

Empirical Study for Evaluating Evolvability Requirements

Christa Schwanninger, Iris Groher, Regine Meunier, Uwe Hohenstein

Siemens AG, CT SE 2, Otto-Hahn-Ring 6,81739 Munich, Germany

{christa.schwanninger, regine.meunier, uwe.hohenstein} @siemens.com, Iris.Groher@students.jku.at

ABSTRACT

To convince the industry to use aspect-oriented programming techniques we have to defeat the prejudice against this paradigm. One mean to achieve this are studies proving the benefits of AO and disproving the retentions against it. We describe a small comparative study comparing one OO with three AO languages. The study results and the experience we gained help to plan and conduct further studies to show the value of AO.

1. INTRODUCTION

1.1 Evolvability

Aspect-oriented (AO) languages improve the modularity of systems. The claim that increased modularity improves understandability and thus the evolvability of systems is as old as modular and object-oriented (OO) programming. AO proponents see improved modularity as the main advantage of this paradigm [1, 2], critics argue that developers loose the overview when crosscutting concerns get modularized. A modularized concern still crosscuts the application. Understanding its effect when separated could be harder than if it was tangled with the code. The modules that are influenced by the crosscutting concern miss part of their context, the aspect contains pointcut declarations that are brittle and often hard to understand and handle without tool support. If the first position is wishful thinking or the second is a prejudice needs to be proven first before many practitioners in industry are willing to adopt AOP. We suggest giving evidence to practitioners that AO really improves the evolvability of software. One means to do so is to conduct empirical studies that prove that AO programs are easier to understand, easier to extend and thus easier to evolve than procedural or OO programs.

1.2 Background

Our industrial research group evaluates AO languages, tools and methods (among other paradigms) to find out, which AO technologies are helpful and mature enough to be used in industry. We recommend mature concepts to internal partners in product development, prepare teaching material and provide support.

To get evidence on the maturity and applicability of different AO languages in comparison to OO, we implemented a demonstrator that simulates a couple of use cases of an industrial application (size is 57 classes in Java) in three different AO languages, AspectJ [3], HyperJ [4] and CaesarJ [5], and in Java using OO design patterns [6]. We compared the implementations according to

- Quantifiable criteria such as size, coupling, cohesion, complexity, inheritance hierarchy depths, and runtime
- Qualitative criteria such as understandability and extensibility

This paper presents the results of a small comparative study falling into the second category.

We hope to get feedback from the workshop participants, hear about similar experiences and trigger more studies to

- Tackle the prejudice against AOP, like programs are less easy to understand, to debug and to extend.
- Get material to convince management and development teams to use AO languages when appropriate.

Our main intention for this first comparative study was to get a first sense on how different the language ratings are and also for how much preparation such a study requires. Our sample is small, but the resulting data allows creating hypotheses for further investigations. We asked Siemens employed diploma and PhD students and two interns to participate.

A second source for comparing the languages qualitatively is professionals. Considering the time constraints of professionals we plan a briefer version of our study, where we introduce all languages to a group of professionals and explain the designs for all our solutions. We will then ask them to rate how understandable, mature and applicable they find these languages for the development tasks they participated in. This will be the next step in our qualitative language comparison.

In the next section we explain the applications we wrote for our comparisons and how the study was prepared and conducted. Section 3 presents the study results, Section 4 gives a summary of the hypotheses we derived from the results and the lessons learned on conducting comparative studies. In 5 we briefly state our further plans and 6 talks about related work.

2. STUDY DESCRIPTION

2.1 Application

We implemented a central and important part of a telecommunication network operation and maintenance application. The kernel entity of such an application is a topological tree (TopoTree) that visualizes the state of hardware elements in a telecommunication network, reflecting the topology of the network.

Topological trees are common in various application domains, e.g. for managing high bay warehouses, power distribution systems or traffic control systems. The main difference between the application domains is the kind of monitored elements, which can be goods, hardware elements or vehicles. For our example the topological tree represents a network infrastructure to be controlled by telecom network operators. Controlling here means that hardware network entities have to be monitored to take action in case of a failure. Relevant data has to be presented to the operator in an easy to grasp way in a tree view. Color encoding is used to indicate the state of the network. Leaf nodes are hardware entities such as CPU boards, which are then aggregated to higher entities like racks containing the boards, and so on. The whole structure forms an acyclic tree. One server application manages the model of

the tree; it writes the recent state into a database and calculates state changes for tree elements. Clients have a GUI and present a view on the tree to operators.

