

1. PatientOS Architecture

1.1 User Interface (UI)

Human Computer Interaction is a [science](#)¹ spanning [many topics](#)² needed to refine the UI design. In this context the user interface consists of the display and how the application requests and responds to data. Most healthcare information systems have fairly inflexible user interfaces.

This product leverages Java's [dominant GUI toolkit](#)³ along with an excellent open source [look and feel](#)⁴ library, and several rich components.

The **look and feel** can even be changed by the users.

More importantly the IT team can manipulate the user interface to meet the ever changing needs of a healthcare institution.

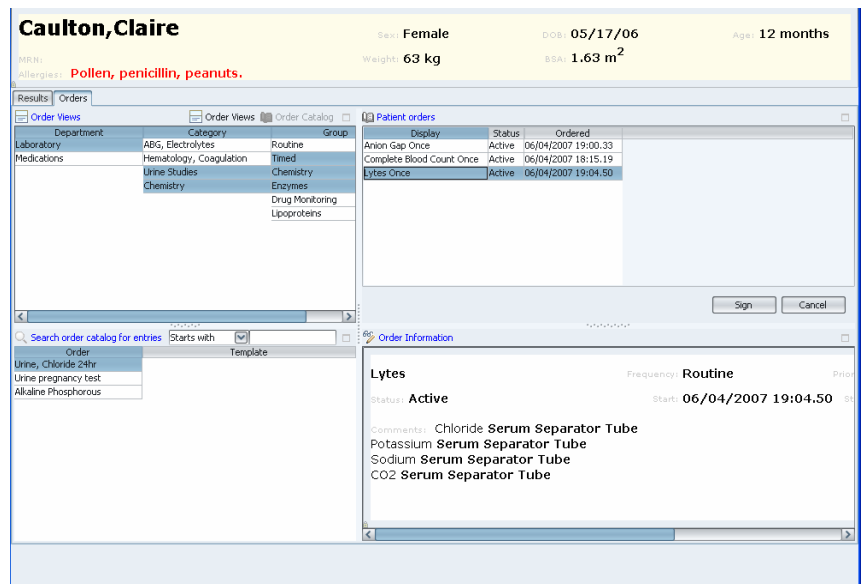
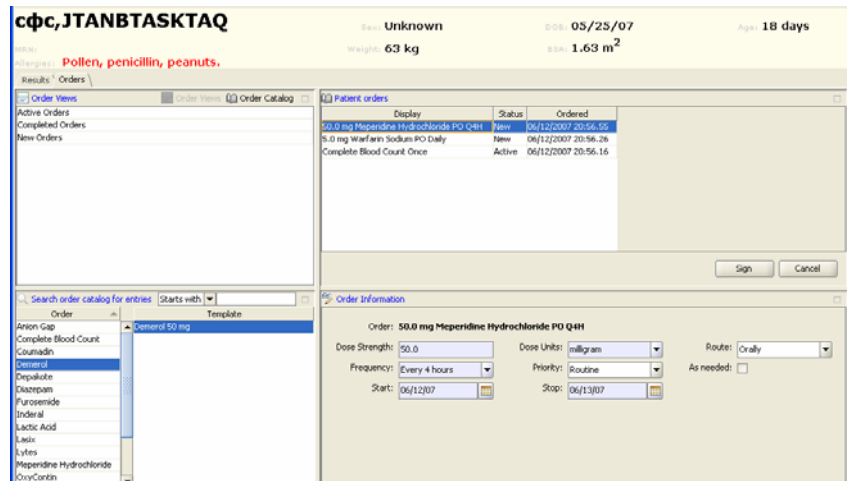
Every piece of text visible on the screens can be customized – even the layout of the panels, the depth of the cascading lists, the patient header or order details.

The interface is consistent throughout the application enabling quick adoption.

Why not web based?

Web interfaces, while [suitable for many internet](#)⁵ and intranet applications, **do not fair well** for complex enterprise wide applications needed in healthcare. Clinicians need precision control, instant feedback, stability, transaction reliability and interaction rich components.

The internet browser was never built as an application development



platform, it evolved, now using a myriad of technologies – HTML, JavaScript, CSS, DHTML, DOM, XML, AJAX, JSP. This requires many more developer resources to build, debug and maintain.

