Directorate-General
Joint Research Centre
European Commission

Asgard

System Design

Martin Jacobson

11 October 2004

SDD–1.5

Support for External Security Unit

ipsc
Institute for the Protection
and Security of the Citizen

Revision History

Revision	Date	Author	Remarks
0.1	2004-06-16	MJ	First draft
0.2	2004-07-07	MJ	Second draft
0.3	2004-07-13	MJ	Third draft
1.0	2004-07-20	MJ	First Release (incomplete)
1.1	2004-07-21	MJ	4th draft
1.2	2004-07-23	MJ	5th draft
1.3	2004-07-27	MJ	yet another draft
1.4	2004-07-30	MJ	*idem*
1.5	2004-10-11	MJ	Changes to Data Descriptions

Summary

The *Earthquake Alert Tool*, which forms part of the DMA, provides a qualitative estimate of the humanitarian impact of earthquakes from a knowledge of where each earthquake occurs, the depth of the epicentre, and the strength of the tremor. Within its limitations, it has proved a useful indicator of which earthquakes are 'important' to aid organisations.

The ISFEREA project has other, similar models at various stages of development, and has also been asked by several organisations to provide websites (or web services) containing the output of these models. One of these, DG-TREN, actually wants to run simulations of disasters, rather than waiting for the disasters to occur.

On the other hand, the current tools are essentially experimental, prototype code, unsuitable for use in a production environment.

Asgard is the basis for the new services required by ECHO and DG-TREN and will provide a robust and scalable system, together with a single modelling environment that should make developing future models far simpler.

This document—the System Design Document—contains a semi-formal presentation of the top-level system requirements, followed by a description of the System Design proper.

Concerning Java This document contains numerous small samples of Java source code. These examples serve two purposes: (i) to *define* method signatures where these are part of the external interface of the component, and (ii) to *suggest* class definitions for classes referred-to in the method signatures. The external interfaces will be *published* in WSDL format, but this can be automatically generated from the Java source code. As the Java source is somewhat more concise, and easier to read (by a Human Being), the Java is given here.

Document Status This document represents the current status of Asgard development — it is neither practical, nor desirable that a system as complex as Asgard be specified completely before development commences. As a consequence of this, this document is incomplete, and is very likely to change in its details as development progresses.

Contents

Summary **3**

1 Introduction **7**
1.1 What is Asgard? 7
1.2 Why Asgard? 7
1.3 Scope of Asgard System 8

2 System Requirements **9**
2.1 Starting Models 9
2.1.1 Web Scraping 9
2.1.2 QDDS 9
2.1.3 Interactive 9
2.1.4 Batched, or Monte-Carlo analyses 9
2.1.5 Interactive Geoprocessing 9
2.1.6 Redo Aborted Jobs 9
2.2 Reporting 9
2.2.1 Templates 10
2.2.2 Alert Definition 10
2.2.3 OCHA 10
2.2.4 Event Mapping 10
2.2.5 Event Graphing 10
2.2.6 GDAS interface 10
2.2.7 Newsletter 10
2.3 Administration 10
2.3.1 Monitoring 10
2.3.2 Subscription maintenance 11
2.3.3 Subscription monitoring 11
2.4 Service Maintenance 11
2.4.1 Scraping 11
2.4.2 Report definition 11

3 Top-Level System Design **12**
3.1 Deployment 12
3.2 Inter-process Messaging 14

4 Principal Common Data Structures **17**
4.1 Of Values, Errors, Accuracy and Precision 17
4.2 Data Representation Requirements 18
4.2.1 Datum Point 18
4.2.2 Measurement 19
4.2.3 Event Data 19
4.3 Asgard Data Implementation 19
4.3.1 Value 19
4.3.2 ValMap 19
4.3.3 Datum 19
4.3.4 Measurement 21
4.3.5 EventData 21

5 Asgard Security **22**

6 SOAP-RPC Services **23**
6.1 Odin 23
6.1.1 Register 23
6.1.2 Address Lookup 24
6.1.3 Add Job 24
6.1.4 Change Job Priority 24
6.1.5 Delete Job 25
6.1.6 List Jobs 25
6.1.7 Notify Events Scraped 25
6.1.8 Proxy Services 25
6.2 Thor 25
6.2.1 Status 26
6.2.2 Capabilities 27
6.2.3 Run Job 27
6.2.4 Cancel Job 27
6.2.5 Get Results 28
6.3 Balder 28
6.3.1 Subscribe 29
6.3.2 Unsubscribe 29
6.3.3 Accept Subscription 29
6.3.4 Reject Subscription 30
6.3.5 Make Report & Alerts 30
6.3.6 Get Report 30
6.4 Norn 30
6.4.1 Store Results 30
6.4.2 Store Measurement 31
6.4.3 Get Event Parameters 31
6.4.4 Get Event List 31
6.4.5 Get Episode List per Event 32
6.4.6 Get Episode Results 32

7 Non-SOAP Component Functionality **33**
7.1 Hugin 33
7.2 Freya 33
7.3 End User UI 34

References **35**

Java Examples

6.1 register() 23
6.2 lookupAddress() 24
6.3 jobAdd() 24
6.4 changePriority() 24
6.5 jobDelete() 25
6.6 jobList() 25
6.7 notifyEventsScraped 25
6.8 getStatus() 26
6.9 getCapabilities() 27
6.10 runModel() 27
6.11 cancelModel() 27
6.12 getResults() 28
6.13 subscribe() 29
6.14 unsubscribe() 29
6.15 acceptSubscription() 29
6.16 rejectSubscription() 30
6.17 storeResults() 30
6.18 storeMeasurement() 31
6.19 getEventData() 31
6.20 getEventList() 31
6.21 getEpisodeList() 32
6.22 getEpisodeData() 32