In our implementation clients can change the state of nodes and push these state changes to the server, which is then responsible for calculating resulting state changes and updating all other clients. To make it easy for the operator to spot problems in the network, nodes are colored. The color code is: GREEN indicating normal operation, YELLOW for a warning, ORANGE for a major problem, RED for an error and WHITE for unknown state. These colors not only show the working state for each concrete leaf network element, the states of leaf nodes get aggregated and determine the color of the father node. This helps the operator to detect problems in a big network where most nodes representing higher level entities in the visualization tree are collapsed (e.g. node Device_5 in Figure1). Depending on the specific requirements, different state propagation strategies exist. If for example more than half of the children of a node are in a critical state, the father also shall show the coloring for critical state to make the operator aware of a major problem in a collapsed part of his visualization tree. Figure 1 shows a screenshot of such a client.

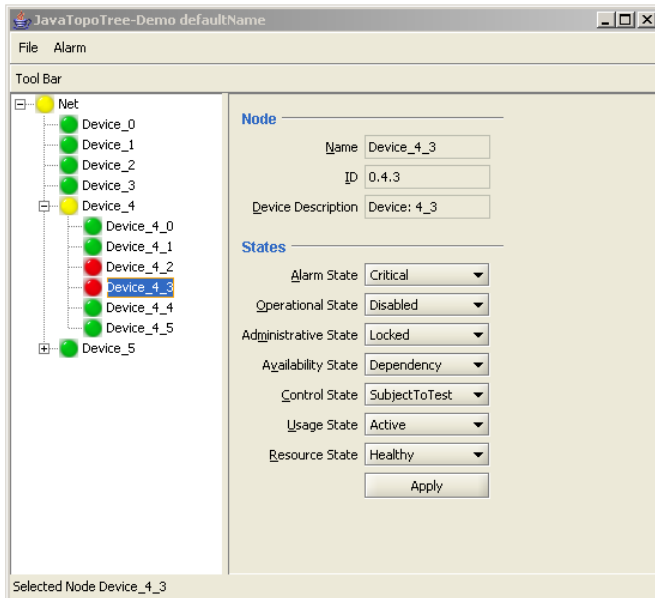


Figure 1: TopoTree Client

2.2 Implementation Variants

We implemented the same requirements in four different languages, three of them aspect-oriented extensions to Java – AspectJ, Hyper/J and CaesarJ – and one in pure Java. The purpose of this is to have a basis for comparing the AO languages with pure OO regarding quality criteria like efficiency – runtime and code size – and development requirements like understandability and extensibility.

For all versions together we upfront decided on five concerns that were implemented with either the aspect modularization mechanism of the respective AO language or with design patterns in the OO case. We implemented these concerns such that they are easily exchangeable in each application, for four of the five concerns even several implementation variants exist. Beyond these concerns the implementers of the four versions were free to choose

other concerns to modularize in aspects as they see fit best in the specific language. For the five predefined concerns we were interested in how good these concerns can be encapsulated in each language without corrupting the understandability of the whole application. We found out that all four languages were well suited for encapsulating these concerns. But as we stated in the beginning encapsulation is not a goal in itself but should serve the goals of understandability and extensibility. For this purpose we conducted an extensibility study to help us judge how well the different languages support understandability, extensibility and thus evolution. We did not yet check for properties like testability and independent reusability of modules or aspects, which are also very important for evolvability.

2.3 Concerns

The concerns we chose up front were:

- **Persistency:** The TopoTree application supports three different variants for retrieving the data: 1. reading the whole tree from the database as soon as the first client requests a node (eager persistence), 2. reading only the requested nodes, further nodes are read when a node gets expanded for the first time by a client (lazy persistence) or 3. not reading the database but generating the nodes only in main memory (no persistence).
- **Propagation:** Whenever an alarm state is raised on a node, a propagation strategy determines how its ancestors are affected. Two strategies are implemented in each version of the TopoTree.
- **Remoting:** The TopoTree either runs as a stand-alone application or in a client/server mode, where the server administers the model and arbitrary many clients show views on the model and feed back changes to the operator. For the second variant nodes should be remote objects.
- **Client-Update:** The client update concern cares for updating all client views when a change in the model occurs. This concern can be switched on or off. When it is missing, the application does not fulfill all its requirements, since operators should always see the current state of the network.
- **Tracing:** Different tracing levels should be possible.

