Res. Lett. Inf. Math. Sci., 2005, Vol. **7**, pp 157-160
Available online at http://iims.massey.ac.nz/research/letters/

Implementing a map based simulator for the location API for J2ME

D. PARSONS

Institute of Information & Mathematical Sciences
Massey University at Albany, Auckland, New Zealand

The Java Location API for J2ME™ integrates generic positioning and orientation data with persistent storage of landmark objects. It can be used to develop location based service applications for small mobile devices, and these applications can be tested using simulation environments. Currently the only simulation tools in the public domain are proprietary mobile device simulators that are driven by GPS data log files, but it is sometimes useful to be able to test location based services using interactive map-based tools. In addition, we may need to experiment with extensions and changes to the standard API to support additional services, requiring an open source environment. In this paper we describe the implementation of an open source map-based simulation tool compatible with other commonly used development and deployment tools.

1 Introduction

With the increasing deployment of Location Based Services (LBS) and the significant market penetration of mobile phones supporting the Micro Edition of the Java 2 platform (J2ME™), there are new opportunities to develop rich client systems that leverage multiple location aware technologies. Useful tools to support such development include Java APIs and simulation tools. The standard Location API for J2ME™ is currently poorly supported by simulation tools, in particular those that can support map-based simulations. In this paper we review the available tools for this API and discuss some aspects of our own implementation, which is specifically designed for interactive testing. Being based on standard Java development and runtime tools, it is also easily extended to enable the testing of additional components to support various types of location based service, which in practice cover a wide range of applications meeting demands from different user sectors. These sectors include generic consumer services such as local weather information, niche consumer and business areas including targeted product or service information and enterprise applications such as supply chain management (Rao & Minakakis, 2003). Peer to peer contexts are seen as important areas for location based services (Thilliez & Delot, 2004) and many services can also be built upon generic notification mechanisms that build on the retail coupon model (Munson & Gupta, 2002). Supporting such services requires development of additional features above and beyond the current API specification.

LBS can be supported by a range of technologies that occupy a continuum between location awareness, where a device is fully aware of its location without assistance from

external sources, and device awareness where an external system is used to find the position of a device (Butz, 2004). In practice, implementations fall somewhere between these two extremes, and the balance between device awareness and location awareness will vary according to the features of the mobile terminal. For example, a device that incorporates a Global Positioning System (GPS) facility will know more about its location than one that relies only on information from a cellular network, and will therefore be more location aware.

2 The Location API for J2ME

Location APIs provide developers with the means to acquire positioning data on a client device. The Java Location API for J2ME™ (Loytana, 2003) defines a generic interface for positioning that is intended to work with most positioning methods. To maintain its generic nature, it does not expose any features that are based on specific technologies, though extensions are allowed for specific purposes. Although the aim of providing a generic interface might be regarded as a limitation, since it excludes some information from specific technologies, it also enables us to implement systems that span multiple sources of location information at the same time. This will become increasingly important as the devices and channels for tracking locations increase, enabling us to aggregate and prioritise different information sets that relate to the same target (Myllymaki & Edlund, 2002), as well as being able to choose between multiple methods of determining the location of a single device. Such hybrid systems give advantages such as fail-over, indoor/outdoor transparency and a choice between the speed and accuracy trade offs that could be made between GPS, cellular, or other positioning data (Ranchordas & Lenaghan, 2003).

2.1 The location API object model

The published Location API specification is primarily based on code documentation and does not include any design model or rationale. In addition, the documentation of the reference implementation (Nokia, 2004) is confined to technical discussions and does not include an architectural overview. Therefore we begin by providing some analysis of the API.

The Location API object model consists of eleven classes and two listener interfaces. Their design approach uses or implies several standard patterns, including the Façade, Factory Method, Singleton and Strategy patterns (Gamma, Helm, Johnson, & Vlissides, 1995) and standard Javabean–style accessors. Of the eleven classes, two are Exception classes (LocationException and LandmarkException) and another four (AddressInfo, Criteria, Orientation, and QualifiedCoordinates) are primarily Value Objects (Fowler, 2003). Some of the data stored by these objects may be unobtainable, depending on the mobile infrastructure and device being used, but this API anticipates likely future developments in mobile networks and devices and the level of location and context detail that they will be able to provide (Haiges, 2003).

