A Model for IM and Media Driven Communication Services

Leigh Griffin [1], Eamonn de Leastar [1]

[1] Waterford Institute of Technology, Cork Road, Waterford Ireland
{lgriffin, edeleastar} @ tssg.org

Abstract

Constructing graphical client applications for the Instant Messaging (IM) domain can be complex and error prone. As well as coping with the details of a specific IM protocol, the developer must also master specific user interface toolkits, deal with media streaming protocols and codecs, handle capability negotiation and deploy a robust event handling strategy for this highly asynchronous application style. These often competing concerns must be encapsulated in a clean design that can be evolved to cater for an ever expanding set of capabilities now viable for IM client applications. This paper proposes an architecture, component set and pattern based framework to encapsulate this domain, which facilitates the rapid construction of rich media client IM applications. These may be conventional or specialised IM clients or add-on components for existing applications.

Keywords: Instant Messaging, Media, Service Creation, Components, Communications Services

1. Introduction

Modern software development is becoming profoundly multi-disciplinary. As well as core software engineering skills, the scale and complexity being addressed by many applications is such that there is increasing specialisation required by development teams. Web applications, rich graphical user interfaces, media processing, protocol handling, multithreading, database access, configuration, build and deployment are often considered separate disciplines, serviced by different individuals with specialised tools, skill sets and experiences. This trend is particularly true of communications applications, which carry an additional burden in that they must interface with a communications infrastructure, often highly demanding of the applications they interact with and intolerant of any misuse. This paper tackles one specific domain – Instant Messaging communications client applications – and proposes a model + pattern based framework to significantly ease the conceptual burden on the developer creating such clients. The framework is targeted at the general developer, one with intermediate level knowledge of the Java programming language and general familiarity with the Swing GUI framework. As such, it attempts to counter the trend of increasing specialisation by elegantly and efficiently encapsulating a complex domain, yielding a comprehensible model and a powerful component set.

The paper is divided into 9 sections. *Section 1* is this introduction. *Section 2: IM Clients* briefly reviews the nature of IM applications and emerging features of clients within this domain. *Section 3: Complexity* explores some of the difficulties with IM applications development, particularly the diverse technology sets the domain encompasses. *Section 4: Methodology* relates this work to the broader software engineering discipline, particularly to current software architecture, component and design pattern thinking. *Section 5: Architecture* presents the core domain model for IM applications as a UML model + a set of java interfaces. *Section 6: Components* explores how this model is encapsulated within a Graphical User Interface context. *Section 7: Patterns* discusses aspects of the framework design exposed to the developers. *Section 8: Applications* previews some clients that can be built using the framework. *Section 9: Implementation* reviews some implementation experiences.

2. IM Clients

Instant Messaging (IM) is a form of text based, near real-time communication between two or more users over a network. Simple IM clients provide the basic functionality for users to communicate in a one to one fashion while providing basic error checking and receipt acknowledgement. Richer IM clients retain the core text based communication but enhance the user experience by integrating group communication and richer media. Employing microphones and web cams, a quick, cheap and effective conference service can be created, allowing users to engage in real-time conversations. Adding to this already rich suite of services additional features have been aggregated, including conversation logging, file transfer, games, whiteboards, chat rooms and other new features, often added with each new release of a particular client.

IM usage has expanded in recent years to become an essential service for both business and social use. Potential applications for IM have emerged, evolving it from its roots in text based, presence aware communication while still retaining this core functionality. In recent surveys it was found that 70% of teenagers send more IM's then emails showing the surge in popularity of this means of communication [1]. Networked games have been integrated to some IM clients offering new types of experience [2]. IM Robots (or Bots) are software applications that run automated tasks within an IM context [3]. By adding a Bot buddy to a buddy list a user can interact with this buddy which in turn may query a database, access an external system or perform some other task. RSS feed bots, customer support bots and other interactive bots are extending the range and application of the IM paradigm considerably beyond its original conception. Many IM clients now incorporate full multimedia capabilities, both voice and video [4][5]. In this configuration the IM client can be the primary communications end point for a user, taking on the role of a conventional telephone, mobile and video conference facility.

