OpenMRS Manual

Contents

- 1 Preface
- 2 Introduction
  - 2.1 Database API
  - 2.2 Useful links:
  - 2.3 OpenMRS Team
  - 2.4 Requirements
    - 2.4.1 OpenMRS
    - 2.4.2 OpenMRS-MD
  - 2.5 Installation for Implementers
  - 2.6 Ongoing Support
- 3 Installation and Configuration
- 4 Overview
  - 4.1 Steps Involved
  - 4.2 Minimum Requirements
- 5 Install Firefox
- 6 Install Java
- 7 Install Tomcat
- 8 Install MySQL
- 9 Build the OpenMRS Database
- 10 Runtime Configuration
- 11 Deploy OpenMRS
- 12 Install Navicat
- 13 Starting your OpenMRS program
- 14 Troubleshooting
- 15 User's Guide
  - 15.1 FormEntry
  - 15.2 Concept Dictionary
- 16 Administrator's Guide
- 17 Form Design Process
  - 17.1 Define/Locate Concepts
  - 17.2 Create Form MetaData
  - 17.3 Design Schema
    - 17.3.1 Form Schema Requirements
  - 17.4 Other Resources
  - 17.5 Adding A New Patient Identifier Field
  - 17.6 Overview
  - 17.7 Design Form
  - 17.8 Publish Form
  - 17.9 Editing a Form
  - 17.10 Duplicating a Form
  - 17.11 Moving Forms Between Servers
  - 17.12 Other Resources
- 18 Technician's Guide
  - 18.1 OpenMRS Server Installation Overview
  - 18.2 Minimum Reqiurements
  - 18.3 OpenMRS Server Installation Step-By-Step

  - 18.4 Latest revision: 1.0β (03/19/06)
  - 18.5 What's the difference between concept_set and concept_set_derived tables?
  - 18.6 Understanding by example
  - 18.7 Why all this monkey business with concept_set_derived?
  - 18.8 Designing Forms
  - 18.9 Submitting Forms
- 19 Glossary
- 20 Appendix A
  - 20.1 Calculate a check digit
  - 20.2 Why bother with check digits?
  - 20.3 What is the Luhn algorithm?
  - 20.4 Our variation on the Luhn algorithm
    - 20.4.1 Here's how we handle non-numeric characters
- 21 Appendix B
  - 21.1 See also
- 22 Appendix C
- 23 Appendix D
  - 23.1 Beginning Definitions
  - 23.2 Why I should care about the concept dictionary
  - 23.3 Creating new concepts for the concept dictionary
  - 23.4 Further reading
  - 23.5 Modeling diagnoses

Preface

Documentation Preface Documentation Authors and Contributors

Introduction

OpenMRS is an application which enables design of a customized medical records system with no programming knowledge (although medical and systems analysis knowledge is required). It is a common framework upon which medical informatics efforts in developing countries can be built. The system is based on a conceptual table structure which is not dependent on the actual types of medical information required to be collected or on particular data collection forms and so can be customized for different uses. OpenMRS is based on the principle that information should be stored in a way which makes it easy to summarize and analyze, i.e. minimal use of free text and maximum use of coded information. At its core is a concept dictionary which stores all diagnosis, tests, procedures, drugs and other general questions and potential answers. OpenMRS is a client-server application which means it is designed to work in an environment where many client computers access the same information on a server.

At the moment much of the information on these pages is aimed at programmers/developers; however an aim is to provide comprehensive information for:

- Programmers/developers: The people who do the actual programming of the application and who have designed and modify the table structure.
- Implementers: The people who decide what sort of medical records need to be collected at a particular health facility. They use the application to adjust the concept dictionary as required and to design forms to collect this information. Ideally the designers for a particular health facility should be a team of systems analysts who understand the conceptual structure of the system and clinicians who understand medicine and their own health facility's medical record information requirements.
- Users: The people use the system to enter, retrieve and analyze information in the system in their particular health facility.

There are several layers to the system:

1. The OpenMRS data model -- our work borrows heavily from the Regenstrief model, which has a 30-year history of proven scalability and is based on a concept dictionary
2. A database API -- the database API provides a programmatic wrapper around the data model, allowing developers to program against more simplified method calls rather than having to understand the intricacies of the data model
3. An application API -- the application API uses collections of database API functions to present higher-level functions for developers
4. Application layers, including web front-ends and other modules -- these are the user interfaces and applications themselves built upon the lower levels

Database API

Most of the API is provided in "services." First, you obtain a `Context` from a `ContextFactory`, then obtain any services from the `Context`.

See the javadoc for details.

OpenMRS Team

Current team members:

- Christian Allen
- Paul Biondich
- Hamish Fraser
- Darius Jazayeri
- Burke Mamlin
- Justin Miranda
- Ben Wolfe

Are you interested in joining the team? Check out our Help Wanted ad. :o)

Requirements

Requirements as of September 2006

OpenMRS

- Tomcat expertise
    - Install and manage Apache Tomcat
    - upload and install new WAR files
    - troubleshoot, read log files
- Database expertise
    - Install and manage MySQL environment
    - understand the OpenMRS data model
    - perform SQL queries and run SQL scripts
- Clinical form design
    - understanding of how to create meaningful, useful, and non-ambiguous questions/answers
        - Medical expertise -- to understand what questions/answers make sense, what's clinically relevant
        - Technical expertise -- to understand how questions/answers can be interpreted by a computer
        - Data Management expertise -- to understand how questions/answers will be used for reporting, research, etc.
- Dictionary design
    - ability to infer dictionary concepts from a form (both coded questions and answers), modeling expertise -- e.g., do you create `CHEST PAIN` as a boolean (true/false) or do you create a `CARDIAC REVIEW OF SYSTEMS` as a coded concept with `CHEST PAIN` as a possible answer?
- InfoPath expertise
    - Install and manage Microsoft InfoPath
    - Advanced Form design
    - Understand basic XPath functions
- Ability to install and configure Apache + SSL *(if extending network beyond a single LAN)*

OpenMRS-MD

This presumes that there will be a pre-baked set of concepts, forms, and reports within the OpenMRS-MD starter database.

- Ability to install, manage, and troubleshoot Tomcat and MySQL
- Create and manage users and roles

Installation for Implementers

1. Design paper encounter forms (getting input from clinical and IT teams)
2. Install server
    1. Server w/ power backup -- UPS for server and power backup (?solar)
    2. Install Windows Server 2003 on server along with supporting software (e.g., antivirus, firewall)
    3. Install OpenMRS system (mysql, apache, tomcat, etc. — see Setting up an OpenMRS Server)
    4. Configure server to serve OpenMRS web application through Apache over HTTPS
3. Setup OpenMRS core data set
    1. Install core data set (using SQL)
    2. Make user account and define privileges
    3. Define locations
    4. Define tribes
    5. Define encounter types
    6. Build dictionary concepts around forms
4. Design electronic form(s) within OpenMRS
    1. Define forms within OpenMRS
        - *Must follow current structure limitations unless another XSLT transform from XML to HL7 can be designed locally)*
    2. Download blank InfoPath form based on form schema
    3. Design InfoPath form
        1. Drawing form
        2. Building logic
    4. Test form(s)

5. Build LAN infrastructure
    1. Ethernet cables, hubs, etc. as needed
6. Setup client workstations for data entry
    1. Firefox
    2. InfoPath 2003 with Service Pack 2 or later
7. Install VSAT for remote support and/or data entry

Ongoing Support

1. Review new concept proposals
2. Make changes to forms over time (both paper and electronic versions)
3. Building reports from data exports
4. Managing user accounts
5. Server maintenance
6. Client maintenance
7. VSAT maintenance

Installation and Configuration

Overview

NOTE: we are in the process of making an install wizard that will greatly simplify the installation process

Steps Involved

1. Install Firefox
2. Install Java 5 runtime environment (JRE)
3. Install Tomcat 5.5+
4. Install MySQL 5+
5. Build the OpenMRS database
6. Create a runtime configuration
7. Install OpenMRS

Minimum Requirements

1 GHz processor or better, 256 MB of memory or more, 40 GB hard drive or larger. You can set up the server on a laptop for demonstration or testing purposes. For production usage, we recommend one or two processors 1.5+ GHz, 2 GB of memory, and 150+ GB of disk space with RAID and appropriate backup facilities.