1 Introduction

1.1 What is Asgard?

Put in the broadest of terms, Asgard is a system for running *Disaster Alert Tools* that either exist already (such as the Earthquake Tool), or that might be developed in the future. These tools have certain characteristics in common:

- A tool is run when the appropriate phenomenon is detected. Salient features of the event are passed to the tool as parameters.
- The tools comprise a series of sub-processes, including at least one geo-referenced analysis, and are computationally intensive.
- The tools are long-running processes, and cannot be expected to provide 'real-time' responses.
- The tools comprise an analysis phase that can trigger asynchronous messages.

Asgard then, provides a framework for running the models—it will schedule processes that perform automatic event detection, and when an event is detected, will start off the appropriate model, passing it the parameters for the event. If the analysis phase of the model returns an indication that an automated response is required, Asgard will send messages to its subscribers.

1.2 Why Asgard?

There are several reasons for developing Asgard:

- The existing Earthquake Alert Tool is unstable; it was developed as a research project to investigate the model, and needs to be re-engineered for production use.
- There are many possible Alert Tools that are either being developed (eg Hurricanes), or envisaged (eg Flooding), and that would benefit from a common framework in which they could be developed and run.
- New User Requirements mean that an interactive *front-end* has to be developed; this should be conceived in a way that makes re-use possible.
- New User Requirements mean that a new alerting & reporting mechanism (probably using XML) will have to be developed; this should be generalized to enable Asgard to act as a 'Web Service'.
- A common framework, will allow future development of the framework to benefit all Alert Tools. Such developments could include:
  - Performance enhancements (clustering, load-balancing, etc.)
  - Tool configuration via GUI

1.3 Scope of Asgard System

In terms of functionality, Asgard includes all the fundamental components necessary for scheduling and running models. Although it specifies the characteristics of an End-User Interface, the UI is not part of 'core' Asgard functionality[1].

For the developer of models, Asgard will present an API[2] to enable new, or improved models to be provided. This API will be in Java only, although it is hoped to be able to make the model 'parts' scriptable, probably using JavaScript.

[1] Although a very simple UI will be developed for testing purposes
[2] Application Programming Interface

2 System Requirements

2.1 Starting Models

How do models get run? Or, more precisely, given that it is Asgard's responsability to run models, what are the circumstances under which models will get run by Asgard?

2.1.1 Web Scraping

Certain specific urls point to resources that can be analysed to provide information regarding important natural events, such as earthquakes. There will be a process type in Asgard that will trigger the appropriate model based on parameters 'scraped' from the web page.

2.1.2 QDDS

For Earthquakes, we can use the QDDS service to detect new events, instead of 'scraping' a web-site. The software 'drops' information into a local directory whenever a new event is reported on the Net.

2.1.3 Interactive

There will be a web interface for interactive running of models. Note that models cannot be run in real time, so user will not receive results, but a page showing the status of his request(s).

2.1.4 Batched, or Monte-Carlo analyses

The same model is run multiple times, changing parameters, with the results expected together. This is a specific type of model, rather than a different interface to a single model.

2.1.5 Interactive Geoprocessing

It will be possible to perform interactive GIS queries... This is a low priority requirement.

2.1.6 Redo Aborted Jobs

Jobs that failed should be re-scheduled automatically (we need to be sure, however, that the job *can* be re-run.

2.2 Reporting

The result of running a model is a set of numbers. The 'normal' reporting process is for a web page to be produced containing the numbers produced by the model. This would normally be preceded by an analysis phase that produces a qualitative judgement from the output of the model.

Alerts are a special case of reports in that instead of the user requesting a report on a specific event, the user registers his desire to be informed automatically whenever a 'significant' event occurs.

2.2.1 Templates

The content and format of Reports and Alerts will be defined in Template files.

2.2.2 Alert Definition

A Report is generated on demand, while an Alert is generated & sent automatically. Users subscribe to an alert service, which implies a channel (email, SMS, fax, carrier pigeon, ...), a schedule (period of day when alert can be sent), and an alert level (ie, minimum severity to be alerted).

2.2.3 OCHA

This may be better defined as part of GDAS rather than in Asgard.

2.2.4 Event Mapping

It will be possible to provide various kinds of maps showing modelled data. The precise kinds of maps, the type of data they will display, the format of the map (JPEG, SVG, ...) will be defined later.

2.2.5 Event Graphing

The results of models will be graphable in the same as they are 'mappable'. A *lot* more detail will be required before developement can occur.

2.2.6 GDAS interface

Events are shown as clickable icons on a map. Clicking an icon gives detailed event report

2.2.7 Newsletter

A regular newsletter will be sent to alert subscribers (and registered users?) to provide them with summary statistics for the preceding period.

2.3 Administration

2.3.1 Monitoring

The system administrator needs to know that the various components of the system are running, and working properly. Although a sophisticated system seems, at first glance, to be desirable, the reality is that the opposite is true—the more sophisticated the monitoring process, the more likely it is that the monitoring process itself is a source of failure. Moreover, making the monitoring process part of the system it is monitoring is a 'Bad Idea' as a failure of the system is likely to make the monitoring inoperable.