¹ University of Toronto, "Research. Human Computer Interaction". <http://www.cs.toronto.edu/dcs/research-hci.html>

² Wikipedia, http://en.wikipedia.org/wiki/List_of_human-computer_interaction_topics

³ Muller, Hans, "Official: Swing is the Dominant GUI Toolkit". http://weblogs.java.net/blog/hansmuller/archive/2005/10/official_swing.html

⁴ Lentzsh, Karsten, "JGOODIES. Java User Interface Design". <http://www.jgoodies.com/freeware/looksdemo/index.html>

⁵ Nimphius. "Swing or JavaServer Faces: Which to Choose?". <http://www.oracle.com/technology/pub/articles/nimphius-mills-swing-jsf.html>

1.2 Installation

The ability to install and manage applications correlates to the stability of the system's environment. With simple configurations, the implementation cycle and risk of downtime is lowered.

Installation and Configuration

Typical installation: Only the vendor can install the software.

Large system installation: The vendor will do it or the hospital if they hire and train systems managers and database administrators to follow a lengthy process. The installation spans multiple hardware operating systems and hardware.

PatientOS installation: Anyone can run the wizard setup to install the software to test features or evaluate the product.

For large production systems the database and application servers would be configured in a traditional fashion to optimize performance.

1.3 Hardware and Software Requirements

Hardware constrained to specific platforms or vendors is both costly to the organizations and support unfriendly.

Hardware and Software

Typical requirements: Database software is limited to one option, MUMPS, Oracle, etc. Operating systems supported on a single platform – either windows or UNIX. Limited choice in hardware vendors. Citrix is needed to serve across more than one client platform.

PatientOS requirements: The client and application servers will run wherever java runs (Java runs [everywhere](http://www.sun.com/java/everywhere)⁶). Application servers will be supported wherever JBoss runs (JBoss runs everywhere Java runs). The application is essentially database independent, out of the gate two databases will be demonstrated, one commercial, one open source (Oracle and PostgreSQL).

1.4 Languages

To become a truly global the product needs to support a wide variety of languages.

Languages

Typical system: One language - English.

Advanced system. Ability to change reference data, end user screens to display an alternate language. System would not support more than one language at a time.

PatientOS system. Ability to access the entire list of all text displayed throughout the application and change to language of choice. Most of the application will support multiple languages in the same database.

⁶ Sun microsystems, "Java Everywhere". <http://www.sun.com/java/everywhere>

1.5 Build and Configuration

Hospitals' IT departments need to be able to build, configure, and maintain multiple system environments (production, test, train and often many more) during the project phases. Tools need to be simple and efficient to use but flexible to adjust to the wide variety of clinical workflow needs.

1.5.1 Registration Build tools

As an example build registration forms are often time consuming and complex with legacy applications. Registering inpatients can be hundreds of data elements for the patient, visit, contact and insurance information. The registration application should **auto fill** as much as possible for user efficiency.

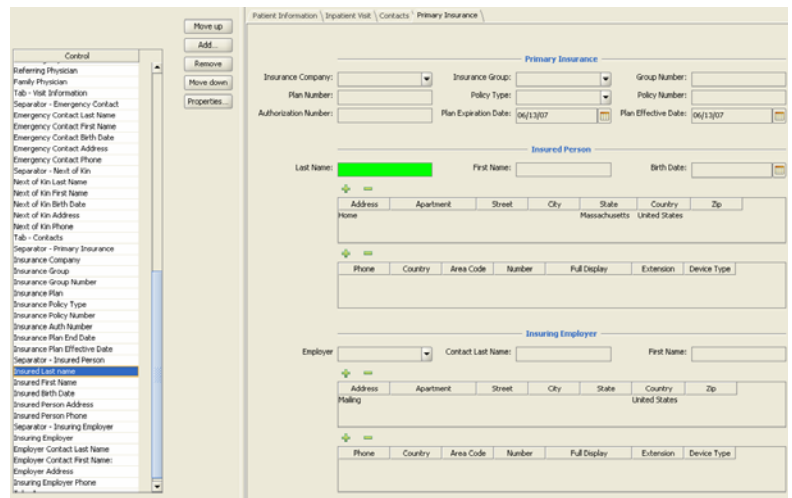
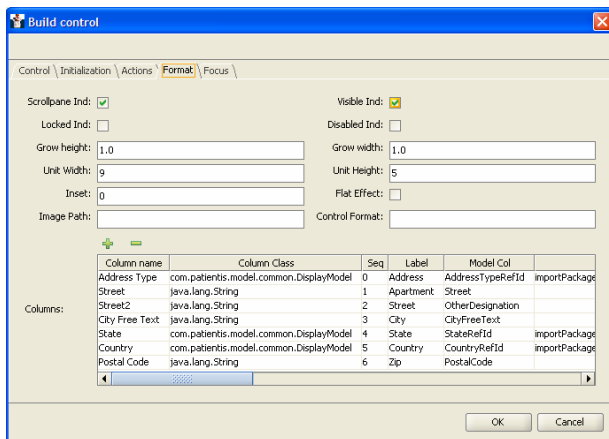
Moreover there are **many** forms needed for inpatient, outpatient, day surgery admissions, transfers, discharges, updates and more.