Install Firefox

1. Download the latest stable release of Firefox and run installation program
2. Accept the license agreement
3. Select Standard or Custom installation to install to c:\Program Files\Mozilla Firefox

Install Java

1. Download the latest stable release of the Java Runtime Environment (JRE)
2. Run the install program (e.g., jre-1_5_0_07-windows-i586-p.exe)
3. Accept the license agreement and default installation directories

Install Tomcat

1. Download latest stable release of Tomcat — e.g., [http://apache.tradebit.com/pub/tomcat/tomcat-5/v5.5.17/bin/apache-tomcat-5.5.17.exe apache-tomcat-5.5.17.exe) and execute file
2. Accept the license agreement
3. Accept Component defaults (Tomcat, Start Menu Items, Documentation)
4. Accept default destination folder: C:\Program Files\Apache Software Foundation\Tomcat 5.5
5. Accept HTTP/1.1 Connector Port (8080)
6. Set Administrator login (admin/password)
7. Accept path for the JRE (C:\Program Files\Java\jre1.5.0_07) or navigate to the actual path and select
8. Select "Install Tomcat"
9. Open the Tomcat users file (c:\Program Files\Apache Software Foundation\Tomcat 5.5\conf\tomcat-users.xml) in a text editor and add the roles admin and manager for the user "tomcat"

```
<user username="tomcat" password="tomcat" roles="tomcat,admin,manager"/>
```

10. *(Optional)* Set Tomcat to start automatically

1. Start → Settings → Control Panel → Administrative Tools → Services
2. Right Click "Apache Tomcat" → Properties → Set "Startup Type" to Automatic

Install MySQL

1. Download the latest stable (recommended, generally available) release of MySQL — e.g., for "Windows (x86)" → mysql-5.0.22-win32.zip
2. Extract with Winzip
3. Run the MySQL install program (Setup.exe) and accept defaults (Typical or Complete Setup)
4. Accept the license agreement
5. Accept to Configure Instance → Select `Detailed Configuration' → Developer Machine → Multifunctional Database → InnoDB Settings (c: and Installation Path) → DSS/OLAP → Enable TCP/IP Networking (Port 3306) → Select "Manual Selected Default Character Set / Collation" and set the character set to utf8 → Check "Installed as Windows Service" → Root password (password) **(note: do NOT forget this root password for MySQL! You will need it later)**
6. Execute the configuration, the installer should step through each step and give you green check marks at each stage
7. If you are stopped with a message saying that the server could not be started, your Windows Firewall may be blocking the MySQL port (default port is 3306)
    1. Open Windows Firewall (Start → Settings → Control Panel → Windows Firewall)
    2. Under the "Exceptions" tab, click the "Add Port" button
        - Name: **MySQL**
        - Port number: **3306**
        - *(optional)* for added security, click the "Change scope" button and limit to "My network (subnet) only" — this will prevent computers outside of your local area network from being able to access your database directly

Build the OpenMRS Database

1. Download the latest demo script (currently here)
2. Extract with WinZip
3. Open the MySQL Command Line Client (Start → Programs → MySQL → MySQL Server 5.0 → MySQL Command Line Client)
4. Enter your MySQL root password (defined as you installed MySQL)
5. Run the demo script

```
source c:\path\to\script\openmrs-all.sql
```

Runtime Configuration

1. Using a text editor, create a runtime configuration file
    - We typically name this file OPENMRS_RUNTIME_PROPERTIES.PROPERTIES
    - You can get the basic (default) settings from this page
2. Now you must create a system variable that directs the OpenMRS application to the runtime configuration file
    - Right-click on the My Computer icon (typically on the desktop or under the Start menu) and select "properties"
    - Under the "Advanced" tab, click the "Environment Variables" button
    - You should see two sections: one for user variables and another for system variables
    - Click the "New" button within the *System* Variables section (near the bottom)
      Variable name: **OPENMRS_RUNTIME_PROPERTIES_FILE**
      Variable value: **C:\path\to\your\OPENMRS_RUNTIME_PROPERTIES.PROPERTIES**
      (note: you will need to specify the path to the runtime configuration file you created earlier)
3. Reboot **(you must reboot for the new system variable to be available to all programs)**

Deploy OpenMRS

1. Ensure that Tomcat is started
2. Navigate to http://localhost:8080/manager/html and enter your Tomcat administrator creditials (username and password chosen when installing Tomcat)
3. In the Tomcat Web Application Manager, enter the location of the OpenMRS WAR file (openmrs.war) to deploy. (Note that the

OpenMRS.WAR file is most easily downloaded with Mozilla FireFox. Internet Explorer tries to open the file as a Zip file). The deployment could take some time while the file is copied to the folder c:\Program Files\Apache Software Foundation\Tomcat 5.5 \webapps and decompressed.

4. At the end of this process, the web page will refresh and /openmrs should be displayed under Applications. Apache Tomcat should also start the application (Running = True; and in Commands, Stop is underlined)

Install Navicat

This step is optional. Navicat is a commercial tool we have found handy for interacting with MySQL. If you prefer a free alternative, you could use the Query Browser provided by MySQL.

1. Install Navicat (navicat2005ent.exe or get trial version from Navicat's website)
    1. Accept default location (C:\Program Files\PremiumSoft Navicat)
    2. Accept Program shortcut (PremiumSoft Navicat)
    3. Create a desktop icon
    4. Select Install
2. Configuration
    1. Start Navicat
    2. select Connection
    3. Enter Connection Name (Localhost) and select defaults
       Host Name / IP address: **localhost**
       Port: **3306**
       User name: **root**
       Password: **password**
3. Create Database (Navicat) — *if not created previously*
    1. Right Click localhost and select "Execute Batch File"
    2. Run the openmrs-all.sql script that is inside the Image:Openmrs-all.zip file.

Starting your OpenMRS program

After you have finished deploying OpenMRS in Tomcat, and it is being displayed under applications, you can click on /OpenMRS tab (on the left side of the Tomcat Manager window) to start your application. You will need to login initially using Username: **admin** Password: **test** (both are in lowercase). Alternatively, while Tomcat is running you can start OpenMRS by entering http://localhost:8080/openmrs/login.htm (assuming 8080 is your port number for Tomcat - insert the appropriate port number if it is not 8080).

Troubleshooting

When uploading the war file, Tomcat hangs and stops responding

: This typically occurs when you have not defined a MySQL user account that OpenMRS can use to access the database *or* you have not granted this user full access to the openmrs database. The default username is *test* with password *test*. The default username/password can be overridden in the OPENMRS_RUNTIME_PROPERTIES.PROPERTIES configuration file. To fix this: use Navicat or the MySQL administration tool (available from mysql.com) to verify that you have a user with username "test" and password "test" and this user has full access to the openmrs database.

Cannot connect to Tomcat on port 8080

: This port is sometimes used by other programs, such as Popfile and TivoServer. You can use the Windows "netstat -ao" command to discover if another process is using port 8080.

As of Oct 13, 2006, database and WAR files are not compatible

: When you run OpenMRS the software and database versions are shown in very small text at the bottom of the browser window.
The openmrs-all.sql file mentioned above is for version 1.0.26. It seems to work with the WAR file for 1.0.25 found by following the link mentioned above. The latest WAR file is 1.0.42, and you can use it if you adjust the database as follows. Go to http://trac.openmrs.org/browser/trunk/metadata/model and find the file openmrs_1.0.0-to-latest-mysqldiff.sql. Download this file by clicking on the link and using the Plain Text link at the bottom of the page. Edit out about half a page of code at the end that pertains to version 1.0.43. Build the database as mentioned above with "source openmrs-all.sql" and then "source" the file you just edited.

User's Guide

All OpenMRS users must authenticate to the system using a username and password. If you do not know your username, you should contact your system administrator. If you have forgotten your password, the system may allow you to reset your password by using a secret question mechanism.

FormEntry

Using FormEntry Finding a Patient Selecting a FormEntry Form Completing a FormEntry Form

Concept Dictionary

Viewing the Dictionary

Administrator's Guide

Administering Users Administering Patients Identifiers are used uniquely identifier patients within the system. Different types of identifiers are distributed by various health care systems. Some of these systems will be within your control, so you will be able to control how identifiers are created and distributed; however, there will likely be identifiers that are not within your control but are useful to record within your system to aid in patient identification.

1. Go to Administration → Manage Patients → Manage Identifier Types
2. Click on "Add Patient Identifier Type"
3. Provide a name the new identifier type
   - The name should be specific to the authority providing the identifiers — e.g., "Wilson Hospital Medical Record Number"
4. Provide a clear description of the identifier types, including identifying the authority responsible for distributing the identifiers
5. Optionally provide a format for the identifiers
   - The "Regex Format" allows you to enforce a pattern for identifiers using a Regular Expression (describes the identifier pattern in a way the computer can understand) — e.g., "\d{1,8}-\d" would allow 1 to 8 digits followed by a dash and a single digit
6. The "Description of Format" allows you to describe the required pattern in a way that users of the system can understand — e.g. "Must follow the pattern NNNNNNNN-N (up to 8 digits followed by a dash and a final digit)"
7. Specify whether the identifier is required
8. Specify whether check digits should be enforced for this identifier type
9. Click on the button to save your new identifier type

Administering Encounters Administering Observations Administering Orders Administering Concepts *NOTE: if you are running your server on Linux, please see the documentation about Installing An OpenMRS Server On Linux.*

Form Design Process

Define/Locate Concepts

This is a gradual process that involves identifying/constructing concepts within your concept dictionary. This assumes that you have an available paper form for use in a clinic.

To identify concepts

1. Log into OpenMRS.
2. Enter the Dictionary use case.
3. Review your existing paper form.
4. For each field on the paper form, search the Concept Dictionary to locate an existing concept.
5. Mark the concept number on the paper form.
6. If a corresponding concept does not exist, add it to the concept dictionary.