The requirements are: (i) all serious errors (ie, ones for which Administrator intervention is required to resolve them) are logged to a single log file, in a single format, separately from any other logged messages, (ii) A low-level script (shell script, or Perl script, for example) should run regularly (via 'cron', for instance) to verify that Asgard is running, and that there are no messages in the log file. If there is a problem, an email should be sent to the system administrator.

2.3.2 Subscription maintenance

Query/List subcriptions, accept, refuse, add, modify and delete subscriptions.

2.3.3 Subscription monitoring

Tom wants list of who received which alerts, and when. What was the deal with reporting who didn't receive an alert, but should have done?

2.4 Service Maintenance

2.4.1 Scraping

Web scraping can be defined as follows: (i) get web page via supplied url, (ii) turn it into XML, (iii) transform the XML using XSLT, (iv) store results for processing. Step (iii) is the part that changes for different types of event, and is defined in an XSL file. This is not so difficult to change. However, the 'Scraper' should be able to gather information in other formats (TBD) as well.

2.4.2 Report definition

It should be possible to easily define and change report (and alert) templates, and mapping from variables to actual data.

3 Top-Level System Design

3.1 Deployment

Deployment is usually not a concern at this level of design. However, by grouping Asgard functionality appropriately, we obtain a design comprising a small set of loosely-coupled components. By treating each component as a separate process, we obtain several important advantages:

- The system is much more fault-tolerant—each component can fail individually without the entire system failing.
- Limiting the functionality of components makes each one simpler, thus more robust, and much less likely to fail.
- The system scales easily through distribution of components across multiple systems

Each component is implemented as one or more *servlets*[3], which can be deployed to any servlet container—in this instance, the Apache 'Tomcat' application server. Hence, deployment could be anything from a minimal 'everything-in-one-tomcat-instance' configuration, to one component per processor, with further load balancing possible.

Asgard's components are shown in figure 1 (page 13), and are described briefly below:

End-User Interface (User) Not strictly-speaking part of Asgard. Stands here as a proxy for an eventual real UI (such as in GDAS). Allows interactive submission of parameters for models, and enquiries on models that have completed

***Freya* (Administartor Interface)** Allows the administrator to monitor the models currently being run, and waiting to run. Also allows jobs to be re-scheduled, cancelled, and (perhaps) shuts-down the system.

***Odin* (Scheduler)** Manages the queue of jobs (both waiting to be started, and in progress), and communicates with the Model Runners to monitor the progress of each job

***Thor* (Model Runner)** Thor is responsible for running a job. It announces itself to Odin when it starts, and will report its status (and that of any job it is running) when asked by Odin.

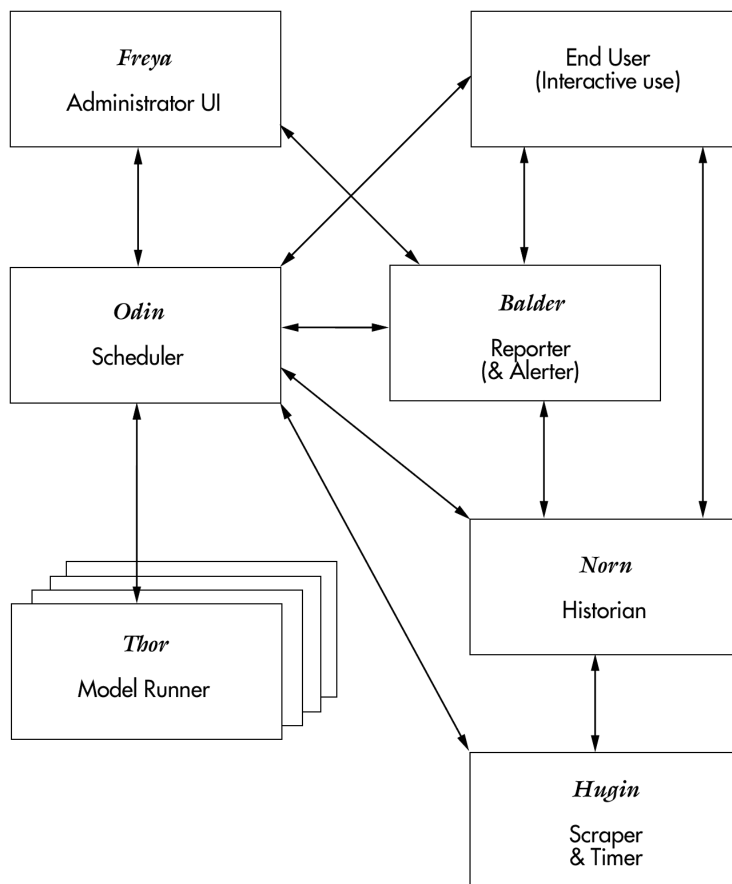
***Balder* (Reporter)** The reporting subsystem produces a report page for each new event for later display. Also, the messaging subscriptions are examined to see whether any messages (SMS, email, fax, telex, . . .) need to be sent out.

***Hugin* (Scraper)** Hugin is responsible for the automatic detection of new events. Although the Earthquake tool currently examines an HTML page from a public web site (known as "Web Scraping"), Hugin should also be able to parse other formats, including XML and ASCII text.

***Norn* (Historian)** Norn is responsible for recording details of events in a database for later reporting.

[3] Servlets are a standard Java technology, in some ways analogous to Applets, that run within a "Servlet Container" application, to provide server-side logic, and dynamic web pages. Servlets are the modern, fast and efficient replacement for CGI scripts.