Implementation of Registration

Fast implementation. Hardcode the forms – no options for the user to change how patients are added.

Typical implementation. The top vendors provide tools that allow the creation of forms from a list of predefined fields (occasionally custom fields). Form building varies but where difficulties are met is the ability to dynamically affect the forms based upon data entry. At best some control properties are affected such as visibility, current value, required indicator etc through proprietary methods.

PatientOS implementation. This product provides a WYIWYG view of the registration forms through the ordering and extensive properties of a control list. Custom controls can be plugged in, dynamic scripting on UI events; even table columns can be defined (below).



From a tool perspective power comes from the ability to copy and pastes controls between forms – or make copies of forms and modify.

For a dynamic UI one of the **most powerful** capabilities is the ability to change the underlying **model** and have that automatically **reflected in the UI** – instead of directly addressing UI controls.

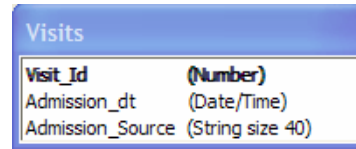
1.5.2 Reference data model

The application architecture makes extensive use of organizing reference data into tree structure. This becomes critical to expand functionality without a convoluted data model. Here is an example.

Inpatient 'admission source'.

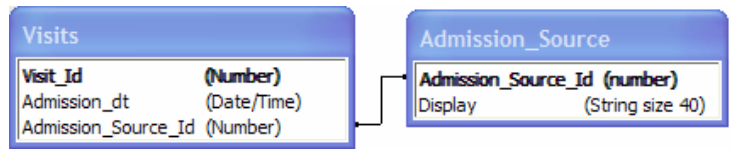
Typical System – (simple)

Has the source as an inflexible string on the visit table.



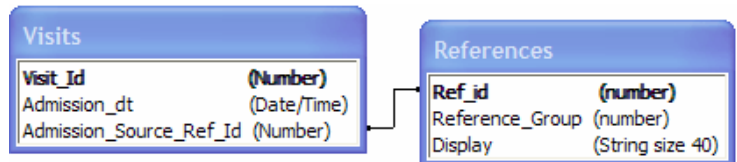
Typical System -

Note that creating a new table for every reference value of this type (there are hundreds) becomes unmanageable.



Typical System (advanced)

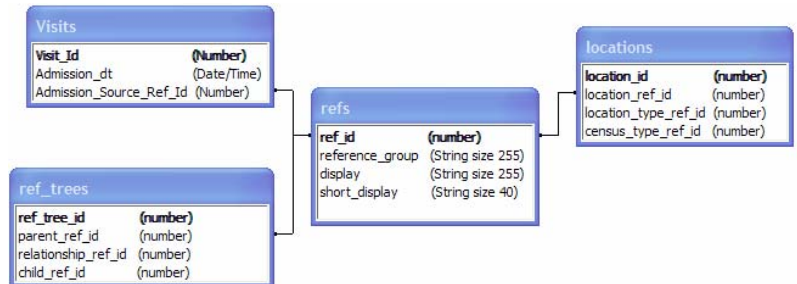
Creating a single table which has a column to define the groups of reference works well. Where problems arise is when the need to capture more properties of the reference data than is common to the shared table.



