SPACE ENVIRONMENT TECHNOLOGIES
Advanced Space Weather Products and Services

A DATA CONVERSION SUB-SYSTEM FOR AN IONOSPHERIC FORECAST SYSTEM

A Thesis
Presented to
The Department of Computer Engineering
San Jose State University
In Partial Fulfillment
of the Requirements for the Degree
Master of Science

by
Paulus Zegwaard
December 2004

Space Environment Technologies Technical Report TR2006-001

A DATA CONVERSION SUB-SYSTEM

FOR

AN IONOSPHERIC FORECAST SYSTEM

A Thesis
Presented to
The Department of Computer Engineering
San Jose State University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

by
Paulus Zegwaard
December 2004

APPROVED FOR THE DEPARTMENT OF COMPUTER ENGINEERING

[illegible]

Dr. Lee Chang

[illegible]

Dr. Michael Robinson

[illegible]

Dr. Kent Tobiska, Space Environment Technologies

APPROVED FOR THE UNIVERSITY

[illegible]

ABSTRACT

A DATA CONVERSION SUB-SYSTEM
FOR
AN IONOSPHERIC FORECAST SYSTEM
by Paulus Zegwaard

The data conversion subsystem of this thesis project is part of an Ionospheric Forecast System (IFS). The forecast system employs a number of asynchronously linked space-physics models that use and produce data in a variety of formats (usually simple ASCII formats), which are expected to change relatively often.

This report describes a GUI web application that allows a model maintainer to generate a definition for a changed ASCII file format very easily. This definition is then used to generate Java classes dynamically, which convert ASCII files to data objects (Java classes) or visa versa. The IFS stores the serialized data objects in a database to bridge timing differences resulting from the asynchronous execution.

The chosen solution allows for a fast adaptation to changes in ASCII file format. The dynamic loading of the conversion Java classes avoids a system restart in the case of a changed file format.

ACKNOWLEDGMENTS

I would like to thank the people who made this project possible and enjoyable to work on. I am very grateful that Dr. W. Kent Tobiska, President of Space Environment Technologies, gave me the opportunity to work on this research project, which is a key element in the development for the IFS. About more than halfway through the project he had the insight that a small change in requirements would be a big improvement in the final result. I'm glad he pointed out this somewhat subtle change in direction of the project. The impact on the implementation was substantial and in hindsight it became very clear that this made the result much more valuable and generic. Yes it was more work, but real hacking is about throwing code away when a better solution becomes obvious.

I would like to thank Dave Bouwer, Chief Engineer at Space Environment Technologies, who helped me to understand the internals of the IFS and gave me feedback on the day to day work. He also was inspirational and came up with ideas of how to approach the implementation of the conversion system.

Last, but not least, I want to thank my professors at San Jose State University. In particular I thank Dr. Lee Chang, my primary advisor, for encouraging me to take on this project, staying out of my way when I needed the freedom to explore, and always being right there when I needed his advise and support. I also thank Dr. Michael Robinson, member of my thesis committee, who is a very good teacher and helped me to understand some of the essential nitty-gritty details of computer engineering.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

1.1 Objective and Project Scope

The goal of this thesis project will be the assessment of the scope and risks of the proposed conversion sub-system, according to the criteria mentioned in section 1.6.1. The work will involve building a prototype demonstrating the feasibility of the sub-system, following the iterative development cycle in use by Space Environment Technologies (SET), the principal developer of the Ionospheric Forecast System (IFS).

1.2 Deliverables

The project deliverables are:

1. Artifacts of all iterations and stages of the development process, such a requirements document, Unified Modeling Language (UML) diagrams of the design and a software specifications document, a test plan and an evaluation-feedback document after an iteration
2. A functional prototype of the proposed sub-system
3. A final report combining all the important parts of the documentation

1.3 Report Overview

The following sections of this introduction chapter explain more about the background of the IFS in terms of the design challenges and the chosen solutions to meet these challenges. This chapter then continues with describing the problem topic of this thesis project and the proposed solution.

The methodology is discussed in Chapter 2, subdivided into sections on compatibility, programming environment, software development model, and object oriented design. Chapter 3 covers the system design, with sections on requirements, architecture, software design, and testing approach. In Chapter 4 the results are discussed, covering the performance experimenting, the building of the GUI web application, the prototype system, and the test suite.

1.4 Background

The Ionospheric Forecast System (IFS) is under development by a group of organizations involved in developing space-physics models with Space Environment Technologies (SET) in the role as primary systems developer.[1] This section contains a description of the functionality and system design challenges of the IFS, together with the chosen solutions. It serves as a background in order to gain a better understanding of the data conversion topic of this thesis project.

[1] W. Kent Tobiska and Dave Bouwer, Sections 1.4 to 1.7 are a summary of personal communication and internal SET documents, February 2004.

1.4.1 Ionospheric Forecast System Forecasts Space Weather

The IFS is, in popular terms, a space-weather forecast system. The colloquial expression “space weather” is commonly used to describe the effects of solar radiation on the Earth’s ionosphere and thermosphere. For example, space weather forecasts can be used to improve satellite orbit and attitude estimation, HF radio propagation, and GPS signal availability applications.

A more scientific description of “space weather” is “the shorter-term variable impact of the Sun’s photons, solar wind particles, and interplanetary magnetic field upon the Earth’s environment which can adversely affect technological systems.” It includes, for example, the effects of solar coronal mass ejections, solar flares and irradiances, solar and galactic energetic particles, as well as the solar wind, all of which affect Earth’s magnetospheric particles and fields, geomagnetic and electrodynamical conditions, radiation belts, aurorae, ionosphere, and the neutral thermosphere and mesosphere during perturbed and quiet levels of solar activity.

1.4.2 Ionospheric Forecast System (IFS) Functionality

The ISF produces forecasts based on:

1. Real-time input data from SET, partner institutions, and third parties as well as statistical climatology data sets.
2. Remote distributed clients executing space-physics models that use data and produce numeric ionospheric forecasts. The models also use data from other models as input.

Customers and client models will request data sets and the system will provide data sets “Just-in-Time,” meaning that the data sets are the most recent and best estimates. These data sets contain the parameters that form the actual forecasts.

1.4.3 Design Challenges of the IFS

The process of producing forecasts depends on a number of independent factors that basically fall in one of two categories: factors with impact on the data communication between geographical-distributed nodes in the system, and factors influencing the data streams resulting from the execution of models and inter-related models. If the system were implemented as synchronous end-to-end system of models and data, the impact of a small-scale failure somewhere (communication latency or one of the models failing), would result in a cascade of failures.

This is unacceptable for mission-critical operations. For that reason the design philosophy has two important principles (described in more detail in the next section, *IFS Implementation Solutions*): asynchronous execution of the models and ensuring an uninterrupted data stream.

These design principles facilitate a system that results in a gradual degradation of forecast accuracy in the event of data outages or latency. Because of this “graceful degradation” design philosophy operators can respond in a timely fashion and avoid a failure of the entire system.

1.4.4 IFS Implementation Solutions

The first design principle is implemented by asynchronously linked models (that is, individual models run independently of each other and deposit their outputs to a central database accessed by a central server that maintains the current state of all the data sets and models). This results in models being less dependent on the performance of other related models.

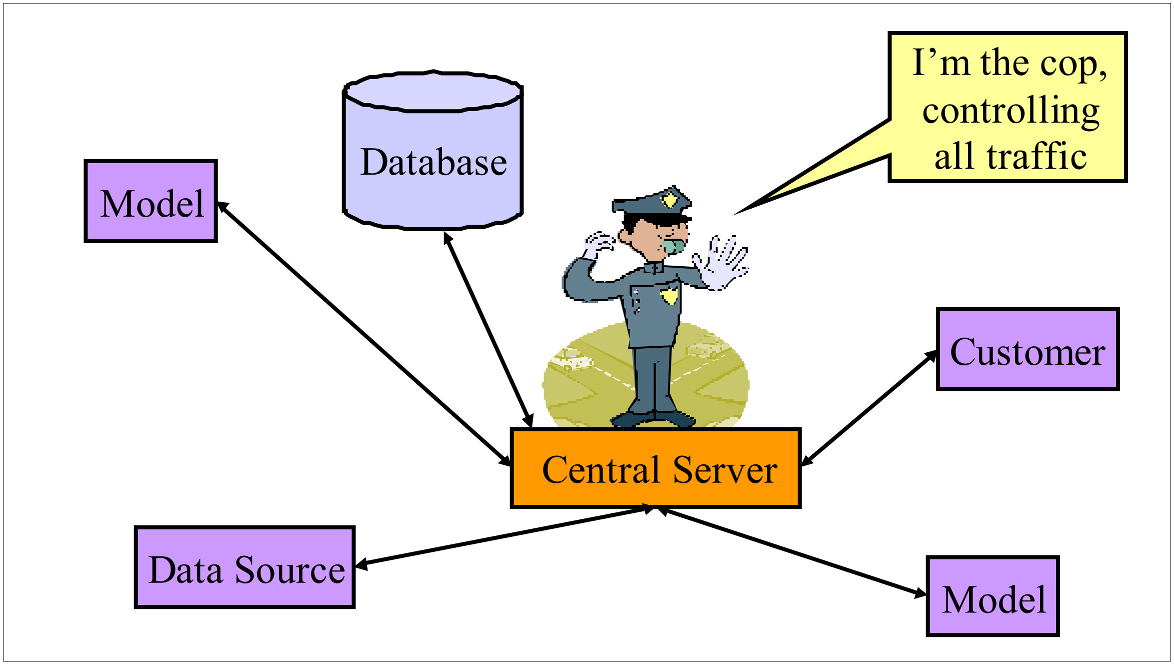


Figure 1: Simplified Overview of the IFS Architecture.

Note that the design goal here is not archiving data, but that the database is used to bridge timing differences resulting from asynchronous data handling and model operations. In other words, the data sets needed by all the models and end-users are retrieved and disseminated on a "Just-in-Time" basis, ensuring a particular validated data set is the most recent.

Since data rapidly evolve from forecast, to "nowcast" (term used by SET), and finally to historical status, there is also a need to add tags to the data sets that make it possible to determine the state on the time line. One function of maintaining the temporal state of a data object is to monitor the utility of the data. Subsequently, data need to be validated at each state change, and in the event of invalid data, a "best-estimate" can be used as a replacement.

The second design principle, an uninterrupted data flow, is implemented by having a "second best" replacement data stream available at all times at the central database. A primary stream is called the "A" stream, whereas the secondary stream is labeled "B."

The "A" data stream is the most accurate and the purpose of the "B" stream – often based on statistical estimates based on historical climatology – is to ensure that there is reasonable data available in case data from the “A” stream is not available. For instance the "A" stream could be interrupted because of operational reasons, such as network unavailability. The “B” stream is not as accurate and/or extensive as the “A” stream, yet it is good enough to keep the models going. The models will still produce valid data, although without the accuracy that the “A” stream would give.

Of course other, more traditional, ways of mitigating risks of system failure are used as well, such as backup computers and independent network connections.

If errors occur, the system will report the error condition (possibly by sending email), so that operational staff can undertake corrective actions.

1.4.5 Four-tier Architecture of the IFS

1. Database - The function of this database is to maintain all data objects and their current state.
2. Client server - The client server maintains communication between models, data, and end-users. This clear separation ensures systems security, performance, robustness, extensibility, ease of use, and maintainability.
3. Compute engine - Because of the high performance and security requirements of space-physics models, and because a number of models will be executing at the same central location as the database server, compute engines located at partner institutions comprise one of the tiers.
4. Clients and Customers - In this architecture, clients can be either remote models or end-users (customers retrieving forecasts).

1.5 Thesis Project Challenge

The design choice in the IFS is to encapsulate the data within Java data classes (data objects). One of the advantages of data objects over files is that a data object can contain methods that operate on the data, such as methods that have the knowledge how to validate the data set contained in the data object.

Different independent researchers, partners of SET, are developing the models used in the IFS and each model has its own file format, which is expected to change since development of the models is still an ongoing process. Imposing a single common data format for all model input/output (I/O) instances is rather impractical. For that reason the IFS designers envision a more flexible approach. The ideal design goal is to go beyond static data standards for the models and create full data transparency. Creating a conversion sub-system that can handle all the different formats in use and can be adapted to changes fast and easy is probably the closest as one can get to the ideal of the design goal. Researching the feasibility of such a conversion sub-system is where this thesis project comes in.

1.5.1 Data Formats Challenges

Although data format standards are a useful approach, in practice they also impose problems within the scientific community, such as manipulatibility and differences between the output of a model and the input of another. Challenges in general, and specific to this proposed thesis project, are summarized in section 1.5.1.1 and section 1.5.1.2.

1.5.1.1 General Challenges

1. As data standards evolved (for example FITS, net-CDF, and so on), they grew more and more complex, and are often beyond the time constraints and resources of many of the small scientific groups,
2. Individual researchers have neither the inclination nor resources to re-write all their legacy software to meet the protocol of a standard (the vast majority of scientific data is in simple ASCII formats specific to a particular model), and this results in the growth of native formats for each model,
3. A logical format for one type of data (e.g., solar irradiances sorted by wavelength) is clearly inappropriate for other formats (e.g., ionospheric electron density data).

1.5.1.2 Specific Challenges

The space-weather sciences are rapidly evolving and so are the models used. That means that data conversion requirements will often need to be adapted to changing model I/O requirements. The question is who should do that? Is it the task of the scientist that wrote the model and knows it intimately? Or does this job land on the plate of the engineer that maintains the central server that integrates the unique models? Regardless who is doing the job, what about tools to make it easier and faster?

1.6 Proposed Solution

1.6.1 Requirements for Solutions

A solution that solves these challenges requires software that:

1. Is easy to use for people with no specific programming skills. A basic understanding of the model I/O file formats is necessary.
2. Can be quickly adapted to changing data formats (model input or output),
3. Supports the current formats used by the models and the data objects in the IFS. However the formats need to be unambiguous enough to be processed by a machine, in other words some requirements for the file format apply.
4. Is extendable, which means that new conversion functionality can be added without rewriting/recompiling, the existing code by hand.

1.6.2 Vision of an Implementation

In terms of implementation, one of the tools is a GUI web application that can be used to specify data format definitions. The GUI is designed to make it easy to define the rules for a specific data format, not only ASCII file formats from the existing models, but also changed or new data formats. A simple example of how the GUI front end is envisioned is the way the Microsoft Excel wizard guides the user through the process of importing an ASCII file into a spreadsheet. After processing the file with the GUI, the input is then translated into a Data Definition Specification (DDS). Another component of the system generates Java source code from the DDS, which code is then compiled. Finally these new Java classes (performing the actual conversion) are dynamically loaded into the system. This dynamic class loading allows for change in data format without restarting the system.

The GUI system is a web application and the user only needs a browser to access the web application. Thus a platform independent solution is easy to accomplish, since the web application will be implemented in a platform independent way.

1.7 Motivation for Solutions

The two most important reasons for the choice of proposed solution is:

1. To avoid the burden of maintaining a hand-coded collection of conversion classes. Maintenance by programmers is costly, time consuming, and can potentially introduce new bugs.
2. To ensure a fast adaptation to changing formats.

2 Methodology

2.1 Compatibility

The SET developers have a preference for the Java platform as the glue that links the models together. (Note: the models themselves are often written in other languages. Java will be used to wrap these applications). For easy maintenance and use, the code repository of the conversion sub-system should be located on a central server, such that whatever server or client needs the conversion sub-system has a central place to check out updated code. (Note: clients and servers that run the conversion sub-system can be located on any Java capable platform). Secure connections are mandatory. Given the context mentioned above, the Java virtual machine is the environment of choice, since it is one of the solutions that is highly platform independent.

2.2 Programming Environment

The programming environment of choice is Java, however for certain functionality Java is not the most productive language. Therefore Jython will be used in combination with Java to act as middle-ware.

