4. OBSERVATIONS, RESULT & DISCUSSION

In this fourth chapter several observations, results and the cross discussions will be presented. This is the most important chapter of this thesis. Research is an academic activity and as such the term should be used in a technical sense. For this first of all, observations given from empirical data gathered of previous research studies & analysis with compare with previous delivery architecture and purposed deliverance model within an integrated implementation. This chapter shows the strengths and weaknesses of Plug-in Architecture which are observed in development of a project i.e. Earth Navigator on Rich Client Platform. It also shows the comparison of Linux and Windows Platform, standard widget toolkit and swing, Problems with Eclipse, with Discussion.
4.1 OBSERVATION
After all, handling the Rich Client Platform and learning how to develop applications on top of it is a challenge. First of all, one has to get familiar with the IDE itself. That is not much of an effort, but Eclipse introduces various new concepts that have to be understood. These include the editor, view and perspective approach, the plug-in concept that allows extending the IDE and building automation. The next step is to go through plug-in development. Due to the flexibility of the plug-in mechanism, one has to examine much source code and documentation files to fully understand the concept. Relying on the PDE and plug-in creation wizard is simply not enough for Rich Client development, because one has to know the details of how plug-ins is loaded to include the proper libraries at manifest file. The process of creating manifest files is error-prone, and due to the fact that each plug-in is loaded at runtime; one does not know whether the plug-in executes properly or not at compile time.

4.1.1 Earth Navigator Prototype
As soon as the obstacles mentioned above are mastered, one can proceed to the development of the first views and editors for his application. After understanding how the Rich Client Platform handles extensions like new editors and views, adding other user interface extensions is a breeze. Creating the Earth Navigator application was all about thinking how the Navigator works and gets the desired data or map using NASA jar files and application programming interfaces. One does not have to waste a thought on the platform itself or portability issues anymore while working on the views and editors. The graphical elements were implemented using standard widget toolkit. The extension mechanism allows the integration of other optimization algorithms without recompiling the application. The single steps of the framework are included as single pages in a tabbed editor. Furthermore, the Eclipse forms layout could be integrated successfully.

4.1.2 Plug-in Architecture
This section highlights strengths and weaknesses of the Eclipse plug-in architecture.

4.1.2.1 Strengths of the Plug-in Architecture
Java Development Tools (JDT) and CVS part of the Eclipse platform's plug-ins, provide us all the facilities to creating this functionality from scrape. It provides us with valuable information on software artifacts. In this way it releases the problems and troubles of a developer who is working on the framework of Eclipse. As the tool developed in this manner, is now part of a full-featured Java IDE, this provides opportunities to developer to focus on application's visualizations which plays a great role at the side of end user. It is necessary to think about the complete architecture of application before the process of integration so that there will not be any problem when it get plugged into some another environment [35].

Eclipse has a multi-level plug-in architecture whereas most plug-in systems only provide for one level of plug-ins. Applications like Adobe Photoshop or Windows Media Player make it possible to extend the framework with self-written components, but there is no generic way to allow the extension of these components. On the contrary, Eclipse itself is realized as a couple of plug-ins that is written around the core plug-in. Additional components depend on this primary plug-in and are therefore secondary to them. In Eclipse, there is a great feature of extension-points that other plug-ins can connect to any other plug-in.

Additionally, Eclipse requires explicit ports. These ports or extension-points to a component are defined at manifest file which is provided in that plug-in. The manifest file is a specification of optional requirements. By providing an extension-point, the client declares that it supports any extension that adheres to the defined interface. The advantage of making the ports to the component explicit is that it is easy to see where a component may contribute to the system.

Eclipse plug-ins is self-descriptive. They provide a manifest-file whose content is partly redundant to the information in the code. The redundant information is given to improve the performance because it allows the platform to know about essential properties before it loads the .class files.

Another specific of the plug-in model is that it uses containers to encapsulate components, so the single plug-ins is strictly separated from each other. If a plug-in is not declared as required by manifest file of another plug-in, it cannot be accessed. To realize this, every plug-in has its own
classloader that verifies whether an attempted access to another class is legal. The single classloaders realize a container concept and avoid that undocumented and unintended dependencies between plug-ins come into existence.