PatientOS Product's

This product expands the previous implementation in two ways

- a) allows the data to be grouped by using a parent/child relationship between any reference data.
- b) Uses the reference id as a unique index to entity tables which define domain specific properties of that reference data.



This data model pattern is used throughout the system. The impact is profound. a) Simplifies otherwise complex reference relationships. b) Allows the population of entity result sets without pulling from large number of reference tables. c) Just caching and returning refs helps performance and scalability.

1.5.3 Rules Engine

Every organization has many common data requirements for registration, billing, lab and pharmacy processing. Facility specific requirements creates complexity. Different workflows can be found from one nursing unit to another (let alone country to country).

This product uses [JBoss Rules](#)⁷, an open source standards-based rules engine. Through integration with the well defined business objects there is the ability to create simple, powerful business rules.

Integrated into the business logic for save and updating records is the ability to directly affect the business objects using either JavaScript with direct references to the data and/or using a rules engine.

```
rule "New Patient Transaction"
  when
    patientTransaction : PatientTransaction(newPatient == true, incoming == false)
  then
    InterfaceUtility.newInterfaceMessage( patientTransaction );
end

rule "Update Patient Transaction"
  when
    patientTransaction : PatientTransaction(newPatient == false, incoming == false)
  then
    InterfaceUtility.newInterfaceMessage( patientTransaction );
end

rule "New incoming ADT Patient Transaction"
  when
    adtTransaction : InboundPatientTransaction(incoming == true)
  then
    InterfaceUtility.saveTransaction( adtTransaction );
end
```

1.6 Interface Engine

Every sustainable HIS requires the ability to integrate with other hospital systems. For example, single database enterprise-wide systems will have the requirement to integrate with medication dispensing machines, pumps, laboratory analyzers and other medical devices. Presently [Health Level 7](#) (HL7)⁸ is the standard used to communicate with external systems. Alternate web service based options are also becoming popular.

Integration support

Typical system – (simple): Provides limited support of a few interfaces – ADT, results, orders outbound. Basic mapping capabilities only with rudimentary logic.

Typical system – (advanced): Mapping and scripting using a proprietary language. Multiple interfaces within the constraints of fields made available to the interface processing code.

PatientOS system. WebReach Inc. advanced [Open Source interface engine](#)⁹ included with offering. Capability to send any data element outbound using rules and/or scripting engine. Processing inbound transactions significantly easier by direct interaction with the business objects.

⁷ JBoss, "JBoss Rules". <http://www.jboss.com/products/rules>

⁸ Health Level 7 (HL7) <http://www.hl7.org>

⁹ Mirth Project. "Open Source Cross-Platform HL7 Interface Engine". <http://www.mirthproject.org>

1.7 Quality

1.7.1 Software code base

The quality of a typical HIS is generally high only when meeting specific FDA guidelines – such as [510k](#)¹⁰ for medical device applications. The original systems are typically released with quality compromised for the sake of meeting deadlines and demand. The majority of the product is traditionally coded by developers under pressure to produce functionality as quickly as possible, under severe time constraints. Code becomes uncommented; design blurred, original requirements scarce let alone the ‘current’ requirements. The system grows, developer’s leave, only mammoth heroic efforts of the software development teams allow the product to be advanced.

PatientOS will have an unprecedented level of quality and design control – far more than design reviews, and code reviews alone offer. Traceability from source code to design documents, requirements to features, and full test coverage will bring the quality of this information system to a level that will be impossible for legacy systems to reach. This frankly, should and will be the new standard in Healthcare Information Technology.

Maintaining this high level of quality is the undisputed mission of this company. The only changes allowed to the source code will be those following the software development procedures, change control process which will enforce all the documentation necessary to advance the product under design control.

The overhead of this process will be compensated through reduced testing cycles, significantly lower support costs and faster implementation.

1.7.2 Configuration management

Often businesses will place the source code under version control but leave a gap for the database installation, application server configuration, reference settings, and generally all other components of the system. This company will ensure that the resources are in place to manage and maintain these processes and files.

1.7.3 Documentation

Many open source projects pay little attention to documentation due to the technical nature of those projects and users. Documentation suitable for the IT trainers, healthcare analysts, and clinical super users is essential. Even more importantly the documentation must keep pace with the software changes – all documentation must be synchronized as one.

