Application Fields of the Lyee Technology - Comparison of Lyee and Catalysis

Volker GRUHN, Raschid IJIOUI, Dirk PETERS, Robert QUECK, Clemens SCHÄFER
Chair for Applied Telematics / e-Business[*],
Universität Leipzig, Klostergasse 3, D-04109 Leipzig, Germany

Abstract.
Throughout numerous analyses and proposals of the capabilities of the Lyee methodology for software development there has been done a lot of work concerning how Lyee works, what its benefits are, and what its perspectives will be. Most of the time this has been done in relation to automated software generation. But concerning all the proposed benefits and the announced major breakthrough Lyee would be for software development, Lyee has also positioned itself in direct concurrence to traditional, already established software development models.

The intention of this article is to put the Lyee technology into perspective to established software technologies, which is done at the example of a comparison of the Lyee and the Catalysis approach for software development.

1 Introduction

1.1 Motivation

Throughout several proposals and articles during the last 3 years the Lyee methodology (GovernmentaL MethodologY for SoftwarE ProvidencE) and its theoretical background of Requirement and Intention Engineering has been announced as a fundamental new approach to install a linkage between the intentional requirements for a computer based assisting system and the final outcome of a development process, a software product [NEG00]. Within these proposals the Lyee methodology very often puts itself into relation to established software development technologies and frameworks such as DOA (from Object Management Group) [NEHA2001a], and "traditional software development" as a whole [NEHA2001b, TOM02]. These articles characterize the Lyee methodology as profiting from two major aspects, which will be described in more detail in the introduction of Lyee within this article. The first one is the underlying "metaphysical" model or ontology invented by Fumio Negoro [NEG02]. The second is the actual way of developing software utilizing a so called "Signification Vector" as algorithm for automated software generation [NEG02, INT01].

To use the Lyee methodology in praxis there is obviously a need for a tool that realizes this automatic generation process. This tool is called LyeeAll. These three components of the Lyee methodology are the basis of the claimed improvements of using the Lyee methodology, namely a shorter period of development

*The Chair for Applied Telematics / e-Business is endowed by Deutsche Telekom AG.

time, a drastical reduce of documentation effort, the standardization of the development process, highest level of automation, reduced effort for tests and verification, avoiding "spaghetti" programs, implementation of human thinking, and simplified logic to establish software [NEG00, NEHA01a, TOM02].

As it is stated in many proposals and as it comes apparent from reviewing the Lyee development process (see Introduction section) there are fundamental differences in the design and development approach compared to "traditional" development methodologies as e.g. DOA, OOP, and Catalysis. Just to give one example, the extensive and profound documentation throughout all development stages is the main basis of most of these appointed approaches whereas Lyee claims to reduce this drastically [INT01]. Considering these fundamental incompatibilities and the wholehearted announcements a view of Lyee as a universal software development methodology is imposed [e.g. TOM01] – the identification of Lyee as a "methodology" supports this impression.

Because of this self-positioning and the rather propriety approach of Lyee software development it gets necessary to evaluate Lyee in direct comparison to these established single-standing methodologies. With a comparison of Lyee and Catalysis – quite a typical and up to date representative for the object oriented and component based software development approach - this article wants give an example of such an evaluation. To designate a ground basis for this analysis one can say that both methodologies are claiming to gain profit out of the standardization of software development, although in a different way. This seems to make them very appropriate to compare. Inside this article it is only possible to give an access point to this comparison. A more profound analysis would go beyond its scope and is intended to be done in later work.

To get a better picture of the two different development technologies several figures are presented. These figures show small outcuts of a development process of an insurance system (**see Figure** 1) as introduced in by Gruhn [GIPSKA02] both for Lyee and Catalysis.

Illustrating example – Insurance System [source: GIPSKA02]

Figure 1: main frame of the application [source: GIPSKA02]

1.2 Lyee

The Lyee methodology of software development was invented by Fumio Negoro and consists of two rather independent parts, an underlying meta-model and a more practical part consisting of algorithms and a necessary tool for their realization. The underlying meta-model is very often described as being complex and using obscure terms [PFI02]. The model itself can be situated in the neighborhood of ontologies and philosophical descriptions of the world.