Plug-in systems are developed in the way so that they can provide the benefits from the following advantages [36]:

- Stability of system design. New features are added through plug-ins, independent of the core functionalities of the application.
- Reduced frequency of context switches. The user remains in the same integrated environment, experiencing a feeling of continuity.
- Increased usability. The user does not need to learn to use a new environment for the system functionality.
- Re-usability of framework functionality. Basic shared functionality is provided by the framework, so liberating the plug-ins from assuring it, reducing complexity and increasing modularity and understandability [37].
- High flexibility in tool customization. The user can select exactly the plug-ins tailored to her needs.
- Interoperability. In many research communities, all tools are developed using the same framework.
- Easy extensibility [38]. New tools can be added without the need to understand the framework code. Extendibility [39] is defined as the degree of usability and safety in contexts beyond those initially intended. Extendibility includes, but is not restricted to, extensibility.

4.1.2.2 Weakness of the Plug-in Architecture

No strict requirements are possible. The Earth Navigator tool would not make sense without at least one optimization plug-in. It should be possible to plug alternative engines, which are for example optimized for a specific platform, into the application. But there should at least one to enable the full functionality. In this case, the plug-in does not declare an optional extension but a
required component that has alternative implementations. Eclipse does not support such required components.

Another weakness of the plug-in architecture is that it does not support explicit connectors. Every plug-in extends an extension-point of another plug-in. As soon as it is plugged into the platform, its functionality is available. Explicit connectors would allow installing a plug-in without exposing its functions. It would only be available if it is connected to the extension point of another plug-in. This would allow the installing of various optimization engines, for example a fast and imprecise and a slow and precise one, and connect only one of those.

### 4.1.3 Comparison of Linux and Windows Platform.

The testing results are based on an Intel Pentium 3.06 GHz laptop computer, which has 768 MB of RAM and hyper-threading support. The operating systems that are available on this computer are Windows XP Home Edition with Service Pack 1 and Debian Linux based on the Linux kernel 2.4 with KDE V3.1 as windowing system.

The average start up time on both platforms is about 13 seconds, so there is no human recognizable difference. So standard widget toolkit can be used to create graphical, sophisticated and demanding portable applications. The response time of the standard widget toolkit widgets is the same on both platforms; only the look-and-feel is specific to either the Linux GTK framework or the Windows windowing system.

### 4.1.4 Standard widget toolkit vs. Swing

Since IBM released Eclipse and with it the Standard Widget Toolkit (standard widget toolkit), it came to lively discussions about which framework, standard widget toolkit or Swing, is the better one. Never ending articles, postings in forums and news lists could be read about this topic. Fact is that there is no simple answer to this question. It depends on the requirements of the application, the experience of the development team in one of these two frameworks and the target group to make a decision on which one should be used.
At first, standard widget toolkit may seem simpler because it does not require the developer to get acquainted with the Model-View-Controller (MVC) pattern. Swing comes with more sophisticated features, is more elegant, easier and more flexible when it comes to building complex Graphical user interfaces. There is one big problem of standard widget toolkit is that the management of resources. Resources, such as fonts, colors and images have to be freed manually to prevent memory leaks in the application, whereas Swing follows the Java paradigm of automatic garbage collection. This is a serious point for the average Java developer who never heard about disposing objects and may directly reflect in the project costs. In addition, the application programming interface is not that extensive. Application programming interfaces like the Java2D package are completely missing.

Standard widget toolkit has a very tight integration with the operating system. If one considers writing Java applications for a certain platform, standard widget toolkit can be useful in terms of invoking native features such as ActiveX or OLE automation. As a matter of fact, standard widget toolkit is, because of its thin pass-through layer to the native widgets, faster and more responsive than comparable Swing user interfaces. This is useful when the target hardware is quite old and does not perform too well.

4.1.5 Problems

The platform is still under massive change and evolution, and for the reason that the application programming interfaces change between the versions, developing plug-ins for a certain milestone may result in heavy rewriting efforts when a new Eclipse version is released. Worst of all is the lack of documentation.