1.7.4 Traceability

In order to evaluate the progress of these quality controls, an open source requirements management system is being used to trace all components of the system.

¹⁰ Center for Devices and Radiological Health, “Information on Releasable 510(k)s”. <http://www.fda.gov/cdrh/510khome.html>

1.8 Code Implementation

The software architecture, design and implementation are of fundamental importance for any large information system. With a clear vision, well defined architecture, modular organization, standard design patterns, and careful coding and documentation, anything is possible.

However usually it becomes too unwieldy for a small team to manage, the team grows, politics and preferences separate the system functionally and ultimately different people come up with different solutions. The data model expands in different directions, shared tables used in different ways, testing coverage falls to the wayside, clients become the real testers and end to end costs rise.

Open source projects often face the same challenges. However, with a critical mass [Linus said](#)¹¹ "Given enough eyeballs, all bugs are shallow.". While reassuring, the goal is to not have to rely on the critical mass at the outset.

1.8.1 Data model

Often information systems create the data model and rarely make major structural changes. This explodes the number of database tables as functionality is added. Typically different development teams add new tables with different concepts in mind. With these large systems, no one person can visualize the entire data model and control is divided amongst the separate teams (further compounding the problem).

With this product, a significant amount of time and care have been spent to optimize the database to be in 3rd normal form and be pure in representation of the software entities. This purity has a profound effect to simplify implementation of the functional components.

1.8.2 Object Relational Mapping

The amount of hand coded SQL was minimized. This was designed to maximize the benefits of Object Orientated Programming, use traditional design patterns, and break the system into manageable components. SQL, while extremely powerful, once written, ties down the data model down such that changing the data model requires identifying all affected SQL. Some systems tackle the issue by using stored procedures, but this splits the development environment into two, adding an additional language. It also creates a dependency on a specific database.

To minimize hand coded SQL this product uses Hibernate, an Open Source ORM product, which provides a façade for the data access layer. Tables are mapped to objects using XML configuration files – one per entity.

As the **data model** dynamically matures (change is expected, and planned for) the XML and objects need to follow suit. Rather than hand code the changes, the development environment uses '**code generation**' to a) read the data model from the database b) read the entity configuration c) generate the XML files and d) generate the new data access object java classes.

As this code generation has matured over time, the process is now simple and very seamless to the expansion of the data model. One of the major benefits to this approach is that prior to making larger changes to the data model – the '**refactoring**' capabilities of Eclipse can be used to change the data access object class definition. Renaming fields, changing data types, adding/removing fields, has become **an order faster**.

¹¹ Torvalds, Linus, "Linus' Law". http://en.wikipedia.org/wiki/Linus's_Law

1.8.3 Layered Architecture

Most successful large systems divide the system into layers to achieve architectural goals. This system is no different with the Client Layer→Middleware→Business Layer→Data Access Layer. The business layer resides on the application server with the middleware provided by the excellent Open Source [JBoss](#)¹² Enterprise Application Server. By using EJB 3.0 the server side code is masked from the complexities of process management, RMI, even XML configuration files or annotations are minimally required.

The client uses the rich Swing framework which has been greatly enhanced in version 1.6 such that it delivers a responsive user interface.

1.8.4 Client Objects

The subclassed data access objects are directly sent to the client over RMI. The client populates and manipulates the object, returning it to be updated by the server. Using the same object, without extensive hand coded mapping removes a large burden for the developer. With a data model flexible to change, this removes significant complexity. Data needs are clearly defined and owned. Moreover the use of Hibernate means that the data objects can represent **complex hierarchal relationships**. Not having to build additional objects for that purpose, and then map to the database by hand is extremely beneficial.

1.8.5 User Interface Code

UI code is notoriously poor, lacking design, performing business logic, intertwined with UI components, growing more and more complex with each feature added. In healthcare the problems are further compounded by the complex workflow of clinicians – each has their own way, their standard, their own requirements, and their own needs out of the user interface. The manage them all is daunting to say the least.

Designing user interfaces using traditional UI builder tools tends to doom the UI to become 'legacy' as soon as the lines of code are generated. Rather than follow that path this product takes an extreme approach.

