

“Roundtrip Architectural Modeling”

Leszek A. Maciaszek

Department of Computing
Macquarie University
Sydney, NSW 2109, Australia

<http://www.comp.mq.edu.au/~leszek/>

Abstract

This paper puts forward a proposition that the production of enterprise information systems must embrace a roundtrip architectural modeling lifecycle. Such a lifecycle begins with the definition of a meta-architecture aimed at minimizing and managing software complexity. It then embraces various supportability metrics to ensure that the implementation conforms to the architectural design and that the resulting system is supportable, i.e. understandable, maintainable, and scalable.

The paper describes the meta-architecture called PCMEF (Presentation, Control, Mediator, Entity, Foundation), explains the dependency management principles in PCMEF, presents selected supportability metrics, describes a tool for the roundtrip development with metrics, defines the overall process for roundtrip architectural modeling, and compares the PCMEF approach with the OMG's MDA (Model Driven Architecture) framework.

Keywords: system development, modeling, architectural design, quality metrics, supportability, roundtrip engineering, UML, PCMEF, MDA.

1 Introduction

According to the title, this paper is about system “modeling” in general, about “architectural modeling” more specifically, and about “roundtrip architectural modeling” in particular. The objective of the paper is to explain and justify an observation that roundtrip architectural modeling is a necessary condition for building *supportable enterprise information systems*. Ignoring this condition is equivalent to building *legacy systems*, which may even do the job when delivered to the users but are going to be a maintenance nightmare.

System development is about *modeling*. Models are abstract representations of reality. Short of staging an untenable argument that a computer program is a reality, software production is all about modeling and working with abstraction. A computer program is the final and most detailed model that executes on a computer.

When faced with a small problem domain, system development can be all about modeling of user *functional requirements*. However, when faced with a large problem domain, such as with an enterprise information system, system development must recognize the demands placed on it by the *non-functional system qualities*. The quality of interest in this paper is *supportability* (Maciaszek and Liong 2003; Maciaszek *et al.* 2004). Supportability is really the combination of three qualities – understandability, maintainability and scalability. An unsupported system is a legacy system.

Delivering supportable systems is conditional on constructing architectural models that provide directions for functional modeling. *Architectural modeling* aims at defining the organization of structural and behavioural components of the system together with *statically visible relationships* and *externally visible interfaces* between these components (Maciaszek, 2004a). Static relationships enable to understand *dependencies* between system components (Maciaszek, 2004b). External interfaces enable to understand the *services* provided by the components.

Finally, architectural modeling must be *roundtrip*, i.e. it must combine forward and reverse engineering processes that ensure that the software product conforms to and is synchronized with the architectural model.

2 A meta-architecture for supportable systems

Production of *supportable systems* has an unpleasant side-effect of restricting the intellectual freedom of system developers. Large systems must be “architected” before they are built and this means that the “builders” (programmers) must obey the architectural design. Software architecture determines the *system qualities*, which define the general goodness of the system (as opposed to its fit-to-purpose).

There are two principal aspects of software architectural design. The first aspect is the definition of a framework (*meta-architecture*) that determines the layers of the (necessary) hierarchical structure and specifies allowed dependencies between the layers. The meta-architecture rules certain high-level implementation choices out and guides various selection decisions and trade-offs. The second aspect focuses on guiding and constraining low-level implementation choices by applying more specific policies and principles. Many of these policies and principles are available as published *design patterns* (Gamma *et al.*, 1995; Fowler, 2003) and are indispensable for production of an architecture-compliant system (Maciaszek 2004a; Maciaszek and Liong 2005).

There is no one unique or best meta-architecture that can result in supportable systems. One such architecture in the Java world is known as the *Core J2EE tiers* (Alur *et al.*, 2003). The meta-architecture, which we advocate and which is nicely aligned with the Core J2EE tiers, is called *PCMEF* (Maciaszek 2005; Maciaszek and Liong, 2005). The *PCMEF* framework “layers” the software into five subsystems of classes – presentation, control, mediator, entity and foundation.

Figure 1 illustrates the PCMEF meta-architecture. The arrows specify allowed dependencies between *subsystems*. The subsystems are UML packages stereotyped as «subsystem». Each subsystem encapsulates intended behavior of the layer. The services that a subsystem provides are defined by its *interfaces* and implemented by its *classes*. For reasons explained later, the PCMEF subsystems can also implement a set of separate interfaces called *acquaintance* (technically, these interfaces are contained in a separate acquaintance subsystem).

