

Cybertete — An E4 Rich Client On Smack

Contents

Introduction	3
Start with a blank application.....	6
Chat Service.....	16
Smack XMPP Client	26
Tycho and Equinox P2	32
Gaining Status.....	39
Options	48
Security	55
Testing	60
Status Bar Plugin.....	66
Getting Ready for Release	69
To be continued.....	71

Introduction

Eclipse Rich Client Platform (RCP) has been around while. I know because I have a book on it first released in 2005 written by Jeff McAffer and Jean-Michel Lemieux. I picked up the book because want to create a development tool to go with a new programming language I had written called Expression Pattern Language (eXPL). The book was published when Eclipse was at version at 3.1. That was the last unnamed version. The next one that came out was named Callisto and the one I am using at the moment is Mars version 4.5. So why did I bother with such a hopelessly out of date book? The answer is in the preface, which opens as follows:

In many ways, the book is one of the design documents for the Eclipse Rich Client Platform (CP). It was written during the Eclipse 3.1 development cycle by members of the development team. Its chapters were sometimes written before the related function was even implemented.

This book therefore gives a unique perspective on RCP development. It takes the reader through the stages of developing from scratch an example RCP application called Hyperbola, an instant messaging chat client. I decided that I would update Hyperbola and therefore learn what changes have been made to RCP over time. This is important not only to appreciate what improvements have been made, but as I perform searches on the internet, I know what to filter out because it's obsolete.

I diverged from the book right from the start by taking the offer to create a new application from a template. I saw immediately that commands had superseded actions. I came across other significant cases of things no longer used such as delta packs for exporting to other platforms. The Smack Third Party Library had also gone from version 1.5, as used by the book, to version 4.1.4. To cut a long story short, I ended with a contemporary version of the original Hyperbola application, binding to the latest version of Smack. One thing I did stick with was Eclipse 3.

Why E4?

This is what the E4 Eclipse project says about E4 in a nutshell:

Eclipse APIs are refactored into services that make up a uniform application model, which supports dependency injection to run in multiple different contexts such as desktop or web; the Workbench is uniformly modeled to provide introspection, flexible shaping, CSS styling and declarative UI markup; SWT

target platforms are added to run in the Browser; and many more initiatives in areas such as flexible resources, command recording, scripting, and plugins in other languages...+

In a few words, the E4 project is responding to changes in technology which harness the improvements made to hardware and approaches to software development. The main drawback of encompassing anything new is it takes time to learn and the best way is to plunge in and make lots of mistakes. Updating Hyperbola to E4 seems a good way to get started.

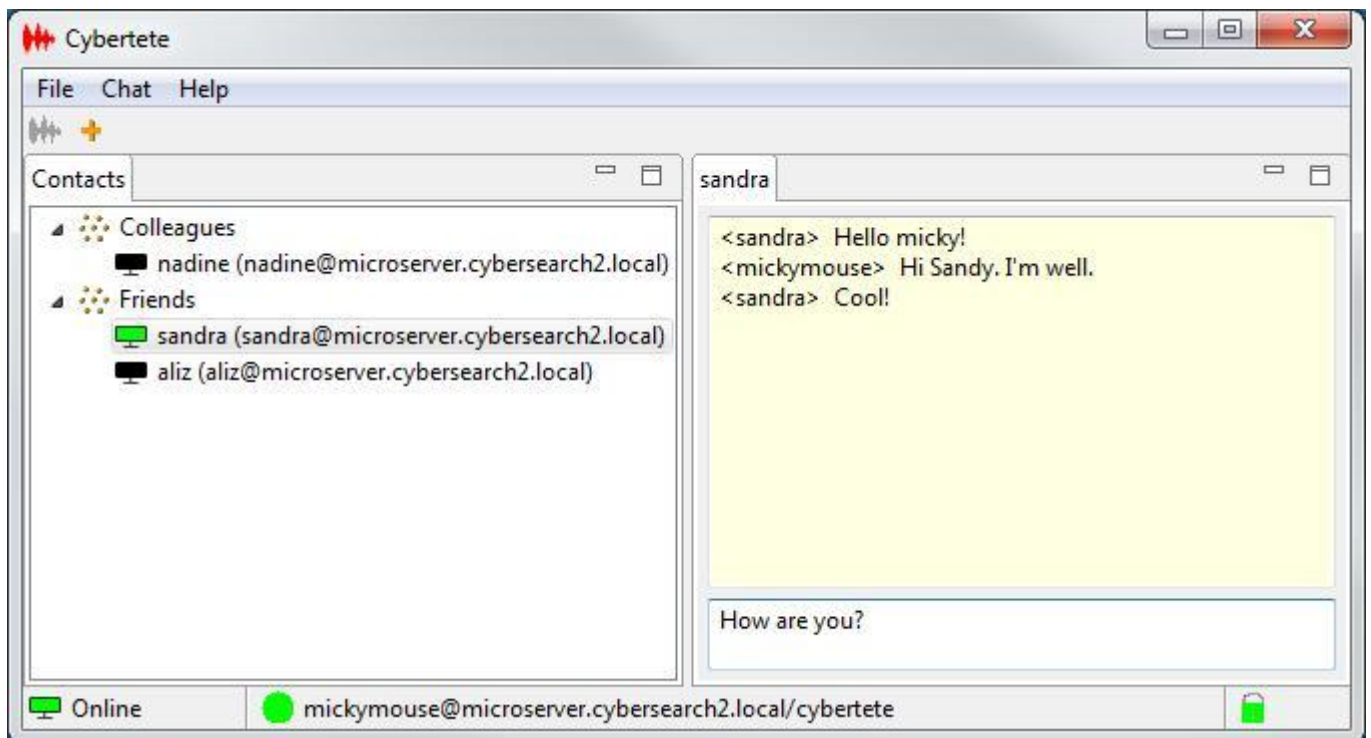
Hyperbola Goes to Mars

The first step towards E4 is to migrate to the latest and greatest version of E3 on a current version of Eclipse, which for me happens to be Mars 4.5. I decide to start from the ~~New~~ Plug-in Project+ wizard, which offers as the highest E3 version ~~3.5~~ 4.5 and above+. Using the wizard means having to navigate options, including whether to use a template, for which I opt in. This leads to the early discovery that commands have replaced actions as favoured approach to responding to events originating from the user interface. There are other changes to deal with such as the latest version of the Smack library having moved on from v1.5 to v4.1.4. I had to deal with differences between the two versions which was a challenge. Basically, Smack increased the use of listeners for application interactions with the library.

I get my start-with-a wizard version of Hyperbola running on Mars with a contemporary version of Smack and am ready to begin my E4 journey. That takes us to the next chapter...Start with a blank application

Cybertete — An E4 Rich Client On Smack

Screenshot of finished Hyperbola migration to Eclipse E4. The application is named **Cybertete**. The split screen with contacts on the left and chat sessions with a simple text format on the left is the only visual element of Hyperbola that survives.

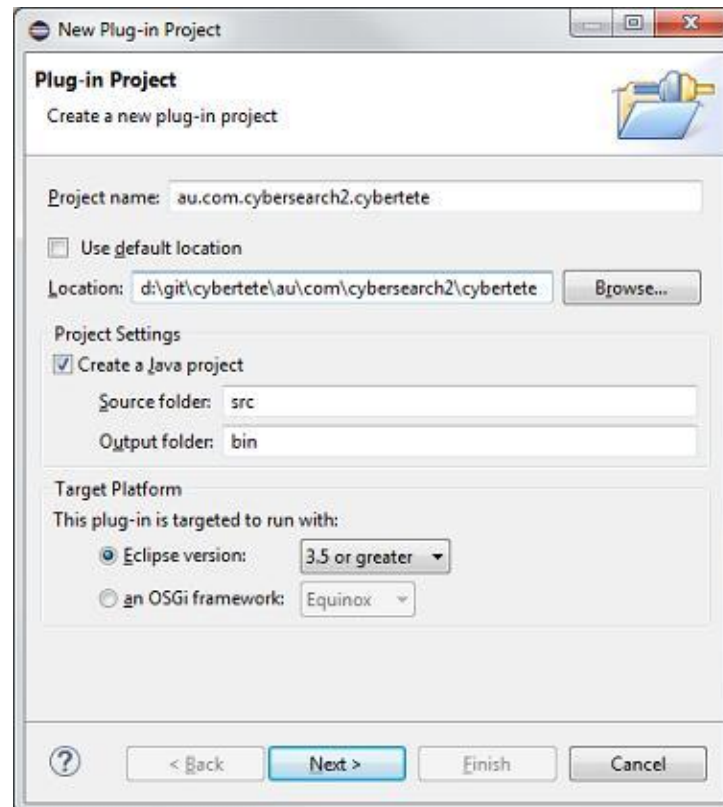


Start with blank application

To launch CyberTete, I select the Plug-in Development+Eclipse Perspective and go File -> New -> Plug-in Project

I name the project to match plug-in ID I will use:

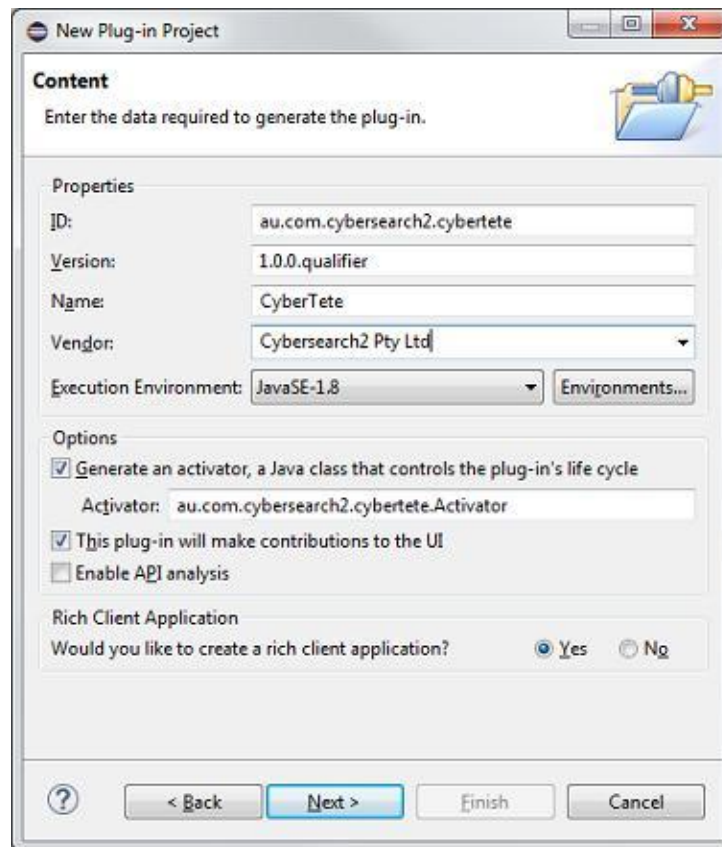
au.com.cybersearch2.cybertete+. I place it in a location outside the workspace where Git repositories reside.



Note that I encountered the default selection of Equinox OSGi framework for this plug-in is targeted to run with:+. Not sure if this is appropriate for a Rich Client plug-in, so I change it to Eclipse version 3.5 or greater+. For more details on OSGi, see <http://www.vogella.com/tutorials/OSGi/article.html>.

Cybertete — An E4 Rich Client On Smack

I click Next >

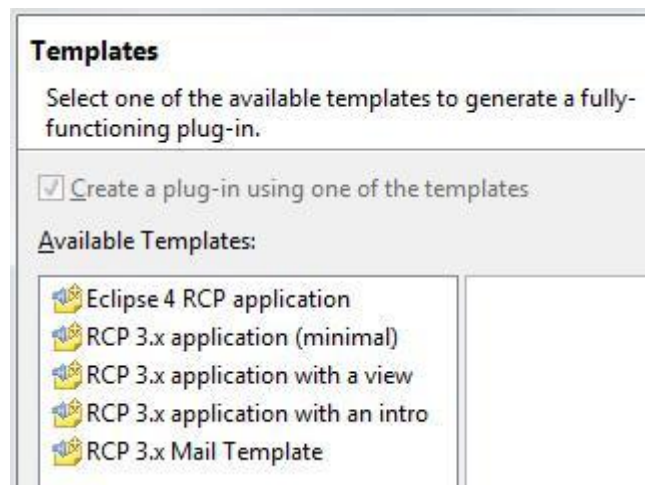


I select JavaSE-1.8 for Execution Environment as this is a development project, but which is best choice for end consumers would require research. How many are ready for JavaSE-1.8?

I tick ☒ Generate an activator+as I found a need for one when upgrading Hyperbola previously. Notice ☒ This plug-in will make contributions to the UI and ☒ Rich Client Application+are both ticked.

I click Next >

I select %Eclipse 4 RCP application+.



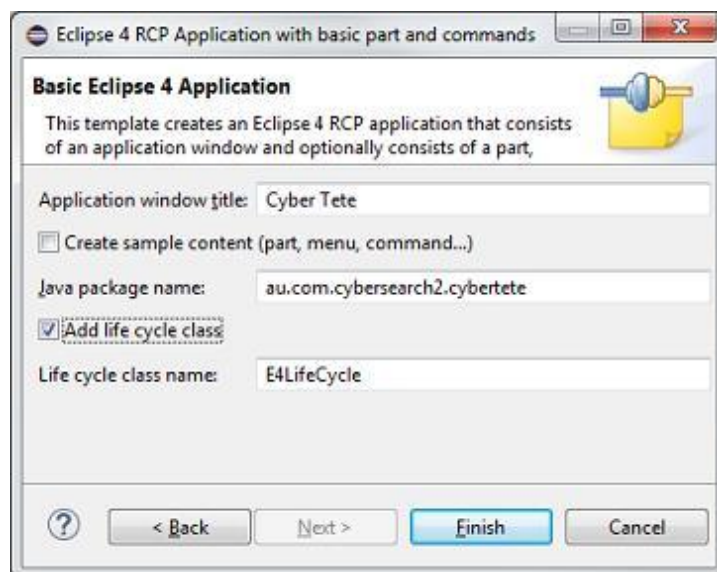
I see description:

This wizard creates a standalone Eclipse 4 RCP application with a sample part and the basic commands to quit the application.

Notice using a Template is not optional.

I click Next >

I do not select %Create sample context+as I want to start with just a main window. I tick %Add life cycle class+in anticipation of it being required.



I click Finish and the new project is created. Two editors are instantly opened, an Application Model editor and a Manifest editor. I notice the project has a Hyperbola4.product file and I open that with the Product Editor.

I click on the `Launch an Eclipse Application` on the Product Editor overview page and get an empty window with title `Empty E4 Application`. Progress!

Now is a good time to sidetrack and create a target definition for the project so I have one that is specific to CyberTete. I place the definition file in provided folder `External Plug-in Libraries`.

I use a software site location `Eclipse Mars repository` - <http://download.eclipse.org/releases/mars/>. I select Eclipse RCP 4.5.1 as the initial content. Note that `Group by Category` must be unticked to see it. A test launch confirms the target is configured correctly to support the current empty application. I also update the Run Configuration to clear the workspace before each launch to ensure progressive changes are applied during development.

Create Application Model.

Work now commences on using the Application Model editor to add the User Interface menus and commands and Contacts View defined in the E3 Hyperbola plugin.xml. The E4 model has one Trimmed Window with ID `org.eclipse.e4.window.main`. So the first thing to do is change the window label from `Empty E4 Application` to `CyberTete`.

The same window has an Icon URI, which prompts the question how this relates to the branding images in the *.product file. Apparently, they are not picked up by e4. From Stack Overflow:

There are two ways to get an icon into the corner of your window.

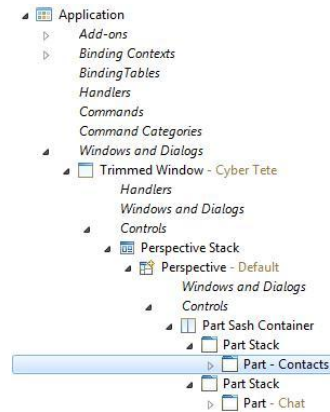
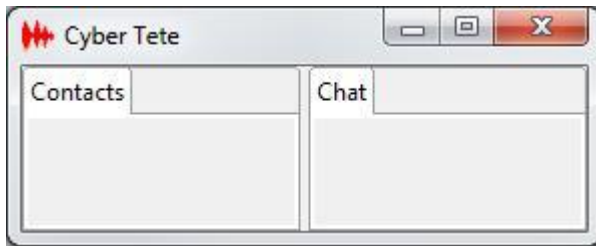
1. Set the "Icon URI" attribute of your Window in the *.e4xmi
2. Set the default icons using the JFace Window Singleton API `Window.setDefaultImages(new Image[] {...});`

I decide to go with solution 1. for now and assign a gif to the Icon URI of the main window. The application now has the right logo and title when launched, but the size needs to be increased. This is done by updating the window bounds.

The Application Model will ultimately contain more than one perspective, so I add a Perspective Stack with a Default child. As views and editors are both

Cybertete — An E4 Rich Client On Smack

Parts in e4, I need to add a Contacts part and a Chat part to the Default Perspective. Each Part needs a Part Stack parent to get a title tab and a Part Sash Container to display the Contacts View Chat Editor side by side.



play the Contacts View Chat Editor side by side.

