THE IMPACT ON SOFTWARE MAINTENANCE OF FOURTH-GENERATION LANGUAGES AND OBJECT-ORIENTED PARADIGMS

Dr. Hardeep Singh
Kulwant Kaur

The attributes of good software should deliver the required functionality and performance to the user and should be maintainable: it must evolve to meet changing needs; dependable: it must be trustworthy; efficiency: it should not make wasteful use of system resources; and acceptable: it must be understandable, usable and compatible with other systems. The key challenges facing software engineering include Heterogeneity (developing techniques for building software that can cope with heterogeneous platforms and execution environments); Delivery (developing techniques that lead to faster delivery of software) and trust (developing techniques that demonstrate that its users can trust software).

Software change is inevitable. Its new requirements emerge when the software is used; the business environment changes; errors must be repaired; new computers and equipment is added to the system; and the performance or reliability of the system may have to be improved. A key problem for organisations is implementing and managing change to their existing software systems.

For software evolution, organisations must have huge investments in their software systems as they are critical business assets. In order to maintain value of these assets to business, they must be changed and updated with time. The majority of the software budget in large companies is devoted to evolving existing software rather than developing new software (Sommerville, 2003).

The paper is organized into following sections: Section 2 describes Program Evolution Dynamics. Section 3 states Software Maintenance Strategies. Section 4 discusses the Impact Analysis on Fourth-generation Languages and Object-Oriented Paradigms.

PROGRAM EVOLUTION DYNAMICS
Program can be classified into three types:

I. S-type Programs (“Specifiable”)
   Description: Problem can be stated formally and completely
   Acceptance: Is the program correct according to its specification?
   Specification: This software does not evolve. A change to the specification defines a new problem, hence a new program
II. P-type Programs (“Problem-solving”)

Description: Imprecise statement of a real-world problem
Acceptance: Is the program an acceptable solution to the problem?
Specification: This software is likely to evolve continuously
(a) The solution is never perfect, and can be improved; and
(b) The real-world changes and hence the problem changes.

III. E-type Programs (“Embedded”)

Description: A system that becomes part of the world that it models
Acceptance: depends entirely on opinion and judgment.
Specification: This software is inherently evolutionary changes in the
software and the world affects each other.

After major empirical studies, Lehman and Belady proposed that there were a number of ‘laws’, which applied to all systems as they evolved, are described in table 1.

<table>
<thead>
<tr>
<th>Lehman’s Law</th>
<th>Law Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. Continuing change</td>
<td>A program that is used in a real-world environment necessarily must change or become progressively less useful in that environment.</td>
</tr>
<tr>
<td>II. Increasing complexity</td>
<td>As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving and simplifying the structure.</td>
</tr>
<tr>
<td>III. Large program evolution</td>
<td>Program evolution is a self-regulating process. System attributes such as size, time between releases and the number of reported errors is approximately invariant for each system release.</td>
</tr>
<tr>
<td>IV. Organisational stability</td>
<td>Over a program’s lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.</td>
</tr>
<tr>
<td>V. Conservation of familiarity</td>
<td>Over the lifetime of a system, the incremental change in each release is approximately constant.</td>
</tr>
<tr>
<td>VI. Continuing growth</td>
<td>The functionality offered by systems has to continually increase to maintain user satisfaction.</td>
</tr>
<tr>
<td>VII. Declining quality</td>
<td>The quality of systems will appear to be declining unless they are adapted to changes in their operational environment.</td>
</tr>
<tr>
<td>VIII. Feedback system</td>
<td>Evolution processes incorporate multi-agent, multi-loop feedback systems and you have to treat them as feedback systems to achieve significant product improvement.</td>
</tr>
</tbody>
</table>
Lehman’s laws seem to be generally applicable to large, tailored systems developed by large organisations. It is not clear how they should be modified for Shrink-wrapped software products; small organisations; and medium sized systems [Lehman2000].

SOFTWARE MAINTENANCE STRATEGIES
Software maintenance is a process of modifying a program after it has been put into use. Maintenance does not normally involve major changes to the system’s architecture. Changes are implemented by modifying existing components and adding new components to the system (Jawadekar, 2004). The maintenance process is similar to development process. The focus of maintenance is on the following aspects:
1. Control on Software System Functions like reliability and usage.
2. Control on System Modifications to maintain originality of software system integrity, reliability and performance.
3. Make changes for better Quality of software system functioning, improving response, documentation, help and usefulness.
4. Software performance should not decline.