Create Form MetaData

This process involves using the OpenMRS application to populate the metadata related to your form. You can create a new form or duplicate an existing form (which allows you to reuse an existing form's metadata and, more importantly, its schema).

To view all forms in the system

1. Log into OpenMRS as an administrator.
2. Select the Administration link in the top navigation menu.
3. Select Manage Forms under the Forms link section.

To create a new form, follow these instructions:

1. Select the Add Form link at the top of the page

   NOTE: currently, the form submission engine requires encounter-based forms with a specific hierarchy (contained in the Basic Form definition); therefore, you should start new forms by copying the Basic Form and adding observations to that foundation.

To duplicate an existing form , follow these instructions:

1. Select an existing form from the Duplicate Form field — e.g., the Basic Form contains all of the currently required fields in the proper hierarchy to be handled properly by the FormEntry engine.
2. Click the Duplicate button.
3. Update the form metadata, including the form name.

NOTE: While editing the form schema, ensure that the Published checkbox is **unchecked**. When we are ready to make the form available through the Form Entry module, we will need to check this checkbox.

Design Schema

This process involves using the OpenMRS application to create a schema for your form.

To add new concepts to a form schema:

1. Choose the Design Schema use case.
2. Type the desired concept into the Find Field Elements search box (consult Define Concept section above). The search results should automatically display in the area below the search box.
3. Press the Enter key. The search box and search result index (i.e. the number to the left of the search result) should show be highlighted in gold.
4. Type the search result index in the search box.
5. Press the Enter key.

NOTE: You can also drag n' drop the desired field element from the search result box to the form schema. However, when starting with a blank form schema it is difficult to locate the place to "drop" the field element. You can also double-click the desired field element, but this requires you to "update" the its metadata.

To move form elements within the form schema:

1. Click the concept in the form schema
2. Move the mouse to the desired location.
3. Drop the form element into the desired location in the form element tree.

To update form element metadata:

1. Right-click the desired form element in the schema.
2. Click 'Edit Field'.
3. Select 'Edit for this form only' to edit the form element for the current form.
4. Enter the appropriate information (in most cases you will only need to change Multi?, Field #, Field Part, Page #, Min, Max, and Required).
   - Multi - Indicates whether the schema should allow multiple values for a single form element (i.e. multiple answers to a single question like "What medications are you currently taking?").
   - Field # - The field number corresponding to the actual form element in your form. This value may be left blank.
   - Field Part - The sub field number/letter corresponding to the actual form element in your form (i.e. 'a' in 1a). This value may be left blank.
   - Page # - The page number where this form element appears. This value may be left blank.
   - Min - The minimum number of times that this form element should appear on the form. This value may be left blank. Default = 0.
   - Max - The maximum number of times that this form element should appear on the form. This value may be left blank. Default = 1. Use -1 to allow for an arbitrary number of values.

- Required - Indicates whether the form element should be required.

To delete the form element from the schema:

1. Select the X icon next to the form element name.

Form Schema Requirements

At this point, the form schema is fairly constrained by the XSLT that translates the submitted form data (within FormEntry) into HL7. So, if you are designing forms for FormEntry (for use with InfoPath), you must follow these guidelines. Initially, we thought we'd be making up a separate XSLT for every form; however, a single XSLT has gotten us much further than anticipated.

First of all, you want to start with the basic form schema. Here are some guides...

- Form schemas should have top level nodes: PATIENT, ENCOUNTER, OBS, PROBLEM LIST, and ORDERS
- Most of the children in the PATIENT section (in the starter schema) are necessary, but I think only patient.patient_id is required
- While the XSLT can handle alternate identifiers (there's an example of our MTCT-PLUS identifier in the default XSLT), InfoPath is not the desired way to add identifiers to the system. Rather, we'd encourage you to edit patient demographics through the patient administration functions of the web application.
- In the ENCOUNTER section, encounter_datetime, location_id, and provider_id are required for things to work.
- The OBS section should linked to the concept MEDICAL RECORD OBSERVATIONS (in the schema designer, you should see the concept id for MEDICAL RECORD OBSERVATIONS in parentheses after "OBS" -- e.g., on the demo site, you'll see "OBS (1238)" for the OBS section.
- Within the OBS section, you may place either simple observations (with coded, date, boolean, numeric, or text datatype) or sets. Sets should contain 1 or more chidren that are all simple observations. The XSLT does not support sets-within-sets, so your OBS section may contain elements and elements w/ children, but should not go any deeper.
- PROBLEM LIST (if you use it), should be just like the starter schema (problem list with two children: problem added and problem resolved). If you are not collecting diagnoses on your form, you could try deleting this section in the schema, but I'd probably just leave it there and not use it on the form.
- The ORDERS section should contain only sets (no simple observations directly under ORDERS) following the design in the starter schema. Again, only one level is supported -- no sets within sets. At this stage, we are converting all of these orders to observations and have not completed a path to actually generate entries in the order tables from an HL7 message (i.e., an InfoPath form submission).

Other Resources

- See Administering FormEntry

Adding A New Patient Identifier Field

You can look at pre-existing identifier fields in your system or on the demo website for examples. If the patient identifier type has already been defined within a field, then you should try to re-use the existing field if possible; otherwise, follow these steps to create a new identifier field:

1. Go to Administration → Manage Fields
2. Click "Add New Field"
3. Create your field with the following information

Field Name	Enter a name for the field, e.g. "Medical Record Number"
Description	Describe the field to help other administrators and users — e.g., "Unique patient identifier for Wilson Hospital"
Field Type	Database element
Database	Table = `patient_identifier`, Attribute = `identifier`

Default value	`$!{patient.getPatientIdentifier(1).getIdentifier()}`, replacing the **1** with the internal identifier type id (you can find this under "Manage Identifier Types", hovering your mouse over the identifier type and looking at the link's address)
Select Multi	Leave unchecked

NOTE: typically, patient demographic fields (like an identifier) should be placed under the "PATIENT" section of a form schema.

Administering Reports

Overview

The OpenMRS FormEntry module uses Microsoft® InfoPath™ to gather data for the repository.

- You must have dictionary concepts defined for each question and answer on your form.
- The form definition is created by defining a hierarchy of fields that will be used on the form.
    - The layout of the form hierarchy currently has some restrictions imposed by how the forms are eventually translated into the database
        - Form hierarchies should have the following sections (in order)
            - **PATIENT —** *as a section field*
            - **ENCOUNTER —** *as a section field*
            - **OBS —** *as a concept field with the concept MEDICAL OBSERVATIONS*
            - **ORDERS —** *as a section field*
        - There are certain fields that are required for an encounter and these must be present within the form hierarchy under the appropriate section.
        - We have created the "Basic Form" as a starting point. The basic form contains all of the required fields to get a form working. You should always start a new form by making a copy of the Basic Form and giving your copy a new name.

Design Form

1. If you have not just finished designing the schema...
    1. Log into OpenMRS as an administrator
    2. Navigate to Administration → Forms
    3. Click on the Edit Metadata link next to your form name
2. Click on the Download XSN link
3. Do *not* open the file with InfoPath (the default action); rather, save the file to disk
4. Right mouse click the saved XSN file and click 'Design' to open it in Microsoft Infopath in design view

To add form controls:

1. Select Data Source from the right navigation menu.
2. Right-click a form element from the Data Source menu.
3. Drag-and-drop the form element into the main content section.
   NOTE: for typical observations, you may want to drag the chlid "value" element instead of the entire observation element
4. Select a form control from the context menu that is displayed.

To add layout elements:

1. Move the blinking cursor to the content area where you would like to add a new layout element
2. Select Layout from the right navigation menu.
3. Select a layout option from the Layout menu.

To get just the concept name displayed you need to use some xpath scripting:

1. Right click on the input box in design mode and select "Expression Box Properties"
2. On the "General" tab, in the "Data Source" section, in the "XPath:" textbox, paste:

```
substring-before(substring-after(., "^"), "^")
```

Publish Form

To publish your form to the web (general instructions):

1. Save XSN to filesystem (assuming you are still working within InfoPath).
2. Close InfoPath.
3. Log into OpenMRS as an administrator and go to Administration → Forms
4. Click on the Upload XSN link
5. Browse to your XSN and upload it

Editing a Form

DO NOT design the XSN that the server uses. The folder specified in the runtime properties for OpenMRS (via the formentry.infopath. output_dir property) is for use ONLY by OpenMRS. You should never manipulate these files directly. In order to get files into that folder, use the "Upload XSN" function. In order to get files out of that folder, use the "Download XSN" function.