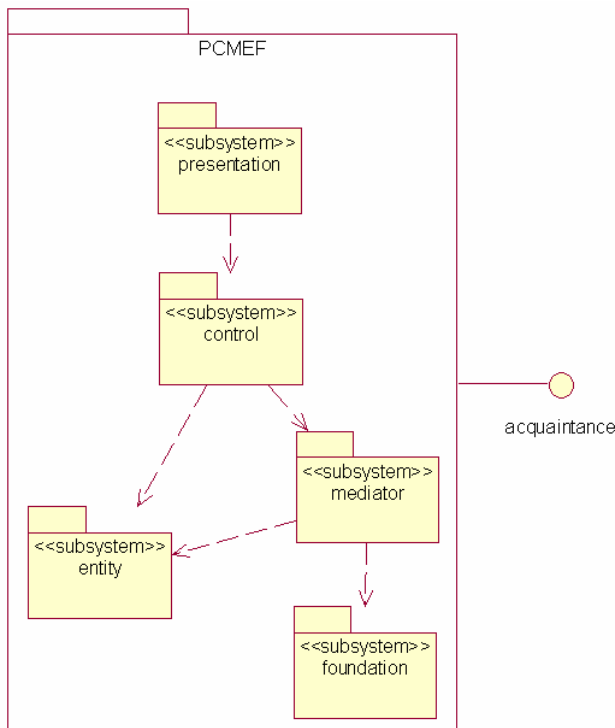


Figure 1: The PCMEF meta-architecture

The PCMEF meta-architecture embraces the separation of concerns introduced in the MVC (model, view, controller) framework (Krasner and Pope 1988) and extends this framework to address the concerns of large enterprise information systems.

The model layer of MVC corresponds to the entity subsystem. *Entity objects* represent the business data objects loaded from the database to the application. The MVC view layer corresponds to the presentation subsystem. *Presentation objects* represent GUI objects that render the state of the entity objects on a graphic display. The MVC controller corresponds to the control subsystem. *Control objects* represent mouse and keyboard events and have an understanding of the

application logic (not to be confused with the business logic).

The remaining two PCMEF subsystems address the client’s “back-end” of the client/server architecture. *Foundation objects* represent database access objects. These are the only client (application) objects that are allowed to communicate with the server (database). *Mediator objects* represent the business logic, i.e. they ensure that the business transactions initiated by the application do not break any enterprise-wide business rules. These are also the objects responsible for the mapping between the entity objects and the database tables (and indeed any other data sources).

3 Supportable dependency management

The *dependency relationships* between PCMEF subsystems are downward. Hence, for example, control depends on entity, but not vice versa. Accordingly, lower-level subsystems should be designed to be more *stable*. This is required because changes to a lower-level subsystem may require changes in the higher-level subsystems (as they depend on lower-level subsystems). The downward dependency requirement is one of the seven PCMEF principles (Maciaszek and Liong 2005). It is known as DDP – Downward Dependency Principle.

The DDP does not prevent objects in lower subsystems to be able to communicate with objects in higher subsystems. The point is that while downward communication is via message passing, the upward communication is required to use *event processing* (disassociated further by means of *interfaces* if the upward communication is not between neighbouring layers). In this scenario, known as UNP – Upward Notification Principle, objects in higher layers act as subscribers (*observers*) to state changes in lower layers. When an object (*publisher*) in a lower layer changes its state, it sends notifications to its subscribers. In response, subscribers can communicate with the publisher (now in the downward direction) so that their states are synchronized with the state of the publisher.

While upward communication can in some well-justified cases span non-neighbouring layers (as just explained), the downward communication must always obey the NCP – Neighbour Communication Principle. The NCP is fundamental for ensuring that the complexity of object inter-communications remains manageable (Maciaszek 2004b) and that it does not disintegrate to an incomprehensible network of intercommunicating objects. To enforce this principle, message passing between non-neighbouring objects uses *delegation* (Gamma *et al.* 1995).

As stated in the Introduction, an important aspect of architectural modeling is the definition of statically visible relationships between objects communicating by means of message passing. This requirement is legitimised in PCMEF in the EAP – Explicit Association Principle. The EAP demands that *associations* are established on all directly collaborating classes. Associations that span layers are unidirectional.

Associations within layers can be both-directional (but require due attention to possible circular dependencies).

In fact, circular dependencies are explicitly addressed by the next principle called CEP – Cycle Elimination Principle. This principle ensures that *circular dependencies* between layers, subsystems and classes within subsystems are resolved. Cycles can be resolved by placing offending classes in a new subsystem/package created specifically for the purpose or by forcing one of the communication paths in the cycle to communicate via an interface.

The simplest but also one of the most effective PCMEF principles is called CNP – Class Naming Principle. The CNP requires that each class name is prefixed with the first letter of the subsystem name (e.g. POrderEntry is a class in the presentation subsystem). The same principle applies to interfaces. Each interface name is prefixed with two capital letters – the first is the letter “I” (signifying that this is an interface) and the second letter identifies the subsystem. The CNP forces developers to produce cohesive classes offering services strictly delineated by the services of the subsystem to which they belong. The CNP turns out to be one of the most effective project management techniques.

Finally, as also stated in the Introduction, an important aspect of architectural modeling is the definition of *externally visible interfaces* between components. This requirement is tacitly enforced in all communication between subsystems (by definition of the subsystem – ref. Section 2). However, there is a more explicit dimension to this requirement. It is known as the APP – Acquaintance Package Principle. The PCMEF meta-architecture provides for an acquaintance subsystem, consisting only of *interfaces* that an object passes in arguments to method calls, instead of concrete objects. These interfaces can be implemented in any PCMEF subsystem. This effectively allows communication between non-neighbouring subsystems while centralizing dependency management to a single acquaintance subsystem.

