Plan.

The Business Function Specification

In the business function specification, we define all the "WHATS" from an end-user's point of view. WHAT will the system do? WHAT information will it handle? WHAT will the user interface look like? WHAT does the business use the data for? WHAT rules are used to manage the business? This document reads like a users guide for a completed application.

The Functional Design Specification

In the functional design document, we define all of the "WHATS" from a technologist's point of few. WHAT does the database look like? WHAT fields appear on each screen? WHAT business rules apply to what data? This document is targeted at the developer and viewed as a bridge document between the end-users view of the system as represented in the BFS and the system construction details as represented in the DDS.

The FDS begins from a system-wide perspective and defines all of the data used in the system, represented with an entity-relationship diagram (ERD). After defining the data used in the system, divide the system into business functions and associate one or more screens (objects) with each function. Each screen consists of fields defined in the ERD and is available through one or more menus. Finally, we define each object.

The Detail Design Specification

The detail design document defines all of the "HOWS". HOW does the application store and retrieve data? HOW does the application enforce referential integrity? HOW does the application handle screen navigation? HOW is the code structured? By defining *how* each event works, common functions and processing steps are identified and incorporated into library routines.

The process of defining components for an event-driven GUI system differs from defining components for traditional 3GL-based systems. Instead of dividing a system into program modules, divide the system

into the application objects (menus, fields, screens, and reports) that are analogous to the objects identified in the functional specification. For each object, design the actions the system takes for a specific event. An event might be a double-click with the mouse or the failure of a database transaction. Each object's events and actions are documented; similar to the way a module or a subroutine in a 3GL is documented.

Metamor uses a design process called OEA Analysis™ (Object Event Action). First, specify the actions that need to be taken for every object/event pair. The documentation for an action might include the SQL to store or retrieve data, the logic for calculating a derived field, the processing steps for maintaining referential integrity, or the code for moving the cursor to the next field. After documenting single objects, such as fields, document collections of objects, such as screens and menus. After defining an application's objects, events, and actions, ensure the design is consistent and complete using a technique called Action Normalization. This technique has three steps:

1. Ensure that all common object types have common characteristics.
2. Ensure that common events have consistent actions.
3. Gather all similar actions and define common routines (subroutines or methods) that handle the processing.

Following the OEA normalization process, begin the process of building the application. Two key documents are used in the construction plan; the ERDs, which document the database design, and the tables of objects, events and actions that allow you to find common routines and build clean codes.

In summary, many client-server development efforts have failed due to improper planning. To succeed in large client-server implementations, the best of traditional technologies and today's tools and techniques must be leveraged.