An Approach with Maintainability, Structured Design and Automation with the Intension of Software Engineering

N.BASKAR
Research Scholar in Computer Science
SRMV College of Arts and Science, Coimbatore – 20,Tamilnadu India.
Email: n_bas@rediffmail.com

A.V.RAMANI
Department of Computer Science
SRMV College of Arts and Science, Coimbatore – 20,Tamilnadu India.
Email: avvramani@yahoo.com

Abstract

Scientific software must be personalized for dissimilar execution environments, problem sets, and existing resources to make sure its competence and consistency. Even though adaptation patterns be able to be found in a extensive percentage of new scientific applications, the conventional scientific software stack be deficient in the sufficient adaptation concept and tools. As a consequence, scientific programmers physically implement ad-hoc solutions that are tough to retain and reuse. In this paper, illustrate the experimental evaluation of the level of inheritance in five object-oriented systems. The systems considered differ in together its size and application domain. A consequence from our analysis mutually with added new investigations appears to support the thesis that inheritance is used moreover sparingly or incorrectly. Statistical relationship among four inheritance metrics and a position of reliant variables (non-comment source lines, software understandability, known errors and error density) present evidence for this claim. It is also not clear that systems with the use of inheritance will essentially be more maintainable than those that do not. The data examined from two of our systems recommends that deeper inheritance trees are characteristics of systems which are harder to understand and (by implication) harder to preserve. By examining, why? This may be the case, and suggest ways of remedying this state of affairs.

Keywords: Inheritance, SPARQL Query Language, Aspect JAVA

I. INTRODUCTION

The implementation model in the greater part of computing domains has been constantly appropriating more active. In a dynamic execution model, the precise execution steps are known merely at runtime, as determine by means of input parameters and resource allocation. A considerable portion of enterprise software, for instance, is written in manage languages such as Java and C#. These languages not only a dispatch method strongly to support polymorphism, excluding also deeply rely on dynamic class loading and Just-in-Time compilation. The default implementation semantics is often modified by means of Aspect-Oriented Programming [1], a programming model that presents concepts and tools to systematically enhance or even entirely redefine the semantics of method invocations and object construction.

The AOP functionality can also be implemented vigorously to adapt an application's semantics supported on some runtime conditions. Numerous researchers have newly recognized dynamic adaptation as proficient of gaining scientific software [2, 3]. Individual skills of collaborating by means of scientific programmers authenticate that the conventional execution model of scientific applications build, run, change, and run an added no longer gives the required flexibility necessary to contain the advanced requirements of modern scientific applications. Such applications function over ever-expanding data sets and need important algorithmic sophistication to accomplish the needed performance levels.

The traditional scientific software stack is tailored toward static execution by unfortunate. A significant portion of scientific applications are still written in Fortran, which in spite of all of its newest extensions at rest remains a glorified formula translator presenting few conveniences to support any execution dynamicity. The execution path of a distinctive scientific application is encoded at compile time and rarely changes in response to any runtime events. To the most excellent of our knowledge, no conventional AOP extension has ever been developed for FORTRAN.

A key feature of the object-oriented (OO) paradigm is with the intention of inheritance [3]. By make use of inheritance is claimed to reduce the amount of software maintenance crucial, simplify the burden of testing [4], and generate more consistent, high quality software [5]. This examine five OO systems, all written in Java, and mutually with two further inheritance-based design metrics developed as part of the MOOPS project [6][7]. The outcome of our analysis (and that of other investigations) looks to support the thesis that inheritance is moreover used cautiously in the development process, or is used mistakenly [8] [2]. By making use of inheritance does not looks to be delivering the benefits it agreed, and it is not clear that systems using inheritance are easier to maintain than those that do not. In this paper, an analysis of the faults is found in three of the five systems investigated showed very little relationship with any of the inheritance-based metrics. A collection of Inheritance-based metrics for all five systems also show to be a lacking in of relationship with three of the dependent variables collected for the five systems (i.e., the number of non-comment source lines, the number of known errors and error density). Though, an attractive relationship was established between the depth of the inheritance tree and software understandability. It describes the five application domains studied and the metrics together for each also that it presents data for each one of the five systems. Relationship and data analyses of each system, integrating analyses of data errors in three of the systems, inferences of our consequences are then described. At last, some conclusions and suggestions for further work are obtainable.