Launch the application now and the layout is as expected:

Live Editor

The Live Editor allows you to access the application model of a running application, modify it and highlight selected components. In the running application

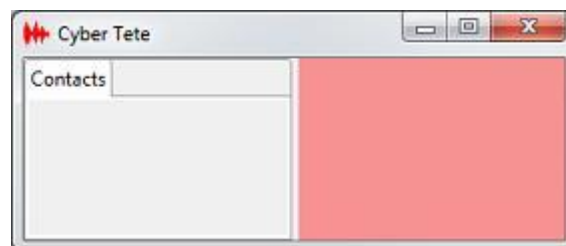
- ▶ <http://download.eclipse.org/e4/snapshots/org.eclipse.e4.tools/> 69 plug-ins available
 - ▶ E4 Context Spy Feature (Incubation) 0.17.0.v20150714-0417
 - ▶ Eclipse 4 - Model Spy (Incubation) 0.1.0.v20150714-0417
- ▶ <http://download.eclipse.org/releases/mars/> 195 plug-ins available
 - ▶ Eclipse Platform 4.5.1.M20150904-0015
 - ▶ Eclipse RCP 4.5.1.v20150904-0015

you can start the live editor via ALT+SHIFT+F9. This editor works exactly like the editor in your IDE, however, it directly accesses the application model of CyberTete. To set up the Live Editor requires additions to the target definition and runtime configuration. What works for me is this Target Location setting:

- ▶ ☒ [org.eclipse.e4.tools.context.spy](http://download.eclipse.org/e4/snapshots/org.eclipse.e4.tools.context.spy/) (0.17.0.v20150713-1844)
- ▶ ☒ [org.eclipse.e4.tools.emf.liveeditor](http://download.eclipse.org/e4/snapshots/org.eclipse.e4.tools.emf.liveeditor/) (0.12.0.v20150714-0417)

Note the snapshots URL is <http://download.eclipse.org/e4/snapshots/org.eclipse.e4.tools/>. Also add these plug-ins and their dependencies to the runtime configuration:

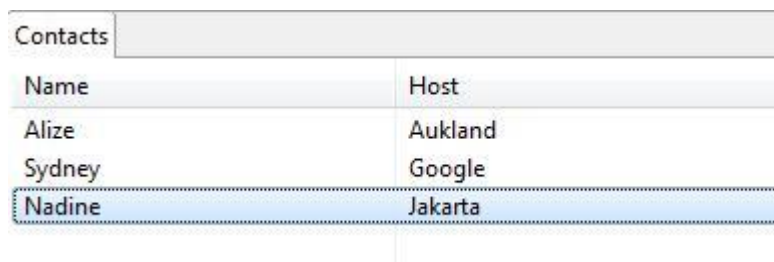
The Chat part can be highlighted by right-clicking the Live Editor element and



selecting %Show Control+. The following screenshot shows the Chat part highlighted and without an enclosing Part Stack:

Contacts View Test

Having got this far, it would be good to have a test Contacts implementation to render in the left hand part. I see there is an e4 Contacts Demo which will guide me on a contemporary approach to displaying a contacts list (see https://wiki.eclipse.org/E4/UI/Running_the_contacts_demo). This demo uses a TableViewer instead of a TreeViewer, which is needed to show both Groups as well as individuals, but changing from one to the other is hopefully easy. The demo code uses **ESelectionService**, **ObservableListContentProvider** and **ObservableMapLabelProvider** to manage the Workbench- Contacts interac-



Name	Host
Alize	Auckland
Sydney	Google
Nadine	Jakarta

tions.

The test Contacts implementation appears first time:

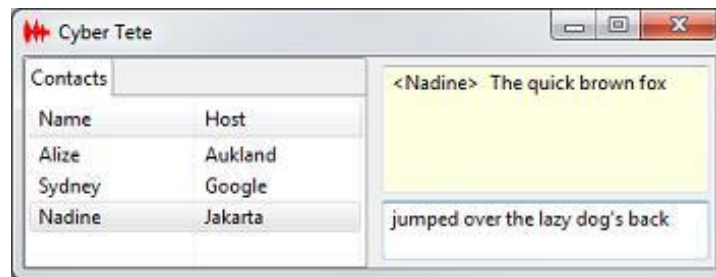
The demo relies on Java Bean utilities to access and track changes in the Contact details. If this approach is used in the real implementation, then some thought will be required on how to bind Smack objects to Java beans.

The next thing to get going is a popup context menu to start a Chat by selecting a client and then right mouse click. The demo already has such a menu to initiate an email, so it should be mostly copy. paste. To the Chat part I add a Popup Menu containing a Direct Menu item to which is assigned a handler to popup a message. This works, but the Popup Menu item will need to be Handled Menu item in the final design.

Chat Session Test

Now is a good time to implement the Chat Session view, which combines 2 text controls, one to input text and another to display the ongoing conversation. The e4 Contacts Demo Details View will be used as a template for how to organize the code, while the Hyperbola ChatEditor provides the text control details. The resulting code contains classes **ChatSession** and **ChatSession-Composite**, which for test purposes, display a message instead of sending it. The transcript is recorded correctly as seen in the following screenshot:

Cybertete — An E4 Rich Client On Smack



Note that the Selection service notifies the Chat Session view of the selected contact, which is an elegant way to decouple UI elements.

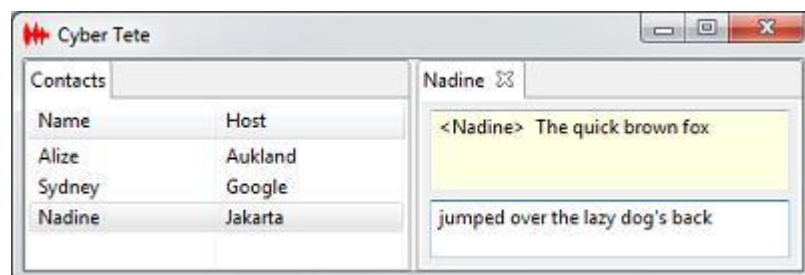
The Chat Session view needs to be hidden until it is initiated, so the handler to initiate a session will need to change the visibility of the Chat part. The Part Service will be used to show the target part. To hide the Chat Session, I have to untick the `%To Be Rendered+` option on the Model part. I also tick the `%Closeable+` option to allow the session to close.



To activate the Chat Session view:

```
partService.showPart("au.com.cybersearch2.cybertete.part.chat",  
                    PartState.ACTIVATE);
```

The problem now is the transcript control gets the initial focus. This is easily fixed by overriding `setFocus()` on `ChatSessionComposite`.



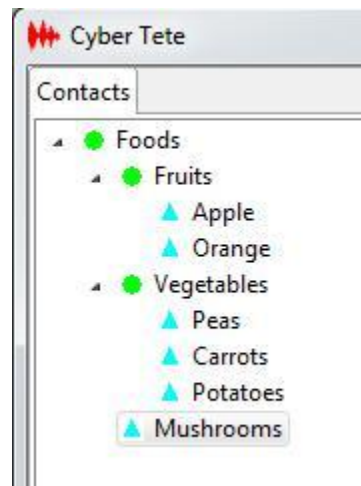
Notice the Chat Session now has a title tab with user name and a close button.

Contacts Tree View

Contacts view needs more work. It needs to be changed to a tree view to show groups as well as individuals and decorated with labels to show the status of users.

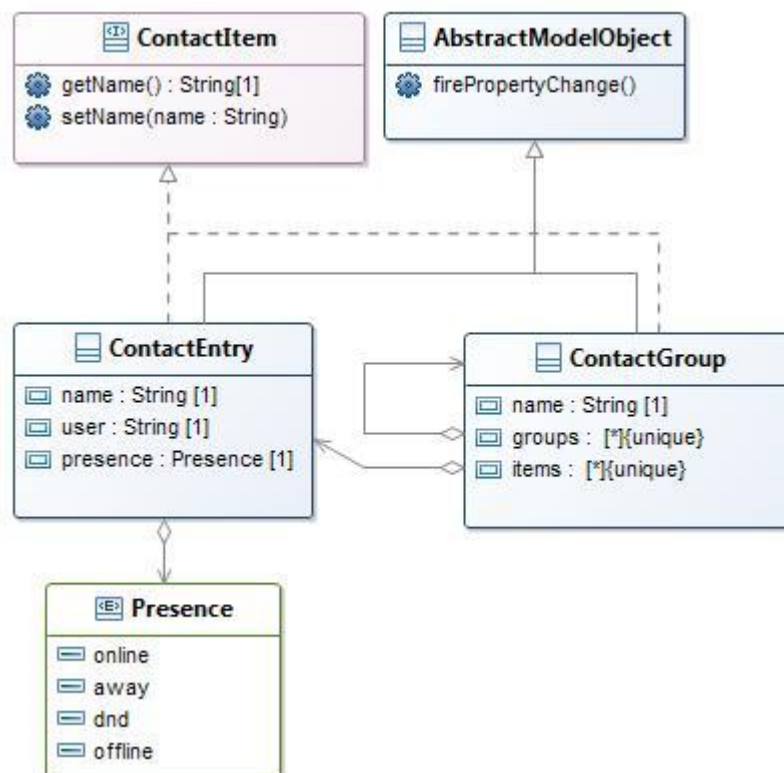
A code snippet in the Eclipse UI Platform examples is worked into a replacement for the original Contacts Table View. It is `Snippet029TreeViewerMultiListProperty`. The snippet displays 2 types of nodes: items and item containers.

The tree is populated with food categories and items as seen in the following



screenshot:

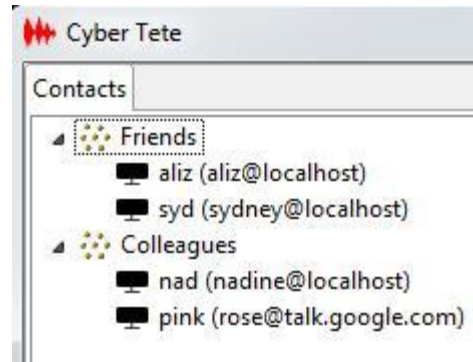
The domain model needs to be filled out so the Contacts view display the cor-



rect details:

There are two Contact Java Bean types to populate the TreeView: **ContactEntry** and **ContactGroup**. They inherit **AbstractModelObject** with its `firePropertyChange()` method to notify the TreeView of changes. Interface **ContactItem** is shared by both **ContactEntry** and **ContactGroup** and is used as the type which operates with the Selection Service to notify when an item in the

TreeView is selected. The Presence enumeration informs the Label provider of what icon to display for the status of a user.



The following screenshot shows the new Contact model classes in use:

Main Menu

Now to add some main menu items. The most important are Exit, Add Contact and Chat. For an e4 application, I have to roll my own Exit Handler, but examples are easy to find.

Hit a bug with Application.xml KeyBinding entries. They are discarded by Eclipse unless tagged with `%type:user+`.

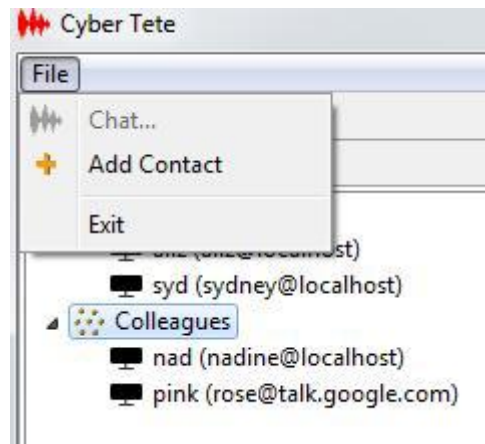
Also I discover the utility of the `@CanExecute` annotation in a Handler assigned to a menu item. This annotation is called continuously and is timer-based and the associated menu item is enabled only when the `canExecute` method returns true. This method checks if the following injected variable is null:

```
@Inject @Optional @Named(IServiceConstants.ACTIVE_SELECTION)
ContactGroup selected;
```

This beats the old way of having to use core expressions in an extension point. If I make the variable type interface `ConactItem`, then the menu item is enabled for any of the TreeView items selected.

A toolbar is now included and key-bindings applied, but the latter have disappeared for some unknown reason. The toolbar is out of sync with the TreeView selection events, and this looks like a bug. Here is how the main menu looks now:

Cybertete — An E4 Rich Client On Smack



That is it for the Application model for now. Trappings such as Help and Status bar are not a priority. What we need now is a Chat Services implementation so we can complete the Domain design and provide the context to apply the remaining UI elements such as user login and preferences.

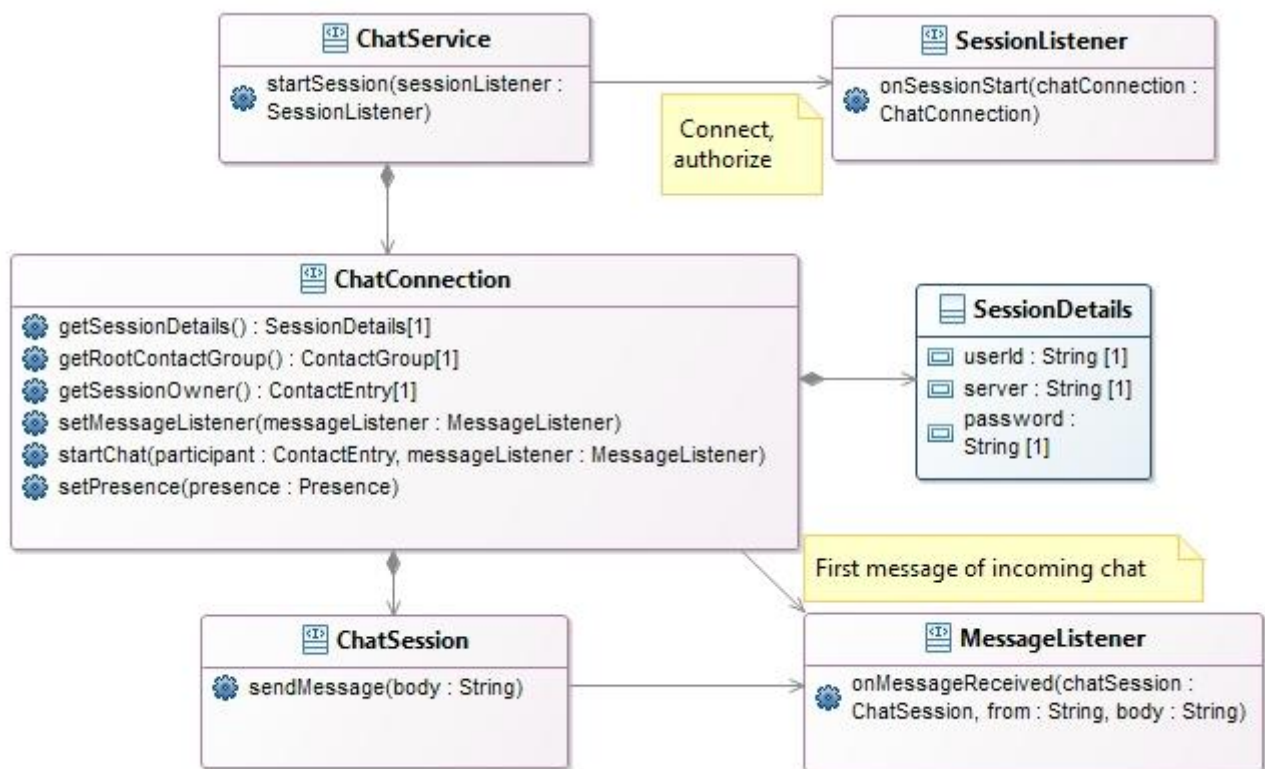
Chat Service

A Chat Service component will encapsulate and hide the actual Messaging implementation. Required operations are:

- Establishing a connection with user authentication
- Handling Chat initiation by either the user or a remote party
- Sending and receiving Chat messages

The contact entities of the existing domain model will be referenced by the Chat Service and will need to be mapped to objects in the actual implementation.

Here is the initial Chat Service class diagram:



The Chat Service establishes a connection with the chat server asynchronously. The login, if required, is delegated to the implementation as well as the persistence of the connection information contained in a **SessionDetails** object. The chat implementation is required to provide contact details as a tree consisting of **ContactGroup** and **ContactEntry** objects. Each item of this tree will have to be kept in synch with its implementation object. For example, changes in **Presence** status need to be tracked for **ContactEntry** objects.

A chat session, once established by the **ChatConnection** instance, is represented by **ChatSession** type regardless of whether the chat is initiated locally or remotely.

Dependency Injection

A service is an excellent candidate for dependency injection as it allows a test service to be substituted to facilitate testing and promotes a modular design. Eclipse 4 makes objects placed in the Eclipse context available for injection, and as a bonus, facilitates the hiding of concrete classes behind interfaces. The key is to have an `IEclipseContext` reference. Here is the code to place a `TestChatService` instance in the Eclipse context to be injected in variables of interface type `ChatService`:

```
chatService chatService = (ChatService)
    ContextInjectionFactory.make(TestChatService.class,
        workbenchContext);
workbenchContext.set(ChatService.class, chatService);
```

Here `workbenchContext` references an `IEclipseContext` object.

The `E4LifeCycle` class `PostContextCreate` method is selected as the location to place the above code. This allows login to the chat server can be completed before displaying the main application window.