2.4 Concern Implementation

Table 1 shows how the different concerns were implemented in the 4 versions of the TopoTree.

2.5 Study Set-Up

We chose the Client Update concern to be implemented by our test persons. We omitted the aspect in the AO examples, for the OO version we threw out updating code. For each of the AO languages two students implemented the concern independently, for the Java version we had three students, two of them did not know Java before. We were interested in how hard it is to learn an AO language in comparison to learning the first OO language.

	Persistence	Propagation	Remoting	Client Update	Tracing
Java	Strategy pattern; strategy set in a property file and implementing class instantiated at startup time	Visitor pattern; propagation strategy set in a property file and implementing class instantiated at startup time	Proxy pattern; when no remoting necessary direct communication of view with model, otherwise through proxy object	Observer pattern, not unpluggable	Tracing class with choice of tracing level at runtime
AspectJ	Abstract aspect for common retrieving/storing functionality and specific aspects for each persistence strategy	One abstract aspect with common functionality and one concrete aspect for each propagation strategy	Aspect for exception softening, for remote reference creation and retrieval	Observer pattern with inter type declarations adding functionality to the base classes	Several tracing aspects
HyperJ	One hyperslice per persistence strategy	one hyperslice per propagation strategy	One hyperslice for remote reference creation/retrieval and BCEL adding of Remote interface and RemoteException	One hyperslice for implementing the Observer functionality	Several tracing aspect hyperslices
CaesarJ	On cclass (aspect)implementation for each concern and one concern selection cclass	One abstract aspect for propagation and concrete aspects for the different implementations.	Making use of CaesarJ's remoting feature	One tree update aspect conforming to the Observer pattern	Several tracing aspects

Table 1: Concern Implementation

Three of the students are master students in computer science; they never participated in product development, the programs they write for their diploma theses are their first reasonably sized projects ever. Four students are PhD students in computer science; two of them have considerable experience with projects of about 80 to 100 classes each. The third one did mainly database programming and had nearly no experience with object-oriented languages and never used Java before. The last two are interns, one of them studying computer science and having a lot of development experience in Java and one bio-engineer that had only developed software in assembler and C so far. They all had heard about AOP, but none of them had investigated the paradigm more closely or ever used an AO language. To be able to connect the student education with his/her feedback, we give them names assembled from the language they worked in and a number. So Java1 is the PhD student who mainly developed database applications and has nearly no OO experience, Java2 is a diploma student who has OO and Java experience, Java3 is an intern who never used OO before. AspectJ1 is a PhD student with considerable Java knowledge, AspectJ2 a diploma student with some experience, comparable to student HyperJ1, HyperJ2 is a PhD student with some Java experience again. CaesarJ1 is an intern with a lot of programming experience in research projects and CaesarJ2 a PhD student with similar experience. To make participating in the study affordable for the students we

- taught mainly the language features that were necessary to implement the extension
- gave quite concrete hints where the application should be extended; thus the students did not have to understand all parts of the application thoroughly before being able to make the extension. (Future studies require more emphasis on the understandability of bigger portions of code.)
- supported the students if they had non-AO specific problems. We e.g. readily helped them solve RMI related problems asking them not to add the time for this to the final effort.

The students all together got a 2.5 hour's introduction to the application and to AO in general with a brief summary of the three aspect languages. Every students group that implemented in an AO language then got a one to two hour's crash course in their AO language concentrating on the general concept and on the specific

concepts they needed for their solution. For all groups including the three Java extension developers we walked through the relevant parts of their specific application and gave rather concrete hints what was missing where.

Every student got documentation on their language, which was

- popular Java books for the Java developers
- AspectJ in Action [6] and the AspectJ Quick Reference for the AspectJ developers
- The Hyper/J User and Installation Manual [4] for the Hyper/J developers
- Tutorial slides from the CaesarJ language development team [5] for the CaesarJ developers

During implementation, we immediately helped when we saw that the students had problems with non-AO related tasks or when we thought we didn't introduce a language feature well enough.