2.2 Location related classes

Location objects are aggregates of AddressInfo and QualifiedCoordinates objects. They are immutable and transitory, reflecting the dynamic movement of a mobile device, and semantically are compositions rather than aggregations. It is interesting to note that the level of aggregation is very shallow, so that the Addressinfo object that is aggregated inside the Location object does not encapsulate any further containment graph. This means that we cannot, for example, identify containment relationships between buildings and rooms. This restricts the opportunity for leveraging this API for service discovery using proximity models (Jose, Moreira, Rodrigues, & Davies, 2003).

Location instances are acquired from a LocationProvider, a façade to the mobile device's underlying location information that consists of a factory method (parameterised by a Criteria object) to retrieve a LocationProvider instance, methods to return current or last-known Location objects and methods to register listeners for location and proximity events. The Coordinates class (the superclass of QualifiedCoordinates) encapsulates geometric methods such as calculating the azimuth (angle) and distance between locations. The Orientation class is completely separate from the rest of the object model, having no association or dependency relationships with any other classes. This is presumably because not all devices will be able to support orientation information. At a minimum, the device must be able to provide a compass azimuth value to support Orientation objects, with optional support for pitch and roll values.

Orientation objects can be derived from a factory method, but additionally there is a parameterised constructor, though it is unclear from the specification why the constructor is given public access. The classes discussed so far encapsulate the subset of the API that is directly related to the acquisition of location information from whatever underlying technology is available to the device. These are summarised in Figure 1.

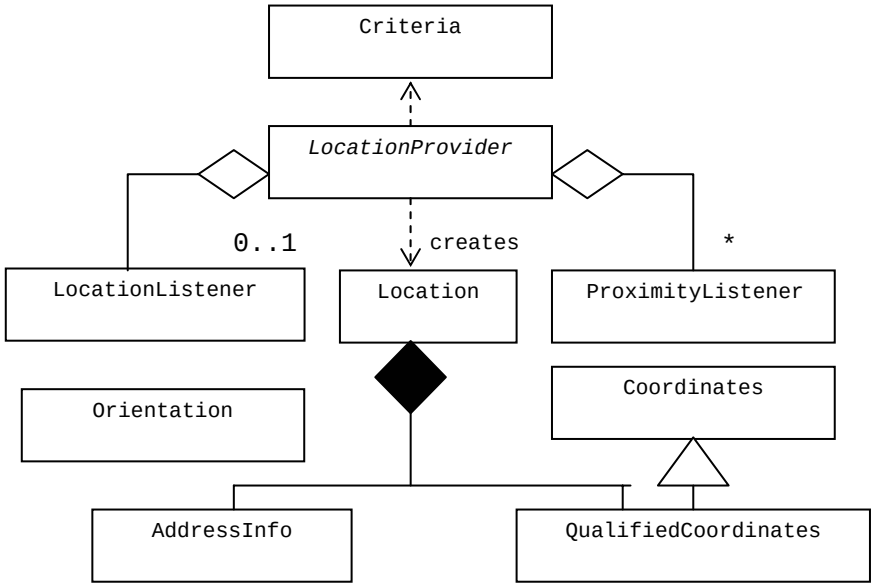

Figure 1: The Location related subset of the API

2.3 Landmarks and the LandmarkStore

What particularly marks out this API from other location based class libraries is the use of local storage to provide a persistent database of landmarks. This shifts the emphasis very much onto the mobile client in terms of location aware applications, enabling a local mapping from physical positions to symbolic locations (Hightower & Borriello, 2001). This approach means that the bulk of a location aware application can be installed on the mobile device rather than on the server. One key advantage of this is that applications are more likely to have useful functionality in a mostly-connected context.