Footed on ongoing association with scientific programmers, have observed a tendency in which esoteric solutions to dynamic adaptation are skilled for individual applications, most important adaptations that are neither maintainable nor reusable. Even though such solutions are returning their non systematic implementation practices gain an important software maintenance burden. For that reason, there is great potential advantage in implementing such dynamic adaption patterns more logically and taking advantage of the position of the art tools and techniques created for that purpose.

Supported on these results, work creates the following contributions:

- An approach to renew scientific applications: Adapting scientific applications presents efficiency, stability, or increased accuracy advantages. An approach grants a systematic method to adapt scientific programs written in FORTRAN, allowing them to gain from the mentioned adaptation advantages.
- An approach to use again the adaptation code: By stating recurring adaptation patterns as abstract features that can be extended, An approach offers a reusable and customizable library of adaptations that can be used by different scientific applications.
- Democratizing the scripting of scientific adaptation functionality: With the exposure of the adaptation functionality as standard AOP code, an approach increases the population of programmers who can write and uphold such code. While adapting scientific applications at rest needs proficient in the scientific domain at hand, implementing the functionality no longer have need of intimate knowledge of the intricacies of Fortran. This is for the reason that adaptation functionality is commenced through AOP.

II. System Architectures

The inheritance hierarchies leaned to be moderately shallow (with a mean depth of inheritance tree of 0.93). The inheritance structure, on the other hand, enclosed a large number of small inheritance trees (the mean depth of inheritance trees was 0.63). As a result, would anticipate more coupling owing to inheritance to be found in System. In cooperation, System widely use was made of friend functions thus avoiding the use of inheritance by means of invisible coupling. Fascinatingly, in all systems, there were high quantities of singleton classes (i.e., classes without any inter-class coupling. In each of the systems, substantiation was found of several base classes including a large number of methods from which a large number of classes inherited. All systems examining showed some substantiation of non-inheritance based coupling.

2.1. Data analysis

Connections were performed for each of the systems in opposition to SU and NCSL. The SU and NCSL connection for System 1. Only metrics for which a relationship was recognized. The only remarkable feature is the positive considerable relationship between DIT and SU. This may point towards that as the depth of the inheritance tree increases, the SU score increases, reflecting a decrease in understandability. This would look to support the claim that increasing difficulty at deeper levels in the inheritance hierarchy sources more errors to be invested at those levels.

2.2. Toward a software maintenance methodology

A software maintenance methodology is expected, such that a developer or team of developers would physically input information regarding requirements into an IDE, along with the entry of code. Functionality in the IDE (perhaps implemented as a plug-in) would store software system metadata in an RDF (Resource Description Framework) graph and accept SPARQL queries (Query Language), the outcome of which may possibly be used to change user interface elements. Changes in the metadata, together with when changes were made to software components (inheritance), requirements, metrics and tests, would be pathway. Just as existing IDEs allow for the automated running of tests and calculation of metrics, a background method would maintain the metadata up to date. SPARQL queries would be run to monitor the results of changes to the metadata.

Developers would be informed by the use of the IDE's user interface when changes required action. Examples would consist of:

1) Change to a software component of inheritance would need a revalidation of any related requirements. The names of invalidated requirements would be displayed to the user until revalidated.
2) Failing tests would relate to requirements, as well as to software components in the course of the metadata. Requirements impacted by failing tests might be displayed along with test results.
3) Metrics calculated to be out of preferred extent could be treated as the same to failed tests. The names of software components and requirements connecting to such metrics could be displayed until the metrics are calculated to be within desired scope.

Software system metadata be supposed to be kept in a state adequate for the above notifications to be made. Each requirement, metric and test should have associated software components. Metrics and tests must be illustrated in metadata adequately to be run automatically. In absent of metadata should be carried to the attention of the user so that it might be added.