Caused by this independency one can use the practical deductions from the meta-model such as signification vectors and the LyeeAll tool without concerning the difficult to understand model to much. Since there is nothing comparable to the Lyee meta-model in Catalysis and since this article is concentrated on the practical process of software development, just the very basic ideas of the model will be given here. The meta-model of Lyee is focused on the connection of intentions and what might be the outcome of modeling as necessary for software development. Therefore it distinguishes between two spaces, the physical and the non-physical world. While trying to transfer (user-) intentions, several real world objects which can be described using the natural language are designated. Once outlined in natural language these objects have unbreakable connections to objects in the non-physical space that already have existing memories. These objects consist of so-called consciousness atoms. The goal of the modeling process is to establish a cognition object which, in terms of the model, is situated at the intersection of the physical and the non-physical world and consists of cognition atoms [NEG00]. As one goes further into the theory its descriptions often involve unnecessarily obscure terms and unnecessarily rigid hypotheses [PFI02], which makes it hard to adopt the model unconfined. For a more detailed view it is referred to proposals directly introducing the meta-model and proposing Intention Engineering [NEG00, NEHA01a, and NEG02].

As the very basic principle of the Lyee methodology the so called “signification vector” and its implementation can be seen. This vector is a representation of Lyees basic algorithm for code generation. At the beginning, given the objects (in Lyee terminology: words) named by the user and there definitions (Lyee: requirements) (**see Figure** 2) a calculation network is created automatically which connects the necessary object inputs with outputs of other objects (**see Figure** 3). Missing definitions or errors are reported to the user and can be fixed throughout this assistance. The finally generated code implements the objects, their requirements, and the signification vector algorithm. During execution time one instance of this algorithm is called for each object in an unspecified order. Within this call it checks whether all the attributes of the objects definition are already calculated or not. If not it will be tried to “significate” another object first which continues until the whole network is calculated [INT01].

A practical implementation of this source code generation algorithm is given in the LyeeAll tool which can be seen as the one IDE for Lyee developers. Beside the standard source code generation process, the LyeeAll tool also allows to include so called “boundary software” into Lyee programs. These are code fragments or bigger peaces of software created outside of the Lyee environment. The use of boundary software is very often unavoidable since Lyee is not capable of fulfilling the whole bandwidth of software

development tasks as will be discussed later. Typical examples of boundary software are GUI implementation, complex database queries, little code peaces for object/word definitions and conditions, and more.

Because of its automated source code generation capability Lyee claims to have substantial benefits over traditional software development methods, such as 80 to 90 % reduced man-hours and therefore development time, as well, 90% reduced document volume, and 90 to 99% reduced man-hour for program maintenance [INT01].

Screen Words

WordName	WordID	Data Type	Length	Object ID	Property ID	Array H	Array V
cmdMenu1	cmdMenu1	B		cmdMenu1	01		
cmdMenu2	cmdMenu2	B		cmdMenu2	01		
cmdMenu3	cmdMenu3	B		cmdMenu3	01		
cmdExit	cmdExit	B		cmdExit	01		

Figure 2: Lyee - Screen definition of the frame [source: GIPSKA02]