Figure 1: Deployment Diagram—arrows indicate inter-process communication channels



3.2 Inter-process Messaging

Figure 1 (page 13) shows the message flows between the different processes in Asgard. The message flow is provided via SOAP-RPC so that each service is implemented as a particular method. Analysing the messages flows provides the information shown in table 1 (page 15). Some components, such as Freya for example, do not have any services shown. This is normal, in that only the internal SOAP-based services are shown; Freya provides services to Administrators through a Web interface that is described elsewhere.

If we reorganise and consolidate the information in table 1 so that the services are shown by server, we obtain table 2 (page 16). This is the 'Web Service' view, which regards the application as a collection of service providers.

Table 1: Services by Connection

Client	Server	Service
Freya	Odin	Register Add Job Change Job Priority Delete Job List Jobs
Odin	Freya	(none)
End-User	Odin	Register Address Lookup Add Job Delete Job List Jobs
Odin	End-User	(none)
Balder	Thor	Register Address Lookup
Thor	Balder	(none)
Norn	Odin	Register Address Lookup
Odin	Norn	Save Results
Hugin	Odin	Register Address Lookup Add Job
Odin	Hugin	(none)
Thor	Odin	Register
Odin	Thor	Status Capabilities Run Job Cancel Job Get Results
End-User	Balder	Get Report Subscribe Unsubscribe
Balder	End-User	(none)
End-User	Norn	Get Events
Norn	End-User	(none)
Balder	Norn	Get Event Data
Norn	Balder	New Event
Hugin	Norn	Is New Event?
Norn	Hugin	(none)

Table 2: Services by Component

Component	Service	Response
Odin	Register Service	
	Address Lookup	IP Address & Port number of service
	Add Job	
	Change Job Priority	
	Delete Job	
	List Jobs	List of Jobs in queue with status of each
Thor	Status	Status of Thor & any running job
	Capabilities	List of model names that can be run
	Run Job	
	Cancel Job	
	Results	Results of last job run
Balder	Subscribe	
	Unsubscribe	
	Accept Subscription	
	Reject Subscription	
	Make Report & Alerts	
	Get Report	Formatted Details of Event
Norn	Store Results	
	New Event Check	Yes or No
	Get Event Paramester	Parameters
	Get Event List	List of Event IDs
	Get Results per Event	List of Results
	Get Results	Results

4 Principal Common Data Structures

As a community of cooperating processes, Asgard components must use a common set of data representations and definitions whenever they exchange information. The issue of data representation is partly solved by use of Java for all components, and partly by the use of SOAP as the exchange protocol. However, there remains the question of what information needs to be exchanged, and how the data is to be defined.

4.1 Of Values, Errors, Accuracy and Precision

Note: for a fuller discussion of these issues, see any good text on Numerical Analysis, such as *Numerical Recipes*[5], chapter 1.

Values and Units Asgard is concerned with running mathematical models based on data obtained ultimately by observation and measurement of physical processes. As we shall see, one datum point is a *Vector* of (measured) values—dependant variables, and which is identified by a Vector of independant variables.

Table 3: Data Types

Value type	Java Class	Example
Real Number	double	Measurement
Whole Number	long	Count
Date/Time	Date	Date and Time of event
Logical	boolean	Is this a measurement? (or a forecast)
Text	String	Name of data source

The kinds of value that we are interested in are given in table 3, but note that we may need to extend this in the future to include, as an example, *Rasters*. It is interesting to observe the following:

- Values all originate as text (They are read from HTML, which is a textual format)
- Values are moved around the system as text (SOAP is based on XML, and XML is a textual format)
- Values need only be stored as text
- Values will be reported as text
- Values need only be treated as, say, real numbers when calculation have to be performed upon them.

It is therefore reasonable to propose that values will normally be represented as text fields, with an accompanying type indicator.

Related to the values themselves are the units in which they are expressed. This applies solely to the real numbers and counts, and needs to be explicit if there are different ways of describing the same value. For example, temperatures can be expressed in degrees Celsius, or in Kelvin, or even in degrees Fahrenheit, distances in Kilometres or Nautical Miles, and wind speed can be given in Km/h, or Knots. Where there can be this variability in units, the unit must accompany the value.

Errors and Accuracy Both errors and accuracy are measures of the *quality* of a reported value; however, the distinction that is usually made between the two is that errors affect *what* is measured, whereas accuracy affects the representation of the value. However, in either case, the result is that a reported value is almost invariably given with an associated tolerance value, such as "220Km/h ±15Km/h".

An interesting point to note is that geographical locations are often given in decimal degrees, plus or minus so many Kilometres—in other words, the units of the error value need not be the same as the units of the value.

Precision This is just the usual reminder that Precision $\neq$ Accuracy. Precision is a measure of the number of digits used to express a value, whereas accuracy describes how well the value is known. For example, the Earthquake model could calculate that the number of people affected by an earthquake is 753,003. If stated as such, without any indication of accuracy, the assumption is that the value is 100% accurate. If, as is more likely, this value is known only with an accuracy of 10%, then it is misleading to suggest that it is known more accurately: it would be better to say "750,000±10%", or "c.750,000".

4.2 Data Representation Requirements

In order to operate, Asgard needs to send information around the system concerning events. This section describes the data requirements.

An event is *an* Earthquake, *a* Hurricane, *a* Flood, and so forth, and each is characterized by a **Set** of measured and/or estimated data values.