All students worked with Eclipse 3.1.0. The AspectJ version was 1.5.0 for compiler and runtime and 1.2.0 for AJDT, the Hyper/J version was the latest downloadable, and the CaesarJ version 0.5.3.

Our questions fell into four categories. The first one was on the effort for understanding and implementing the extension, the second on language understandability, then on development and documentation support and the last block on how the student judges the usefulness and applicability of the language for development projects.

3. STUDY RESULTS

3.1 Effort Comparison

In general all students managed to solve the implementation task in nearly a day including understanding the application, understanding the language basics, designing, implementing and testing the solution. Figure 2 shows the effort the students spent on understanding the base application, designing their specific AO or non-AO solution, implementing and testing it. The number of participants is too low to draw strong conclusions, but we can derive some trends. These hypotheses need to be proven or invalidated by studies done with a bigger sample.

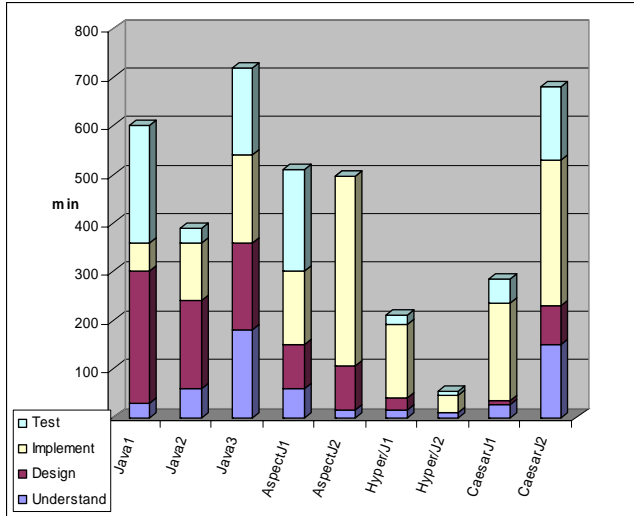


Figure 2: Development Effort

In general the students Java1 and Java3 who were new to Java and had to implement the Java extension had more problems dealing with the language than students who knew Java but were new to AO and had to implement one of the AO parts. Java3 never had used an OO language before; she was the one who had most difficulties in understanding the application and implementing the extension. The results for AspectJ and CaesarJ are similar, although in advance we were rather concerned that CaesarJ would be too different from the way how average OO developers think to get the task done without getting a full CaesarJ tutorial beforehand. However the students who volunteered to do the CaesarJ extension were the ones with the most programming experience; they already were familiar with generative programming and model driven development concepts and had more development experience than the other students.

We were surprised that the experienced Java developers who had to extend the HyperJ version were faster than the AspectJ developers. They felt that understanding the composition language was quite easy and developing a hyperslice requires mainly just Java knowledge. On the other hand they also said that it would have been really hard for them to identify all hyperslices in the first place, while the AspectJ developers found the concerns that were implemented as aspects quite obvious and would also have identified those as aspects.

We also found out that having the test persons measure the time for certain tasks themselves can temper the results. While some include every five minutes they thought about the problem during the coffee break, others tend to count only the time they fully concentrated on the solution.

3.2 Language Evaluation

In the next block of questions we wanted to know how easy to understand the students considered the languages and specific application. Table 2 “Language Evaluation Results” presents the results. 1 is the best grade and 6 the worst.

In average it seems to be harder for somebody who never used an OO language to understand Java than for a Java programmer to switch to an AO language based on Java.

For CaesarJ the documentation is not very rich, yet, this was one reason for the CaesarJ1 candidate to rate the language concept understandability very low. In general language concept understandability is better for AspectJ and Hyper/J, however the two CaesarJ ratings for how well the problem solving is supported are even slightly better than the AspectJ ratings. We had guessed AspectJ would be easiest to understand beforehand while we considered CaesarJ as the most different from the OO paradigm and thus hardest to understand. But this notion was not confirmed by the study results.

The fact that CaesarJ was considered more mature than AspectJ was also not what we expected. This definitely needs more validation in a study where each participant gets to know and rate all three AO languages.