The first standard widget toolkit book is about to be released and the existing Eclipse books describe Eclipse itself and the plug-in developing process, but not how to handle the Rich Client Platform. Exactly three articles exist about the Rich Client Platform, but there are no examples of documented applications. In the end, some of the problems can only be understood and solved by browsing the Eclipse source code.
4.2 DISCUSSION

Before discussing our research, we would like to recapitulate our research outline in the following diagram (Figure-4.1)

![Diagram of Research Outline]

Figure 4.1 Research Outline
4.2.1 Web Usability vs. Desktop Usability

Compared to the relatively well explored fields of both web usability and Desktop user interface design, there has been only little work done to support web developers in designing usable Rich Internet Applications. One reason is the novelty and ever-changing technology for developing such applications with new AJAX frameworks emerging constantly and new features made possible for web applications. Developers, facing new technologies and having no guidance how to use them, design interfaces for web applications that are neither consistent with other web applications, nor with existing desktop operating systems. The result is that users have to learn the features of every web application they encounter and they cannot transfer their knowledge from one web site to another. In case of desktop applications, this would be a violation of the consistency principle, but as every web application has its own interface, users have learned how to interpret interface elements that look differently but have the same function, like site navigation, dynamic menus, hyperlinks, etc. This is something that Rich Internet Applications don’t communicate to the users, so they have to explore the interface in order to learn the application’s features. Some features are easier to communicate than others, because they resemble desktop user interface counterparts, but certain behaviors, like drag & drop were never before seen in web applications and it is unlikely that users will try to explore them, unless they are told to do so. New ways of communicating such features will have to be found and applied consistently on many websites. As an example, there exists some common understanding of how to communicate a web link to a file to be downloaded. In most cases it is indicated by a downward-facing arrow. Though the icon is never exactly the same, the meaning seems to be clear to the users and web designers and it seems to be universally recognizable. If the same development happens with Rich Internet Applications, there is a chance that although having completely different visual appearance, they will be universally understandable.

4.2.2 Perception of Guidelines by Developers

In general, guidelines have been found useful both by researchers and practitioners. Our literature study has revealed that guidelines are successfully used at various stages of development and they exist in a variety of forms. Our case studies have also shown that
developers appreciate having guidelines, however they prefer them to be on a general level, to be universally applicable and not constrain developers in their creativity. Guidelines therefore should not be solely targeted at achieving consistency, but on the overall user experience, which is in fact difficult to define, but the best solution to the problem is formulating guidelines on a high level. We think however that guidelines should be formulated on a level understandable for the average developer, who has no background in cognitive psychology or graphic design. In fact, guidelines are meant to fill this knowledge gap, without the developers having to know why they work. High-level guidelines can still have useful character, but in our opinion they don’t need to legitimate their existence to the developers.

Another interesting aspect found during the interview with Netvibes is the fact that developing guidelines should be done with a bigger audience in mind. Originally, Netvibes’ widget guidelines were targeted at in-house developers, but by opening the application programming interface to third party developers, these guidelines could be used in a much bigger scope. A similar development took place with Apple’s Human Interface Guidelines (Apple, 1992), which became an important reference, not only to Apple developers, but also to web designers and desktop application developers on other platforms. The reason for that were the universally applicable principles that apply to any desktop (or post-desktop) platform. This is another argument for formulating guidelines on a general level. One important element desired by our interviewees was including examples for rules as they help to understand the rules better and it is always a good learning experience to see how something can be solved.

4.2.3 Accessibility of Rich Internet Applications

Then, there is also the problem of accessibility of Rich Internet Applications, which is connected to usability, but has a far more severe impact on people with disabilities that the usability problem has on regular users. Users can work their way around a system with poor usability if they have to work with it (and if not, they will use a competitor’s system), but when a disabled user has no access to information at all, this is the same type of exclusion if a public library would have no wheelchair ramp. In the case of Rich Internet Applications the solution is not straightforward, because appropriate standards are still in development. Until then, an accessible
alternative solution should be provided for equal access to information and services. Also, the awareness of developers for accessibility issues has to be increased and this can be only achieved if accessibility becomes an integral part of RIA guidelines.

4.2.4 Distinctive Features of Web vs. Desktop Guidelines

Regarding the question, why there are no guidelines yet existing for Rich Internet Applications may have several answers. We think that the most probable answers the fact that the related technologies are very young and so are the applications built with these technologies. There is a lot of experimenting going on with Rich Internet Applications and it is not yet clear which solutions are the best ones. If a part of a page has been updated without reloading, then how to indicate the changes? How to communicate that the application has lost the connection to the server? These problems have no relevance in desktop applications and the solutions are still being explored. There are some issues that can be directly transferred from desktop user interface guidelines to web applications, others are completely new and yet others are irrelevant. In order to design Rich Internet Application guidelines, one should compare desktop guidelines with web guidelines (regarding different levels of specificity) and find common and distinct features in both of them, and check their applicability in Rich Internet Applications. Previous research has shown that Web guidelines have very little in common with desktop user interface guidelines (Ratner, 1996), and it seems obvious, since the desktop and the web are two essentially different platforms. However, with Rich Internet Applications gaining ground, it makes sense to learn from desktop guidelines.

