Software Information Leaks: A Complexity Perspective

Boby George and Shawn A. Bohner
Virginia Tech
boby@vt.edu, sbohner@vt.edu

Ruben Prieto-Diaz
James Madison University
prietodiaz@cisat.jmu.edu

Abstract

Software development can be thought of as the evolution of abstract requirements into a concrete software system. The evolution, achieved through a successive series of transformations, is inherently a complex process. The inherent complexities, that often make these transformations sub-optimal, are further aggravated by inefficient capture and usage of requisite information during transformation. While some understanding of software may be reasonably clear at a given time, the future dependencies may not be fully understood or accessible. The clarifications obtained over time make the system more concretely understood, but there may be Software Information Leaks (SILs) as some relevant information is lost. Some key SILs may be due to failure to be fully acquainted with dependencies between various software artifacts. In this paper, our objective is to define SILs as a concept, codify an essential set of canonical leaks, and introduce solutions for dealing with some of them.

1. Introduction

Software systems face an uncertain development environment as stated by the uncertainty principle - "Uncertainty is inherent and inevitable in software development process and products" [1]. From the first requirement statement itself, there is an element of uncertainty. This might be due to the abstractness of software system requirements during project initiation stage. The abstractness may be due to the property of requirements that "they specify what a system is to do without detailing how it does it" [2]. The requirements, which describe only the external behavior of a system [2], evolve to become clearer during requirement elicitation phase.

Software analysts and customers interact during requirement elicitation phase. Such interactions enable the analyst to understand systems' expected external behavior and the customer to appreciate the implementation approaches. It should be noted that neither party fully understands the ultimate system at this stage. However, during this learning process, decisions are being made; certain interpretations are formed and dependencies (explicit and implicit) are created. Yet, these often elude proper recording. In particular, there are pieces of dependent information that are necessarily separated or even ignored to simplify the thinking about the problem (due to a legitimate principle called, "separation of concerns"). Further, a software engineer's ability to understand the full corpus of information and retain all of the dependencies is limited both by natural capacities of the human mind as well as the temporal nature of acquired knowledge. Hence, unknown to the players, software information (information related to software system) has already started to leak.

It is important to recognize that the progression from abstract requirements to more concrete code representations is essentially a language translation process. There are many design alternatives that will satisfy a set of requirements, just as a sentence can be translated in many ways. The one-to-many relations between requirement-to-design translation result in misconceptions. For example, someone who has in mind a business value calculation with several calibrating points specifies a software requirement. When the software engineer reads the requirement, s/he envisions a single calculation like what might be seen in a business book. The requirement originator has a holistic perspective, while the software engineer has a simplistic view. The result is information missing – a leak. While better requirements engineering techniques can certainly solve the above misconception, we must ask how dependent is the software engineer on the information, and if the requirement originator had, at her/his disposal, software dependency connectors, could this situation be made better?

As the project proceeds, the leaks get compounded, as there are many places where information is uncertain or not captured. There have been techniques to address this problem, like Structured Analysis and

Proceedings of the Ninth IEEE International Conference on Engineering Complex Computer Systems Navigating Complexity in the e-Engineering Age
1050-4729/04 $20.00 © 2004 **IEEE**

Design Technique (SADT) [3] and traceability based software documentation [4]. Most have attempted to impose some semantic structure on the solution domain through an intermediate form like a design model or language. However, with the information not yet complete or stable, the formalism introduced tends to unnecessarily burden the effort and most software engineers are inclined to omit it. Further, the use of modeling notations earlier on in the process, without due consideration on understandability among all stakeholders, results in information restriction [2]. Later in the development and maintenance activities, two events occurs: (1) the uncertain or omitted information becomes a problem as the software is changed without the requisite insights into prior dependencies, and (2) the new knowledge and design decisions taken at later stages are not recorded and factored into earlier artifacts.

This paper introduces and examines the problem of SILs. We attempt to categorize key SILs and propose a dependency-based model for dealing with them. SILs occurring at the transition from requirements to design appear to be the most severe leaks, for the translation of requirements into design is a significantly larger step than the one from design to code. This is because requirements address the external behavior of the system, while design addresses the internal behavior that would enable the external behavior. The transition from design to code is smoother as the code is essentially an implementation of the design.