3. Complexity

An alphabet soup of protocols, APIs, components and toolkits are required to build even the most conventional IM client application. IP, UDP, SIP, XMPP, XML, RPC, SOAP, RSS, RTP, JMF, SWING, SMACK, TCP are some of the protocols, SDKs and applications that must be mastered to produce a fully working IM client and service suite. This complexity is masked somewhat by the use of API's to allow the developer easier access to the lower stack calls that make up the building blocks of the protocols. However there is no singe API that encompass the full IM "stack" [6] and the developer must currently cope with a broad range of tools and APIs. Moreover, current APIs do not encapsulate media capabilities, requiring the developer to gain a comprehensive understanding of the intricacies of IP, TCP, UDP and RTP, including the fine details of session set up, capability negotiation and quality of service issues.

Added to this already steep learning curve is competing IM infrastructures. Google Talk, MSN, AIM and XMPP are leaders in the IM field. Each boasts their own infrastructure which does not easily allow users on one client to avail of the services offered by another. XMPP is the most open of these standards (it has been adopted in modified form by Google Talk). However, even it imposes a significant complexity burden, even for relatively straightforward activities. An entirely XML based protocol – the schema has grown considerably since its first publication [7]. Similarly, media has a diverse range of standards, codecs and device capabilities that must be mastered. AVI, DivX, MPEG, WMV, WMA and MP3 are some of the better known formats. These formats are standardised and documented, but applications and services have to know how to handle and render such formats on client devices. This is accomplished using codecs. A codec is a device or program that is capable of encoding and decoding a digital data stream. Each codec comes with its own performance overhead, be it quality or compression size. This adds an extra layer of complexity to the process of handling media which the programmer must currently take into account while trying to integrate media into an application.

4. Methodology

Designing a coherent framework to support the creation of Instant Messaging client applications and services is a significant challenge. It requires a detailed grasp of both a complex and continuously evolving problem domain and the employment of a conceptual toolkit to master this complexity. A number of design decisions taken early in the development will have a profound impact on the structure and usefulness of the framework, component set and services. This work takes advantage of multiple paradigms currently in software development methodologies to inform key architectural decisions. To guide these decisions, we acknowledge the structural hierarchy of paradigms adapted from [8] (fig 1) and deploy specific approaches at a number of levels in this hierarchy. Specifically, key decisions at the Architecture, Component and Design Pattern levels.

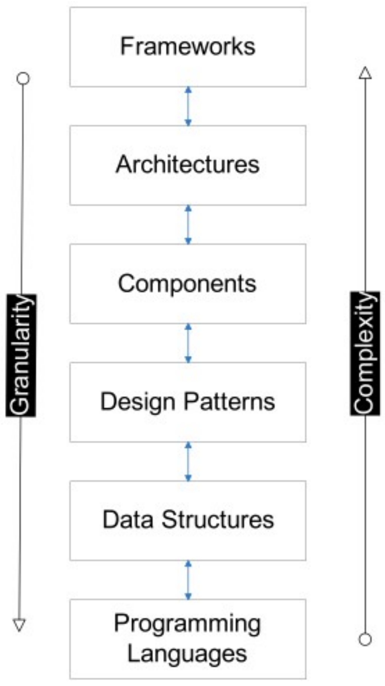


Figure 1: Software Paradigms

At the architecture level we apply Domain Driven Design (DDD) [9].This is a modern approach to architecting software systems that places strong emphasis on coherent and comprehensible domain models – drawn directly from the problem space. Sometimes regarded as a rediscovery of the benefits of modelling within an agile development methodology [9][10], DDD enumerates core categories of building blocks coupled with a “supple design” process to yield a highly declarative and “intention revealing” design. This is grounded by a “Ubiquitous Language” - a type of system wide data dictionary. Applying DDD principles to the IM domain yields a set of core classes and relationships that elegantly and expressively captures key abstractions on the domain, facilitating comprehensibly and consistent usage by client applications and services.

At the component level we apply Interface Oriented Design [11]. This is a set of useful practices for designing interfaces to capture the responsibilities of a component set. Although overlapping somewhat with DDD, Interface Oriented Design proposes more specific guidelines for representing a design as a set of related interface specifications within a programming language that supports the interface construct. When directly supported by a language, careful use of interfaces to encapsulate an architecture represents an ideal specification language for a model of a system. Interfaces are unambiguous, highly expressive, have a very low signal to noise ratio (when contrasted with classes for instance) and are directly consumable by client applications.

Applying principles from both DDD and Interface Oriented Design attractively marries architectural concerns with an unambiguous model specification language. We choose the Java programming language, which has a mature implementation of interfaces, coupled with useful modern programming language features such as annotations and generics. The challenge is to use these tools to render a flexible, comprehensible and powerful model for IM client application and service development. This model should be reasonably quick to explain to a developer and implementable across multiple IM technologies.

