The Main Architectural Principles Responsible for the Successes of VueCentric and CareWeb

Douglas K. Martin, M.D.
Regenstrief Center for Biomedical Informatics
410 West 10th Street, Suite 2000
Indianapolis, IN 46202
dkmartin@regenstrief.org

Introduction

VistA has a long history of collaborative evolution. The M language, by its very nature, promotes community-based development in providing readily accessible source code that may be easily modified. This capability is further enhanced by the open architecture of FileMan and the clinical packages built upon it. Leveraging this open and extensible design, the more innovative and tech-savvy medical centers could modify or enhance nationally developed and distributed software to meet their needs. Where such enhancements were deemed sufficiently generalizable, they often made it back to the national code base, completing the cycle of collaboration.

In late 1990's, CPRS was making its debut in VA medical centers across the country to mixed reviews. Considered innovative and groundbreaking for its time, CPRS was not without its flaws or its detractors. It seemed as though everyone had a different opinion about how to improve it. Historically, implementation sites would experiment and innovate on alternative designs, finding what worked for them and offering those innovations back to the community. But CPRS was built upon a very different technology stack from its M-based predecessors. A thick client application programmed in the Delphi computer language, CPRS presented a much more closed architecture and monolithic design. The traditional model of applying local innovation to address perceived shortfalls in functionality proved difficult if not impossible.

A Framework Pedigree

One of the early pain points with CPRS was the inefficiency of documenting clinical encounters. Not unique to CPRS, this was a common complaint against electronic health record systems of the day. In 1998, a consortium of four VA medical centers pooled resources to fund an experiment to integrate a third-party clinical documentation tool into CPRS. As an informatician at one of the participating sites, my colleagues and I recognized the immensity of the task given the architectural constraints imposed by CPRS. We proposed a novel, albeit somewhat radical, multi-phased approach. First, create an open framework to provide the infrastructure to host modular plugin components. Second, componentize the CPRS source code along functional cleavage planes, creating discrete modules to be deployed into the new framework. Finally, integrate the third-party clinical documentation tool by wrapping it with code to make it framework aware.

This effort culminated in the creation of the VistAtion Framework. Written in Delphi, the framework used Microsoft's Component Object Model (COM) standard to achieve interoperability among modular components. The framework offered a number of important core services to its hosted components. These services have persisted and been refined through multiple generations of framework design and are the primary focus of this paper.

Despite a successful pilot of the VistAtion-based CPRS with an enhanced clinical documentation capability, VistAtion never achieved traction within the agency. In 2001, Clinical Informatics Associates, Inc. (CIAI) (cofounded by the author) unveiled the next generation of the VistAtion Framework. Dubbed the VueCentric Framework, this new framework represented a significant refinement of its predecessor, building upon the lessons learned in the pilot project.

In 2002, searching for a graphical user interface for its RPMS system (a close derivative of VistA), the Indian Health Service (IHS) elected to conduct a pilot of the VueCentric Framework. Having considered CPRS as a potential solution, IHS recognized that CPRS would require significant modifications to adequately serve a patient population very different from that for which it was designed. However, despite the absence of support for pediatric and women's health services, CPRS had many benefits to offer. Again, its relatively closed architecture presented a significant barrier to introducing the needed modifications. The VueCentric Framework provided an attractive alternative.

In 2002, CIAI conducted a successful pilot of the VueCentric-based CPRS complete with modules designed to facilitate the care of women and pediatric patients. The pilot was eventually followed by an agency-wide deployment of the product under the moniker of the RPMS Electronic Health Record.

In 2008, the author, now at the Regenstrief Institute (RI) in Indianapolis, was faced with an aging software infrastructure and the daunting task of modernizing 30 years of in-house developed software. Based on previous experience, a modular architecture seemed a prudent choice for the re-engineering effort. While the VueCentric Framework showed promise, the requirement of a web-based application eliminated it as a viable option.

The CareWeb Framework emerged out of this effort – built upon the same architectural principles that led to the success of its predecessor, but hosted in a web environment using readily available open source libraries. The CareWeb Framework represents a significant evolutionary step in that it focuses on generalizability. Whereas VueCentric has hard dependencies on the VistA/RPMS infrastructure, CareWeb eliminates direct dependencies on any specific EMR implementation. Interoperability with a given EMR system is achieved by way of adapters that plug into the framework. RI has developed adaptors for its own Regenstrief Medical Record System (RMRS), as well as for VistA/RPMS and OpenMRS.

Architectural Successes

While each framework generation has seen refinements and improvements and the CareWeb Framework of today is a far cry from its humble beginnings as the

VistAtion Framework of over a decade past, many key architectural features have nonetheless persisted largely unchanged in their specification, though with significant changes in their underlying implementation. Here, I will highlight those aspects of the framework architecture that have contributed most to the ongoing success of the approach.