The Software Maintenance is handled through four strategies:
I. Corrective Maintenance
   Description  Changing a system to correct deficiencies in the way meets its requirements.
   Nature       Strategy to keep system in tune for day-to-day functions. The fault/errors are supposed to be of minor nature.
II. Adaptive Maintenance
   Description  Changing a system so that it operates in a different environment (computer, OS, etc.) from its initial implementation.
   Nature       The changes are of primary and secondary nature.
III. Perfective Maintenance
   Description  Improving the as delivered software for user enhancements.
   Nature       Improving the software as efficiency improvements.
IV. Preventive Maintenance
   Description  Strategy adapted to for continuous improvement in software, without waiting for failure to occur or for the user to ask for a change.
   Nature       Modifying the system to satisfy new requirements.

Software Maintenance Cost
The maintenance process is similar to development process. Maintenance costs is generally greater than development costs (2* to 100* depending on the application). It is affected by both technical and non-technical factors. It increases as software is maintained. Maintenance corrupts the software structure so makes further maintenance more difficult. Ageing software
can have high support costs (e.g. old languages, compilers etc.). Various maintenance cost factors involved are (Jawadekar, 2004).

I. Team stability

Maintenance costs are reduced if the same staff is involved with them for some time.

II. Contractual responsibility

The developers of a system may have no contractual responsibility for maintenance so there is no incentive to design for future change.

III. Staff skills

Maintenance staffs are often inexperienced and have limited domain knowledge.

IV. Program age and structure

As programs age, their structure is degraded and they become harder to understand and change.

Maintenance prediction is concerned with assessing which parts of the system may cause problems and have high maintenance costs

1. Change acceptance depends on the maintainability of the components affected by the change;
2. Implementing changes degrades the system and reduces its maintainability;
3. Maintenance costs depend on the number of changes and costs of change depend on maintainability.

Change prediction includes

1. Predicting the number of changes required and understanding of the relationships between a system and its environment.
2. Tightly coupled systems require changes whenever the environment is changed.
3. Factors influencing this relationship are
   • Number and complexity of system interfaces;
   • Number of inherently volatile system requirements;
   • The business processes where the system is used.

IMPACT ANALYSIS

The failure to use paradigms that enable software engineers to build and sustain maintainable software products has contributed in part to the software maintenance crisis and its associated cost. The results of a survey in the mid 1990’s of approximately 250 of the UK’s largest investors in software systems indicated that object-oriented technology was becoming a mainstream technology and that its potential to increase maintainability was one of the key driving forces behind this. Another trend which has had an impact upon the long-term maintainability of software systems is the adoption of quality assurance procedures. The
number of software companies adopting the ISO 9000 quality standards series has increased rapidly since the early 1990’s.

Managing change is fundamental to software maintainability. A program comprehension is one of the most expensive parts of software evolution. A key aspect in understanding the system is determining the impact of the changes that are proposed. This is impact analysis. An impact analysis looks at questions such as which system will be affected by a change, where do the changes need to be made, how much code needs to be modified. Effective impact analysis is a vital element in retaining an augmenting the maintainability of a system.

Good example of the key importance of impact analysis in modifying software systems is the tackling of the Year 2000 problem, identifying all systems and subsystems that would not be able to cope with the changeover from 1999 to 2000.

Techniques developed to facilitate impact analysis allow greater efficiency and accuracy in assessing the impact and risks of making a change, thus decreasing the chances of inappropriate changes being embarked upon. Impact analysis is important factor in planning and managing software change and has the effect of making the resultant changed systems better and more maintainable.