In addition to those classes already discussed, which relate directly to the acquisition of dynamic location information, the API includes two classes to support the persistent storage of location related data, namely the Landmark and the LandmarkStore (Figure 2). Landmark objects, like Location objects, are partial aggregates of AddressInfo and QualifiedCoordinates objects, but Landmarks and Locations have different roles in the architecture. Location objects are immutable and transitory, reflecting the dynamic movement of a mobile device, and semantically are compositions rather than aggregations. In contrast, Landmark objects are intended to be persisted in the mobile data store and are mutable, so might be updated over time.

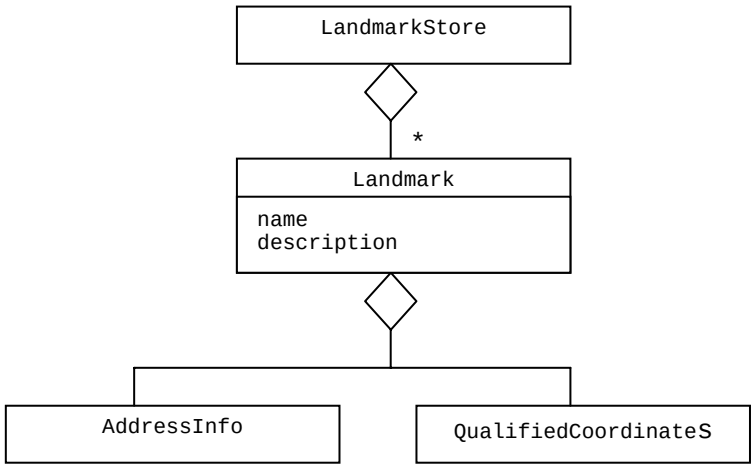

Figure 2: The classes related to persistent storage

The LandmarkStore acts as a facade to the underlying data store on the device, which in the case of a mobile phone application will probably utilise the Mobile Information Device Profile (MIDP) Record Management System (RMS) or a lightweight database implementation. On the surface, the LandmarkStore is simply a collection of landmarks. However, there can be many LandmarkStores on a device, shared by multiple applications. Landmarks may optionally be stored under a category name, and may be added to multiple stores and multiple categories. The only restriction is that a Landmark cannot be added to the same category in the same LandmarkStore more than once. This means that the underlying implementation needs to be reasonably clever in its use of the data store implementations available on mobile devices to ensure compatibility with the specification while maintaining performance and minimising resource demands.

An important feature of the API is that it implies (by means of a common, but unformalised interface) that the Locations generated by the LocationProvider can be used to assist in the construction of Landmark objects by providing initialisation data for their coordinate and address data, though there is no constructor that directly enables this. A mobile application can use the LocationListener interface to access regularly updated Location objects. Only one LocationListener can be registered with the LocationProvider at any one time. This listener therefore acts as a kind of Singleton for all components that need location information, regardless of whether the required information relates to landmarks or more dynamic location features such as speed and course, which are not stored persistently. The LocationListener is updated with current location information at specified intervals and it can expose and process that information in an application specific context to be accessed by other components on demand.

In addition, Landmark objects already in the LandmarkStore can be linked with Location objects in terms of listener behaviour. A mobile application can register multiple ProximityListeners that can be triggered when the current location is within a specific range of a given landmark. The application can retrieve the

QualifiedCoordinates from each landmark in order to register multiple listeners with specific coordinates to ascertain their proximity to the device. Having multiple listeners means that when proximity events are notified the listeners can be specifically targeted.

There are two basic scenarios for an application that utilises the LandmarkStore. First, the store can be a small static collection of application specific Landmarks that rarely need updating. ProximityListeners could be permanently registered to trigger proximity responses to the fixed landmarks. Systems that must deal with a larger set of Landmarks would require more dynamic provision of landmark information. In this context, the Landmark store would not contain preloaded objects but would add and remove them dynamically, using suitable push and/or pull mechanisms. ProximityListener registration and de-registration would be similarly dynamic. A hybrid approach might also be used, with batch replacements of data based on movement between larger areas.

3 Location APIs and simulators

Location APIs alone are difficult to leverage in development because their deployed context is a mobile system. Therefore some kind of testing environment is necessary to provide reference implementations and simulators for these APIs. Such tools assist us in building applications that can be tested on the desktop or on mobile devices to evaluate their usefulness or footprint on a small device. In this section we review some of the available Java location APIs and simulators.