Jython is a pure Java implementation of Python, an object-oriented scripting or programming language.[2] Jython/Python is very extendable and has a remarkably clear and powerful syntax, which is one reason for compact code. The type of objects in Python is resolved at run time rather than at compile time. This eliminates the need for type declarations and or type casts, which is another reason why coding Python is relatively fast. Python is regarded as a language that boasts developer productivity because of its powerful features. Mark Lutz, author of one of the leading Python books, mentions that Python is a language that is designed rather than just accumulated; he says, "…Python emphasizes concepts such as quality, productivity, portability, and integration."[3]

Jython adds the general Python features to the Java platform; it not only extends Java with dynamic scripting, but also allows Java classes to be used inside the Jython code. It is even possible to compile Jython code (classes) into Java class files, either by sub-classing a Java class or by providing the method signatures needed for type resolution at compile time. So real Java class files can be created, such as applets, servlets and beans.

The GUI web application prototype is developed in Zope, a web application framework written in Python.[4] Given that Java is the platform for the middle ware tying the IFS components together, that might come as a surprise.

However, the coupling between the GUI web application and the rest of the system is relatively low, so the benefit of using Zope outweighs the change of environment. One important reason to develop the GUI application in Zope is

[2]*Jython Home Page*, June 20, 2004, <http://www.jython.org/>, (July 3, 2004); *Python Programming Language*, 2004, <http://www.python.org/>, (July 3, 2004).

[3] Mark Lutz, *Programmin Python,* (Sebastopol: O'Reilly, 2001), 1.

[4]*Zope*, 2003, <http://www.zope.org/>, (July 3, 2004).

development speed. In general one can write Python code faster than Java code, which is used in Java Server Pages (JSP).

2.3 Software Development Model

SET already uses an iterative software development model and in this project the development is also done in the spirit of an iterative method. An iterative method is very appropriate for a project like this because it supports feedback and adaptation, while also allowing for adding features and improvements step by step.[5]

2.4 Object Oriented Design and Design Patterns

As many textbooks on modern software development teach, the core concepts of Object Oriented Design (OOD) are encapsulation and information hiding, which facilitate better software designs. OOD makes it easier to combine related functionality into a component that interacts with the rest of the system through a well-defined and usually simple interface. This technique endorses the kind of flexibility that makes changes easier, not only for development but also for maintenance. These principles are referred to as high coherence, which means that a component containing related functionality, and low coupling, which refers to a simple interface between components.[6]

Another tool that is frequently used in OOD is a design pattern. Design patterns have become well known from the book *Design Patterns: Elements of Reusable Object-oriented Software*, in the computer science world also known as the "Gang of Four" (GoF) design pattern book.[7]

In this project two design patterns play an important role in the design:

1. The factory pattern
2. The Model-View separation principle

Both patterns are discussed in the following sections.

2.4.1 Factory Pattern

In his book *Java Design Patterns* Cooper covers the classic GoF patterns from the Java viewpoint and he describes the factory patterns as: "A simple Factory pattern returns an instance of one of several possible classes depending on the data provided to it."[8]

The conversion sub-system is built on this concept. Based on the kind of conversion needed, class instances that together implement this conversion functionality are created and loaded.

This pattern is chosen because it nicely de-couples the conversion component from the main component of the program. The main component only needs to know how

[5] Craig Larman, *Applying UML Patterns: an Introduction to Object-oriented Analysis and Design and the Unified Process,* (Upper Saddle River: Prentice Hall, 2002), 14.

[6] Bernd Bruegge and Allen H. Dutoit, *Object-oriented Software Engineering:Conquering Complex and Changing Systems,* (Upper Saddle River: Prentice Hall, 2000), 174-175.

[7] Erich Gamma et al., *Design Patterns: Elements of Reusable Object-oriented Software,* (Reading: Addison-Wesley, 1995).

[8] James W. Cooper, *Java Design Patterns: a Tutorial,* (Boston: Addison Wesley, 2000), 19.

to find and load the desired conversion component, it does not need to know anything about how conversion is done.

The dynamic code generation that creates the classes for the conversion component is not following a classic GoF design pattern, although one might consider the way the code generation is used is a cousin of the GoF template method pattern. The code generation indeed subclasses a base class and provides more functionality specific for the task it is intended for. The difference with the usual situation is that normally a programmer writes the subclass, while here another program does that job. However that program follows a certain pattern, or in other words a template.

2.4.2 Model-View Separation Principle

The Model-View separation principle comes from the key concept of the Model View Controller (MVC) pattern, which predates the book from the GoF.[9] The concept of MVC goes back to the mid-eighties and was initially closely related to the programming language Smalltalk, especially the GUI components and in an article from before the GoF book the MVC-Smalltalk relation is mentioned.[10] But the authors don't talk about MCV as a design pattern, because the term design pattern apparently wasn't coined until later.

The benefit of the Model-View separation principle is that changing one of the components has little repercussions on the other component. For instance the screen (View) can stay the same if a calculation method for data to be displayed in the screen changes. The only thing that matters is that the format stays the same.

As mentioned earlier the GUI web application is developed in Zope, which makes is relative easy to apply a Model View separation type of design. Zope's design of the Zope Page Templates (ZPT) encourages a low coupling between the business logic and the presentation logic.[11] The ZPT represent the View or the presentation logic and a ZPT contains XHTML with special attributes that can call Python code, which represents the Model or the business logic. The Python code does not need to know about how to display its results, that is the task of the ZPT and, in turn, a ZPT does not need to know how to create the results to display.

2.5 Resources

2.5.1 Software Resources

This project uses free available software:

1. A recent stable Java SDK from Sun, J2SDK 1.4.2_04 is the current version.[12]

[9] Larman 2002, 471.

[10] Emil F. Girczyc and Tai Ly, "Stem: an IC Design Environment Based on the Smalltalk Model-View-Controller Construct", *Proceedings - 24th ACM/IEEE Design Automation Conference*, 1987, IEEE, 757-763.

[11] Amos Latteier et al., *The Zope Book (2.6 Edition)*, 2003, <http://zope.org/Documentation/Books/ZopeBook/2_6Edition/ZopeBook-2_6.pdf>, (July 3, 2004), 128.

[12] *Java 2 Platform, Standard Edition (J2SE)*, 2004, <http://java.sun.com/j2se/>, (July 3, 2004).

2. Jython 2.1, an implementation of Python in Java, which is open source.[13]
3. A UML program, like ArgoUML v0.14, an open-source Java program.[14]
4. A Java Integrated Development Environment (IDE) like Jbuilder 9 personal from Borland (available for Windows, Solaris and Linux) or Netbeans version 3.6, a Java application from Sun.[15] Netbeans has also Jython support.
5. A recent version of SciTE, currently that is version 1.61, SciTE is a very flexible open source programmers editor with simple IDE like features.[16] Available for Unix (with X windows and GTK2) and Windows.
6. A recent version of Zope, currently Zope is at version 2.7.[17] Zope is an open source web application framework somewhat similar to JSP in functionality.

2.5.2 Hardware Resources

All hardware resources used for this project were already available:

1. Initially developing from home on a dual boot AMD Athlon system, which has a Unix universe (well, actually Linux land) and a Windows world (2000). Additionally using a laptop running Windows 2000 and Linux for developing distributed components. Both machines are networked.
2. Using a remote web server (third party service provider) for developing the GUI web application.
3. Deploying the prototype on a server running the IFS development project, a Mac OS X server.
4. Utilizing a high speed internet connection.

[13] *Jython.*

[14] *ArgoUML Project Home*, 2003, <http://argouml.tigris.org/>, (July 3, 2004).

[15] *Jbuilder The Leading Development Solution for Java*, 2004, < http://www.borland.com/jbuilder/ (July 3, 2004).

[16] *SciTE A Free Source Code Editor for Win32 and X*, May 29, 2004, <http://www.scintilla.org/SciTE.html>, (July 3, 2004).

[17] *Zope*.

3 System Design

This chapter contains the system design.[18] It only covers the parts of the system that are directly related to the topic of this thesis project. The high level requirements were rather clear from the beginning, but filling in the details turned out to be a search for the best approach. During the project development it became clear that the initial direction of the design and implementation of the system should be changed, and in hindsight it was a change for the better. The requirements and the design presented here are a reflection of these changed insights.

3.1 Concepts and Definitions

For a better understanding some concepts and definitions are given. Some of them are rather generic, but some are more specific to this project and/or the IFS.

3.1.1 Client and Server Concept

3.1.1.1 Client Concept

A client in the IFS:

1. Is a stand-alone computer or virtual top-level process that interacts with the central server.
2. Is responsible for executing a model, acquiring data, or disseminating data that is related to a particular model or measurement.
3. Executes processes that can exist independently from the central server (for instance in a test mode), but for operations must communicate with the server for all data I/O (except when it has the key responsibility for acquiring data from an instrument or separate data source).
4. Has a single point-of-contact (POC, for example a researcher) that is responsible for the definitions, operations, and so forth.
5. Has a unique URL/IP address.
6. Has software that must be maintained (namely configuration and control management) as part of the larger system.

3.1.1.2 Central Server Concept

The central server or “client server” handles all the data IO from and to the database, that is the central server receives the data from the model output or third party data and serves requests for data coming from models and end users.

3.1.1.3 Repository Sub-system

The repository sub-system serves as place for all centrally maintained code:

1. It is the place where to check for new DDS files and new dynamically generated classes, depending on how the Conversion Engines are set up.

[18] W. Kent Tobiska and Dave Bouwer, Specific information about the IFS in Chapter 3 is a summary of personal communication and internal SET documents, February 2004.

2. It is also the place to check for client software and dynamic class generation code.

3.1.1.4 Conversion Engine Concept

The conversion engine is responsible for the conversion between data suitcases and model input/output or converts a data suitcase into an output file for an end user.

3.1.2 Model and Data Concept

3.1.2.1 Model and Acquired Data

A model:

1. Is the primary source for data files, together with data acquired from outside the system (actual definitions).
2. Runs as a process on a client computer (remote) or on a compute engine (local).

3.1.2.2 Data Suitcase

A data suitcase is:

1. An object used to store model data along with functionality to manipulate that data,
2. A "self describing" entity,
3. An element of the internal data representation of the system.

3.1.2.3 Data Definition Specification (DDS)

A Data Definition Specification is a file that describes how source code can be generated that converts to/from a structured ASCII file from/to a data suitcase. These ASCII files are usually a model input file or output file or an external data source.

3.1.3 Responsibilities

In this section the role of "maintainer" is defined. Although the researcher that is responsible for a particular model might perform this role, it is also possible that this role sooner or later will be performed by one of the system administrators.

3.1.3.1 Maintainer Responsibility

The maintainer is responsible for queuing up sample ASCII files, to be used with the GUI, and the maintainer operates the GUI in order to:

1. Define records, fields (intended definitions).
2. Specify validation criteria.
3. Document data.
4. Store the resulting DDS and code in the right place (Repository).

3.1.3.2 Central Server Responsibilities

The responsibility of the Central Server is to receive data from model I/O and to create "Just-In-Time" output upon requests from clients, models or end users. To fulfill these requests the central server needs the assistance of the conversion engines.

3.1.3.3 Repository Sub-system Responsibilities

The repository sub-system is responsible for the following tasks:

1. Facilitating the check on the availability of a new DDS.
2. Serving a DDS or a new dynamic class upon request.
3. Serving client software or new dynamic class generation code upon request.

3.2 Requirements

The requirements are expressed as narrative use cases. This form is chosen over UML use case diagrams and function specifications because it is probably more instructive.

3.2.1 Use Cases for Conversion from Model Output to Data Suitcase

3.2.1.1 Create Data Definition Specification (DDS) for Model Output

Conversion from the model output file into a data suitcase is the task of a conversion engine. However since model output files are expected to change relatively often, the conversion engines need to be flexible. This flexibility can be achieved by dynamically creating pieces of the conversion software. The IFS GUI is meant to facilitate the dynamic construction of conversion code by "translating" the structure in a model data file in source code in a visual, "wizard like" way.

In order to build a data object from a model file the conversion software needs information such as field names, beginning of the data table and delimiters used.

The idea is that the IFS GUI web application reads in the crucial parts of the file, such as beginning of the file, and starts "asking questions". The questions have a suggested answer based on a guess. A guess can be made because the input file has some structural patterns often used in IFS models output files.

After finishing the wizard, the user can perform a test to check validity of the created data definitions and generated source code.

The use case for creating a DDS for model output is as follows:

Primary actor:	GUI user (maintainer responsible for the maintenance of the conversion DDS)
Stakeholders and Interests:	GUI user, researcher, developers
Preconditions:	The (changed) model is operational. The IFS GUI system is available. Researcher responsible for model has prepared a sample ASCII file that is representative for the model's output.
Main Success Scenario:	1. GUI user uses the IFS GUI sub-system with this valid sample file to perform the following tasks: 1.1. Locates and defines the table header line in the input file. 1.2. Adds, if desired, additional meta-data in the form of tag-value pairs. 1.3. Specifies the field delimiters and or makes sure the proper number of columns is set. 1.4. Defines field names and field types, indicates which fields present in the sample file to include in the data suitcase. 1.5. Specifies, if used, validation criteria such as upper and lower boundaries. 1.6. Gives replacement values for "bad" records or indicates that median values should be used. 2. User test the just created data definition specification (DDS), see section 3.2.1.2. 3. User saves the DDS and conversion code in the repository sub-system.
Post conditions:	The new DDS and conversion code is available at the repository sub-system.
Special Requirements:	IFS GUI system is a web application, requiring only a browser as client.
Issues:	How to link the GUI system to the repository system.

3.2.1.2 Test Parser Code Created from a New DDS

Automation not only reduces work, but maybe its biggest value is the increased reliability. Before allowing the generated code to be used, a test needs to be done. The dynamic generation of parsing code also requires generating dynamic testing because static testing would defeat the benefit of the dynamic creation of code.

The use case for testing the parser code generated from a new DDS:

Primary actor:	System, back-end of GUI web application
Stakeholders and Interests:	Researcher, developers
Preconditions:	The GUI user has just created a new DDS from a sample file.
Main Success Scenario:	1. GUI user clicks the "Test" button. 2. Conversion source code is generated. 3. Source code is compiled. 4. DDS file is generated. 5. Test files are generated (test input files and expected values files). 6. The test suite is run, the test parses a test file and creates a test data suitcase for each test case and this suitcase is the tested against the expected values. 7. Test results are displayed on the GUI screen.
Post conditions:	The GUI user is informed about the test results.
Special Requirements:	None
Issues:	None

3.2.1.3 Create New Data Object from a Model Output File

The ConversionEngine is envisioned as a sub-system that can be on the central server or at the client that runs the model. The ConversionEngine is activated when a file needs to be processed.

The use case for converting a file into a data suitcase:

Primary actor:	System, ConversionEngine for input (file to data suitcase)
Stakeholders and Interests:	Researcher, Developers
Preconditions:	Model file is available.
Main Success Scenario:	1. The ConversionEngine checks for new conversion code, if so download this code. 2. The ConversionEngine creates an instance of the proper Parser class (Factory/Builder pattern and dynamic class loading). 3. The Parser instance performs following tasks: 3.1. Parser add meta-data to data suitcase 3.2. Parser parses file and fills the with data while validating data set. 4. ConversionEngine serializes and compressed data suitcase. 5. Data suitcase is uploaded and stored in database.
Post conditions:	Data suitcase is created and stored in database.
Special Requirements:	None
Issues:	How to link the ConversionEngine with the model.

3.2.2 Use Cases for Conversion from Data Suitcase to Model Input

3.2.2.1 Create DDS for Model Input or Client Request

This use case has two slightly different uses. One use is to create a DDS for automated conversion of model input files. The other use is to provide end users with a means to create a file from a data suitcase and determine the format by defining a DDS themselves just for the occasion, or multiple occasions.