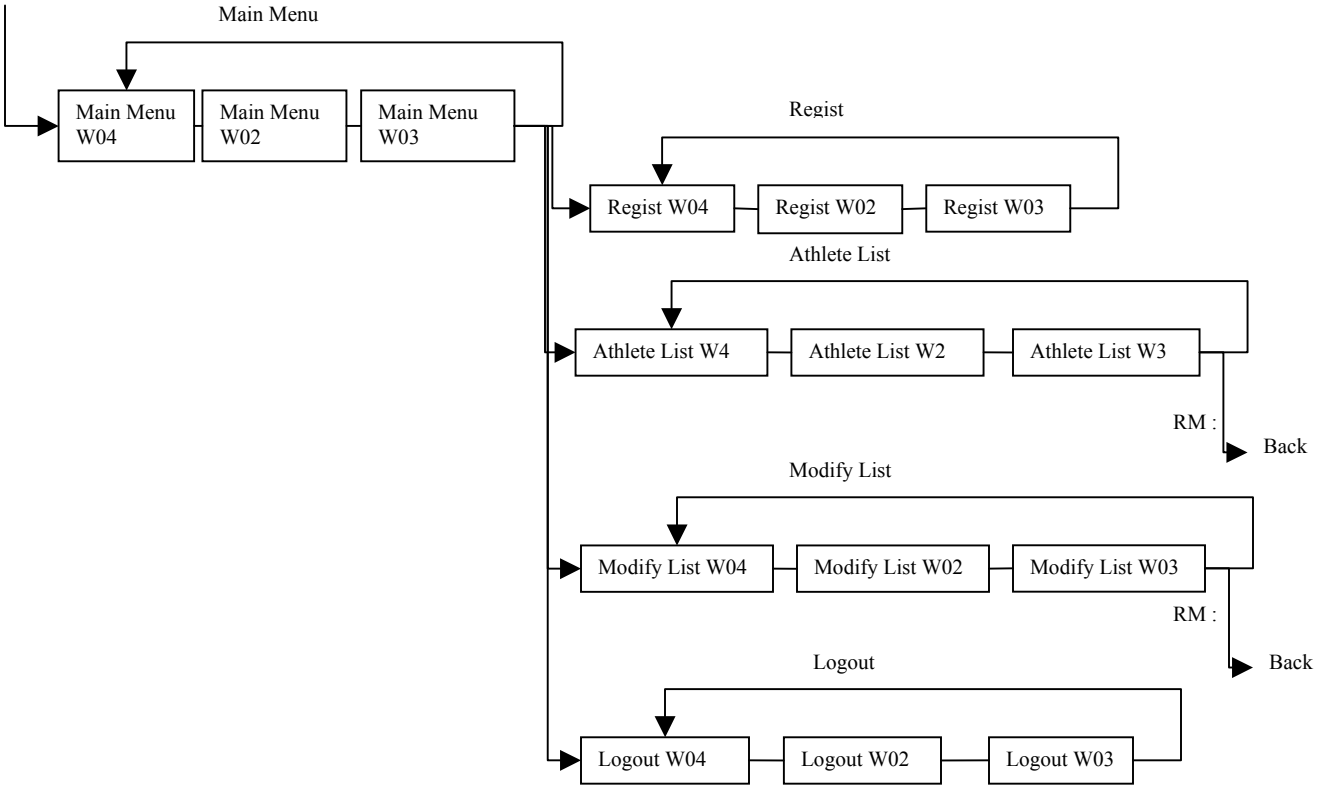

Figure 3: Lyee - Process Route Diagram as created from the LyeeAll tool[source: GIPSKA02]

1.3 Catalysis

Catalysis, as described by d'Souza and Wills [dSWi01] is a software development methodology for component based and object oriented software development. Grown out of experiences with former object oriented methodologies the authors put much emphasis on integrity, team development, and flexibility of the final software product [dSWi01 p. xv]. The main weapons to fulfill these requirements are a sophisticated design framework, enforcing and supporting several sequential and parallel development layers, extensive documentation focusing on the reasons for design decisions rather than on the decisions itself, and unambiguous standards for design models as well as component and object designs. In comparison to Lyee the Catalysis approach does not produce any source code itself. It acts more as a guide and supporter for the software developer. But because consisting of "borrowed" standardized techniques and notations there are several tools such as Rose, Select, or Rhapsody which support the Catalysis approach also by having implemented some automated tasks to assist the developer [dSWi01 p.510].

There are several ways through the Catalysis methodology adapting the size and other characteristics of the project to be [dSWi01 p.510]. As a typical example for a top-down approach the following sequence of development layers is given: business modeling (**see Figure** 4), requirement specification, component design (**see Figure** 5 and 6), object design (**see Figure** 7), and constructing a component kit architecture. Since abstraction and refinement are big issues in Catalysis the order of this sequence is regarded more as an assisting guide and the development process is considered more as a nonlinear, iterative, and parallel one [dSWi01 p.510].
For a more theoretical introduction to the Catalysis approach it is referred to the introducing scientific work [e.g. dSWi95, dSOU01, and related work].