3.1 LIF-based APIs and simulators

Most location based APIs that are compatible with mobile telephone networks are based on the interfaces published by the Location Interoperability Forum (LIF), though these interfaces are not necessarily fully adopted in practice by all system providers (e.g. (Ericsson, 2003b). The location API published by the LIF, which is now subsumed into the Open Mobile Alliance (OMA, 2004), is the Mobile Location Protocol (LIF, 2002). This API works on the assumption that the system is a device aware cellular network, and that location information is pulled from a server-based API. However, in location systems built on non-cellular or converged networks where a client device incorporates positioning technology the device is likely to push its location to a positioning component, and the location information will be quantitively and qualitatively different to that derived from a cellular system. Given the range of technology options available for positioning, generic platforms for location based services such as the Location Operating Reference Model (LORE) (Chen et al., 2004) integrate both pull and push models.

There are a number of publicly available Java location APIs and simulators that are based on the device aware LIF specifications. The O2 API and simulator is based on Redknee's ELS (Enabling Location Services) system (Redknee, 2002). It is essentially a server side API that communicates with mobile devices using web services. Specifically, it uses Glue as the wrapper around SOAP/XML data exchanges between client and server. A location query in this API returns a transaction ID, a result code, the

MSISDN (Mobile Station International ISDN Number) of the subscriber, the geodetic location format and the age of the information.

In the Orange API (Orange, 2004), location requests are parameterised by the MSISDN and authentication parameters. The response, which takes the form of XML-RPC types, includes a service response success flag, latitude, longitude, privacy options, timestamp and accuracy information. Although the data format is differently configured, the information returned is essentially similar to that in the O2 environment.

Ericsson's Java Mobile Location Application Programming Interface (JML API) supports both the standard LIF Mobile Location Protocol (MLP) version 3 (LIF, 2002) and the proprietary Mobile Positioning Protocol (MPP) version 5 (Ericsson, 2003a).

LIF based APIs may contain a number of interfaces and classes, but in general they contain very few methods and there is no attempt to include application-centric entities such as Landmarks. However, it is interesting to note the inclusion of cell topology related interfaces such as Polygon and EllipticalArea, which are notably missing from the Location Java API for J2ME.

It is instructive to look at other types of location API that do not restrict themselves to the basic device aware information set of the LIF protocol. One of the most interesting is the Oracle API (Oracle, 2004), which provides a small but rich set of classes that in some aspects go beyond other APIs, including features such as YellowPages, Routing and Mapping classes. In the Oracle context these objects are representations of server side features that are provided by the Oracle spatial database, so they are intended for server centric applications. However, there are certain features that might usefully be incorporated into client side location based applications.

3.2 Java Location API Implementations

The Nokia reference implementation for the Java Location API (Loytana, 2003) includes a GPS based simulator. It includes a single concrete location provider, the GPSLocationProvider, which relies on a stream of GPS data in NMEA 0183 format, provided either via a serial port or from a log file, to drive simulations. It provides a full implementation of the Java Location APIs along with some utilities to create LandmarkStores into the MIDP Record Management System. Its main drawbacks as an API simulator are that it only partially integrates with the Sun J2ME Wireless Toolkit and does not support map based simulations.

The Nokia prototype SDK for J2ME (Nokia, 2005) is targeted towards the emulation of specific Nokia devices. It includes an implementation of the Java Location API and integrates with third party IDEs such as Eclipse and JBuilder. It provides some routing simulation facilities via its *route tool*. This tool enables the definition of a series of coordinates that are linked by routes, which can optionally be overlaid across a map image. The tool then generates a GPS log file from this static route data.

Ericsson also provide an implementation of the Java Location API both on physical mobile devices and in a dedicated simulator. Being a stand alone tool it does not integrate with other tools, though it can read GPS data using the same log file format as the Nokia simulator. It does not provide any mapping or routing facilities.