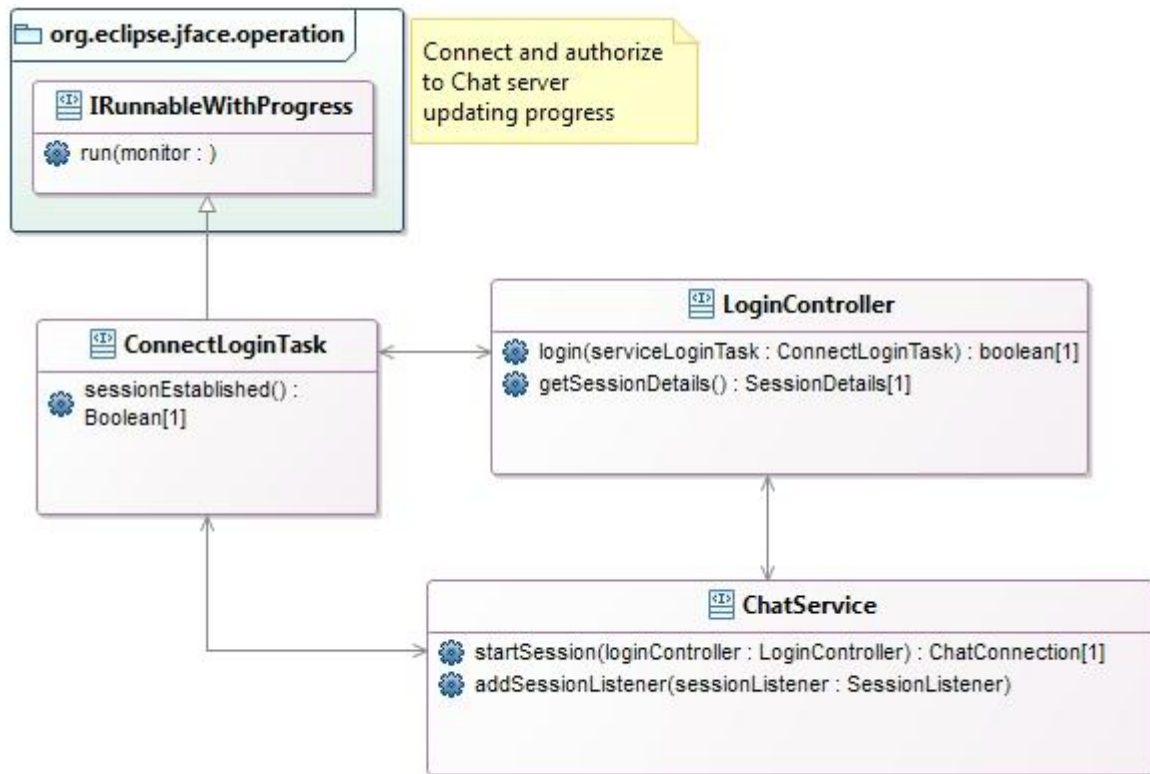
Some experimentation is required to find how to prevent the test `ChatService` implementation from being exported with the the rest of the bundle. If placed in a fragment project, then I am unable to make the test service implementation visible to the host project. I therefore set up a second source folder as follows:

1. Add a new source folder for tests, `src-test`
2. Exclude this in `build.properties` by adding `src.excludes = src-test/`

This topic will need to be revisited once the real `ChatService` implementation comes along as there will need to be a way to swich beteen it and the test one.

Login

The displaying of the login dialog and persistence of connection details is delegated to a `LoginController` class. This class has a mutual dependency on `ChatService` which requires connection details to be supplied by `LoginController`. In turn, `LoginController` requires `ChatService` to indicate if the connection details are valid. A `ConnectLoginTask` interface is created to allow `LoginController` to delegate chat server connection and authentication to `ChatService`. The service is also responsible for updating a progress monitor modal dialog. Here is the Chat Service refactored to collaborate with Login Controller:



Preferences

LoginController is responsible for persisting login dialog values and does so using IEclipsePreferences implementation provided in the Eclipse context. The default preferences storage uses a text file. Here is an example of a preferences file contents:

```
eclipse.preferences.version=1
prefs_last_connection=mickymouse
saved_connections/mickymouse/server=microserver
```

For some unknown reason, the preferences were persisted only if a data location is specified on the command line. I use `%data @user.home+` to specify the user's home directory. The preferences file is placed under this directory at `.metadata/.plugins/org.eclipse.core.runtime/.settings`.

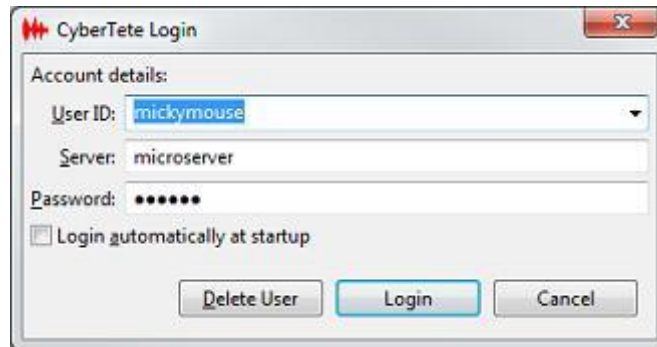
The passwords should not be persisted as clear text, so an Equinox ISecurePreferences implementation is used in this case. The default secure preferences node is obtained by calling `SecurePreferencesFactory.getDefault()`.

The user needs to control the auto login preference and the easiest way to do this is by using a third party `%4PreferencePages+` plugin from OPCoach. This is a replacement for the E3 Workbench Preference Pages.

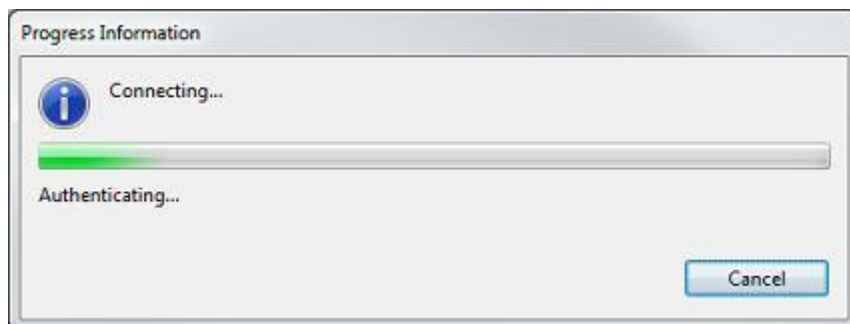
The main menu is updated to include selection Help -> Preferences.

Login Revealed

The TestChatService is updated to mimic the session establishment with sufficient delays to make the progress monitor messages visible. Here is how the login dialog now looks:



And here is an example of the progress monitor:

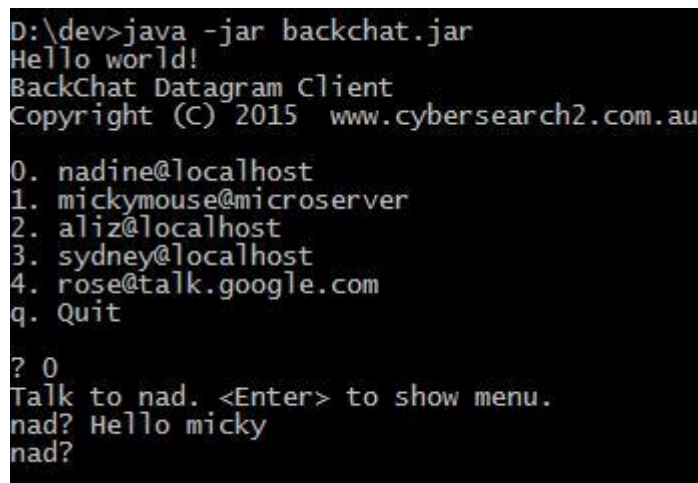


Test of Cancel button fails as there is no response. This is a failure to regularly check for cancel and is subsequently fixed. However, the real ChatService implementation may not be able to provide a similar responsive cancel function.

Going Online

A chat conversation involves at least two parties, so to go forward requires having a remote buddy to talk to. I want to continue prototyping as it allows various scenarios to be readily orchestrated and keeps the design open to creative possibilities. I elect to use a message transfer system based on datagrams which are packets sent over the network using the connectionless User Datagram Protocol (UDP). Not having the overhead of establishing connections keeps the code simple and UDP is reliable when it involves using only the local host. The packet payload consists of the user handle eg. `%andrew@letschat.com` followed by the message. This will be enhanced in the future to allow presence updates to be signalled and add or delete contacts.

I create a Java Console application called backchat to as a datagram messaging client. Here is a screenshot of backchat in action:



```
D:\dev>java -jar backchat.jar
Hello world!
BackChat Datagram Client
Copyright (C) 2015 www.cybersearch2.com.au

0. nadine@localhost
1. mickymouse@microserver
2. aliz@localhost
3. sydney@localhost
4. rose@talk.google.com
q. Quit

? 0
Talk to nad. <Enter> to show menu.
nad? Hello micky
nad?
```

When it comes to networking, receiving is always more complicated than sending. The Chat Service must expect messages to arrive at any time. Each message must be unmarshalled, validated and dispatched according to message type. If going online to the internet, we must also worry about security and denial-of-service attacks, but I am assuming the real implementation will be able to deal with such issues. The remote connection details must be saved so as to send responses back to the remoter user. Most importantly, the application must respond appropriately to incoming messages.

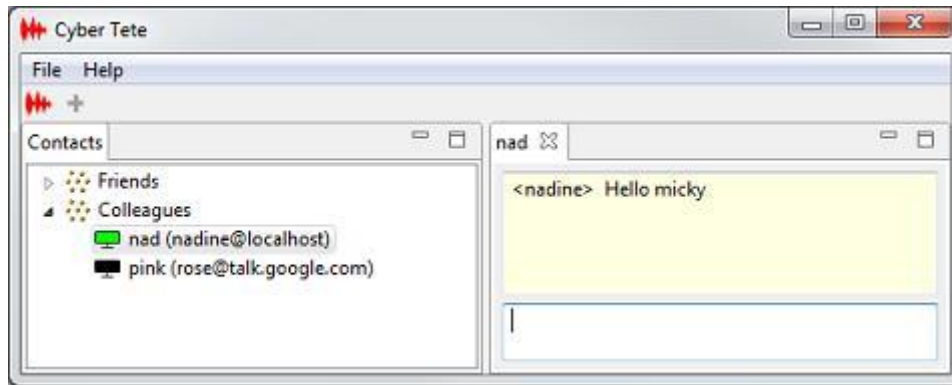
Cybertete TestChatService runs a receiving thread in a continuous loop, forwarding packets to TestChatConnection:

```
datagramSocket.receive(packet);
chatConnection.receiveMessage(packet);
```

TestChatConnection extracts the user handle and checks for an existing TestChatSession for that user. If there is no TestChatSession then one is created and TestChatConnection invokes its own MessageListener:

```
messageListener.onMessageReceived(chatSession, from, body);
```

The message listener needs to reveal the remote user details in the Contacts View , select the contact and bring up the Chat Session View on the right, if not already active. Finally, the message is to be appended to the chat transcript. This activity is taken care of by a class called **FirstMessageHandler**. Here is how Cybertete looks after login and receiving the above backchat message:



First Message Handler

Developing the FirstMessageHandler class requires delving into obscure Eclipse RCP techniques. Control of the application programmatically on behalf of a remote user is not well documented.

The first challenge is to gain access to the view objects specified by the model, in this case ContactsView and ChatSessionView. The first step is easy- get the MPart for the view using IPartService:

```
MPart chatSessionViewPart = partService.findPart(ChatSessionView.ID);
```

Now the ChatSessionView can be obtained by calling getObject() on the MPart, but there is a catch. It appears that the view has to be visible and in focus for this to work. So the handler gives it a try and if null is returned, then get the Part Service to activate the view.

```
MPart part = partService.showPart(ChatSessionView.ID, Part-  
State.ACTIVATE);
```

The next challenge is how to expand the Contacts TreeView to show the remote user and select it. Only by doing some research with the debugger do I realize that a TreePath object is required to do this:

```
Object[] segments = new Object[2];  
segments[0] = contactEntry.getParent();  
segments[1] = contactEntry;  
TreePath treePath = new TreePath(segments);
```

You can see a TreePath contains an Object array tracing the sequence of tree items from root to the item of interest. The ContactItem interface has to be updated to include getParent(). It also emerges that the TreePath segments need to have well-behaved equals() and hashCode() method overrides. So these are added to ContactGroup and ContactEntry.

With the TreePath available, the ContactsView selection and expansion can now be performed:

```
ISelection selection = new TreeSelection(treePath);
contactsViewer.setSelection(selection, true);
contactsViewer.setExpandedState(treePath, true);
contactsViewer.refresh(contactEntry, false);
```

I am pleased to see the Java Beans LabelProvider update the label of the contact to show online presence (green). Overall I am impressed with how clean and compact is the code for Cybertete is at this stage, but there is room for improvement - maybe with experience I will be able to do some things more elegantly.

Chat Window Handler

When I got the First Message Handler to work I was happy with it but as soon as I have to deal with more than one Chat session in progress at a time, I find the going rough. Each Chat session requires its own view which leads to the need for dynamic part creation, something I have never done previously. It gets tricky because the application on start creates a Chat Session View but does not render it. Without this view, the Contacts View occupies all available space of the main window, which looks odd.

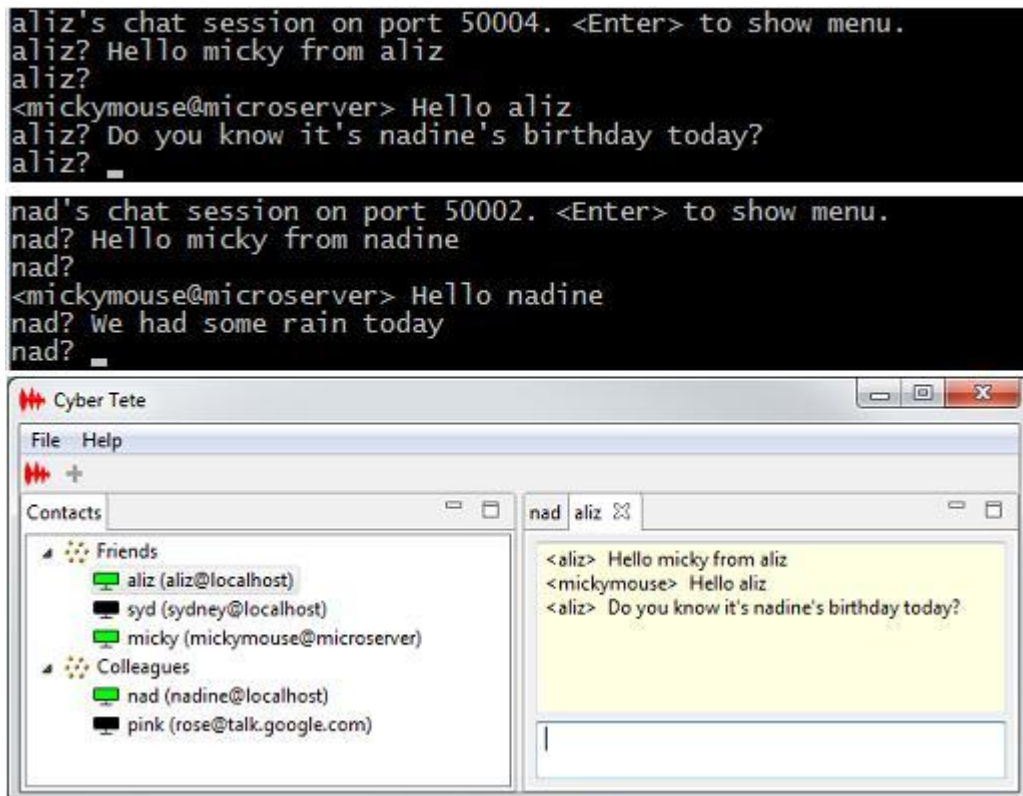
There are two show stoppers:

1. The original design assumes there is only one Chat Session View and that it can pick up the correct Chat participant from the last Contacts View selection. However, with multiple sessions, each Chat Session View participant must be set independently.
2. The First Message Handler does not know who the participant is if the application user has initiated a Chat session.

The fix requires some code refactoring, so the First Message Handler becomes a Chat Window Handler shared by both application and remote user for Chat session initiation. This allows tracking of all windows and participants, solving item 2. above. The handler also sets the Chat Session View participant value, solving item 1. and additionally sets the Label to the participant's nickname, which the original design failed to do for dynamically created windows.