1.1. Problem Definition

Over the years, researchers have tried to identify and define SILs and their occurrence [5]. Software architecture researchers argue that over a period of time, the techniques and notations developed for software design have been integrated into the implementation process [6]. Such integration has tended to blur, if not confuse, the distinction between design and implementation [6]. This blurring may result in information lost or SILs. Further, proponents of agile methodologies such as Kent Beck [7] argue that heavy weight methodologies cause a significant documentation burden that increases the risk of project failure. We believe that SILs occur in heavy weight methodologies due to the plethora of information that must be recorded in a rigorous manner, distracting the software engineer from the key objectives of development.

We use the term “Software Information Leak” in the context of all the information that gets lost during the evolution and/or transformation of software. *More precisely, Software Information Leak (SIL) is defined as the relevant information once known, but not incorporated in later stages and thereby lost in software evolution.*

SILs are difficult to identify and analyze, as the object of study is information and quantifying information is a subjective process. Further, SILs occur almost transparently; however, the impacts of SILs are apparent and symptomatic. SILs can be categorized into two key categories:

1) Information lost within a life cycle phase or activity, due to improper recording and inadequate development process, and
2) Information lost between lifecycle phases or activities, due to improper translations.

As stated before, the second category of SILs are more complex, for the information conveyed using one language by a group of people with certain domain expertise needs to be translated into another language used by a different set of people with a different set of domain expertise. The problem is further compounded by the one-to-many relationships in the translation.

1.2. Metaphoric Illustration of Complexity

Researchers define complexity by citing the "phase transitions" occurring between highly ordered and highly disordered physical systems [8]. They argue that water possess complex physical properties that lie between the highly ordered state of ice crystals and the highly disordered movements of steam molecules. Corning states that complexity often implies the following attributes [8]:

1) a complex phenomenon consists of many parts,
2) many relationships/interactions exist among the parts; and
3) the parts produce combined effects (synergies) that are often novel, unexpected, even surprising.

Both water and software possess these attributes. The evolution of requirements into code is metamorphically similar to the transformations that occur when steam condenses into ice. Steam, is similar to requirements in many aspects. Both are ambiguous in nature and steam is as abstract with respect to ice, as requirements are with respect to code. The steam undergoes a set of transformations in which it gets solidified into ice, a more concrete form. During the transformations, the steam changes or leaks out some of its properties to become water, which changes some more properties to become ice. The final output, ice, has little resemblance to steam just as code has little

Proceedings of the Ninth IEEE International Conference on Engineering Complex Computer Systems Navigating Complexity in the e-Engineering Age
1050-4729/04 $20.00 © 2004 **IEEE**

resemblance to requirements. Within the water phase itself, there is a minimal version of the transformations from boiling water to cold water, just as there is a minimal transformation from architecture to design. Note that during this transformation, losing some properties such as heat is good, while losing some other such as molecular flexibility is not good. Likewise, losing some of the properties in requirements, such as ambiguity is good where as losing others such as change flexibility is not good.

2. Background

SILs occur through out the software process and beyond, hence a comprehensive discussion on all related areas is beyond the scope of the paper. However, some background relevant to our preliminary research into SILs is presented in this section.

2.1. Modeling Software Process

To respond to the ever-growing complexities of software, researchers like Walt Scacchi [9] have used modeling tools and techniques to understand and refine the software process. Modeling the software process helps to analyze and simulate complex software process execution. The analysis of the process model helps to study the static and dynamic behavior of the process including its consistency, completeness, internal correctness, traceability, as well as other semantic checks. Symbolically enacting the process helps to determine the path and flow of the intermediate state transitions in ways that can be made persistent, replayed, queried, dynamically analyzed, and reconfigured into multiple alternative scenarios. Visualizing the process and instances of the models provide a rich environment to view, navigate, interactively edit, and animate key software elements and convey an intuitive understanding of static and dynamic process characteristics.