4. Requirements for a Java location API simulator

There are a number of possible developer requirements for simulating the location API for J2ME that are not always met by the current tools. First, there is the issue of integrated development environments. Software developers in the J2ME space are likely to use one of the more common generic IDEs such as the Sun J2ME Wireless Toolkit (Sun, 2004), the NetBeans Mobility Pack or IBM Websphere Device Developer. The advantage of these IDEs is that they are intended for generic J2ME development and are not targeted to specific devices. Some of the currently available location simulation tools do not integrate with other IDEs. Another issue is the level of coupling between the layers of the simulation environment. If a tool only allows simulation to take place on the screen of the same PC as the simulation data then this may not be flexible enough for a developer who wishes to run simulations on physical devices. Finally there is the limitation that the only data input mechanism available for the simulators discussed previously is log files of raw GPS data. Although this can be derived from a real GPS device, this assumes the software is being run with such a device attached to the serial port. Even the Nokia route tool does not provide a dynamic interface, and generates GPS data as a batch process from the pre-loaded route coordinates.

To explore some of the features of the Java Location API, we have implemented a simulator that can be used in conjunction with freely available Java 2 Micro Edition development tools such as the Sun J2ME Wireless Toolkit. Unlike other implementations we have designed a loosely coupled system so that an interactive map based simulation is possible. The implementation of such a simulator falls into three areas of development. First, the implementation of the necessary APIs such that they can be deployed on a mobile device, specifically using the restricted libraries available on the Java phone platform. Second, the implementation of a simulation engine that can feed a virtual or actual mobile device with generated location data. The third aspect is how these two elements should best be coupled together.

Implementing the Java Location APIs on a mobile phone platform assumes a minimum platform, mandated by the location API specification, of the J2ME Connected Limited Device Configuration (CLDC) version 1.1. This is mostly a requirement based on the mathematical and geometric processing required by the API. Version 1.0 of the CLDC does not support floating-point numbers, and provides fewer of the java.lang.Math functions and Number objects. Even with the enhanced support of version 1.1, geospatial processing is difficult. The Coordinates class includes the ‘distance’ and ‘azimuth’ methods that require a number of mathematical functions. The code used in the implementation was based on a C++ implementation (McGovern, 2004) that

included the use of an ‘atan2’ function. The role of the function is to covert rectangular coordinates to polar coordinates. Although the Math class in standard edition of Java supports this function, it is not included in the CLDC libraries. Therefore a third party library was used for this implementation (Henson, 2004).

When implementing the rest of the classes that must reside in a telephone environment, perhaps the most limiting factor of CLDC is that it does not support the Java 2 collections framework. The containers that are available are those from version 1.0 of Java, principally Vector and Hashtable, and these have been used in the initial version of the implementation discussed in this paper. However to improve efficiency and reduce footprint these might in some cases be better replaced by optimized custom collection implementations.

4.1 Implied additional classes

To provide an implementation of a published API, developers are generally free to build classes to implement standard set of interfaces using whatever mechanisms they choose. However the Location API provides some constraints, partly because it is based on classes rather than interfaces and partly because of the combination of abstract methods and hidden constructors that are included. For example, the LocationProvider is an abstract class, necessitating the implementation of one or more suitable concrete subclasses to provide location information. On a device that supported more than one location aware technology (e.g. network based and GPS) separate implementations could be provided and switched according to availability or priority using the Strategy pattern (Gamma et al., 1995). For example in a device that could determine its location via both GPS and a cellular network, the GPS system could be given priority, but a network based strategy with a lower priority could be swapped in whenever the device was unable to utilise GPS services.

The Location class as defined in the API is immutable and only has a protected constructor, so it is necessary to provide some mechanism in order to set the properties of a Location within the LocationProvider. The protected constructor implies that a subtype will be used to set these properties. Thus an object can be created and initialised before exposing the supertype class interface to the client via the getLocation method. Figure 3 shows the implied subclasses in the location API.