Duplicating a Form

Using the "duplicate form" feature under the form administration screen only duplicates the form schema at this point. Ben will be adding XSN duplication soon (see this ticket). In the meantime, you can duplicate a form *and* XSN using these steps (adapated from Andy's message):

1. Duplicate the Schema by using the duplication option from the Admin → Form screen
   Note: this duplicates build number as well, reseting the build number is not necessary, but can be done through direct manipulation of the data within the database — e.g., using Navicat or the MySQL command line.
2. Download the XSN for the original form and save it to disk. Make a copy of the original form amd give it a new name - this will be the first version of the duplicate form
3. Open the XSN of the duplicate form in *Design mode* (right mouse click the file and choose design) and select **File → Extract form files** from the menu to save the form as separate files. Then close infopath.
4. You will now see a folder with the name of your duplicate form. Inside this folder you will see some files. Edit the FormEntry.xsd file with a text editor (wordpad or notepad) and find the line:

   ```
   <xs:attribute name="id" type="xs:positiveInteger" fixed="n" use="required" />
   ```

   where *n* is the id of the form.
5. Change the *n* within `fixed="n"` to the form id of the duplicate form (the new form id can be found by (a) examining the URL for the new form within OpenMRS' Admin → Form screen, e.g. hovering over the "metadata" link for the new form and looking at the end of the URL in the browser's status bar at the bottom) OR (b) looking up the form table in the MySQL back end. Save and close FormEntry. xsd
6. In the same folder, right-click on the Manifest.xsf file and selected Design.
7. Choose File, Save As and save the form to disk with another name to get the second version of the duplicate form. You can delete the first version of the duplicate form. Upload the second version of the duplicate form into OpenMRS.

OpenMRS uses the form id within the FormEntry.xsd file (compressed within the XSN) to determine the form to which the XSN belongs.

Moving Forms Between Servers

OpenMRS does not currently support the transfer of FormEntry forms between servers using different dictionaries or different form definitions

Assuming the two servers are using the same dictionary concepts and form and field definitions (all internal identifiers aligned), you need only upload the XSN to the destination server. If there are subtle differences (different answers to a concept on the desintation server or an additional

field in the schema), you *might* be able to get the form working by uploading the XSN to the destination server, then downloading it and uploading it again -- we plan to add a "Rebuild XSN" link to simplify this process.

Uploading an XSN (also known as "publishing an XSN")
: updates the URLs and advances the build number (the last number in the InfoPath solution version inside the XSN files). Uploading does *not* change the schema or templates.

Downloading an XSN
: refreshes the schema and templates with the latest data from the form schema definition and the concept dictionary. Downloading does *not* change the build number and does not mess with the URLs since editing in InfoPath resets the URLs anyway.

Editing with InfoPath
: changes the internal URLs anytime you save the file. Anytime an XSN is saved from InfoPath, it must be uploaded ("published") to correct the URLs before it can be used to collect data within an OpenMRS system.

Other Resources

- InfoPath Resources
  - InfoPathDev.com
  - InfoPath SDK

Administration Maintenance Backing up your data

Technician's Guide

OpenMRS Server Installation Overview

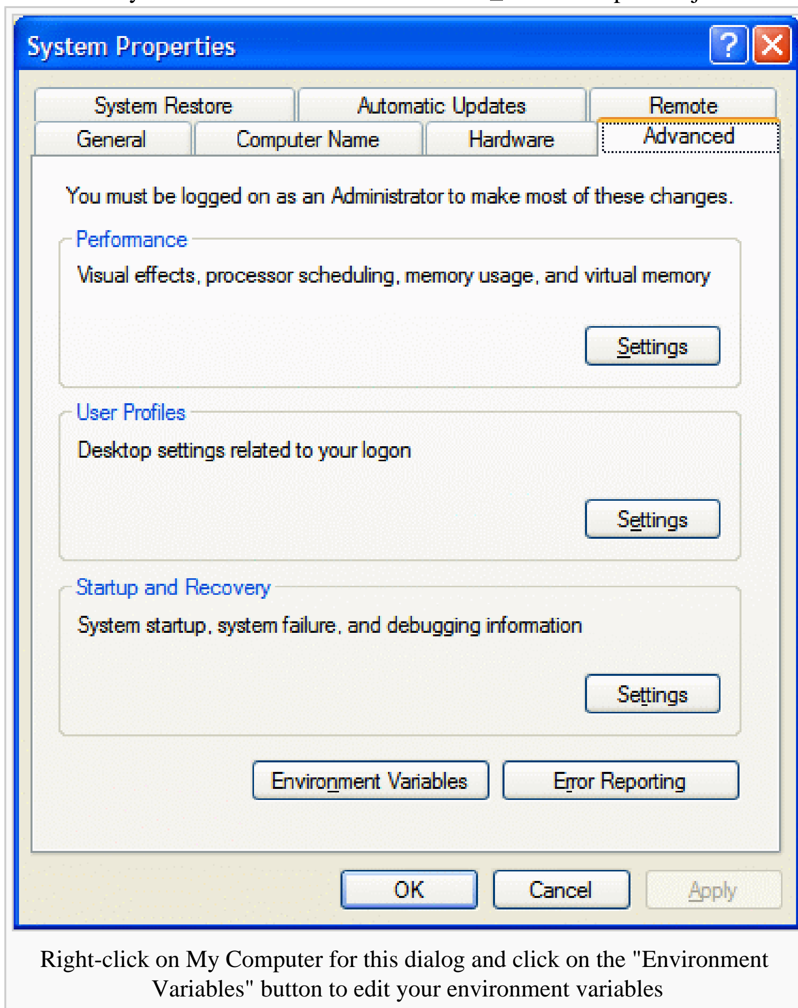
1. Install Java 5
2. Install Tomcat 5.5+
3. Install MySQL 5+
4. Install OpenMRS

Minimum Reqiurements

1 GHz processor or better, 256 MB of memory or more, 40 GB hard drive or larger. You can set up the server on a laptop for demonstration or testing purposes. For production usage, we recommend one or two processors 1.5+ GHz, 2 GB of memory, and 150+ GB of disk space with RAID and appropriate backup facilities.

OpenMRS Server Installation Step-By-Step

1. Install Java SDK
   - Set system environment variable JAVA_HOME = /path/to/javasdk



Right-click on My Computer for this dialog and click on the "Environment Variables" button to edit your environment variables

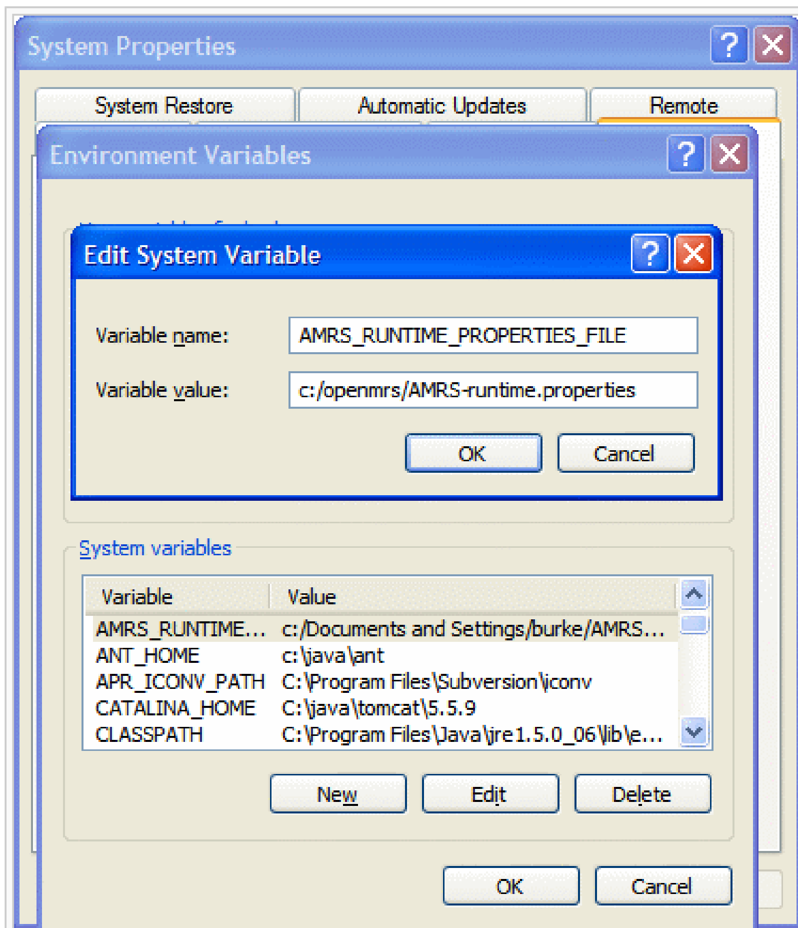
2. Install Tomcat
   - You will need to know the port used by tomcat (default is 8080)
3. Install MySQL (must be 4.1 or higher, 5.x recommended)
   - You will need to know the MySQL port (default is 3306)
   1. Within MySQL
      - Create a database called "amrs" **with default encoding of utf-8**

```
CREATE DATABASE amrs /*!40100 DEFAULT CHARACTER SET utf8 */;
```

- Create a user account and grant full rights to the amrs database to this user

2. Execute the OpenMRS Demo SQL Script to load the database into the "amrs" schema

4. Create your runtime properties file (this file tells OpenMRS where to find the database and how to authenticate to MySQL). You must make a system variable called AMRS_PROPERTIES_FILE that points to this properties file.