Component Metadata

An intelligent framework needs to know something about the modules it hosts. From our very early framework designs, we understood that a plugin module needs to expose metadata that describes its capabilities. These metadata provide the means whereby the framework can properly interact with the associated module, whether it be materializing the module from its serialized state, serializing the run time state of the module to a persistent data store, or providing module configuration capabilities to the end user. The format, composition, and location of these metadata have changed significantly over time. Most of these changes have been dictated by the requirements of the supporting technologies. However, the categories of information represented in the metadata have generally remained the same:

- Component identification – The need to uniquely identify a plugin component is necessary for a number of operations, serialization and dependency management among them.
- Serialization control – These metadata determine how the run-time states of components are serialized and allow the layout designer to present configuration options to the user.
- Access control – Permits the restriction of user access at the plugin level by specifying the required permissions.
- Dependency management – Specifies other components required for the operation of the plugin. This is important for both deployment and run-time loading of dependent components.

VueCentric stores metadata in a FileMan file. These metadata are delivered as part of the installation package (KIDS build) for the component. Under CareWeb, metadata are packaged within the plugin itself. This simplifies plugin deployment by making plugins truly self-contained and self-describing. CareWeb leverages the Spring Framework, an open source Java-based framework for managing component lifecycle, to represent much of its metadata. The Spring Framework is especially well suited for managing the complexity of a highly modular application environment.

Component Registration and Discovery

A modular framework introduces some level of optionality. Component developers need a means to accommodate situations where component behavior must be modified depending on the presence or absence of an optional component. Component registration enables a component instance to declare its presence to the framework. Having registered itself to the framework, component discovery allows another component to detect its presence and gain direct access to its services.

The CareWeb Framework introduces an additional benefit to component registration. Components that implement a special registration callback interface will be notified each time a component is registered with the framework and be given an opportunity to inspect the component to determine if it is of interest. This capability is used, for example, by the context manager to detect shared contexts that it will manage and by the shared contexts to locate their respective context subscribers.

Dependency Management and Deployment

Dependency management is complex in an application environment with lots of moving parts. Ensuring that all the necessary components are present in the environment requires a means for enumerating all the interdependencies (both direct and transitive) as well as a mechanism for deploying them to the runtime environment. VueCentric handles dependency management and deployment directly. Component metadata are used to identify a component's dependencies. Since VueCentric runs on the client, its deployment capability pulls components from a central repository and installs them on the client on an as needed basis.

CareWeb, being a web-based framework, has a much simpler deployment profile – required components need only be deployed to the web server. CareWeb uses an established, open source dependency management system called Maven to accomplish this. Maven provides sophisticated dependency management capabilities and enjoys broad adoption. Maven retrieves dependent components from a central source that may be a single repository or a federation of repositories.

Context Management

The ability to share key context states across the application and manage the transition of these context states in a collaborative fashion is crucial. For example, the currently selected patient is a context state required by most clinical modules. These context subscribers need to not only know the current state, but also have the ability to know when the state changes and possibly block a state change when doing so might be detrimental to an operation in progress. When the original VistAtion concept was being developed, a new standard was emerging under the moniker CCOW (Clinical Context Object Workgroup). This standard provides a means for multiple independent applications running on the same physical machine to collaboratively share context states. While not intended to coordinate context state within an application, we found that the CCOW approach with certain modifications was a good fit for our framework design.

The CCOW standard dictates a two-phase approach to a context change transaction. A transaction is initiated when a context participant requests a context change (e.g., a new patient is selected from a patient selection dialog). In the first phase, each context participant is polled to determine its willingness to allow the change to proceed, essentially returning a yes or no vote. The second phase depends on the outcome of the first. If all participants concede to the context change, the context change is committed and each participant is notified of the change. On the other hand, if any participant declined the change, the requesting application may cancel the request (with each participant being notified of the cancellation) or may overrule the veto and force the change to proceed (with the transaction being committed).

Certain behaviors dictated by the CCOW standard are not a good fit for managing context state across modules within an application. First, CCOW allows a user to request that a context participant secede from the shared context. This means that the context state for that application would no longer be synchronized with the remaining participants. Within an application, however, the burden for maintaining a synchronized context state across modules is much higher. Allowing a module to withdraw its participation would create a confusing user experience indeed. While it is technically possible for a context change subscriber to revoke its subscription or ignore a change in context state, this practice is strongly discouraged within framework guidelines.

Second, CCOW allows a participant requesting a context change to force the change over the objections of the other participants. We found this too dictatorial and potentially detrimental and disallow this behavior within the framework, with certain exceptions. Again, the expectation of a more democratic approach is higher in a collaboration among modules residing within a common application instance than for that across separate applications. Respecting this, the framework's context manager disallows this veto behavior. Rather, whenever a context change subscriber vetoes a request, polling stops and the transaction is canceled. There are two exceptions to this. First, in the event of a forced logout (such as an inactivity timeout or a server shutdown), all contexts are forced to a null state regardless of the outcome of the polling phase. Second, if the context change request originated external to the application via another CCOW participant, the requester may force the change in accordance with the CCOW specification.