5. Architecture

The model (fig 2) was evolved using Domain Driven Design [9] principles to capture the core domain artifacts for the IM framework. This model provides the building blocks for IM service creation, delivering the key abstractions from which IM applications can be constructed. The model has two distinct subsets; the core domain and the event model. The domain model captures the core mechanics of an instant messaging client application: connecting to an IM server, initiating conversations, establishing media sessions and generally managing the buddy list and connection settings. The event model binds these artifacts together, delivering an elegant model for presence updates, buddy initiated conversations and messages and general management of asynchronous event information within the domain.

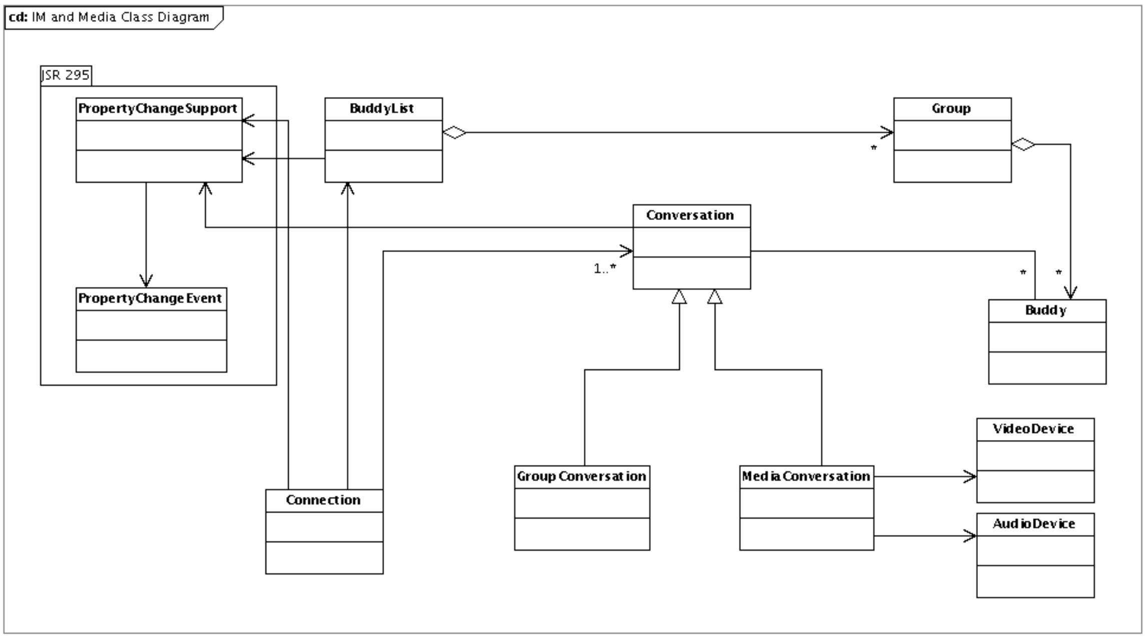


Figure 2: Domain and Event Class diagram

Adhering to the principles of Interface Oriented Design [11] each of these domain classes is realised as a Java interface. The interface can be considered the most expressive representation of a domain artifact - relevant information is presented, without any implementation details. In fact we can review the full interface specifications here:

- Buddy: used to represent a single contact that the user may wish to communicate with.

```
interface IBuddy
{
  String getName();
  String getStatus();
  String getUserID();
}
```

- Group: In Instant messaging applications it is common practice to assign contacts to groups for easier management and for quick reference. Common groups such as family, work and friends are mainstays on most instant messaging applications.

```
interface IGroup
{
  String getName();
  void remove(IBuddy Buddy);
  void add(String buddyID);
  List<IBuddy> getBuddies();
}
```

- BuddyList: A buddy list is a collection of groups. Like most subsequent interfaces, The BuddyList can be updated asynchronously, hence the PropertyChangeSupport accessor (discussed below).

```
interface IBuddyList
{
  void add(String buddyName, String ID);
  IBuddy get(String buddyName);
  List<IGroup> getGroups();
  PropertyChangeSupport getPCS();
}
```