2.2. Software Architecture

To respond to the sheer volume of software and consequential complexity, the software community has increasingly embraced architecture principles. Software architecture provides a framework to understand dependencies that exist between the various components, connections, and configurations reflected in the requirements. In Aladdin [10], an architectural dependency analyzer, the formalization of the dependency is done by means of the Rapide Architecture Description Language (ADL). While this is helpful, it is important to recognize that current research activities to define dependency analysis semantics are still at a nascent stage. The research in this area is aimed at providing a framework, which would advance the understanding of the dependencies expressed in ADLs.

2.3. Software Dependencies

Software dependencies can be classified into two important categories - software implementation dependencies and software architectural dependencies. There are two principal differences between the two. First, the objects in analysis are different; implementation dependencies reason with variables, statements and procedures, while architectural dependencies reason with more abstract objects such as components, connectors and ports. The second difference is in the execution models used for analysis; implementation dependency analysis deals with sequential, procedural languages while architectural analysis deals with ADLs that may use concurrent, event-based execution models [10].

Software implementation dependencies can be classified, based on the type of relation, into data dependencies (between two variables), calling dependencies (between two modules), functional dependencies (between a module and variable it computes), and definitional dependencies (between a variable and its type) [11]. The traditional dependency analysis, based on Program Dependence Graph (PDG) [12], combines the control and data flow relationships associated with functions and variables in a single, compact representation.

Software architectural dependencies can be direct or indirect [10]. In indirect dependence, the relationship that exists between the vertices in a PDG is transitive in nature (through intermediate vertices). Indirect dependency reduces tight coupling between components, but at the cost of complexity. In direct dependence no transitive relationships exist. Further, direct dependencies need not be and most of the time will not be reflexive or associative in nature. Software architectural dependencies can also be classified based on components, as between components (inter-component) and within component (intra-component). The intra-component dependencies help to understand various anomalies and interface behavior that is not evident in inter-component dependency view [10].

From evaluation perspective, software architectural dependencies can be evaluated, in terms of the distance (farther or nearer) or in terms of semantics (strongly related or weakly related). An object O is said to

Proceedings of the Ninth IEEE International Conference on Engineering Complex Computer Systems Navigating Complexity in the e-Engineering Age
1050-4729/04 $20.00 © 2004 IEEE

semantically depend on another object O' if the execution of O' affects the behavior of O [13]. Software architectural dependency analysis based on semantics requires formalization of the semantics language. The challenge in applying dependency analysis to architecture is to recast control and data flow relationships in terms of abstract components and their interactions.

2.4. Software Change Impact Analysis

Software is supposed to change and in that change there are key SILs. Change tolerance in software architectures introduces key complexity issues for today's software systems. Software Change Impact Analysis (SCIA) is defined as the determination of potential effects to a subject system resulting from a proposed software change [14]. Figure 1 depicts the basic SCIA process and its iterative nature.

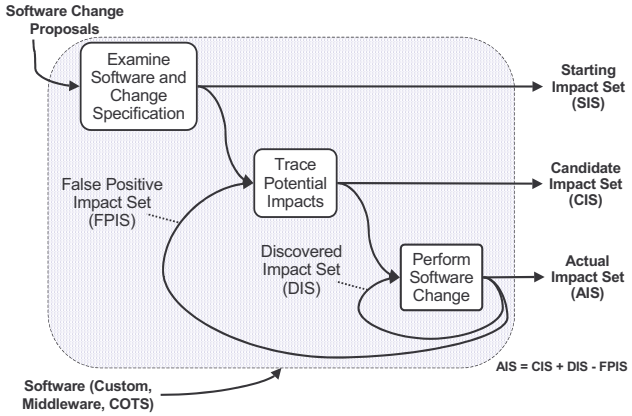

Figure 1. Software change impact analysis

The process starts by selecting an initial set of objects that are thought to be affected (SIS), and then by means of impact analysis the CIS is formed. The impacts found, while the actual change is performed, are grouped into DIS and FPIS represents the false positives that occur during estimation. The CIS plus additions of the DIS and minus deletions of FBIS should represent the AIS, which may not be unique, as a change can be implemented in several ways.