Model-View-Controller is well known as a design pattern to build user interfaces. This product takes the design seriously by storing the entire User Interface hierarchy of components – Frames, dialogs, panels, panes, lists, menus, and toolbars in the database. Having the configuration as a tree of objects allows for automation of the layout, at runtime, of the user interface.

This design has **far reaching effects** for the expansion and maintenance of the user interface. Apart from removing UI generation code from the core controller classes, the UI can now be maintained through its own user interface. So the tools to build the product are the same tools administrators can use to customize the product. The concept of using the build UI tools to change the build UI tools is quite significant.

These build tools have and will continue to evolve – copying and pasting entire applications to create the base for a new one is an incredible time saving. In addition the healthcare organization can use the same tool to customize the user interface to integrate into their environment. Furthermore the door is opened to **plug-in** being written to create special purpose user interfaces – without affecting the core architecture.

¹² JBoss, "JBoss Application Server". <http://www.jboss.org/products/jbossas>

1.8.6 Data binding

With a single set of code to generate the UI, the ability to integrate data binding became another critical improvement. Data binding is an essential component of Model-View-Controller as it provides for a data object to be represented by a view, such that updating the view (text box, combo, list, table) should automatically update the object and vice versa. In general data binding appears to be a complex problem however with this architecture it became possible.

With a hierarchy of objects one challenge is to be able to reference specific fields on specific objects further down the hierarchy. Finding the right object is handled a few different ways, outside the scope of this document but finding a field on the object is important. In this architecture every column on every table is represented by an entry in the reference table under a group called ModelReference. The UI build tools have a list of these values so that a control is 'bound' to a object field. Since code generation is used for data objects, providing mapping for that reference value to the field was trivial. The UI generation code ties the controls and objects together at runtime.

The compound effect of the code generation for Hibernate, Data Objects, coupled with the dynamically built user interface is that there is **no hand written code to store data entry values in the database.**

The net effect is that one developer can create new applications at an astounding rate with near bug-free results. What bugs are uncovered, are central and quickly fixed.

Take an inpatient registration form as an example. These screens along with the insurance form represent a significant number of data elements over several tables with a number of relationships. The hierarchal relationships are built out in our code generation database.

This screenshot shows the 'Demographics' tab of a patient registration form for Greg Caulton. The form includes fields for Last Name, First Name, Middle Name, Gender, DOB, Race, Ethnicity, Religion, Primary Language, Nationality, and Marital Status. It also has sections for Address and Phone with corresponding input fields and a table for phone numbers.

This screenshot shows the 'Visit Details' tab of the patient registration form. It includes fields for Visit Status, Type of Visit, Hospital Service, Admission Type, Admission Source, Admission Date, Accommodation, Discharge Date, Discharge Location, and Discharge Disposition. A calendar widget is visible for the 'Admitted' date, showing June 2007. There are also fields for 'Physicians' and 'Family'.

This screenshot shows the 'Emergency Contact' tab of the patient registration form. It includes fields for Last Name, First Name, Birth Date, Address, and Phone. There is also a section for 'Next of Kin' with similar input fields.

The code listing on the following page represents all the client code supporting this data entry form.

1.8.7 UI code sample

This is the registration User Interface code as an example of the net effect of hibernate and data object code generation, data binding, and dynamic UI generation. Any form specific dynamic logic is written in JavaScript and stored with the rest of the UI definition. The script has references to the model itself so updating the model is immediately reflected on the screen.

The code is stripped of comments for brevity.

```
public class RegistrationController extends BaseController {

    private PatientModel originalPatient = null;
    private static RegistrationController controller = null;

    private RegistrationController() {

    }

    public static RegistrationController getInstance() {
        if (controller == null) {
            controller = new RegistrationController();
            controller.setDefaultBaseModel(new PatientModel());
        }
        return controller;
    }

    public void start(PatientModel patient) throws Exception {
        this.originalPatient = patient;
        super.getDefaultBaseModel().clear();
        super.getDefaultBaseModel().copyAllFrom(this.originalPatient);
        super.startFrame(ApplicationDialogReference.REGISTRATION);
    }

    @Override
    protected void mediateMessages() {
        getFormMediator().register(new IReceiveMessage() {
            public boolean receive(ISEvent event, Object value) throws Exception {
                switch (event) {
                    case EXECUTEACTION:
                        BaseAction action = (BaseAction) value;
                        switch (action.getActionReference()) {
                            case OKSUBMITFORM:
                                save();
                                return true;
                            case CANCELSUBMITFORM:
                                getDialogModel().setVisible(false);
                                return true;
                            case SYSTEMREMOVE:
                                removeSelectedTableRows(action.getContextRefId());
                                break;
                        }
                }
            }
        });
    }
}
```