3.3 Tool Support and Debugging

The next cluster of questions deals with the IDE and debugging support, see Table 3 “Tool Support and Debugging”. Again ratings were from 1 (very good), to 6 (not provided or very bad). Since all students used Eclipse as IDE with the plugins provided by the AO languages, there was a very good Java support and reasonably good AspectJ support, but no real Hyper/J and CaesarJ support, when it comes to language parts that are not Java.

For Hyper/J and CaesarJ (in the version we used) the IDE support beyond Java support is missing. This is also reflected by the answers to the questions concerning tool support. Debugging support is integrated for Java and AspectJ, some of the students rather used console printouts. Most of the students on the other hand really missed better browsing support especially for Hyper/J and CaesarJ. They said that for a bigger example they would need better visualization of the relationships between aspects and base code. For getting a good evaluation of the IDE integration the extension task obviously was too small. A study with a bigger extension task and a group of professional software developers would definitely lead to stronger results.

3.4 General Valuation

Finally, we wanted to get more general statements on whether the students can think of families of tasks or challenges they would consider the tested languages especially useful for, whether they would like to use and recommend the languages for their daily work and if they think the languages were mature. Table 4 “General Valuation” summarizes the results.

The whole Java group thinks that the language scales for industry projects, which is no surprise since everybody knows it does. From the lack of examples that could easier be solved with one of the languages we conclude that the students did not have enough experience to see a paradigm suitable for solving specific problems or that the time they spent with the language was too short to see its value. Again, a study with professionals has to be done to get better data on these questions.

One interesting fact is that one AspectJ implementer first solved his task with OO and then pulled the code out into an aspect. This developer also had major concerns that AspectJ could result in a bad design or easily be misused for patching a system. He therefore would not consider it for industrial use.

AspectJ and Hyper/J results are rather similar in the evaluation. CaesarJ is considered to be more a research language and needs a lot more tool support to be applied in real development projects.

4. SUMMARY AND LESSONS LEARNED

We conducted a small comparative study to evaluate how easy it is to extend an application written in 3 AO languages and one OO language. The study sample was small but sufficient to derive some hypotheses that have to be proven in studies with bigger samples. We encourage the AO community to conduct such studies to give evidence to the industry that AO is mature enough to be adopted and that quantifiable and qualitative attributes improve when applying AO technologies.

Some of our hypotheses that need further investigation are

- It is easier for an experienced OO developer to switch to an AO language that extends an OO language than it is for a developer who is only familiar with procedural programming to switch to an OO language.
- New language constructs are harder to learn than an external connector file; in this case AspectJ and CaesarJ were slightly more challenging than Hyper/J.
- The better a language supports modularization the more IDE support is needed to understand the result as a whole.

This was our first comparison study on this topic. The lessons we learned for further studies are:

- To compensate the variation in education and experience of test persons it is necessary to have the same test persons implement all 4 extensions if the sample is small or have a by far bigger sample (e.g. 25 per group) to get statistically valid data.
- To get comparable time measurements it is better to have a facilitator who watches the test persons and notes the time.
- For a more realistic assessment of how a language supports evolution several extension tasks with varying effect on the architecture should be performed for one application.
- Questions concerning IDE support need bigger examples and professional software developers as test persons.
- Questions falling into our general validation block need test persons with more experience than we had.

5. FUTURE WORK

In the near future we want to do a study with professionals whom we present all four solutions and ask for their evaluation of the languages. We also need measures on testability independent reusability.

6. RELATED WORK

There exists a number of studies that compare AO languages [8,9], AO with OO [10,11,12] or evaluate the suitability of AO for certain programming domains [13,14,15]. Due to space limitations we can't discuss them in this position paper.

7. ACKNOWLEDGMENTS

We thank all students who participated in our study.