With this ever-increasing need to wade through volumes of software information, there is a need to identify the parts pertinent to software decisions. Since the human mind can handle only a limited number of these connections, automated assistance is the key to cutting analysis time and reducing the perceived complexity. Dependency semantics provide a means of characterizing dependency relationships between software life cycle objects. The semantics must be based on the expression medium as well as the kinds of dependency involved. SCIA methods are based on search algorithms and can be classified as follows:

- Semantically Guided: impact analysis directed by predetermined semantics of objects and relationships (e.g., may use semantic network algorithms) [15].
- Heuristically Guided: impact identification is directed by predetermined heuristics to suggest possible paths to examine or dubious ones to avoid (e.g. use rule-base of change experience) [13].
- Stochastically Guided: impact determination is guided by probabilities derived for the situation (e.g., based on characteristics of the SLOs to determine likelihood of the impact) [16].
- Hybrid Guidance: identify impacts using a combination of the above methods (e.g., transitive closure coupled with impact probabilities) – program slicing [17].
- Unguided/Exhaustive: identify impacts in a "brute force" manner using simple traceability relationships in the repository (e.g., transitive closure algorithms).

All of these represent mechanisms to deal with complexity in software system. However, to be effective, the types of dependencies analyzed must respond to key areas where SILs occur. In the following sections we examine how we propose to address SILs.

3. Technical Approach

There are natural patterns of dependencies that occur in developing systems. Modeling software has centered largely on functional, behavior, and data aspects of systems. However, with the advent of components and architecture, there is clearly other key dependency information that must be accounted for evolving a software system effectively.

The technical approach for this research started with the development of a software security impact analysis model for common criteria security evaluations [18]. Key dependencies were found to aid in identifying aspects about the software system. However, the decisions involved in evaluating security impacts were found to be complicated by the absence of dependency information. The lack of key dependency information not only made evaluator's job more complex by forcing a manual exploration of the entire corpus of software life cycle information, complex impact analyses of observed criteria with insufficient cues rendered the decision process often inconclusive. The very complex structure of security

requirements to the relevant software life cycle objects are frequently too difficult for a software engineer to understand without the aid of an impact model and visualization support [19].

We take this research further by generalizing the notion of dependencies beyond the software security criteria arena. Our initial impressions are that the concepts scale and the dependency structures hold for supporting the reduction of SILs.

3.1. SIL Categories based on Patterns

SILs can be grouped into three generic categories based on the methodologies and techniques available to identify and quantify them. The groups are:

- Defined: precise identification and quantification of such leaks are possible
- Partially defined: identification of such leaks is possible however quantification of such leaks is time consuming
- Undefined: identification and quantification of such leaks is either highly complex or impossible.

For example, data leaks (data loss) belong to the defined category since data losses can be readily identified and quantified. The SILs are identified to occur in technology level, domain level, requirement level, architecture level, design level and code level. Hierarchically, technology level is the most abstract level followed by the domain level. The SILs occurring in these two levels are generally undefined. By the time, the code level is reached, the SILs gets defined as code is concrete representation.

It is important to recognize the difference between lack and loss of information. SILs are concerned with information that was once known and lost, and not with information that was not known at all. The information that was never known cannot be captured using the environment developed to prevent SILs. However, there exists a possibility that the environment might nudge stakeholders towards areas where information is missing or lacking. Table 2 explains the various types of leaks briefly.

3.2. Evidence of a Trend

The upshot of this examination is that some of the leaks that occur cannot be seen or identified easily, but their effects are observable. Some leaks can be fixed with disciplined software engineering while others have a dependency implication that requires some automation to reduce the complexity.

Table 1 identifies some example leaks that cause certain changes to occur, that in turn, produce some effects on the transformation from abstract requirements to concrete implementation.

Table 1. Example cause and effect of SILs

Cause	Effect
Improper Documentation	Architectural Drift
Technical Expertise, Requirement elicitation, Language translation, subjective interpretation.	Incorrect Design
Requirements, Business constraints	Incorrect Architecture Style
Behavior, Control, Data	Software Defect

We believe that the use of a dependency model, that is based on SILs listed above, provide an initial framework from which to model.