The following composite screenshot shows two simultaneous chat sessions on both backchat clients and Cybertete:

Cybertete — An E4 Rich Client On Smack



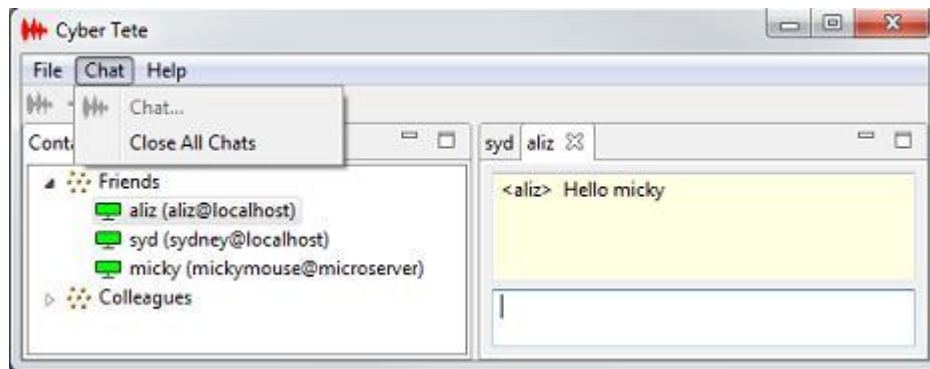
Closing Windows

Work on the Chat Window Handler is not complete because the dynamically created Chat Session Views are closable. The Chat Service needs to know when a session is closed so it will respond correctly to the next Chat establishment request. The ChatConnection interface acquires a new `close()` method which is called within a `@PreDestroy`-annotated method of the closable view. As a precaution, the Chat Window Handler now always activates a window before accessing it in case it has been closed.

The model-rendered Chat Session View is not closable in order to maintain the split window layout of Cybertete. However, the user may want to close the Chat session in that view and refresh it. Rather than have a menu option for just this purpose, I provide a `%Close All Chats+` menu option to close any dynamically opened windows at the same time. As there are now two Chat session-specific menu items, the other being `%Start Chat+`, I create a new top-level `%Chat+` menu item to contain them. For consistency, I add `%Close All Chats+` to the Contacts View popup menu too.

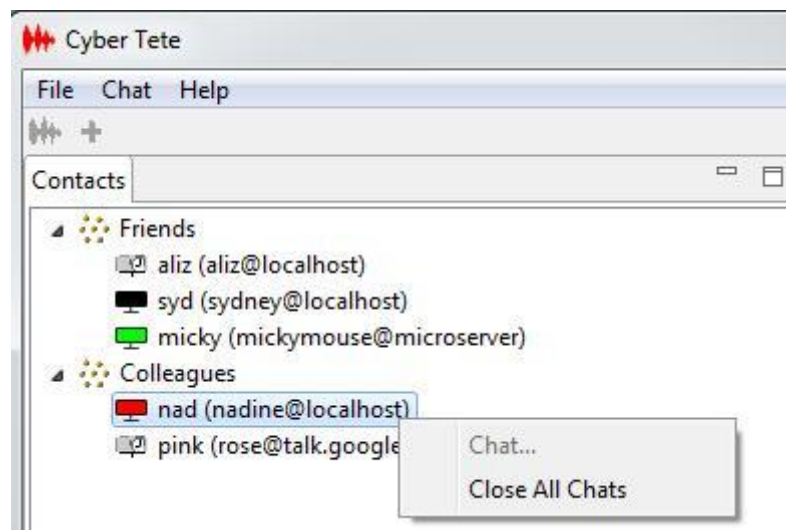
While changing the menus, I also add more control to the `@CanExecute` method of the Start Session Handler so the `%Start Chat+` menu item is only enabled if the selection is for a remote user is not already engaged in a Chat session. You can observe this in the following screenshot:

Cybertete — An E4 Rich Client On Smack



Presence

So far Cybertete is not concerned that a remote user may be offline, which should disable the **Start Chat** menu item. A user having **do not disturb** status should also not be a Chat session candidate. I therefore update the backchat test client to send a presence status when a contact is selected from the menu. The Test Chat Service takes these status messages and updates the participant ContactEntry presence value accordingly. I also disable the **Chat** menu item for selected contacts in **do not disturb** or **offline** state. The following screen shot shows the results of these changes:



The red icon is for **do not disturb** and the grey one with a tiny clock face is for **away**.

The last thing to do is to allow the user to change one's own presence status, which is accomplished by using a **Chat** sub menu item and a dialog.



Chat Service Summary

Starting with an initial service model, I progressively fleshed out a Test Chat Service implementation which allowed me to focus on the user-facing application layer above the service implementation. I had to unearth the wizard-manufactured E4LifeCycle class to implement login as splash start up screen. I worked out how to persist login details, including passwords as preferences.

I then went online using datagrams and a simple Java console application called %backchat+. Getting Cybertete to respond to remote user actions was a lot more tricky than normal user interface design. The first milestone was to expand the remote contact in the Contacts View and display a message in the Chat Session View, even if it had not yet been rendered. The second milestone required some refactoring of the code to overcome problems with the initial design. That was dynamic creation of Chat Session Views.

Tying up loose ends entailed closing dynamic windows and refining control over when the %Chat+ menu item is enabled. Finally , I checked presence status icons and added a new dialog for the user to set presence status.

At this stage, some additions have been made to the original Cybertete model, but all are incremental rather than dramatic. There is a lot more to consider such as help, update support, status bar, task bar icon etc, but while Chat Service is fresh in my mind, I want to proceed with adapting Cybertete to use the Smack XMPP library for the Chat Service implementation. So begins a new chapter.

Smack XMPP Client

Smack is an project on site <https://www.igniterealtime.org/projects/smack>, which states:

Smack is an Open Source XMPP (Jabber) client library for instant messaging and presence. A pure Java library, it can be embedded into your applications to create anything from a full XMPP client to simple XMPP integrations such as sending notification messages and presence-enabling devices.

Smack and Kxml2 Plug-ins

To incorporate Smack into Cybertete, certain jar files from the Smack distribution need to be placed in an Eclipse plug-in. I name the plug-in

org.jivesoftware.smack+and import these release 4.1.4 jars:

smack-core-4.1.4.jar
smack-extensions-4.1.4.jar
smack-im-4.1.4.jar
smack-sasl-javax-4.1.4.jar
smack-sasl-provided-4.1.4.jar
smack-tcp-4.1.4.jar
smack-java7-4.1.4.jar

Note.

Later added to Smack plugin to
lookup DNS SRV records -

smack-resolver-javax-4.1.4.jar

smack-resolver-dnsjava-4.1.4.jar

and org.xbill.dns_2.1.7.jar

from

[http://mvnrepository.com/artifact/
dnsjava/dnsjava/2.1.7](http://mvnrepository.com/artifact/dnsjava/dnsjava/2.1.7)

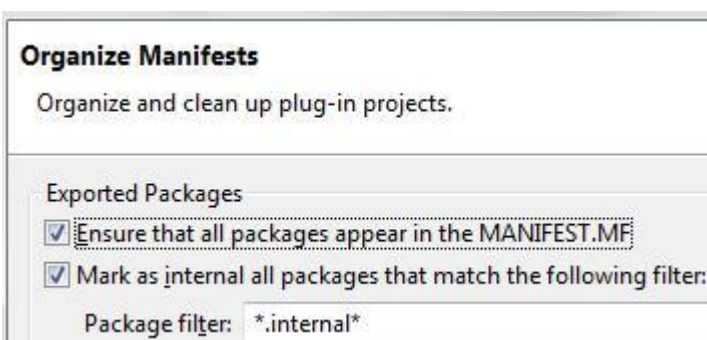
dependency placed in dnsjava plug-

I also have to import 4 dependency jars
available from

<http://mvnrepository.com/artifact/org.jxmpp>:

jxmpp-core-0.4.2.jar
jxmpp-util-cache-0.4.2.jar
jxmpp-jid-0.4.2.jar
jxmpp-stringprep-libidn-0.4.2.jar

To export all the classes I open the **Organize Manifest Wizard** in the smack
plug-in manifest editor **Overview** tab and tick **Ensure that all classes appear
in the MANIFEST.MF**. I accept other defaults by clicking Finish.



The smack plug-in has one third party dependency which requires yet another plug-in to be prepared with name `org.kxml2+`. This exports the classes in `kxml2-2.3.0.jar` which can be downloaded from <http://sourceforge.net/projects/kxml/>

XMPP Login

I make the smack plug-in a dependency of Cybertete, generate Smack Chat Service classes for the three main service model interfaces and commence work on these classes. I have to blend code snippets from the Test Chat Service classes with snippets from Hyperbola specific to Smack. What is good to see is that Smack code scattered around in different places in Hyperbola is now coming together in three Smack-specific classes:

1. **SmackChatService** connects to Chat server, authorizes user, loads roster and monitors contact changes
2. **SmackChatConnection** establishes Chat sessions
3. **SmackChatSession** retains information specific to one Chat session

After development to the point I can login and see contacts with current presence indicated, this screenshot shows what I see:



The first thing that stands out is that mickymouse, the logged in user, is not in either of the contact groups

(Colleagues and Friends). This contact was added by the Chat service because, as it turns out, the roster of a user does not contain the user's own entry. I plan to add a status bar so the logged in user can be shown there along with a pop up menu to allow the user's presence to be changed.

Another observation is that all contact names are account names. Nick names are a feature which has to be implemented locally. A more complete Chat client would allow names to be renamed and persisted.

How to associate Smack roster items with Cybertete contact items took one failed approach before getting it right. The first approach was to use inheritance eg. `SmackContactEntry` extends `ContactEntry`, but this broke the scheme for updating the Contacts View when property changes occurred. Therefore presence updates were not appearing the Contacts View.

The second approach is to use containment . A roster item and contact item are both fields of an association class, which for entries, has a `sync()` method

to keep the contact entry in sync with the roster entry.

Chat Sessions

I progressively get Smack to establish Chat sessions and send and receive messages. It is like putting together a jigsaw puzzle where the pieces consist of code snippets. Something new to deal with is that the Spark Chat Client with which I am testing sends a message as soon as the user starts typing. I assume this allows %remote user is typing+to be indicated. Fortunately, I can add a message listener to the Smack Chat object to start the session on the Cybertete side when the first message is received. However, I then have to arrange for this special message listener to be removed when the normal one is set.

I believe the decision to get the application layer design sorted out, working with a domain model prototype is the right one. When it came to swap in the real domain implementation, I had 5 cohesive classes to create with a well-defined interface to the rest of the application. The fallout from having a few concrete classes in the interface was that I to create 2 small association classes, one of which had a sync() method to keep domain and application layers in sync.

The next concern is completion of the various contact operations that need to be implemented. There is the %Add Contact+menu item, which is enabled when a group is selected in the Contacts View. There are also notifications received from the Chat server of changes to the roster, which consist of collections of entries added, deleted and updated, all identified by user JID. Hence the following topic is %working with Smack Roster+.

Working with Smack Roster

Upon successful login, the Smack Chat Service builds a contacts tree consisting of ContactGroup and ContactItem model objects. The information required to do this is obtained from the roster of the signed in user. The contacts tree has been designed with the simple assumption there is one contact entry per contact and it is assigned to just one group. This does not reflect real life as a contact may be a member of more than one group. A contact may also be online with more than one resource, but in this case, each contact plus resource is treated as a distinct entry.

Each Smack roster entry has a list of groups to which it belongs. To adapt such roster entries to contact entries, there needs to be one contact entry created for each group belonging to the roster entry. This is a consequence of using a TreeView to display contacts. To handle the complexities of building a contacts tree and keeping it up to date, I create a **ShadowRoster** class.

The ShadowRoster class imports the contacts tree construction code and presence update code that I initially placed in SmackChatService. The ShadowRoster class extends ContactGroup so it can take on the role of being the root of the contacts tree. Changes to other classes at this time include:

1. Updating ContactEntry class to include parent name in equals and hashCode evaluation as the user value is no longer unique on its own
2. Deleting SmackContactGroup class as it has no operations to perform
3. Adding a next-link to SmackContactEntry class so objects can be chained for a user belonging to more than one group

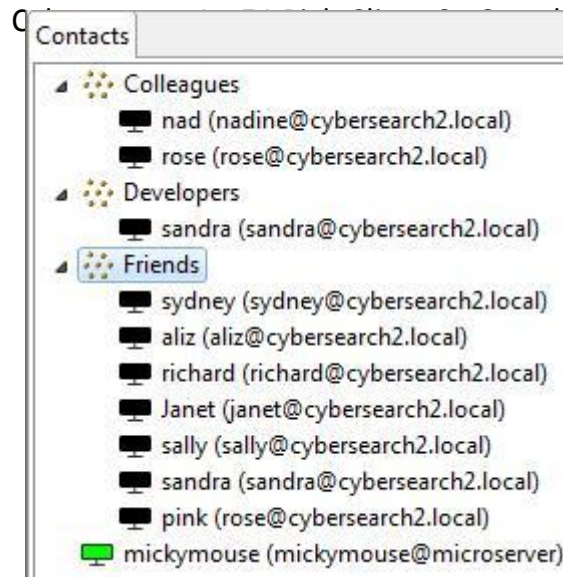
Add Contact

XMPP allows a client to add a new contact entry to the roster, but the process depends on subscription type and may involve participation of the chat buddy+. When the process completes successfully, a notification is received indicating a new entry has been added to the roster. Note there are a lot of caveats in how this feature would be used in a practical system. It is a matter of policy whether a client can add contacts at all, and if so, whether only local accounts are permitted.

It takes a while to get a successful Add Contact+operation to work. The server keeps sending back XMPP error not-acceptable - modify+response. I find this indicates a problem with the subscription process. I install on the Chat server (Openfire) a plugin that can be configured to automatically accept subscription requests and set the server up to accept local accounts. I still get same error until I modify the create entry request to include existing groups to which the new contact belongs as well as the new one.

Now to add a new contact at the server and see if Cybertete shows it in the Contacts View while running. Having the target group expanded in Cybertete, I expect the new contact will appear as soon as it is added at the server. It is a relief that this works almost straight away. I now try to add a new client to a new group and again without much trouble the new group to appears in the Contacts View containing the new contact.

The following screenshot shows at this point I now have more contacts in the Friends+group and a new Developers+group which popped up when I added it at the server:



One thing to note is that contact `rose` in group `Colleagues` has been added by me to `Friends` with nickname `pink`.

Reflection on Design

I reflect at this point on what the author of *Eclipse Rich Client Platform* chapter 10 says about approaches to integrating the Smack infrastructure into Hyperbola. They are:

- Proxy all the Smack classes behind the existing prototype model and try to run without any changes to the actions and UI
- Delete the prototype model and replace it with Smack directly

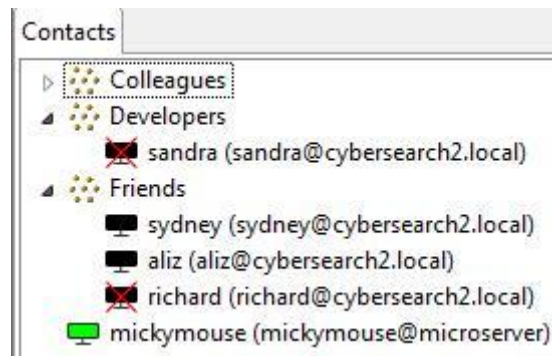
Of the proxy approach the author says `it creates duplicate classes and overhead for keeping the proxies synchronized with the Smack classes`. Now I believe from a design perspective, only the `proxy` approach makes sense, otherwise the well-established principle is violated of layering applications to defeat complexity. I understand the approach used in the book to mean there was pressure to meet a project deadline. Development is always about trade offs and compromises. What Eric Evans says in his book *Domain-Driven Design* is that when development is iterative, and therefore involves frequent code refactoring, `past design choices make refactoring itself easier or harder`. Retaining a prototype model in code is always a good choice so that the inevitable future changes in the design are more manageable than they would otherwise be.

In Cybertet's case, the Chat service model, consisting mostly of interfaces, has both a prototype and Smack implementation. Developing the Smack version revealed unforeseen requirements such as the need to support a contact being assigned to more than one group. These requirements were addressed mostly in the confines of the Smack service and the rest of the application

code was largely untouched.

Update/Delete Contact

Finishing off the contact operations is easy. I find that the deleted entries call is made post deletion, so in that case, only shadow details need changing. I decide to respond to contact removal notifications by simply changing the contact presence to a new `%deleted+` value which has a icon the same as offline, but crossed out. This is preferable to contacts that simply vanish from view. Here is a clear case where having a proxy `%presence+` is better than directly using the Smack implementation. It allows flexibility in the application design. The addition of a new icon is easy because all icons are provided by a factory class. The following screenshot shows the result of, on the server, contact `%richard+` deleted and sandra removed from group `%Developers+`:



Tycho and Equinox P2

I now have an application with 42 Java source files that is the basis for a deliverable product. So now is a good time to consider the process to build and package the application and provide updates as bugs get fixed and features are added. If possible, I would like to do production builds on the Jenkins continuous integration service I have running on my development server. Fortunately, the Eclipse community provides the technologies for a mature production process: Tycho and Equinox P2.

Tycho is a set of Maven plugins and extensions for building Eclipse plugins and OSGI bundles. Tycho uses native metadata to discover dependencies, location of resources and other information which otherwise would have to be explicitly placed in the Maven configuration.

Equinox P2 is the provisioning technology for OSGI applications. It provides a repository system and supports the various aspects of installing applications and keeping them updated.

Both Tycho and Equinox P2 are new to me, but I have learnt that I will be creating a number of projects to cover specific aspects of product definition and construction activity.