- Connection: This interface is responsible for dealing directly with the messaging server and negotiating a client-server session with it. It contains the key factory methods for the client application. Conversations, Conferences, Media based Conversations and Presence awareness are all supported in a short, clear and unambiguous specification.

```
interface IConnection
{
  void connect(String server, int port);
  IConversation createConversation(IBuddy buddy);
  IConference createConference(String roomName);
  IMediaConversation createMediaConversation
                   (IBuddy buddy,
                    IAudioDevice audio,
                    IVideoDevice video);
  void authenticate(String username, String password)
  void setPresence(PresenceType presence,
                   String message)
  IBuddyList getBuddyList();
…
}
```

- Conversation: This interface represents a one to one conversation between the user and a buddy. It is responsible for creating, managing and terminating the conversation.

```
interface Conversation
{
  IBuddy getBuddy();
  void sendMessage(String message);
  PropertyChangeSupport getPCS();
}
```

- GroupConversation: A form of conversation except this class handles multiple conversations within the one session. This class is responsible for creating and managing a Multi User Chat, creating the room, the access rights and populating the room with the requesting owner and sending invites to any party who wishes to partake in the conversation.

```
interface IGroupConversation extends IConversation
{
  void inviteUser(ImpBuddy buddy, String message);
  BuddyList getParticipants();
  PropertyChangeSupport getPCS();
}
```

- MediaConversation: A form of conversation that adds a media element to the already existing textual conversation. This class looks after the establishment of a connection between the two clients who wish to participate in a Media based conversation.

```
interface IMediaConversation extends IConversation
{
  IVideoDevice getVideoDevice();
  IAudioDevice getAudioDevice();
  PropertyChangeSupport getPCS();
}
```

- VideoDevice: This interface is used to control the video device used in a Media Conversation. It provides the means for basic playback functionality and allows the user have full control over the video device that is currently connected and in use. If multiple devices are available the interface provides a provision for choosing which device to use as the primary device.

```
interface IVideoDevice
{
  void pause();
  void play();
  void play(String file);
  void record(String filename);
  void chooseVideoDevice(int DeviceNumber);
}
```

- AudioDevice: Provides a means to control the audio device used during a Media Conversation. Several audio devices can be connected to a single piece of hardware and this interface allows different audio devices to be selected as the currently used device.

```
public interface AudioDevice
{
enum VolumeLevel {off, low, med, high, max}
void setVolume(VolumeLevel level)
void mute();
void record(String filename);
void play(String filename);
void chooseAudioDevice(int DeviceNumber)
}
```

In any application exhibiting highly asynchronous behaviour, an appropriate event model is of central importance. Failure to devise an application wide strategy for event handling will quickly lead to ad-hoc approaches among different subsystems and ultimately a degraded and over complex design. As the work in this paper is focussed in IM client development, an event model from the JDK was adopted directly into the model. This is the Java Specification Request (JSR) 295 – Beans Binding implementation [12]. This specification has been incorporated into the JDK relatively recently defining (among others) PropertyChangeSupport and ProperteChangeEvent classes, and a PropertyChangeListener interface. To be a source of events, a class can create a PropertyChangeSupport instance and feed events to it. To consume events, a class can implement a PropertyChangeListener interface and receive PropertyChangeEvent objects through an implemented method on that interface. The simplicity of this approach belies some important benefits: events can be tagged, with listeners only listening for specific categories; events can be routed into a specific thread – very important for GUI applications which usually require UI updates on a specific thread (the Event Dispatch Thread), and finally the GUI designer directly understands this mechanism and can "wire" components together visually whilst generating the appropriate code based on JSR 295 artefacts.

6. Components

This domain model serves as the core communication model for the framework. It is as an elegantly encapsulated foundation for the next layer – the GUI components that render IM views within appropriate contexts. When approaching the development of GUI components, tool support is particularly important. The Netbeans IDE [13] incorporates a breakthrough GUI design tool (previously called project Matisse [14]). This facilitates rapid visual prototype and creation of GUI's. It comes with built-in support for JSR-296 (Swing Application Framework) [15] as well as JSR-295. Layout designs and visual forms have traditionally been a stumbling block for java developers, with considerable expertise required to hand-code convincing user interfaces. Netbeans design tool dramatically simplifies this task with a high quality visual designer. Precision placing of components is now carried out by moving the component with the mouse to the desired location and adjusting behaviour and alignment via intelligent guides that cue at appropriate locations during the design process.