4. Towards a Solution Approach

As with any problem, there are many ways to reduce SILs. Outlined below are short descriptions of some of the solution approaches.

4.1. Requirements Traceability

Requirements traceability techniques offer considerable help in reducing these SILs. Requirements traceability [20, 21] helps to ensure that the system is developed in accordance with the specified requirements and that these can be traced to their implementation components.

A robust requirement traceability model possesses bi-directional traceability, nomenclature of requirements based on criticality, capturing of design rationale, document consistency, interdependencies of components capture, accountability and automation support [21]. Hence, such a requirement traceability model effectively prevents requirement leaks in many situations. However, many requirements traceability models trace only requirements and not all dependencies within the plethora of software artifacts [22]. Further, while the traceability is conducted at a coarse grain, it is labor-intensive (resulting in inconsistency issues) [23] and is not formalized adequately to address the transition from requirements to design or from design to implementation. This calls for a more general approach.

Table 2. Software Information Leaks Categories

Name of Leak	Description	Type / Reason	Occurrence	Example
Tacit Knowledge Leaks	Experts' tacit knowledge getting lost during software evolution.	Undefined – as formalization of tacit knowledge is still not feasible.	Technology and domain level.	Domain experts' tacit knowledge encoded during requirements phase gets lost during coding phase.
Technical Expertise Leaks	Stakeholders' inexperience with technical considerations.	Undefined – as these leaks not identifiable since the information lost is not evident until much later time.	Technology level	The stakeholders may not decipher the technical aspects encoded by experts.
Business Constraints Leaks	Loss of business constraints information during system development.	Undefined – as many constraints are not fully definable from a computing perspective.	Domain level	Business people identify constraints, yet these constraints are often not fully defined from a systems perspective.
Domain Boundary Leaks	Loss of information due to vague domain analysis and boundary separation.	Undefined – as domain models are not formalized.	Domain level	Loss of domain related information when a software system designed for a specific domain is generalized to cater multiple domains.
Requirements Elicitation Leaks	Information loss due to improper requirements. The information loss occurring during requirements phase is aggravated by other leaks.	Partially Defined – Requirements get partially defined during customer-developer interaction.	Requirements level	Vague requirements that cannot be quantified (e.g. system should have good interface).
Design Variability Leaks	Design variability results in avenues for multiplicity in implementation and thereby leaks.	Partially defined - Design reviews reduce these leaks; yet reviews are seldom complete.	Architecture, Design level.	Design variations can occur when information changes or gets lost over time, resulting in potential leaks.
Design Decision Recording Leaks	Loss of information due to ineffective recording of design decisions.	Partially defined – Documentation policies exists; yet these policies are seldom sufficient.	Architecture, Design level.	Failure to update design records properly results in information loss.
Temporal Leaks	Information loss based on different context at different times due to recollection dissimilarities.	Defined	Spans across requirements, architecture, design and code level.	New requirement introduced that relates to an existing code, yet its dependency is not recorded.
Language Construct Leaks	Inadequate implementation due to inefficient language constructs and semantics.	Defined – as they are easy to identify and reduce (as source code is a formal representation).	Code level	Poor selection of language constructs miss convey intend and lead to errors in comprehension.

Proceedings of the Ninth IEEE International Conference on Engineering Complex Computer Systems Navigating Complexity in the e-Engineering Age
1050-4729/04 $20.00 © 2004 **IEEE**