In contrast to Desktop Interface guidelines, which are structured quite similarly and address the same topics for various operating systems, web usability guidelines can differ very much in the level of specificity and can address web usability from many perspectives. Also the border between web sites and Rich Internet Applications is vague: a simple JavaScript for revealing elements of the page could be already considered a Rich Internet Application in the same way as a complex Web application using desktop-like patterns of behavior but not using any dynamic web technologies could be categorized as a “classic” Internet Application.
In our case studies we found out that developers don’t follow guidelines very strictly, all of them have read usability guidelines in some point of their work, but they rarely refer to them, because they have experience doing development work and they intuitively apply solutions that work best (so their claim). The question is if the work results confirm this statement or if guidelines are written in a way that is not suitable or attractive to web developers. An analogous study has been conducted with accessibility guidelines and it has been found that developers had troubles implementing them correctly.

The fact that Rich Internet Applications develop so rapidly could be a reason for authors to restrain from publishing work that could be outdated in a few months. Online magazines for web developers publish and discuss ideas related to interface design of Rich Internet Applications. As the discussion elements seems to be very important in this early phase of guideline creation, a suggestion that we got from one of our interviewees makes sense: creating the guidelines not as a linear document, but as a wiki, which can contain articles that anybody can discuss about and contribute new solutions on the fly. Once a stable set of guidelines is established, one could move on to creating a compendium as a static document. A wiki is often the preferred documentation and communication tool for developers, and we believe that communication between developers is particularly important in this stage.

During our work creating a set of basic principles for Rich Internet Application guidelines we couldn’t work closely with developers and users, therefore we decided for a literature review and a case study to find out how developers work and how they envision possible Rich Internet Application guidelines. We learned that web applications are so diverse and multifaceted that it makes much more sense to discuss general principles and issues than trying to solve concrete problems of how to design widgets. We believe that a top-down approach is more suitable, because first one has to gain understanding of the principles and the context of Rich Internet Applications and only after doing that it makes sense to go into detail. As we believe that Rich Internet Applications are not yet fully understood by users and as long as developers are still exploring the possibilities, the understanding of Rich Internet Applications is resides in the shaping of the process. With our outline for RIA guidelines we would like to give suggestions
and point out issues that we believe are important to consider while designing Rich Internet Applications.

4.2.5 Guideline Development Methodologies

We were considering employing one of the methodologies for creating guidelines but they turned out not to be suitable for our needs, as they were intended for creating specific guidelines or to be employed in an organizational context. We therefore decided to create our own approach, taking interaction design aspects from existing desktop usability guidelines and content organization aspects from web guidelines. We also incorporated accessibility, Web 2.0 and mental model aspects from our literature review, all of which were confirmed as important elements by the developers. As we touch upon new ground within Rich Internet Applications, we believe that this “pragmatic” approach will produce a guideline outline, which can serve as a basis for further research.

When Rich Internet Applications will reach a sufficient level of maturity, we expect patterns to emerge, which will represent a consensus in how the user interface should look like and behave. The question whether guidelines are necessary for these patterns to emerge, or if the patterns are caused by guidelines is a typical chicken/egg problem, but we think it has also to do with a critical mass of applications being developed and actively used and shaping the understanding of the usage we mentioned before. In any case, guidelines could further ensure consistency within and between Rich Internet Applications. Novice developers would get an understanding for the context of use and mental models users have of particular types of sites and interaction modes. We again would like to stress that we believe high-level principles to be more important here, as they exhibit a general understanding without constraining the developers in their creative work.

Take a shopping web site as an example:

There is metaphor of a shopping cart users can put products in and stroll around the shop while the shopping cart still contains the products. To buy the products the users use a “checkout” site where they choose a payment method, enter their address details and finalize the transaction. The metaphor of a shopping cart and the mental model that items are stored in a (virtual) shopping
cart during a session on a shopping website has become the way how all shopping web sites work. If a new shopping web site is being developed, this mental model should be kept in mind if one wants the users to use the shop intuitively. One could introduce new ways of placing the items in the shopping cart (i.e. by drag & drop of the representing pictures), but the mental model of a shopping cart still remains unchanged.