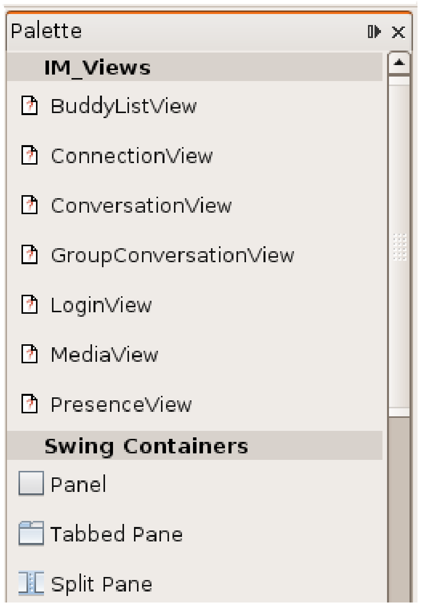


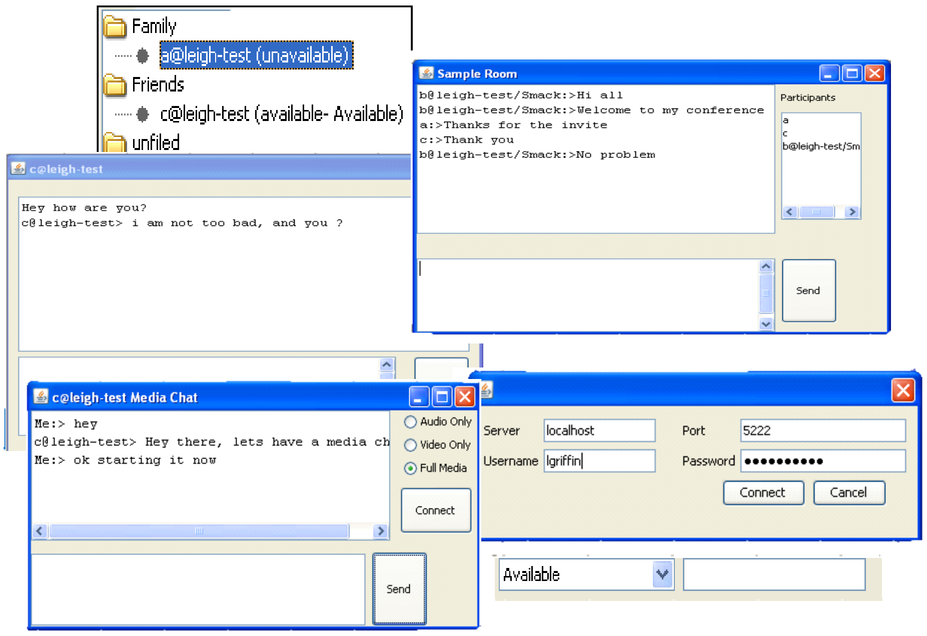
Fig 3: Netbeans IM_Views Palette

In Netbeans GUI designer the palette is a standard holding area for common components used in creating a standard GUI. It allows the user to select components from a set of default Swing components and drag and drop them onto the canvas and thus create an effective user interface. An important capability of the designer is the ability to extend the palette with custom components. Appropriately packaged, the custom components can be visually assembled into client applications as it they were standard swing components. A set of standard components, bound to the domain model, can be defined and loaded onto a Netbeans palette. These components can provide stock set view of domain artefacts, facilitating extremely rapid IM applications construction. Among the views created are:

- BuddyListView: A tree structure showing a users buddy list. The ability to add users, remove users and a group management feature were all provided by this visual component.

- ConnectionView: Manages the user profile settings – server name, authentication details etc..

- ConversationView: This view was used to house a one to one text based conversation. In the lower pane the user can type messages and send them to the buddy currently in the conversation. A copy of the message is sent to the upper pane. Incoming messages from the buddy are displayed also in this pane allowing a user see a full history of the current chat as it is occurring.



```
Hey how are you?
c@leigh-test> i am not too bad, and you ?
```

```
b@leigh-test/Smack:>Hi all
b@leigh-test/Smack:>Welcome to my conference
a:>Thanks for the invite
c:>Thank you
b@leigh-test/Smack:>No problem
```

Participants: a, c, b@leigh-test/Sm

```
Me:> hey
c@leigh-test> Hey there, lets have a media ch
Me:> ok starting it now
```

Audio Only / Video Only / [✓ Full Media]

Server	localhost	Port	5222
Username	lgriffin	Password	●●●●●●●●●●

Available

Figure 3: Selected Views

- GroupConversationView: This view is created to house the multi user chat feature of the IM client. A lower and upper pane similar to the conversation view's panes are utilised to keep a history of the conversation. A panel to the upper right hand side is used to keep a track of the current participants in the chat room.

