NORTH ATLANTIC TREATY ORGANISATION

RESEARCH AND TECHNOLOGY ORGANISATION

AC/323(IST-026)TP/190

www.rto.nato.int

**RTO TECHNICAL REPORT** **TR-IST-026**

# Evolutionary Software Development

(Développement évolutionnaire de logiciels)

Final Report of the Task Group IST-026/RTG-008.

Published August 2008

NORTH ATLANTIC TREATY ORGANISATION

RESEARCH AND TECHNOLOGY ORGANISATION

AC/323(IST-026)TP/190

www.rto.nato.int

**RTO TECHNICAL REPORT** **TR-IST-026**

# Evolutionary Software Development

(Développement évolutionnaire de logiciels)

Final Report of the Task Group IST-026/RTG-008.

# The Research and Technology Organisation (RTO) of NATO

RTO is the single focus in NATO for Defence Research and Technology activities. Its mission is to conduct and promote co-operative research and information exchange. The objective is to support the development and effective use of national defence research and technology and to meet the military needs of the Alliance, to maintain a technological lead, and to provide advice to NATO and national decision makers. The RTO performs its mission with the support of an extensive network of national experts. It also ensures effective co-ordination with other NATO bodies involved in R&T activities.

RTO reports both to the Military Committee of NATO and to the Conference of National Armament Directors. It comprises a Research and Technology Board (RTB) as the highest level of national representation and the Research and Technology Agency (RTA), a dedicated staff with its headquarters in Neuilly, near Paris, France. In order to facilitate contacts with the military users and other NATO activities, a small part of the RTA staff is located in NATO Headquarters in Brussels. The Brussels staff also co-ordinates RTO's co-operation with nations in Middle and Eastern Europe, to which RTO attaches particular importance especially as working together in the field of research is one of the more promising areas of co-operation.

The total spectrum of R&T activities is covered by the following 7 bodies:

- AVT Applied Vehicle Technology Panel
- HFM Human Factors and Medicine Panel
- IST Information Systems Technology Panel
- NMSG NATO Modelling and Simulation Group
- SAS System Analysis and Studies Panel
- SCI Systems Concepts and Integration Panel
- SET Sensors and Electronics Technology Panel

These bodies are made up of national representatives as well as generally recognised 'world class' scientists. They also provide a communication link to military users and other NATO bodies. RTO's scientific and technological work is carried out by Technical Teams, created for specific activities and with a specific duration. Such Technical Teams can organise workshops, symposia, field trials, lecture series and training courses. An important function of these Technical Teams is to ensure the continuity of the expert networks.

RTO builds upon earlier co-operation in defence research and technology as set-up under the Advisory Group for Aerospace Research and Development (AGARD) and the Defence Research Group (DRG). AGARD and the DRG share common roots in that they were both established at the initiative of Dr Theodore von Kármán, a leading aerospace scientist, who early on recognised the importance of scientific support for the Allied Armed Forces. RTO is capitalising on these common roots in order to provide the Alliance and the NATO nations with a strong scientific and technological basis that will guarantee a solid base for the future.



Published August 2008



ISBN 978-92-837-0042-5

# Table of Contents


Page

**Executive Summary and Synthèse** ES-1

**Chapter 1 – Introduction, Motivation, Hypothesis** 1-1
1.1 Introduction 1-1
1.2 Motivation 1-1
1.3 Hypothesis 1-2
1.4 References 1-3

**Chapter 2 – Key Concepts** 2-1

**Chapter 3 – Supporting Architectures** 3-1
3.1 The Role of Architectures 3-1
3.2 Model Driven Architecture Development 3-2
3.3 Adaptable Architectures 3-2
3.4 Dynamic Architectures 3-2
3.5 Product-Line Architectures 3-3
3.6 Commercial-Off-The-Shelf Software and Architecture: Friend or Foe 3-3
3.7 References 3-5

**Chapter 4 – State of the Art / Practice** 4-1
4.1 State of the Art 4-1
4.1.1 Other Workshops and Conferences 4-1
4.1.2 Tom Gilb 4-1
4.1.3 Evolutionary Procurement 4-1
4.2 Concrete Examples Taken from Symposium IST-034 RSY-010 4-2
4.3 Related Concepts 4-3
4.3.1 Prototyping 4-3
4.3.2 User-Centered Design 4-3
4.3.3 Agile Software Development 4-3
4.3.4 Achieving Specific Non-Functional Requirement Targets 4-4
4.4 References 4-4

**Chapter 5 – Issues and Concerns (FAQ)** 5-1
5.1 Questions and Answers 5-1
5.1.1 Why do Developers and Customers Use the Evolutionary Process? (Benefits) 5-1
5.1.1.1 Partial Functionality is Available Early 5-1

5.1.1.2 Partial Functionality Can be Achieved Before Complete Funding 5-1
5.1.1.3 Learn-From-Use Feedback Corrects Requirements 5-1
5.1.2 Why Don't Developers and Customers Use the Evolutionary Process? 5-1
5.1.2.1 Lack of Familiarity with ESD 5-1
5.1.2.2 Concern that ESD is Not Permitted 5-2
5.1.2.3 Lack of Tool Support for ESD 5-2
5.1.3 How Can ESD be Managed? 5-2
5.1.4 How Can ESD be Taken into Account in Current Processes? 5-2
5.1.5 Cost Estimation of ESD 5-2
5.1.6 QA (Quality Assurance) and ESD 5-3
5.1.7 ESD and Business Decisions 5-3
5.1.8 Relationship of ESD to Open Source Movement 5-3
5.1.9 Relationship of ESD to the Agile Movement 5-3
5.2 Concerns 5-3
5.2.1 Evolution is Risky 5-3
5.2.2 Evolutionary Process Facilitates Decreasing Risk 5-4

**Chapter 6 – Unresolved Challenges (Future Research, Possible Improvements) 6-1**
6.1 Software Architecture 6-1
6.1.1 Dynamic and Adaptive Architectures 6-1
6.1.2 Product Lines 6-1
6.1.3 Service Oriented Architectures 6-2
6.2 Implications for Procurement 6-2
6.3 Cost Estimation 6-3
6.4 Project Management, Including Product and Project Metrics 6-3
6.5 Tool Support, Including Configuration Management 6-3
6.6 How to Economize in Testing 6-3
6.7 Documentation 6-4
6.8 Aids to Retrain Users 6-4
6.9 Scaffolding 6-4
6.10 COTS-Based Systems 6-4
6.11 Interoperability 6-5
6.12 Simultaneous Field Support for Multiple Releases 6-5
6.13 Automated Upgrading 6-6

**Chapter 7 – Recommendations/Conclusions 7-1**
7.1 ESD Should be More Widely Adopted for Long-Lived Systems 7-1
7.2 The TG-008 Web Site Should be Continued as an Active Vehicle for Information Interchange 7-1

**Annex A – Definitions of Key Concepts A-1**

**Annex B – Links to Other Research Groups and Workshops Addressing Software Evolution B-1**

**Annex C – Review of History of Task Group C-1**

Annex D – Members of Task Group 026/RTG-008 D-1

Annex E – Iterative and Incremental Development: A Brief History E-1

# Evolutionary Software Development

## (RTO-TR-IST-026)

# Executive Summary

This is the final report of the task group IST-026/RTG-008.

For many years, military software development in many countries mandated a sequential predictive development process, often characterized by US DoD MIL-STD-2167A and referred to as the waterfall model. This is despite the fact that ever since the 50's, some successful military software projects have instead employed iterative development processes, and in each decade leaders of software thought have advocated them. The situation persists even though 2167A itself was revoked in 1994 because it led to many project failures: cost and delivery date overruns, projects abandoned before completion and products which even if delivered were never fielded because they were too far from what the customer actually needed. The intrinsic flaw in the waterfall process was that it did not acknowledge inherent incompleteness of project requirements and uncertainties in available implementation technology at the time of project initiation. Moreover, it did not acknowledge the evolution in system role, available technology, economics or user expectations that naturally occurs during an extended development period, never mind during a long operational system life. Software is not subject to the driver of hardware procurement policy whereby, once delivered and deployed, changes to individual units are so expensive that (except perhaps for minor maintenance and a possible "mid-life kicker" upgrade) the requirements that the product meets must remain stable throughout an extended operational life. Many DCI and PCC call for software systems that can be fielded early and can be adapted rapidly as needs evolve.

The commercial world, needing to respond quickly to business opportunities, as well as to respond to pressures from competitive product offerings, had moved long ago to a periodic release cycle over the whole product life. This process, is called evolutionary software development (ESD), typically involves several releases being in planning, perhaps even development, concurrently. Successive releases can accommodate changes in requirements. Features not complete in time slip to the next release rather than holding up the release ship date. The Agile Manifesto, 2001, has been particularly influential.

We found that ESD is appropriate for military software development and there are success stories of its use. However, we found that there are unresolved challenges in how best to manage ESD, to do cost estimation for ESD, to perform quality assurance for ESD and to manage risk with ESD. Tool support for ESD, appropriate scaffolding to facilitate ESD, methods of reducing costs associated with repeated integration, repeated testing, repeated retraining, all are open topics for improvements. We conclude that ESD is effective, and should be the normal process used for software that is expected to evolve.

# Développement évolutionnaire de logiciels

## (RTO-TR-IST-026)

# Synthèse

Ceci est le rapport final du groupe de travail IST-026/RTG-008.

Pendant longtemps, dans de nombreux pays, le développement de logiciels militaires a requis un processus de développement prédictif séquentiel, souvent caractérisé par le DoD MIL-STD-2167A américain et désigné sous le nom de modèle en cascade. Et cela en dépit du fait que, depuis les années 50, certains projets réussis de logiciels militaires ont utilisé à la place des processus de développement itératif, et qu'à chaque décennie, les leaders de la pensée logicielle ont prôné leur cause. Cette situation perdure, bien que le 2167A lui-même ait été retiré en 1994 car il menait à l'échec de nombreux projets : dépassements des coûts et des dates de livraison, projets abandonnés avant leur achèvement, ou produits qui, bien que livrés, n'étaient jamais utilisés car trop éloignés des besoins réels du client. Le défaut intrinsèque du processus en cascade était de ne pas prendre en compte l'inachèvement inhérent aux exigences des projets et les incertitudes de la technologie d'implémentation disponible au moment du lancement du projet. En outre, il ne reconnaissait pas l'évolution du rôle du système, de la technologie disponible, de l'économie ou des attentes de l'utilisateur, qui se produisait naturellement au cours d'une période de développement prolongée ou de la longue durée de vie d'un système opérationnel. Les logiciels ne sont pas soumis à la politique d'acquisition des pilotes de matériel selon laquelle, une fois ces derniers livrés et déployés, les modifications apportées aux unités individuelles sont si onéreuses que (à l'exception peut-être d'une maintenance mineure ou d'une éventuelle mise à niveau à la moitié de sa vie) les exigences auxquelles répond le produit doivent rester stables tout au long d'une durée de vie opérationnelle prolongée. Nombre d'Initiatives sur les capacités de défense (DCI) et de Cellules de coordination du partenariat (PCC) demandent des systèmes logiciels pouvant être déployés tôt et s'adapter rapidement en fonction des besoins.

Le monde commercial, devant saisir rapidement les opportunités professionnelles tout en faisant face à la pression des offres de produits concurrents, a depuis longtemps évolué vers un cycle de versions périodiques tout au long de la durée de vie du produit. Ce processus, appelé développement évolutionnaire de logiciels (ESD), implique généralement que plusieurs versions soient en projet, peut-être même en cours de développement, simultanément. Les versions successives peuvent répondre à l'évolution des exigences. Les fonctions qui n'ont pu être terminées à temps passent sur la version suivante, plutôt que de retarder la date d'expédition de la version. Le Manifeste Agile de 2001 a eu beaucoup d'influence.

Nous sommes parvenus à la conclusion que l'ESD est adapté au développement de logiciels militaires. Il existe des cas où il a déjà été employé avec succès. Toutefois, nous avons découvert qu'il restait encore des défis à relever : déterminer la meilleure manière de gérer l'ESD, réaliser une estimation de son coût, mettre en place une assurance qualité et gérer les risques. Un support d'outils pour l'ESD, une architecture appropriée pour sa facilitation, des méthodes de réduction des coûts associées à une intégration répétée, des tests fréquents, des recyclages réguliers, il s'agit là de sujets ouverts à l'amélioration. En conclusion, l'ESD est efficace et devrait être le processus normal utilisé pour les logiciels que l'on s'attend à voir évoluer.

# Chapter 1 – INTRODUCTION, MOTIVATION, HYPOTHESIS

## 1.1 INTRODUCTION

Military software procurement has traditionally followed a phased development process called the waterfall model. A project moves sequentially through stages of concept, requirements elicitation, specification definition, preliminary design, detail design, unit implementation, system integration, acceptance testing, and deployment. Reviews between the phases provide convenient assessment points for project review, as well as a convenient mechanism for gating progress payments to contractors.

Despite this, iterative [1] and incremental development processes have a long history going back to the 1950's, of successful application to software systems, many military, and "in each decade has been advocated by prominent leaders of software engineering thought" [2].

An excellent survey of this history, published in IEEE Computer June 2003" by C. Larman and V. Basili, is attached to this report by permission as Annex E. It is also worth noting that, if read carefully, the 1970 article [3] by W. Royce generally regarded as the origin of the waterfall model can be seen as actually advocating iterative development. Today iterative and incremental development is promoted as an essential aspect of Agile Programming, a rapidly growing trend in software development process [4].

MIL-STD-2167A of the US DoD was widely regarded as prescribing a development process based on the waterfall model, and the failures ascribed to this model lead to that standard being replaced in December 1994. Nevertheless, more than a decade later, waterfall development of military software is still prevalent in many nations. In the meantime, commercial practice has heavily shifted to a software development process called Evolutionary Software Development.

## 1.2 MOTIVATION

Many DCI (Defence Capability Initiatives) [5] and PCC (Prague Capabilities Commitment) [6] describe systems which require software that can be fielded early in the lifecycle of the systems, and can be readily adapted as those systems evolve. The traditional development process has a poor record in meeting these objectives, so a different development process is called for.

The procurement agencies of many nations follow a deliberate policy that alternate suppliers should be eligible for follow-on programs. This carries over from hardware, where the functionality and performance of a product is largely fixed at initial delivery, and on-going maintenance is considered a low level activity of repairs, preventative precautions, and minor enhancements. The assumption is that these activities require different resources than initial development, and so should be open to alternate suppliers. Again, following hardware practice, a major redevelopment effort may be undertaken after a system has been field for a few years, a so-called "mid-life kicker". The assumption is that this redevelopment might benefit from fresh ideas, so should be open to alternate suppliers.

Experience with software systems, however, is that software support often needs to be quite different than this. Changing requirements, the availability of new technology, and rising user expectations result in software support involving frequent and substantial modifications to fielded software systems. Ramping up a new team each time to learn the system in sufficient depth of understanding to be able to make these changes effectively is time-consuming and expensive. For example, when a series of weapons has been developed over time,

each offering exactly the same functionality, this policy has been responsible for development of software over the entire series at excessive cost, not just in fiscal terms. The intermediate position, that the developer should be responsible for a planned sequence of releases, has not even been considered.

The Tomahawk missile is but one of many examples of a system where repurposing over the system lifetime has resulted in decades of almost continuous software development.