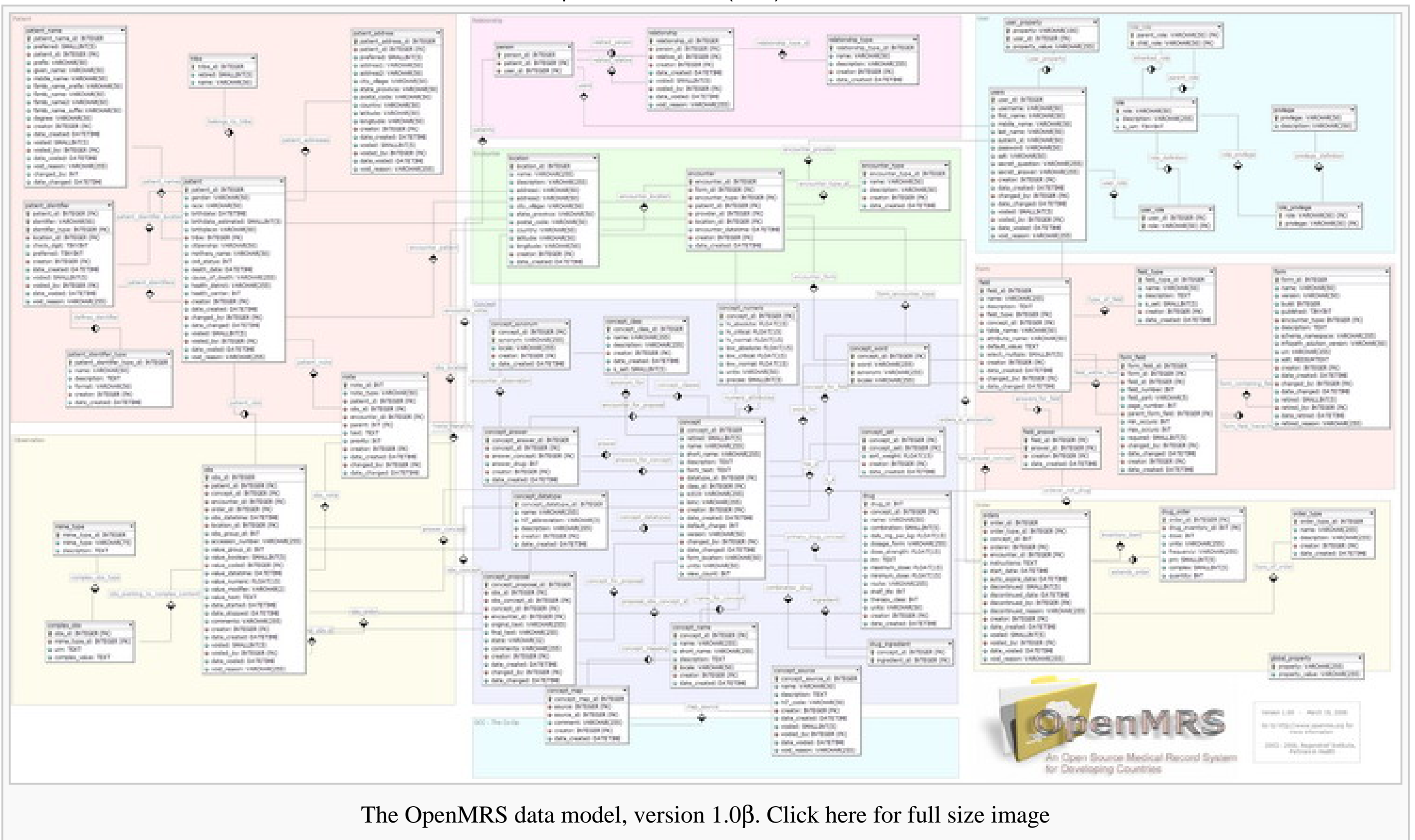


Create a system variable for JAVA_HOME and another that points to your runtime properties file. JAVA_HOME should point to the top of your JAVA installation — e.g., *C:\Program Files\Java\jds1.5.0_06*. The name of the system variable pointing to your runtime properties should be *AMRS*_RUNTIME_PROPERTIES_FILE — where *AMRS* is the name of your OpenMRS implementation

1. Install OpenMRS
    1. Log into tomcat manager at http://localhost:8080/manager/html
    2. Deploy war: amrs.war

At this stage, clients only need a browser (we're currently building for Firefox...if you use Internet Explorer . They need Microsoft InfoPath® (with Service Pack 2) to perform FormEntry functions.

Technical Overview



The OpenMRS data model, version 1.0β. Click here for full size image

Latest revision: *1.0β* (03/19/06)

- Change Log: History of all changes to the OpenMRS model
- Proposed Data Model Changes: Proposed changes to the OpenMRS data model
- Documentation: Full documentation of the OpenMRS schemata, including specifics of every field and table — also an open forum for discussion of issues surrounding their design
- Downloads: All files needed to build the database within your installation
- Preview version 1.10

What's the difference between `concept_set` and `concept_set_derived` tables?

Humans (you and me) should only edit set relationships in the `concept_set` table. These relationships are then programmatically *burst* into the `concept_set_derived` table for "real world" use.

- **Humans use `concept_set`**
- **Computers use `concept_set_derived`**

Understanding by example

Here's an example (using made-up concepts). Suppose we want to relate the following three concepts using concept sets:

1. *metoprolol*
2. *Beta Blockers*
3. *Anti-Hypertensive Medications*

We (humans) use the dictionary editing tools to define **two** relationships:

concept_settable

concept	set
metoprolol	*Beta Blockers*
Beta Blockers	*Anti-Hypertensive Medications*

The above relationships, state that *metoprolol* is a *Beta-Blocker* and that all *Beta-Blockers* are *Anti-Hypertensive Medications*. The idea that *metoprolol* is an *Anti-Hypertensive Medication* is **implied**.

We then run a script (e.g., *AdministratorService.burstTermSets()*) to programmatically burst all of the implicit relationships and put these into the `concept_set_derived` table. Each time the script is run, the `concept_set_derived` table is cleared, and all entries in the `concept_set` table are copied into `concept_set_derived`, along with any *implied* relationships.

concept_set_derivedtable

concept	set
metoprolol	*Beta Blockers*
Beta Blockers	*Anti-Hypertensive Medications*
metoprolol	*Anti-Hypertensive Medications*

The third item in the list above is **derived** from the manually defined relationships.

Why all this monkey business with `concept_set_derived`?

The `concept_set` table is displayed in the Data Model. The `concept_set_derived` is considered part of the *business stuff* that's needed to make OpenMRS work. The bottom line is that bursting out implicit relationships into a `concept_set_derived` table prevents the application from having to calculate all of the implicit relationships in real time. Likewise, we don't want to burden humans with having to explicitly define all of these implicit relationships (why create three relationships when two says it all — in the example above). OpenMRS Architecture The OpenMRS FormEntry module uses Microsoft® InfoPath™ to gather data for the repository.

Designing Forms

- The form hierarchy (or *schema*) created within the OpenMRS web application defines an XML Schema for data collection. Essentially, the form hierarchy defines all of the data points that will be (or could be) gathered on any particular form.
- When you download a form for the first time, the schema your have defined along with an XML template that follows the schema are injected into a *starter form* that contains the basic layout and plumbing for InfoPath. A downloaded XSN should be saved to disk and opened with InfoPath in design mode — e.g., right-click the XSN file and select *"Design"* from the context menu.
- During the design process, forms that are saved to disk are automatically altered internally by InfoPath. The URLs in the form are changed to point to the local file system.
- When you upload a form through the OpenMRS web application, the internal URLs are converted to the appropriate value based on the server's URL and the file's location on the server.

Submitting Forms

- When an InfoPath form is submitted to the server, an XML file containing the form data is actually posted to the server (via HTTP POST protocol, just as you were submitting a web-based form)
- The server places the XML file (unchanged) into the `formentry_queue` table
- A scheduled FormEntry queue processing task scans the `formentry_queue` table every 30 seconds and, when records are found, process queue entries. The XML is translated using the form's XSLT template to create an HL7 message to be placed in the

hl7_in_queue (HL7 inbound message queue)

- If errors occur, the record is transferred to the formentry_error table and processing stops
- If no errors occur, the record is transferred to the formentry_archive table and an entry is made into the hl7_in_queue (HL7 inbound message queue)

- An HL7 processor task scans the hl7_in_queue table every 30 seconds and processes any entries that are found.
  - The open source HL7API engine is used to parse the HL7 message and create the appropriate encounter and obs records.
  - If any errors occur, they are logged in the hl7_in_error table; otherwise, the entry is moved to the hl7_in_archive table when processing is completed successfully.

In summary, form data are initially placed in the FormEntry queue, translated into HL7 messages in the HL7 inbound queue, and then parsed into the individual encounter and observational data points. Subsequently, all data operations (other than auditing FormEntry submissions) are performed on data from the encounter and obs tables.

By FormEntry 1.2, we should be translating order data into appropriate HL7 messages to follow the same general flow and generate data within the order tables.