Name of Leak	Description	Type / Reason	Occurrence	Example
Functional/ Behavior Leaks	Inadequate recordings of functional details that manifest in implementation errors.	Defined	Code level	Ambiguous functional design results in program errors.
Control Leaks	Loss of control information or improper control sequences that are manifest in the implementation.	Defined	Code level	Incorrect control sequences coded in by programmer.
Data Leaks	Loss of data during the execution of the software that result in software defect.	Defined – as data is a concrete entity.	Code level	Incorrect storage and execution of data.
Subjective Interpretation Leaks	Information leaks that occur because of different perceptions of the developer or other stakeholders.	Undefined in early phases and defined in code phase.	Spans across all levels.	Interpretations lead to imperfect capture or understanding of information.
Language Translation Leaks	Translation losses due to an incomplete mapping of terms and differences in semantic inference between languages.	Undefined in early phases and defined in code phase.	Spans across requirements, architecture, design and code level.	Translation leaks impede the understanding and introduce unwarranted complexity in the translations.
Documentation Leaks	Leaks due to inefficient software documentation process. Rampant documentation does not necessarily reduce the loss (just shifts it).	Undefined	Usually spans across requirements, architecture, design and code level.	Difficulty to determine, when and how to document, the level of granularity required, and how to retrieve information efficiently.
Traceability Leaks	The rapid movement or "opportunistic process" [24] coupled with missing leaks in the traceability matrix results in traceability leaks.	Undefined	Usually spans across requirements, architecture, design and code level.	In making design decisions, designers move back and forth between levels of abstraction ranging from application domain issues to coding issues or called "opportunistic process". This results in leaks.
Technology Implication Leaks	The impedance mismatch we find in technology with respect to business needs results in leaks.	Undefined - as leaks can be identified; but are often not fully defined.	Technology level	Most of these leaks impose additional complexity on the system solution space and if not detected lead to complications.

4.2. Dependency Modeling and Analysis

Of the various solution approaches considered, dependency modeling and analysis appears to be the most promising. For software, the most often applied area has been in source code with program dependence graphs [13]. However, this need not be the only focus of the analysis. With a reasonably formal model of dependencies between software life cycle objects, this notion can be extended to the rest of the software artifacts [14].

The dependency modeling builds a model of the software system under consideration based on the dependency between various artifacts. Determining the weight or the importance of each dependency link can be done by means of a relational approach and a weight assignment mechanism. There are two primary types of weight assignment mechanisms: Static Assignment and Dynamic Assignment.

Static Assignment. In static weight assignment, the weights for each type of dependencies are pre-calculated. The dependencies are assigned weights based on these pre-calculated weights. The process is both inefficient and inaccurate. However, the simplicity of the method makes it attractive for developing a proof-of-concept prototype.

Dynamic Assignment. The dynamic assignment method is more accurate and effective, since the weights are assigned dynamically based on fuzzy or other dynamic weight assignment logics. The difficulty comes in determining an effective mechanism for tuning the equation for the application to more universal sets of software artifacts.

Relational Approach. The relational approach is based on two main characteristics of dependencies: distance and semantic relationship between the dependencies. The distances, which indicates how far the elements are located from the object of interest, is indicative of the importance or strength of that dependency. The distance also indicates the level of indirection that exists between components. The semantics of a relationship help to identify the "influence" a dependency has on the component. The far most challenging part of relational approach is to gauge and normalize the semantic relationship.

Domain Analysis. To ensure that the dependencies are relevant and reflect the application domains, a domain analysis must be conducted. This generates key vernacular for the object and link types as well as an appropriate association mechanism for determining the strength of the dependencies encountered. Domain modeling and analysis have been effectively applied to software architectures and reuse [25]. In addition, a domain analysis method based on facets can support not only the creation of a common vocabulary capable of semantically unifying concepts perceived differently by different communities but also it can be used for creating ontologies that represent semantically uniform structures [26].

5. Development of the Prototype

As stated earlier, this work started with developing a Security Impact Analysis Visualization Environment (SIAVE) [19]. We have generalized the concepts to the broader set of software dependencies and intend on applying the prototype to the problem - SILs.

The most important advantage of the prototype would be the ability to enable visual identification of dependencies among various software objects, not only within each phase (e.g., within design), but also across all phases. In short, the prototype adapts the concept of forward and backward tracing as with roundtrip engineering, whereby a single line of code can be traced back to requirements and vice versa. We believe some SILs can be reduced using this approach as (1) it provides a common framework for all stakeholders to provide vital dependency information (no language translation leaks), (2) nudges stakeholders towards filling up incomplete/incorrect information, and (3) provides a framework for stakeholders to understand the impact of changes to a software system.