Getting Started with Tycho

I use the *EclipseCon 2014 Tycho Tutorial* to guide me on the steps to take to Mavenize the project and apply the Tycho automation to it. This tutorial comes with detailed step-by-step instructions and screen shots. As one would expect, some of the details are out of date, so I will point out where the tutorial needs amending. The version of Eclipse I am using is Eclipse for RCP and RAP Developers, Version: Mars.1 Release (4.5.1). For Tycho, this version requires downloads from two sites:

Name	URL
m2e releases	http://download.eclipse.org/technology/m2e/releases/
Tycho m2e connector	http://repo1.maven.org/maven2/.m2e/connectors/m2eclipse-tycho/0.8.0/N/LATEST/

Note that suggestions the Tycho m2e connection is available through the Eclipse Maven Discovery function are incorrect. Also the Tutorial also makes an obsolete reference the Eclipse E4 Tools, which are now included in the Mars releases.

The tutorial originally came with an Maven repository snapshot to permit working off line. As it turned out, I have downloaded Tycho version 0.22.0 to my local repository to build a sample project, and as it is recent, will use this version for my build. Another version needing attention is the Eclipse repository URL. The tutorial points to Luna, but I am using Mars, so I select:

<http://download.eclipse.org/eclipse/updates/4.5>

A Tycho RCP application consists of several Eclipse projects. There is a single top level project which is parent to all the rest. The parent project defines a Tycho plugin version which is shared by the application modules. It also defines an Eclipse repository which needs to align with the version of Eclipse used for development. Each module starts life as a non-Maven project, gets converted to Maven and then is assigned as a module of the parent project. This process is a little clunky because spurious errors are generated until dependencies are resolved.

This is the Maven conversion step:

Right-click on the eclipse project and select
Configure > Convert to Maven Project

Parent Project

Before attending to Cybertete, I prepare the parent project, basing it on the one I prepared for the Tycho tutorial. I copy the files Tycho parent files to a new Cybertete folder named `au.com.cybersearch2.cybertete.parent`. I fix up details to suit Cybertete such as the name in the `.project` file and the modules in the `pom.xml` file. To start with there is just one module- the Cybertete project I have been working on, but shortly there will be several more. Here are the chief elements of the parent POM:

```
<modules>
    <module>../au.com.cybersearch2.cybertete</module>
</modules>
<repositories>
  <!-- configure p2 repository to resolve against -->
  <repository>
    <id>eclipse-project-mars</id>
    <layout>p2</layout>
    <url>http://download.eclipse.org/eclipse/updates/4.5</url>
  </repository>
</repositories>
```

continued next page/...

```

<build>
  <plugins>
    <plugin>
      <!-- enable the Tycho build extension -->
      <groupId>org.eclipse.tycho</groupId>
      <artifactId>tycho-maven-plugin</artifactId>
      <version>${tycho-version}</version>
      <extensions>true</extensions>
    </plugin>
  </plugins>
</build>

```

I now import the parent project into the workspace.

Maven Conversion

Now I convert the Cybertete bundle project to Maven. This involves navigating the %Create new POM+dialog and setting the %Packaging+field value %eclipse-plugin+, which has to be typed in as this option is not available in the drop-down control.

Upon completion of the Maven project conversion, there is one expected error: %Unknown packaging: eclipse-plugin+. This is caused by the tycho-maven-plugin extensions not being wired in. The just created POM needs updating to insert a parent element. This is what the Cybertete POM contains when I get the first successful build:

```

<artifactId>au.com.cybersearch2.cybertete</artifactId>
<packaging>eclipse-plugin</packaging>
<name>Cybertete</name>
<description>Instant Messaging Client</description>
<parent>
  <groupId>au.com.cybersearch2</groupId>
  <artifactId>cybertete.parent</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  <relativePath>../
au.com.cybersearch2.cybertete.parent</relativePath>
</parent>

```

Some things to note here:

1. The artifactId of this POM must match the project name. This means giving forethought to the project name knowing it will become an artifactId.
 2. The groupId and version are specified in the parent element
 3. The parent POM location needs to be specified in a relativePath element
- Apart from minor glitches with the new POM, I also have to deal with this error:

Missing requirement: au.com.cybersearch2.cybertete 1.0.0.qualifier requires 'bundle com.opcoach.e4.preferences 1.0.0' but it could not be found

So I have to find the answer on how to resolve the third party plugin dependencies when using Tycho.

Third Party Plugins

I find that although such dependencies can be resolved using a Maven repository, it is recommended to use an Equinox p2 repository. In my research I find that if is one thing Eclipse is not short of, it's ways to create a p2 repository, including using Tycho itself. These are the steps I take:

1. Export the third party plugins to a directory location
2. Create an eclipse target using File -> New -> Target definition
3. Install the third party plugins into it, then
4. Create a feature, add the plugins
5. Select the feature project and export the p2 repository to a directory location.
6. Add the p2 repository to the parent POM using a file URL

It is regrettable that there is not a simpler way to get going with Tycho and this is apart from possibly having to deal with non-bundle dependencies and placing the repository on a web server for team access. I add a `third-party-repo` property for the repository URL in the parent POM:

```
<repository>
  <id>third-party-plugins</id>
  <layout>p2</layout>
  <url>${third-party-repo}</url>
</repository>
```

Add a Feature and p2 Repository

I skip adding a test fragment and packaging it as an eclipse-test-plugin as there are no JUnit tests at this stage. I go on to adding a feature, which I understand is a preliminary step for deployment to a p2 repository. These are the steps:

1. Create a new feature project `au.com.cybersearch2.cybertete.feature` which includes the `au.com.cybersearch2.cybertete` plugin
2. Convert the feature project to a Maven project (packaging: `eclipse-feature`)
3. Add the feature project as a module to the reactor, and configure the feature project's parent POM
4. Perform Maven Update Project on feature project

A test build completes successfully, so on to creation of a p2 repository. To quote the Tycho Tutorial:

%A p2 repository is the format through which we can deliver new features to existing applications, or deliver updates of our RCP application.+

1. Create a new Update site project named
au.com.cybersearch2.cybertete.repository
2. Rename the site.xml to category.xml
3. In the category.xml editor, create a new category
au.com.cybersearch2.cybertete.category and add the feature
au.com.cybersearch2.cybertete.feature to it
4. Add the new project to the build with eclipse-repository packaging type

A test build completes successfully, so apart from having to spend time puzzling over third party plugins, the Tycho set up goes smoothly. The next step is working out how to build a product distribution.

Product distribution

There are several steps this time to take with the end goal being a Windows exe file I can run on my PC. I find that the product file editor now has a %Contents+tab instead of %Dependencies+and pressing %Add required+button has different results to those shown in the Tycho Tutorial. When I try to launch the application from the overview tab, I get an exception for a missing Equinox security bundle. That would be from the secure preferences implementation. The editor helpfully shows all the features with that bundle and I take a stab at org.eclipse.equinox.p2.rcp.feature. That fixes the problem and Cybertete launches successfully.

After finishing the repository build configuration, there is an eclipse.exe in the repository. I change the name to cybertete.exe using the Launching tab of the product editor. I also remove the %clearPersistedState+program argument which was copied from my Eclipse run configuration.

I now add a p2 advice file (p2.inf) to the repository project to customize:

1. Remove file eclisec.exe from the distribution- it is a console launcher and not required
2. Specify the p2 repository location for update functionality which I am just about to add to the application

Here are the contents of the p2.inf file (last line wrapped to fit the page):

```
instructions.configure=\
    addRepository(type:0,location:file${#58}/C${#58}/target/repository);\
    addRepository(type:1,location:file${#58}/C${#58}/target/repository/);
instructions.configure=org.eclipse.equinox.p2.touchpoint.natives.remove
(path:${installFolder}/eclisec.exe);
```

This is another place in the sources where the details need adjustment on in-

stallation until I make the repository publically available.

p2 Update Functionality

This step adds an Update sub menu item under the Help main menu item. I follow the example in the [Eclipse 4 application—Eclipse RCP Cookbook Blog](#).

This adds is an update handler to launch a Provisioning Session to check if updates are available and install them if the user wishes to proceed. The p2.inf repository instructions shown above informs the Update Operation where the updates are located.

The update functionality requires some Equinox bundles to be added to the application dependencies. Also required is feature org.eclipse.equinox.p2.core.feature, but this is already installed. There is not much to do apart from add a command, handler and update menu item.

Unfortunately this simple change fails with message %Unable to resolve provisioning job+. I puzzle over this for some time until I find [Lars Vogel Eclipse p2: Updates Tutorial](#) where the same error is more helpfully reported with this message: %Trying to update from the Eclipse IDE? This won't work!+.

Testing this time with the deployed cybertete.exe, I get expected %Nothing to update+. Now a bigger step is to port everything to my Linux server where Jenkins performs continuous integration builds .

Linux x86_64

I copy the entire project to the Linux server using Git clone. To start a build, I go to the parent project and run %mvn clean install+. One of the advantages of a clean build on a different platform is it exposes portability issues and the first one is reported for a relative path which is Windows-specific. The next issue is the third party repository cannot be located. No wonder as this repository has not been incorporated into the project. I do so now and update the parent POM %third-party-repo+property to portable %file:\${basedir}/../third-party-repo+. Now I get %BUILD SUCCESS+but it is a Linux x86_64 executable that is produced. Back to the product configuration to explicitly specify what platforms I want to support.

To get the %Export for multiple platforms" checkbox to appear on the product editor, I have to update the Cybertete build target to add %Eclipse RCP Target Components+and %Equinox Target Components+. The details are on the [Building+page of the Eclipsepedia](#). In the IDE I can now select which targets to export, but how to tell Tycho? Fortunately I quickly find there is a %target-platform-configuration+Tycho artifact which allows the same selection. The next Linux build produces 4 targets which are Windows and Linux x86 and x86_64 architectures.

As an aside, it took me a while to come competent with setting up an Eclipse target. It is actually quite complicated dealing with different sites of different

versions which display grouped by category, or not and show features when it's a missing bundle I am looking for. Compounding all this is the fact the plugin error markers in the various editors do not clear when a target configuration is fixed. I have to restart the IDE to see the effect of any target configuration change. To help those following in my wake, I add the Cybertete target file to the project.

Jenkins



Jenkins is a continuous integration and continuous delivery application which requires minimal configuration when used on a Maven project which Cybertete has now become. The configuration basically needs three pieces of information:

1. A Git repository URL to fetch the project
2. A path to the parent POM relative to the project root location
3. The Maven command line goals and parameters

Build #1 fails because I get item 2 wrong.

Build #2 fails because of an issue with item 3. The error `%Could not assemble p2 repository: Mirroring failed: No repository+` is very obscure, but fortunately I find the solution on an Eclipse blog and that is to change my goals from `%clean package install+` to `%clean package+`. The original goals I copied from an article on using Tycho with Jenkins, but it turns out the `%install+` goal is superfluous and it makes Tycho misbehave.

I have great hopes for Build #3 and decide it is time to make changes to set up a workspace Maven repository, otherwise all Maven dependencies have to be downloaded every build. Build #3 works!

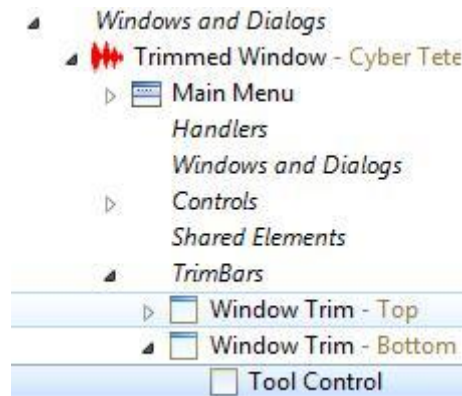
Automation Positives

When I started work on automated builds, I only had a single application project, three separate bundles containing third party jars and no support for multiple platforms or installing updates. By applying the Tycho and Equinox p2 technologies, the project has grown and the issues just mentioned addressed while at the same time getting it set up for an agile development process involving Continuous Integration. What is now wanted is an application worthy to ship to the eagerly waiting crowd of potential users.

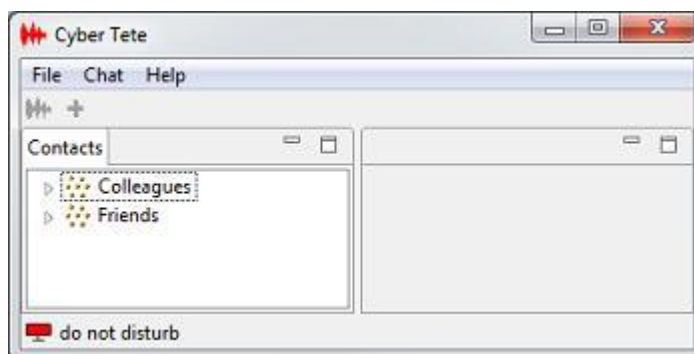
Gaining Status

The next challenge comes from *Eclipse Rich Client Platform* chapter 6 in a section titled *Adding to the Status Line*. Most instant messaging applications place an indicator in the status line to show the user's online status and presence. Eclipse 4 does not by default place a status line in the main window, and if one is wanted, it is up to the developer to put one in.

The first step is to put a Tool Control in a bottom Window Trim of the main window:



Now what to put in the Tool Control? The answer for someone like me not strong on JFace is to use the Workbench StatusLineManager class. When used in Eclipse 4, this class is under direct control of the application and there are no surprises. I soon get a status line with *Hello world!* displayed aligned left in what is called the *message area*. This is a good place to put the presence indicator consisting of both image and text. Here is an example of *do not*



disturb being indicated on the status line:

One thing to note is that the message width does not appear to automatically adjust for change of content causing text to be elided eg. *do not disturb* appears as *do not disturb*. A pack operation performed after changing text fixed it:

```
statusLineManager.getControl().pack(true);
```


An E4 recommendation I follow is to use Events to signal changes to the status line. The dependency injection system even takes care of event registration. In this case, the event sender is the **ChangePresenceHandler** class. It uses an injected `IEventBroker` to post an event. The receiver is the class assigned to the bottom Tool Control and it has one `@UIEventHandler` annotated method for each of the possible presence states. Event types are strings which need to be unique to the application as well as the Eclipse infrastructure. I create a new `CyberteteEvents` interface to define an event type for each presence state value. The content of an event is the text to display such as `%do not disturb+`. Here is the online event handler as an example:

```
@Inject @Optional
void onlineHandler(@UIEventTopic
(CyberteteEvents.ONLINE) String presence)
{
    statusLineManager.setMessage(
imageFactory.getImage(Presence.online), presence);
}
```

Note that the `@UIEventHandler` annotation ensures the event runs on the main thread. This is very elegant.

Presence Broadcast

So far the application only displays presence status, but it must also broadcast to the contacts in the roster. There is a `SmackChatConnection setPresence()` method, but it is not implemented. This method receives a model `Presence` enum and must create a corresponding XMPP `Presence` object. The Smack XMPP connection is then delegated to send a `Presence` packet. The same connection will also send an `%available+` `Presence` packet when the connection is established and an `%unavailable+` one on connection close. The following screenshot shows a Spark Chat client displaying first successful presence up-

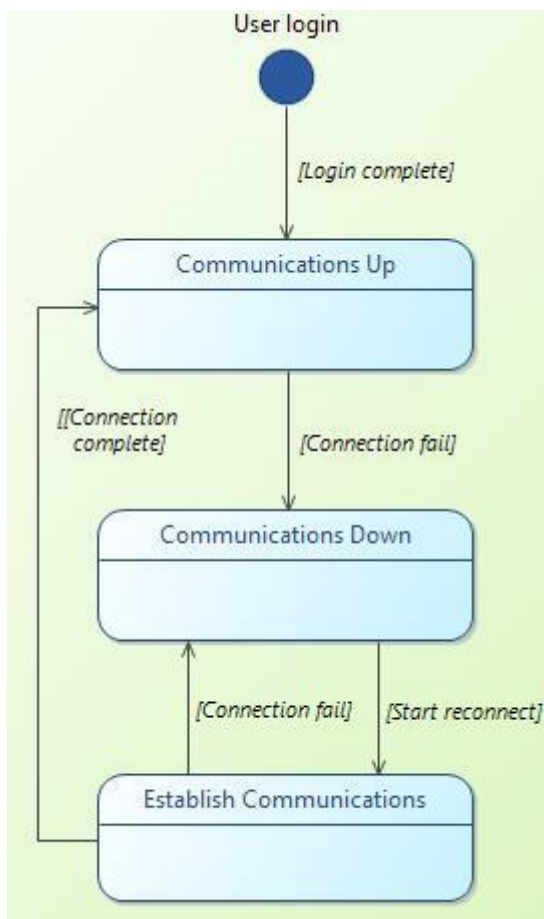


date - mickymouse status `%do not disturb+`

Through the presence facility the user gets to set availability status from %Free to chat+(online) to %Do not disturb+, but there is also the status of being unavailable due to being offline. The Chat Service model does not make provision for going offline, but it may happen if there is a network failure or the Chat server goes down for some reason, perhaps for scheduled maintenance. That leads to the topic of handling changes in connection status.

Connection States

Cybertete does not show the main window until login is successfully completed. Currently, if the connection fails, there is no indication of failure given to the user or any attempt to reconnect. Once these two issues are sorted out, there is the question of how to assure the user the application is running a recovery operation and not simply hanging. And the final question is how to indicate reconnection is complete and resume normal Chat activity. The best way to handle this is to think of the application having a state machine where there three states as shown in the following diagram:



The states are:

1. Communications Up
2. Communications Down
3. Establishing communications

The machine starts upon successful login. When communications fails, it is expected that the Chat Service implementation will initiate recovery using a sensible strategy that will not stress the network and Chat server. Once reconnection is complete, the machine returns to its original state. It may take more than a single reconnection attempt for this to happen, hence transitions shown going in both directions between the second and third states.