It could eagerly be extended by prototyping the concepts in an existing IDE with a plug-in architecture and conducting usability studies. The collision of this work on the upholding costs of both new projects and existing projects desires to be studied to measure its effectiveness.

An added SPARQL queries could be developed to demonstrate functional most important indicators so that a software project may be directed toward maintenance activities that will lengthen its life cycle. For example, metrics calculating the rarely-implemented Maintainability Index [9,10,11] could be used to suggest when a particular software component should be considered for refactoring and which requirements that would impact.

This methodology is measured to be dependent solely upon an RDF (Resource Description Framework) data store with a SPARQL query engine. That is, an OWL-DL reasoner is not essential. OWL reasoners may possibly be used to ensure the logical consistency of ontologies, establish the logical formalisms requisite to express ontologies, and assume new statements from existing statements. Scalability concerns go ahead us to evade runtime reliance on a reasoner, even though one was used all through the development of the ontology. Reasoners do not presently scale for the reason that they must recompute all suggestions following a change to the fundamental data. The study of large software projects would outcomes in large RDF graphs, thus reasoning difficulty upholding the position of inferred statements. [12, 13, 14]The scalability of reasoners is at rest being studied. Our SEC ontology was consequently designed (e.g. by liberal use of inverse properties) to evade runtime dependence on a reasoner.

It would be reasonably helpful to logically deduce relationships between software components and their metadata for the intentions of reducing query difficulty. For example, an object-oriented class can include methods which it inherits from a super class. A basic SPARQL query might miss the methods in the super class if a reasoner were not used. Additional study of the best use of reasoners is enviable.

The existence of software system metadata is a suggestion of how entirely this methodology is being used. Lack of metadata, or of a certain type of metadata, would point toward a condition which must be brought to a user's consideration.

Some means of authenticating obtainable metadata ought to be developed. Constraints on the application of ontological elements previously point toward some areas where the SEC ontology constrains application. For instance, a test ought to be implemented by a software component and a metric must be associated with a software component. Further refinement of these constraints and subclassing within the ontology will be necessary to permit a greater degree of automated validation of proper usage. This is particularly true of metrics. A particular metric can be applied at the program, package, and class or method level. Making of a new ontological element in the ontology, which subclasses Metric and is leap to relate to a particular type of software component, would support automated validation.

2.3. Reusability

An enviable software design objective is code reusability, which permits using the same code fragments in multiple scenarios whether within the same application or across different applications. The capability to use the code again to improve programmer productivity, as the programmer does not contain to implement the same functionality multiple times. This, in turn, guides to a smaller code size, which condenses the maintenance burden and the risks of introducing software defects [15, 16, and 17].

In object-oriented programming, a significant technique to endorse code reusability is class inheritance. Common functionality is summarized in a base class that is extended by subclasses which include only the unique functionality. In AOP, features can inherit from each other. In adaptation implementation, uses inheritance to widen adaptivity schema aspects, in that way reducing the total size of the adaptive code.

The amount of Uncommented Lines of Source Code (ULOC) written by a programmer among the Aspect JAVA and hand-coded implementations. `aux' signifies auxiliary code that is not significant to an adaptation logic implementation (designated as `logic' in the table), such as header includes, helper functions, and linkage macros to determine name mangling between Fortran and JAVA. The hand-coded implementations also require using the framework's APIs to setup the introduction of adaptation code to GenIDLEST.

The Aspect JAVA versions acquire less code to implement among the hand-coded ones in all cases. The code reduction ranges between 15% for the most complex dynamic tuning adaptation and 40% for the simple time step control adaptation. [18]The adaptive functionality for the supplementary part can be implemented in fewer lines of code by with the use of Aspect JAVA. The hand-coded implementation also needs some hand-written code to appropriately instantiate the ACC framework.

In adding together, Aspect JAVA implementations exercise fewer lines of hand-written code by inheriting features of schema. As code turn into complex and its size grows, explains that the gain becomes smaller because the point cuts defined in all adaptation implementations specify a limited number of join points such as loop ends. Therefore, application-specific adaptation schemes that involve multiple join points can benefit more from subclassing schema aspects.