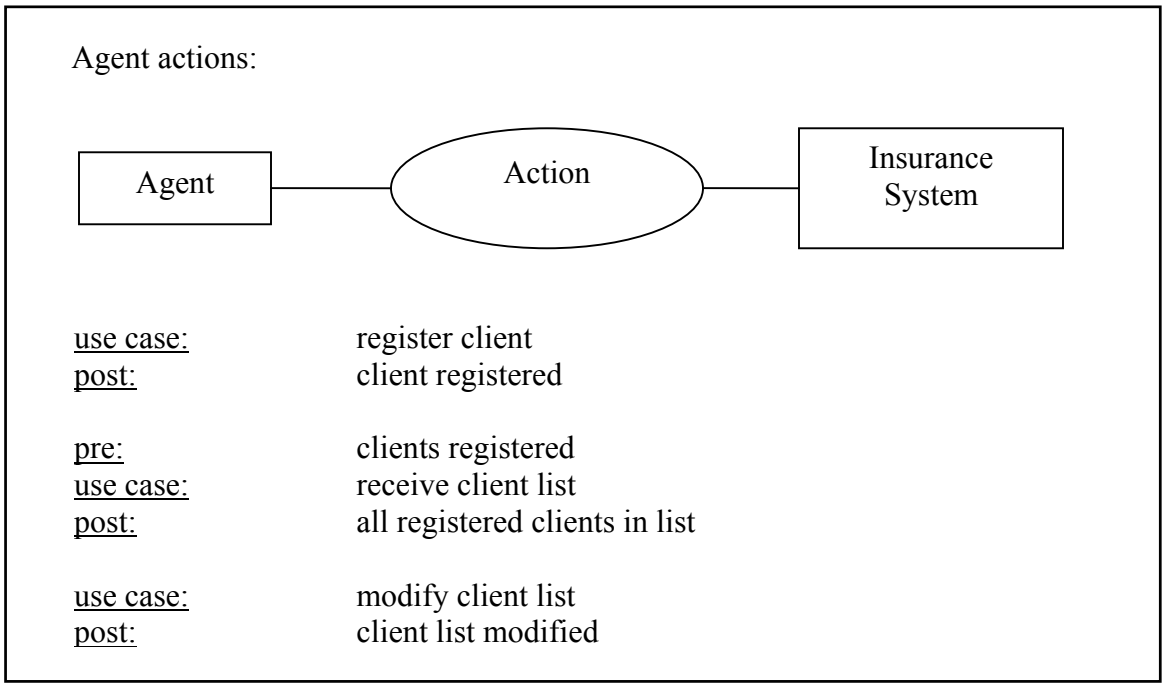

Figure 4: Catalysis – cutout of a Business Model

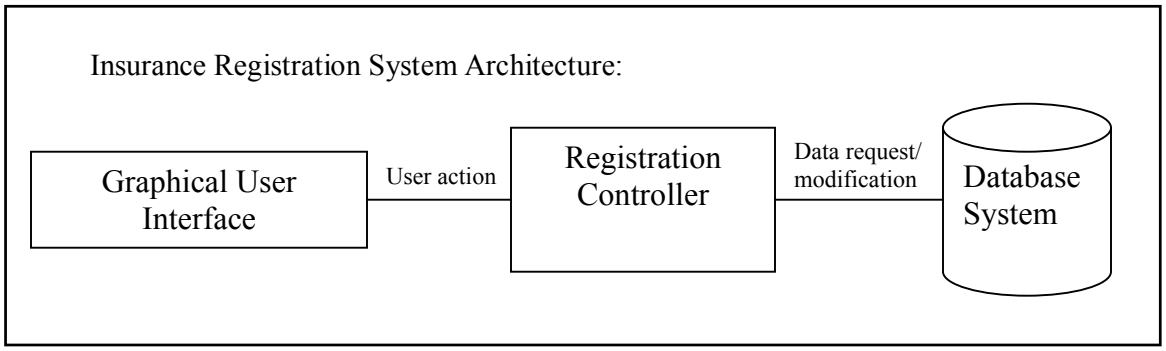

Figure 5: Catalysis - Component architecture

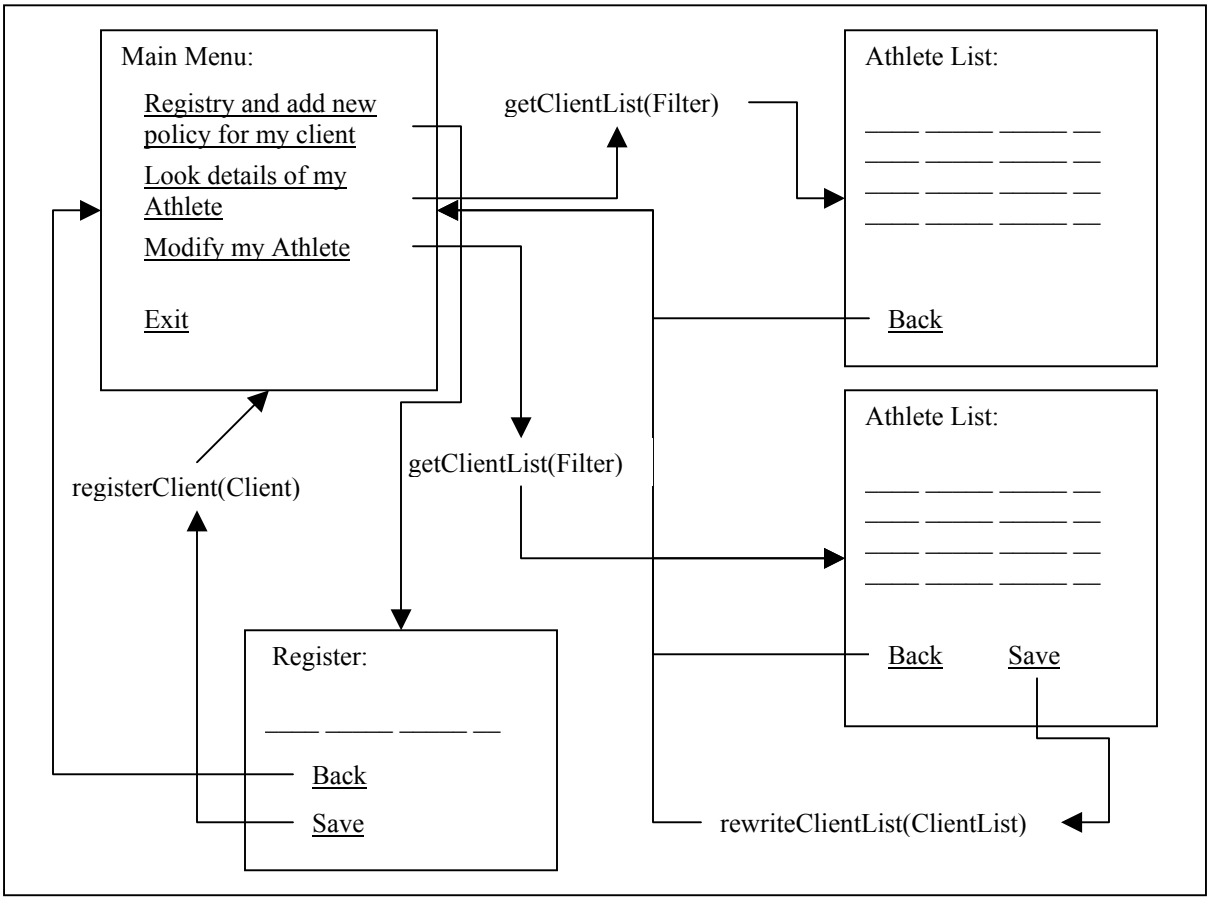

Figure 6: Catalysis – cutout of a user interface scatch

Figure 7: Catalysis - Class diagram of the registration controller

2 Comparison

2.1 Powerfulness and Complexity

As mentioned in the introduction section the execution of a LyeeAll generated program strongly bases on the algorithm for the signification vector. While this algorithm is the source of the automatic code generation features of Lyee, it is also the cause of weaknesses in terms of the complexity of generateable algorithms. The program structure determined by the signification vector algorithm is similar to functional terms and therefore restricts the algorithmic powerfulness of Lyee generated programs to linear computational complexity [MAL02]. On the first view this restriction seems to be a very dramatic, but actually the majority of computer-aided applications (at least parts of them) is performing rather simple algorithms. Therefore, if it comes necessary to provide a Lyee generated software product with more complex algorithms Lyee gives the opportunity to include so called boundary software. Speaking in terms of development cost this can cause additional investments into human recourses, since at least some percentage of the development team has to be able to deal with other development methodologies to design, to implement, to test, and to maintain these non-Lyee parts of the final software product.

Other problems arise when trying to include parallel algorithms into Lyee programs. Since up to now the signification vector implementation does not support parallelization the only chance would be to import the critical parts as boundary software which could cause even bigger performance losses. Further more flexibility and tunability of paralyzed code is restricted by the fixed algorithm of code generation [MAL02].