## 1.3 HYPOTHESIS

In contrast to the conventional delivery of a product in a single "big bang", the Evolutionary Software Development process delivers a product over a s planned sequence or releases based on a "learn from experience in use" cycle. This philosophy arises in response to the following hypotheses:

1) Early fielding of partial functionality is generally better than no deployment until full functionality is confirmed. Funding or staffing uncertainty can delay full functionality indefinitely, and a working system with some functionality is usually better than none. It follows that incremental delivery may happen over several releases.

2) Requirements are never completely defined (large and complex systems, customers have trouble specifying needs, new technology triggers new requirements, ...). Since incremental delivery only attempts to support partial functionality in each release, new requirements may be accommodated in subsequent releases.

3) Incremental delivery by itself is not sufficient, because deployed functionality may have to be withdrawn if requirements are dropped, or if conflicting new requirements take precedence. Experience with partial functionality is often a source of changes in requirements.

4) Obsolescence of hardware, software, laws, and regulations causes requirements to be withdrawn over the product lifecycle, perhaps even during initial development.

5) Environment of the system is changing all the time (technology, politics, procedures, mission, tactics, etc.). A system that is not matched to the current environment may no longer be useful.

6) Civil and military systems are sometimes perceived to have very different requirements (specifically the military point of view regarding evolutionary system development). In fact this perceived difference is often better characterized as a difference in point of view between commercial versus governmental (long-term systems). The difference often derives from the procurement process, i.e. open tendering, gated progress payments, etc.

7) The customers are often not themselves the end-users. Indeed, for many systems the actual end-users are unknown or at least unavailable during procurement and initial development. This forces feedback options that might not be chosen when known and identifiable users are available.

8) Implementation issues are not necessarily well understood at the time that development is initiated, with the consequence: that some exploratory developments are needed, recognizing that unsuccessful directions that should be abandoned must be expected.

9) Time and finance constraints often do not allow you to develop the system you ideally would like to have, nevertheless you would like to get a part of the system that can later be extended rather than something you will have to throw away. This has significant consequences for the procurement process and for the development process, especially for project management.

10) End-users are typically able to criticize/invalidate models and prototypes, but not to specify requirements precisely in advance, nor to validate before a fully working system is available.

11) Multiple releases of a system imply continuous integration and testing. This has the consequence: of accentuating process improvement.

Note that because Evolutionary Software Development addresses changes over the full life cycle of a system, it is not quite the same as either the spiral model or the agile process, which both typically use iteration during the initial development process but then may revert to a conventional maintenance and enhancement lifecycle.

## 1.4 REFERENCES

[1] Luckey, P.H., Pittman, R.M. and LeVan, A.Q. (1992). "Iterative Development Process with Proposed Applications", Technical Report, IBM Owego, New York.

[2] Larmen, C. and Basili, V.R. "Iterative and incremental development: a brief history" IEEE Computer June 2003, pp. 2-11.

[3] Royce, W.W. (1970). "Managing the development of large software systems: Concepts and Techniques", Proc. WESCON, IEEE Computer Society Press, Los Alamitos, CA. Reprinted at the ICSE'87, Monterey, California, USA. March 30 – April 2, 1987.

[4] The Agile Manifesto, Snowbird, UT, February 11 – 13, 2001 http://agilemanifesto.org/

[5] NATO Factsheet 9 August 2000 http://www.nato.int/docu/facts/2000/nato-dci.htm

[6] A reader's Guide to the Prague Summit and NATO's Transformation.

# Chapter 2 – KEY CONCEPTS

This report is a report on **Evolutionary Software Development**[1]. This phrase can be interpreted in two ways, which both exist equally widespread in literature:

1) The **Evolutionary Development** of software, and
2) The development of **Evolutionary Software**.

The first interpretation considers evolutionary software development to be a **Software Development Process**, and a special instantiation of **Iterative Development**. Iterative development has been around successfully for a long time [2], but has been overshadowed for a long period of time by the **Waterfall Model**, which has a purely sequential character, although the original description of the waterfall model did include iteration [3].

In this report, Evolutionary Software Development is interpreted in the second way and is defined as the:

> *Development of (Systems and) Software, able to evolve with little effort to meet changing user needs, to interact with changing environments, and to benefit from emerging new technology.*

The crucial difference between these two interpretations is that the first interpretation regards only software development, resulting in a delivery of a final product for installation, albeit with intermediate partial releases. The second interpretation regards the entire **Software Lifecycle** and treats software as a living entity, whose life does not end when the "final" development product is released, but rather starts with the initial release and continues as long as the software is in use. And during that lifetime, the software will grow and evolve in response to changes in requirements, in technology, in hardware, and in other parts of the environment of the software product. Evolutionary Software Development according to the second interpretation distinguishes itself from other development models by already planning for these changes during the development of the initial product.

To phrase it differently:

> *In ESD there is no such thing as a "final" product, other than the state the product is in when it is taken out of operation at the end of its lifetime.*

Although the evolutionary development approach has been embraced as a good thing by most of the software engineering research community, warnings have been issued [4]:

> *The difficulty [with the evolutionary development model] is to distinguish it from code-and-fix models, whose spaghetti code and **lack of planning** were the initial motivation for the waterfall model. It is also based on the often-unrealistic assumption that the operational system will be flexible enough to accommodate unplanned evolution paths. This assumption is unjustified in three primary circumstances:*
>
> 1) *Circumstances in which several independently evolved applications must be closely integrated,*
> 2) *"Information-sclerosis" cases, in which temporary work-arounds for software deficiencies increasingly solidify into unchangeable requirements on evolution, and*
> 3) *Bridging situations, in which the new software is incrementally replacing a large existing system. If the existing system is poorly modularised, it is difficult to provide a good sequence of bridges between the old software and the expanding increments of the new software.*

[1] Key concepts are given in **Bold** when introduced for the first time in this section. Definitions of these terms are given in Annex A.

*Under such conditions, evolutionary development projects have come to grief by pursuing stages in the wrong order: evolving a lot of hard-to-change code* ***before addressing long-range architectural and usage considerations****.*

These warnings have not been ignored and large interest has grown in the area of **Software Architectures** to accommodate for software evolution over the lifetime of a product. A software architecture is the fundamental organisation of software embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution. An overview of the different attempts on architectures for evolution and the evolution of architectures themselves is given in Chapter 3.

[1] http://www.cs.dal.ca/ESD.

[2] Larman, C. and Basili, V.R. *Iterative and Incremental Development: A Brief History*, cover feature of IEEE Computer, June 2003.

[3] Royce, W. *Managing the Development of Large Software Systems*, Proceedings Westcon, IEEE CS Press, 1970, pp. 328-339.

[4] Boehm, B. *A Spiral Model of Software Development and Enhancement*, IEEE Computer, May 1988, pp. 61-72.

# Chapter 3 – SUPPORTING ARCHITECTURES

In this section, software architectures that support evolution are described. First of all, the role software architectures play in today's software development is discussed. Then, a number of different types of architectures addressing the management of change (evolution) are discussed. Finally, a section is devoted to the implications on architecture and evolution of using Commercial-Off-The-Shelf (COTS) components in software product development.

## 3.1 THE ROLE OF ARCHITECTURES

In [1], an overview of software architecture is given, including *"three fundamental reasons why software architecture is important, and architecture-based development is worthwhile:*

1) ***Mutual Communication**. Software architecture represents a common high-level abstraction of the system that most, if not all, of the system's stakeholders can use as a basis for creating mutual understanding, forming consensus, and communicating with each other.*
2) ***Early Design Decisions**. Software architecture represents the embodiment of the earliest set of design decisions about a system, and these early bindings carry weight far out of proportion to their individual gravity with respect to the system's remaining development, its service in deployment, and its maintenance life.*
3) ***Transferable Abstraction of a System**. Software architecture embodies a relatively small, intellectually graspable model for how the system is structured and how its components work together; this model is transferable across systems; in particular, it can be applied to other systems exhibiting similar requirements, and can promote large scale reuse."*

For discussion on support of architectures for evolutionary software development, the second reason is the most relevant one. More specifically, the early design decisions and the resulting architecture provide a basis for reasoning about, and management of, change. Deciding when changes are essential, determining which change paths have least risk, assessing the consequences of proposed changes, and arbitrating sequences and priorities for requested changes all require broad insight into relationships, dependencies, performance, and behavioural aspects of software components. Reasoning at an architecture level can provide the insight necessary to make decisions and plans related to change.

Furthermore, a system architecture provides the structure of the system in components and relations between components. Changes to the system may either affect only a single component (local change), or affect multiple components, or, in the worst case, affect the underlying structure between components. An architectural change that affects the way in which components interact with each other will probably require changes all over the system. Therefore, the architecture plays a crucial part in anticipating change by structuring the system in components in such a way that the most likely changes will be local to a single component.

The discussion above shows the importance of having a good architecture in place early in the development process. This has been widely recognised in the software engineering community and has led to the introduction of numerous "architecture first" development methods, usually referred to as architecture-based development methods. Architecture-based development starts with understanding the domain requirements [1]. Since a primary advantage of architectures is the insight they lend across whole families of systems, some extra effort should be spent on studying the requirements, not only for the current system, but also for the

whole family of systems the current system is or will be a part of. These families can be a set of related systems, all fielded simultaneously, but with small differences, or a single system that exists in many versions over time, each differing from each other by small changes. The domain analysis should therefore investigate the requirements as well as anticipate, enumerate and record changes, variations, and similarities.

In the remainder of this section, architectural principles and example architectures that specifically aim at reducing the impact of change will be discussed.

## 3.2 MODEL DRIVEN ARCHITECTURE DEVELOPMENT

A few years ago, the Object Management Group, known for the specification of the Common Object Request Broker Architecture CORBA, introduced the concept of Model Driven Architecture (MDA) [2]. MDA attempts to separate the logic of a system from the platform it is actually implemented on. MDA is organised around a Platform-Independent Model (PIM), which is a specification of the system in terms of domain concepts. These domain concepts are to a degree independent of different platforms of similar type (e.g., CORBA, .NET, and J2EE). The PIM can then be compiled towards any of those platforms by transforming the PIM to a Platform Specific Model (PSM). The PSM specifies how the system uses a particular type of platform.

This approach particularly addresses the evolution of the platform (hardware and operating system) and tries to minimise the impact of such changes on the overall software. Instead of having to re-design and re-implement the entire software, only the PSM and the transformation from the PIM to the new PSM have to be developed.

## 3.3 ADAPTABLE ARCHITECTURES

Adaptable is defined in many different ways, but the general consensus seems to be that a system is called adaptable if it can easily be changed. In order to ensure that the software system finally developed exhibits adaptability, this non-functional requirement should be considered during the development of the architecture.

In [3], numerous adaptation techniques are mentioned, categorised into the following: architecture-based techniques, component-based techniques, code-based techniques, genetic algorithm techniques, dynamic adaptation techniques, and adaptation methodologies.

These categories differ from each other in the phase of the software lifecycle (including operation and maintenance) in which the adaptation itself takes place. But they all agree on the fact that the structure to allow adaptation has to be designed, or at least considered, during the initial development of the software architecture.

A comprehensive adaptation technique in the architecture-based category is described in [4]. In this technique, an adaptable system has two managers – one for adaptation and one for evolution. In an iterative process, the adaptation manager takes high-level decisions, which are implemented by the evolution manager.

## 3.4 DYNAMIC ARCHITECTURES

Dynamic architectures [4,5] allow a system to adopt its behaviour at run-time, either by selecting alternatives within the system (constrained dynamism), or accepting new modules at run-time (unconstrained dynamism). The latter is often referred to as plug-and-play.

In case of constrained dynamism, all possible changes must be known a priori. By means of techniques such as parameterised instantiation of elements, conditional reconnection, and modification event handling, the architecture may be changed at run-time, but only in ways that have been designed into the architecture (hence the term "constrained" dynamism). The advantage of this approach is that system integrity is preserved because all run-time options may be tested before operation.

"Unconstrained" architectural dynamism allows "any" change in principle, but the validity of these changes must be ensured at run-time. Changes may include addition of components, removal of components, replacement of components, or re-configuration. Frequently used design patterns in these types of architectures are the observer pattern and the mediator pattern [6].

## 3.5 PRODUCT-LINE ARCHITECTURES

Development of software product lines [7,8] relies heavily on the use of variability to manage the differences between similar products (often referred to as a product family) by delaying design decisions to later stages in the development and usage of the constructed software. There are different variability realisation techniques depending on the binding time (e.g., architecture derivation, compilation, linking, and run-time) and the involved software entities (e.g., frameworks, components, classes, lines-of-code).

A taxonomy of variability realisation techniques is given in [9]. This taxonomy includes the following mechanisms:

- Inheritance: different or extended implementations of methods;
- Extension points: additional behaviour or functionality of components;
- Parameterisation: selection of component behaviour at build-time, or run-time;
- Configuration: selection or de-selection of components as-a-whole at build-time, or run-time; and
- Generation: components are generated from a higher-level language in which change may be expressed more easily.

An attempt to use the Model Driven Architecture from Section 3.2 as an approach towards variability management is described in [10].

The research into product-line architectures and the management of change is ongoing and new directions still arise. But, in general, the trend in variability seems to be towards two directions:

1) Addressing variability more and more in software instead of hardware, and
2) Preparing for variation earlier in the development process to postpone the selection of a single variant as long as possible into run-time to allow the user more flexibility.

## 3.6 COMMERCIAL-OFF-THE-SHELF SOFTWARE AND ARCHITECTURE: FRIEND OR FOE

Today, many operational systems and systems under construction contain a large number of Commercial-Off-The-Shelf (COTS) software components. Reasons for using COTS are mainly [11]:

- Cost savings (development and maintenance costs spread over many customers),

- Better products (more feedback),
- Lower training costs, and
- Faster integration of new technologies and new standards.

However, the use of COTS is not without risk. In the same report [11] as well as in [12], a long list of risks that come with COTS is given:

- **Loss of Control:** COTS tools are marketed by a vendor according to the schedule of the vendor, and may be discontinued at any time by the vendor. The vendor may even go out of business. License agreements may change over time, and costs for maintenance and upgrades may increase unanticipated.
- **Lack of Understanding:** Developers do not have access to source code, complete and correct behavioural specifications may not be available, and analysis and testing must be done in a black box manner. COTS products may not meet non-functional requirements such as availability, integrity, and security. This deficiency of information increases the possibility of introducing design errors.
- **Frequent Updates:** Commercial products are continuously being upgraded and customers are often required to upgrade in order to fix bugs, or in order to keep receiving maintenance and support. Replacing a component may be time-consuming because of regression testing and development of new workarounds, changes in integration software, or necessary updates of hardware.
- COTS components may not operate well together, because of lacking conformance to standard interfaces and exchange formats.
- **Troubleshooting:** When the system fails, the component that causes the failure must be determined. As mentioned before, COTS are usually black boxes and determining which component causes a failure may therefore be difficult. Furthermore, the exact circumstances under which a component fails must be determined to provide to the support organisation and the priority given to this problem by the vendor may be low.
- **COTS Products Come with their own Architectures:** These may not match the intended architecture of the system as a whole. Therefore, the COTS product may dictate some aspects of the system architecture and limit alternatives, or the system architecture may limit the choice of COTS products you can consider.

Because of these potential risks, several researchers have proposed extra development activities and architectural solutions to mitigate these risks [11,13]:

- Experience with (combinations of) COTS products early to gain better understanding;
- Evaluate alternative architectures to identify weaknesses or deficiencies in COTS products;
- **Architecture Migration:** Design an architecture in such a way that a transition to an architecture without a certain COTS product, or with a different (version of the same) COTS product, is easy; and
- Evaluate the proposed architecture against the characteristics of evolution to identify the ability of the architecture to change. In [14], such characteristics, called Evolutionary Characteristics Of Architecture (ECOA), are described.