2.4. Limitations

Key limitations of our approach stem from the semantic differences between Fortran and JAVA, and to effectively adapt Fortran programs, we necessarily had to limit the subset of the Fortran language with which we want to be able to interface through Aspect JAVA.

A current engineering limitation of our tool infrastructure is a lack of support for composite types. A FORTRAN composite data type is a global structure that can be mapped to some JAVA global variable. As long as both the JAVA and FORTRAN parts of an application conform to the ELF (Executable and Linking Format) specification, a FORTRAN composite data type can be mapped to some JAVA structure to ensure that both structures have the same layout. Thus, to use composite types with our approach, the programmer has to define an appropriate JAVA counterpart for a FORTRAN complex data type, which can be tedious and error-prone if done manually. As a specific example, the FORTRAN complex data type does not have a native counterpart despite the presence of complex types in the JAVA Standard Template Library [19].

Our approach requires that only FORTRAN functions be exposed through interface with JAVA aspect code. The leverage Aspect JAVA execution point cut through which the programmer can specify callee-site join points at the execution of wrapped Fortran function calls. The resulting callee-site weaving is easier to implement than caller-site weaving (e.g., the call pointcut);only one exposure point is required for each intercepted Fortran function, whereas the caller-site weaving has to examine the entire Fortran codebase to find every call-site for each intercepted function that must be exposed at all the call-sites. As a result, our callee-site weaving approach may be insuffciently powerful in the case of complex adaptations requiring complete context information, which may be unavailable from the exposed signature-only Fortran function data.

While Aspect JAVA supports other kinds of point cuts, they would not be applicable for the needed adaptations. In addition, some of the Aspect JAVA point cuts simply cannot be used due to the fine-grained differences in semantics between FORTRAN and JAVA, which restrict the range of applicability of certain Aspect JAVA features. For instance, Aspect JAVA offers class and namespace matching mechanisms to specify join points with the granularity of JAVA classes or structures. However, it is not immediately obvious how one can apply them to Fortran, which does not have a direct counterpart to JAVA classes [20].

As it turns out, even using a limited subset of Aspect JAVA features makes it possible to flexibly adapt Fortran programs at the function level, whereby separately developed Fortran and JAVA programs work in concert to achieve a common goal of implementing adaptable scientific software.

III. CONCLUSION

In this paper, present an approach that expresses returning adaptation functionality patterns of scientific computing as reusable aspect-oriented code. Our approach uses cross-language adaptation implemented using code generation and an aspect library. The software engineering benefits of our approach by obtaining the ULOC and cyclomatic complexity metrics from the original (hand-coded) and our (aspect-based) versions of a computational fluid dynamics scientific application. The results of the evaluation show that using aspects can reduce the amount of code needed to implement the adaptivity functionality by as much as 27% on average. Our approach does not incur an unreasonable performance overhead.

Overall, the software engineering benefits of our approach include improved maintainability, more structured design, and greater automation. Greater reusability enabled by our approach also allows scientific programmers to subclass the schema aspects provided by our library, thereby reducing the programming effort. Future work directions will focus on providing a more complete library of adaptivity schema aspects to support additional adaptivity schemas [10].

Facing the unprecedented challenges of modern scientific applications requires the adoption of state-of-the-art software engineering techniques and approaches. In that light, the maintainability advantages offered by AOP can other viable solutions to these challenges. The ideas presented in this paper can inform the designs that transfer the lessons gleaned from constructing and maintaining mainstream software systems to help address the challenges of emerging scientific software.

There is clearly a need for research to address many urgent issues arising from the use of inheritance and its effect on the process of maintenance. Fundamental issues such as whether inheritance can make maintenance of OO systems easier, whether there is an optimum level of inheritance, and whether we should focus on alternative features as a means of reducing the maintenance burden all need be addressed. Ideally, this empirical research should be carried out on as many real-world systems as possible, (with subjects of varying experience), supported by well-designed hypotheses. Industrial-strength tools need to be provided to aid the speedy collection of data and dissemination of results.