The use case for creating a DDS for model input is as follows:

Primary actor:	GUI user (end user or maintainer responsible for the maintenance of the conversion DDS)
Stakeholders and Interests:	GUI user (maintainer or end user), developers
Preconditions:	The IFS GUI system is available.
Main Success Scenario:	1. GUI user uses the GUI sub-system to perform the following tasks: 1.1. Selects data suitcase type to build file from. 1.2. Indicates the mapping between columns of the table in the file and fieldnames in the data suitcase. 1.3. Adds, if desired, additional meta-data in the form of tag-value pairs. 1.4. Specifies the field delimiters. 1.5. Specifies the print format of fields 2. User specifies validation criteria such as upper and lower boundaries, if out of bound values needs to be replaced. 3. User gives replacement values for out of bound values, or indicates that median values should be used as replacement values, in case replacement for out of bound is used. 4. User test the just created data definition specification (DDS), see section 3.2.2.2. 5. User saves the DDS and conversion code in the repository sub-system. Alternative flow: Instead of saving the DDS in the repository, the DDS can be used specifically for a request of a file by an end user. The end user is in this case the one who creates the DDS and from that DDS a one time FileGenerator class is created and used to produce the file.
Post conditions:	The new DDS and conversion code is available at the repository sub-system.
Special Requirements:	IFS GUI system is a web application, requiring only a browser as client.
Issues:	How to link the GUI system with the Repository system and in case of the "end user" part of this use case, with the Central Server.

3.2.2.2 Test File Generation Code Created from a New DDS

In a similar way as the use case in section 3.2.1.2 this use case covers the testing from the IFS GUI sub-system:

Primary actor:	System, back-end of GUI web application
Stakeholders and Interests:	Researcher, developers
Preconditions:	The GUI user has just created a new DDS from a sample file.
Main Success Scenario:	1. GUI user clicks the "Test" button. 2. Conversion source code is generated. 3. Source code is compiled. 4. DDS file is generated. 5. Test files are generated (test input files and expected values files). 6. The test suite is run, the test parses a test file and creates a test data suitcase for each test case and this suitcase is the tested against the expected values. 7. Test results are displayed on the GUI screen.
Post conditions:	The GUI user is informed about the test results.
Special Requirements:	None
Issues:	None

3.2.2.3 Create New Model Input File from a Data Object

The ConversionEngine is envisioned as a sub-system that can be on the central server or at the client that runs the model. The ConversionEngine is activated when a file is requested.

The use case for converting a data suitcase into a file:

Primary actor:	ConversionEngine
Stakeholders and Interests:	Researcher, Developers
Preconditions:	Data Object is available after request to central server.
Main Success Scenario:	1. The ConversionEngine creates an instance of the proper FileGenerator class (Factory/Builder pattern and dynamic class loading). 2. This FileGenerator generates the file from data suitcase. 3. File is send to requesting client.
Post conditions:	Requesting client acknowledges the receipt of the file
Special Requirements:	None
Issues:	How to link the ConversionEngine with the model

3.2.3 Nonfunctional Requirements

3.2.3.1 Requirements for the Input ASCII Files

To facilitate the model developers it is required that the ASCII file format is in a sense "free form" with as little structure as possible. The bottom line is that the file must be unambiguous enough in order to make parsing possible. The ASCII input file is also the defining source for a data suitcase.

Therefore the structure in input file is required to facilitate the extraction of:

1. Meta-data (just as a string or set of strings is sufficient),
2. The field names,
3. The field values and obviously a way to tell one record apart from another.

Without defining the structure here, one possible way to structure the input file is as follows:

1. Separate the meta-data with a table header from the actual table data, this table header also contains the field names.
2. Use a delimiting character for the record, a new line character will do nicely.
3. Use a delimiting character for the fields like a space or tab character.

3.2.3.2 Performance Requirements

Although there is currently no strict performance requirement, it is important that conversion related operations are done fast enough so the intended cadence of the model execution cycles will not be disturbed.

3.2.3.3 Security Requirements

To facilitate reliable operation the network traffic is envisioned to happen over secure channels. The local network is considered secure in this context.

3.2.3.4 Software Quality Attributes

Aside from correctness, maintainability, and reliability, here the flexibility in the design is one of the most important attributes of the software design. Since the models are under constant development, their input and output requirements will change over time too. Design for change is one of the driving forces behind this part of the project.

3.2.3.5 Design for Change

Building for changes requires ease of update, which will be supported by central code repositories. Also the design needs to be such that sub-systems can be easily linked together, while maintaining a low level of coupling. The idea is to provide the sub-systems with means to communicate with each other over network connections according to standard protocols, which allows for flexibility regarding the physical location of the sub-systems.

In short these core points are envisioned to facilitate a design for change:

1. A well defined but flexible internal format for data: data suitcases

2. A fast and easy procedure to adapt to changes in model I/O, the core topic of this project
3. Central maintenance of code base

3.3 Architecture

The architecture of the IFS is envisioned as in the following diagram, Figure 2.

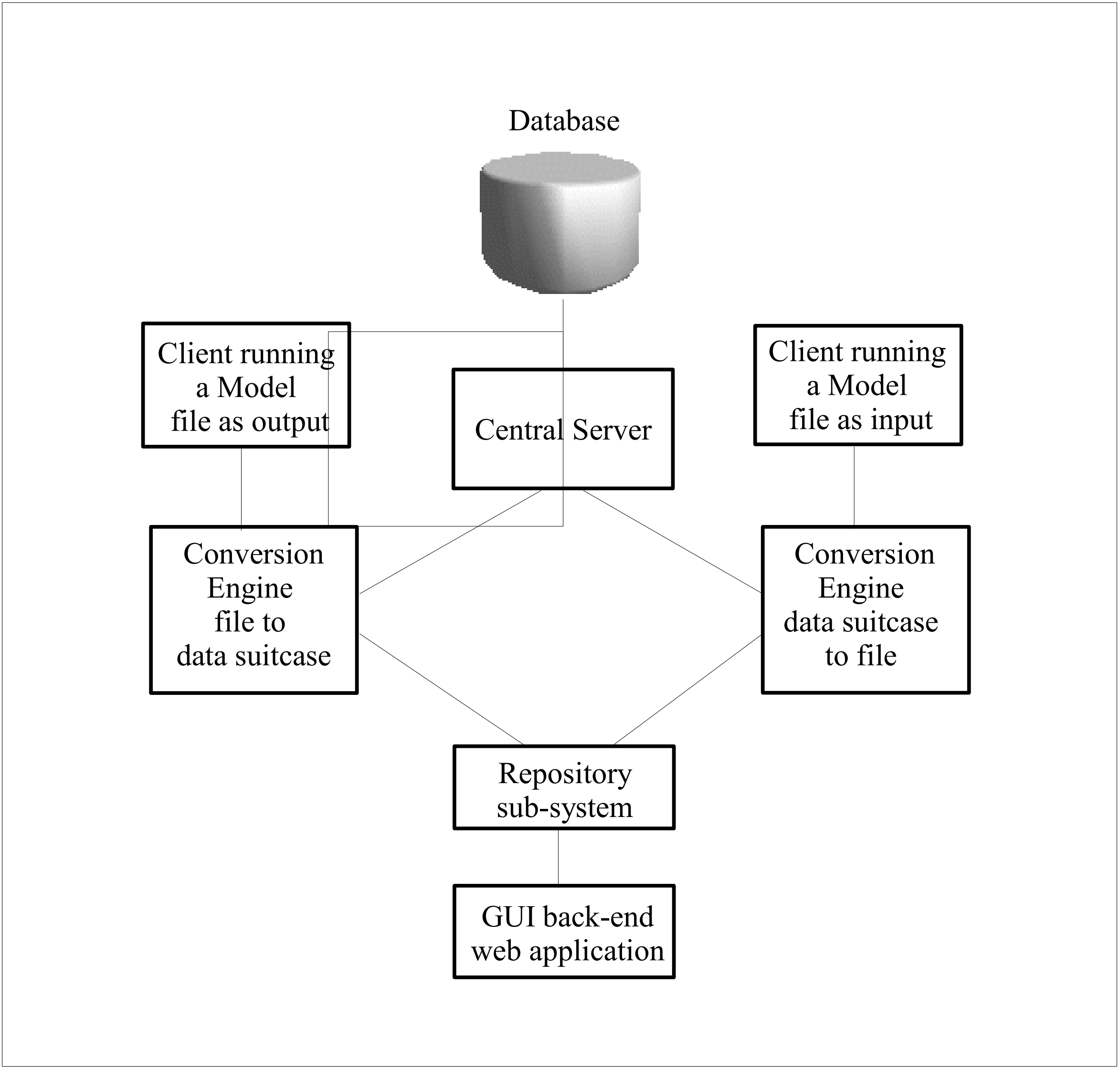

Database
Client running a Model file as output
Central Server
Client running a Model file as input
Conversion Engine file to data suitcase
Conversion Engine data suitcase to file
Repository sub-system
GUI back-end web application

Figure 2: Overview of Sub-systems.

The architecture of the IFS consists of a number of subsystems:

1. The sub-systems that are basically outside the scope of this thesis project, such as the database and compute engines (except that the compute engines are clients running models and consume or produce data files), are not discussed any further.
2. The "client server," the server in the IFS setup that controls the I/O for the database.
3. The back end of the GUI system, a web application server that generates DDS files (and probably the Java source code) as output based on user input. Normally these files are to be stored in the repository sub-system and used by the conversion engines.
4. The repository sub-system containing the current DDS files and probably Java source files. There is a choice here: either the dynamic classes are generated centrally in the repository sub-system, or the generation of code is part of the conversion engine itself. From the perspective of the conversion engine the biggest difference is what to check for, a new DDS file (or possibly Java source) or for a new class. One of the uses of the conversion engine is by an end user creating a DDS (and conversion classes) for only one occasion. In that case it looks more natural having the dynamic class generation code live inside the conversion engine itself. However the conversion engines need to check for two things, not only for a new DDS, but also for changes in the code that performs the dynamic class generation.
5. The conversion engines, which are actually two sub-systems.

3.3.1 Interaction between Sub-systems

The conversion engines, the GUI back end, the repository sub-system and the "client server" can potentially be on different server nodes in a network. One physical layout of the sub-systems, Figure 3, shows conversion engines at clients' location.

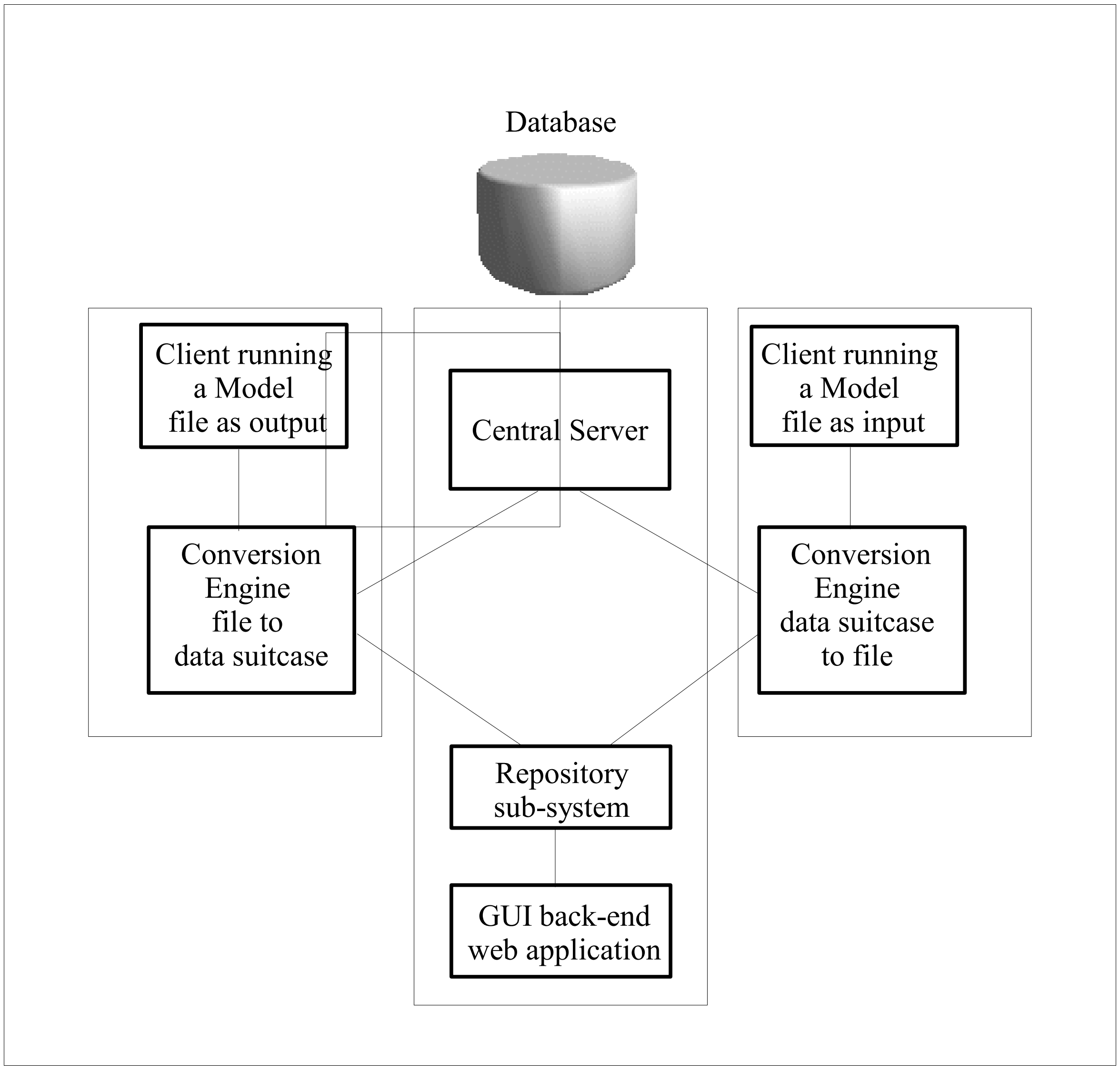

Database
Client running a Model file as output
Central Server
Client running a Model file as input
Conversion Engine file to data suitcase
Conversion Engine data suitcase to file
Repository sub-system
GUI back-end web application

Figure 3: Overview of Sub-systems, Conversion Engines on Clients

Figure 4 centralizes everything except the clients, which can be remote.