Most of these solutions try to embed the COTS in the architecture in such a way that changes (upgrading a COTS component, or replacing a COTS component by another) can be made more easily. These solutions

may also be used for integrating in-house developed components. Because of the emphasis of these solutions on managing change, they are also useful to be considered for use in architectures for evolving systems, even if they do not use COTS components.

A discussion of COTS cannot be closed without referring to the Open Source movement. In recent years, open source software has become popular as a basis for developing products.

Although the use of open source software mitigates the first two risks mentioned at the beginning of this section, the other risks still apply. The proposed solutions in this paragraph therefore also apply to product development based on open source components.

## 3.7 REFERENCES

[1] Clemens, P.C. and Northrop, L.M. *Software Architecture: An Executive Overview*, Technical Report of the Software Engineering Institute, CMU/SEI-96-TR-003, February 1996.

[2] MDA Guide v1.0, OMG, 2003 (http://www.omg.org).

[3] Subramanian, N. and Chung, L. *Software Architecture Adaptability: An NFR Approach*, Proceedings of the International Workshop on Principles of Software Evolution (IWPSE 2001), ACM Press, Vienna, September, 2001, ACM Press, pp. 52-61.

[4] Oreizy, P., et al., *An Architecture-Based Approach to Self-Adaptive Software*, IEEE Intelligent Systems, May/June 1999, pp. 54-62.

[5] Oreizy, P. Dynamic Software Architecture Resources Web pages, http://www.ics.uci.edu/~peymano/dynamci-arch/.

[6] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns*, Addison-Wesley, 1995.

[7] Lai, C.T.R. and Weiss, D.M. *Software Product-Line Engineering: A Family Based Software Development Process*, Addison-Wesley, 1999.

[8] Clements, P. and Northrop, L. *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2001.

[9] Svahnberg, M., van Gurp, J. and Bosch, J. *A Taxonomy of variability realization techniques*, technical paper ISSN: 1103-1581, Blekinge Institute of Technology, Sweden 2002.

[10] Deelstra, S., Sinnema, M., van Gurp, J. and Bosch, J. *Model Driven Architecture as Approach to Manage Variability in Software Product Families*.

[11] Vigder, M. *An Architecture for COTS Based Software Systems*, NRC Report No. 41603, National Research Council of Canada, 1998.

[12] Davis, R. *The Role of Architecture in Managing COTS Based High Integrity Systems*, Presentation from the Ground Systems Architecture Workshop, March 2003, http://sunset.usc.edu/gsaw/gsaw2003/s8b/davis.pdf.

[13] Carnegie Mellon Software Engineering Institute Architecture Lessons: http://www.sei.cmu.edu/cbs/architecture/lessons.htm.

[14] Davis, L., Payton, J. and Gamble, R. *Toward Identifying the Impact of COTS Evolution on Integrated Systems*, Paper of the COTS Workshop on Continuing Collaborations for Successful COTS Development, Held in conjunction with ICSE 2000, Limerick, Ireland, June 4 – 5, 2000, http://wwwsel.iit.nrc.ca/projects/cots/icse2000wkshp/.

# Chapter 4 – STATE OF THE ART / PRACTICE

## 4.1 STATE OF THE ART

Symposium IST-034 RSY-010 Technology for Evolutionary Software Development was held in Bonn, 23 – 24 September 2002. Key issues from the papers in this symposium were:

- The concept of Evolutionary Software Development is not well understood, perhaps not even agreed upon. Many papers in the Symposium described practices that do not meet the definition used here.
- Experience reporting is not systematic or consistent, making comparison difficult.
- There is a lack of training in Evolutionary Software Development.
- There is no NATO standard for Evolutionary Software Development.

### 4.1.1 Other Workshops and Conferences

There have been many other workshops and conferences where the subject of software evolution and how to cope with it has been discussed. Two of the most important workshops held annually since 1998 and 2002 respectively have been IWPSE (International Workshop on Principles of Software Evolution) and USE (International Workshop Unanticipated Software Evolution). Although papers at these workshops do not always refer to Evolutionary Software Development, the examples and theories that they discuss do lead to an increased understanding of software evolution, and the problems that Evolutionary Software Development is trying to address. Mention of this topic would be incomplete without reference to Lehman and Belady's Laws of Software Evolution that were originally put forward in the 1970's, and updated in Project FEAST in the 1990's. These laws, especially the second law, are still highly controversial.

### 4.1.2 Tom Gilb

One of the original advocates of Evolutionary Software Development is the internationally known consultant from Norway, Tom Gilb. He served as the technical rapporteur for Symposium IST-034 RSY-010, he put on a pro bono course on evolutionary development for the IST Panel, and a CD-ROM of material collected by him on evolutionary development was distributed to every member of the IST Panel. His website www.gilb.com is an important source of tutorial and reference material, as well as examples.

### 4.1.3 Evolutionary Procurement

Without the acquiescence, indeed more than that, the support and encouragement by procurement practices, Evolutionary Software Development would only be a theory. These practices have been called Evolutionary Acquisition or Evolutionary Procurement. Evolutionary procurement implies Evolutionary Software Development. There are many obvious challenges for evolutionary procurement compared to traditional procurement processes. Competitive bidding to select a supplier or choose options requires different processes when the requirements are known only in broad outline and the ability to respond to changes must be an essential criterion. Contracting a project in separate phases each subject to competitive bidding is an option, but delays in approval and risk of loss of continuity of staff are countervailing arguments. How to gate progress payments in the absence of conventional milestone reviews requires new metrics of progress. Testing and evaluation no longer is an isolated end-of-project, perhaps even post-delivery event but must become an

integral activity of each release cycle. Early delivery of partial functionality is only of value if the user organization actually exploits it by at least subjecting it to operational testing and preferably fielding that increment. To avoid the increment remaining shelfware implies a significant commitment by the user organization and even individual users. An excellent discussion of these and other issues is contained in the report DSTO-TR-0481 http://www.dsto.defence.gov.au/publications/2095/ of the Australian Defence Organization, published in 1997 by Derek E. Henderson and Andrew P. Gabb [1].

Evolutionary procurement is the preferred practice for procurement of software intensive systems by NC3A http://www.afcea.org/signal/articles/anmviewer.asp?a=720&z=7. To promote the practice and to advance the technology, NC3A sponsored the conference Evolutionary Procurement of Information Systems (EPIS '90) in 1990, followed by EPIS 2000 in 2001. Several noteworthy NATO systems, including NATO ICC (Integrated Command and Control), Cronos (eventually to become ACCIS, Automated Command and Control Information System) and ADAMS (Allied Deployment and Movement System), were acquired by evolutionary procurement, The US DoD has also endorsed Evolutionary Acquisition http://www.dtic.mil/whs/directives/corres/html/50002.htm. Progressive Acquisition is a further refinement of Evolutionary Acquisition endorsed by WEAG (Western European Armaments Group) TA (Technical Area) 13 http://sesam.tranet.fmv.se/dokumentation/rapporter/rapporterna/FD2-1.doc.

## 4.2 CONCRETE EXAMPLES TAKEN FROM SYMPOSIUM IST-034 RSY-010

Two of the papers from Symposium IST-034 RSY-010 discussed the application of Evolutionary Software development to concrete examples of systems. These examples help understanding of the role that evolution can play in software development and throughout the lifetime of software.

François B. J. de Laender, in *Toward an Evolutionary Strategy of Developing a Software Product Line in the Field of Airport Support Systems*, describes the effort NLR is making to bring together all of its civil airport support systems into a software product line. A software product line is a set of software products that share a managed collection of resources. The shared resources, called ASAP (Airport Scenario Analysis Platform), are currently under development. Existing products must evolve to use these shared resources instead of, as currently, each implementing things in its own unique way. New products will use the shared resources ab initio. However it must be expected that the shared resources will themselves evolve, both in that the design and implementation of a particular resource will change over time, and in that the set of resources to be shared may be augmented by additional resources or may be reduced by resources being dropped.

Capt Roberto Ing. Ambra and Ing. Fabio Ruta, in *The Evolutionary Software Development Process used in the Upgraded AMX Human Machine Interface Design*, describe the process by which the Human Computer Interface in the cockpit of the AMX aircraft was evolved to accommodate new weapons and navigational systems. This is particularly interesting because a predictive elicitation of requirements by itself is insufficient for Human-Computer Interfaces. The requirements must b validated by experimental evaluation with pilots actually working with proposed new interfaces. In all, thirteen prototypes were needed to explore the different aspects of the interface that were changed. Concurrent engineering was critical to developing the new interface on an acceptable schedule. The project faced both management and contractual challenges in using this unfamiliar process.

## 4.3 RELATED CONCEPTS

### 4.3.1 Prototyping

Prototyping is commonly used in engineering design as a way of answering specific design questions, by producing a working system model that can be tried and studied. These questions may be, as in the example above, about requirements. They may instead be questions about implementation alternatives. They may also be operational questions about the system once deployed. Sometimes the questions can be answered economically by mock-ups or other technology unrelated to the production system. On the other hand, sometimes the economic way to answer the questions is by creating a modification of an existing production system. Often, once the questions have been answered, the prototype is a throw-away. On the other hand, sometimes the prototype needs to be preserved, in case similar questions arise in the future. In many situations, the investment in creating the prototype is substantial, and it makes sense to consider evolving the prototype into the production system.

Thus prototyping is not necessarily associated with evolutionary software development, but it can be.

### 4.3.2 User-Centered Design

From the Wikipedia (free on-line encyclopedia):

> *"In broad terms, user-centered design (UCD) is a design philosophy and a process in which the needs, wants, and limitations of the end user of an interface or document are given extensive attention at each stage of the design process. User-centered design can be characterized as a multi-stage problem solving process that not only requires designers to analyze and foresee how users are likely to use an interface, but to test the validity of their assumptions with regards to user behaviour in real world tests with actual users. Such testing is necessary as it is often very difficult for the designers of an interface to understand intuitively what a first-time user of their design experiences, and what each user's learning curve may look like.*
>
> *The chief difference from other interface design philosophies is that user-centered design tries to bend and structure the functioning of a user interface around how people can, want or need to work, rather than the opposite way around."*

Thus user-centered design is typically iterative, and addresses both functionality and the way that functionality is experienced by the end-user through the human-computer interface. It typically evolves as each issue is resolved. However, it can differ from evolutionary software development in that it focuses on the design stage of a product, and as such may not address the entire life of that product once deployed.

### 4.3.3 Agile Software Development

Agile software development is an iterative approach to software development that attempts to minimize risk by using very short duration timeboxes for each iteration, typically one to four weeks. Close interaction with the customer during or between iteration is key.

The Agile Manifesto cites:

> We emphasize:
>
> Individuals and interactions over processes and tools
>
> Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

Each of the various different agile methodologies incorporates additional practices into its methodology, but the philosophy is a common thread through all. Experience to date has been that any agile methodology can be very effective for software development, especially fir small projects.

Agile methodologies only focus on initial development, and not on the whole life cycle of the software product, and so differ from evolutionary software development.

### 4.3.4 Achieving Specific Non-Functional Requirement Targets

A key difference between functional and non-functional requirements is that whereas functional requirements can often be proved to have been met by appropriate design, for typical non-functional requirements (such as performance, availability, scalability, or interoperability) with specific targets, design can only be shown to have intended to meet the criteria. For such non-functional requirements, the real satisfaction of criteria must be demonstrated after the system is in operation. This often leads to iteration, where in system test the targets are found not to be met, and tuning, adjustments, and even re-implementation of critical components must be repeatedly tried until the targets are achieved. During operation of the fielded system, changing operational conditions may well upset the balance so carefully reached, so that if the specific target is of ongoing importance, the system may periodically require retuning. A concrete example of this situation was a post-mission sonar reprocessing system, which had a non-functional requirement that in order to keep up with presented load, it to operate at three times real time. That capability could only be ascertained once the system was deployed, and over time, changes in sonar processing algorithms and changes in behaviour of sonar targets required retuning of system parameters.

Once again, this does result in iterative development, but it is not the planned and scheduled succession of releases that constitute evolutionary software development.

## 4.4 REFERENCES

[1] Henderson, D.E. and Gabb, A.P. (1997). “Using Evolutionary Acquisition for the Procurement of Complex Systems”, Australian Defence Organization, Report DSTO-TR-0481 http://www.dsto.defence.gov.au/publications/2095/.

# Chapter 5 – ISSUES AND CONCERNS (FAQ)

## 5.1 QUESTIONS AND ANSWERS

### 5.1.1 Why do Developers and Customers Use the Evolutionary Process? (Benefits)

#### 5.1.1.1 Partial Functionality is Available Early

The most common attraction to customers of ESD is that early releases of the software are fully deployable, albeit with only a subset of the ultimate functionality. Partial functionality is usually more useful than none at all. The customers can thus get on with at least part of their work, often months earlier than they would be able to if they had to wait for completion of the full functionality. One typical use of partial functionality is training, not just for the operators of the system, but also for others whose activities are affected by the system. Another important use of a system with partial functionality is where it will be used with other existent systems in a system of systems. Use of that system of systems is often inhibited until the existent systems have at least something to interoperate with that can take on the role that the system under construction is intended to fulfill. Developers are attracted to ESD when it leads to customer satisfaction.

#### 5.1.1.2 Partial Functionality Can be Achieved Before Complete Funding

Both in the commercial world and in the military world, funding for the complete system is often not available immediately when the need, and the scope, for the complete system are identified. If meaningful increments of partial functionality can be identified that deliver significant value, and can be funded sequentially, then the system can be built over time as funds become available. The successful deployment of the previous increments often assists in raising the funding to develop successive increments.

#### 5.1.1.3 Learn-From-Use Feedback Corrects Requirements

A different attraction of ESD arises from the recognition that initial requirements documents are often flawed, and consequently that specifications derived from them are therefore also flawed. This is particularly true for novel applications, where due to lack of experience with the application, detailed requirements follow from analysis of hypothetical situations that may not turn out to be what happens in practice. Users typically are much better at recognizing what is wrong with a system that they can work with than they are at predicting whether a system described in a design document will satisfactorily meet their needs. Consequently, producing a system with partial functionality and then trying to use it in a production environment can be the most effective way to refine requirements documents to identify the system that should have been built, particularly when the system is then revised in keeping with this increased understanding. Both the customers and the developers are happier when the customers are satisfied. In the long run, even the cost of rework to accommodate improved understanding of requirements is more than compensated for by the better acceptance of a system that meets the right requirements, and the greater effectiveness in its use.

### 5.1.2 Why Don’t Developers and Customers Use the Evolutionary Process?

#### 5.1.2.1 Lack of Familiarity with ESD

The most common explanation for ESD not being used is that customers and developers are often unfamiliar with the process. This situation is made worse by the fact that popular software engineering and project

management textbooks and courses still rarely mention iterative development processes, never mind drawing the distinctions among them. A particular concern is that because the process is apparently novel, it may impose risks in the management of the development activity. Questions such as how to measure progress do not have self-evident answers.