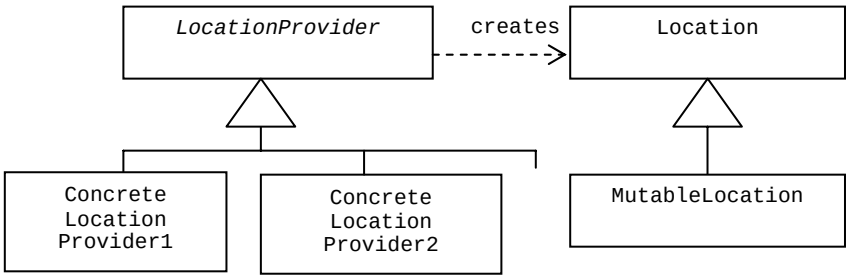

Figure 3: The implied subclasses in the Location API

A number of other classes were created to implement the APIs, including a ProximityDetails class, which proved useful in encapsulating the data received by proximity listeners. Since a single listener can register for multiple proximity events, which may overlap due to the closeness of coordinates, it was helpful to encapsulate this information in a value object (Fowler, 2003). Other additional classes were mainly helpers for data access (both locally to implement the LandmarkStore and over HTTP to couple the client side APIs to the remote simulation component) and parsing.

4.2 The map-based simulator

In order to provide the client API implementation with simulated location data, a separate Swing application was built to enable a user to control the direction and speed of a virtual mobile device moving across a map in a desktop environment.

Having a location API implementation running in a the simulated J2ME client and a moving object simulated in a separate J2SE application meant that the two elements needed to be run as separate applications and loosely coupled. The main question was how best to communicate between the LocationProvider running in the J2ME context and the mobile object running in a separate JVM process. Since the LocationProvider could only work with libraries available on the CLDC/MIDP platform (or we would not be able to also use the implementation on a mobile device) it was necessary to find a suitable mechanism for communicating between the mobile device and the moving object. In a normal desktop environment, applications running in separate JVMs can communicate via RMI. However, the only standard RMI client implementation for J2ME requires the Connected Device Configuration (CDC), a set of libraries that can be installed on large mobile devices but is not generally available in Java phones (Hodapp, 2002). The options for communicating data on the client side are limited to sockets (often not available in actual phones) SMS, HTTP and XML. In order to keep the client as simple as possible, the chosen solution was to add an adapter layer between the J2ME client and the J2SE application. This adapter layer consists of a simple web application that acts as an RMI client to the mobile object simulator and a JSP server to the mobile device, presenting moving object data via a JavaBean. The mobile client reads the location data from the JSP via HTTP. One major advantage of this architecture is that both simulated and actual wireless devices can easily access the same data. An alternative approach would have been to use XML communication between the mobile client and the web server, using J2ME web services (Ellis & Young, 2004). This would have been more in keeping with the other simulators that are publicly available, and also some mobile location interoperability specifications. However, the main advantage of reading non-XML data over HTTP was that the J2ME web service libraries are not required, reducing the client footprint, and the message sizes are smaller. Figure 4 summarises the various layers in the simulation system.