4.2.1 Datum Point

Current impact estimation models treat an earthquake as a point event—that is, a phenomenon ocurring at a fixed, single location, at a single time.

Dependent Variables If we examine Earthquake data, we see that the measurements are of:

- **Geographical Position** The latitude and longitude of the epicentre
- **Phenomenological Time** The time of the shock (date and time)
- **Energy** i.e., 6.5 Richter, or 5.7 MM
- **Depth** depth of the epicentre, usually in Km

Independent Variables The datum itself is a function of the source of the data (eg USGS), and the time measurement was made.

A Datum Point is thus the set of variables that constitute our knowledge of a phenomenon and is characterized by position and time.

Impact estimation models for Hurricanes take into account the dynamic aspect, and deal with the known and predicted 'track' of the storm. Unlike earthquakes, hurricanes move, and thus, new measurements need to be taken at regular intervals. Also, different data sources can produce different measurements: particularly of wind strength. As we observed, hurricanes move, and it is important to know

where it is likely to be at some future time; for this purpose, data sources also supply predicted positions for future times.

Clearly, a hurricane is better described by a collection of Datum points, one for each track position and time.

4.2.2 Measurement

In order to reconcile the different natures of hurricane and earthquake events, and as a generalisation, we introduce the notion of a measurement. A measurement is one or more Datum points that together encapsulate our knowledge of a phenomenon at a particular time. An earthquake measurement will comprise one Datum, while Hurricanes will comprise several, though rarely more than ten.

4.2.3 Event Data

The sum of all the Datum points recorded for a given event constitutes the Event Data. Structurally, the `EventData` class is identical to the `Measurement` class, but they have been separated to underline the different concepts they represent.

4.3 Asgard Data Implementation

Unusually for a System Design Document, we need to define some important data structures — Java classes — since the high-level inter-component SOAP interfaces require them.

Figure 2is a UML class diagram that illustrates the static design of the common data structures used within (and between) the various Asgard components. The following sections describe the most important of the classes in more detail.

4.3.1 Value

The `Value` class represents a single, scalar value, such as "Latitude". It is identified by its `name`, and has an associated `value`. The value may have a unit, and may have some error estimate attached. Note that the value is held as a `java.lang.String`, and the "real" type is indicated by its `valtype`.

4.3.2 ValMap

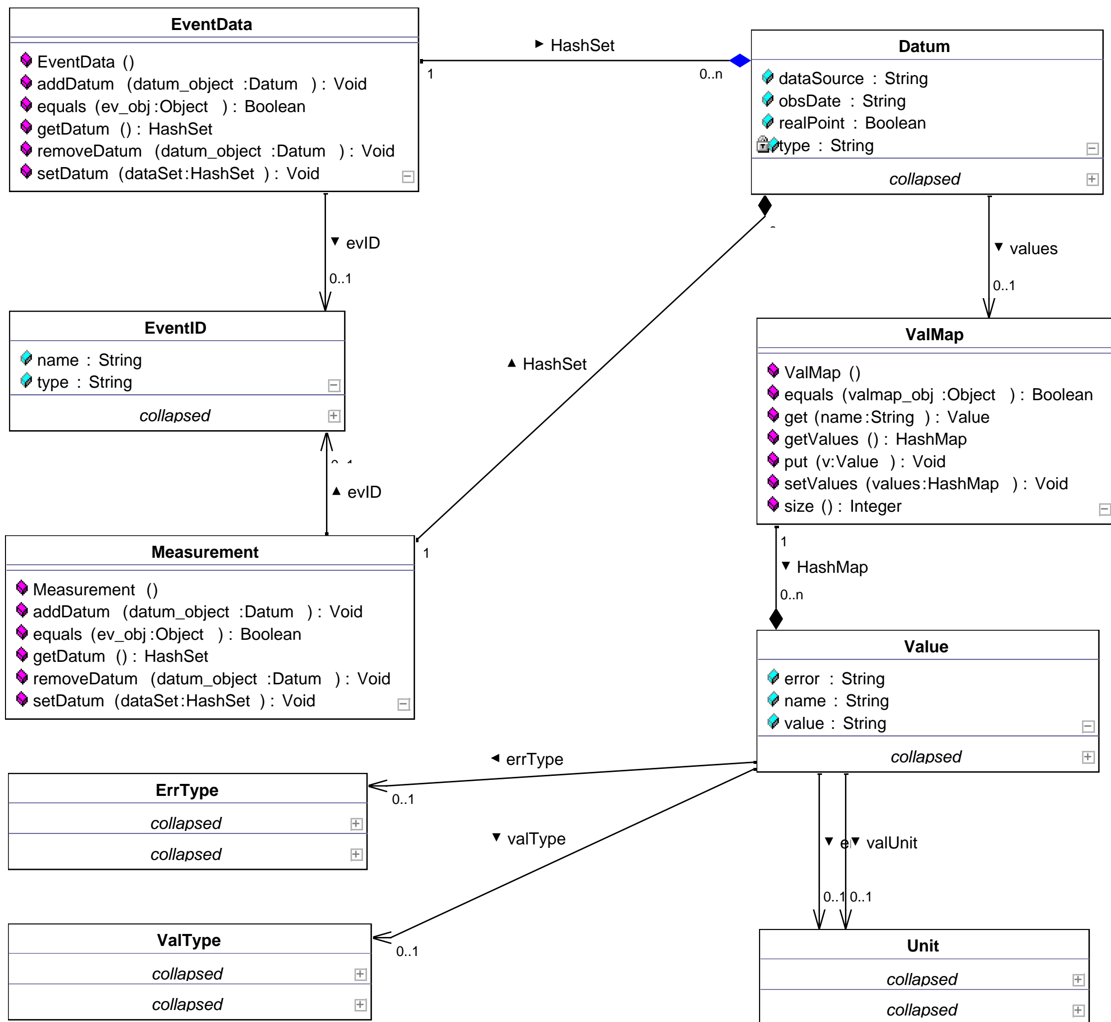
A `ValMap` is a type-safe implementation of a `java.util.HashMap`[4]. It exists to hold the set of values that constitute one datum point.

