**Software Architecture for the**

**eChronos Game Framework**

**The eChronos Gaming Framework**

The **eChronos** gaming framework consists of two applications with two shared libraries and a shared database. The authoring application, Quest Master, allows the user to create an adventure game or interactive novel. The other application, Adventurer, enables the user to play the games created with Quest Master. Both applications share the same database: Quest Master contributes to the database with read/write updates, but Adventurer can only read from the database. Adventurer’s state of play is saved as a copy so that the original data from Quest Master is not changed, and multiple games of Adventurer can exist concurrently for the same or different players.

Both Adventure and Quest Master use a shared library that is specific to these two applications called ChronosLib. Both applications and ChronosLib also use a general application-independent utility library MyLibrary.

**The MVP Software Architecture**

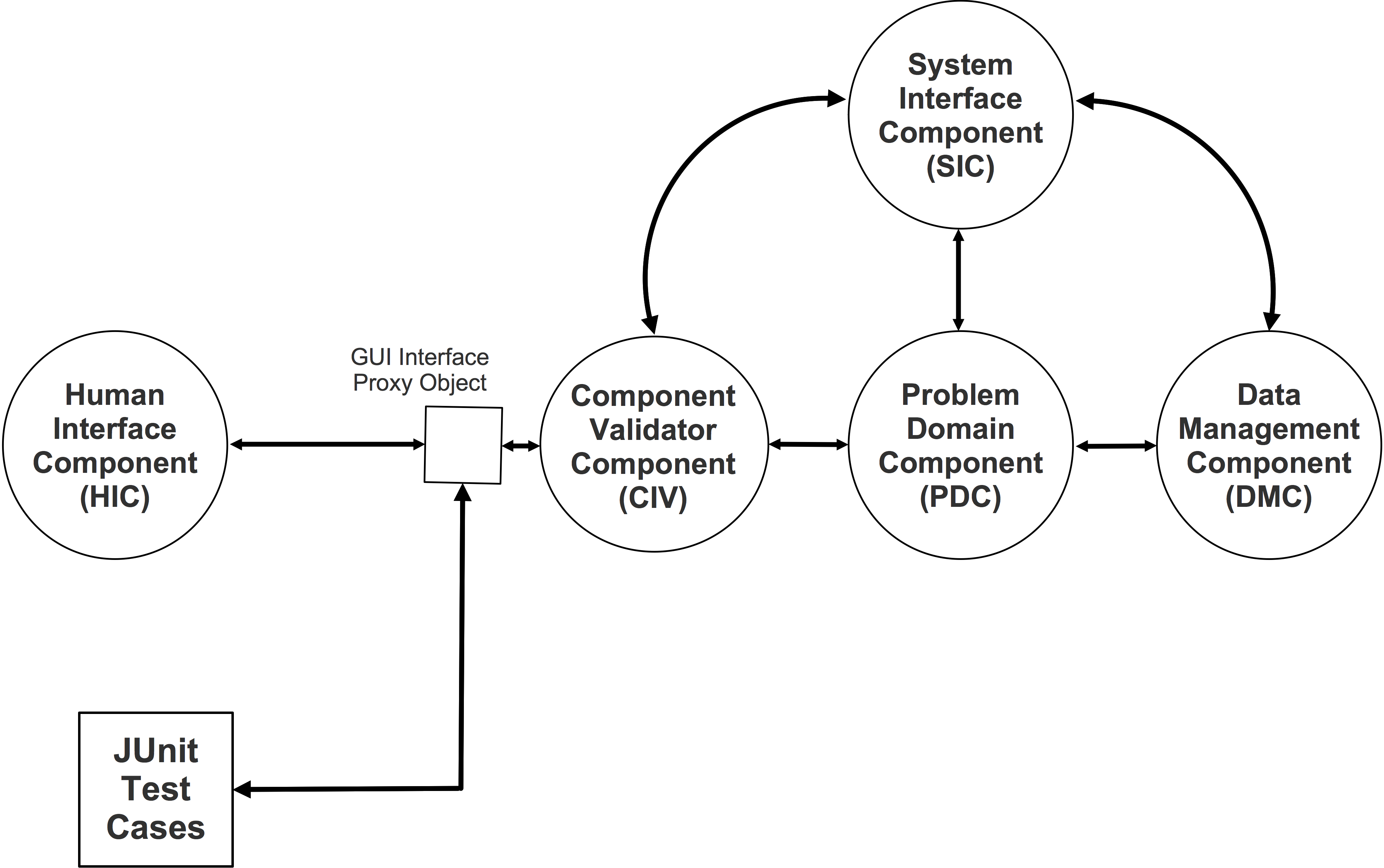
Both applications use an enhanced version of the classic Model-View-Controller[[1]](#footnote-1) architecture called Model-View-Presenter (MVP), which maximizes the separation-of-concerns principle, a critical architectural design. MVP was invented by Peter Coad and Ed Yourdon in the late 1980’s, which they called the “4 Component Model”. Carolla Development added to the MVP model in 2009 by adding the CIV component to allow more comprehensive automated testing. It became known as the “4+1 Component Model”, but is still an enhanced MVP.

The MVP architecture allows cleaner logical separation for clarity, easier maintenance, and enables better automated testing. (A discussion of the components mentioned below follow Figure 1.)