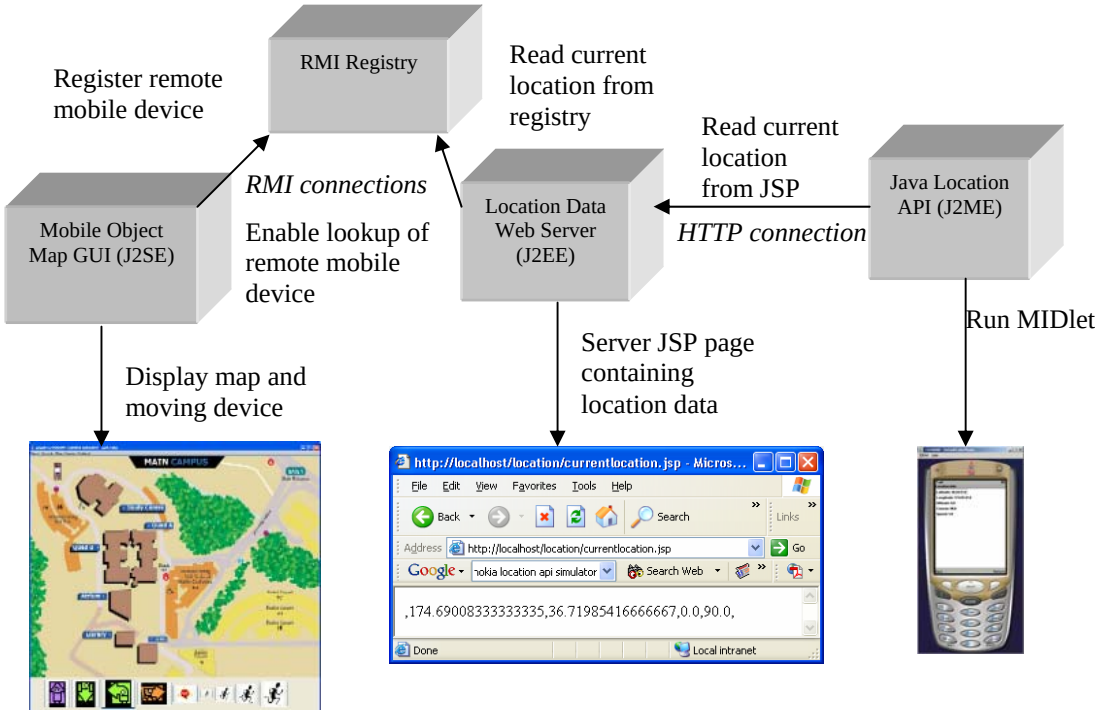

Figure 4: The layers of the simulation system

5 Tool integration

One of the main criteria for this simulator was that it would be easily used with commonly available, preferably open source, Java tools. This will enable third parties to extend the system as an open source project without dependency on proprietary tools. The chosen tool set is shown in table 1.

Table 1: Tools used in the simulator development and runtime

Type	Tool	Version	Usage	Source
Java Language	Java SDK	1.4.2	Development and runtime	Sun Microsystems
Java phone emulator	J2ME Wireless Toolkit	2.2	Runtime	Sun Microsystems
Java application server	Tomcat	4	Runtime	Jakarta project
Java build tool	Ant	1.6	Development and runtime	Jakarta project
Java unit test framework	JUnit	3.8	Development	JUnit.org
Java Integrated Development Environment (IDE)	Eclipse	3.0	Development	Eclipse.org

The main issue with these tools at development time was the incompatibility of the libraries used in the Eclipse IDE (used to develop the J2SE components) and those used by the J2ME wireless toolkit (used for the J2ME components). Although it is possible to import the necessary Java Archive (jar) files into Eclipse from the wireless toolkit, there are some aspects that cannot be tested in that environment. For example, JUnit tests run within Eclipse could not access the MIDP Record Management System implementation in the wireless toolkit due to some native code interfaces; this meant that a separate set of unit tests had to be written to run within the toolkit itself.

Problems at run time are largely to do with the rather complex layering of the system, It is essential that both the map based system and the web server can effectively locate the current RMI classes via the RMI registry. This means some careful configuration of the Java class path is required. Also, the location simulator cannot run without the web server running first, and the web server cannot provide location information until the map based simulator is running.

6 Summary and further work

The simulator currently meets the basic requirements of providing a map based interface for interactive testing of location based services. However there are a number of areas that need to be developed further if this system is to be a useful open source tool that could be used to test extensions to the location API. It may be necessary to focus development on a single IDE, such as NetBeans, that would enable a full test suite to be run in a single environment. Further, testing facilities would be enhanced by enabling the generation of a GPS log file from the interactive map, and also being able to use the map in a similar way to the Nokia route tool, so that a route can be drawn on it to generate the GPS data. The map based simulator itself needs to be extended so that it is easy to replace the map with any image, providing that the latitude and longitude of the corners can be identified. It is expected that the system will be developed in the light of feedback from users and may become a formal open source project if others wish to become involved in its continuing development.

References

Butz, A. (2004). Between location awareness and aware locations: where to put the intelligence. *Applied Artificial Intelligence, Special Issue on AI in Mobile Systems, 18*(6).

Chen, Y., Chen, X. Y., Rao, F. Y., Yu, X. L., Li, Y., & Liu, D. (2004). LORE: An infrastructure to support location-aware services. *IBM Journal of Research and Development, 48*(5/6).