- MediaView: The media view is used to house a one to one media based conversation with another user of the application. Buttons are provided to choose between Audio Only, Video only and Full Media Conversation. Panes similar to the conversation pane are utilised to hold a simultaneous text based chat conversation.

- LoginView: This view provides the means for the user to enter their credentials. A username and password field is provided as well as information about the server that they wish to connect to.

- PresenceView: This view allows the user to dynamically change their online presence so that other IM users can see if they are available, away, busy or a custom presence message.

7. Patterns

Realising graphical user interface applications can be a particularly complex design and development activity. GUI components are difficult to configure, contain multiple event interfaces and often embody complex interdependencies that will quickly overwhelm the cleanest design. Separation of concerns is paramount, but choosing which concerns to isolate and how the isolation is achieved can be taxing. A venerable design pattern – Model View Controller – has been a mainstay of GUI application design for many years and has been partly or fully incorporated into a range of GUI frameworks [16]. MVC is now most commonly encountered in Web application frameworks [17]. A recent review of GUI patterns – and a recasting of MVC into a modern GUI context [16] have yielded a refinement called Model View Presenter (MVP). This pattern has been adopted into the IM framework, specifically to relate the views (presented above) with the appropriate domain model classes discussed earlier.

In MVP the model is an artefact drawn from the domain – the IM model in our instance. The View is the GUI rendering classes - these are designed and configured to reside on the Netbeans GUI designer palette. The View is "passive" [18], which implies that there is no behaviour (event handlers, view synchronisation logic) in the view code at all. These views are thus very low maintenance, and can be totally maintained by a designer using a standard visual tool (Netbeans Visual designer in this case). The Presenter then has the task of binding the View to the Model – and in particular is tasked with ensuing that the View components are appropriately "armed" with the correct behaviour logic.

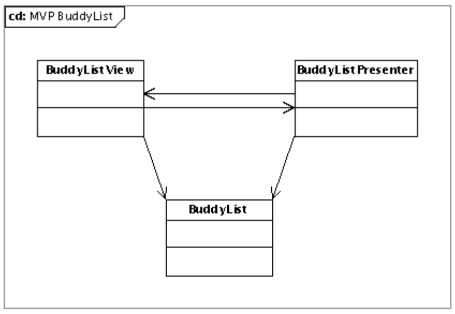


Figure 4: MVP BuddyList Triad

Taking the buddy list as an example, BuddyListView is a visual component, available on the palette, which can render a buddy list data structure. The BuddyListPresenter subscribes to a number of events in the View. When an event of interest takes place, the Presenter decides on a course of action and invokes methods on the model component. Changes are then made in the View to let the user know that their interaction has resulted in a change occurring. This triad is replicated for each of the views/domain model classes in the framework (Fig 3).

Applying MVP yields a comprehensive design context for the GUI components, enabling the rapid visual construction of IM client applications. An IM client application can be created by selecting/painting the appropriate visual components into a GUI application, and then customisation of the presenters to achieve specific behaviour as appropriate. The developer experience, assuming a general familiarity with MVP, is significantly simplified over creating a IM client directly using the model classes themselves. MVP, combined with the domain model, delivers a robust event model, appropriately configured and primed visual components enabling interesting and innovative IM client applications to be defined, tuned and tested within a high quality development environment.

8. Applications

The model and it's in built flexible component rendered within the Netbeans palette, facilitates the rapid creation and development of IM based service suites. This allows for several types of applications to be rapidly developed as derivatives of the standard IM model. These could be innovative, but conventional, IM applications, or they could be clients offering specific collaboration facilities layered on top of the IM infrastructure and model. Avatar based chat rooms, customer service bots, presence triggered meeting clients or interesting integrations into calendaring or email systems could be rapidly prototyped.

Besides more conventional applications, the components could be used in a collaboration or monitoring role. For instance a security camera monitoring application – configured as a set of IM clients – could be assembled rapidly. An application of this nature could be quickly put together and provide a cheap perhaps short term solution to a problem. Furthermore, the components could be deployed to augment an existing application with IM capabilities. For instance a Customer Relationship Management application could be modified to incorporate IM capabilities relatively easily, enabling presence, messaging and even media communications taking advantage of the additional contextual information available with the host application.