The proposed prototype would consist of three main components: the dependency framework, the analysis engine, and the visual environment (VE).

5.1. Dependency Framework

The generic dependency framework will define the rules and criteria on which the dependencies will be found and ranked. Developing this framework will take considerable effort but will provide the foundation for analysis and visualization components. The framework must be flexible to adapt to the domains of software. However, the adaptation must also have sufficient rigidity to ensure that definitions can be formalized into concrete logical decision syntax.

The framework development will require considerable domain analysis expertise to ensure that all the requisite information is captured with sufficient flexibility (to ensure future technology and domain knowledge changes) and rigidity (to ensure that formalization techniques can be used) for the analysis engine's use. The framework must possess evolution capability so as to expand to include other domains

and new technologies. The framework's rules will be encoded in XML format and stored in a database. It should be noted that the linchpin of the system is the framework and that an inappropriate definition of the framework can result in SIL, the very problem the solution is trying to address.

5.2. Analysis Engine

The Analysis Engine will be responsible for reading in the various software artifacts, applying the framework rules to the artifacts and extracting the important information pertaining to the dependencies from the artifacts. The analysis engine will have the capability to accept various types of inputs including word documents, design formats such as UML and code, and apply rules to those documents and find dependencies across and within those documents. The engine will also be responsible for processing the various inputs, such as type of dependencies that user is interested in.

5.3. Visual Environment

It is not enough to have long lists of software dependencies or even characterizations of the objects involved, if the information is so complex that the software engineer cannot comprehend it. This is the impetus of the visualization part of our research. We believe the information that would be generated from the analysis engine based on the framework would be enormous even for a typical system. Beyond 3D, virtual environments open possibilities of immersion and navigation to more effectively explore software dependencies. It is possible that all software artifacts from requirements to source code can be represented in a virtual environment to improve comprehension [27]. We believe that the VE benefits are important for our analysis of software dependencies. We have applied VE technology to security impact analysis [19]. This experience was convincing and indicated how VE should provide an effective mechanism for conveying software dependencies and navigating them for software understanding and complexity evaluation.

6. Projected Benefits and Limitations

We believe that the development of this capability would have a profound effect on the way software is developed and maintained. Letting the computer take on more of the burden of culling through the software artifacts and presenting the relevant map of dependencies between the software life cycle objects, frees the software engineers (i.e., developers, architects, maintainers) to focus on their job at hand. Further, formalizing the dependencies and providing navigation would enable better analysis and visualization of impact or ripple effects of software changes. Enabling forward and backward impact analysis visualization can help stakeholders to fully understand effects of the change.

The visual environment being developed cannot solve all SILs. Our current research gives us the confidence to further develop the formalization of dependencies. However, there exists a possibility that not all dependencies can be formalized and included in the system. Further, the system might require calibrations at regular intervals of time.

7. Conclusions

As society increasingly depends on software, the size and complexity of software systems continues to grow making them more difficult to understand and evolve. Manifold dependencies between critical elements of software now drive software architectures and increasingly influence the system architecture. Various SILs occurring in software development have a negative effect on the complexity of the software system by obscuring relevant information. SILs can be effectively reduced with a mechanism for identifying and quantifying the various software dependency relationships. A lightweight mechanism could span across various development phases, collating and helping different stakeholders to understand the dependency relationship between various artifacts and software objects. Further, the mechanism could have the capability to support easy updating of dependency information as and when it changes. Concepts of such a mechanism were presented in the paper along with a discussion of our solution.

References

[1] H. Ziv and J.D. Richardson. *The uncertainty principle in software engineering*. in *International Conference on Software Engineering*. 1997.

[2] A.M. Davis, *System Phenotypes.* IEEE Software, June 2003. **20**(4): p. 54-56.

[3] D. Marca and C.L. McGowan, *IDEFO/SADT Business Process and Enterprise Modeling*. 1998: Eclectic Solutions.

[4] RADC-TR-86-197, *Automated Life Cycle Impact Analysis System*. 1986, Rome Air Development Center, Air Force Systems Command, Griffiss AFB: Rome, NY.

[5] M.M. Lehman, *Software Evolution.* Encyclopedia of Software Engineering, 1994: p. 1202-08.