Ellis, J., & Young, M. (2004). *JSR 172: J2ME™ Web Services Specification*. Retrieved December 20th, 2004, from http://jcp.org/en/jsr/detail?id=172

Ericsson. (2003a). *Mobile Positioning Protocol Specification Version 5.0*. Retrieved December 20th, 2004, from http://www.ericsson.com/mobilityworld/developerszonedown/downloads/docs/mobile_positioning/mpp50_spec.pdf

Ericsson. (2003b). *Statement of Compliance to LIF TS 101 version 3 specification*. Retrieved December 20th, 2004, from http://www.ericsson.com/mobilityworld/developerszonedown/downloads/docs/mobile_positioning/mlp300_soc.pdf

Fowler, M. (2003). *Patterns of Enterprise Application Architecture*. Boston: Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass.: Addison-Wesley.

Haiges, S. (2003). The location API: simplify access to mobile positioning methods. *Java Developer's Journal*, *8*(10), 52(53).

Henson, N. (2004). Float11: Class for float-point calculations in J2ME applications CLDC 1.1 (Version 0.5) [Java class].

Hightower, J., & Borriello, G. (2001). Location systems for ubiquitous computing. *IEEE Computer*, *34*(8), 57 -66.

Hodapp, M. (2002). *JSR 66: J2ME RMI Optional Package Specification Version 1.0*. Retrieved September 13th, 2004, from http://www.jcp.org/en/jsr/detail?id=66

Jose, R., Moreira, A., Rodrigues, H., & Davies, N. (2003). The AROUND Architecture for Dynamic Location-Based Services. *Mobile Networks and Applications*, *8*, 377-387.

LIF. (2002). *Mobile Location Protocol Specification Version 3.0.0*. Retrieved December 17th, 2004, from http://www.openmobilealliance.org/tech/affiliates/lif/lifindex.html

Loytana, K. (2003). *JSR-000179 Location API for J2ME™ (Final Release)*. Retrieved December 17th, 2004, from http://jcp.org/aboutJava/communityprocess/final/jsr179/index.html

McGovern, A. (2004). *Geographic Distance and Azimuth Calculations*. Retrieved 14th June, 2005, from http://www.codeguru.com/Cpp/Cpp/algorithms/general/article.php/c5115/

Munson, J., & Gupta, V. (2002). *Location-based notification as a general-purpose service.* Paper presented at the International Conference on Mobile Computing and Networking, Atlanta, Georgia.

Myllymaki, J., & Edlund, S. (2002, 8-11 January 2002). *Location aggregation from multiple sources.* Paper presented at the Third International Conference on Mobile Data Management.

Nokia. (2004). RI Binary For JSR-179 Location API For J2ME™ (Version 1.1): Nokia.

Nokia. (2005). *Nokia Prototype SDK 2.0 for the Java™ 2 Platform, Micro Edition*. Retrieved June 21st, 2005, from http://www.forum.nokia.com/main/0,,034-761,00.html

OMA. (2004). *Open Mobile Alliance*. Retrieved December 20th, 2004, from http://www.openmobilealliance.org

Oracle. (2004). *Introduction to Location APIs*. Retrieved December 20th, 2004, from http://www.oracle.com/technology/sample_code/products/iaswe/iASWE-LocationSample/doc/LocationAPI.html

Orange. (2004). Orange UK Location API.

Ranchordas, J., & Lenaghan, A. (2003). *A Flexible Framework for using Positioning Technologies in Location-Based Services.* Paper presented at the EUROCON 2003 - Computer as a Tool., Ljubljana, Slovenia.

Rao, B., & Minakakis, L. (2003). Evolution of Mobile Location-based Services. *Communications of the ACM, 46*(12).

Redknee. (2002). *Synaxis-2200™: ELS Release 2.0 Client Interface Specification Document*. Retrieved December 17th, 2004, from http://www.sourceo2.com/O2_Developers/Tools/Location_API.htm

Sun. (2004). J2ME Wireless Toolkit (Version 2.2): Sun Microsystems.

Thilliez, M., & Delot, T. (2004). A localization service for mobile users in peer-to-peer environments. *Mobile and Ubiquitous Information Access, 2954*, 271-282.