### Table 2: Quality factors and their impact upon maintainability

<table>
<thead>
<tr>
<th>I. Fitness for Purpose</th>
<th>Does the product do the job it was intended to do?</th>
<th>-criterion by which to measure quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>II. Correctness</td>
<td>Structured system development and maintenance can reduce in the number of errors. Building correctness into a system has the obvious advantage that less time will be spent on corrective maintenance.</td>
<td></td>
</tr>
<tr>
<td>III. Portability</td>
<td>Portability encompasses hardware platforms, operating systems, and programming languages. The building of portability into a system means the avoidance of features which tie a software system to a particular platform or language.</td>
<td></td>
</tr>
<tr>
<td>IV. Testability</td>
<td>A system that is easy to test is also easier to change effectively because it is easier to test the changes made.</td>
<td></td>
</tr>
<tr>
<td>V. Usability</td>
<td>If a system for any reason is not used, it may as well not exist. Maintenance is only an issue for a system that is used and that evolves with use.</td>
<td></td>
</tr>
<tr>
<td>VI. Reliability</td>
<td>Reliability is closely allied to trust. If a customer has no trust in a system, he or she will not use it.</td>
<td></td>
</tr>
<tr>
<td>VII. Efficiency</td>
<td>The efficiency of a system, how it makes use of a computer’s resources, is not always easy to quantify.</td>
<td></td>
</tr>
<tr>
<td>VIII. Integrity</td>
<td>The integrity of a system can be interpreted in two ways: · Is the system safe from unauthorised access? · Can system integrity be guaranteed from a</td>
<td></td>
</tr>
</tbody>
</table>
configuration management point of view? Has the system been built from a consistent and reproducible set of modules?

<table>
<thead>
<tr>
<th>IX. Reusability</th>
<th>Reusability of a system is likely to have an advantageous effect upon future systems and enhancements.</th>
</tr>
</thead>
<tbody>
<tr>
<td>X. Interoperability</td>
<td>The ability of a system to interact with other systems.</td>
</tr>
</tbody>
</table>

I. Fourth-Generation Languages

_Characteristics_: 4GLs which consists of simple query languages, complex query and update languages, report generators, graphics languages, decision-support languages, application generators, specification languages, very-high-level programming languages, parameterised application packages and application languages. There is a wide variety of commercially available 4GLs.

1. SQL (Structured Query Language) is a query language that is used in conjunction with a database and allows the user to obtain various types of information from the database.

2. Oracle and Ingres are both examples of decision-support languages. They permit users to build databases and manipulate data in order to obtain decision-support information. These tools are commonly used for business-oriented problems such as analysis of financial matters and for making investment projections.

Some 4GLs are described as non-procedural languages because they allow the user to specify 'what' the application is to do but not in detail 'how' this is to be achieved. Examples of non-procedural languages include application generators, database query languages and form designers. Fourth-Generation Languages are easy to use, can be employed by non-professional programmers to obtain results, require an order of magnitude fewer instructions than other conventional languages such as COBOL, designed for on-line operations, non-technical users can learn to use a subset of the language after a two-day training course and documentation is automated.

**Impact on Maintenance**: By virtue of the above characteristics, the use of 4GLs to develop and maintain applications impacts on software maintenance in many ways.

1. **Increased Productivity**
   One of the major problems with applications developed with third-generation languages such as Cobol and Fortran is the time required to effect change. In organisations that depend on the effective operation of their software systems, delays in implementing change can be disruptive and costly. One of the strengths of
II. Reduction in Cost

4GLs is that they enable more rapid implementation of change, thereby increasing productivity. Due to the reduction in time required to develop and maintain applications using 4GLs, they tend to cost less than conventional application development. Major enhancements can be undertaken in hours, unlike traditional programming language applications which may take several weeks or months.

III. Ease of Understanding

In an industry where there is a shortage of qualified staff and a high turnover of maintenance personnel, the desire to shorten comprehension time is one of the driving forces behind 4GLs.

IV. Automatic Documentation

In many 4GL-oriented applications, a large proportion of the documentation is generated automatically. Considering the significant role that documentation plays in maintenance of other people’s programs, this feature can ease considerably the job of maintenance personnel.

II. Object-Oriented Paradigms

Characteristics: Early procedural programming is analogous to the ‘waterfall’ model. As the problems to be addressed became more complex, procedural languages and programming techniques became more sophisticated to deal with them. Techniques of structured systems analysis and design were developed. There are almost two dozen major object-oriented programming languages in use today. C++ and Java are the most popular object-oriented languages today. The Java programming language is designed especially for use in distributed applications on corporate networks and the Internet. Comparison between different object oriented languages is shown in table 3.