There is a lot to do so I will approach development in stages to make it manageable. Stage one is to add a communications indicator to the status line and use a

Smack Connection Listener to send events to the status line when Communications state transitions occur. The second stage is to add the state machine to the Cybertete domain model to achieve proper architectural layering.

Communications Indicator

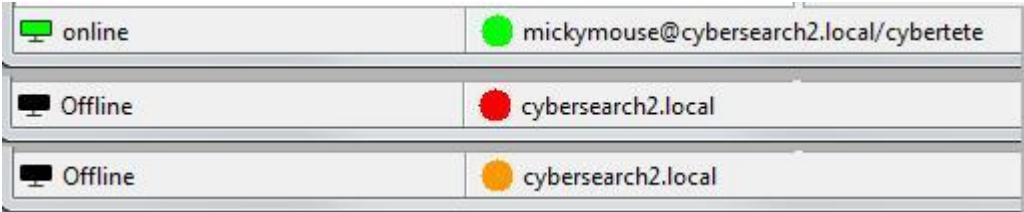
The new indicator will go to the right of the presence indicator on the status line. It will consist of a circle image with a traffic light color and text:

Communications State	Color	Text
Up	Green	User JID
Establish	Orange	Host domain
Down	Red	Host domain

To place this indicator in the status line, it must be added as a contribution item. Fortunately the Hyperbola sample code comes with a ready-to-go `Status-LineContribution` class which can be extended to create a `ConnectionStatus-Contribution` class. Like the presence indicator, I can use events to signal status changes to decouple it from the connection listener.

The Smack library comes with a reconnection manager and it is enabled by calling a static method. There is also a `ConnectionListener` interface which is used to track changes in (re-)connection status. I add a connection topic to `CyberteteEvents` and events `%up+`, `%down+` and `%establish+`. I am then able to develop a `ConnectionListener` implementation to signal to the communications indicator. The following composite screenshot shows how the status line looks

for each of the three states:



Once the reconnection manager starts trying to recover communications, the indicator is mostly red, turning orange momentarily at each reconnection attempt. Note that the presence indicator goes to offline state when communications goes down.

Domain Module

To add the communications state machine to the domain model, I create a

new Chat Service interface called `ChatConnectionListener` which has three event methods: `onCommsUp()`, `onCommsDown()` and `onEstablishComms()`. These events are intended to drive the communications indicator on the status line, but may also be of interest to a logger. I add one more method, `isOnline()` as this is an important piece of application status information which any implementation can readily supply. I create an implementation of the new interface which as it tracks connection status, is called `CommunicationState`. In the `E4LifeCycle` `postCreate()` an instance of `CommunicationState` is added to `SmackChatService` and also inserted in the Eclipse Context so it can be used in the `canExecute()` methods of handlers which should only be enabled when communications is up. The `Smack ConnectionListener` from stage one of development is modified so it triggers `CommunicationState` instead of the status line. When I crank `Cybertete` with the revised domain model it works the same as in stage one, so now it's time to see what remains to be done to recover completely from communication interruptions.

Complete Recovery

Here is a list of things to fix to facilitate resumption of service when the Chat server is stopped and then started a minute later:

1. Disable `%Start chat+` and `%Add contact+` menu items while communications is down
2. Inhibit the text entry fields on all chat session views

Item 1 is fixed by injecting the `CommunicationState` object into handlers for menu items and adding check for online in `canEnable()`.

Item 2 takes more effort as the `ChatSessionView` class and its `ChatSession-Composite` dependency need to be enhanced so the view can be disabled and enabled. The ability to display a message is also added so the user can be informed what is happening.

`CommunicationState` is updated to send new session events which signal when the online status changes. `ChatWindowHandler` is made the receiver of these new messages. There is a catch in that an event is fired every time a reconnect attempt fails, so a guard against receiving duplicates is required. `ChatWindowHandler` loops through the collection of chat session windows to enable or disable each one according to session state indicated in the incoming event.

Now that the original goal of showing the user's online status and presence has been fulfilled, is there anything else to go on the status line. The answer is yes and that is because security is a concern when one is online to the world

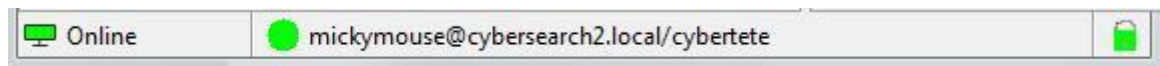
Security Indicator

The Smack library by default establishes a secure connection to the Chat server. The most important component of online security is message encryption using a trusted third party to prevent man-in-the-middle snooping. I propose to add an icon to the status line on the left to show the connection is secure. As a bonus, I plan to show the server certificate when the user clicks on the security indicator.

The security indicator, like the communications status indicator, must be a contribution item to the status line. Getting the desired layout with the icon in the left hand corner is difficult and then I find that there is no automatic adjustment when the main window is resized. It looks like a custom status line may be a better option.

To drive the new indicator I define in `CyberteteEvents` a network topic with three events - `%unavailable+`, `%connected+` and `%secure+`. To incorporate security in the domain model I add a `NetworkListener` interface to handle these events. The secure event passes on the peer certificates received during the SSL handshaking. This information is obtained by hooking the `HostnameVerifier` that Smack is using and accessing the `SSL Session` object passed when the host name is verified.

Here is the status line with the security indicator showing SSL is protecting the link to the Chat server:

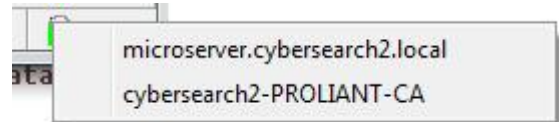


Status Line Popups

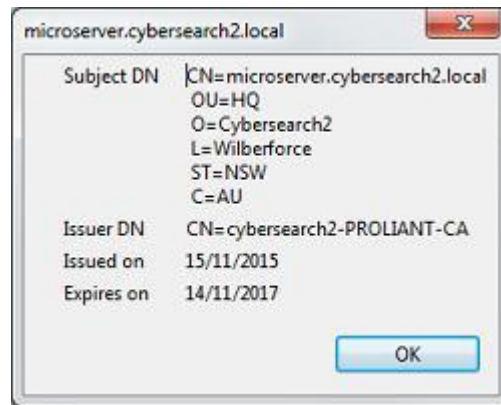
Each item on the status line should have a context popup menu or dialog to allow the user to change settings and view more information. However, I find there is a context menu added to the status line message area to copy the text to the clipboard and I cannot override it in some way. Therefore I am forced to reinvent the status line so the presence indicator can have a popup to change the setting. I end up with a simpler status line implementation. The only workbench class still employed is `ContributionManager`. I also fix the resizing problem by making the status line width the same as that of the shell client area. Before, the width was fixed as the sum of all the content item widths.

The status line is enhanced to when the presence indicator is right clicked, a modal dialog pops up to allow the user to change the presence setting. The security indicator, when showing security is on, gains a context popup menu with the items being the subjects of the certificates received from the Chat

The first item is the server certificate, which has the full host name bound to the server. The second item the Cybersearch2 trust certificate.



Upon selection of a menu item, a simple certificate dialog appears eg.



The communications status indicator is a good place to popup a context menu to allow the user to sign off and log on with different user and /or host details. This is going to be a sizeable piece of work as the current connection has to be shutdown, all the chat sessions and associated views closed and everything tidied up so the application can start over again. The menu is easy. It is what happens when the %New logon+item is clicked is where the real challenge lies.

Logon Logoff Cycle

As there is now a NetworkListener interface for the ChatService, the communications establishment can take place in its own thread while the login task tracks progress by listening for network status updates. This will also allow the cancel button on the login progress dialog to be responsive. I also assume that the Smack code is going to complete faster when there are no thread context changes to update a modal dialog.

Some thought into the design is now required because signing off is much like communications down, but in this case, it is not an error condition. The best way to handle this is a combination of the following:

1. Set presence indicator to offline state
2. Blank both the communications status and security indicators
3. Close all Chat session views.
4. Present login dialog and exit the application if cancel button is pressed

A new connection event for log off is required to trigger these actions. This event can also be used to initiate an orderly shutdown giving the application

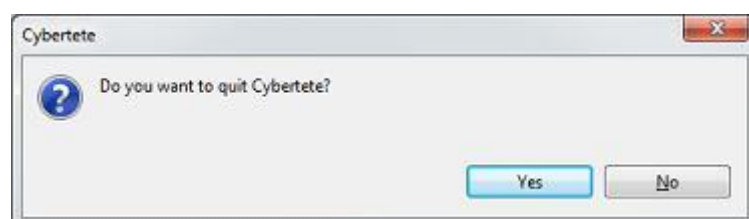
the opportunity to save chat session conversations and update any logs that are running. Hence the logout event data needs to indicate the required following application state- logon or shutdown in contrast to the current running state. A new Application state machine is to be added to the domain model.

The obvious receiver of the logoff event is CommunicationsState class, which tracks communications status events to determine if the application is online. Upon receiving a logoff event, it will call close() on the Chat service, change the online value to %false+ and display the login dialog if the next application state is %logon+. The Smack Chat Service close() method has not been implemented, so this needs to be done. The close operation needs to call disconnect() on the current connection and prepare for startSession() to be called on the service.

To say there are challenges to getting the Chat service to close and establish an Application state machine proves not to be an overstatement. These are some of the issues faced:

1. The Smack Chat class has a close() method so I have to arrange to call it for every Chat session in progress.
2. The Smack Connection listener has to now check for connection in not connected state.
3. The Smack Connection class has to be able to remove Chat sessions both individually and collectively depending on context
4. Application state machine ended up more complicated than first envisaged because all events are posted to follow best practice and therefore care is required in order to get things to happen in the right sequence, which varies according to whether communications is up or done when a new login is requested.
5. The login dialog was originally designed to appear before the main window and now positions itself in the wrong place if launched once the application is running
6. The communication status indicator has to ignore communications down events during logout.

Now when the
ton on the Login
the user is now
application will



Cancel but-
dialog is hit,
warned the
shut down:

Status Progress

Starting with the principle that a Chat client is expected to show the user's online status and presence, a status line has been added to Cybertete and three indicators placed in it to show not only information about the user, but the communications and security status too. With Eclipse 4 there is no standard status line component ready to plug in and commandeering the Eclipse 3 workbench Status Line classes did not save time as I had to re-engineer them to achieve some modest goals like:

- resizing status bar when the main window resized and
- adding a popup context menu to the message area which was employed as a presence indicator.

The domain model grew by the addition of interfaces and events to delineate 3 state machines operating at different levels in the domain. At the lowest level is the Network state machine which drives the security indicator. At the next level is the Communications state machine which drives both the communications state and presence indicators. The Chat session views are also disabled while communications is down. At the highest level is the Application state machine which drives the logon-logout cycle.

Now it is time to consider what options the user of an instant messaging client might want.

Behind the Scenes

Before moving on to the topic of options I want to note that for the sake of keeping the narrative going I have not reported on code updates to fix bugs or improve code quality. There has been plenty of such activity so the final code you see is more polished than the work in progress. This is manifest in at least the following ways:

1. Each class fulfils a very specific purpose
2. Consistent usage of events, dependency injection and Eclipse framework features
3. Maintaining separation of concerns so code is clean and easy to comprehend
4. Handling errors gracefully.

Options

When modelling, one makes simplifying assumes to avoid getting bogged down with details. The original Cybertete design was for a planet where network failures do not occur and standards are applied uniformly so there is only one way to do anything. That does not sound like planet Earth! As development proceeds, more details are slipping in to the design to make it better adapted to real life. This involves not only the application itself, but the development environment as well. First there was a mock Chat service and client communicating using connectionless datagrams. The next step was to set up a Chat server running on a separate Linux box. The application that was selected was Openfire. It is easy to install and set up is straight forward except for subscriptions as a plugin needs to be installed to get automatic acceptance. With a fully functioning Chat server it is now possible to start adding and testing options to help users deal with real life.

Logon with JID and Password

Guidance on what options to consider comes from [XMPP IM Client Design Guidelines](#). The first guideline is %Do not to split up JIDs into multiple input fields, require only the user's JID and password+. This is an ideal option which assumes:

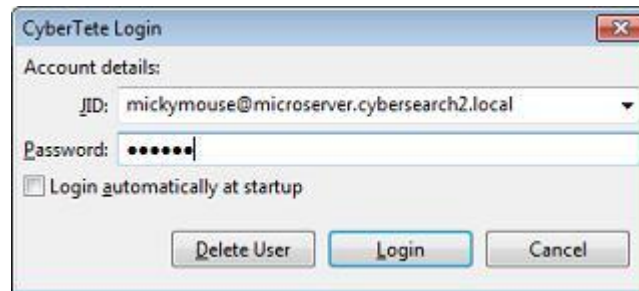
1. The local part of the JID corresponds to the authorization id
2. The domain part of the JID can be used to perform a DNS SRV record lookup to get connection details

I will change the logon dialog so it asks for only JID and password. I have already set up the local DNS server to provide an xmpp-client SRV record which is displayed as follows when I do a DNS lookup for SRV records:

```
_xmpp-client._tcp.cybersearch2.local  SRV service location:
    priority      = 0
    weight        = 0
    port          = 5222
    svr hostname   = microserver.cybersearch2.local
microserver.cybersearch2.local internet address = 192.168.3.21
```

You can see the record provides a hostname and port, which is very handy. Note that the Smack plugin needs [DNS Resolver components](#) and the [dnsjava](#) dependency for this to work.

I find I can login with just the JID and password. This is a snapshot of the new login dialog:



Now this is a good login dialog in that it does not confuse the user with options that, at least for new users, may be confusing. It asks for two pieces of information that any login requires- a unique ID and a password. So how to cater for the less-than-ideal situation where extra information is required such as host name and port? The answer I believe is to have a separate advanced login for the purpose of setting options. Once set, the above minimal login can be employed with the JID acting as a key to get any optional settings. As the main purpose of the advanced login is configuration, it makes sense to make it part of the main window rather than as a separate dialog. This is where Eclipse perspectives becomes useful.

Offline Perspective

To logon requires going offline, but if working in the main window then the normal views and the associated menu items need to be disabled or hidden. A new offline perspective would be helpful in getting this to work with minimal effort. The offline state can be indicated on the status line by introducing a new communication status of offline. Some of the Chat menu items such as %Close all Chats+ would need to be disabled, but the File -> Exit and all Help items can remain enabled.

I start with the Application Model Editor and add a new Offline perspective and nest inside it a parts stack and an %Advanced Login+part. The class for this part is initially just a simple form from an Eclipse 4 example just to have something to show it is there.

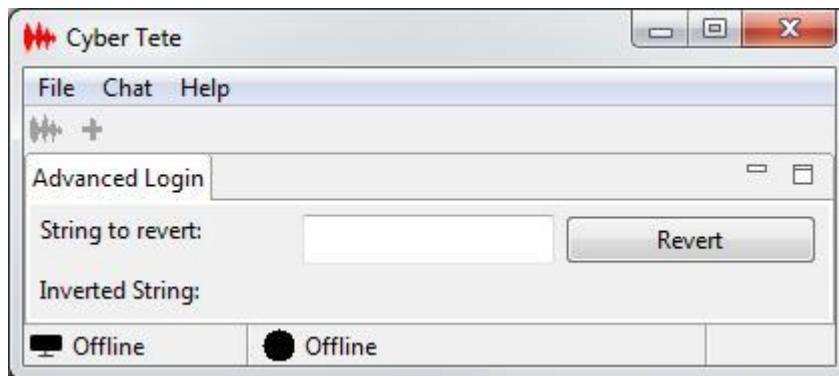
The tricky bit is how to switch to the new perspective. Firstly, the %Delete User+ button on the login dialog is moved to the Advanced Login view and replaced by an %Advanced+button, which causes the dialog to exit with a distinctive return code. Next a new method is added to LoginController interface to return a flag to indicate if Advanced button has been pressed. Finally, the E4LifeCycle class is reworked so the Offline perspective is selected if required. It turns out that the @ProcessAdditions annotated method is invoked too early to allow perspective switching, so an event handler performs this task. There is a Lifestyle event trigger on application start up to which the handler is subscribed.

The following extract shows how perspective switch takes place:

MPerspective perspective =

```
(MPerspective) modelService.find(OFFLINE_PERSPECTIVE, application);
partService.switchPerspective(perspective);
```

A test of the perspective switch in the handler works and I can now address the communications state indicator which needs a new offline state displayed as a black circle with text %Offline+. The presence indicator also need fixing as it shows %Online+by default. Here is the new perspective with the distinctive



status line indication:

Note that the presence indicator can still be right clicked to change the presence setting. This is okay because the Smack library allows the initial presence value to be set when establishing communications. This is something I need to implement.