4.3.3 Datum

The `Datum` class represents a datum point, and comprises some meta-data fields, and a `ValMap` to hold the values.

[4]The standard collection classes store and return objects, and so are not inherently type-safe. This will be remedied in the forthcoming Java 1.5 release

Figure 2: Class Diagram — Data classes



4.3.4 Measurement

The `Measurement` class represents, . . . , a Measurement. It contains some meta-data, including, for example, the Event ID, and a `java.lang.HashSet` that contains the set of `Datum` objects that together form this measurement.

4.3.5 EventData

The `EventData` class represents the entire collection of information known about an event. Its structure is identical to that of `Measurement`.

5 Asgard Security

One of the foreseen uses of Asgard is that it be used to run models that simulate the outcome of man-made disasters. While useful for disaster management planning, it would also be very helpful to criminal organisations to be able to see the consequences of placing a dirty-bomb, say, in downtown Brussels. As a consequence of this, it will be necessary to design-in the ability to secure access to Asgard such that unauthorised use is not possible.

Intra-component security Although a distributed system, it is expected that Asgard components will be geographically adjacent, and probably on the same sub-network. This opens the possibility of isolating Asgard behind a firewall/router, which would be an effective way to avoid many kinds of security attack. However, this cannot be guaranteed, and may not be possible to obtain in practise. As a result, all communications between Asgard components

6 SOAP-RPC Services

This section describes the external interfaces provided by Asgard components. These are the interfaces provided via SOAP-RPC and include both those which will be published and accessible from other systems, and those which are purely internal. The services described correspond to those shown in table 2 (page 16).

6.1 Odin

General Remarks Odin is the 'heart' of Asgard; it coordinates the activities of the other components, channels requests from clients to the appropriate servers, and monitors the state of the system as a whole. Before describing the semantics of the web services it provides, let us examine its function as a whole by following the lifecycle of an 'episode'.

1. Odin receives an 'NotifyEventsScraped' command from Hugin, with a list of the events that were affected by the scrape. Odin
2. Eventually (never mind how!) the job arrives at the front of the queue. Odin has a list of Thors that are registered with it. It sends each one a 'Status' message. If it finds a Thor that is idle, it sends a 'Capabilities' message, and tests to see whether this Thor can run the requested model. If it can't, it continues with the next Thor, ...
3. If it finds a Thor that is idle, and can run the model, it sends a 'Run Job' command to it with the name of the model and the parameters.
4. Periodically, Odin will send a 'Status' message to the Thor, until it receives a 'Failed', or a 'Complete' in return. We'll assume that it worked.
5. It sends a 'Results' message to the Thor, who returns the values stored by the model.
6. It sends an 'Accept Results' message to Norn, who will archive the results.

There are some interactions which do not require the intervention of Odin; for example, a User subscription request to Balder. However, since Asgard should appear as a single entity to 'The Outside World', it will act as a proxy in these cases.

In order for the inter-process messaging to work, each process must know the URL where a service is found. In order to avoid the complication of maintaining configuration files for each process, Asgard uses a registry system instead. On startup, each process registers itself with Odin. If process 'A' wants to send a message to process 'B', but doesn't know B's address, it can send an 'Address Lookup' message to Odin, and use the returned address thereafter.

6.1.1 Register

Java Listing 6.1 register()

```
void register(String componentName, String serviceUrl, String nodeName)
  throws OdinException
```

In order to participate in an Asgard cluster, components have to send a Register message to Odin. In a real-world system, we should be concerned about system security, and so we should be ensuring that registering components are correctly identified and authorized. This is probably superfluous in the JRC. However, we ought to check that the name is recognized, and that the supplied URL corresponds to the caller's URL (at least, the host part).

6.1.2 Address Lookup

Java Listing 6.2 lookupAddress()

```
String[] lookupAddress(String componentName)
  throws OdinException
```

The only 'well-known' service address is that of Odin. If one component wants to send a message to another component, then it must first discover the target component's service url. This is the URL that the component supplied when it registered.

6.1.3 Add Job

Java Listing 6.3 jobAdd()

```
JobID jobAdd(String modelName, String endUser, EventData data)
  throws OdinException
```

Allows clients to request that a particular model be run.

We will work on the assumption that a queueing system is in vigour: that is, when the 'add job' task runs, it simply has to place the request, and its parameters, into the queue. The queue implementation should be able to identify individual jobs, by means of a unique ID. This value should be returned to the calling process.

6.1.4 Change Job Priority

Java Listing 6.4 changePriority()

```
void changePriority(JobID jid)
  throws OdinException
```

Allows an administrator to change the priority of a waiting job in the queue.

This request will only be accepted from a source identified with an Administrator rôle. Only values from 0 to 5 will be accepted. The Job ID must correspond to a Job that is in the queue, and which has not yet been started. The algorithm for determining in which order queue entries are processed has not yet been established, but it will tend to deal with more urgent entries before less urgent entries.