#### 5.1.2.2 Concern that ESD is Not Permitted

Many customers believe that the procurement policies of their organization do not permit iterative development. Many developers of software under contract believe that the contract forbids iterative development processes. While this may be true, frequently it is not. MIL-STD-498 was explicitly introduced to encourage iterative development; ISO/IEC-12207 was carefully worded not to prohibit iterative development. Yet both standards were widely interpreted as requiring waterfall development, as MIL-STD-2167A had been. However some developers had even found it possible to do iterative development under 2167A.

#### 5.1.2.3 Lack of Tool Support for ESD

Modern software development relies intensively on interactive development environments (IDEs), configuration management systems and other tools. Project management has relied for decades on Gantt charts, PERT charts, CPM, Earned Value Analysis and similar tools. These tools are often thought to be incompatible with ESD, and tools explicitly to support ESD are rare.

### 5.1.3 How Can ESD be Managed?

Between customers and developers, prioritizing features to incorporate in the next few releases rarely appears difficult. Concurrent engineering practices, such as workspace based configuration management and nightly build ensure that development proceeds in a way such that there is always a version of the software ready to ship: fixed release dates or funding constraints on releases can always be met by slipping functionality still in progress to subsequent releases. Including documentation. installation aids and other ancillary artifacts under configuration management ensures they match the software in the release. Unrolling the development iteration for the next few releases so that the work items in each appear explicitly in the Work Breakdown structure (WBS) means standard project management tools can be used in almost conventional ways.

### 5.1.4 How Can ESD be Taken into Account in Current Processes?

The obvious answer for most processes is to consider interim releases before the current final delivery, or subsequent releases upgrading the product after the current planned final delivery. The former might provide increased value within the current funding envelope; the latter might provide a way of generating value after the completion of the current funded activity. If the current process is already an iterative one, such as Agile methodology, then taking EDS into account might involve carrying occasional iterations through to operational use, not merely demonstration.

### 5.1.5 Cost Estimation of ESD

Formalized cost estimation methods, such as COCOMO or function points, estimate cost using predictors that are functions of sizing parameters of the project with coefficients that are statistically determined form experience with similar projects. The same kinds of predictors are equally appropriate for ESD, although similar projects being different, the coefficients and the accuracy of prediction can be expected to be different.

The more serious challenge is that estimation of per release cost corresponds to upgrade cost estimation, which traditional cost estimation does not do well. It is unclear what are appropriate factors to consider, or what constitute similar projects. In practice this means that cost estimates will likely be based on judgment and experience, as they usually are.

### 5.1.6 QA (Quality Assurance) and ESD

Regression testing is essential with ESD, as it is with any process that leads to many releases of a product. The efficacy of ensuring changes in this release did not impact things that should not have been affected is less critical than avoiding the embarrassment of reintroducing bugs already solved. To the extent that the customer perception of quality includes responsiveness of the supplier to newly recognized needs, software with an ESD life cycle can be seen as higher quality than software upgraded through a more cumbersome process.

### 5.1.7 ESD and Business Decisions

ESD permits early fielding of partial functionality. This facilitates revenue generation before the product is complete, creates a presence in the market before the product is complete, and can even be crucial in attracting investor as well as customer support in a speculative product. The machinery for adapting to changing requirements provides a vehicle for responding to pressure from competitors with somewhat different feature sets.

### 5.1.8 Relationship of ESD to Open Source Movement

ESD is as relevant to the Open Source movement as it is to more conventional contract software development. In fact it may even be more relevant, because an Open Source project must attract and maintain a volunteer group of developers, and the success of early releases can be a large factor in sustaining and growing commitment.

### 5.1.9 Relationship of ESD to the Agile Movement

ESD and Agile development are both iterative processes, and sometimes might be synonymous. The first difference is that Agile development iterations may only involve demonstrating the product to the customer, whereas ESD iterations involve deploying the product in production. The second difference, not unrelated to the first, is that whereas Agile iterations occur every week or two, ESD iterations might occur only every several months. Finally, Agile development may only involve iterations until the product is delivered, whereas ESD iterations continue until the product is retired.

## 5.2 CONCERNS

### 5.2.1 Evolution is Risky

The whole point of evolution is that it facilitates change as the driving factors change over time. What that means, of course, is that at project initiation you can't predict the form of the product many cycles out. Some people see that as a concern – some people see it as a strength. It depends how wisely the evolution is guided.

### 5.2.2 Evolutionary Process Facilitates Decreasing Risk

One principle that can be used for steering evolution across individual releases or across multiple releases is that the changes implemented should reduce risk. This evaluates the evolution in terms of how well the risks were understood and appreciated, and how wisely the judgments were made and implemented.

# Chapter 6 – UNRESOLVED CHALLENGES (FUTURE RESEARCH, POSSIBLE IMPROVEMENTS)

## 6.1 SOFTWARE ARCHITECTURE

Software architecture, the structure of a software-intensive system, plays a huge role in how amenable the system is to evolutionary development. This software structure comprises the software elements, the externally visible properties of those elements, and the relationships among them. Every software system has an architecture, whether or not it was explicitly designed. Software architecture has been recognized as a discipline for about 15 years and many aspects have been studied.

Nevertheless, little work has been done yet on understanding what architectural aspects assist or hinder evolution, and hence how to choose an architecture for a system which is known to need to support evolution. Indeed, little work has yet been done in understanding how to use specific architectural features to facilitate evolutionary development. Future research is needed to resolve this. Some particular such features are the following:

### 6.1.1 Dynamic and Adaptive Architectures

Dynamic architectures are architectures that change over time. Not only are changes of which components are connected to which supported, but the nature of the components can change, and so can the nature of the connectors. The study of a dynamic architecture includes the details of how a transition takes place, as will as an understanding of when different transitions are permitted. Evolution of a software architecture can often be modeled with a dynamic architecture.

Adaptive architectures are a different way of addressing a somewhat similar situation. Rather than making the changes in the architecture explicit, an adaptive architecture hides the changes at an architectural level by defining components and connectors in such a way that changes affect the nature or usage of each component and connector, but not the connections that are established between components. The individual components and connectors are said to “adapt” to evolving circumstances, and the way that adaptation can take place is studied. As an example, rather than defining point-to-point connectors between components, an adaptive architecture might connect all the components with a bus, and then examine the traffic on that bus changes with demand. Evolution of a software architecture can therefore also sometimes be modeled with an adaptive architecture.

Examples of dynamic architectures and of adaptive architectures have appeared in the literature, illustrating how they can be used. There is a need for more study of real systems undergoing typical evolution, framing these systems as dynamic or adaptive architectures to see how these formulations can assist in analysis, understanding and design of systems that evolve.

### 6.1.2 Product Lines

One of the hot topics in software architecture in recent years has been the study of product lines. A product line is defined as a set of products that share a managed set of common resources. The study of product lines includes not just how to best exploit the commonalities, but how to manage the common resources. Often a product line is thought of as a set of products that exist at the same time, but it can also be useful to think of versions of a product that exist over time as forming a product line, with the shared resources being the

software components, documentation, training material etc. that are re-used from one version of the product to the next. Although this perspective has been noted, there is as yet no literature illustrating how this observation should be used to manage the re-use activities that make up a labor-intensive part of the evolutionary software development process.

Research into thinking of product evolution as a product line would be particularly natural in the case of products that are highly configurable, that is, that already exist in many versions simultaneously. A material management system that needs to be configured differently for each base or ship on which it is used, for instance, might be a naturally subject of study for how the configuration mechanism could be used to accommodate upgrades to respond to changes in operational procedures.

### 6.1.3 Service Oriented Architectures

One very popular software architecture style stimulated by the worldwide web has been Service Oriented Architectures. Such architectures decouple the usual association between a component requiring some service and the component providing that service by delaying the binding until run-time, and by doing the binding with any server on any computer that can provide a suitable service. This fits well with the commercial realities of the web, where there are many suppliers competing to provide essentially equivalent services, and where the choice of which service to use depends on price, convenience, responsiveness, relationship sustainability, and numerous other business factors. Similar factors can occasionally affect relationships for support of military services. Coalition operations, for instance, involve interoperability issues whereby on different occasions different partner nations provide essential services, and the integration of complex systems of systems from independently developed and independently operated subsystems is a critical necessity. What is more germane, however, is that even within systems fielded by a single nation, the effect of evolution is that components must be replaced or upgraded, and that changing requirements and expectations require connections amongst components to change. Service oriented architectures provide a standard way to think about this, as well as standard mechanisms to accomplish it.

Experience with service-oriented architectures on the web is growing day by day. Research is needed, however, to see how to exploit this experience to see how to use it to build better military systems that can be more responsive to change.

## 6.2 IMPLICATIONS FOR PROCUREMENT

The issues for future research discussed above are essentially technical issues addressing how to carry out the process of evolutionary software development. However, as pointed out in earlier sections, for military systems evolutionary software development can only be practiced when permitted by the procurement process. In many nations, the procurement process appears to actively discourage taking into consideration the entire life cycle of a system, or even just a large multi-year portion of it. Often this is an unintended consequence of other policies, policies that in themselves seem reasonable and even desirable. Limited budget horizon, competitive bidding for successive contract phases, no rolling contracts, no bridging funding between contracts among others are all procurement policies that have some justification, but make it difficult for a contractor to sustain a team to work on the product over many releases, let alone to plan several releases ahead.

Despite this, project officers working with procurement agencies in many nations have found ways to conduct projects with evolutionary software development. More experience with novel approaches to procurement and

contracting needs to be shared, both to promote successes with the process, and to suggest approaches that others might try.

## 6.3 COST ESTIMATION

Cost estimation is necessary for developers bidding on RFPs, for project proponents attempting to secure budgets, for project managers monitoring burn rates, and others. Unfortunately, despite attempts over the years to automate cost estimation via tools such as COCOMO or function points, it remains true that the most reliable cost estimation is still the judgment of experienced managers. This is particularly true for projects that are experimental, novel, and where expectations are evolving rapidly – exactly the kind of projects for which evolutionary software development is best suited. Despite the disappointing progress in cost estimation in the past, renewed efforts to develop tools for cost estimation are needed. Iterative processes would seem to be particularly difficult to estimate, because the number of iterations is often unknown. However, when project pricing is on a fixed price basis, instead of time and materials or cost plus, project goals are continually adjusted to stay within time and money budgets, which makes the overall project cost much more predictable.

## 6.4 PROJECT MANAGEMENT, INCLUDING PRODUCT AND PROJECT METRICS

Standard project management tools, from CPM, PERT Earned Value Analysis and Gantt Charts through to common software metrics and their analysis, are not designed to work with iterative processes such as evolutionary software development. Unorthodox usage such as unrolling the next few iterations, can provide workarounds to overcome some of the shortcomings, but other deficiencies, such as support for moving work breakdown structure activities between releases are still poorly served. Some of the Agile methodologies have introduced new tools, such as burndown charts, to support their needs in project management. There is great opportunity for creativity in this area.

## 6.5 TOOL SUPPORT, INCLUDING CONFIGURATION MANAGEMENT

Practical software development and support today depends heavily on tools, from Interactive Development Environments (IDE) and Configuration Management Systems to Program Understanding and Reverse Engineering tools. As evolutionary software development continues into later releases, it is increasingly important that new work integrates well with what is already there, and as corporate and individual memory fades, this is only possible with tool support. A well structured repository of source code and other development artifacts can simplify this, but it is not unusual for evolution to proceed in unanticipated direction, necessitating refactoring of repository and artifacts. Tools to support this have typically been produced from a re-engineering perspective, but the evolutionary software development need is somewhat different, as in that case several releases are often under development concurrently. Advances in tools would be very beneficial.

## 6.6 HOW TO ECONOMIZE IN TESTING

The traditional approach to testing new releases of software is to apply two different kinds of tests: functional tests to confirm that the new functionality is working as intended, and regression tests to confirm that things that were not supposed to change did not. This is fine as far as it goes. However, there are many aspects of the

software system that it does not test, from security to performance to robustness. Furthermore, for the testing it does do, is that testing done effectively in terms of time, cost and effort? Is that testing being done effectively in terms of sensitivity to what it is intending to detect? Current industry practice leaves great opportunities for improvement, but even theoretical approaches have many unexplored gaps.

## 6.7 DOCUMENTATION

Classic paper documentation, especially MIL-STD-2167A style documentation, is notorious for being incorrect and obsolete – and cumbersome and expensive to update. In a world that is actively intending to support evolution, it is essential to do better. Web-based documentation, subject to configuration management and exploiting hyperlinks and multimedia, increasingly is being used as a vehicle to facilitate better documentation. Increased reliance on search engines has reduced the enormous overhead of restructuring documentation as usage and expectations change. Keeping documentation current and relevant to all stakeholders is still a Herculean task, and one for which there are few exemplars, much less guidelines and tools to assist documenters.

## 6.8 AIDS TO RETRAIN USERS

Related to the last point about documentation is that there are really two purposes for documentation: for training novices and for reference by old-timers. The challenge for old-timers with a rapidly evolving system is that what they know and don't think they need to look up may no longer be correct. The challenge of how to retrain users without insulting them or wasting their time is not insignificant. Today most people learn to use software systems not from manuals but from experimenting and on-line help. Correspondingly, they re-learn changed software systems best not from update documents, but from on-line wizards that watch what they do, and draw their attention to actions indicating that they have yet to assimilate some procedural revision. Excellent examples of this kind of re-education exist, but there is as yet little assistance for someone responsible for producing such wizards.

## 6.9 SCAFFOLDING

The delivered software system is only part of the software built in constructing that deliverable. Additional software, specific to that product, has had to be built in order to generate parts of it, prepare data for it, configure it, migrate existing repositories to be compatible with it, compare alternatives, install it and carry out many other tasks that affect, but are not part of, the deliverable. This supporting software is called scaffolding. Some of these tasks are not simply one-time efforts, but recur at least for every release of the software. With luck, the same scaffolding can be re-used each time it is needed. It is not uncommon, on the other hand, to find that as the product itself evolves, there must be corresponding evolution in some of the scaffolding. Today that secondary evolution is normally manual. It would be preferable by far if we knew how to generate both the change in the system and the change in the scaffolding automatically from the same change action.

## 6.10 COTS-BASED SYSTEMS

Today very few systems are built from a blank piece of paper: most systems employ pre-existing components. These pre-existing components form a platform on which the system is built, a platform whose characteristics can permeate the resulting product. When the components in this platform are obtained from a commercial off

the shelf (COTS) source, an important characteristic is that the COTS component has a life of its own. The military customer is rarely a significant part of the COTS components supplier's marketplace. The military customer must thus accept fixes and enhancements if and when the commercial marketplace motivates the supplier to make them. The military customer must also accommodate new features and revisions entirely unrelated to what is needed for the component in his system. Postponing adoption of the updated component is only sometimes possible, and then not always a wise choice. The ultimate such revision is when the supplier discontinues support for the component, and the system can only continue to operate if the component is replaced in the platform by a different nearly equivalent one.

This constant, uncontrolled revision of the platform is familiar to all computer users through the impact of major components such as operating systems, compilers, databases, networking protocols, email systems, etc. Unfortunately, it occurs with all third party components whatever scale, whether commercial or Open Source. Portability at one time was often thought of as being an issue of adapting to different hardware vendors. Today this constantly changing software platform is a much bigger issue. The "write once run anywhere" slogan of Java, purportedly achieved by standardization, does not deal with this problem, and to build long-lived systems, the domain of evolutionary software development, will require new technology, or perhaps a return to some of the older technologies for coping with portability challenges that have recently fallen out of fashion.