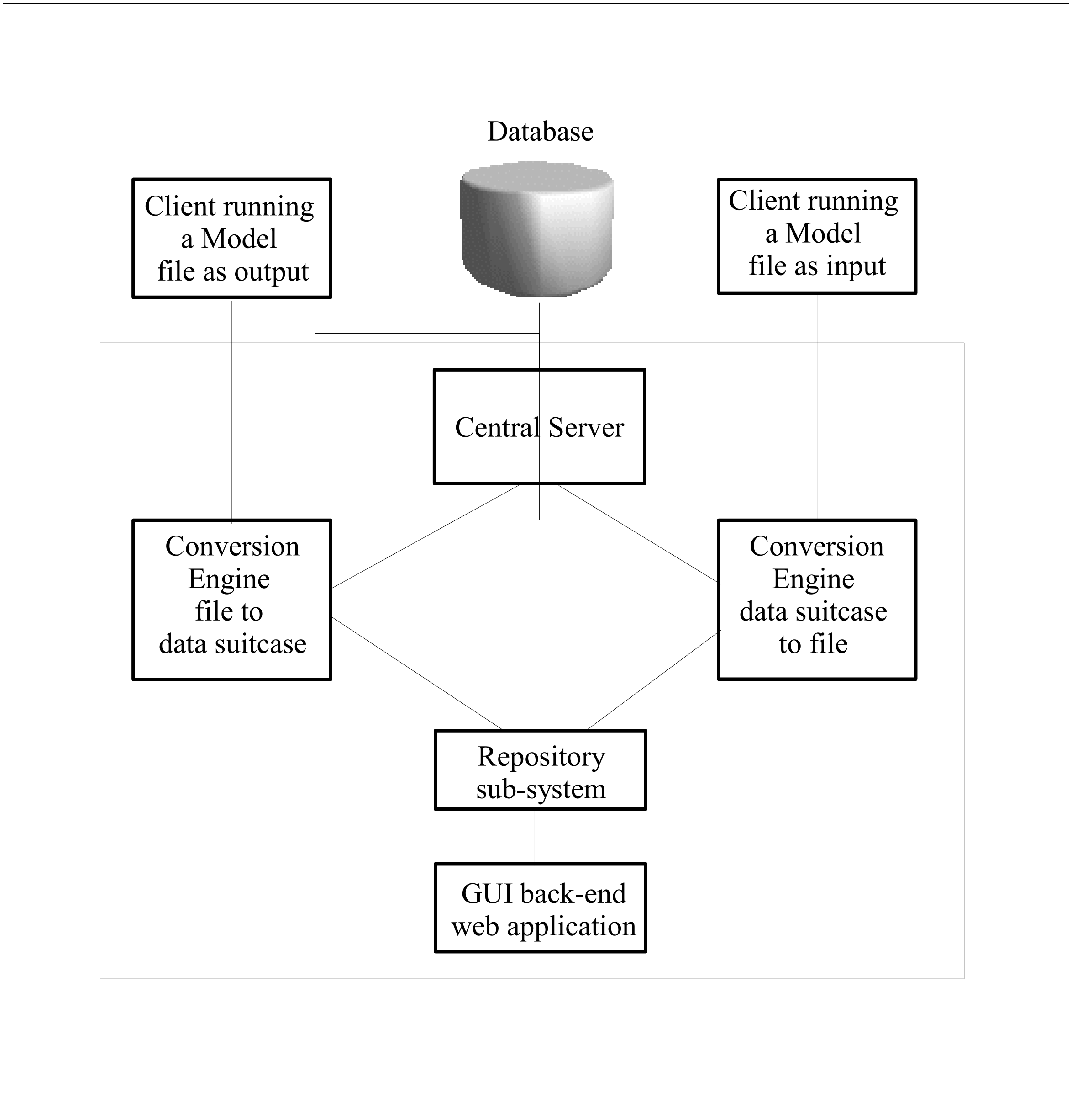

Database
Client running a Model file as output
Client running a Model file as input
Central Server
Conversion Engine file to data suitcase
Conversion Engine data suitcase to file
Repository sub-system
GUI back-end web application

Figure 4: Overview of Sub-systems, Conversion Engines on Central Server

If sub-systems are on different server nodes in a network, then there is a need to decide on communication protocols between the sub-systems.

The flow of data in the IFS goes from/to models to/from the central server, with conversion engines sitting in between the models and the central server, usually as a link in an automated system allowing the following characteristics:

1. The conversion engine that creates a data object from a file is triggered as a model or external data provider produces a file. So a file is received and a data

object is sent. Since the data object is serialized and compressed, the data object is essentially a file. Yet little more than moving files around is needed, the conversion engine needs to do some query for a possible new DDS and/or Java parsing code, so some form of message-passing is required.

2. The conversion engine that does the opposite action, data object to file, needs basically the same functionality, since a data object coming out the database is a serialized and compressed object in the form of a file. Here a message-passing requirement also applies because a request for a data object will probably be in the form of a SQL query in order to get the correct data object converted.

The conversion engines need to be able to check for new DDS files and for updates of the code that generate the dynamic classes (or only the dynamic Java classes depending where the code generator lives), which is also exchanging files and messages.

Thus all the communication of the conversion engines boils down to message-passing and the sending/receiving of files.

The GUI back end produces DDS files (and probably Java source files), which are sent to the repository sub-system. And again some message-passing is needed to keep track of what is happening.

The bottom line in all cases is that two things are needed:

1. For uploading and downloading of files a simple FTP or copy protocol will suffice. Secure copy is preferred (SCP) if the link includes an insecure network.
2. For the message-passing XML-RPC over HTTP (with SSL in case of an insecure network) is suitable, XML-RPC (see section 4.2 for more details) is universal, yet not very fast for passing around big chunks of data; a simple (secure) copy is much faster for that purpose. In case both sides are Java programs, (secure) RMI is an alternative too.

Although plain sockets can be used, it is rather tedious to implement and probably not worth the gain in performance.

At this stage in the design no hard choices are made how to implement this connectivity. The master daemon on the "client server" and the conversion engines are intended to be a Java based implementations, thus the communication between them can be any of the choices mentioned above. For communication between the conversion engines and the models, things depend on the capabilities of the models and/or possibly needed glue software.

There is another issue that needs attention. Which piece of the system is responsible for starting the conversion? In the overall design the central server plays a somewhat passive role (as usual for servers, waiting to fulfill requests) and only accepts data output from models if it is presented and also only responds to requests for data. So the conversion engines' connection towards the central server is clear: the conversion engines act as a client and initiate the connection. But what is the situation at the other end? That depends on the model software. Do models just dump their output in a directory? Then the conversion engine needs to be responsible for picking up the newly produced files. This can be done by creating some wrapper code around the model and make the wrapper responsible for the handling of the I/O files.

The models that need input from the system are most likely a little more active. After all they need the input to work, so there will be some sort of mechanism to request data.

In general the most natural situation would be that the conversion engines act as servers towards the models, which possibly requires wrappers around the models.

The GUI back end is implemented in Zope/Python right now (see 3.4.2 for design details). That means that RMI is not possible without some glue software that makes RMI available as an external program. In Zope XML-RPC can be supported almost "out of the box" and secure copy is also possible by calling an external program.

3.4 Software Design

This section covers the data design, the design of the GUI web application and the conversion code.

3.4.1 Data Design

3.4.1.1 ASCII Input and Output Files

It is desirable to allow the ASCII input file format as "free form" as possible, but still unambiguous enough to be able to parse the file. For that reason there are "hard rules" and "desirable rules".

Hard rules are:

1. The delimiter for the records in the data table must be one of the common newline codes (UNIX: LF, Mac: CR, Windows: CR + LF), preferably the platform newline (it is possible store the platform newline in the DDS, but that is probably unnecessary).
2. The fields in the data table must either be separated by a unique delimiting character (delimiter) or have a fixed field width.
3. When using a space as delimiter, the record still must be unambiguous, which means that only the last field is allowed to have spaces.
4. There must be a line with the column headers for the table, where the column headers follow the same rules for separating the fields in the table. However there are spaces allowed in the field names since it is possible to accommodate this ambiguity in the GUI web application.

Desirable rules:

1. Preferably use "whitespace" as delimiter, where whitespace is defined as one of more tabs and/or one or more spaces. Note: there are definitions of whitepace that include newlines, however newlines are excluded in this definition of whitespace.
2. Preferably use a unique delimiter if the table in the file consists of delimited fields.
3. Preferably avoid using spaces in the field names.
4. Preferably use a space between fields when using fixed width fields, although fixed width strictly does not require a space between the fields, it does make detecting the width of a column in the GUI web application much easier.

Note: currently this space is actually a requirement. An extension of the GUI can make it a preference.

```
                   SOLAR2000 proxy table
RESEARCH GRADE V2.22/256E19

YYYY-DDD/hh:mm:ss DD-MMM-YYYY/hh:mm:ss  Julian day   F10.7  F-81     Ly-a     Ly81
1948-001/12:00:00 01-JAN-1948/12:00:00 2432552.00000 200.5 165.2 5.32E+11 5.09E+11
1948-002/12:00:00 02-JAN-1948/12:00:00 2432553.00000 208.1 164.2 5.29E+11 5.09E+11
1948-003/12:00:00 03-JAN-1948/12:00:00 2432554.00000 179.3 163.1 5.24E+11 5.08E+11
1948-004/12:00:00 04-JAN-1948/12:00:00 2432555.00000 169.2 162.1 5.16E+11 5.07E+11
```

Note: the shaded area represents the table header; the area above the header is usually meta-data and the area under the header is the table data.

Figure 5: Example Snippet of an ASCII Input File.

3.4.1.2 Data Definition Specification (DDS)

Based on the requirements for the input file mentioned in the previous section the design of GUI web application was implemented. The DDS produced by the input wizard contains the following information:

1. Properties of the data table, such as the header line text (needed to determine the table start), the means of separating the fields (delimiter or fixed width).
2. The field properties, such as field name, type or format of the field, an upper boundary value and a lower boundary value (if "out of bounds" is used, the field is flagged as out of bound in the data suitcase). Other field properties include a replacement value for "bad data", and an indicator whether to include or exclude the field in the data suitcase (date-time fields are always automatically included, so these can be marked as excluded, otherwise they will be redundant).
3. The date step size. This value determines the proper interval between the date-time of each record.
4. A hash table object that holds meta-data added while creating the DDS, the meta-data is stored as tag-value pairs, or key-value pairs in software engineering parlance.
5. The bucket size. A bucket is a reasonable subset of the records in a data suitcase.
6. A flag to determine whether the median value of the fields in the bucket is used as the replacement value for a bad field value, like missing value or wrong type. The other choice is to use the replacement values provided in the field properties.
7. A flag that determines whether the boundary values of the fields in the data set are checked or not checked for extreme values.

The DDS from the output wizard looks very similar. But of course now the source is a data suitcase and the target is an ASCII output file.

Although the same kinds of properties are present, the function of the properties is to help generate a file, not to parse one. The list is as follows:

1. Properties of the data table, such as the header line text (needed to write out the column headers), the means of separating the fields (delimiter or fixed width).
2. The field properties, such as field name and type, as well as a format for writing out the field. Other properties are an upper boundary value and a lower boundary value and a replacement value. If boundary values are used, out of bound values will be replace by the replacement values. Finally the field properties have an indicator whether to include or exclude the field in the output file. If desired a field can be included more than one time.
3. A hash table object that holds meta-data added while creating the DDS: the meta-data is stored as tag-value pairs, or key-value pairs in software engineering parlance.
4. A flag that indicates whether or not to include the meta-data stored in the suitcase as well as a flag that indicates whether or not to include the meta-data that was added to the suitcase at creation time.
5. A flag to determine whether the median value of the fields in the bucket is used as the replacement value for a out of bound field value, like missing value or wrong type. The other choice is to use the replace values provided in the field properties.
6. A flag that determines whether the boundary values of the fields in the data set are checked or not checked for extreme values.

3.4.1.3 Data Suitcase

Based on the findings that serialization in Java is an expensive operation, it makes sense to make the serialization of a data suitcase a more layered operation (unless a serialization implementation is found that is about 10 times faster then the standard implementation). De-serialization does not seem to be as slow as serialization though.

After a discussion with the IFS development team, one of the developers came up with an idea how to approach the layered structure of the data suitcase: use a set of containers in a container. Or in IFS terminology, a data suitcase contains one or more buckets, which contain records.

This design idea is detailed as follows:

1. There is a record class that holds the fields, such that fields are retrieved by fieldname. The record class has also a set of indicators reflecting the validity of each field and also a validation indicator for the record as a whole (one “out of bounds” field = whole record marked as “out of bounds”, or mark the record as having replacement values to replace “bad” values). Finally there is a parameter that indicates the state of the date-time value, whether this record represents a jump in the sequence or is normal.
2. There is a bucket class. The bucket holds certain time span’s (year, month, day) worth of records, as well as similar validity indicators and date states as the record class. The bucket has the median values of the records it contains stored as a separate attribute.
3. There is a suitcase class. This class holds the field properties and the meta-data as well as the buckets. The buckets are stored serialized (and can be compressed) in the suitcase. The main reason for this is to allow fast querying

of the suitcase because for querying the data suitcase needs to be de-serialized, but the buckets are untouched, they stay serialized. The suitcase has a method that allows for extracting a subset of buckets wrapped in another suitcase but only containing the subset. In this way it does not make any difference for the conversion subsystem whether the consumer asks for one bucket, ten buckets or all buckets. The conversion engine just sees a suitcase containing one or more buckets.

3.4.1.4 Indicators for Data States

To evaluate the value of the data in a suitcase the suitcase contains indictors at all levels. This way it isn't necessary to drill down to the deepest layer if only "perfect" data is acceptable. See Tables 1–3 on next page.

Tables 1–3 give the meaning of the various indicators (here called flags).

Table 1: Record Class Flags
These flags are attributes in the Record class.

Field validationFlag	Record validationFlag	Record dateFlag
1 = OK 2 = above upper boundary 3 = below lower boundary	1 = OK 2 = not used 3 = not used 4 = field values replaced 5 = at least one field out of bounds 6 = no bound check performed	1 = OK 2 = jump in date

Table 2: Bucket Class Flags
These flags are attributes in the Bucket class.

Bucket validationFlag
1 = OK 2 = not used 3 = not used 4 = at least one record with values replaced 5 = at least one record with fields out of bounds 6 = no bound check performed 7 = one or more records with flag 4 and one or more with flag 5/6
Bucket dateFlag 1 = OK 2 = at least one record with jump in date

Table 3: Suitcase Class Flags
These flags are attributes in the Suitcase class.

Bucket validationFlag	Suitcase validationFlag
1 = OK 2 = not used 3 = not used 4 = at least one record with values replaced 5 = at least one record with fields out of bounds 6 = no bound check performed 7 = at least one record with flag 4 and one with 5/6	1 = OK 2 = not used 3 = not used 4 = at least one bucket with replaced record values 5 = at least one bucket with fields out of bounds 6 = no bound check performed 7 = at least one bucket with one record with flag 4 and one with 5/6
Bucket dateFlag 1 = OK 2 = at least one record with jump in date	Suitcase dateFlag 1 = OK 2 = at least one bucket with “jump date” record

3.4.2 GUI Web Application Design

The GUI web application is created with the web application framework Zope.[19] Zope is an object oriented framework and has a number of build in object types, such as container objects (no surprise that they are called folders), file and image objects, Python scripts and the Zope Page Templates (ZPT).[20] Folders not only act as containers but also form a name space. Web pages are composed out of the Zope objects and Zope renders these objects in order to serve the content as web pages. These Zope objects are located inside an object database and standard Zope objects cannot access the file system. Python scripts inside Zope have restricted power too, no access to the `eval` function and no regular expressions for instance.

However Zope is very extendable, in principle in three different ways:

1. The most powerful way is writing Python modules that inherit from important Zope core classes in order to facilitate tight integration. These extension modules (called Products in Zope terminology) can be used from Zope as if they were just build-ins. Since the code of Products is located on the file system, they can be written such that they can access the file system.
2. A much simpler way to extend Zope is the use of a so called External Method, which has access to the file system. The actual code of an External Method lives in a Python module (file) as a regular Python function. The only two special things are:
   - 2.1. The module needs to be in a specific directory inside Zope.
   - 2.2. Inside the Zope object machinery there is an object that act as the doorway to this External Method, just as an alias.

Interacting with other processes is limited. Standard Zope does not allow starting a process from an external method. A work around is to use an XML-RPC client and server, where the server runs the new processes. This is described in the section.4.3 Demo Extension to GUI System.

The two core elements from Zope that are used in this application are the Zope Page Template (ZPT) and Python scripts. The ZPT's are the presentation layer and Python takes care of the business logic.

ZPT's are valid HTML with additional attributes, which are used for interaction with the Python scripts. Zope renders the ZPT's before sending the pages to the requesting browser and during this rendering the instructions contained in the page template attributes are executed.

Here is a snippet of example code in a ZPT:

```
<p tal:content="here/hello">Hello Example</p>
```

The `tal:content="here/hello"` attribute will replace the text “Hello Example” with what the Python script `hello` returns. Assume the Python script `hello` reads:

```
return "Hello World!"
```

Then after rendering the ZPT snippet above Zope will send `<p>Hello World!</p>` to the browser.

[19] *Zope*.
[20] *Python*.