[6] D.E. Perry and A.L. Wolf, *Foundations for the Study of Software Architecture.* ACM SIGSOFT, 1992. **17**(4): p. 40-52.

[7] K. Beck, *Extreme Programming Explained: Embrace Change.* Reading, Massachusetts. 2000: Addison-Wesley.

[8] P.A. Corning, *Complexity is Just a Word!*, in *Technological Forecasting and Social Change*. 2001, Institute for the Study of Complex Systems: Palo Alto.

[9] W. Scacchi, *Experience with Software Process Simulation and Modeling.* Journal of Systems and Software, 1999. **46**: p. 183-92.

[10] J.A. Stafford and A.L. Wolf, *Architecture-Level Dependence Analysis for Software Systems.* International Journal of Software Engineering and Knowledge Engineering, 2000. **11**(4): p. 431-52.

[11] N. Wilde and R. Huitt, *Maintenance Support for Object-Oriented Programs.* IEEE Transactions on Software Engineering, 1992. **18**(12): p. 1038-44.

[12] J. Ferrante, K.J. Ottenstein, et al., *The Program Dependency Graph and its Use in Optimization.* ACM Transactions on Programming Langauges and Systems, 1987. **9**(3): p. 319-49.

[13] A. Podgurski and L.A. Clarke, *A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance.* Transactions on Software Engineering, 1990: p. 965-79.

[14] S.A. Bohner, *A Graph Traceability Approach to Software Change Impact Analysis*, in *Computer Science*. 1995, George Mason University: Fairfax, VA.

[15] M. Moriconi and T.C. Winkler, *Approximate Reasoning About the Semantic Effects of Program Changes.* IEEE Transactions on Software Engineering, 1990. **16**(9): p. 980-92.

[16] O.C. Gotel and A.C. Finkelstein, *An Analysis of the Requirements Traceability Problem.* Proceedings of First Conference on Requirements Engineering, 1994.

[17] M. Hutchins and K.B. Gallagher. *Improved Visual Impact Analysis*. in *International Conference on Software Maintenance*. 1998.

[18] R. Prieto-Diaz, *The Common Criteria Evaluation Process: Process Explanation, Shortcomings and Research Opportunities Commonwealth Information Security Center Technical Report CISC-TR-2002-003*. December 2002, James Madison University: Harrisonburg, VA.

[19] S.A. Bohner, D. Gracanin, et al., *Software Security Impact Analysis Visualization Research Report*. 2003, Commonwealth Information Security Center Technical Report, James Madison University.

[20] R. Balasubramaniam and J. Matthias, *Toward Reference Models for Requirement Traceability.* IEEE Transactions on Software Engineering, 2001. **27**(1): p. 58-93.

[21] B. Ramesh and M. Edwards, *Issues in the development of a requirements traceability model.* IEEE International Symposium on Requirements Engineering, 1993. **4**(6): p. 256-59.

[22] S. Bohner and R. Arnold, *Software Change Impact Analysis*. 1996: for the IEEE Computer Society Publications Tutorial Series.

[23] J. Cleland-Huang, C.K. Chang, et al., *Event-Based Traceability for Managing Evolutionary Change.* IEEE Transactions on Software Engineering, 2003. **29**(9): p. 796-810.

[24] B. Curtis. *Three Problems Overcome with Behavioral Models of the Software Development Process*. in *11th International Conference on Software Engineering*. 1989.

[25] W.R. Frakes, Prieto-Diaz, et al., *DARE: A Domain Analysis and Reuse Environment.* Annals of Software Engineering, 1998. **5**: p. 125-41.

[26] R. Prieto-Diaz. *A Faceted Approach to Building Ontologies*. in *IEEE International Conference on Information Reuse and Integration (IRI 2003)*. October 2003. Las Vegas, NV.

[27] S. Charters, C. Knight, et al. *Visualisation for Informed Decision Making; From Code to Components*. in *Workshop on Software Engineering Decision Support, 14th International Conference on Software Engineering and Knowledge Engineering*. 2002. Ischia, Italy.