Login Options

There are 3 options needed to overcome assumptions in use of JID to logon not being true plus one more related to authentication:

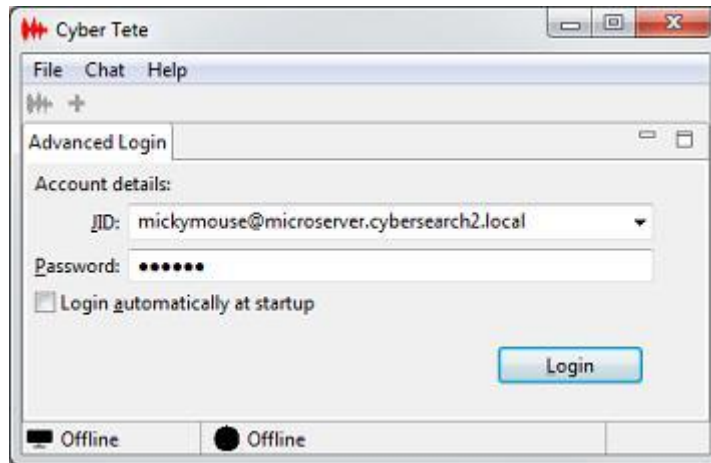
1. Host · there are some servers like Google where host would be "talk.google.com" and the service name would be "gmail.com"
2. Port · Smack only regards port setting if host is also specified
3. Authentication Id · From Guidelines % there are services which need the SASL authentication identity to be something else [than localpart of the users JID].+
4. PLAIN SASL Mechanism · From Guidelines %PLAIN] is still needed for servers who use a different hashing mechanism than SCRAM-SHA-1

To proceed, new properties need to be added to the SessionDetails class. An Advanced Login class needs to be created to render the content of same-named part. The existing Login dialog needs updating to show any options, if set. The option fields will not be editable, however. The user will need to click

on the Advanced button to update options. Finally, the Smack Chat service needs updating to be able to respond to the new options.

The first iteration of Advanced Login view, which only replicates the Login dialog except for no Cancel button, looks as follows:

For the first cut I reuse most of the original dialog code by copying it. Now it is



time to focus on behaviour, as this has not been considered deeply up till now. The intention is to achieve graceful handling of errors.

Login error scenarios

Error entering JID and password

When entering a new account the user may type the local part of the JID incorrectly or type the wrong password. The SASL authentication fails with error `%not-authorized+`. The Smack library throws a `SASLErrorException` with message `%SASLError using PLAIN: not-authorized+`. The user should get an error popup with text "Password or account invalid". On OK clicked, the login dialog should reappear with JID field filled in and receiving the focus while password is blank.

The user may also type the domain part of the JID incorrectly. The Smack library throws a `ConnectionException` with a verbose message containing the cause `%UnknownHostException+`. The user should get an error popup with text "Chat server at <address> is in unknown domain". On OK clicked, the login dialog should reappear with both JID and password fields filled in and JID receiving the focus.

Chat server not online

The user should get an error popup with text "Connection request to Chat Server at <address> refused".

Response timeout

The user should get an error popup with text "Connection timed out . Maybe Chat server busy or firewall issue".

Outgoing port blocked by firewall

The user should get an error popup with text "Connection to Chat server blocked by firewall".

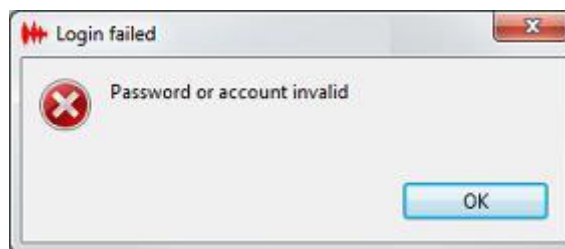
No trust certificate for SSL Handshake

The user should get an error popup with text "Missing trust certificate for SSL connection".

Other errors

It is impossible to anticipate all the errors that may occur, so the fallback will be to show the message of the exception thrown from the Smack library. This will be targeted at the service desk rather than the ordinary user.

To incorporate error handling into the Cybertete design, there needs to be a set of common errors and associated messages to display and provision for a



fallback for unanticipated errors. As well, the Login Controller implementation needs to show a popup error message box when login fails, with click OK to proceed. Then any special requirements for the Login view will need to be considered. Here is an error message box example:

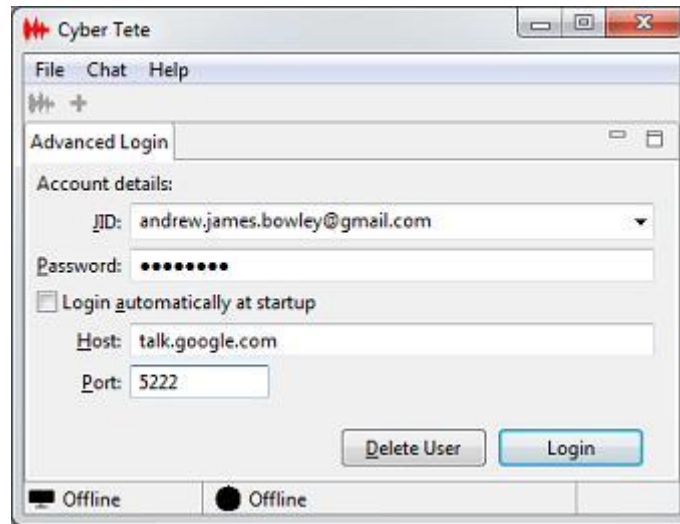
The Advanced Login view behaviour has to adapt that of the Login dialog to suit the context.

1. When the %login+button is pressed, the subsequent operation either fails or succeeds. There is no retry as with the Login dialog.
2. If the Login succeeds, then the perspective switches to the online default
3. If the Login fails, then an error message is popped up and the user is taken back to the Advance Login view on OK. If either the Host or Authentication ID fields are non-empty, then they may be targeted to receive focus depending on the type of error.

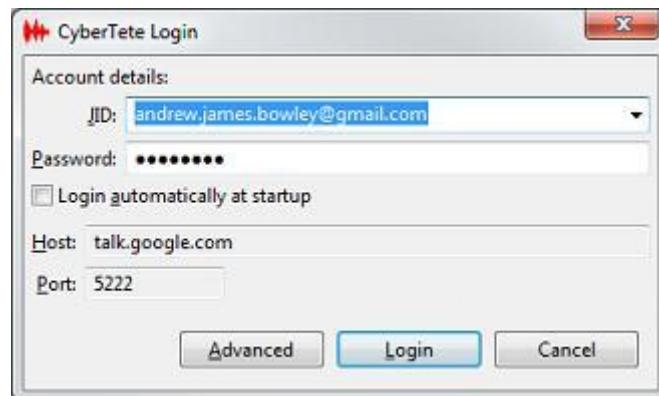
Login Controls

I fix some minor errors with the Login dialog and realize I have the same errors in the code which was replicated for the Login view. This motivates me to encapsulate all code for rendering the Login interaction screens into a single class. I name the class LoginControls. A flag passed in the constructor indicates whether a dialog or view is being rendered. This flag determines such things as which buttons to display and how to present options.

I add the host and port options and test them out by logging in to



talk.google.com host using my gmail account as the JID:



I get to see the %friendly+error handling in action. I get the password wrong first time and then I have a firewall issue. Eventually I succeed in logging in, so the Host option passes this test. Here is the Advanced Login view:

This is the next Login dialog, with the last user selected by default, and Host and Port displayed but not editable:

Having pioneered the first set of options, completing the rest should be straight forward. Also having worked with the Advanced Login view, I realize it needs

an **Apply** button which is enabled when a change to the details needs to be saved and a warning popup message when just about to complete a delete.

I add the username and plain SSL options and find both pass testing and make an interesting discovery along the way. The Smack Library supports a number of SASL mechanisms by default, including PLAIN. I found that when I blacklisted PLAIN mechanism, then I got a **No authentication mechanisms available** error when connecting to the Openfire server. Apparently support for SCRAM mechanism is only slated for a future release, so I will have to find another test application in order to test authentication completely.

I install Prosody on a virtual machine and find by default it uses SCRAM. Here is the sequence of xmpp-sasl packets from the log:

```
SENT (0): <auth mechanism='SCRAM-SHA-1'>...</auth>
RECV (0): <challenge>...</challenge>
SENT (0): <response>...</response>
RECV (0): <success>...</success>
```

I find if I do not set the password, I get a packet response timeout . This demonstrates the importance of testing on a real server. I discover three issues :

1. Validate password for sensible minimum length
2. Add packet response timeout to set of anticipated errors
3. Prevent automatic reconnection during login.

Having addressed these issues and many more of a minor nature as testing proceeds, the original LoginControls design has now been refined so there is a better separation of concerns. The management and persistence of the login configurations is now handled by a LoginData class which is now part of the domain model.

Options Advancing

Cybertete is now better equipped to deal with the real world marked by compromise and downtime. A user can now log in using just Jabber ID (JID) and password. If something else is required, then advanced options can be selected, bringing the user into the main application window in a new offline perspective. Here the user can make changes, including deleting accounts that are no longer needed. Any errors are displayed in a popup message box and anticipated errors are given user-friendly description.

So what else is there about the real world that Cybertete not paying attention to? The answer is the need for security. The current security measures of supporting TLS and SCRAM authentication mechanisms merely scratches the surface of what is possible. Now to look at such things as client certificates and Single Sign On.

Security

An application needs to integrate with the environment in which it runs. Eclipse and Java together hide most of the issues regarding running on different platforms but security is one area where the application design has to support various technologies. I will be looking first at Single Sign On as it opens the opportunity for Cybertete to be used just about anywhere. Any organisation using a number of applications on the desktop does not want employees having to repeatedly sign on.

Single Sign On with GSSAPI

The Generic Security Service Application Program Interface (GSSAPI) is an application programming interface for programs to access security services. Kerberos is in most cases employed as the security mechanism and Java natively supports GSSAPI with Kerberos. My goal is to add a %Single Sign On+ option to Cybertete and when it is selected, have the application automatically login without user intervention.

Getting set up for XMPP SSO proved to be challenging and it took several days to get all pieces in place. Working on a LAN with a Windows Server 2008 domain controller and Kerberos Key Distribution Centre (KDC), both client and server need to be joined to the local domain (cybersearch2.local). Both XMPP Servers I am currently using are not joined to the domain, so I proceed with installing Openfire on a Windows box (Windows 10, but no issues encountered).

Tips on Installing Openfire SSO

Shortly into the Openfire installation, it crashed because of a browser compatibility issue (I was using Firefox) I was forced to switch to Internet Explorer. Instead of selecting the Embedded Database option I chose an external database one, in this case I selected MS SQL Server. I would recommend the external database option as there was more than occasion where I had to edit the database to fix a configuration issue which prevented me from logging into the administrator console. In the installation I also hooked Openfire to the Active Directory LDAP server which automatically configures the SSO users.

I set up Openfire to run as a Windows service using a newly created %Openfire+account with delegation set to %Trust this user for delegation for specified services only+with the xmpp service account set up for Kerberos being specified. I did not set up krb5.ini on either client or server and instead used command line properties specifying realm and KDC. The service Java VM options were placed in a %openfire-service.vmoptions+file in the Openfire bin directory.

To get Openfire to start reliably, I had to make the Windows service a dependency of MS SQL Server, which in turn has to be changed to start up type %Automatic (Delayed Start).

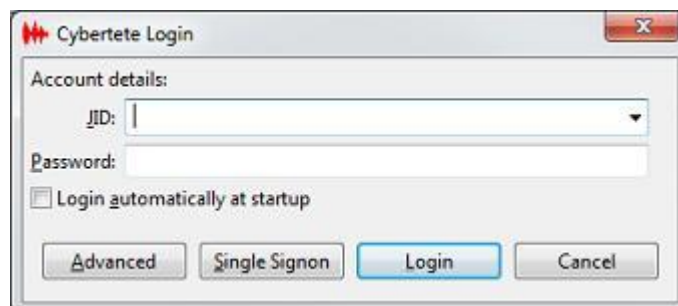
Single Sign On Design

The Smack library supports GSSAPI authentication mechanism using the Java GSS implementation. A drawback of this approach is that Java GSSAPI depends on an external login configuration file to provide important parameters. This leads to a number of scenarios such as a login configuration already exists but does not have an entry required by Java GSSAPI. The parameters are mostly concerned about how to obtain the credentials for authentication and among the options is the non-SSO approach of forcing the user to enter username and password. Another design consideration is that XMPP uses SASL GSSAPI for authentication which binds a client and server and requires the fully qualified domain name (fqdn) of the server to be provided.

The existing Login arrangements will need to change to accommodate SSO. The aim is to hide the complexities of SSO from the user and gracefully handle the many variations on how the environment is set up. Here are some behaviours that I wanted to incorporate into the design:

1. SSO is only available if a valid Java GSS client login configuration is present.
2. With SSO, the user only has to deal with a single JID entry field
3. The local part of the JID is filled in from the Kerberos principal and the JID will be validated to check the user has not changed it
4. The user enters the host part of the JID and then hits Login button to proceed.
5. If the `%doNotPrompt+login` module option is set to permit user interaction, then popup dialogs will collect username and password (**Not** SSO, but may be required for some workaround of a Kerberos issue).
6. If there is no login configuration file set in System properties, then one will be created for `%Cached Kerberos Ticket+credentials`. If there is such a file, but does not have a Java GSS entry, then one will be appended. Note in the latter case, file permissions may prevent the file being updated.

This is the initial Login dialog with a GSS login configuration file present and new installation:



Note new `%Single Signon` button. This is what happens when this button is clicked on my computer:



The username is obtained natively and any other value would be invalid. I just append the host fqdn and click %Login+button to complete login. Note that the username that is resolved by Kerberos need not be the currently logged in user if the login configuration has keytab and principal parameters commonly used on the server side, but this is not likely to be useful.

Finally, this is the login configuration that is created by default:

```
com.sun.security.jgss.initiate {
com.sun.security.auth.module.Krb5LoginModule
required
useTicketCache=true
doNotPrompt=true;
};
```

Client Certificate Authentication

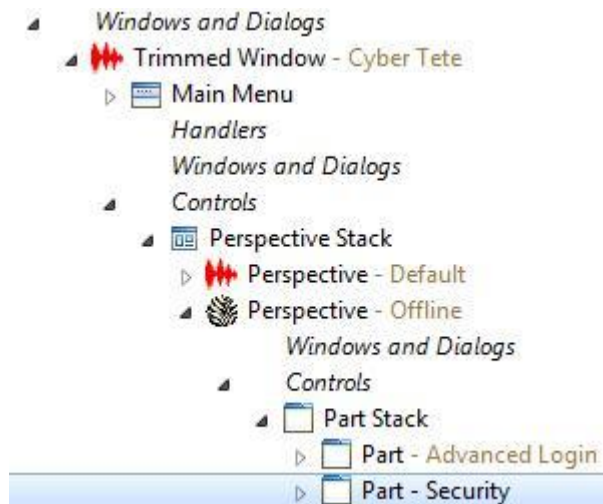
Client certificates may be used together with passwords to create 2-factor security. The burden of creating and maintaining a public key infrastructure (PKI) prevents the use of client certificates being widely used, but they do have a presence in private networks. Java natively supports PKI with its Java Secure Socket Extension (JSSE) library, which can be replaced with a number of offerings such as [Bouncy Castle](#) and [OpenSSL](#). Adding support for Client Cert Auth does not involve a lot of work as long as the application only has to access the certificate from a keystore file in a JSSE-compatible format. Then Cybertete only needs 3 pieces of information:

Field	Applies to
KeystorePath	Where certificates are stored
KeystoreType	How certificates are stored
Keystore password	How to secure certificates

As this is an obscure feature, it makes sense to tuck it away in a new Security view stacked beside the Advanced Login view. This new view can include other odds and ends such as setting minimum password length, which is currently hard coded. The KeystorePath field should have a %Browse+button which brings up a system file selection dialog.

Used on its own, Client Cert Auth is the same as GSSAPI in that the %Single Signon+button has to be clicked to avoid having to enter a password. With 2-factor authentication, unless the other factor is GSSAPI, the normal %Login+button will be clicked after entering the password.

Adding a %Security+tab to the Advanced Login view is achieved by introducing a Part Stack to the Offline perspective:



One more checkbox is added to the Security view to control whether Client Cert Auth is enabled. The Keystore fields are disabled when the feature is not enabled. Here is how the view now looks:



The context menu of the Communications State indicator is changed to allow the Client Certificate and its trust chain to be examined similar to the way the Server Certificate chain can be examined:



The Security Status indicator tool tip is also changed to show the SSL protocol and cipher suite when a secure connection is established.



Security Summary

The focus shifted on how the application can operate in environments where security stands require going beyond password authentication. The first technology out of the starting gate was Single Sign On (SSO) with GSSAPI which employs tokens stored in a Key Distribution Centre to allow already authenticated users to gain access to services on the network. Development was performed using Windows Active Directory for authorization and Openfire XMPP server running as a Windows service. A strategy for SSO was developed so the user can log in pretty much as normal, but without entering a password. The setup was complicated by the need to have an external login configuration file with a number of options supported, including entering a password. The Smack Library proved it supported GSSAPI by default and worked with Openfire as soon as the Active Directory side of things was made to work.

Next up was Client Certificate Authentication, which is often selected when 2-factor authentication is required. An external file was again a component, this time a keystore to hold the certificate. The keystore configuration was placed on a new Security view stacked beside the Advanced Login view. Again the Smack Library supported Client Cert Auth by default and achieved a successful login as soon as the Openfire server configuration was set up correctly.