8. REFERENCES

- [1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.M. Loingtier, J. Irwin, "Aspect-oriented Programming", In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), LNCS 1241, Springer-Verlag, 1997
- [2] P.L. Tarr, H. Ossher, W.H. Harrison, S.M. Sutton Jr., "Multi-Dimensional Separation of Concerns in Hyperspace", In Proceedings of the International Conference on Software Engineering (ICSE), pages 107-119, 1999.
- [3] AspectJ, Aspect-oriented Programming in Java, <http://www.aspectj.org>.
- [4] Hyper/J web site: www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm
- [5] CaesarJ Website, <http://www.caesarj.org>
- [6] Ramnivas Laddad. AspectJ in Action: Practical Aspect-Oriented Programming. Manning, 2003
- [7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
- [8] Christina Chavez, Alessandro Garcia, and Carlos Lucena. Some insights on the use of AspectJ and Hyper/J. In Rashid [1181].
- [9] Mik Kirsten. Aop@work: Aop tools comparison, part 1: Language mechanisms. Technical report, IBM Developer Works, February 2005.
- [10] Robert J.Walker, Elisa L.A. Baniassad and Gail C. Murphy, An Initial Assessment of Aspect-oriented Programming, In Proceedings of the 21st International Conference on Software Engineering, 1999
- [11] Alessandro Garcia, Cl'audio Sant'Anna, Eduardo Figueiredo, Uir'a Kulesza, Carlos Lucena, and Arndt von Staa. Modularizing design patterns with aspects: A quantitative study. In Tarr [1361], pages 3-14.
- [12] Shiu Lun Tsang, Siobhán Clarke, Elisa Baniassad, Object Metrics for Aspect Systems: Limiting Empirical Inference Based on Modularity, Trinity College Dublin technical report
- [13] Kersten, A. and Murphy, G. Atlas: A Case Study in Building a Webbased Learning Environment Using Aspect-Oriented programming. Proceedings of OOPSLA'99, November 1999.
- [14] Lippert, M. and Lopes, C.V. A Study on Exception Detection and Handling Using Aspect-Oriented Programming in Proc. ICSE 2000. Limerick Ireland. 2000.
- [15] Shiu Lun Tsang, Siobhan Clarke, Elisa L. A. Baniassad, An Evaluation of Aspect-Oriented Programming for Java-based Real-Time Systems Development, ISORC 2004

	How easy was it to understand the language concept? (1=easy, 6=difficult)	How good was the documentation? (1=very helpful, 6=no help)	How hard was it to solve a problem in this paradigm? (1=very easy, 6=very hard)	Do you think this language is mature? (1=fully developed, 6=immature)
Java1	4	3	4	1
Java2	1	2	2	2
Java3	4	2	4	1
AspectJ1	2	2	4	3
AspectJ2	2	4	5	4
Hyper/J1	2	3	3	3
Hyper/J2	2	6	2	4
CaesarJ1	6	6	3	2
CaesarJ2	3	4	2	3

Table 2: Language Evaluation Results

	IDE integration	editing and browsing support	missing editing and browsing support	debugging integration
Java1	2	2	browsing support, which class calls a method	did not use
Java2	2	1	missed none	3
Java3	2	2	yes	3
AspectJ1	3 (buggy)	2	Refactoring (finish pure Java and refactor into aspects); see woven code	not tested
AspectJ2	2	2	code completion, context help (search for references)	didn't use
Hyper/J1	6	2	no, am happy with command line tool, too	didn't use
Hyper/J2	5	2 for Java, 6 for Hyper/J	support to write HyperJ files; management of slices and relationships	for java yes, for HyperJ no
CaesarJ1	4	5	code completion, browsing support, context help, refactoring support	not integrated
CaesarJ2	4	3	context sensitive help, code completion, optimized views (e.g. layered architecture view, Cl/Implementation/Binding hierarchy/relationship view)	no support

Table 3: Tool Support and Debugging

	challenges/tasks to be solved more easily	want to use and recommend	scalability for industry
Java1	for developing protocol interfaces	yes	yes
Java2	no	1	1
Java3	no	don't know	can't judge
AspectJ1	could result in bad design, patching	3 (depends on relationship between effort to learn / number of potential aspects in project)	5 (results in bad design?)
AspectJ2	no	4; nice idea, but I prefer to have whole code together	2; good extensibility for existing code (extra features ..), different versions of features
Hyper/J1	for introducing new parts without knowing all existing methods	3	2
Hyper/J2	yes, e.g. logging	5	5
CaesarJ1	No	5	6
CaesarJ2	yes, use cases with layered architecture and AO requirements	5; not suitable for the industry in the current state, 2 in research area	4 not in current state (challenging developer skills and critical tool support)

Table 4: General Valuation