Another problem of Lyee caused by the execution structure of the signification vector is its inherent danger of performance loss. Since Lyee does calculate interlinked object/word values in a non-sequential iterative (perhaps even randomized) algorithm, there is a chance that in case of many cyclically connected variables the processing time of Lyee programs might follow an exponential complexity related to the number of attributes/words to deal with. Nowadays, where computational resources with lower costs are available more easily, this might be more efficient than developing proper adapted algorithms, although there might be a critical line [MAL02]. Additionally in time critic and customer contact applications it gets more and more standard to include load-balance and performance guaranties into a software contract.

All these restrictions of Lyee generated software have to be considered as smaller chances or prize reducers for Lyee produced software products compared to software

developed using other technologies and not having these restrictions even though they might have bigger costs for the development. They also shorten the applicable fields of Lyee software.

2.2 Design and Documentation

As quoted before it is a major intention of the Lyee methodology to reduce the need for design and especially for documentation. One can see at the example of Catalysis, that hierarchical design phases and profound documentation are major pillars in traditional software development. Since the traditional methodologies have evolved over decades and their must be good reasons for developers to put that much emphasis to these rather implementation accompanying tasks. Therefore it seems to be legitimitated to review the reasons for this evolution and to question whether they are still existent for the Lyee methodology or how Lyee deals with them.

For beginning with the designated competitor, Catalysis uses its extensive design framework mainly for one reason: to deal with complexity. Additionally one say that a good design also supports the documentation of the software product, which is another problem and will be referred to later on. According to the philosophy of catalysis there