## 6.11 INTEROPERABILITY

Military systems today can no longer be traditional stovepipes – to be useful they need to interoperate with many other systems. This is particularly a challenge because regardless of any evolution that the system itself may need to accommodate its changing requirements, it needs to change to accommodate the continuing churn of change in the systems it interoperates with. Some examples of military systems cite that as hundreds of systems, each changing in its own way on its own time schedule. Just tracking the changes is a horrendous task. Experience even trying just to synchronize the updates has shown the hopelessness of such a coordination approach.

Fortunately, we have before us the experience with the commercial Internet, which while not perfect has shown that it can be viable to interoperate between systems operated by independent uncoordinated organizations. The secret is a service-oriented architecture based on flexible delayed binding protocols such as SOAP. Communicating objects self-identify and negotiate interface requirements via XML descriptions. If a communicating partner changes, the interface must be re-negotiated. With numerous success stories evident in e-commerce on the web, and Microsoft .Net demonstrating that such an approach works in corporations, we need more examples of Net-centric military systems taking advantage of this approach.

## 6.12 SIMULTANEOUS FIELD SUPPORT FOR MULTIPLE RELEASES

One of the critical consequences of the evolutionary development process is that several releases of software are fielded in a short time. A consequence of this is that unless all users of the system can be forced to upgrade simultaneously, at any time there will be multiple releases of the software operating in the field. There are many valid reasons for individual sites to postpone upgrades, so a plethora of versions of the software running at any time is the normal situation. At the very least this means that communications between instances of the software, or communication between interoperating systems and instances of this software, need to be tagged as to what version of the software is involved, and inconsistencies need to be negotiated. The complications go well beyond this initial step. From exception reporting to user interactions to configuration, any thing that

can be different between versions needs to be handled. Masking all differences is an illusory goal – after all, the software evolved exactly in order to realize those differences. We need examples of how this can be done and still leave operations to be manageable.

## 6.13 AUTOMATED UPGRADING

In recent years one approach some suppliers have taken to minimizing the disparity between systems found in the field is to automatically update systems that are permanently on-line. Microsoft and Apple are among the suppliers who do this for their operating systems, but many other vendors do it for any other products they supply. Applying updates automatically can reduce the investment in deploying updates for the supplier, and can reduce the cost of performing upgrades for the system operator, but it is fraught with risks and discouraged by many users, Ways need to be found to reduce the risks and increase the enticement of this potentially valuable practice.

# Chapter 7 – RECOMMENDATIONS/CONCLUSIONS

Task Group IST-026/RTG-008 has reached two conclusions to summarize its work.

## 7.1 ESD SHOULD BE MORE WIDELY ADOPTED FOR LONG-LIVED SYSTEMS

Evolutionary Software Development is a successful technique to solve a very real need shared by many military systems, and its widespread adoption should be encouraged.

## 7.2 THE TG-008 WEB SITE SHOULD BE CONTINUED AS AN ACTIVE VEHICLE FOR INFORMATION INTERCHANGE

Many stakeholders in the development of military systems are currently unfamiliar with the Evolutionary Development process. This often includes the technical development staff themselves. Moreover, there is and will continue to be progress made in improving the practice. Consequently, it would be a valuable contribution to have a single web site that would be regarded as a source of up-to-date information about Evolutionary software Development. For this reason, the task group web site will be restructured for improved information interchange on the topic, and maintained indefinitely.

# Annex A – DEFINITIONS OF KEY CONCEPTS

In this Annex, definitions of key concepts used in Chapter 2 are given.

**Software Development Process –** The process by which user needs are translated into a software product. The process involves translating user needs into software requirements, transforming the software requirements into design, implementing the design in code, testing the code, and sometimes, installing and checking out the software for operational use.

**Software Lifecycle Process –** The process governing the entire lifecycle of a software product, from development of an initial release through maintenance and subsequent releases to out-of-service. Includes **Software Development Process**.

**Sequential Development –** A general term for **software development processes** in which requirements definition, design, implementation, testing, and installation phases are executed in a sequential order, usually with an official review and approval of the products of one phase before moving into the next, and with little or no overlap between phases, and little or no iteration over phase boundaries. *Contrast with*: **Iterative Development**.

**Iterative Development –** A general term for **software development processes** in which the requirements definition, design, implementation, and testing occur in an overlapping, iterative (rather than sequential) manner, with user feedback at the end of each iteration, providing input for the next. *Contrast with*: **Sequential Development**.

**Incremental Development** – A **software development process** in which the requirements definition, design, implementation, and testing occur in an overlapping, iterative (rather than sequential) manner, resulting in incremental completion of the overall software product. Requirements are more or less fixed at the start, each increment delivers more and more parts of the final system. *Example of:* **Iterative Development**, *Contrast with*: **Waterfall Model**.

**Incremental Delivery –** A **software lifecycle process** in which not only is the software development is conducted by **incremental development**, but moreover as each increment is completed it is immediately delivered to the customer and fielded. This makes for early delivery of partial functionality. However, it assumes that functionality once deployed is not subsequently withdrawn.

**Evolutionary Software Development** – The development of (Systems and) Software, able to evolve with little effort to meet changing user needs, to interact with changing environments, and to benefit from emerging new technology.

**Waterfall Model –** A model of a **software development process** in which the constituent activities, typically a concept phase, requirements phase, design phase, implementation phase, test phase, and installation and checkout phase, are performed in that order, possibly with overlap but with little or no iteration. *Example of:* **Sequential Development**, *Contrast with*: **Incremental development**; **Rapid prototyping**; **Spiral model**.

**Spiral Model** – A model of a **software development process** in which the constituent activities, typically requirements analysis, preliminary and detailed design, coding, integration, and testing, are performed in iterations until the software is complete. The system is only delivered at the end of the whole process. *Example of:* **Iterative Development**, *Contrast with*: **Waterfall Model**.

**Architecture [IEEE1471 2000]** – The fundamental organisation of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution.

**Software Architecture (*see* Architecture) –** The fundamental organisation of software embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution.

# Annex B – LINKS TO OTHER RESEARCH GROUPS AND WORKSHOPS ADDRESSING SOFTWARE EVOLUTION

1) http://swerl.tudelft.nl/, web site of The Software Evolution Research Laboratory of the Technical University of Delft in The Netherlands. This research group aims at finding the principles, concepts, tools and methods that are needed to keep software flexible and adjustable to ever changing business and technological requirements.

   *"A software system that is actually used is a living, dynamic entity, which is subjected to continuous change."*

2) http://www.rug.nl/informatica/onderzoek/programmas/softwareEngineering/, the Software Engineering and Architecture (SEARCH) group at the University of Groningen. The SEARCH group was founded in September 2000 when Jan Bosch was appointed as a Software Engineering professor at the University of Groningen. Before that he had been leading the RISE (research in software engineering) group at the University of Karlskrona/Ronneby (http://www.ipd.hk-r.se/). The focus of the SEARCH group is on software architecture assessment and design, software product lines, software architecture and usability and software reuse.

3) http://www.rcost.unisannio.it/iwpse2005/, International Workshop on Principles of Software Evolution (IWPSE). Topics of this workshop include theory of evolution, architecture for evolution, evolution of architecture, software process for evolution, evolution of software process, methodology for evolutionary design and development, and many more. Workshops have been held at:

   2005 Lisbon, Portugal
   2004 Kyoto, Japan
   2003 Helsinki, Finland
   2002 Orlando, Florida, USA
   2001 Vienna, Austria
   2000 Kanazawa, Japan
   1999 Fukuoka, Japan
   1998 Kyoto, Japan

4) http://esecfse05.di.fct.unl.pt/, European Software Engineering Conference (ESEC, biannual) in co-operation with the International Symposium on Foundations of Software Engineering (FSE). During interleaved years, the FSE is held in USA together with the ACM SIGSOFT (Special Interest Group on Software Engineering, http://www.acm.org/sigsoft/) conference. These general software engineering-conferences also include tutorials, workshops and paper sessions on evolution and architectures. Past conferences:

   2005 Lisbon, Portugal, ESEC/FSE-'05
   2004 Newport Beach, California, USA, ACM SIGSOFT/FSE-12
   2003 Helsinki, Finland, ESEC/FSE-11
   2002 Charleston, South Carolina, USA, ACM SIGSOFT/FSE-10
   2001 Vienna, Austria, ESEC/FSE-9
   2000 San Diego, California, USA, ACM SIGSOFT/FSE-8

# Annex C – REVIEW OF HISTORY OF TASK GROUP

The idea for an Exploratory Team on Evolutionary Software Development was first proposed in the second IST Panel meeting in 1998. Following initial preliminary discussions over the Internet and in a short meeting by interested parties, a presentation was given to the Panel during its Fall 1999 meeting which was widely accepted. A sufficient number of Panel members supported the formation of a Task Group on the subject for more detailed and substantial work.

The Exploratory Team that preceded the Task Group met in Brussels in 2000.

The Task Group itself held meetings in:

1) Istanbul, Turkey, October 2000
2) Paris, France, March 2001
3) Quebec City, Canada, May 2001
4) Warsaw, Poland, October 2001
5) Estoril, Portugal, May 2002
6) Bonn, Germany, September 2002 (Symposium IST-034/RSY-10)
7) Toulouse, France, January 2003
8) Chester, UK, April 2003
9) Prague, Czech Republic, October 2003

# Annex D – MEMBERS OF TASK GROUP 026/RTG-008

## TASK GROUP CHAIRMAN

**CANADA**
Dr. W. Morven GENTLEMAN
Global Information Networking Institute
Dalhousie University
6050 University Avenue
Halifax, Nova Scotia B3H 1W5
Email: Morven.Gentleman@dal.ca
Tel. +1 (902) 494-2652
Fax. +1 (902) 492-1517

## TASK GROUP MEMBERS

**CZECH REPUBLIC**
Dr. Milan SNAJDER
Military Technical Institute of Electronics
Pod Vodovodem 2
15801 PRAGUE 5
Email: msnajder@vtue.cz
Tel. +420 (2) 20.20.77.31
Fax. +420 (2) 57.21.60.54

**FRANCE**
Dr. Michel LEMOINE
ONERA/DPRS/SAE
2, avenue Edouard Belin
B.P. 4025
31055 Toulouse Cedex 4
Email: Michel.Lemoine@cert.fr
Tel. +33 (0) 5.62.25.26.45
Fax. +33 (0) 5.62.25.25.93

**HUNGARY**
Mr. Attila SAPO
Computer and Automation
Research Institute of the Hungarian Academy of Sciences
PO Box 63.H 1518 Budapest
Email: sajo@ilab.sztaki.hu
Tel. +36 (1) 466.56.44 Ext. 417
Fax. +36 (1) 466.75.03

**NETHERLANDS**
Mr. Yves Van de VIJVER
National Aerospace Laboratory
Anthony Fokkerweg 2
PO Box 90502
1006 BM Amsterdam

**POLAND**
Capt. Ryszard RUGALA
R&D Marine Technology Centre
Dickmana 62
81-109 Gdynia
Email: rru@obr.ctm.gdynia.pl
Tel. +48 (58) 666.53.40

Capt. Dr. Grzegorz BLIZNIUK
Military University of Technology
Kaliskiego 2
00-908 Warsaw
Email: gbliz@isi.wcy.waw.pl
Tel. +48 (22) 685.71.28
Fax. +48 (22) 685.78.58

**PORTUGAL**
Mr. Helder Antonio De CAMPOS DORES
Army Software Center
Lisbon
Email: Dores.hac@mail.exercito.pt

Mr. Antonio RITO SILVA
INESC – Software Engineering Group
Technical University of Lisbon
Lisbon
Email: Rito.silva@inesc.pt

**SLOVAK REPUBLIC**
Ir. Luboslav LACKO
Military Technology Institute
ul. Kpt. Nalepku
03101 Lipovsky Mikulas
Email: lacko@vtu.army.sk
Tel. +421 (849) 552.56.37
Fax. +421 (849) 552.56.37

**TURKEY**
Dr. Fuat INCE
ISIK University
Büyükdere Cad
80670 Maslak
Istanbul
Email: fince@isikun.edu.tr
Tel. +90 (212) 286.29.60 Ext. 22.51
Fax. +90 (212) 285. 28.75

The active participants, who attended almost every meeting and regularly conducted business electronically, were Gentleman, Snajder, Lemoine, Van de Vijver, Rugala and Lacko.

# Iterative and Incremental Development: A Brief History


**Although many view iterative and incremental development as a modern practice, its application dates as far back as the mid-1950s. Prominent software-engineering thought leaders from each succeeding decade supported IID practices, and many large projects used them successfully.**

*Craig Larman*
Valtech

*Victor R. Basili*
University of Maryland

As agile methods become more popular, some view iterative, evolutionary, and incremental software development—a cornerstone of these methods—as the "modern" replacement of the waterfall model, but its practiced and published roots go back decades. Of course, many software-engineering students are aware of this, yet surprisingly, some commercial and government organizations still are not.

This description of projects and individual contributions provides compelling evidence of iterative and incremental development's (IID's) long existence. Many examples come from the 1970s and 1980s—the most active but least known part of IID's history. We are mindful that the idea of IID came independently from countless unnamed projects and the contributions of thousands and that this list is merely representative. We do not mean this article to diminish the unsung importance of other IID contributors.

We chose a chronology of IID projects and approaches rather than a deep comparative analysis. The methods varied in such aspects as iteration length and the use of time boxing. Some attempted significant up-front specification work followed by incremental time-boxed development, while others were more classically evolutionary and feedback driven. Despite their differences, however, all the approaches had a common theme—to avoid a single-pass sequential, document-driven, gated-step approach.

Finally, a note about our terminology: Although some prefer to reserve the phrase "iterative development" merely for rework, in modern agile methods the term implies not just revisiting work, but also evolutionary advancement—a usage that dates from at least 1968.

## PRE-1970

IID grew from the 1930s work of Walter Shewhart,[1] a quality expert at Bell Labs who proposed a series of short "plan-do-study-act" (PDSA) cycles for quality improvement. Starting in the 1940s, quality guru W. Edwards Deming began vigorously promoting PDSA, which he later described in 1982 in *Out of the Crisis*.[2] Tom Gilb[3] and Richard Zultner[4] also explored PDSA application to software development in later works.

The X-15 hypersonic jet was a milestone 1950s project applying IID,[5] and the practice was considered a major contribution to the X-15's success. Although the X-15 was not a software project, it is noteworthy because some personnel—and hence, IID experience—seeded NASA's early 1960s Project Mercury, which did apply IID in software. In addition, some Project Mercury personnel seeded the IBM Federal Systems Division (FSD), another early IID proponent.

Project Mercury ran with very short (half-day) iterations that were time boxed. The development team conducted a technical review of all changes, and, interestingly, applied the Extreme Programming practice of test-first development, planning and writing tests before each micro-increment. They also practiced top-down development with stubs.


Published by the IEEE Computer Society

The recollections of Gerald M. Weinberg, who worked on the project, provide a window into some practices during this period. In a personal communication, he wrote:

> We were doing incremental development as early as 1957, in Los Angeles, under the direction of Bernie Dimsdale [at IBM's Service Bureau Corporation]. He was a colleague of John von Neumann, so perhaps he learned it there, or assumed it as totally natural. I do remember Herb Jacobs (primarily, though we all participated) developing a large simulation for Motorola, where the technique used was, as far as I can tell, indistinguishable from XP.
>
> When much of the same team was reassembled in Washington, DC in 1958 to develop Project Mercury, we had our own machine and the new Share Operating System, whose symbolic modification and assembly allowed us to build the system incrementally, which we did, with great success. Project Mercury was the seed bed out of which grew the IBM Federal Systems Division. Thus, that division started with a history and tradition of incremental development.
>
> All of us, as far as I can remember, thought waterfalling of a huge project was rather stupid, or at least ignorant of the realities… I think what the waterfall description did for us was make us realize that we were doing something else, something unnamed except for "software development."