The concepts and rules used in object-oriented programming provide following benefits like:

- The concept of a data class makes it possible to define subclasses of data objects that share some or all of the main class characteristics. Inheritance is property of OOP forces a more thorough data analysis, reduces development time, and ensures more accurate coding.

- Since a class defines only the data it needs to be concerned with, when an instance of that class (an object) is run, the code will not be able to accidentally access other program data. This characteristic of data hiding provides greater system security and avoids unintended data corruption.

- The definition of a class is reusable not only by the program for which it is initially created but also by other object-oriented programs.
The concept of data classes allows a programmer to create any new data type that is not already defined in the language itself.

Table 3: Programming Language Comparison

<table>
<thead>
<tr>
<th></th>
<th>Smalltalk</th>
<th>Java</th>
<th>C#</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object-Orientation</strong></td>
<td>Pure</td>
<td>Hybrid</td>
<td>Hybrid</td>
<td>Hybrid / Multi-Paradigm</td>
</tr>
<tr>
<td><strong>Static / Dynamic Typing</strong></td>
<td>Dynamic</td>
<td>Static</td>
<td>Static</td>
<td>Static</td>
</tr>
<tr>
<td><strong>Generic Classes</strong></td>
<td>N/A</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Inheritance</strong></td>
<td>Single</td>
<td>Single class, multiple interfaces</td>
<td>Single class, multiple interfaces</td>
<td>Multiple</td>
</tr>
<tr>
<td><strong>Feature Renaming</strong></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Method Overloading</strong></td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Operator Overloading</strong></td>
<td>Yes?</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Higher Order Functions</strong></td>
<td>Blocks</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Lexical Closures</strong></td>
<td>Yes (blocks)</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Garbage Collection</strong></td>
<td>Mark and Sweep or Generational</td>
<td>Mark and Sweep or Generational</td>
<td>Mark and Sweep or Generational</td>
<td>None</td>
</tr>
<tr>
<td><strong>Uniform Access</strong></td>
<td>N/A</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Class Variables / Methods</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Reflection</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Access Control</strong></td>
<td>Protected Data, Public Methods</td>
<td>public, protected, &quot;package&quot;, private</td>
<td>public, protected, private, internal, protected internal</td>
<td>public, protected, private, &quot;friends&quot;</td>
</tr>
<tr>
<td><strong>Design by Contract</strong></td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td><strong>Multithreading</strong></td>
<td>Implementation-Dependent</td>
<td>Yes</td>
<td>Yes</td>
<td>Libraries</td>
</tr>
<tr>
<td><strong>Regular Expressions</strong></td>
<td>No</td>
<td>Standard Library</td>
<td>Standard Library</td>
<td>No</td>
</tr>
<tr>
<td><strong>Pointer Arithmetic</strong></td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Language Integration</strong></td>
<td>C</td>
<td>C, some C++</td>
<td>All .NET Languages</td>
<td>C, Assembler</td>
</tr>
</tbody>
</table>

*Note: N/A indicates that a topic or feature is not applicable to the language.*

*Based on number of source code lines per function point*

*Source: http://www.jvoegele.com/*
The OO Method targeted identifies a spectrum of software domains, including but not limited to:

- Batch transformation
- Continuous transformation
- Interactive interface
- Dynamic simulation
- Real-time system
- Transaction manager

While these classifications are useful, the criteria selected compares information systems, real-time embedded systems, and component libraries (e.g., class libraries and application frameworks). Some of major advantages are simplicity, modularity, modifiability, extensibility, maintainability, re-usability, modular structure and Graphical user interfaces. In order to maintain and modify existing code, new objects can be created with small differences to existing ones. The typical programmer’s advantages are

1. Multiple-inheritance, where classes can inherit from multiple super classes.
2. Multi-method dispatch, where operations can be specialized not just on a single object, but on any of the arguments for the operation, or any combination of those arguments.
3. Method combination, in which the means by which several applicable operations to a particular set of arguments can be combined, is itself an object-oriented specification, definable by the programmer.
4. Operations that can specialize on particular instances of classes, rather than on just the broad class of the arguments.
5. Shared class variables (slots, attributes) that are shared by all instances of a class, in addition to instance variables which are unique to each instance.
6. Specification of the creation operation for instances of a class as an object-oriented program, definable by the programmer.
7. The behaviour of operations on objects (methods) is themselves defined as first-class objects, which can be specialized by the programmer.
8. Access to classes themselves as first-class objects. The objects which define the behaviour of classes are themselves classes, usable directly by the programmer.