9. Implementation

During the development of the framework a set of open standards and components was selected as part of the initial implementation. These included Openfire [19], which is a real time XMPP based collaboration server, Smack [20] an open source java XMPP library, JMF [21] a framework for media communications in Java. These three technologies delivered a useful, open source and largely reliable test bed on which to evolve the components. Bridge [10] is the key design pattern deployed in the implementation. This enables the model to be cleanly decoupled from the implementation and thus alternative bindings developed as additional protocols, codecs and standards are tackled. The model developed in conjunction with Bridge thus allows the implementation to vary quite significantly depending on critical project factors such as time, money and resources.

10. Conclusion

This paper explored the viability of applying Domain Driven Design, Interface Oriented Design and the Model View Presenter design pattern to engineer an elegant yet powerful component set to support the development of Instant Messaging Graphical User Interface client applications. The components hide considerable complexity, yet enable interesting variations on an IM client to be rapidly constructed within a high quality Integrated Development Environment (Netbeans). These IM components become useful and interesting building blocks for a broad range of communications applications – either standalone variations on IM functionality, innovative new areas not necessarily considered within the remit of IM and also as a component set that can augment an existing application.

Acknowledgements:

This research activity is funded under the EU IST FP7 project, Autonomic
Internet (Grant agreement no.:216404). Some contributions to this paper
have been sponsored by Science Foundation Ireland (SFI) under grant number
03/CE3/I405.

References

[1] IM vs. Email with Different users. [online]. Available at: http://communication.howstuffworks.com/im-and-email2.htm [Accessed on 31-JUL-2008]

[2] MSN Games. [online]. Available at: http://zone.msn.com/en/general/article/generalmessengergames.htm [Accessed 31-JUL-2008]

[3] Zimbie. [online]. Available at: http://www.zimbie.com/ [Accessed on 1-JUL-2008]

[4] Google Talk. [online]. Available at: http://www.google.com/talk/ [Accessed on 15-JUL-2008]

[5] MSN Messenger. [online]. Available at: http://webmessenger.msn.com/ [Accessed on 15-JUL-2008]

[6] Smack. [online]. Available at: http://www.igniterealtime.org/projects/smack/index.jsp [Accessed on 1-JUL-2008]

[7] XMPP Schemas [online]. Available at: http://www.xmpp.org/schemas/ [Accessed on 31-JUL-2008]

[8] Kaisler, Stephen H., 2005. Software Paradigms. Great Britain: Wiley.

[9] Evans, Eric., 2004. Domain Driven Design: Tackling Complexity in the Heart of Software. United States of America: Addison Wesley.

[10] Holub, Allen., 2004. Learning Design Patterns by Looking at Code. United States of America: Apress.

[11] Pugh, Ken., 2006. Interface Orientated Design. United States of America: Pragmatic Bookshelf.

[12] JSR-295: Beans Binding. [online]. Available at: http://jcp.org/en/jsr/detail?id=295 [Accessed on 16-JUL-2008]

[13] Netbeans IDE [online]. Available at http://www.netbeans.org/ [Accessed 27-JUL-2008]

[14] Java GUI's and Project Matisse Learning Trail [online] http://www.netbeans.org/kb/articles/matisse.html [accessed on 29-JUL-2008]

[15] JSR-296: Swing Application Framework. [online]. Available at: http://jcp.org/en/jsr/detail?id=296 [Accessed on 16-JUL-2008]

[16] Fowler, Martin., 2006. GUI Architectures. [online] Available at: http://martinfowler.com/eaaDev/uiArchs.html [Accessed 02-JUL-2008]

[17] Spring Framework. [online]. Available at: http://www.springframework.org/ [Accessed on 19-JUL-2008]

[18] Fowler, Martin,. Passive View. [online]. Available at: http://martinfowler.com/eaaDev/PassiveScreen.html [Accessed on 30-JUL-2008]

[19] Openfire Server [online]. Available at: http://www.igniterealtime.org/projects/openfire/ [Accessed on 12-JUN-2008]

[20] Smack. [online]. Available at: http://www.igniterealtime.org/projects/smack/index.jsp [Accessed on 1-JUL-2008]

[21] Java Media Framework [online]. Available at: http://java.sun.com/products/java-media/jmf/ [Accessed on 10-JUN-2008]