The earliest reference we found that specifically focused on describing and recommending iterative development was a 1968 report from Brian Randell and F.W. Zurcher at the IBM T.J. Watson Research Center.[6] M.M. Lehman later described Randell and Zurcher's work and again promoted iterative development in his September 1969 internal report to IBM management on development recommendations:[7]

> The basic approach recognizes the futility of separating design, evaluation, and documentation processes in software-system design. The design process is structured by an expanding model seeded by a formal definition of the system, which provides a first, executable, functional model. It is tested and further expanded through a sequence of models, that develop an increasing amount of function and an increasing amount of detail as to how that function is to be executed. Ultimately, the model becomes the system.

Another 1960s reference comes from Robert Glass:[8]

> It is the opinion of the author that incremental development is worthwhile, [it] leads to a more thorough system shakedown, avoids implementer and management discouragement.

## THE SEVENTIES

In his well-known 1970 article, "Managing the Development of Large Software Systems," Winston Royce shared his opinions on what would become known as the waterfall model, expressed within the constraints of government contracting at that time.[9] Many—incorrectly—view Royce's paper as the paragon of single-pass waterfall. In reality, he recommended an approach somewhat different than what has devolved into today's waterfall concept, with its strict sequence of requirements analysis, design, and development phases. Indeed, Royce's recommendation was to do it *twice*:

> If the computer program in question is being developed for the first time, arrange matters so that the version finally delivered to the customer for operational deployment is actually the second version insofar as critical design/operations areas are concerned.

Royce further suggested that a 30-month project might have a 10-month pilot model and justified its necessity when the project contains novel elements and unknown factors (hardly a unique case). Thus, we see hints of iterative development, feedback, and adaptation in Royce's article. This iterative feedback-based step has been lost in most descriptions of this model, although it is clearly not classic IID.

What did Royce think about the waterfall versus IID when he learned of the latter approach? In a personal communication, Walker Royce, his son and a contributor to popular IID methods in the 1990s, said this of his father and the paper:

> He was always a proponent of iterative, incremental, evolutionary development. His paper described the waterfall as the simplest description, but that it would not work for all but the most straightforward projects. The rest of his paper describes [iterative practices] within the context of the 60s/70s government-contracting models (a serious set of constraints).

**"We were doing incremental development as early as 1957, in Los Angeles, under the direction of Bernie Dimsdale [at IBM's Service Bureau Corporation]."**

**The first major documented IBM FSD IID application was the life-critical command and control system for the first US Trident submarine.**

This was an ironic insight, given the influence this paper had as part of the bulwark promoting a strict sequential life cycle for large, complex projects.

The next earliest reference comes from Harlan Mills, a 1970s software-engineering thought leader who worked at the IBM FSD. In his well-known "Top-Down Programming in Large Systems," Mills promoted iterative development. In addition to his advice to begin developing from top-level control structures downward, perhaps less appreciated was the related life-cycle advice Mills gave for building the system via iterated expansions:[10]

> … it is possible to generate a sequence of intermediate systems of code and functional subspecifications so that at every step, each [intermediate] system can be verified to be correct…

Clearly, Mills suggested iterative refinement for the development phase, but he did not mention avoiding a large up-front specification step, did not specify iteration length, and did not emphasize feedback and adaptation-driven development from each iteration. He did, however, raise these points later in the decade. Given his employment at the IBM FSD, we suspect Mills's exposure to the more classic IID projects run there in the early 1970s influenced his thought, but we could not confirm this with colleagues.

Early practice of more modern IID (feedback-driven refinement with customer involvement and clearly delineated iterations) came under the leadership of Mike Dyer, Bob McHenry, and Don O'Neill and many others during their tenure at IBM FSD. The division's story is fascinating because of the extent and success of its IID use on large, life-critical US Department of Defense (DoD) space and avionics systems during this time.

The first major documented IBM FSD application of IID that we know of was in 1972. This was no toy application, but a high-visibility life-critical system of more than 1 million lines of code—the command and control system for the first US Trident submarine. O'Neill was project manager, and the project included Dyer and McHenry. O'Neill conceived and planned the use of IID (which FSD later called "integration engineering") on this project; it was a key success factor, and he was awarded an IBM Outstanding Contribution Award for the work. (Note that IBM leadership visibly approved of IID methods.)

The system had to be delivered by a certain date or FSD would face a $100,000 per day late penalty. The team organized the project into four time-boxed iterations of about six months each. There was still a significant up-front specification effort, and the iteration was longer than normally recommended today. Although some feedback-driven evolution occurred in the requirements, O'Neill noted that the IID approach was also a way to manage the complexity and risks of large-scale development.[11]

Also in 1972, an IBM FSD competitor, TRW, applied IID in a major project—the $100 million TRW/Army Site Defense software project for ballistic missile defense. The project began in February 1972, and the TRW team developed the system in five iterations. Iteration 1 tracked a single object, and by iteration 5, a few years later, the system was complete. The iterations were not strictly time boxed, and there was significant up-front specification work, but the team refined each iteration in response to the preceding iteration's feedback.[12]

As with IBM FSD, TRW (where Royce worked) was an early adopter of IID practices. Indeed, Barry Boehm, the originator of the IID spiral model in the mid-1980s, was chief scientist at TRW.

Another mid-1970s extremely large application of IID at FSD was the development of the Light Airborne Multipurpose System, part of the US Navy's helicopter-to-ship weapon system. A four-year 200-person-year effort involving millions of lines of code, LAMPS was incrementally delivered in 45 time-boxed iterations (one month per iteration). This is the earliest example we found of a project that used an iteration length in the range of one to six weeks, the length that current popular IID methods recommend. The project was quite successful: As Mills wrote, "Every one of those deliveries was on time and under budget."[13]

In 1975, Vic Basili and Joe Turner published a paper about iterative enhancement that clearly described classic IID:[14]

> The basic idea behind iterative enhancement is to develop a software system incrementally, allowing the developer to take advantage of what was being learned during the development of earlier, incremental, deliverable versions of the system. Learning comes from both the development and use of the system, where possible. Key steps in the process were to start with a simple implementation of a subset of the software requirements and iteratively enhance the evolving sequence of versions until the full system is implemented. At each iteration, design modifications are made along with adding new functional capabilities.

The paper detailed successful IID application to the development of extendable compilers for a family of application-specific programming languages on a variety of hardware architectures. The project team developed the base system in 17 iterations over 20 months. They analyzed each iteration from both the user's and developer's points of view and used the feedback to modify both the language requirements and design changes in future iterations. Finally, they tracked measures, such as coupling and cohesion, over the multiple iterations.

In 1976, Tom Gilb published *Software Metrics* (coining the term), in which he discussed his IID practice—evolutionary project management—and introduced the terms "evolution" and "evolutionary" to the process lexicon. This is the earliest book we could find that had a clear IID discussion and promotion, especially of evolutionary delivery:[3]

> "Evolution" is a technique for producing the appearance of stability. A complex system will be most successful if it is implemented in small steps and if each step has a clear measure of successful achievement as well as a "retreat" possibility to a previous successful step upon failure. You have the opportunity of receiving some feedback from the real world before throwing in all resources intended for a system, and you can correct possible design errors…

The book marked the arrival of a long-standing and passionate voice for evolutionary and iterative development. Gilb is one of the earliest and most active IID practitioners and promoters. He began the practice in the early 1960s and went on to establish several IID milestones. His material was probably the first with a clear flavor of agile, light, and adaptive iteration with quick results, similar to that of newer IID methods.

By 1976, Mills had strengthened his IID message:[15]

> Software development should be done incrementally, in stages with continuous user participation and replanning and with design-to-cost programming within each stage.

Using a three-year inventory system project as a backdrop, he challenged the idea and value of upfront requirements or design specification:

> ...there are dangers, too, particularly in the conduct of these [waterfall] stages in sequence, and not in iteration-i.e., that development is done in an open loop, rather than a closed loop with user feedback between iterations. The danger in the sequence [waterfall approach] is that the project moves from being grand to being grandiose, and exceeds our human intellectual capabilities for management and control.

**Tom Gilb introduced the terms "evolution" and "evolutionary" to the process lexicon.**

And perhaps reflecting several years of seeing IID in action at FSD, Mills asked, "...why do enterprises tolerate the frustrations and difficulties of such [waterfall] development?"

In 1977, FSD incorporated the Trident IID approach, which included integrating all software components at the end of each iteration into its software-engineering practices—an approach McHenry dubbed "integration engineering." Some Trident team members and Mills were key advisers in this incorporation effort.[16] Integration engineering spread to the 2,500 FSD software engineers, and the idea of IID as an alternative to the waterfall stimulated substantial interest within IBM's commercial divisions and senior customer ranks and among its competitors.

Although unknown to most software professionals, another early and striking example of a major IID success is the very heart of NASA's space shuttle software—the primary avionics software system, which FSD built from 1977 to 1980. The team applied IID in a series of 17 iterations over 31 months, averaging around eight weeks per iteration.[17] Their motivation for avoiding the waterfall life cycle was that the shuttle program's requirements changed during the software development process. Ironically (in hindsight), the authors sound almost apologetic about having to forego the "ideal" waterfall model for an IID approach:

> Due to the size, complexity, and evolutionary [changing requirements] nature of the program, it was recognized early that the ideal software development life cycle [the waterfall model] could not be strictly applied...However, an implementation approach (based on small incremental releases) was devised for STS-1 which met the objectives by applying the ideal cycle to small elements of the overall software package on an iterative basis.

The shuttle project also exhibited classic IID practices: time-boxed iterations in the eight-week range, feedback-driven refinement of specifications, and so on.

The first IID discussion in the popular press that we could find was in 1978, when Tom Gilb began publishing a column in the UK's *Computer Weekly*.

**The IID practice of evolutionary prototyping was commonly used in 1980s efforts to create artificial intelligence systems.**

The column regularly promoted IID, as well as evolutionary project management and delivery. In his 6 April 1978 column, Gilb wrote,

> Management does not require firm estimates of completion, time, and money for the entire project. Each [small iterative] step must meet one of the following criteria (priority order): either (a) give planned return on investment payback, or, if impossible, then (b) give breakeven (no loss); or, at least, (c) some positive user benefit measurably; or, at least (d) some user environment feedback and learning.

Another discussion of incremental development, although published in 1984, refers to a System Development Corp. project to build an air defense system, which began in 1977 and finished in 1980. The project combined significant up-front specifications with incremental development and builds. Ostensibly, the project was meant to fit within DoD single-pass waterfall standards, with testing and integration in the last phase. Carolyn Wong comments on the unrealism of this approach and the team's need to use incremental development:[18]

> The [waterfall] model was adopted because software development was guided by DoD standards…In reality, software development is a complex, continuous, iterative, and repetitive process. The [waterfall model] does not reflect this complexity.

## THE EIGHTIES

In 1980 Weinberg wrote about IID in "Adaptive Programming: The New Religion," published in Australasian *Computerworld*. Summarizing the article, he said, "The fundamental idea was to build in small increments, with feedback cycles involving the customer for each." A year later, Tom Gilb wrote in more detail about evolutionary development.[19]

In the same year, Daniel McCracken and Michael Jackson promoted IID and argued against the "stultifying waterfall" in a chapter within a software engineering and design text edited by William Cotterman. The chapter's title, "A Minority Dissenting Position," underscored the subordinate position of IID to the waterfall model at the time.[20] Their arguments continued in "Life-Cycle Concept Considered Harmful,"[21] a 1982 twist on Edsger Dijkstra's late 1960s classic "Go To Statement Considered Harmful."[22] (The use of "life cycle" as a synonym for waterfall during this period suggests its unquestioned dominance. Contrast this to its qualified use in the 1990s, "sequential life cycle" or "iterative life cycle.")

In 1982, William Swartout and Robert Balzer argued that specification and design have a necessary interplay, and they promoted an iterative and evolutionary approach to requirements engineering and development.[23] The same year also provided the earliest reference to a very large application successfully built using evolutionary prototyping, an IID approach that does not usually include time-boxed iterations. The $100 million military command and control project was based on IBM's Customer Information Control System technology.[24]

In 1983, Grady Booch published *Software Engineering with Ada*,[25] in which he described an iterative process for growing an object-oriented system. The book was influential primarily in the DoD development community, but more for the object-oriented design method than for its iterative advice. However, Booch's later 1990s books that covered IID found a large general audience, and many first considered or tried iterative development through their influence.

The early 1980s was an active period for the (attempted) creation of artificial intelligence systems, expert systems, and so on, especially using Lisp machines. A common approach in this community was the IID practice of evolutionary prototyping.[26]

In another mid-1980s questioning of the sequential life cycle, Gilb wrote "Evolutionary Delivery versus the 'Waterfall Model.'" In this paper, Gilb promoted a more aggressive strategy than other IID discussions of the time, recommending frequent (such as every few weeks) delivery of useful results to stakeholders.[27]

A 1985 landmark in IID publications was Barry Boehm's "A Spiral Model of Software Development and Enhancement," (although the more frequent citation date is 1986).[28] The spiral model was arguably not the first case in which a team prioritized development cycles by risk: Gilb and IBM FSD had previously applied or advocated variations of this idea, for example. However, the spiral model did formalize and make prominent the risk-driven-iterations concept and the need to use a discrete step of risk assessment in each iteration.

In 1986, Frederick Brooks, a prominent software-engineering thought leader of the 1970s and 1980s, published the classic "No Silver Bullet" extolling the advantages of IID:[29]

> Nothing in the past decade has so radically changed my own practice, or its effectiveness [as incremental development].

Commenting on adopting a waterfall process, Brooks wrote

> Much of present-day software acquisition procedure rests upon the assumption that one can specify a satisfactory system in advance, get bids for its construction, have it built, and install it. I think this assumption is fundamentally wrong, and that many software acquisition problems spring from that fallacy.

Perhaps summing up a decade of IID-promoting messages to military standards bodies and other organizations, Brooks made his point very clear in his keynote speech at the 1995 International Conference on Software Engineering: "The waterfall model is wrong!"

In 1986, David Parnas and Paul Clements published "A Rational Design Process: How and Why to Fake It."[30] In it, they stated that, although they believe in the ideal of the waterfall model (thorough, correct, and clear specifications before development), it is impractical. They listed many reasons, including (paraphrased)

- A system's users seldom know exactly what they want and cannot articulate all they know.
- Even if we could state all requirements, there are many details that we can only discover once we are well into implementation.
- Even if we knew all these details, as humans, we can master only so much complexity.
- Even if we could master all this complexity, external forces lead to changes in requirements, some of which may invalidate earlier decisions.

and commented that for all these reasons, "the picture of the software designer deriving his design in a rational, error-free way from a statement of requirements is quite unrealistic."

In 1987, TRW launched a four-year project to build the Command Center Processing and Display System Replacement (CCPDS-R), a command and control system, using IID methods. Walker Royce described the effort in 60 pages of detail.[31] The team time-boxed six iterations, averaging around six months each. The approach was consistent with what would later become the Rational Unified Process (to which Royce contributed): attention to high risks and the core architecture in the early iterations.