are several decisions to make during a development process. Many of these decisions occur at different stages of the development, e.g. architectural decisions for the whole product earlier than refinement decisions for a single component. The purpose of all the design guidance and all the models is to assist the developer at the time as these decisions arise, in the order of their importance. This is also the reason why Catalysis imposes more than one design stages, each one an abstraction layer of its successor. This strategy is seen as to be crucial because of the complexity of many of today's software applications [dSWi01 p.37]. Therefore a typical first development layer for the Catalysis approach is creating Business Model (often directly followed by the Requirement Specification), which aims at modeling the real existing world. Once created, this model can assist in possible stages of requirement specification and component design [dSWi01 p.10].

Passing the ball to Lyee, there does not exist anything similar. Although there is an awareness of the need for something like project planning and outline design [NEG00], according to Lyee documentation, the development process starts with transferring the user requirements or intentions into the LyeeAll tool. But in many applications, especially in more complex ones, the user does not exactly know what he or she wants [PFI02]. Fortunately there is no objective reason why Lyee cannot adopt parts of other design frameworks, despite philosophical contradictions perhaps. The other possibility would be to provide Lyee with its own design framework, which may be could be more compatible to the Lyee framework. There is some work going into this direction [e.g. ROL02] but it is rather theoretical and so far none of it has found its way to common acceptance. Technically seen there is little reason that the Lyee methodology is incompatible to design frameworks at all. Although the creation of such a framework might be a task rather not necessarily to be done by Lyee supporters its benefits for the Lyee methodology as a product would be very important.