The URL `"here/hello"` is a way to indicate the name space (the `here` part), which defaults to the container in which the ZPT lives (there is more to the name space story but that is left out the discussion for now). Thus the Python script `"hello"` is normally (normally, but not always) in the same folder as the ZPT.

Of course there are other tal keywords (`tal` is a actually a namespace) as well:

1. `define`, for defining a variable to be used inside the tag and nested tags
2. `condition`, a condition whether to render the tag or not
3. `repeat`, repeating a tag
4. `content` or `replace`, content is filling the tag's content, replace replaces the whole tag
5. `attributes`, setting attribute values of a tag
6. `omit-tag`, for omitting the tag in the output

Here is an example how to use ZPT's:

```
<html>
    <head>
    </head>
    <body>
       <table tal:define="mylist python:[1, 2, 3, 4, 5]">
            <tr tal:repeat="item mylist">
                <td>This is </td>
               <td tal:condition="python:item%2 == 0"
          tal:attributes="bgcolor python:'FFFFC6'">
                    <span tal:content="item"
          tal:omit-tag="">#</span></td>
               <td tal:condition="python:item%2 != 0"
          tal:attributes="bgcolor python:'FFFFFF'">
                    <span tal:content="item"
          tal:omit-tag="">#</span></td>
            </tr>
        </table>
  </body>
</html>
```

There is one new element in the template above, `python:` followed by a Python expression, which shows that it is possible to use Python expression in ZPT's as well.

The rest shows the use of the tal keywords in a simple way. Although there are two `td` tags for the number only one at a time is rendered because of the condition, the even numbers render with a gray background, the odd numbers with a white background. The `span` tag is only used to demonstrate the `omit-tag`.

The output of this rendered ZPT is:

```
This is 1
This is 2
This is 3
This is 4
This is 5
```

The GUI web application is structured as a wizard, a series of screens where on each of them some value is set or a choice is made. Most of the installer programs on Windows are examples of wizards. The MS Excel import wizard, used to import a table from an ASCII file, was the inspiration for the wizards in this application.

The idea behind a wizard is to break down a more complex task into a set of simpler sequential steps. At the end all the settings are done and in the case of these wizards, the Data Definition Specification is complete and the Java source code can be created.

The following two figures, Figure 6 and 7 show the sequence of events for the input and output wizard.

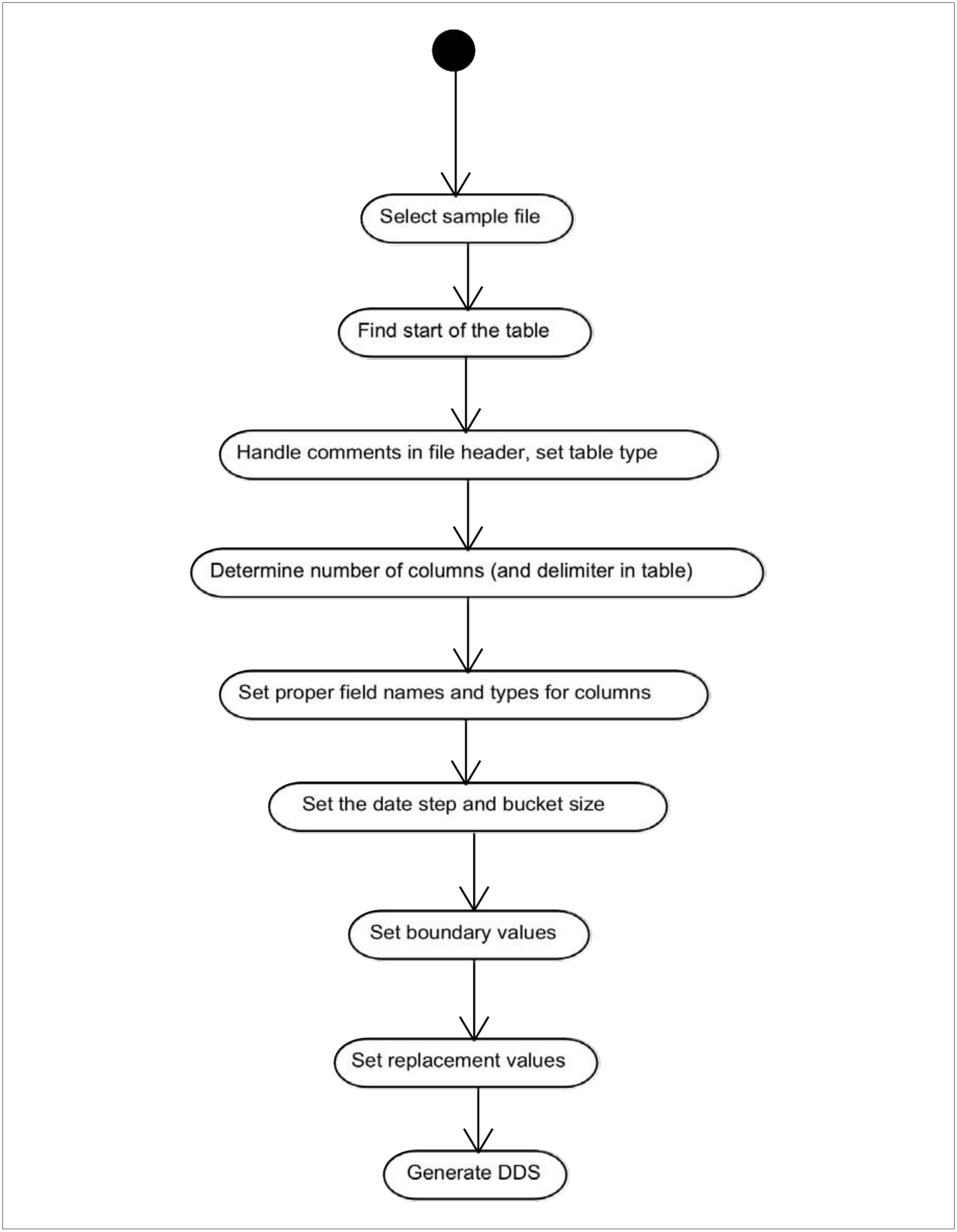

Select sample file
Find start of the table
Handle comments in file header, set table type
Determine number of columns (and delimiter in table)
Set proper field names and types for columns
Set the date step and bucket size
Set boundary values
Set replacement values
Generate DDS

Figure 6: Sequence of Events of Input Wizard

The sequence of events for the output wizard is shown in the Figure 7.

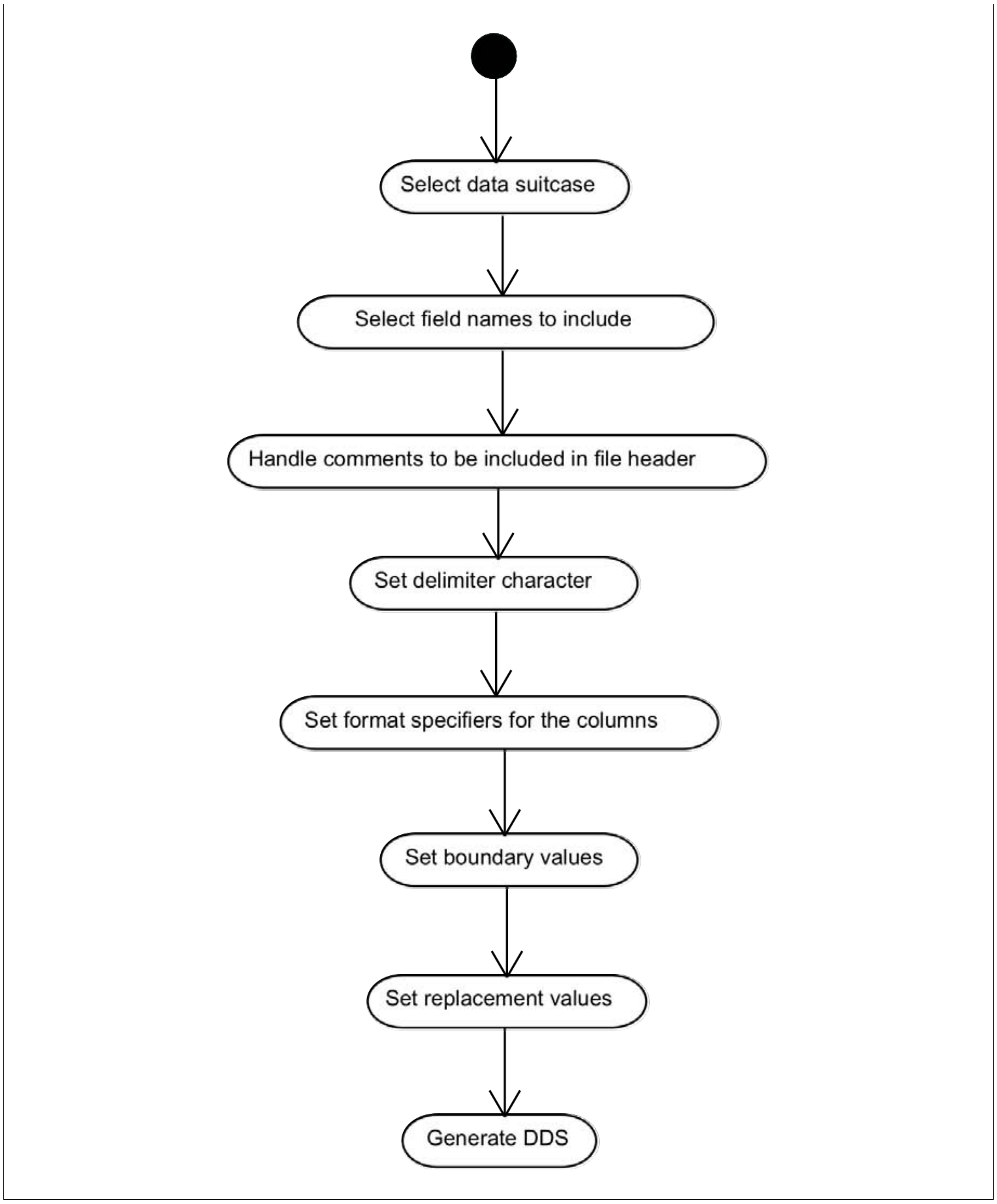

Select data suitcase
Select field names to include
Handle comments to be included in file header
Set delimiter character
Set format specifiers for the columns
Set boundary values
Set replacement values
Generate DDS

Figure 7: Sequence of Events of Output Wizard

3.4.3 Dynamic Class Loading

One of the key concepts of the conversion engines is that the conversion code, specific for a particular type of an ASCII data file, is loaded at the moment that this code (a class file) is needed. Even more important is that this class is unloaded if it is not needed anymore. Its place is taken by another specialized class file. This behavior is quite different from normal class loading in Java: the default class loader loads a class file if this class is needed, but never unloads it.

The benefit of the dynamic class loading is that the conversion engine can be an continuously running process and still handle all kind of files. The conversion process just loads the appropriate class file, which is garbage collected when another class file takes its place. See *The Java Developers Almanac 1.4* companion web site.[21]

In order to use dynamic class (re)loading a few points are important.

1. The class to be loaded should not be on the class path, because in that case the default class loader will not allow the class to be reloaded. The default class loader will allow the user defined class loader to do its work only if it does not know about the class to be loaded.
2. But that brings up another issue: if the class to be loaded is not on the class path, its type is unknown and thus the class cannot referred to in the code. To solve this problem the class to be loaded needs a base class and referring to the newly loaded class happens by type casting to the base class.

[21] *Dynamically Reloading a Modified Class*, 2002, <http://javaalmanac.com/egs/java.lang/ReloadClass.html?l=rel>, (October 8, 2004).

Figures 8 – 10 contain the source code for a dynamic class loader example that illustrates the principle. The example consists of two normal classes and one dynamically created and loaded class. First, in Figure 8, the class that is the "Factory", the class that does the dynamic loading.

```
Import java.io.*;
import java.lang.*;
import java.net.*;

public class Factory {
   private File file = new File("");
   private ClassLoader cl = null;
   public Factory(String path) {
      try {
         file = new File(path);
         URL url = file.toURL();
         URL[] urls = new URL[]{url};
         cl = new URLClassLoader(urls);
      } catch (Exception e) {
      System.out.println("Exception in Factory constructor " + e);
      }
   }
   public MessengerBase getMessenger(String className) {
      Object o = null;
      try {
         o = (cl.loadClass(className)).newInstance();
      } catch (Exception e) {
      System.out.println("Exception in getMessenger " + e);
      }
      return (MessengerBase) o;
   }
}
```

Figure 8: The Factory Does the Loading of the Class

Figure 9 is the “main” program that creates the dynamic class and the calls the user defined loader 10 times.

```
import java.lang.*;
import java.io.*;

public class MyDynamicLoadTest {
   public void createNewClassFile(int i) throws IOException {
      String srcFile = "../Messenger.java";
      StringBuffer code = new StringBuffer();
      code.append("public class Messenger extends MessengerBase {\n");
      code.append("    private String s = \"Message number ");
      code.append((new Integer(i)).toString() + "\";\n");
      code.append("    public String message() {\n");
      code.append("    return s;\n     }\n}\n");
      FileWriter outputFile = new FileWriter(srcFile);
      BufferedWriter out = new BufferedWriter(outputFile);
      out.write(code.toString() + "\n");
      out.close();
      Runtime.getRuntime().exec("javac " + srcFile);
   }
   public static void main(String[] args) throws IOException,
                        ClassNotFoundException, InterruptedException {
      MyDynamicLoadTest test = new MyDynamicLoadTest();
      int i = 0;
      while (i++ < 10) {
         test.createNewClassFile(i);
         Thread.sleep(3000); // sleep to make sure class is compiled
         // cast the new instance to the base class
         // the type of the actual loaded class unknown by the default class loader
         // note: the path is for the user loaded class is the parent directory
         // which is normally NOT on the class path
         MessengerBase m = new Factory("..").getMessenger("Messenger");
         // print the message from the newly created class
         System.out.println ((String) m.message());
         System.out.println ("================  this was cycle " + i);
      }
   }
}
```

Figure 9: The “Main” Program

Finally Figure 10 illustrates the dynamically generated class. Note that the number in the string "Message number 1" will change while the program runs, this string is the "dynamic part" of this example.

```
public class Messenger extends MessengerBase {
   private String s = "Message number 1";
   public String message() {
   return s;
   }
}
```

Figure 10: The Dynamically Generated Class

3.4.4 Conversion Code Design

As discussed in the architecture section, the interfaces of the conversion engines are still an undecided issue. This project focuses on the feasibility of the challenging areas of the proposed solution, not on the completeness of it.

In the following sections the design of the conversion engines is covered in a rather general way because of the fuzziness of the interfaces between the sub-systems. Thus little attention in paid to the interfaces.

3.4.4.1 Conversion Engine File to Suitcase

The high-level flow diagram of this conversion engine that converts a file into a data suitcase is according to Figure 11.