REFERENCES

[1] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In Proceedings of the European Conference on Object-Oriented Programming, volume 1241, pages 220{242.Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[2] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. Whaley, and K. Yelick. Self Adapting Linear Algebra Algorithms and Software. Proceedings of the IEEE, 93(2):293-312, Feb. 2005.

[3] D. K. Kim, Y. Jiao, and E. Tilevich. Flexible and Efficient In-Vivo Enhancement for Grid Applications.In CCGRID '09: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, pages 444{451, Washington, DC, USA, 2009. IEEE Computer Society.

[4] F2PY: Fortran to Python interface generator.http://cens.ioc.ee/projects/f2py2e/.

[5] Tools Interface Standards (TIS) Committee.Executable and Linking Format (ELF) Specification,1995.

[6] J. Irwin, J.-M. Loingtier, J. R. Gilbert, G. Kiczales,J. Lamping, A. Mendhekar, and T. Shpeisman.Aspect-Oriented Programming of Sparse Matrix Code. In ISCOPE '97: Proceedings of the Scientific Computing in Object-Oriented Parallel Environments, pages 249{256, London, UK, 1997. Springer-Verlag.

[7] P. Kang, N. K. C. Selvarasu, N. Ramakrishnan, C. J.Ribbens, D. K. Tafti, and S. Varadarajan. Modular,Fine-Grained Adaptation of Parallel Programs. In ICCS '09: Proceedings of the 9th InternationalConference on Computational Science, pages 269-279, Springer, May 2009.

[8] T. Aslam, J. Doherty, A. Dubrau, and L. Hendren.AspectMatlab: An Aspect-Oriented Scientific Programming Language. In AOSD '10: Proceedings of the 9th International Conference on Aspect-Oriented Software Development, pages 181{192, New York, NY,USA, 2010. ACM.

[9] VanDoren, E.: Maintenance of Operational Systems An Overview, Carnegie Mellon Software Engineering Institute,http://www.sei.cmu.edu/str/descriptions/mos_body.html (1997)

[10] S. Varadarajan and N. Ramakrishnan. Novel Runtime Systems Support for Adaptive Compositional Modeling in PSEs. Future Gener. Comput. Syst.,21(6):878{895, 2005.

[11] P. D. Hovland and M. T. Heath. Adaptive SOR: ACase Study in AutomaticDierentiation of Algorithm Parameters. Technical Report ANL/MCS-P673-0797, Mathematics and Computer Science Division,
Argonne National Laboratory, 1997.

[12] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers.
Extensibility, safety and performance in the SPIN operating system. In Proc. 15th ACM Symp. on Operating System Principles (SOSP), pages
267–284, December 1995.

[13] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: a platform for secure distributed computation
and storage. In Proc. 22nd ACM Symp. on Operating System Principles (SOSP), pages 321–334, 2009.

[14] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. Whaley, and K. Yelick. Self-Adapting Linear Algebra Algorithms and Software. Proceedings of the IEEE, 93(2):293{312, Feb. 2005.

[15]] P. Kang, N. K. C. Selvarasu, N. Ramakrishnan, C. J.Ribbens, D. K. Tafti, and S. Varadarajan. Modular, Fine-Grained Adaptation of Parallel Programs. In Proceedings of the 9th International Conference on Computational Science, pages 269{279, Baton Rouge, Louisiana, USA, May 2009.

[16] D. S. Myers and A. L. Bazinet. Intercepting Arbitrary Functions on Windows, UNIX, and Macintosh OS X Platforms. Technical Report CS-TR-4585, UMIACS-TR-2004-28, Center for Bioinformatics and Computational Biology, Institute for Advanced Computer Studies, University of Maryland, 2004.

[17] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In Proc. 26th ACM Symp. on Principles of Programming Languages (POPL), pages 228–241, January 1999.

[18] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. ACM Transactions on Software Engineering
and Methodology, 9(4):410–442, October 2000.

[19] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler Transformations for High-Performance Computing. ACM Comput. Surv., 26(4):345{420, 1994.
[20] K. Radhakrishnan and A. C. Hindmarsh. Description and Use of LSODE, the Livermore Solver for OrdinaryDi erential Equations. Technical Report UCRL-ID-113855, LLNL, 1993.