As already mentioned another outcome of design is a better documentation of the software product. This approach of documentation through modeling leads as a thread through the whole Catalysis development process. Additionally Catalysis outlines the importance of an effective documentation preferring to illustrate the rationale for decisions rather than the decisions itself. As another argument for this documentation style the authors proclaim the need to communicate the intentions for decisions made throughout the development process, either to collaborators or redesigners. Further more, according to d'Souza and Wills, the documentation process would help to identify open questions in earlier development stages [dSWI02]. Prima facie Lyee seems to have right the contrary philosophy since it campaigns with a drastical reduce of documentation effort [INT01].

Speaking for traditional software development one can say, the more complex the project (in terms of code lines, number of participating developers, and required flexibility for redesigns) the more efficiency relates to the quality and amount of documentation. Lyee instead, aims at automated code generation and therefore takes decisions off the shoulders of developers which would justify less need of documentation for both, collaboration and redesign. At this point it is hard to accept a universal estimation of the profit out of this effect, which is caused by differentiating project characteristics and the varying quantitative and qualitative documentation philosophies in traditional software development.

Lyee sources proclaim up to 90% of reduced documentation [INT01]. Contrary to the automatic source code generation there are some points that support an estimate closer to the lower end of this interval. First, as mentioned afore Lyee does not have the full power of other programming languages and therefore depends on boundary software, especially for rather complex algorithms, which has to be documented on its own. One can say that the bigger the size of a software product the bigger the percentage effort for design stages not supported by Lyee, which would lessen the overall save. Even though automatic code generation reduces the possibility, there are still sources of errors or other problems while developing Lyee programs, e.g. performance lacks (see also next section). To localize and fix these errors efficiently requires more documentation, also for parts related to Lyee code generation.

2.3 Refinement and Software Reuse

One answer to the question how to minimize the costs of software projects is software reuse. This means including parts of already developed software into existing projects. For producing software capable of reuse Catalysis suggests to put much effort into documentation on the one hand and refinement stages on the other hand. Especially abstraction is very often referred to as one of the main principles. Abstraction, in the eyes of the authors, means to describe only the necessary. This does also include shortening already produced design, code, or documentation during a refinement process [dSWi01 p.37].

The reason for this more and more established behavior is to organize design and implementation to a maximum degree of understandability, maintainability, and reusability. Software reuse is also one of the main motivations of component based and

object oriented development methodologies. This does not only include software itself, it rather includes all products of each development stage to benefit from the additional documentation and refinement work each time a similar task or process needs to be handled. Representatives for reusable software to be are standard packages, class libraries, design patterns, component libraries, and, in the case of Catalysis, also model libraries and model frameworks [dSWi01 p.380].

Having a look at the Lyee software methodology, refinement and software reuse is not addressed in the same way as it is in Catalysis. One reason is that since a big amount of the development process is automated, it is not as necessary to spent effort in here. Actually the LyeeAll system can be seen as a big reuse of software itself since it implements software making use of the same pattern, e.g. the signification vector, many times. Further more the Lyee template mechanism allows the developer to adjust the code generation process to different environments and to different languages. Another opportunity of software reuse in the Lyee methodology is the possible import of specification out of text files and therefore the reuse of "requirements". Having a good documentation this feature of the LyeeAll tool seems to be very helpful. Obviously there is no support for reuse of earlier development stages as business modeling and design since they are not a part of the Lyee methodology as already discussed.

Very interesting in terms of software reuse and interoperability is the work from Jakobssen that aims at possible use of Lyee software in component based environments. He points out that up to now it is not possible, to model component like behavior with the process route diagrams as they are. Seeing the importance of the component based software in todays e-commerce systems and business like applications, he proposes an extension of the process route diagram to make Lyee programs more suitable for a component based approach [JAK02].