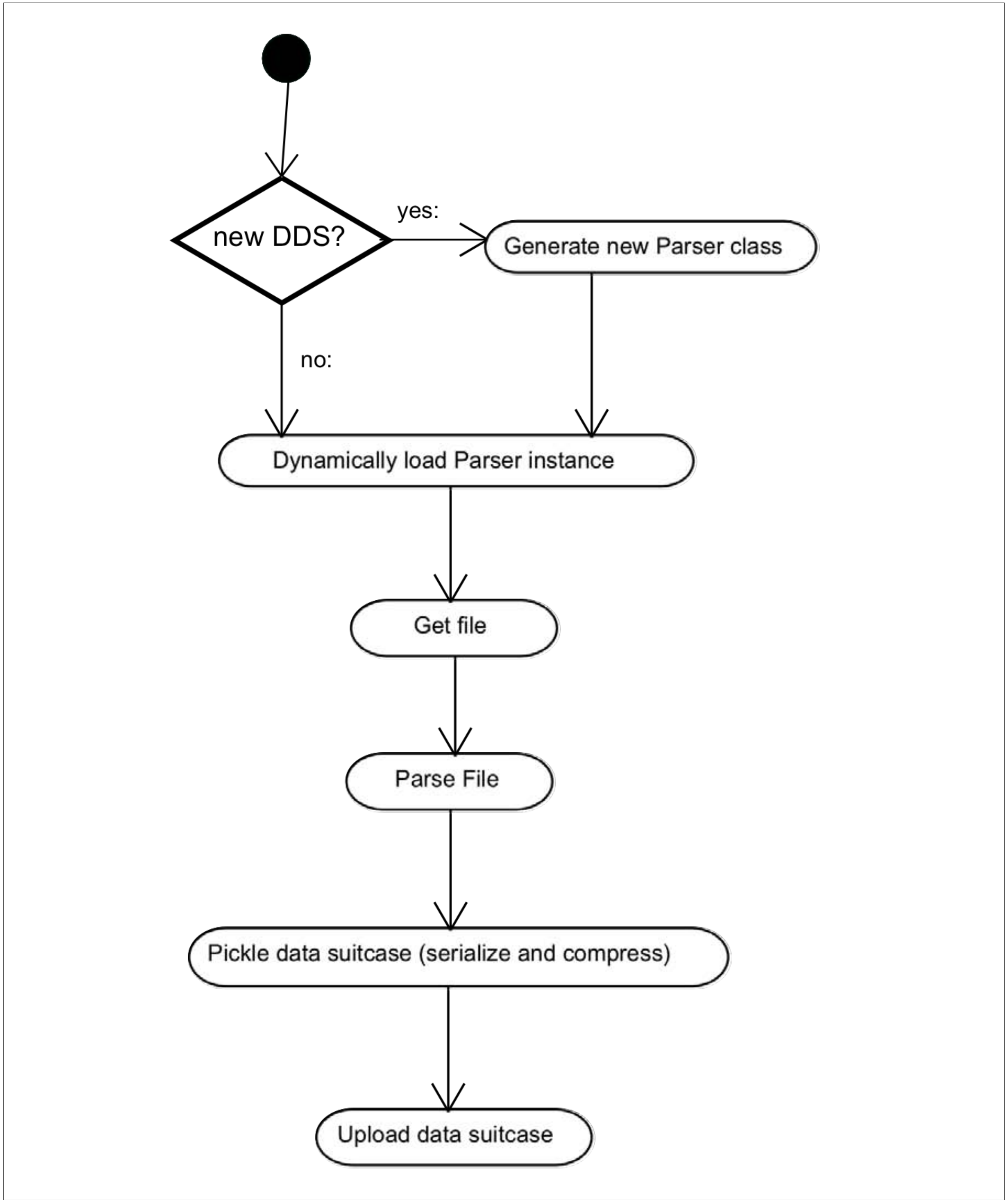

new DDS?
yes:
Generate new Parser class
no:
Dynamically load Parser instance
Get file
Parse File
Pickle data suitcase (serialize and compress)
Upload data suitcase

Figure 11: Flow Diagram for Conversion Engine, File to Suitcase

In the node “Parse file” of the diagram is where the most activity takes place.

These are the steps for parsing a file:

1. Read the file line by line, add all lines above the table header to meta-data attribute of suitcase. After the table header move to point 2.
2. Read line by line and process lines of data.
   - 2.1. Split line in tokens and convert tokens to typed variables (the fields).
     - 2.1.1. If the field creation failed, try to repair the bad line.
   - 2.2. Check the validity of the date variable.
   - 2.3. Put the variables in a Record class and validate variables.
   - 2.4. Store record in a temporarily bucket (CollectorBucket).
   - 2.5. If this temporarily bucket is full, calculate median of values.
   - 2.6. Serialize the bucket and store in suitcase.
3. Serialize last bucket and store in suitcase.

Step 1 to 2.1 happens in the dynamically created Parser class. The rest takes place in the ParserBase class.

In Figure 12 the UML class diagram illustrates the relationships between the classes of the file to suitcase conversion engine. This engine shares the Record class, the Bucket class and the Suitcase class with the code of the suitcase to file conversion engine.

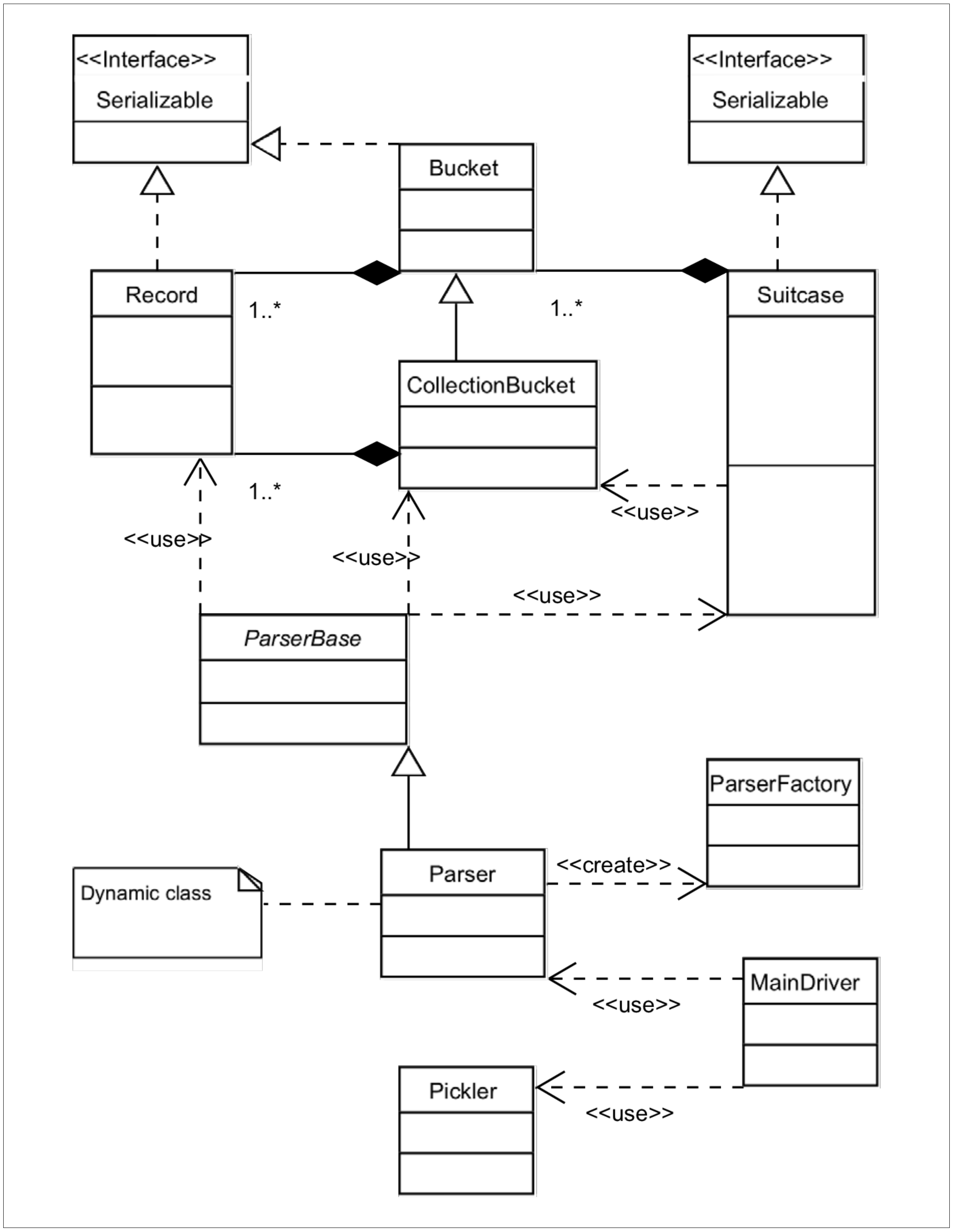
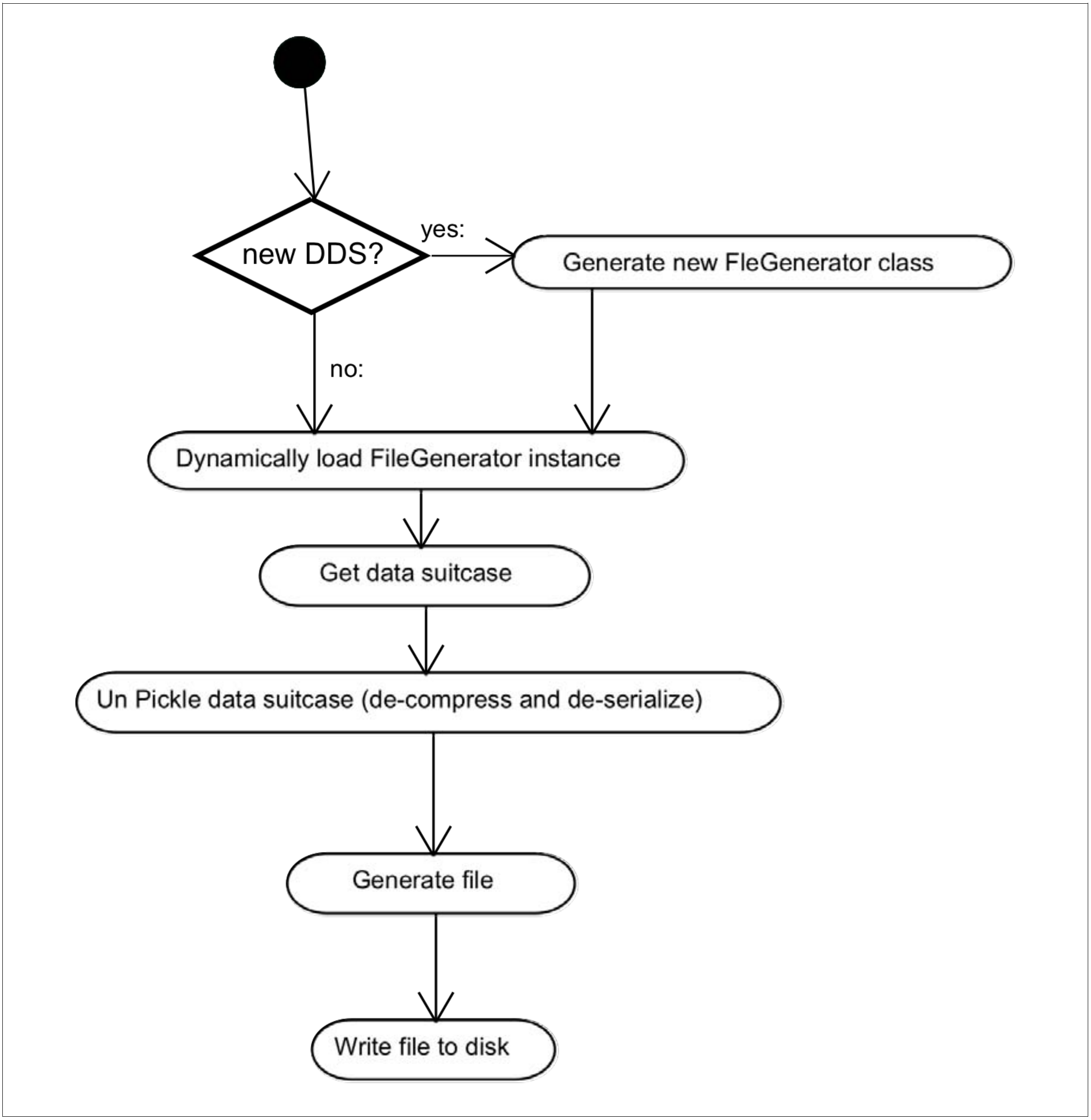


Figure 12: Class Diagram for the File to Suitcase Conversion Engine

3.4.4.2 Conversion Engine Suitcase to File

Figure 13 is a high level illustration of the conversion from suitcase to file.



new DDS?
yes:
Generate new FleGenerator class
no:
Dynamically load FileGenerator instance
Get data suitcase
Un Pickle data suitcase (de-compress and de-serialize)
Generate file
Write file to disk

Figure 13: Flow Diagram for Conversion Engine, Suitcase to File

The details of the “Generate file” step Figure 13 are as follows:

```
1.  Get first bucket from data suitcase.
    1.1.  Loop until all buckets in suitcase processed.
          1.1.1.  Loop until all records in bucket are processed.
                  1.1.1.1.  Compose properly formatted line.
                  1.1.1.2.  Write line to file.
                  1.1.1.3.  Get next record
    1.2.  Get next bucket from data suitcase.
```

All the steps mentioned above are happening in the FileGenerator class. This conversion engine shares the Record class, the Bucket class and the Suitcase class with the code of the file to suitcase conversion engine.

The class diagram for the suitcase to file conversion engine is in Figure 14.

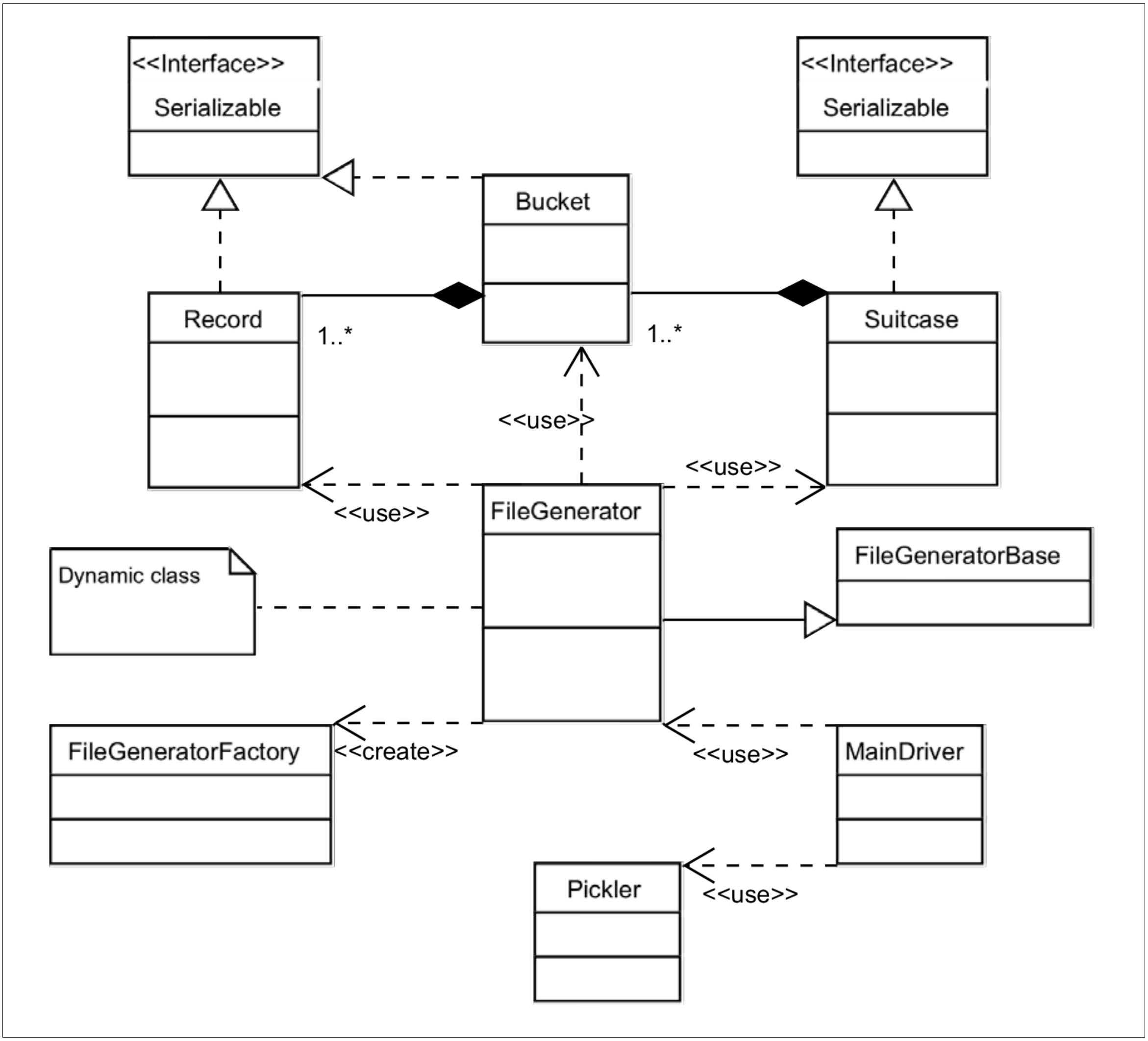


Figure 14: Class Diagram for the Suitcase to File Conversion Engine

3.5 Test Approach

Since this project is about researching the feasibility of the proposed method of data conversion, all testing effort is geared towards the testing of the dynamically generated code. Testing code that does not change after it is written and before use is easier to test and the methods are well understood and documented in literature about testing.