* The *Model* abstraction contains the actual business object under interaction, and is implemented as classes within the Problem Domain Component (PDC).
* The *View* abstraction contains the data types needed to convert internal to external formats, and vice versa; and is implemented as classes within the Component Interface Validator (CIV) Component.
* The *Presenter* abstraction shows the user a black-box view of the business object through interactive visualizations, such as GUI widgets, windows, dialog boxes, and visual controls; and is implemented as classes within the Human Interface Component (HIC).

**The MVP Java Implementation**

The application software consists of four components, each component containing a collection of logically related classes for that component. All classes within each component are contained within a Java package of the same name: pdc, hic, civ, dmc, or sic. Each component is described below. See Figure 1.



**Figure 1. The MVP Architecture with CIV**

* **Problem Domain Component (PDC):** The PDC is the core component. It manages and contains all business logic. All data used internally are formatted for the best efficiency, and operate at the theoretical optimum. The PDC works at the informational and logical level of data.
* **Human Interface Component (HIC)**: The HIC manages all inputs and outputs that originate or pass data or control to and from a user, using a keyboard and mouse interface interacting with GUI widgets. The HIC uses only visualizations and all data is of type String. The HIC is responsible to interact with the user and the CIV, only the CIV. The HIC works mostly at the syntax level of data. There should be no logic in the HIC except that needed to handle keyboard and mouse events, and draw GUI widgets.
* **Systems Interface Component (SIC)**: The SIC manages any communication to and from external applications or systems outside the application, handling local and remote networks, connection pools, HTML protocols, etc. The SIC communicates at the semantic and informational level.
* **Data Management Component (DMC)**: The DMC Manages the persistence data and mechanisms for storage, such as file systems, database management systems, XML parsers/writers, and permanent media. The DMC works at the semantic level of data.
* **Component Interface Validator (CIV)**: The CIV (pronounced *sieve* as an analog to a filter or strainer) is a special component that validates and formats data between the HIC and the PDC (or sometimes the SIC), and provides a “socket” for test cases to simulate HIC. All GUI’s String data must be syntactically validated and reformatted to the application's internal format, and vice versa for outputs to the GUI. It can also contain the semantic validation of the HIC inputs since the HIC should not have any non-GUI logic.

Because GUI objects are difficult to test automatically, the CIV contains as much logic as possible, relegating the GUI code of the HIC to pure aesthetic display. This approach allows the CIV to be a place for the JUnit testing engine to “plug in”, and ensure that as much semantic and informational logic as possible can be tested automatically. My previous projects have about 85% of code automatically tested this way, both unit and integration tests; the GUI code accounts for the remaining 15%, which must be verified by manual inspection periodically.

**Interactions Within the MVP Components**

The lines connecting the components shown in Figure 1 are the only paths data may travels. Note that all components can talk to each other except for two cases.

1. No component can talk to the HIC except through the CIV, and the HIC can not talk to any other component except the CIV. This means that classes of the HIC cannot import the pdc, dmc, or sic packages, and no package except the CIV can import the hic package.
2. The DMC must go through the PDC to talk to the HIC. A communication path between the DMC and the HIC would result in an suboptimal architecture, sometimes called a “two-tiered architecture”, or a “window-on-a-database”, and has been shown to be a fragile, high-maintenance solution.

**The Integration Testing Framework and JUnit**

All integration (and unit) tests are driven by the JUnit test framework and simulates an entire user story, or a collection of stories that comprise a use case. To avoid the various visualizations, pauses, and other click-and-point operations that will stop a test, which JUnit does not handle well, a JUnit test interposes a proxy class test.integ.InterfaceProxy class for each HIC class interacting with its corresponding CIV class.[[2]](#footnote-2) The JUnit test triggers the proxy to send the message under test, and then receives the CIV response. The CIV class only sees an Interface Java type, unknowing whether it is a proxy or the actual GUI object. The proxy contains the proper implementation needed to handle whatever the test requires. Under this arrangement, any public method within the CIV that talks to the HIC must use an *Interface* type as its parameter to support automated testing. Using interfaces in this way is considered a best practice anyway.

For example, the civ.CommandParser expects to receive commands from hic.IOPanelInterface, so its constructor contains the IOPanelInterface parameter so any object of this type can send and receive messages to and from civ.CommandParser.

Therefore, hic.IOPanel must implement IOPanelInterface so that it can connect and talk to the command parser. During production, the user sends commands (e.g. “ENTER Bank”) to the command parser which then makes a Command object to be executed.

During testing, the tester makes a proxy class called IOPanelProxy that implements IOPanelInterface, and therefore contains the same methods as hic.IOPanel (or a working subset sufficient for testing), as required by the rules of interface implementations. The command parser will not know which interface implementation is being used, and will run without test-specific code. The HIC will be bypassed but all logic from the CIV and backend components will be exercised in the test.

The HIC classes are the only ones not automatically tested, which is why the HIC should have a minimum of logic in it. Again, it is important that the civ classes do not invoke any GUI classes, and the HIC classes defer all logic to the civ.

Al Cline

Carolla Development, Inc.

Feb 15, 2015

1. MVC was invented in the late 1960’s with the advent of the SmallTalk object-oriented language by Xerox. It has been the basis of almost all applications, including today’s new mobile apps. [↑](#footnote-ref-1)
2. We haven’t decided yet whether the proxy interfaces and implementing class should be in their own package, e.g. src.test.integ,proxies, or whether they should be gathered in the civ package or somewhere else. [↑](#footnote-ref-2)