Glossary

API
: application programmering interface

EMR
: electronic medial record

Hibernate
: An object-relational mapping tool (see hibernate.org)

JDK
: Java Development Kit (allows you not only to run, but also to compile Java programs — used by developers to write Java programs)

JRE
: Java Runtime Environment (allows you to run Java programs)

MySQL
: an open-source database engine (see the MySQL website)

ORM
: object-relational mapping, maps the world of Java objects to the relational data model of a database (see [http://www.hibernate.org Hibernate[])

URL
: Uniform Resource Locator is essentially a reference to an address on the internet

Appendix A

Calculate a check digit

Why bother with check digits?

The purpose of check digits is simple. Any time identifiers (typically number +/- letters) are being manually entered via keyboard, there will be errors. Inadvertant keystrokes or fatigue can cause digits to be rearranged, dropped, or inserted. Have you ever misdialed a phone number? It happens.

Check digits help to reduce the likelihood of errors by introducing a final digit that is calculated from the prior digits. Using the proper algorithm, the final digit can always be calculated. Therefore, when a number is entered into the system (manually or otherwise), the computer can instantly verify that the final digit matches the digit predicted by the check digit algorithm. If the two do not match, the number is refused. The end result is fewer data entry errors.

What is the Luhn algorithm?

We use a variation of the Luhn algorithm. This algorithm, also known as the "modulus 10" or "mod 10" algorithm, is very common. For example, it's the algorithm used by credit card companies to generate the final digit of a credit card.

Given an identifier, let's say "139," you travel right to left. Every other digit is doubled and the other digits are taken unchanged. All resulting digits are summed and the check digit is the amount necessary to take this sum up to a number divisible by ten.

```
Got it?  Alright, lets try the example.

Work right-to-left, using "139" and doubling every other digit.

9 x 2 = 18

3 = 3

1 x 2 = 2

Now sum all of the *digits* (note '18' is two digits, '1' and '8').  So the answer is '1 + 8 + 3 + 2 = 14' and the check digit is
the amount needed to reach a number divisible by ten.  For a sum of '14', the check digit is '6' since '20' is the next number
divisible by ten.
```

Our variation on the Luhn algorithm

We have borrowed the variation on the Luhn algorithm used by Regenstrief Institute, Inc.. In this variation, we allow for letters as well as numbers in the identifier (i.e., alphanumeric identifiers). This allows for an identifier like "139MT" that the original Luhn algorithm cannot handle (it's limited to numeric digits only).

Allowing letters — even limited to capital letters — does not increase the accuracy of data entry. In fact, the potential for mistaking numbers and letters likely increases the chance for errors. In our case (Regenstrief with the AMRS), we were forced to come up with a simple method for generating identifiers in disparate, disconnected location without collision (giving out the same number twice). Adding a 2-3 letter suffix to the identifer was our solution.

To handle alphanumeric digits (numbers *and* letters), we actually use the ASCII value (the computer's internal code) for each character and subtract 48 to derive the "digit" used in the Luhn algorithm. We subtract 48 because the characters "0" through "9" are assigned values 48 to 57 in the ASCII table. Subtracting 48 lets the characters "0" to "9" assume the values 0 to 9 we'd expect. The letters "A" through "Z" are values 65

to 90 in the ASCII table (and become values 17 to 42 in our algorithm after subtracting 48). To keep life simple, we convert identifiers to uppercase and remove any spaces before applying the algorithm.

Here's how we handle non-numeric characters

For the second-to-last (2nd from the right) character and every other (even-positioned) character moving to the left, we just add 'ASCII value - 48' to the running total. Non-numeric characters will contribute values >10, but these digits are *not* added together; rather, the value 'ASCII value - 48' (even if over 10) is added to the running total. For example, '"M"' is ASCII 77. Since '77 - 48 = 29', we add 29 to the running total — *not* '2 + 9 = 11'.

For the rightmost character and every other (odd-positioned) character moving to the left, we use the formula '2 * n - 9 x INT(n/5)' (where INT () rounds off to the next lowest whole number) to calculate the contribution of every other character. If you use this formula on the numbers 0 to 9, you will see that it's the same as doubling the value and then adding the resulting digits together (e.g., using 8: '2 x 8 = 16' and '1 + 6 = 7'. Using the formula: '2 x 8 - 9 x INT(8/5) = 16 - 9 x 1 = 16 - 9 = 7') — identical to the Luhn algorithm. But using this formula allows us to handle non-numeric characters as well by simply plugging 'ASCII value - 48' into the formula. For example, '"Z"' is ASCII 90. '90 - 48 = 42' and '2 x 42 - 9 x INT(42/5) = 84 - 9 x 8 = 84 - 72 = 12'. So we add 12 (*not* '1 + 2 = 3') to the running total.

So, here's how we would use the Luhn algorithm for the identifier "139MT"

```
   T (ASCII 84) -> 84 - 48 = 36 -> 2 x 36 - 9 x INT(36/5) = 72 - 9 x 7 = 72 - 63 = 9

   M (ASCII 77) -> 77 - 48 = 29

   9 x 2 = 18 -> 1 + 8 = 9 or 9 => 2 x 9 - 9 x INT(9/5) = 18 - 9 x 1 = 18 - 9 = 9

   3 = 3

   1 x 2 = 2 or 1 => 2 x 1 - 9 x INT(1/5) = 2 - 9 x 0 = 2

   Summing the results we get '9 + 29 + 9 + 3 + 2 = 52'.  The next number divisible by ten is 60.  So, our check digit (the
difference) is 8.
```

The modified mod10 algorithm implemented in Java:

```
public int checkdigit(String idWithoutCheckdigit) {

  // allowable characters within identifier
  String validChars = "0123456789ABCDEFGHIJKLMNOPQRSTUVYWXZ_";

  // remove leading or trailing whitespace, convert to uppercase
  idWithoutCheckdigit = idWithoutCheckdigit.trim().toUppercase();

  // this will be a running total
  int sum = 0;

  // loop through digits from right to left
  for (int i = 0; i < idWithoutCheckdigit.length(); i++) {

    //set ch to "current" character to be processed
    char ch = idWithoutCheckdigit
              .charAt(idWithoutCheckdigit.length() - i - 1);

    // throw exception for invalid characters
    if (validChars.indexOf(ch) == -1)
      throw new InvalidIdentifierException(
                  "\"" + ch + "\" is an invalid character");

    // our "digit" is calculated using ASCII value - 48
    int digit = (int)ch - 48;

    // weight will be the current digit's contribution to
    // the running total
    int weight;
```

```
    if (i % 2 == 0) {

      // for alternating digits starting with the rightmost, we
      // use our formula this is the same as multiplying x 2 and
      // adding digits together for values 0 to 9.  Using the
      // following formula allows us to gracefully calculate a
      // weight for non-numeric "digits" as well (from their
      // ASCII value - 48).
      weight = (2 * digit) - (int) (digit / 5) * 9;

    } else {

      // even-positioned digits just contribute their ascii
      // value minus 48
      weight = digit;

    }

    // keep a running total of weights
    sum += weight;

  }

  // avoid sum less than 10 (if characters below "0" allowed,
  // this could happen)
  sum = Math.abs(sum) + 10;

  // check digit is amount needed to reach next number
  // divisible by ten
  return (10 - (sum % 10)) % 10;

}
```

The modified mod10 algorithm implemented in VBA:

```
Function checkdigit(idWithoutCheckDigit)

  ucIdWithoutCheckdigit = UCase(idWithoutCheckDigit)
  total = 0
  For i = Len(ucIdWithoutCheckdigit) To 1 Step -2
      digit = Asc(Mid(ucIdWithoutCheckdigit, i, 1)) - 48
      total = total + (2 * digit) - Int(digit / 5) * 9
      If (i > 1) Then
          digit = Asc(Mid(ucIdWithoutCheckdigit, i - 1, 1)) - 48
          total = total + digit
      End If
  Next i
  total = Abs(total) + 10
  checkdigit = (10 - (total Mod 10)) Mod 10

End Function
```

Note: this VBA algorithm should probably check each character and return an error if any invalid characters are found (as the Java example above does by throwing an exception)

Appendix B

OpenMRS allows you to indicate a required format for patient identifiers, using regular expressions. To specify a format, go to the Administration page, then click on Manage Identifier Types. From here, if you have created Identifier Types (at least one is required by OpenMRS), you can click on the name of the type to edit it. From the edit screen, you will find a field labeled "Regex Expression". Here you can enter a regular expression to specify what format an identifier of that type needs to be.

If you do not enter a regular expression, an identifier of any format can be entered for that identifier type.

If you do enter a regular expression, the system will verify that any identifier of this type conforms to this regular expression whenever an identifier is entered or changed.

The syntax of regular expressions is outside the scope of this page, but there are several sites/pages online that discuss them. Below is a link to

a tutorial on regular expressions.

See also

Regular expression info site

Appendix C

This documentation is designed to give developers a basic description of how clinical observational data is written to the AMRS data repository.

To start, some basic definitions:

An **observation** is any clinical measurement acquired and observable in a clinical setting. If you use your imagination, just about anything can be an observation. For example, a hemoglobin test, a patient's weight, an answer to a question, or a physical exam finding. In short, any piece of data collected in a clinical setting (a patient's demographic information being a notable exception) is considered to be an observation. Note that if a clinician decides to order a hemoglobin, the order itself isn't observational: this is a clinician action, not something measured or acquired directly from the patient.