Bill Curtis and colleagues published a particularly agile-relevant paper during this decade,[32] reporting results on research into the processes that influenced 19 large projects. The authors identified that the prescriptive waterfall model attempted to satisfy management accountability goals, but they did not describe how projects successfully ran. The paper also noted that successful development emphasizes a cyclic learning process with high attention to people's skills, common vision, and communication issues, rather than viewing the effort as a sequential "manufacturing process." As the authors state,

> The conclusion that stands out most clearly from our field study observations is that the process of developing large software systems must be treated, at least in part, as a learning and communication process.

In 1987, as part of the IBM FSD Software Engineering Practices program, Mills, Dyer, and Rick Linger continued the evolution of IID with the Cleanroom method, which incorporated evolutionary development with more formal methods of specification and proof, reflecting Mills's strong mathematical influences.[33]

**The Cleanroom method incorporated evolutionary development with more formal methods of specification and proof.**

By the late 1980s, the DoD was experiencing significant failure in acquiring software based on the strict, document-driven, single-pass waterfall model that DoD-Std-2167 required. A 1999 review of failure rates in a sample of earlier DoD projects drew grave conclusions: "Of a total $37 billion for the sample set, 75% of the projects failed or were never used, and only 2% were used without extensive modification."[34] Consequently, at the end of 1987, the DoD changed the waterfall-based standards to allow IID, on the basis of recommendations in an October 1987 report from the Defense Science Board Task Force on Military Software, chaired by Brooks. The report recommended replacing the waterfall, a failing approach on many large DoD projects, with iterative development:

> DoD-Std-2167 likewise needs a radical overhaul to reflect modern best practice. Draft 2167A is a step, but it does not go nearly far enough. As drafted, it continues to reinforce exactly the document-driven, specify-then-build approach that lies at the heart of so many DoD software problems....

**Tom Gilb's *Principles of Software Engineering Management* was the first book with substantial chapters dedicated to IID discussion and promotion.**

> In the decade since the waterfall model was developed, our discipline has come to recognize that [development] requires iteration between the designers and users.

Finally, in a section titled "Professional Humility and Evolutionary Development" (humility to accept that the 2167's goals—get the specifications accurate without incremental implementation and feedback—was not possible), the report stated:

> Experience with confidently specifying and painfully building mammoths has shown it to be simplest, safest, and even fastest to develop a complex software system by building a minimal version, putting it into actual use, and then adding functions [and other qualities] according to the priorities that emerge from actual use.
>
> Evolutionary development is best technically, and it saves time and money.

Both DoD overseers and contractors often view the updated DoD-Std-2167A, released in February 1988, as the epitome of a waterfall specification. Yet, its authors actually wanted it to be an amendment (hence the A) for life-cycle neutrality that allowed IID alternatives to the waterfall:

> This standard is not intended to specify or discourage the use of any particular software development method. The contractor is responsible for selecting software development methods (for example, rapid prototyping) that best support the achievement of contract requirements.

Despite this intent, many (justifiably) interpreted the new standard as containing an implied preference for the waterfall model because of its continued document-driven milestone approach.

Ironically, in a conversation nearly a decade later, the principal creator of DoD-Std-2167 expressed regret for creating the strict waterfall-based standard. He said that at the time he knew of the single-pass document-driven waterfall model, and others he questioned advised it was excellent, as did the literature he examined, but he had not heard of iterative development. In hindsight, he said he would have made a strong recommendation for IID rather than the waterfall model.

In 1988, Gilb published *Principles of Software Engineering Management*, the first book with substantial chapters dedicated to IID discussion and promotion.[35] In it he reiterated and expanded on the IID material from *Software Metrics*. Gilb described the Evo method, distinguished by frequent evolutionary delivery and an emphasis on defining quantified measurable goals and then measuring the actual results from each time-boxed short iteration.

## 1990 TO THE PRESENT

By the 1990s, especially the latter half, public awareness of IID in software development was significantly accelerating. Hundreds of books and papers were promoting IID as their main or secondary theme. Dozens more IID methods sprang forth, which shared an increasing trend to time-boxed iterations of one to six weeks.

In the 1970s and 1980s, some IID projects still incorporated a preliminary major specification stage, although their teams developed them in iterations with minor feedback. In the 1990s, in contrast, methods tended to avoid this model, preferring less early specification work and a stronger evolutionary analysis approach.

The DoD was still experiencing many failures with "waterfall-mentality" projects. To correct this and to reemphasize the need to replace the waterfall model with IID, the Defense Science Board Task Force on Acquiring Defense Software Commercially, chaired by Paul Kaminski, issued a report in June 1994 that stated simply, "DoD must manage programs using iterative development. Apply evolutionary development with rapid deployment of initial functional capability."

Consequently, in December 1994, Mil-Std-498 replaced 2167A. An article by Maj. George Newberry summarizing the changes included a section titled "Removing the Waterfall Bias," in which he described the goal of encouraging evolutionary acquisition and IID:[36]

> Mil-Std-498 describes software development in one or more incremental builds. Each build implements a specified subset of the planned capabilities. The process steps are repeated for each build, and within each build, steps may be overlapping and iterative.

Mil-Std-498 itself clearly states the core IID practices of evolving requirements and design incrementally with implementation:

> If a system is developed in multiple builds, its requirements may not be fully defined until the final build.... If a system is designed in multiple builds, its design may not be fully defined until the final build.

Meanwhile, in the commercial realm, Jeff Sutherland and Ken Schwaber at Easel Corp. had started to apply what would become known as the Scrum method, which employed time-boxed 30-day iterations. The method took inspiration from a Japanese IID approach used for nonsoftware products at Honda, Canon, and Fujitsu in the 1980s; from Shashimi ("slices" or iterations); and from a version of Scrum described in 1986.[37] A 1999 article described their later refinements to Scrum.[38]

In January 1994, a group of 16 rapid application development (RAD) practitioners met in the UK to discuss the definition of a standard iterative process to support RAD development. The group drew inspiration from James Martin's RAD teachings. Martin, in turn, had taken his inspiration from the time-boxing work at Dupont, led by Scott Shultz in the mid-1980s. The RAD group's process definition would eventually become the Dynamic Systems Development Method (DSDM), an IID method that predictably had more early advocates in Europe and has since spread.[39]

In the early 1990s, a consortium of companies began a project to build a new-generation Canadian Automated Air Traffic Control System (CAATS) using a risk-driven IID method. The project, under the process leadership of Philippe Kruchten, used a series of six-month iterations, relatively long by today's standards. The project was a success, despite its prior near-failure applying a waterfall approach.[40]

In the mid-1990s, many contributors within Rational Corp. (including Kruchten and Walker Royce) and its clients created the Rational Unified Process, now a popular IID method. A 1995 milestone was the public promotion of the daily build and smoke test, a widely influential IID practice institutionalized by Microsoft that featured a one-day micro-iteration.[41]

In 1996, Kent Beck joined the Chrysler C3 payroll project. It was in this context that the full set of XP practices matured, with some collaboration by Ron Jeffries and inspiration from earlier 1980s work at Tektronix with Ward Cunningham. XP went on to garner significant public attention because of its emphasis on communication, simplicity, and testing, its sustainable developer-oriented practices, and its interesting name.[42]

In 1997, a project to build a large logistics system in Singapore, which had been running as a waterfall project, was facing failure. With the collaboration of Peter Coad and Jeff De Luca, the team resurrected it and ran it as a successful IID project. DeLuca created an overall iterative process description, Feature-Driven Development (FDD), that also incorporated ideas from Coad.[43]

In 1998, the Standish Group issued its widely cited "CHAOS: Charting the Seas of Information Technology," a report that analyzed 23,000 projects to determine failure factors. The top reasons for project failure, according to the report, were associated with waterfall practices. It also concluded that IID practices tended to ameliorate the failures. One of the report's key conclusions was to adopt IID:

> Research also indicates that smaller time frames, with delivery of software components early and often, will increase the success rate. Shorter time frames result in an iterative process of design, prototype, develop, test, and deploy small elements.

In 2000, DoD replaced Mil-Std-498 with another software acquisition standard, DoD 5000.2, which again recommended adopting evolutionary acquisition and the use of IID:

> There are two approaches, evolutionary and single step [waterfall], to full capability. An evolutionary approach is preferred. … [In this] approach, the ultimate capability delivered to the user is divided into two or more blocks, with increasing increments of capability...software development shall follow an iterative spiral development process in which continually expanding software versions are based on learning from earlier development.

In 2001, Alan MacCormack reported a study of key success factors in recent projects; first among these was adopting an IID life cycle:[44]

> Now there is proof that the evolutionary approach to software development results in a speedier process and higher-quality products. […] The iterative process is best captured in the evolutionary delivery model proposed by Tom Gilb.

In February 2001, a group of 17 process experts—representing DSDM, XP, Scrum, FDD, and others—interested in promoting modern, simple IID methods and principles met in Utah to discuss common ground. From this meeting came the Agile Alliance (www.agilealliance.org) and the now

**XP garnered significant public attention because of its emphasis on communication, simplicity, and testing, and its sustainable developer-oriented practices.**

popular catch phrase "agile methods," all of which apply IID. And in 2002, Alistair Cockburn, one of the participants, published the first book under the new appellation.[45]

In a typical quip, H.L. Mencken said, "For every complex problem, there is a solution that is simple, neat, and wrong." In the history of science, it is the norm that simplistic but inferior ideas first hold the dominant position, even without supporting results. Medicine's four humors and related astrological diagnosis and prescription dominated Europe for more than a millennium, for example.

Software development is a very young field, and it is thus no surprise that the simplified single-pass and document-driven waterfall model of "requirements, design, implementation" held sway during the first attempts to create the ideal development process. Other reasons for the waterfall idea's early adoption or continued promotion include:

- It's simple to explain and recall. "Do the requirements, then design, and then implement." IID is more complex to understand and describe. Even Winston Royce's original two-iteration waterfall immediately devolved into a single sequential step as other adopters used it and writers described it.
- It gives the illusion of an orderly, accountable, and measurable process, with simple document-driven milestones (such as "requirements complete").
- It was promoted in many software engineering, requirements engineering, and management texts, courses, and consulting organizations. It was labeled appropriate or ideal, seemingly unaware of this history or of the statistically significant research evidence in favor of IID.

This brief history shows that IID concepts have been and are a recommended practice by prominent software-engineering thought leaders of each decade, associated with many successful large projects, and recommended by standards boards.

Yet, even though the value of IID is well known among literate, experienced software engineers, some commercial organizations, consulting companies, and standards bodies still promote a document-driven single-pass sequential life cycle as the ideal. We conclude with this recommendation: In the interest of promoting greater project success and saving taxpayer or investor dollars, let's continue efforts to educate and promote the use of IID methods. ■

## References


1. W. Shewhart, *Statistical Method from the Viewpoint of Quality Control*, Dover, 1986 (reprint from 1939).
2. W.E. Deming, *Out of the Crisis*, SPC Press, 1982; reprinted in paperback by MIT Press, 2003.
3. T. Gilb, *Software Metrics*, Little, Brown, and Co., 1976 (out of print).
4. R. Zultner, "The Deming Approach to Quality Software Engineering," *Quality Progress*, vol. 21, no. 11, 1988, pp. 58-64.
5. W.H. Dana, *The X-15 Lessons Learned*, tech. report, NASA Dryden Research Facility, 1993.
6. B. Randell and F.W. Zurcher, "Iterative Multi-Level Modeling: A Methodology for Computer System Design," *Proc. IFIP*, IEEE CS Press, 1968, pp. 867-871.
7. M.M. Lehman, "The Programming Process," internal IBM report, 1969; reprinted in *Program Evolution—Processes of Software Change*, Academic Press, 1985.
8. R. Glass, "Elementary Level Discussion of Compiler/Interpreter Writing," *ACM Computing Surveys*, Mar. 1969, pp. 64-68.
9. W. Royce, "Managing the Development of Large Software Systems," *Proc. Westcon*, IEEE CS Press, 1970, pp. 328-339.
10. H. Mills, "Debugging Techniques in Large Systems," *Software Productivity*, Dorset House, 1988.
11. D. O'Neill, "Integration Engineering Perspective," *J. Systems and Software*, no. 3, 1983, pp. 77-83.
12. R.D. Williams, "Managing the Development of Reliable Software," *Proc. Int'l Conf. Reliable Software*, ACM Press, 1975, pp. 3-8.
13. H. Mills, "Principles of Software Engineering," *IBM Systems J.*, vol. 19, no. 4, 1980, pp. 289-295.
14. V. Basili and J. Turner, "Iterative Enhancement: A Practical Technique for Software Development," *IEEE Trans. Software Eng.*, Dec. 1975, pp. 390-396.
15. H. Mills, "Software Development," *IEEE Trans. Software Eng.*, Dec. 1976, pp. 265-273.
16. D. O'Neill, "The Management of Software Engineering," *IBM Systems J.*, vol. 19, no. 4, 1980, pp. 421-431.
17. W. Madden and K. Rone, "Design, Development, Integration: Space Shuttle Flight Software System," *Comm. ACM*, Sept. 1984, pp. 914-925.
18. C. Wong, "A Successful Software Development," *IEEE Trans. Software Eng.*, no. 3, 1984, pp. 714-727.
19. T. Gilb, "Evolutionary Development," *ACM Software Eng. Notes*, Apr. 1981, p. 17.
20. W.W. Cotterman et al., eds., *Systems Analysis and*

*Design: A Foundation for the 1980's*, North-Holland, 1981.
21. D. McCracken and M. Jackson, "Life-Cycle Concept Considered Harmful," *ACM Software Eng. Notes*, Apr. 1982, pp. 29-32.
22. E. Dijkstra, "Go To Statement Considered Harmful," *Comm. ACM*, Mar. 1968, pp. 147-148.
23. W. Swartout and R. Balzer, "On the Inevitable Intertwining of Specification and Implementation," *Comm. ACM,* July 1982, pp. 438-440.
24. D. Tamanaha, "An Integrated Rapid Prototyping Methodology for Command and Control Systems: Experience and Insight," *ACM Software Eng. Notes*, Dec. 1982, pp. 387-396.
25. G. Booch, *Software Engineering with Ada*, Benjamin-Cummings, 1983.
26. R. Budde et al., eds., *Approaches to Prototyping*, Springer Verlag, 1984.
27. T. Gilb, "Evolutionary Delivery versus the 'Waterfall Model'," *ACM Software Requirements Eng. Notes*, July 1985.
28. B. Boehm, "A Spiral Model of Software Development and Enhancement," *Proc. Int'l Workshop Software Process and Software Environments*, ACM Press, 1985; also in *ACM Software Eng. Notes*, Aug. 1986, pp. 22-42.
29. F. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," *Proc. IFIP,* IEEE CS Press, 1987, pp. 1069-1076; reprinted in *Computer*, Apr. 1987, pp. 10-19.
30. D. Parnas and P. Clements, "A Rational Design Process: How and Why to Fake It," *IEEE Trans. Software Eng.*, Feb. 1986, pp. 251-257.
31. W. Royce, *Software Project Management*, Addison-Wesley, 1998.
32. W. Curtis et al., "On Building Software Process Models under the Lamppost," *Proc. Int'l Conf. Software Eng.,* IEEE CS Press, 1987, pp. 96-103.
33. H. Mills et al., "Cleanroom Software Engineering," *IEEE Software*, Sept. 1987, pp. 19-25.
34. S. Jarzombek, *Proc. Joint Aerospace Weapons Systems Support, Sensors and Simulation Symp.*, Gov't Printing Office Press, 1999.
35. T. Gilb, *Principles of Software Engineering Management*, Addison Wesley Longman, 1989.
36. G.A. Newberry, "Changes from DOD-STD-2167A to MIL-STD-498," *Crosstalk: J. Defense Software Eng.*, Apr. 1995, www.stsc.hill.af.mil/crosstalk/frames.asp?uri=1995/04/Changes.asp.
37. H. Takeuchi and I. Nonaka, "The New New Product Development Game," *Harvard Business Rev.*, Jan. 1986, pp. 137-146.
38. M. Beedle et al., "SCRUM: An Extension Pattern Language for Hyperproductive Software Development," *Pattern Languages of Program Design*, vol. 4, 1999, pp. 637-651.
39. J. Stapleton, *DSDM: Dynamic Systems Development Method*, Addison-Wesley, 1997.
40. P. Kruchten, "Rational Development Process," *Crosstalk: J. Defense Software Eng.*, July 1996, www.stsc.hill.af.mil/crosstalk/frames.asp?uri=1996/07/rational.asp.
41. J. McCarthy, *Dynamics of Software Development*, Microsoft Press, 1995.
42. K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.
43. P. Coad et al., "Feature-Driven Development," in *Java Modeling in Color with UML,* Prentice Hall, 1999.
44. A. MacCormack, "Product-Development Practices That Work," *MIT Sloan Management Rev.,* vol. 42, no. 2, 2001, pp. 75-84.
45. A. Cockburn, *Agile Software Development*, Addison-Wesley, 2002.