6.1.5 Delete Job

Java Listing 6.5 jobDelete()

```
void jobDelete(JobID jid)
  throws OdinException
```

Request that a job be deleted.

This request will only be accepted from a source identified with an Administrator rôle. The Job ID must correspond to a Job that is in the queue, and which is not marked as completed. If the job has not started, it can be deleted immediately; if the job is in progress, Odin must send a cancel job message to the appropriate Model Runner, which may or may not be able to cancel the job. The routine returns immediately, which means that the job, although having been requested for deletion, may still be present in the queue.

6.1.6 List Jobs

Java Listing 6.6 jobList()

```
QueueStatusBean[] jobList()
  throws OdinException
```

6.1.7 Notify Events Scraped

Java Listing 6.7 notifyEventsScraped

```
void notifyEventsScraped(EventID[] events)
  throws OdinException
```

Sent by a Web Scraper (or equivalent) to inform Odin that the supplied Events were affected by the scraping. The intention is that Odin will determine whether models need to be run for any of the events, and add jobs to its queue as neccessary.

6.1.8 Proxy Services

Odin must be able to act as a proxy server for services that should be directly accessible. This includes, for example, Balder; if a remote user wants to subscribe to an alerting service, say, Odin should be able to pass on the request to Balder. This should be a configurable item.

6.2 Thor

General The only working model (at the time of writing) is the Earthquake Alert Tool, which will be re-engineered as a parallel activity to this project. Thor is responsible for starting, monitoring, and cancelling jobs. Thor is the component

which could benefit the most from careful, efficient coding techniques, so let us list the short- to medium-term requirements for this component:

- We would like to be able to decompose a model into separate sub-models whilst maintaining a single model 'facade' for the rest of the system. The sub-models could then be treated as separate tasks, and run in their own right as if they were whole models (Recursive Task Decomposition).
- By default, the sub-tasks will be run sequentially
- We would like to be able build a model from its sub-tasks in a user-programmatic way, for example using the Bean Scripting Framework, and JavaScript as the scripting language.
- We would like to design processor-intensive tasks in a way that would allow an eventual migration to 'Grid' technology, or some other distributed resource system.
- Sub-models (or sub-tasks) that use external resources (databases, or GIS engines, ...) typically are penalized by the comparitively long connection and start-up times involved in their use. Resources pooling techniques should be used to alleviate these overheads.
- Each model should maintain statistics about its historical performance on a particular computer (No of times run, Mean run time, StdDev)
- We would like a model to be able to report on its own progress. Ideally, this would be based on actual progress, but an alternative would be to estimate it from the statistics.

It will not be possible to implement these requirements in their entirety for the first release of Asgard, but the detailed design of Thor will take them into account in order to facilitate their later development.

6.2.1 Status

The Status message is used to obtain the status of Thor, and that of any models it is currently running. It is handled by the `getStatus()` method.

Java Listing 6.8 getStatus()

```
ThorInfoBean getStatus(JobID[] jobs)
  throws ThorException
```

Request Parameters An array of JobID. The Job ID sent with the 'Run Job' command (see below).

Return Value(s) The status will be returned in a `ThorInfoBean` which contains the status of the Model Thor as a whole, plus the status of each job requested.

Semantics If the `jobs` array is of zero length (but is not null), then only Thor status information will be supplied—that is, `jsbs` will be an array of zero length. In contrast, if the `jobs` array is null, then information for every job will be returned.

6.2.2 Capabilities

Show the list of model names that this component has been configured to run.

Java Listing 6.9 getCapabilities()

```
String[] getCapabilities()
  throws ThorException
```

Request Parameters None

Return Value(s) Array of model names

Processing Logic

6.2.3 Run Job

Ask Thor to run a model

Java Listing 6.10 runModel()

```
void runModel(String modelName, JobID job, Measurement data)
  throws ThorException
```

Request Parameters (i) Model name, (ii) Job ID, (iii) Model Parameters

Return Value(s) None, unless an exception is thrown

Processing Logic

6.2.4 Cancel Job

Ask Thor to cancel a model

Java Listing 6.11 cancelModel()

```
void cancelModel(JobID job)
  throws ThorException
```

Request Parameters Job ID

Return Value(s) None, unless an exception is thrown

Processing Logic

6.2.5 Get Results

Ask Thor to return the results of a model

Java Listing 6.12 getResults()

```
Measurement getResults(JobID job)
  throws ThorException
```

Request Parameters Job ID

Return Value(s) Either the results of the job, or an exception is thrown

Processing Logic

6.3 Balder

Roles & Responsibilities Balder and Norn are closely related, their roles being illustrated by the following (typical) scenario:

1. Odin determines that a model has finished, and requests the results from Thor
2. Odin sends the results to Norn, who archives them
3. Norn sends a make report message to Balder
4. Balder then sends a get event data message to Norn
5. Balder generates a report (in XML) and caches it.
6. Balder examines its rules for this type of model, and determines whether an email and/or SMS alert should be sent
7. Assuming alerts need to be sent, Balder then processes its subscriber list and sends the requested alert

Another typical scenario is as follows:

1. The End-UserUI requests a report (identified following some interaction with the historian) from Balder
2. Balder checks its cache to see whether it has already generated a report—assume it hasn't
3. Balder then sends a get event data message to Norn.
4. Balder generates a report (in XML) and caches it.
5. Balder then sends the cached report back to the End-UserUI

Although it seems a little long-winded (in the first scenario) to get Odin to send the data to Norn, who then sends it to Balder, it avoids a potential timing problem where Balder requests data that Norn hasn't yet recorded.