A **concept** is a means of describing observational information within our system. Inherent within the data model is a collection of definitions for any concept that's collected within the repository: the concept dictionary. Each concept is described with a term, which is represented within the *term* table ([[Dictionary 101|and is described in depth elsewhere).

Storing information within the data repository is pretty straightforward, if you understand the underlying database structure. Let's look at this portion of the data model.

Each observation noted during a clinical evaluation is stored as a row within this table. The **obs_id** is the unique identifier of the table, and should be a simple autonumbered field. Many of the subsequent attributes seen to the right are "meta data" related to the particular visit. **patient_id** is a foreign key to patient.patient_id, and should include the internal patient identifier number for the patient (not the medical record number). **term_id** is another foreign key which points to term. term_id and refers to the concept which describes what is being collected by the system. **location_id** describes where the observation was collected, it will typically be a clinical setting such as a clinic or a laboratory. It has a foreign key which points to location.location_id. More on location_id in a moment. Finally, **encounter_id** refers to the particular patient visit. As you might assume, many observations are collected during the course of a given clinical visit. Each visit's "meta data" is contained within this encounter table. You might wonder what the point of location_id is, after seeing what is stored within an encounter. In short, observations on a patient can be made outside of an encounter, or outside of the purview of our system. For example, a patient could bring copies of their medical record, or observations could be made during a home visit, etc. Therefore, encounter_id is not a required field.

When building an application, you'll have likely acquired all of this "meta data" before results are collected by your application. How you store the value of the observation, depends of the data type of the concept as defined by it's term definition. You'll notice that there are numerous

different fields to store values. You'll only use one of these for each row. Let's walk through each major datatype to demonstrate.

Boolean: On one of the patient encounter forms used in the Eldoret clinics, the following question is asked: "Have you disclosed your HIV status to anyone?" This question has a simple yes or no answer, and is modeled in our dictionary within concept 1048. If you click through the provided link, you'll see that the datatype for this concept is boolean. Because of this, you will store a bit answer within *value_boolean*.

Coded: Another example, concept 1061. This is another question asked on an encounter form: "How do you think you were exposed to HIV?" There are six possible answers to this question, all of which are ALSO concepts within the system. Click through the link provided for more details. To store the answer for this question, you need to store the *term_id* for the corresponding concept. Say, in this case, the answer was "through a blood transfusion", which corresponds to concept 1063. You would store "1063" in *value_coded*. As you might have noticed, *value_coded* in fact has a foreign key to concept.concept_id to ensure this linkage occurs and maintain integrity.

Datetime: Concept 1113. In this particular example, the AMRS encounter form is capturing some historical information related to a previous date in which tuberculosis therapy was started. Note the "date" datatype. The answer to this question is a SQL compliant date, time or datetime. 1/1/2005, 1-1-2005, etc. depending on the database specifics within *value_datetime*.

Numeric: Let's look at a lab test result, a hematocrit: concept 1015. This test has a numeric answer, like 33 or 45. Store this answer within *value_numeric*. Most values will be pretty straightforward, like this. What if however, the result is a range, like 2-4, or a value that's "<1" or ">=400"? These more complex numeric answers can be described with the *numeric_modifier* field. *numeric_modifier* is an ordinal field, which has a list of all of the needed modifiers, such as ">", "<", "range", etc. To store a simple modifier, write the numeric value within *value_numeric*, and add the corresponding modifier into *numeric_modifier*. In the case of ranges, store the mean value of the range within *value_numeric*, store "range" in *numeric_modifier*, and the fully described value (for later display purposes) in *value_text*. This allows the system to use an approximation of the value in more computable ways (such as decision support, etc.), and still store the actual answer for display purposes. Contact us for more details on this.

Here are a few actual examples to provide a finer point.

Patient: Jenny D. Patient
MRN: 99TU-2, which corresponds to *patient_id*: 123
Clinic Vist Date: 1/14/05 @ 2:30pm
Location: Mosoriot Medical Center
Hemoglobin: 11.5
Weight: 45 kg
Date of TB treatment: 1/1/2002
Gastrointestinal Exam: Hepatomegaly...

As you can see, this patient has a number of observations ready for storage. The name, and medical record number aren't important for the **obs** table, but you can see a *patient_id*, and a location. You also see a datetime for the visit. The third and fourth bits of information will give you enough information to build an encounter within the **encounter** table. Mosoriot has a corresponding location_id of "2&". Following the demographics are four separate observations to be recorded: hemoglobin, a numeric value; weight, a numeric vital sign; date of TB treatment, a datetime; and gastrointestinal exam finding, a coded answer. If you look at this concept, hepatomegaly is one of the possible answers, and it's the corresponding concept 5008. Knowing all of this, filling in the table is pretty straightforward. Based on the data to the left, you would store 4 rows of data within the obs table with the following information...

obs_id	pat_id	trm_id	lctn_id	encntr_id	sub_id	val_bool	val_cod	val_dt	val_num	num_mod	val_txt	d_e_t	enterer	comment
1	123	21	2	345					11.5			1/15/2005	2	
2	123	5089	2	345					45			1/15/2005	2	
3	123	1113	2	345				1/1/2002				1/15/2005	2	
4	123	1125	2	345			5008					1/15/2005	2	

That should be enough info to get folks started. Feel free to send us mail if you'd like this to be explained in more detail!

-Paul

Appendix D

One of the most critical aspects of an electronic medical record system is a well-designed and maintained concept dictionary. What follows is a primer which will hopefully allow you to understand the basic concepts behind what the dictionary is, how it is to be used, and beginning steps in enriching the current dictionary.

Beginning Definitions

Concept Dictionary — The truly fundamental building block of the entire OpenMRS. Much like a dictionary defines the function, meaning and relationships of words, the "concept dictionary" defines the names, codes, and other attributes for all medical tests, drugs, and coded results contained in the OpenMRS. It is a complete reference for the "language" in which we represent medical concepts within the computer.

Observation — Anything actively measured in a given person. This has a fairly broad implication. If a patient is weighed, their weight is an observation. If a blood test is drawn, that lab result is an observation. If we ask someone how long they've been smoking, their answer is also an observation. Just about anything "measurable" in a patient is an observation. Demographic information, as a result, is an exception.

Demographic — Any descriptive characteristic of a person. Their name, address, date of birth, age, tribe name.. these are all demographic characteristics.

Primary Term — The most "elemental" preferred label for a medical concept within the dictionary. Each concept has one, and only one primary term. As an example, "HEMATOCRIT" is a primary term which describes a blood test. Many people refer to this test with the short name of "HCT." HCT in this case is not a primary term. Any other name outside of HEMATOCRIT makes it something other than the primary term.

Primary terms are used to describe the "questions" like a hematocrit value, as well as the "answers" (Paul Biondich's blood type is "O POSITIVE"). "O POSITIVE" is just as much of a unique medical concept as "BLOOD TYPE" or "HEMATOCRIT". The bottom line is, if you have a medical concept of any sort, and it's needed within our system, it needs to be defined within the dictionary. See "Concept Creation Hints" below for info on how to determine the best way to name/decide on what is the best label for a primary concept.

Synonym — Any label or name that refers to a primary concept. This would be the "HCT", "HEMATOCT", or "HAEMATOCRIT" for a "HEMATOCRIT." Other good examples of this would be trade names for a chemical compound, names that other practitioners label the same test, etc. For specific rules and naming conventions, refer to "Concept Creation Hints" below.

Concept Set — A primary concept created to bundle or refer to multiple other primary concepts. A good example of this would be a "COMPLETE BLOOD COUNT" which is composed of a "HEMATOCRIT" along with other tests, such as "PLATELETS", etc. Concept sets can also have synonyms! (i.e., "CBC" for a "COMPLETE BLOOD COUNT")

Concept Class — A required attribute of a concept. Simply, this is the classification of the medical concept. Every primary concept is required to have a class. Here is the current list of classes to help clarify:

- Test — lab tests (e.g., CD4) or physical exam maneuver (e.g., Babinski)
- Procedure — spinal tap, lumbar puncture, etc.
- Drug — medications
- Diagnosis — a defined medical conclusion (usually in ICD), e.g., diabetes, AIDS
- Finding — physical or exam findings (short of breath, systolic murmur, LLL infiltrate)
- Anatomy — anatomic concept (e.g., right arm, frontal lobe)
- Question — questionnaire item (e.g., "Are you depressed?", "How many kids?")
- LabSet — collection of labs (e.g., CHEM12, VITALS)
- MedSet — collection of medications (e.g., ANTIRETROVIRALS, PENICILLINS)
- ConvSet — collection for convenience (e.g., STUDY XYZ CONCEPTS, DEPRESSION CONCEPTS)
- Misc — unclassifiable concepts (e.g., POS, NEG, LEFT, RIGHT)

Data Type — A required attribute of a concept. Basically, if data is stored within this concept, this describes what it will look like. For example, a "HEMATOCRIT" will store numeric values (ie, 40). A "BLOOD TYPE" will store coded values (i.e., "O POSITIVE"). Here are the current datatypes:

- Coded — terms with coded answers (CXR, PPD) (i.e., test answered with other concept(s))
- Numeric — tests with numeric results (CD4, CREATININE, SERUM GLUCOSE)
- N/A — not a observation/question/test (drugs, sets, answers, misc terms)

Why I should care about the concept dictionary

At its most basic, the OpenMRS is a large filing cabinet. Within that cabinet, each patient has a file. Within that file are a series of data points and information relating to that patient. Those data points are from any place within the AMRS that the patient receives care. These are the patient observations. Within our system, each patient will have dozens, if not hundreds of these observations recorded as they utilize the health care system. Each one of these observations consists of a question (What’s the patient’s hematocrit?) and an answer (38). The computer needs a way of referencing each of these aspects of an observation. This is where a concept dictionary comes in handy. The computer uses it to label our questions (and answers) correctly. Consequently, there’s a need for all medical concepts we’re interested in storing to be referenced uniquely within the dictionary. In other words, each concept needs a primary term.