3 Conclusion

At the beginning of this paper a short introduction into Lyee and Catalysis has been given. Afterwards a comparison suspending to three major aspects of software development was performed. Throughout this comparison it has been pointed out that although having similar pretensions there are some restrictions of the Lyee System that prevent it from being considered as a software development methodology capable of addressing all aspects of the development process. Especially the lack of a multi-level design framework, its algorithmic weaknesses, the inherent danger of performance disadvantages, and the unused potential of interaction with software of established technologies such as component based development restrict the Lyee methodology rather to small or medium-size applications with little complexity in terms of project facilities and algorithms. To accept Lyee as a full featured software development methodology there is an urgent need for a valuable design framework, included in the LyeeAll tool at its best. Other disadvantages related to the basic structure of Lyee programs seem to be persistent in the future. Nevertheless the Lyee methodology gives the opportunity to automate parts of the software development which simplifies the process, causing the consequences of fewer investments into human resources, less error sensitivity, and a faster development process. But in consideration of Lyees inherent restrictions it seems to

be a serious overrating to expect it to revolutionize the software development, although the Lyee technology might find its place in the area of the often quoted business-like applications, information systems, or other form-based systems. With good reasons this is also the area where most of the examples of Lyee developed systems are situated.
The sinlge-standing applicability of Lyee is in fact a very crucial part in terms of an adoption of Lyee in business software projects. Projects that have to make use of more than one development methodology will, despite of the problems caused by conflicting philosophical foundations, suffer additional costs in project structuring, development tool licenses, and human resources since at least some parts of them have to be laid out twice. These additional costs might as well eat up all the benefits gained from using the Lyee methodology at its strengths. There are two directions to go to make the Lyee methodology more attractive for business projects. The first is to provide the Lyee "product" with components that make it capable covering the whole development process. The other one is to make Lyee more flexible in terms of interaction with other established methodologies. This would make it easier to switch to the Lyee methodology and would also provide more security in terms of development investements.

References

[dSWi01] Objects, Components, and Frameworks with UML – The Catalysis Approach, Desmond Francis d'Souza and Alan Cameron Wills, Addison-Wesley, 3rd printing 2001

[dSOU01] Model-Driven Architecture and Integration, Desmond Francis d'Souza, 2001, source: http://www.catalysis.org/publications/papers/

[dSWi95] CATALYSIS – Practical Rigor and Refinement, Desmond Francis d'Souza and Alan Cameron Wills, 1995, source: http://www.catalysis.org/publications/papers/

[GIPSKA02] Software Process of the Insurance Application System by using Lyee Methodology, Volker Gruhn, Rashid Ijioui, Dirk Peters, Clemens Schäfer, Takuya Kawakami and Makato Asari, Lyee_W02, 2002

[INT01] Introduction to Lyee, The Institute of Computer Based Software Methodology and Technology, Lyee Information Service, July 2001

[JAK02] Extending Process Route Diagrams for Use with Software Components, Lars Jakobsson, Lyee_W02, 2002

[MAL02] The Key Features of the LyeeAll Technology for Programming Business-like Applications, Victor Malyshkin, Lyee_W02, 2002

[NEG00] Principle of Lyee Software, Fumio Negoro, International conference on information society in the 21st Century (IS2000), 2000

[NEG01] Intent Operationalisation for Source Code Generation, Fumio Negoro, World Multi-Confernce on Systematics, Cybernetics and Informatics (SCI 2001), and Information Systems Analysis and Synthesis (ISAS 2001), 2001

[NEG02] Lyee's Hypothetical World, Fumio Negoro, Lyee International Workshop (Lyee_W02), 2002

[NEHA01a] A Proposal for Intention Engineering, Fumio Negoro and Issam A. Hamid, International Conference on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet (SSGRR 2001), 2001

[NEHA01b] New Innovation on Software Implementation Methodology for 21st Century - What Software Science can Bring to Natural Language Processing, Issam A. Hamid and Fumio Negoro, World Multi-Confernce on Systematics, Cybernetics and Informatics (SCI 2001)and Information Systems Analysis and Synthesis (ISAS 2001)

[PFI02] The World of Lyee, Roberto Poli and Issam Hamid Fujita and Rashid Ijioui, Iwate Prefectural University, Japan, 2002, ISBN 4-901195-06-9

[ROL02] A User Centric View of Lyee Requirements, Collette Rolland, Lyee_W02, 2002

[TOM02] Toward new software development, methodologies, and techniques, Shigeaki Tomura, SCI2002, 2002.