***Craig Larman*** *is chief scientist for Valtech, an international consulting company, and he speaks and consults worldwide. He is also the author of* Agile and Iterative Development: A Manager's Guide *(Addison-Wesley, 2003), which examines both historical and other forms of evidence demonstrating the advantages of iterative methods. Larman is a member of the ACM and the IEEE. Contact him at craig@craiglarman.com.*

***Victor R. Basili*** *is a professor of computer science at the University of Maryland and executive director of the Fraunhofer Center-Maryland, where he works on measuring, evaluating, and improving the software development process and product. He is an IEEE and ACM fellow and co-editor-in-chief of Kluwer's* Empirical Software Engineering: An International Journal. *Contact him at basili@cs.umd.edu.*

# REPORT DOCUMENTATION PAGE

| 1. Recipient's Reference | 2. Originator's References | 3. Further Reference | 4. Security Classification of Document |
|---|---|---|---|
| | RTO-TR-IST-026<br>AC/323(IST-026)TP/190 | ISBN<br>978-92-837-0042-5 | UNCLASSIFIED/<br>UNLIMITED |

**5. Originator**

Research and Technology Organisation
North Atlantic Treaty Organisation
BP 25, F-92201 Neuilly-sur-Seine Cedex, France

**6. Title**

Evolutionary Software Development

**7. Presented at/Sponsored by**

Final Report of the Task Group IST-026/RTG-008.

| 8. Author(s)/Editor(s) | 9. Date |
|---|---|
| Multiple | August 2008 |
| **10. Author's/Editor's Address** | **11. Pages** |
| Multiple | 62 |

**12. Distribution Statement**

There are no restrictions on the distribution of this document. Information about the availability of this and other RTO unclassified publications is given on the back cover.

**13. Keywords/Descriptors**

Acquisition
Armed forces procurement
Criteria
Decision making
ESD (Evolutionary Software Development)
Evaluation
Management
Methodology
Military applications
Requirements
Software development
Software engineering
Specifications
Standards

**14. Abstract**

This task group investigated iterative processes for software development, especially those (called Evolutionary Software Development) that span many cycles of software implementation, release, fielding of the product, learning from the field experience, then updating the requirements for subsequent releases. This goes beyond the Spiral Model or Agile Methods when they are only used prior to initial delivery, and also beyond incremental delivery. The methodology followed was to review the literature, examine case studies, sponsor a public symposium (IST-034/RSY-010) to collect external input, and then within the task group resolve the best way to present our findings. The group itself did not have the resources to undertake any original research. The principal findings were that iterative processes have been used successfully in military software projects since the 1950's and continue to be viable and exhibit advantages over strictly sequential processes such as the Waterfall Model or the V-Model. Nevertheless, there remain outstanding research questions to be resolved with potential to improve the process.

NORTH ATLANTIC TREATY ORGANISATION

RESEARCH AND TECHNOLOGY ORGANISATION

BP 25
F-92201 NEUILLY-SUR-SEINE CEDEX • FRANCE
Télécopie 0(1)55.61.22.99 • E-mail mailbox@rta.nato.int

**DIFFUSION DES PUBLICATIONS**
**RTO NON CLASSIFIEES**

Les publications de l'AGARD et de la RTO peuvent parfois être obtenues auprès des centres nationaux de distribution indiqués ci-dessous. Si vous souhaitez recevoir toutes les publications de la RTO, ou simplement celles qui concernent certains Panels, vous pouvez demander d'être inclus soit à titre personnel, soit au nom de votre organisation, sur la liste d'envoi.

Les publications de la RTO et de l'AGARD sont également en vente auprès des agences de vente indiquées ci-dessous.

Les demandes de documents RTO ou AGARD doivent comporter la dénomination « RTO » ou « AGARD » selon le cas, suivi du numéro de série. Des informations analogues, telles que le titre est la date de publication sont souhaitables.

Si vous souhaitez recevoir une notification électronique de la disponibilité des rapports de la RTO au fur et à mesure de leur publication, vous pouvez consulter notre site Web (www.rto.nato.int) et vous abonner à ce service.

## CENTRES DE DIFFUSION NATIONAUX

**ALLEMAGNE**
Streitkräfteamt / Abteilung III
Fachinformationszentrum der Bundeswehr (FIZBw)
Gorch-Fock-Straße 7, D-53229 Bonn

**BELGIQUE**
Royal High Institute for Defence – KHID/IRSD/RHID
Management of Scientific & Technological Research for Defence, National RTO Coordinator
Royal Military Academy – Campus Renaissance
Renaissancelaan 30, 1000 Bruxelles

**CANADA**
DSIGRD2 – Bibliothécaire des ressources du savoir
R et D pour la défense Canada
Ministère de la Défense nationale
305, rue Rideau, 9e étage
Ottawa, Ontario K1A 0K2

**DANEMARK**
Danish Acquisition and Logistics Organization (DALO)
Lautrupbjerg 1-5, 2750 Ballerup

**ESPAGNE**
SDG TECEN / DGAM
C/ Arturo Soria 289
Madrid 28033

**ETATS-UNIS**
NASA Center for AeroSpace Information (CASI)
7115 Standard Drive
Hanover, MD 21076-1320

**FRANCE**
O.N.E.R.A. (ISP)
29, Avenue de la Division Leclerc
BP 72, 92322 Châtillon Cedex

**GRECE (Correspondant)**
Defence Industry & Research General Directorate, Research Directorate
Fakinos Base Camp, S.T.G. 1020
Holargos, Athens

**HONGRIE**
Department for Scientific Analysis
Institute of Military Technology
Ministry of Defence
P O Box 26
H-1525 Budapest

**ISLANDE**
Director of Aviation
c/o Flugrad
Reykjavik

**ITALIE**
General Secretariat of Defence and National Armaments Directorate
5th Department – Technological Research
Via XX Settembre 123
00187 Roma

**LUXEMBOURG**
*Voir* Belgique

**NORVEGE**
Norwegian Defence Research Establishment
Attn: Biblioteket
P.O. Box 25
NO-2007 Kjeller

**PAYS-BAS**
Royal Netherlands Military Academy Library
P.O. Box 90.002
4800 PA Breda

**POLOGNE**
Centralny Ośrodek Naukowej Informacji Wojskowej
Al. Jerozolimskie 97
00-909 Warszawa

**PORTUGAL**
Estado Maior da Força Aérea
SDFA – Centro de Documentação
Alfragide
P-2720 Amadora

**REPUBLIQUE TCHEQUE**
LOM PRAHA s. p.
o. z. VTÚLaPVO
Mladoboleslavská 944
PO Box 18
197 21 Praha 9

**ROUMANIE**
Romanian National Distribution Centre
Armaments Department
9-11, Drumul Taberei Street
Sector 6
061353, Bucharest

**ROYAUME-UNI**
Dstl Knowledge Services
Information Centre
Building 247
Dstl Porton Down
Salisbury
Wiltshire SP4 0JQ

**SLOVENIE**
Ministry of Defence
Central Registry for EU and NATO
Vojkova 55
1000 Ljubljana

**TURQUIE**
Milli Savunma Bakanlığı (MSB)
ARGE ve Teknoloji Dairesi Başkanlığı
06650 Bakanliklar
Ankara

## AGENCES DE VENTE

**NASA Center for AeroSpace Information (CASI)**
7115 Standard Drive
Hanover, MD 21076-1320
ETATS-UNIS

**The British Library Document Supply Centre**
Boston Spa, Wetherby
West Yorkshire LS23 7BQ
ROYAUME-UNI

**Canada Institute for Scientific and Technical Information (CISTI)**
National Research Council Acquisitions
Montreal Road, Building M-55
Ottawa K1A 0S2, CANADA

Les demandes de documents RTO ou AGARD doivent comporter la dénomination « RTO » ou « AGARD » selon le cas, suivie du numéro de série (par exemple AGARD-AG-315). Des informations analogues, telles que le titre et la date de publication sont souhaitables. Des références bibliographiques complètes ainsi que des résumés des publications RTO et AGARD figurent dans les journaux suivants :

**Scientific and Technical Aerospace Reports (STAR)**
STAR peut être consulté en ligne au localisateur de ressources uniformes (URL) suivant: http://www.sti.nasa.gov/Pubs/star/Star.html
STAR est édité par CASI dans le cadre du programme NASA d'information scientifique et technique (STI)
STI Program Office, MS 157A
NASA Langley Research Center
Hampton, Virginia 23681-0001
ETATS-UNIS

**Government Reports Announcements & Index (GRA&I)**
publié par le National Technical Information Service
Springfield
Virginia 2216
ETATS-UNIS
(accessible également en mode interactif dans la base de données bibliographiques en ligne du NTIS, et sur CD-ROM)

NORTH ATLANTIC TREATY ORGANISATION

RESEARCH AND TECHNOLOGY ORGANISATION

BP 25

F-92201 NEUILLY-SUR-SEINE CEDEX • FRANCE

Télécopie 0(1)55.61.22.99 • E-mail mailbox@rta.nato.int

## DISTRIBUTION OF UNCLASSIFIED RTO PUBLICATIONS

AGARD & RTO publications are sometimes available from the National Distribution Centres listed below. If you wish to receive all RTO reports, or just those relating to one or more specific RTO Panels, they may be willing to include you (or your Organisation) in their distribution.

RTO and AGARD reports may also be purchased from the Sales Agencies listed below.

Requests for RTO or AGARD documents should include the word 'RTO' or 'AGARD', as appropriate, followed by the serial number. Collateral information such as title and publication date is desirable.

If you wish to receive electronic notification of RTO reports as they are published, please visit our website (www.rto.nato.int) from where you can register for this service.

## NATIONAL DISTRIBUTION CENTRES

**BELGIUM**
Royal High Institute for Defence – KHID/IRSD/RHID
Management of Scientific & Technological Research
for Defence, National RTO Coordinator
Royal Military Academy – Campus Renaissance
Renaissancelaan 30
1000 Brussels

**CANADA**
DRDKIM2 – Knowledge Resources Librarian
Defence R&D Canada
Department of National Defence
305 Rideau Street, 9th Floor
Ottawa, Ontario K1A 0K2

**CZECH REPUBLIC**
LOM PRAHA s. p.
o. z. VTÚLaPVO
Mladoboleslavská 944
PO Box 18
197 21 Praha 9

**DENMARK**
Danish Acquisition and Logistics Organization (DALO)
Lautrupbjerg 1-5
2750 Ballerup

**FRANCE**
O.N.E.R.A. (ISP)
29, Avenue de la Division Leclerc
BP 72, 92322 Châtillon Cedex

**GERMANY**
Streitkräfteamt / Abteilung III
Fachinformationszentrum der Bundeswehr (FIZBw)
Gorch-Fock-Straße 7
D-53229 Bonn

**GREECE (Point of Contact)**
Defence Industry & Research General Directorate
Research Directorate, Fakinos Base Camp
S.T.G. 1020
Holargos, Athens

**HUNGARY**
Department for Scientific Analysis
Institute of Military Technology
Ministry of Defence
P O Box 26
H-1525 Budapest

**ICELAND**
Director of Aviation
c/o Flugrad, Reykjavik

**ITALY**
General Secretariat of Defence and
National Armaments Directorate
5th Department – Technological
Research
Via XX Settembre 123
00187 Roma

**LUXEMBOURG**
*See* Belgium

**NETHERLANDS**
Royal Netherlands Military
Academy Library
P.O. Box 90.002
4800 PA Breda

**NORWAY**
Norwegian Defence Research
Establishment
Attn: Biblioteket
P.O. Box 25
NO-2007 Kjeller

**POLAND**
Centralny Ośrodek Naukowej
Informacji Wojskowej
Al. Jerozolimskie 97
00-909 Warszawa

**PORTUGAL**
Estado Maior da Força Aérea
SDFA – Centro de Documentação
Alfragide
P-2720 Amadora

**ROMANIA**
Romanian National Distribution
Centre
Armaments Department
9-11, Drumul Taberei Street
Sector 6, 061353, Bucharest

**SLOVENIA**
Ministry of Defence
Central Registry for EU and
NATO
Vojkova 55
1000 Ljubljana

**SPAIN**
SDG TECEN / DGAM
C/ Arturo Soria 289
Madrid 28033

**TURKEY**
Milli Savunma Bakanlığı (MSB)
ARGE ve Teknoloji Dairesi
Başkanlığı
06650 Bakanliklar – Ankara

**UNITED KINGDOM**
Dstl Knowledge Services
Information Centre
Building 247
Dstl Porton Down
Salisbury, Wiltshire SP4 0JQ

**UNITED STATES**
NASA Center for AeroSpace
Information (CASI)
7115 Standard Drive
Hanover, MD 21076-1320

## SALES AGENCIES

**NASA Center for AeroSpace Information (CASI)**
7115 Standard Drive
Hanover, MD 21076-1320
UNITED STATES

**The British Library Document Supply Centre**
Boston Spa, Wetherby
West Yorkshire LS23 7BQ
UNITED KINGDOM

**Canada Institute for Scientific and Technical Information (CISTI)**
National Research Council Acquisitions
Montreal Road, Building M-55
Ottawa K1A 0S2, CANADA

Requests for RTO or AGARD documents should include the word 'RTO' or 'AGARD', as appropriate, followed by the serial number (for example AGARD-AG-315). Collateral information such as title and publication date is desirable. Full bibliographical references and abstracts of RTO and AGARD publications are given in the following journals:

**Scientific and Technical Aerospace Reports (STAR)**
STAR is available on-line at the following uniform resource locator: http://www.sti.nasa.gov/Pubs/star/Star.html
STAR is published by CASI for the NASA Scientific and Technical Information (STI) Program
STI Program Office, MS 157A
NASA Langley Research Center
Hampton, Virginia 23681-0001
UNITED STATES

**Government Reports Announcements & Index (GRA&I)**
published by the National Technical Information Service
Springfield
Virginia 2216
UNITED STATES
(also available online in the NTIS Bibliographic Database or on CD-ROM)

ISBN 978-92-837-0042-5