Despite the benefits of deploying object-oriented techniques, their application is not universal. It takes time to bring a new paradigm on stream. Thus, even if object-orientation becomes the accepted paradigm for software development, the need for expertise in non object-oriented techniques will persist. It is not necessarily appropriate or cost effective to
convert existing systems. A large proportion of mainstream programmers had little or no educational background in object-oriented techniques.

The OOP philosophy is not always the best philosophy for many types of applications. Also a language philosophy dictates the built-in functions and operations provided. If neither the syntax nor the built-in functions are optimized for what you are working on, then you are wasting a lot of time.

For simple programs, procedural approach can be a good solution. But for the hard ones it is not. On the other hand for simple problems, approaching OOP concept is rather lengthy, time consuming and not useful. It is however, useful for tough problems.

However, there is a slight cost in terms of efficiency. As objects are normally referenced by pointers, with memory allocated dynamically, there is a small space overhead to store the pointers, and a small speed overhead to find space on the heap (at run-time) to store the objects. For dynamic methods there is an additional small time penalty, as the method to choose is found at run time, by searching through the object hierarchy (using the class precedence list for the object). These days, these efficiency penalties seem to be of relatively little importance compared with the software engineering benefits.

**Impact on Maintenance:** The object-oriented paradigm has main advantage that there is a common view of what happens in the real world, how it is implemented in the software and the transformation from the analysis phase to the implementation phase is much clearer and less error prone (Pressman, 2005). Advantages of an object-oriented view of software development over orthodox methods such as top-down structured design are:

1. It yields smaller systems due to the reuse of common mechanisms.
2. It facilitates the production of systems that are resilient to change and hence easily evolvable considering that its design is based on a stable intermediate form. The potential of object orientation to increase maintainability is a contributory factor to its popularity.
3. The risks associated with complex systems are reduced because object-oriented systems are built to evolve incrementally from smaller components that have been shown to satisfy the requirements.
4. Products from object-oriented decomposition can be stored in libraries and reused and has advantage of expediting personnel productivity, high quality software and a reduction in cost.
5. The complexity of the interfaces are locally defined methods and their number are the most important factors for estimating the adaptive maintenance effort, their complexity is a cost for comprehending the system and thus for its manipulation.

6. Total effort of maintenance can be better estimated by using functional-based metrics since metrics based on counting class members neglecting several aspects that produce a strong impact on documentation and test codes (Fioravanti and Nesi, 2001).

**CONCLUSION**

Software evolution, i.e. the process by which programs change shape, adapt to the marketplace and inherit characteristics from preexisting programs, has become a subject of serious academic study in recent years. There are three principle types of software maintenance. These are maintenance to repair defects in the software, maintenance to adapt the software to different operation environment and maintenance to add or to modify the functionality of the system. Maintenance is a team activity where a team is a cohesive work group working in collaboration to ensure quality and to control the quality from declining. The maintenance team should be knowledgeable and skillful enough to implement strategies. A good software development team with understanding can lead to system design that would score highest score.

Make maintenance a magnet. Find ways to attract people to the maintenance task. Some companies do this by paying a premium to maintainers. Link maintenance to quality assurance and plan for improved maintenance technology. There are now many tools and techniques for doing software maintenance better. Training and tools selection and procurement should be high on the concerned maintenance manager's list of tasks. Emphasize "responsible programming." The maintainer typically works alone. The best way to maximize the effectiveness of this kind of worker is to make them feel responsible for the quality of what they do. Note that this is the opposite of the now popular belief in "egoless programming," where we try to divest the programmer's personal involvement in the final software product in favor of a team involvement. It is vital that the individual maintainer be invested in the quality of the software product if that product is to continue to be of high quality.

**REFERENCES**


**WEB RESOURCES**

http://www.jvoegele.com/