Now maybe a good time to stop adding features and work towards the first production release. Time to apply some polish so Cybertete emerges all shiny and ready for bug-free deployment.

Testing

Cybertete is a first-time Eclipse E4 RCP development effort, so I left writing test code until the design has stabilised and only organic growth is expected from now on. Regression tests of course help to ensure any future changes do not break the design, but writing them is also a chance to review the code and add comments. Some test code was produced for the Chat Service prototype and from this I found that there are various schools of thought on how to test RCP applications. Creating a [plugin fragment](#), as is often recommended, has one drawback in that the fragment is not visible to the code being tested. To use a test Chat Service, I had to use the approach of adding tests in a separate source folder within the Cybertete plug-in project.

Eclipse RCP unit testing

I find a blog site covering the subject [Unit testing Eclipse RCP applications](#) which provides an important clue on how to get Maven to compile and run unit tests and after adding JUnit and [Mockito](#) test scope dependencies, I get 2 unit tests for the Chat Service prototype to run in both Maven and Eclipse.

Maven test summary:

```
[INFO] --- maven-surefire-plugin:2.19.1:test (test)
@ au.com.cybersearch2.cybertete

-----
T E S T S
-----

Running au.com.cybersearch2.cybertete.model.service.internal.ChatServiceTest
..
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
Running au.com.cybersearch2.cybertete.model.service.internal.DatagramTest
...
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

Results :
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

Project changes

1. Add a new source folder for tests, src-test
2. Set the src-test output folder to %target/test-classes+
3. Exclude this in build.properties by adding src.excludes = src-test/

Now for an annoyance with the Tycho Maven plugin. This plugin manages dependencies for runtime, but not testing. Adding test dependencies to the Maven configuration works for tests run from Maven, but are not automatically

added to the Eclipse classpath. Therefore, the Eclipse classpath must be configured for testing in the IDE. JUnit is conveniently available as a library, but other dependencies need to be set as external jars. Using the M2_REPO variable avoids having to copy files. In the Java Build Path dialog, click on the Add Variable button, select M2_REPO and click the Extend to select jar.

Note that performing Maven Update project clears pre-existing classpath settings, so have a contingency to restore the test dependency settings in case this happens.

See the Cybertete pom.xml <build> and <dependencies> elements for the Maven test configuration details.

Testability

Testability is a quality which impacts maintainability. Lack of attention to testability leads to customer dissatisfaction with a product support - bug fixes and upgrades take too long and introduce new bugs. Good code development practices tend to foster testability, but sometimes there are conflicts. For example hiding inner workings of classes by using private scope can impede necessary access required to test code.

Characteristics of good design include classes that are loosely coupled to each other, but function cohesively and are easy to understand. Class names, like variable and method names, should be self-evident so copious comments are unnecessary. Large classes are candidates for review to see if they can be shrunk or split up to reduce complexity and increase focus. All these characteristics support testability.

Unfortunately, the developer has to contend with third-party software, which has the potential to provide compatibility issues with test tools and approaches. Third party software cannot be changed, so work-around strategies are required and are the next topic to be covered.

Strategies

A concerted effort on testing will see further changes being made to the code. For example, private field declarations changed to allow package access. This reduces data-hiding for the sake of testability. More significant is overcoming the inability to use mock objects. Reasons include:

1. Class to be mocked is final
2. Class has runtime dependencies that the test environment cannot provide
3. Code under test does not allow for object replacement eg. construction of a local variable.
4. Code uses static methods or dependencies.

These items can be overcome for application code by redesigning it, but third party libraries and system constraints can make this a challenging process. For example, the Smack Library has Presence class and RosterEntry classes which are final and therefore not capable of being mocked. RosterEntry is a dependency of several classes, If used in unit testing. it will throw exceptions due to lack of a working connection to a Chat server. I will work around this problem by creating a mockable RosterEntry proxy to hide actual RosterEntry instances. This I call the %quarantine+strategy. Note that there is a tool called Powermock which purports to allow mocking of final classes, but I prefer stick to Mockito, which otherwise has not been found to have any other limitations.

Graphic library classes are often not suitable for mocking for one or more reasons given above. SWT widgets Combo, Text, Button and Label fall in this category, so will be quarantined. This strategy allows a high level of unit test coverage to be achieved even for code which renders the application forms.

There is one Eclipse E4 example of item 4 in the above list:

`MBasicFactory.INSTANCE`

This is an Application Model object factory. Fortunately, MBasicFactory is an interface, so can be mocked and inserted into objects being tested using dependency injection.

Code Coverage, Mocks and Assertions

A code coverage tool is also essential to monitor test effectiveness. I use [Eclemma](#). The actual percentage coverage to target is realistically 80%, but time may not be available to reach this. Because of the need to run Eclemma it is necessary to get unit tests going in Eclipse, rather than just use Maven.

Eclemma colors the source code under test to show the degree to which it has been tested. Green is tested, red is untested and yellow is partially tested. The visual coverage indication makes it easy to work out what test cases remain to be done. Achieving the desired level of coverage can be challenging and may required making changes for the sake of testability.

Mockito is used to create mock objects, which are readily set up according to test scenario. Mockito provides powerful test verification expressions to to check expected behaviour of the code under test. Complex code leads to large and complex tests, so is best avoided. Even before I start, I am aware of some classes that would benefit from reduction of complexity to improve testability.

[Fest assertions](#) is used to apply easy-to. read evaluation of test results, for example, testing for matches to expected values. Fest assertions include reporting on failure to throw an expected exception which helps validate error han-

ding code.

Case Studies

ChatLoginController

ChatLoginController is the first large class to be redesigned for testability. Its purpose is to control the process of the user performing login, with auto login allowed and potentially the user having to deal with errors or failures. After significant refactoring, in which several new classes were created, ChatLoginController turned into a small, easily tested class with minimal, high level control code. One of the important changes was to delegate background processing to handlers invoked using events. Code coverage achieved = 96%.

ChatWindowHandler

ChatWindowHandler updates Contacts and Chat Sessions windows when a new Chat session is started or all Chat sessions are closed. The class contains dense logic to navigate the many possible window state combinations. The number of test cases to be written seemed daunting. To reduce code complexity, a new class was spun off called AsyncChatWindowHandler to perform tasks required to run in the main thread. What remains in the original class is easier to comprehend and the number of required test cases is manageable.

Another change was to inject the Application part factory rather than call static

`MBasicFactory.INSTANCE`

The unit test could then return mock objects for factory calls. Mocking the MBasicFactory class also caused class not found errors at runtime due to missing Eclipse EMF dependencies. To fix this was a three-step process:

1. Add Eclipse EMF dependencies to the Maven configuration
2. Run Maven `%test+goal` to get the dependencies loaded to the local repository
3. Use Java Classpath `%Add Variable+option` to add the dependency jars now installed in the local repository (selecting M2_REPO variable).

The final coverage value for ChatWindowHandler is close to 99%.

LoginControls

LoginControls creates the controls shared by both Login Dialog and Login View. This class also handles the dynamics of user interactions at login, including changing user selection and saving the configuration on successful login. The initial design had an unacceptably large control creation method. As well, the code to create SWT controls was not amenable to mocking, and this had to be delegated to a factory class. Finally, two of the login buttons, %Login+ and %Single Signon+required processing on a background thread and therefore best delegated to event handlers.

Changes:

1. A **ControlFactory** class now creates SWT Composite, Text, Combo, Button and Label proxies and the ButtonBar object used in the Login View. ControlFactory is mockable and is an essential tool for unit testing classes which create and manipulate graphical widgets.
2. New event-driven handlers created for saving login configuration and getting single signon configuration. These handlers callback to LoginControls using events to update the display upon processing completion.
3. A new **AccountSelectionHandler** interface created to identify the methods need to handle change of user selection.
4. LoginControls class now delegates most of the work to a base class. It now only manages the creation of the controls and implements the new AccountSelectionHandler interface.
5. LoginControlsBase is comparatively large class but easy to unit test as it now focuses solely on the creation and dynamic changes of Login controls.

The classes which extend LoginControls, DialogLoginControls and ViewLoginControls each now have a helper to hide their dependencies and enhance testability. LoginControls and associated base class code coverage is nearly 97%.

Testing Summary

With just 7 unit test classes, I am getting over 23% code coverage, with the greatest contribution from the package with the .dialogs suffix. This is a great start and indicates an 80% coverage target is achievable with the current strategies.

A focus on testing inevitably leads to design changes as obstacles are recognized and removed. Some changes are simple, such as removing private ac-

cess constraints, and others are significant, such as replacing classes not suitable for mocking with proxies. The process leads to a better overall design as classes end up focused and with reduced complexity on how they depend on each other and third-party libraries.

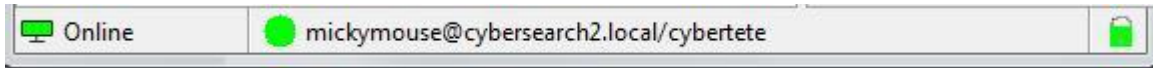
Eclipse E4 supports testability by providing an effective dependency injection system. This allows easy insertion of mock objects during testing. The number of application objects placed in the Eclipse Context at the start of the application life increases as testing development proceeds and includes Eclipse global MBasicFactory Application Model parts factory.

The development cycle under Maven control now includes a regression test phase to help catch bugs before code is pushed out into production.

The next activity is a by-product of refactoring for testability, and that is code reuse.

Status Bar Plugin

Recalling progress towards adding a status line to the application, I got into trouble trying to add the security indicator aligned to the left of the main window and had to do some customisation of the original Hyperbola design. This was the result:



The status line has three items which fill the bottom of the main window. When the main window is resized to be wider, the central item expands to take all available free space. The original design used the Eclipse E3 workbench Contribution Manager which organizes contributions from plugins to such UI components as menus, toolbars and status lines. The final design just borrows a few bits of the Contribution Manager and is fully adapted for E4 RCP. So why not export the Status Line support to a plugin so it is generally available to E4 RCP developers?

Model Fragment

Jonas Helming describes how add E4 features using fragments as follows:

A model fragment is a small application model in itself and defines elements which need to be added to the root application model. Fragments can add anything that can be part of the application model, for example handlers, menu items or even windows.

In the case of contributing a status line, the fragment needs to add a Tool Control child to a Window Trim component positioned at the bottom of the Main Window. The graphical components of the status line cannot be created before Tool Control is activated. Therefore the current design will need tweaking to allow the Tool Control to construct the status line using objects provided by the application. A new `IStatusBar` interface facilitates this:

```
public interface IStatusBar
{
    StatusLine getStatusLine(Composite parent);
    List<StatusLineContribution> getContributions();
}
```

Note that this interface requires a status line to be contributed by the application, but allows any number of additional items including none.

I spend time researching model fragments and end up with a working Status Bar plugin that anyone can import to their E4 RCP application. One point of confusion for newbies is that there exists a type of plugin called `Model Fragment plugin` which is offered as a way of implementing Eclipse unit tests. Therefore

I have to avoid referring to my new plugin as the `%fragment` plugin+even though it contains a model fragment.

As the new plugin contains a single Tool Control, it made sense to firstly develop a plugin which implements the Tool Control class and then use the Model Editor %Extract a fragment+feature to complete the required changes. This is how the fragment now looks in the Model Editor:



The fragment is in file *fragment.e4.xmi* which was created by the Model Editor upon extraction of the Tool Control from the Cybertete *Application.e4xmi*. The Status Bar plugin.xml file, previously empty, was updated with an org.eclipse.e4.workbench.model extension point to add model capability. The application and fragment models are linked by the id of the application element to which the fragment is attached. In this case, the id is `%au.com.cybersearch2.cybertete.trimbar.1`, which is guaranteed be unique but obviously not the best choice except for Cybertete.

Plugin Code

The migration of classes from application to library module has a few challenges. The first is in creation of the new plugin, where plugin dependencies need to be declared. I am not aware of any easy way to map classes to plugins and I opted out this time by copying the plugin list contained in the OP-COACH Preferences plugin used by Cybertete, which I correctly judged would be sufficient.

Changing the package of the classes moved to the plugin meant having to change access from protected to public in some cases, but this was trivial. Requiring more thought was how to remove dependency on a factory class created for testability. This is **ControlFactory**, which creates the graphical objects with which the status line is rendered. The controls used to construct the Status Bar will most likely overlap those in the application which may, or may not, use a factory class. The plugin code therefore declares a factory interface and provides an implementation available for use in the application. However, Cybertete uses its own factory class and implements the **StatusBarControlFactory** interface to interact with the new plugin.

Another consideration is whether to use Tycho to build the new plugin with Maven, but this seemed overkill for such a small body of code. I am porting the Status Bar unit tests, but they will not be automated by Maven.

Code Reuse Potential

Creation of a small plugin to add a feature to the main window of an Eclipse E4 application turned out to be doable and I will look for opportunities to spin off more plugins. However, resolving dependencies so a portion of code can work in a new context is a limiting factor. I will be refactoring the code to reduce the cross-section of code exposed to plugins - for example creating a local roster that has no direct Smack plugin dependencies. Also, any class that is not tied specifically to Cybertete functionality will be placed in a package without a %cybertete+component. One such package that already exists is au.com.cybersearch2.classylog, which implements a Java logger to complement the logger used by Smack..

Getting Ready for Release

Before Cybertete is released to the eagerly waiting public for the first time, there are a couple of additional topics to cover.

Findbugs

The first thing is to run a static analysis tool over the code to uncover bugs and questionable code. For this I employ the venerable Findbugs from the University of Maryland. It integrated seamlessly into Eclipse and reported a number of issues, but fortunately the number was manageable. The most serious issue was a `NullPointerException` being thrown.

Another interesting find was the exit from the login dialog before the main application window was displayed. Findbugs did not like `System.exit()` being called. The problem is that to use the official Eclipse E4 exit method requires application start up to be completed, so the main window is rendered momentarily before the application exits. I realise that I have a quit confirmation dialog for a normal login Cancel button clicked, so I do the same for the splash screen login. The confirmation dialog appears with a newly rendered main window in the background— never before seen. The only problem is the Presence indicator displays `%Online+` instead of `%Offline+`, but this is fixed by sending a message to the Presence indicator on application start complete.

For a last round with Findbugs, I wind down `%Confidence+` from the default `%Medium+` to the lowest setting. This gives me another handful of issues to look at and I manage to clear them all without much effort. None of these were code faults, but just things that were inefficient or not good practice. I can now install the Jenkins plugin for Findbugs to incorporate static analysis into the production build.

Logging

Up till now, there has been no effort to capture exceptions or record information relevant to fault-finding. However, post-deployment, logging is vital for support. There are many Java logging technologies available, including the one packaged with Java itself and encapsulated in the `java.util.logging.Logger` class. As the Smack Library uses the Java logger, it makes sense to use the same logger for all application logging. One caveat is that the Smack library can be replaced by something else which has a different approach to logging. Therefore it is wise to take advantage of the provision Eclipse has made to plug in any logger of choice with its Log Service component.

The Log Service is accessed by grabbing an `ILoggerProvider` instance from the Eclipse Context. Here is a code snippet demonstrating how this is done:

```
@PostContextCreate
void postContextCreate(...
    ILoggerProvider loggerProvider)
{
    Logger logger = loggerProvider.getClassLogger
(E4LifeCycle.class);
    logger.info("In postContextCreate()");
}
```

Here is how the information appears in the log:

```
Dec 22, 2015 10:35:01 AM au.com.cybersearch2.cybertete.E4LifeCycle
INFO: In postContextCreate()
```

The ILoggerProvider in this case is set up in the application Activator class through service registration. The code is trivial and only one method is implemented:

```
Logger getClassLogger(Class<?> clazz)
```

The Logger which is returned is an abstract Eclipse class in package org.eclipse.e4.core.services.log. What needs to be done is to extend the Logger class so it wraps a logger of whatever technology is desired. I choose to the Cybersearch2 ClassyTools JavaLogger as it adapts the Oracle logger implementation to better align with contemporary loggers eg. use level terminology %debug+ and %verbose+ in place of %FINE+ and %FINEST+.

Up till now, every time the Java editor gives a choice to throw or catch an exception, I choose to catch it and let automatic code generation generate a stack trace printout. Now I have to find all these try-catch statements and arrange for an error message to be displayed and a entry to be written to a log.

What I appreciate is how easy it is to make small adjustments for error handling because of the loose coupling of classes. For example, I found there are three places where login configuration changes are persisted and it made sense to add a new event which signals the changes need to be saved. Adding the event definition, three posts and one handler was trivial. Another significant change was to unify handling of Smack exceptions between login and add contacts. There were a lot of small changes and creation of a new Exception to capture reporting information, but it all worked first time.

To be continued...

Testing and reviewing the Cybertete code is turning out to be a much larger task than anticipated. I want to take the opportunities provided by the E4 features further and in a consist approach. I am also keen to get code coverage up to a respectable level, which I see as 66% or higher and that requires a lot of work writing unit tests.

The trend is towards finer granularity, meaning smaller classes, more packages, more events and more use of dependency injection. I will eventually publish Cybertete on Github along with installation instructions. Setting up will require simply cloning the project to your workstation using Git and then running Maven install.

You will then be able to follow as I outline the final design and see the code in action for yourself. I am now a fan of Eclipse and can speak from experience that E4 is the way to go for developing well-engineered applications portable to a large number of platforms.