3.5.1 Introduction

In software quality assurance it is understood that it is most likely impossible to find all errors in a non-trivial piece of software. Often the goal of software quality is considered to be customer satisfaction. It is considered best to approach testing with an appropriate methodology to find as many errors as possible. Random testing, although it is a methodology, isn't a proper way of testing, something more sophisticated is needed.

The general practice of testing uses two ways of testing, black box and white box testing.[22]

In black box testing the tester does not know about the internals of the code to be tested, only about the functionality provided. There are two common methods for black box testing:

1. Partitioning testing. In this method the input domain is divided into equivalent partitions. The reasoning behind this is that two input values from the same partition will likely give the comparable test results. Thus if input values from all partitions are tested, it is likely that the whole input domain is adequately covered.
2. Boundary value testing. In this method the boundaries of the input domain are determined. It is common for software to behave more error prone on or close to the boundaries of the input values then far away from these boundaries. Thus it is likely that better coverage is achieved if testing of all boundaries is done, including just inside and out side the boundary, as well as further away from the boundary.

In white box (or glass box) testing the tester does know the internal structure of the code. And as in black box testing the keyword is coverage. There is one obvious way of white box testing: make sure each statement is at least executed once and that all possible paths are taken. Testing all possible paths might be an almost impossible task in some cases, so the following methods for white box testing are developed:

1. Statement coverage. Here the test cases are designed such that all statements are executed at least once.
2. Branch coverage. Test cases are designed such that all branches in a condition statement are taken at least once.
3. Path coverage. Since it might be impossible to take all possible path, this method designs the test cases such that all independent paths are taken at least once.

[22] Jerry Z. Gao, H. -S. Jacob Tsao and Ye Wu, *Testing and Quality Assurance for Component-based Software*, (Boston: Artech House, 2003), 119 – 154.

4. Loop coverage. In loop coverage the same kind of considerations as in black box boundary testing play a role. Test cases are designed to test the extremes and cases close to the extremes.

3.5.2 Testing of Dynamic Generated Code

Because of the dynamic code generation, the testing needs to be dynamic too. Given the concept of the conversion sub-system, the testing should be automatically performed as part of the creation of new conversion functionality. Automatic testing is quite a challenge, however there is a pattern in the test cases because a data suitcase always has one or more buckets and a bucket has one or more records.

Values that should be tested:

1. suitcase flags
2. suitcase flags after making a slice
3. suitcase, the bucket flags
4. record flags for a certain date
5. field flags for a certain date
6. field values for a certain date

Each new case, meaning new type of input file with a corresponding new DDS and Java conversion code, and possibly a new data suitcase type, needs to be tested since there is new code involved. That also means that the test case test file and test values need to be created, based on the sample file that was used to generate the DDS.

In order to run test cases automatically two pieces of functionality are needed:

1. A test framework that compares the actual values of the data suitcase with the expected values. This is the more common part: a test suite based on JUnit, or in this project PyUnit (with Jython), is a common way to this.[23] PyUnit is part of the Python and Jython standard library and is located in the module `unittest`.
2. A module that generates a test input files and corresponding expected values. This is the challenging part since this is a non-trivial piece of software.

Test input files need to be prepared such that they will cause the conversion code to produce certain values for field values and all the different flags. Thus the test preparation system needs to take the input file (the sample used to generate the DDS) and generate modified copies based on the values of the DDS.

3.5.3 Test Cases

There are several general cases that can occur, divided into two groups:

1. Boundary check is not performed.
   1.1. Normal; everything is normal and as expected and flags indicate valid data (and no bound check).

[23] *JUnit, Testing Resources for Extreme Programming*, 2004, < http://www.junit.org/index.htm >, (October 8, 2004); *PyUnit - the Standard Unit Testing Framework for Python*, 2004, < http://pyunit.sourceforge.net/ >, (October 8, 2004).

   1.2. Date jump; a few “jump in date” cases, flags reflect that and the date before “jump in date” does not exist (this corresponding line in the test file was removed), and no bound check.
   1.3. Replace; a few garbage lines in the file.
      1.3.1. One case is one garbage line, this line will be replaced by the replacement values from the DDS (or replacement values from the median record values, which is currently not implemented); flags indicate replaced values and no bound check performed.
      1.3.2. The other case is two or more consecutive garbage lines, which cannot be replaced and will result in a “jump in date” case, and flags will reflect jump in date (and no bound check).
2. Boundary check is performed, the test file are prepared such that all values are in bounds for the cases “Normal”, “Replace”, and “Date jump”. Only the test file for the “Bound check” case has actual out of bound values.
   2.1. Normal; everything is normal and as expected and flags indicate valid data (including a valid bound check).
   2.2. Date jump; a few “jump in date” cases, flags reflect that and the date before “jump in date” does not exist (this corresponding line in the test file was removed), and bound check indicates valid.
   2.3. Replace; a few garbage lines in the file.
      2.3.1. One case is one garbage line, this line will be replaced by the replacement values from the DDS (or replacement values from the median record values, which is currently not implemented); flags indicate replaced values and bound check indicates valid.
      2.3.2. The other case is two or more consecutive garbage lines, which cannot be replaced and will result in a “jump in date” case, and flags will reflect jump in date (and a valid bound check).
   2.4. Bound check; Some of the test values are in bounds and some are out, the expected values and flags reflect that. In this case the dates are all OK.

3.5.4 Test Sub-system Design

The test files are based on the DDS and the sample file. From these test files the test data suitcases are generated (test data suitcases are output of the parser just created from the DDS), as well as the expected values of the test cases.

The test file preparation has a number of steps:

1. Reading of the sample file and fixing out of bound values if out of bound checking is set in the DDS.
2. Creation of a random set of records to test.
3. Preparation of the test input files such that the conditions to test are met in the test input files, for example to test proper record replacement valid lines in the test input file are replaced with “bad” lines. Or deleting one line in the test input file creates a “jump in date”. At the same time the expected test values files are prepared too, with the values corresponding to the test input files.
4. Creation of the files. The preparation is done in memory. After completion the files are written to disk.

The test suite is rather straightforward, more a matter of implementation than design. Using the PyUnit (as said, very similar to JUnit) framework and Jython, the test suite class sets up each test by creating a data suitcase from the test input file and continues with the actual tests, which are common assert equal tests.

4 Implementation

As already mentioned in Chapter 3 the implementation details in this chapter refer to the latest state of the development. In some places comparisons with older stages of the development are made for more insight in the why of choices.

Although this is the implementation chapter, it does not contain a lot of source code. The reason is that the source code and the source code documentation are proprietary.

4.1 The GUI System

ZPT's support two features that are use in this application:

1. Nesting ZPT's, meaning one template can include another.
2. Calling a ZPT (from a Python script) with a parameter.

That is how the wizards are implemented: a main template embeds or includes the actual wizard screens (based on the screen number) and is called every time the "Next" or "Back" button in the wizard is clicked. Since these buttons are part of a form, a form variable (in the request object) with the current screen number is passed to the form processing Python script. For the "Next" button this is the `next` script. This script does some additional work if necessary, after which it increases the screen number and calls the main template again with the next screen number. The main template includes the next screen because that is the number it got passed in. The "Back" button works the same way, but of course the `back` script decreases the screen number, so when the main template is called it will include the previous screen.

In some screens there is a button to perform some action and returns to the same screen, while the reloading of the screen reflects the changes made. Yet in most cases the processing is done in between screen calls.

The HTTP protocol is stateless, which means that variables that need to be remembered between screens need to be stored somewhere. Usually that happens either by way of browser cookies or by way of hidden form variables. The web application programmer does not need to worry about the low level mechanics, but can use session variables and the effect is that these variables are "stored" somewhere. Usually session variables are set to expire after a certain time.

Error checking is implemented where possible/reasonable. Overall the implementation uses a prototype approach. A Zope connoisseur, sometimes referred to as a "Zopista", will probably acknowledge that a Zope Product approach would probably be a more appropriate choice for a production solution.

4.2 The Conversion Code

For the implementation of the conversion code the goal was to minimize the amount of dynamically generated code without compromising the execution speed. Eventually the amount of dynamic code grew slightly in order to avoid some conditional branches that would otherwise be executed for every line in the input file. Figure 15 shows a driver program that exercises the conversion code.

```
Import java.util.*;
```

```
import java.lang.*;
import java.io.

public class Driver {
   public Driver() {}
   public static void main(String[] args) throws IOException, ClassNotFoundException  {
      int i = 0;
      String tmp = null;
      String path_1 = "../version_1";
      String path_2 = "../version_2";
      while (i++ < 10) {
         System.out.println("Cycle:  " + i);
         System.out.println("Parser path " + path_1);
         //get a parser instance; dynamic loading with user defined class loader
         ParserBase p = new ParserFactory(path_1).getParser("Parser");
         p.parseFile(path_1 + "/data.txt");                    //parse file
         Suitcase s = (Suitcase) p.getContainer();           //get data suitcase
         //serialization,  compression and then store in file
         Pickler pickle = new Pickler();
         pickle.pickleGZIP(s, "pickle.gz");
         //now unpack and de-serialize
         s = (Suitcase) pickle.unPickleGZIP("pickle.gz");
         //print some of the suitcase's properties
         System.out.println("Bucket range: " + s.getBucketRange());
         System.out.println("Field name and type: " + s.getFieldDefs());
         System.out.println("Suitcase validation flag: " + s.getValidationFlag());
         System.out.println("Suitcase date flag: " + s.getDateFlag());
         System.out.println("File generator path " + path_2);
         // get a file generator  instance; dynamic loading with user defined class loader
         FileGeneratorFactory fgFactory = new FileGeneratorFactory(path_2);
         FileGeneratorBase fg = fgFactory.getFileGenerator("FileGenerator");
         fg.setSuitcase(s);                                       //pass in the data suitcase
         fg.generateFile(path_2 + "/data.txt");                   //generate an output file
         //swap the paths to start the alternative cycle
         tmp = path_1;
         path_1 = path_2;
         path_2 = tmp;
      }
   }
}
```

Figure 15: Driver Program to Illustrate the Working of the Conversion Code

The driver program in Figure 15 cycles 10 times. Every cycle the parser and file generator classes are dynamically loaded and the file created in the previous cycle is processed.

4.3 Demo Extension to GUI System

In order to demonstrate actual conversion capability the input wizard of the GUI system was extended with the capacity to generate a data suitcase from the sample file that was used to create a DDS. In the same spirit the output wizard was complemented with the capability to generate a file from the suitcase that was used for the output wizard DDS.

It is not possible to start the execution of another process directly from standard Zope, but Zope does support integrating an XML-RPC client therefore these extensions to the GUI were implemented with help of an XML-RPC server.[24]

The following features characterize XML-RPC protocol:

1. Remote procedure calls in XML format over HTTP.
2. Libraries for many programming languages, usually easy to use.
3. Multiple data types, primitive types and strings as well as lists/arrays and hash tables or dictionaries.

The XML-RPC server was implemented in Jython (in the same user space as Zope), which made it possible to use the Python XML-RPC library as well as access the Java conversion code. Using Jython/Python XML-RPC library is a lot easier then using the Java XML-RPC library. On the client side, creating an XML-RPC client (in Python, because the client lives inside the Zope realm) is extremely easy, just one line of code to create the client object and another line to call a remote procedure.

4.4 Performance Experiments

Although there are no specific requirements for the performance of the conversion sub-system, it is an important topic because it determines the cadence in which the IFS can be run and/or how much hardware is needed to perform a certain cadence. In general the conversion sub-system should be fast and have a small memory footprint when running.

4.4.1 Performance Overview

Given the context of the choice to use Java because it features, the performance of the Java code was compared to Python code because Python possesses much of the same features that were the reasons to choose Java to build the system. It was already clear that Java is rather slow as far as serialization is concerned. That prompted a preliminary test to see how Python performed in that arena. The difference was rather dramatic, enough to investigate this track further.

Since Java code was already available, a trial with the native GCJ compiler was done. This was a rather disappointing exercise. For some strange reason the executable was in the order of eight to ten times slower then the Java VM. The GCJ option was dropped for further comparison.

4.4.2 Comparisons

Four sets of comparisons were made:

1. Comparing plain Java and plain Python, no attempts to enhancements.
2. Because date conversions happen for every line in a file (in both conversions) the date conversions were calculated in advance and stored in hash tables,

[24] *XML-RPC Home Page*, (UserLand Software, Inc: July 3, 2003),), <http://www.xmlrpc.com/>, (October 8, 2004).

which were kept in memory for as long as the process was running. For both Java and Python this was a significant speedup.

3. A variant of number 2, but this time the hash tables with the date conversions were loaded from disk just before the conversion was done. The reason for this was to see the trade off between memory footprint and execution speed. Although the tables for this one test are not that big (2 MB), imagine having to keep the tables for 20 different conversion set-ups in memory all the time, that could become a mere 40 MB.
4. Finally two possible enhancements were tried, for Java that was the uka.transport library from JavaParty that claimed faster serialization and for Python it was Psyco, a just-in-time compiler.[25] The JavaParty library was not an improvement but Psyco showed some performance gain. The charm of Psyco is that this gain was accomplished by installing the library and adding a few lines of code, sort of instant result. One of the downsides of Psyco is that, according to the web site, "It only runs on Intel 386-compatible processors (under any OS) right now."

4.4.3 Results

The conversion was done with a 1.4 MB data file. The file was converted (Parsing) into a data suitcase, which was serialized and compressed into a file. Then this data suitcase was unpacked again and converted back into a file (File generation). The timing is in seconds and was calculated from the average of 10 runs. Table 4 shows the results. While running, the process is the only active process (except some the normal background processes) and takes almost all CPU time, usually more then 95%.