Continued...

```

        case SYSTEMADD: addTableRow(action.getContextRefId());
            break;
        case SYSTEMEXPORTFORMDATA:
            action.setBaseModel(getDefaultBaseModel());
            forward(event, action);
            return true;
        case SYSTEMIMPORTFORMDATA:
            action.setBaseModel(getDefaultBaseModel());
            forward(event, action);
            refreshAllTables();
            return true;
    }
    break;
}
return false;
}
public String toString() {
    return "RegistrationController.mediateMessages";
}
}, this);
}

private void save() throws Exception {
    originalPatient.copyAllFrom(getDefaultBaseModel());
    getDefaultBaseModel().validateDataModel();
    PatientService.store(originalPatient);
    getDialogModel().setVisible(false);
}
}

```

1.8.8 Reference data

Throughout all business logic and controller classes, reference data becomes an integral part of the code. The reference data which ships with the source code is all stored in the database. This allows the 'display' values to be changed to the language of choice, it allows reference data to be grouped and organized to meet the needs of the healthcare organization.

Having to 'look up' specific reference values in the database based upon a string value becomes problematic as the data model and reference data is reorganized as the product evolves. Ideally any changes should immediately be reflected in the code with compile time errors if problems are made.

This is achieved through code generation. All of the reference values which are predefined as 'system reference' i.e. essential to the running of the system are reflected as enum values. As an example above the 'actions' are enums, responding to UI events. The OKSUBMITFORM is a reference value, in group action associated with the OK button using the form builder.

The benefit of this is **compile time safety** for a significant portion of business logic.

1.9 Validation and Testability

The ability to automate software testing is highly desirable, and with an HIS, rarely attainable. Automated test coverage for large systems is likely less than 10% of the code paths. Testing in general of information systems is tricky due to the high level of integration with the database, and complex workflow. Some systems use rudimentary screen scraping and window key events via scripting to manipulate the UI to simulate users. This is very time consuming with limited results.

1.9.1 User Interface Test Automation

With the user interface code centralized, database configuration, and MVC driven the ability to directly manipulate the user interface, significant opportunities are available.

As an example the following code will open the registration form, populate the data model, giving a little time to check the UI controls worked ok and then save the patient to the database.

```
RegistrationController.getInstance().start(new PatientModel());
PatientModel patient = (PatientModel) RegistrationController.getInstance().getDefaultBaseModel();
```

```
RandomGenerator.populateStrings(patient);
patient.getPatientName().setLastName("Testing");
patient.getPatientName().setFirstName("Automated");
```

```
AddressModel address = new AddressModel();
address.setStreet("100 some street");
address.setAddressTypeRef(new DisplayModel(AddressTypeReference.HOME.getRefId()));
IdentifierModel ssn = new IdentifierModel();
ssn.setIdvalue("121-122-1111");
patient.getIdentifiers().add(ssn);
patient.getAddresses().add(address);
```

```
RegistrationController.getInstance().getFrame().repaint();
Thread.sleep(10000);
BaseAction action = new BaseAction();
action.setActionReference(ActionReference.CANCELSUBMITFORM);
RegistrationController.getInstance().getFormMediator().receive(ISEvent.EXECUTEACTION, action);
```

1.9.2 User Simulation

An essential implementation task is testing and validation. Using the UI to register the patients exercises and tests those UI components checks for memory leaks or other more insidious issues only found if the UI is run hundreds of times.

The "holy grail" is the ability to truly simulate large numbers of users on the system. Using some basic configuration options such as # inpatients per day, # outpatients per day, # of beds etc. the number of registrations per hour that the system needs to support is determined. For example an 800 bed hospital might require one patient registered every 10 seconds. By applying a Poisson distribution the simulation produces the load on the server by firing off randomly populated patients (based upon rules). By applying business rules to the simulation large steps are taken towards performance testing.