6.3.1 Subscribe

Ask Balder to subscribe someone to an alert service

Java Listing 6.13 subscribe()

```
void subscribe(String alert, String email, ChannelInfo[] channel)
  throws BalderException
```

Request Parameters

Return Value(s)

Processing Logic

6.3.2 Unsubscribe

Request the cancellation of a subscription

Java Listing 6.14 unsubscribe()

```
void unsubscribe(String alert, String email)
  throws BalderException
```

Request Parameters

Return Value(s)

Processing Logic

6.3.3 Accept Subscription

Accept a subscription

Java Listing 6.15 acceptSubscription()

```
void acceptSubscription(String alert, String email)
  throws BalderException
```

Request Parameters

Return Value(s)

Processing Logic

6.3.4 Reject Subscription

Reject a subscription request

Java Listing 6.16 rejectSubscription()

```
void rejectSubscription(String alert, String email, String reason)
  throws BalderException
```

Request Parameters

Return Value(s)

Processing Logic

6.3.5 Make Report & Alerts

Request Parameters

Return Value(s)

Processing Logic

6.3.6 Get Report

Request Parameters

Return Value(s)

Processing Logic

6.4 Norn

Norn is responsible for archiving model parameters and results, and serving them on request. However, one of the characteristics of disaster 'events' is that they are often of extended duration, and, in the case of meteorological phenomena, such as hurricanes, of changing position.

6.4.1 Store Results

Java Listing 6.17 storeResults()

```
void storeResults(EventID evID, EventData results)
  throws NornException
```

Request Parameters

Return Value(s)

Processing Logic

6.4.2 Store Measurement

Java Listing 6.18 storeMeasurement()

```
public EventID storeMeasurement(EventID proposedEventID, Datum params)
  throws NornException
```

Request Parameters Two parameters are passed, (i) the proposed event ID and (ii) the set of event parameters

Return Value(s) The event id for this measurement. It is possible that the event data is already being archived under a different id—in this case, the previous id is returned.

Processing Logic

6.4.3 Get Event Parameters

Java Listing 6.19 getEventData()

```
public EventData getEventData(EventID evID)
        throws NornException
```

Request Parameters

Return Value(s)

Processing Logic

6.4.4 Get Event List

Java Listing 6.20 getEventList()

```
public LinkedList getEventList()
```

Request Parameters

Return Value(s)

Processing Logic

6.4.5 Get Episode List per Event

Java Listing 6.21 getEpisodeList()

```
public LinkedList getEpisodeList(EventID evID)
```

Request Parameters

Return Value(s)

Processing Logic

6.4.6 Get Episode Results

Java Listing 6.22 getEpisodeData()

```
public EventData getEpisodeData(EpisodeID epID)
```

Request Parameters

Return Value(s)

Processing Logic

7 Non-SOAP Component Functionality

Where the last section described the functionality provided by components via SOAP-RPC, this section describes the functionality provided to users, or autonomously, and which *consumes* the services provided by SOAP-RPC.

7.1 Hugin

Hugin is an autonomous component that is used to detect significant events, obtain parameters, and trigger the running of the appropriate model that calculates its impact. The significant actions of Hugin are:

- Look for event sources at defined intervals. The sources & the intervals will be defined in a configuration file
- Depending of the source type, and the individual source, extract a list of events and their associated parameters from the source data. For example, if the source type is HTML, and the source itself is the USGS, then the process will involve an XML transformation pipeline. Other sources may be easier to extract information from.
- Determine an 'event id', ask Norn to return the previous measurements stored, and check whether this is either: (i) a new measurement, (ii) a significant change to an existing measurement.
- If either (i) or (ii) above is true, send Norn the measurement and ask it to store it.
- Also, if (i) or (ii) were true, put the new measurement into the event data structure, and ask Odin to run a model against it.

7.2 Freya

Freya presents a user interface to allow an admistrator to monitor Asgard, and perform a limited set of actions on components of the system. The interface will be produced in HTML, and Freya itself will be a web application implemented as servlets within a container (almost certainly Apache Tomcat).

The actions that the administrator will be able to perform are:

- System Monitoring. In order to check the 'health' of the system, the administrator will be able to:
  - View system logs
  - Enquire of Odin which components are registered, and what the status is of each Thor component
- Subscription Maintenance. The specific functions are:
  - Create/Modify/Delete/List Channel

  - Create/Modify/Delete/List Alert
  - Create/Modify/Delete/List Subscription
  - Accept/Reject/List Subscription requests

7.3 End User UI

The End-User UI does not, strictly-speaking, form part of Asgard; Asgard provides an interface that allows other applications to include Asgard functionality within them. The functionality described here, therefore, is provided by Asgard in order to allow a user interface to be built using it.

In fact, a minimal UI *will* be built for testing and demonstration purposes. The Asgard functionality that *ought* to be provided through an end-user UI includes:

- List Jobs (should only list user's jobs) with their status
- Submit new job
- Delete submitted job
- List Events
- List Results for Event
- Get Report
- Get Event data — this data could be used to pre-populate a form for submitting a new job.

References

[1] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[3] Jess Garms and Daniel Somerfield. *Java Security*. Wrox, 2001.

[4] Brett McLaughlin. *Java & XML*. O'Reilly & Associates, 2001.

[5] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numeric Recipes in Pascal*. Cambridge University Press, 1989.