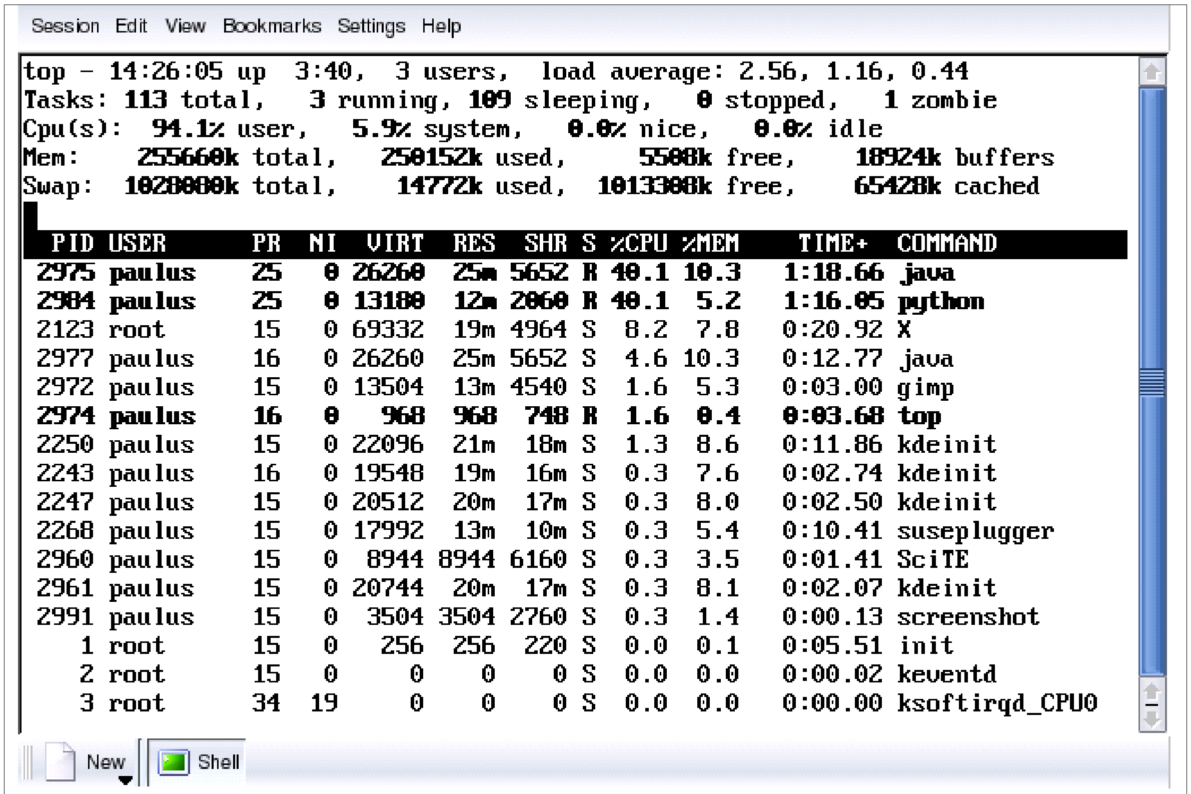
Table 4: Results of the Performance Tests

Set up	***Java - time in sec.***		***Python - time in sec.***	
Processes use almost all CPU time.	**Parsing**	**File generation**	**Parsing**	**File generation**
1. No measurements for speed-up	6.52	7.24	5.99	7.57
2. Date conversion tables	4.98	3.53	5.26	2.40
3. Date tables loaded from disk	5.50	4.77	5.55	2.91
4. Java + JavaParty; Python + Psyco	9.98	5.23	4.81	2.51

There were three date conversion tables used: two of them were 637 kB in size and one was 740 kB.

Figure 16 is a screen shot from the program top, which displays memory and CPU usage for the processes running. Top is a Unix/Linux utility program. As the screen shot shows, the Java performance test uses twice the amount of memory as the Python process. The functionality of both programs is the same.

[25] Bernhard Haumacher, Thomas Moschny and Michael Philippsen, *Fast Object Serialization uka.transport*, November 3 2003, <http://www.ipd.uka.de/JavaParty/ukatransport.html>, (October 8, 2004); *Psyco - Home Page*, July, 30 2004, < http://psyco.sourceforge.net/>, (October 8, 2004).



```
top - 14:26:05 up  3:40,  3 users,  load average: 2.56, 1.16, 0.44
Tasks: 113 total,   3 running, 109 sleeping,   0 stopped,   1 zombie
Cpu(s):  94.1% user,   5.9% system,   0.0% nice,   0.0% idle
Mem:    255660k total,   250152k used,     5508k free,    18924k buffers
Swap:  1028080k total,    14772k used,  1013308k free,    65428k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 2975 paulus    25   0 26260  25m 5652 R 40.1 10.3   1:18.66 java
 2984 paulus    25   0 13180  12m 2060 R 40.1  5.2   1:16.05 python
 2123 root      15   0 69332  19m 4964 S  8.2  7.8   0:20.92 X
 2977 paulus    16   0 26260  25m 5652 S  4.6 10.3   0:12.77 java
 2972 paulus    15   0 13504  13m 4540 S  1.6  5.3   0:03.00 gimp
 2974 paulus    16   0   968  968  748 R  1.6  0.4   0:03.68 top
 2250 paulus    15   0 22096  21m  18m S  1.3  8.6   0:11.86 kdeinit
 2243 paulus    16   0 19548  19m  16m S  0.3  7.6   0:02.74 kdeinit
 2247 paulus    15   0 20512  20m  17m S  0.3  8.0   0:02.50 kdeinit
 2268 paulus    15   0 17992  13m  10m S  0.3  5.4   0:10.41 suseplugger
 2960 paulus    15   0  8944 8944 6160 S  0.3  3.5   0:01.41 SciTE
 2961 paulus    15   0 20744  20m  17m S  0.3  8.1   0:02.07 kdeinit
 2991 paulus    15   0  3504 3504 2760 S  0.3  1.4   0:00.13 screenshot
    1 root      15   0   256  256  220 S  0.0  0.1   0:05.51 init
    2 root      15   0     0    0    0 S  0.0  0.0   0:00.02 keventd
    3 root      34  19     0    0    0 S  0.0  0.0   0:00.00 ksoftirqd_CPU0
```

The top two lines (under the black heading) show that the Java process uses twice as much memory. Screen shot from the program top (Unix/Linux utility).

Figure 16: Screen Shot of Memory and CPU Usage of the Performance Test

4.4.4 Discussion

Without any extra actions in order to enhance the performance, it turns out that Java and Python have similar speed performance. Although Python is very fast regarding object serialization (an extension in the standard library implemented in C), this gain is partly lost in other parts of the program.

Using date tables for date conversions proves to be a big performance gain, especially for the file generation. If the date tables are loaded for each conversion run it shows that Python does a faster job.

However the biggest difference between the two is the memory footprint of the running processes. The Java process uses two times the amount of memory as the Python process, while each uses about the same amount of CPU time.

With Java the ceiling of the memory is reached much earlier. That in itself might not be a problem, nowadays the credo is “memory is cheap”. Compared to the salary of a software engineer for one single day, 1 GB of memory can be considered cheap, labor is usually the biggest cost factor. But this observation about the cost of labor points to the direction of Python's biggest strength, which is programmer productivity. Whether counted as lines of source code or the number of characters in the code, the count shows that Java is more then three times as verbose, all aside from the personal observation that Python is not as complex as Java. If it is true that a software engineer produces the same amount of code regardless of the language, then it will be clear why Python should be considered more seriously. This performance chapter shows that performance is not a compelling reason to keep Python out the picture.

4.5 Test Suite

The implemented test suite is not complete, but complete enough to demonstrate that the concept of generating tests dynamically is possible.

4.5.1 Test Coverage

Since the structure of the dynamically generated code is known, an assessment can be made how well the white box coverage is. The test case "Replace", where there are "bad" lines in the input test file, exercises all statements and all independent paths. With a small addition to the test file preparation this test case will also have complete branch coverage. Due to the structure of the test file generation loop coverage (boundary cases) is not complete. If desirable this could be solved with some additional test cases.

From a black box point of view, the input domain partitioning is covered, except for some settings in the DDS. The test preparation code currently does not deal with fixed width fields and neither does it calculate the median replacement values from the valid records in a bucket.

Black box boundary value testing has incomplete coverage. The test preparation code currently can only partly deal with one record per bucket (for instance bucket size = one day, date step size = one day). However one bucket in a suitcase is handled properly.

There is another way to do black box partitioning. Input files have a meta-data section (above the table header), a table header and a data section. The table header is required, but strictly spoken a meta-data section or a data section is not required.

Thus the following black box partitions exists:

1 A table header only (this is not very meaningful)
2 Meta-data section plus table header
3 Table header plus data section
4 Meta-data section plus table header and data section

4.5.2 Test Framework Implementation

The code (in Python) to produce the test file sets (input files and corresponding files with the expected test values) is integrated with the GUI system build in Zope. Since it is a relatively big amount of code and this code needs to write to disk, it is implemented as a module outside Zopes's object database and accessed with an external method. For the test to execute, Zope needs to start two external processes, the Java compiler and the test runner. As already pointed out in the section about the demo extension of the GUI system, it is not possible to start the execution of another process directly from standard Zope: therefore the same client server approach is used. An XML-RPC server compiles the Java code and runs the test suite. The results of running the test suite are routed back to Zope and displayed in the browser. The XML-RPC server can just run in the same user space as Zope. This solution is only used to circumvent the limitation of starting a process from Zope.

The test suite is based on PyUnit/Jython. The benefit of this is that the "expected values" file is generated as executable Jython code and loaded/executed by the test suite. The downside is that it is sometimes a little tricky to get the assertEqual method to accept two object as equal. Some conversion from Java to Python data types was necessary.

4.5.3 Sequence of Execution for a Test Case

Test case sequence:

1. Setup – create a suitcase from file
2. Test suitcase (flags, bucket range, meta-data, field definitions) – test date flag and validation flag, test range, test both meta-data types and, field definitions, all for suitcase and suitcase-slice (if any)
3. Test bucket flags – test date flag and validation flag
4. Test record (record flags, field flags, field values) – test a few records in each bucket for record flags, field flags, field values

5 Conclusion

This project started with a number of desired features for the IFS:

1. A visual wizard to assist in defining the structure of the input file such that based on that information automatic conversion from input file to data suitcase would be possible. Also conversion the other way, suitcase to file, should be possible.
2. An implementation of the visual wizard that ideally only requires standard software by the user, such as a web based solution accessible with a standards compliant graphical browser.
3. Allowing the input files to be as "free form" as possible, just unambiguous enough to make automatic conversion possible.

The results of the project show that it is possible to:

1. Create a method for automatic conversion of input files based on a DDS file. Conversion the other way is possible too. Note that so far only one type of input file in use by the IFS is covered, although this file type is very common. Future work needs to address the other types of files.
2. Create usable GUI wizards implemented as a web application and based on HTML, cookies and javascript only. This setup does not require anything other then a standard browser as the user client program.
3. Define the input file format requirements and strike a balance between "free form" and the necessary minimal structure.

The project generated some added bonuses related to the core work:

1. A better understanding of how to structure the data suitcases.
2. Possibilities for performance improvements of the conversion code if performance turns out to be a concern.

5.1 Remaining Issues

The remaining issues are divided into two categories:

1. Issues directly related to the topic of the project
2. Issues regarding the general system design

5.1.1 The Project Related Issues

The project-related issues are about the handling of other types of files, such as ASCII files with a very different structure and binary files.

Binary files are usually well defined and require a conversion approach specific for the type of binary file. However, with an extension to the conversion engines, it is possible to handle binary as just a glob of data and store it in a data suitcase without conversion.

To handle ASCII files with a different basic structure, it is necessary to define the rules for the basic structure of a new type of ASCII files. A design for an additional GUI extension is then based on these basic rules. A simple example of a different type of ASCII file is given in Figure 17. The approach here would be to split the line in the part before the colon and the part after the colon. The first part and the second part are then stored as a key-value pair.

```
Flux Density Values in sfu for 2300 on 2004:03:27

Julian Day Number               : 02453092.458

Carrington Rotation Number      :  002014.731

Observed Flux Density           :  000128.4

Flux Density Adjusted for 1 A.U.:  000127.9

URSI Series D Flux (Adj. x 0.9) :  000115.1
```

Figure 17: Example of a Different Type of ASCII File.

5.1.2 General Issues

General issues:

1. In section 3.3.1 is already discussed that there are currently no firm decisions made regarding the interfaces between sub-systems. Reviewing (and possibly changing) the design of the data suitcase is another item on the list. Before the IFS is ready as a production system, these issues need to be addressed.
2. Another area that needs exploration is the handling of errors in case they occur. Errors usually translate to exceptions in the code. How are exceptions handled? There are potentially two extremes: one extreme is to handle all exceptions at one central point and the other extreme is to handle all exceptions at the point where they occur. A sensible approach most likely depends on the situation and probably should follow the responsibilities. That is, if a user (end user or model) is responsible for a certain task and is able to deal with a problem, then the exception should be handled at that level. However, if the responsibility is at the system administration level then the exception need to trigger an event that notifies the system administration.

5.2 Programming Language Choice

The core of the IFS is composed out of a number of models, written in IDL (IDL is a programming language and stands for Interactive Data Language, not to be confused with CORBA IDL, which stands for Interface Definition Language) and legacy code such as Fortran. Java (with help from Jython) is envisioned as the glue that turns the collection of models into a system. An important consideration to choose Java is platform independence, although Java does not really shine as a glue language. Java shares the feature of platform independence with other languages such as Perl and Python. And as the experiments regarding performance show, performance is not a reason to leave Python out of the picture. Although Perl is rather popular, it does not have the clean syntax and design that Python has. Because Python is so well designed, projects in Python are known to scale well to bigger systems.

Python programs are, in general, much smaller and faster to write then their Java counter parts and for that reason Python is probably better suited to act as a glue language.

It is my opinion, since the IFS system is in a relatively early stage of development, that the primary developers should reconsider the choice for Java.

References

ArgoUML Project Home, 2003, <http://argouml.tigris.org/>, (July 3, 2004).

Bruegge, Bernd and Allen H. Dutoit. *Object-oriented Software Engineering: Conquering Complex and Changing Systems*. Upper Saddle River: Prentice Hall, 2000.

Cooper, James W. *Java Design Patterns: a Tutorial*. Boston: Addison Wesley, 2000).

Dynamically Reloading a Modified Class. 2002, <http://javaalmanac.com/egs/java.lang/ReloadClass.html?l=rel>, (October 8, 2004).

Gamma, Erich et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Reading: Addison-Wesley, 1995.

Gao, Jerry Z., H. -S. Jacob Tsao and Ye Wu. *Testing and Quality Assurance for Component-based Software*. Boston: Artech House, 2003).

Girczyc, Emil F. and Tai Ly, "Stem: an IC Design Environment Based on the Smalltalk Model-View-Controller Construct", *Proceedings - 24th ACM/IEEE Design Automation Conference*. 1987, IEEE, 757-763.

Haumacher, Bernhard, Thomas Moschny and Michael Philippsen. *Fast Object Serialization uka.transport*. November 3 2003, <http://www.ipd.uka.de/JavaParty/ukatransport.html>, (October 8, 2004).

Java 2 Platform, Standard Edition (J2SE), 2004, <http://java.sun.com/j2se/>, (July 3, 2004).

Jbuilder The Leading Development Solution for Java. 2004, <http://www.borland.com/jbuilder/>, (July 3, 2004).

JUnit, Testing Resources for Extreme Programming. 2004, <http://www.junit.org/index.htm >, (October 8, 2004).

Jython Home Page. June 20, 2004, <http://www.jython.org/>, (July 3, 2004).

Larman, Craig. *Applying UML Patterns: an Introduction to Object-oriented Analysis and Design and the Unified Process*. Upper Saddle River: Prentice Hall, 2002.

Latteier, Amos et al. *The Zope Book (2.6 Edition)*. 2003, <http://zope.org/Documentation/Books/ZopeBook/2_6Edition/ZopeBook-2_6.pdf>, (July 3, 2004).

Lutz, Mark. *Programmin Python*. Sebastopol: O'Reilly, 2001.

Psyco - Home Page. July 30, 2004, <http://psyco.sourceforge.net/>, (October 8, 2004).

Python Programming Language. 2004, <http://www.python.org/>, (July 3, 2004).

PyUnit - the Standard Unit Testing Framework for Python. 2004, <http://pyunit.sourceforge.net/ >, (October 8, 2004).

SciTE A Free Source Code Editor for Win32 and X, May 29, 2004, <http://www.scintilla.org/SciTE.html>, (July 3, 2004).

Tobiska, W. Kent and Dave Bouwer, Sections 1.4 to 1.6 are a summary of personal communication and internal SET documents, February 2004.

— — —, Specific information about the IFS in Chapters 3 are a summary of personal communication and internal SET documents, February 2004
XML-RPC Home Page. UserLand Software, Inc, July 3, 2003, <http://www.xmlrpc.com/>, (October 8, 2004).

Zope. 2003, <http://www.zope.org/>, (July 3, 2004).