Imagine the difficulties you’d have however, if there are redundant primary terms which refer to the same concept? What if the computer had three primary terms for a hematocrit? When the system later attempts to tell practitioners what these results were, there’s a mess! What hematocrit does it report? In reality, there’s often multiple ways of referring to the same question. But these aren’t all primary terms, they are synonyms.

A point worth repeating...the important distinction between CONCEPTS and SYNONYMS:

```
A fundamental hallmark of the dictionary is that there is one concept entry,
and one concept entry ONLY to describe a particular concept.  For example,
LAMIVUDINE is the proper dictionary term for a HIV drug.  3TC is another way
of describing lamivudine.  Epivir is a brand name of lamivudine.  Based on
how we’ve built the OpenMRS, 3TC and Epivir are synonyms of lamivudine.  They
are not dictionary terms.  All three of these words describe the same concept.
```

Got a new concept that you need the system to track for clinical or research purposes? Then let’s make a new concept...

Creating new concepts for the concept dictionary

So you want to make new concepts? Before you undergo this process, contemplate the following three steps:

1. Make sure the concept doesn't already exist in the dictionary. When searching the dictionary, use partial names (e.g., KALE or KALET instead of KALETRA). Looking for partial names will help catch misspelled entries. Think about what you might most generically refer to this concept as.. then also think about what else you might call it. Then search for all of those things within the dictionary. You might be surprised to see it already in the dictionary.
2. Make sure that you can describe/understand the concept that you're getting ready to enter! Say for example, that you're asked to create a new term for the retroviral drug eliminatehivudine. Knowing that it's a retroviral drug is insufficient, as you're going to need to detail eliminatehivudine's differences from all other antiretrovirals within the term's description. Check with the person that requested the new concept if you're having problems finding the information on your own. Every new term's description, from here on out, should have a reasonably detailed description. We'll send Burke or Sarah Ellen out to punish you if you fail.
3. Does the concept have a standardized representation in either LOINC or ICD10? Look it up and add it to the concept's definition or ask for help if you have no idea what this is.

Do you have all of the information ready? Then it's time to walk through a primary concept definition, and the basic attributes this includes.

Concept Name: Pretty self explanatory. The human readable name of the concept. Here are a few naming conventions to think about before you decide on a name:

1. The concept's name should begin by labeling what identifies it: It's a HEPATITIS B IMMUNIZATION, not a IMMUNIZATION, HEPATITIS B.
2. Should be capitalized throughout the name. (ie, COMPLETE BLOOD COUNT vs. Complete BlOoD count)
3. They should be the complete spelled form of the concept (it's not a HCT, it's a HEMATOCRIT) to ensure proper use. Abbreviations are usually not primary terms, they are synonyms.
4. They should whenever necessary refer to the generic form of a product (AMOXICILLIN, not AMOXIL).
5. When referring to organisms or virii, the full taxonomic name is used: HUMAN IMMUNODEFICIENCY VIRUS, not HIV.
6. They are specific! For example, tests that measure antibodies vs. antigens need two separate terms.
7. They are granular! RIGHT UPPER QUADRANT ABDOMINAL PAIN stacks too many observations together. This is tricky in practice to decide upon and open for interpretation/dispute. Best bet: when you're not sure of the appropriate level of granularity, ask one of the geeks.

Description: A comprehensive description of what this term represents. The term's definition. By the time a total stranger finishes reading this, they without question know what this term refers to. This is absolutely required for all terms. No exceptions.

ClassID: see the above definition. Classifies what sort of concept this term represents. Here's a list of the active classes:

1. Test — lab tests (e.g., CD4) or physical exam maneuver (e.g., Babinski)
2. Procedure — spinal tap, lumbar puncture, etc.
3. Drug — medications
4. Diagnosis — a defined medical conclusion (usually in ICD), e.g., diabetes, AIDS
5. Finding — physical or exam findings (short of breath, systolic murmur, infiltrate)
6. Anatomy — anatomic concept (e.g., arm, frontal lobe)
7. Question — questionnaire item (e.g., "Are you depressed?", "How many kids?")
8. LabSet — collection of labs (e.g., CHEM12, VITALS)
9. MedSet — collection of medications (e.g., ANTIRETROVIRALS, PENICILLINS)
10. ConvSet — collection for convenience (e.g., STUDY XYZ TERMS, DEPRESSION TERMS)
11. Misc — unclassifiable concepts (e.g., POS, NEG, LEFT, RIGHT)

DataType: see the above definition. Classifies what kind of answer an observation receives, if it's a "question." Here are the active data types:

1. Coded — terms with coded answers (CXR, PPD) -- i.e., test answered with other term(s)
2. Complex — avoid this like the plague. we may eliminate it as an option
3. NumVal — tests with numeric results (CD4, CREATININE, SERUM GLUCOSE)
4. N/A — not a observation/question/test (drugs, sets, answers, misc terms)

Further reading

- Modeling Concepts

Here are some notes/conversations related to concept modeling that may be helpful resources when confronting modeling issues.

Modeling diagnoses

Hamish wrote:

```
If I create a concept "malaria" say a boolean and I want to also have
"Malaria diagnosis date" how do I link these two items so that it is
clear that they refer to the same item?

Also as Darius and I discussed today what if Carole our epidemiologist
friend says she wants all the diagnoses for her data collection form
to have true, false, unknown and "no data available". Should we just
create hundreds of coded concepts for this or formalize is as a
standard construct? If we formalize it does that help us to handle such
data efficiently in the same analysis and data extraction tools that
normally take a boolean?
```

I would use DIAGNOSIS or PATIENT REPORTED DIAGNOSIS or similar generically and then record the diagnoses as answers (e.g. MALARIA) for most cases. That gives you the choice of recording the date of diagnosis into the obs date (probably simpler) or creating a DATE OF DIAGNOSIS observation to link dates to diagnoses through an obs_group_id.

If someone wants multiple choice answers for 1..n diagnois, then you are collecting two observations in each case: a multiple choice answer and the diagnosis for which the answer applies. There are a several ways to skin this.

1. Flat by diagnosis (add coded answers for each diagnosis)

```
MALARIA with coded answers TRUE, FALSE, UNKNOWN, NO DATA AVAILABLE
```

PRO
: simple, MALARIA/etc could still be used as answers

CON
: not scalable, ties you to one model of answers to diagnoses and these must be replicated/managed for *every* diagnosis in the system.

2. Fully abstract (one concept for diagnsosis and a 2nd for answer to multiple choice question)

```
INQUIRED DIAGNOSIS as coded answered by diagnosis
DIAGNOSIS STATUS as coded with coded answers TRUE, FALSE, UNKNOWN, NO DATA AVAILABLE
Tied together with obs_group_id
```

PRO
: easily scalable

CON
: needs tools/knowledge to convert back to one complex data point, diagnoses stored in a questionnaire-specific manner

3. Abstract, but use existing DIAGNOSIS concept

```
INQUIRED DIAGNOSIS as coded answered by diagnosis
DIAGNOSIS STATUS as coded with coded answers FALSE, UNKNOWN, NO DATA AVAILABLE
Linked with obs_group_id

Store TRUE answers as DIAGNOSIS with coded answer MALARIA, ASTHMA, etc.
```

PRO
: re-use of DIAGNOSIS, so you can still search DIAGNOSIS concept for all known diagnoses

CON
: application must know to treat TRUE answers differently

4. Flat by answer (one concept per answer for questionnaire)

```
DIAGNOSIS
DIAGNOSIS DENIED
DIAGNOSIS STATUS UNKNOWN
DIAGNOSIS DATA NOT AVAILABLE
each is coded and is answered with the diagnosis

CAROLE1 QUESTIONNAIRE DIAGNOSES OPTIONS as a concept_set of the above concepts
```

PRO
: scalable, could re-use DIAGNOSIS concept or have a separate concept for diagnosis = true on questionnaire

CON
: requires search for multiple concepts (could be facilitated by concept_set)

Retrieved from "http://openmrs.org/wiki/OpenMRS_Manual"

Category: Data Model

- This page was last modified 12:45, 18 February 2007.
- This page has been accessed 74 times.
- About OpenMRS™