Third, CCOW prohibits a participant from engaging in any user interaction while responding to a polling request during a context change transaction. The framework context specification conditionally allows for such interaction. That is, when the framework context manager polls a subscriber for its vote, it indicates to the subscriber whether or not user interaction is permissible. In the aforementioned examples where a context state change is being forced, user interaction is prohibited. However, under more usual circumstances, a subscriber may request user interaction and potentially modify its vote based on that interaction. For example, a clinical note-authoring module might have a draft note under construction at the time of a patient context change request. Presuming that user interaction is permitted, the module might prompt the user for one of several courses of action: the user may want to save or delete the current draft and allow the change to proceed, or block the request and continue editing the draft note.

Finally, both VueCentric and CareWeb introduce the concept of nested context change transactions. This is useful where context states are in some way interdependent. For example, an encounter context has an inherent dependency on the patient context. When the patient context is changed, the encounter context must also be changed. The context manager treats such a scenario as a nested transaction where all of the member transactions must complete or none do.

This context management model has proven itself across each generation of the framework pedigree, though not without some refinement in its underlying implementation. In our early iterations, context management and the domain objects whose context states were being managed were tightly coupled. We now use an adapter design pattern where any domain object may be wrapped with a context-aware wrapper without modification to the object itself. We also tightly bound the context management function to the framework itself. Now, context management is a service that registers itself to the framework like any other plugin module. By leveraging the framework's component discovery services, it is able to link context change producers to their corresponding context change subscribers and manage context states on behalf of those components.

Event Management

The ability to communicate events within an application and across application boundaries is critical to coordinating the activities of multiple moving parts. Both VueCentric and CareWeb use a similar model, though very different implementations, for event subscription and delivery. First, an event is identified by a hierarchical identifier and is bundled with an arbitrary data payload whose content is determined by the event producer. The advantage of a hierarchical identifier is that an event producer can publish an event with a high level of granularity, while a consumer can choose to subscribe at a lower level of granularity. For example, an event published with a hierarchical identifier of ID1.ID2 would be received by consumers subscribing to ID1.ID2 and ID1, but not by those subscribing to ID1.ID3 or ID2.

Second, an event may be published in one of two scopes: local or global. A local event is distributed to subscribers within the same application instance only. A global event is distributed to subscribers system-wide. Global event distribution can be constrained to specific subscribers (via user or application instance identifiers) if desired.

VueCentric implements global event distribution using a global event manager running on the M server. A client application firing a global event does so via its broker-based connection to the M server. The global event manager keeps track of all event subscriptions for all running applications and, upon receiving a global event for delivery, conveys that event to the respective subscribers. For broker-based subscribers, the events are conveyed to the client via a background polling service running at the client. Publishers and subscribers can also be M processes running on the server. This allows server-side processes to send asynchronous notifications to remote clients.

CareWeb creates an abstraction layer for global event delivery, allowing for alternative implementations. An implementation using ActiveMQ, an open source, Java Messaging Service (JMS)-compliant messaging server, is provided as part of the CareWeb Framework distribution. An alternative implementation using the M-based global event manager of VueCentric and a Java-based RPC broker client is provided in the RPMS/VistA support package. While VueCentric constrains the data payload associated with an event to be a string, CareWeb supports any serializable data type for the data payload.

Layout Management

A common complaint about EMRs is that the user interface layout does not match the workflow. Given that workflow may vary from one venue to the next or by user role or by a number of other external factors, the ability to provide flexible layouts that are sensitive to these factors is important.

Both VueCentric (which refers to layouts as templates) and CareWeb implement layout management. There are several key aspects to layout management. First, a layout designer allows a user with sufficient permissions to dynamically create a user interface layout from a palette of available components and save it to the underlying data store. Second, layouts created in this manner can be associated with specific entities such as clinic location, user role, or clinical specialty. Third, a layout manager determines the appropriate layout given these external factors and materializes the layout in the running application. Fourth, layouts can be exported and exchanged among implementation sites.

Security

VueCentric integrates broker-based security into the framework. CareWeb creates a security abstraction layer allowing alternative security implementations. The CareWeb Framework distribution provides core components for implementing a security solution based on the open source Spring Security Framework. The RPMS/VistA support package provides a broker-based adapter for Spring Security.

Conclusion

The VueCentric and CareWeb Frameworks have enjoyed considerable success in real world environments. The latter represents the culmination of over a decade of iterative development, building on lessons learned from earlier efforts. Key evolutionary trends include:

- Separation of domain-specific elements from the framework to maximize generalizability.
- Movement to a web environment to simplify deployment.
- Abstraction of key services to allow alternative implementations.
- Emphasis on open source.