4 Measuring supportability

The PCMEF meta-architecture together with its seven principles defines a desired model to produce supportable systems. However, the meta-architecture is a theoretical objective which may or may not be fulfilled in practice. Also, it is possible to have multiple designs (and corresponding implementations), all of which conforming to the meta-architecture, yet exhibiting various levels of “goodness”. What we need is to be able to measure how “good” a particular software solution is and whether or not it conforms to the meta-architecture.

This Section explains the PCMEF approach to measuring supportability. It explains the notions of *cumulative class dependency* and *unsupportability factor* (Maciaszek and Liong 2003). Related papers address metrics for message dependencies (Liong and Maciaszek 2003a) and for event dependencies (Liong and Maciaszek 2003b).

From a perspective of a system architect and maintainer, the dependencies between classes provide the most valuable metric of system complexity and supportability. It is, therefore, important to make these dependencies explicit and to uncover any hidden dependencies. The *Cumulative Class Dependency (CCD)* is a measure of the total number of class dependencies in a system.

DEFINITION: *Cumulative Class Dependency (CCD)* is the total supportability cost over all classes $C_{i(i=1,\dots,n)}$ in a system of the number of classes $C_{j(j<=1,\dots,n)}$ to be potentially changed in order to modify each class C_i .

The *CCD* definition is intentionally simple. In particular it does not, by itself, judge the quality of the design. Its value is in comparisons between two or more designs for the same system. To this aim, the *CCD* computation strives to validate if a particular design conforms to a chosen meta-architecture (such as PCMEF). Uncovering a class dependency that invalidates the architectural framework leads to the only sensible assumption that the required dependency structure in the system is broken. This in turn means that any dependency is possible and the system supportability has eluded management controls.

The calculation of *CCD* for a particular design starts by assuming an adherence to the architectural framework. If the framework is found to be broken, the *CCD* is calculated as if a class can depend on any other class in the system. The *CCD* is then computed using a probability theory method, namely the *combinations counting rule*. The *CCD* is the number of different combinations of pairs of dependent classes which can be formed from the total number of classes in the design multiplied by 2. The multiplication by 2 accounts for the possibility that a dependency between any pair of classes can be cyclic (i.e. in both directions).

$${}_n CCD_2 = \frac{n!}{2!(n-2)!} \times 2$$

The above formula produces the worst possible *CCD* for a design. It is a penalty for the design that leaves no certainty as far as the supportability of any class is concerned. A change to any class in a system can potentially impact on all remaining classes, and therefore all remaining classes must be subject to monitoring, checking, and possibly modifying. This results, therefore, in an *unsupportable design/system*. The comparison of this worst possible *CCD* with the *CCD* for an architecturally-compliant design leads to so called *Unsupportability Factor (UF)*.

DEFINITION: *Unsupportability Factor (UF)* is the result of the division of the *CCD* for an unsupportable system by the *CCD* for a corresponding supportable system, i.e. the system that conforms to a supportable meta-architecture, such as PCMEF.

Consider the PCMEF design in Figure 1 and assume that there is only one class in each of the five subsystems. The *CCD* for a supportable system conforming to such design is the sum of dependency arrows in Figure 1, i.e. the *CCD* is 5. For a corresponding unsupportable system, the *CCD* would be 20:

$${}_5CCD_2 = \frac{5!}{2!(5-2)!} \times 2 = \frac{120}{12} \times 2 = 20$$

The UF is therefore $20/5 = 4$. The UF factor serves as a modifier of the more detailed metrics computed for designs/systems that were found to be unsupported. Two of these more detailed metrics are Cumulative Message Dependency (*CMD*) and Cumulative Event Dependency (*CED*) (explained in Liong and Maciaszek (2003a) and in Liong and Maciaszek (2003b), respectively).

The *CCD* measures the complexity of static structures between classes in a system. To get a meaningful value for such a complexity metric, the meta-architecture must demand the enforcement of the EAP principle (that forces static association links to exist between classes exchanging messages). The individual dependency values (weights) may be assumed to be always 1 for each association link between classes in the system. These dependency weights can be changed, if desired, to emphasize/de-emphasize different scenarios for metrics calculation. For example, we could use value 1 for any association link between classes in the same subsystem and value 2 if the classes are in different subsystems. Better yet, we could use a value of 2 for association links between classes in *neighbouring* layers and use the UF factor as the value for those association links that violate architectural constraints.

This said, the *CCD* alone says nothing about the number and frequencies of using association links for message passing and event processing. It also says nothing about the length of delegation links when a service request passes continuously between objects until it reaches the object that has the method (code) to perform the service. The *CMD* and *CED* metrics shed some light on how we can measure the dynamic behaviour of programs and establish if the system is supportable as far as the analysis of its runtime behaviour is concerned.

5 Roundtrip architectural modeling with metrics

The architecture-driven approach to system development has a *forward-engineering* flavour. The forward-engineering approach offers an architectural design that minimizes the dependencies. The aim is to implement a software product that minimizes dependencies by imposing an architectural solution on programmers.

However, a successful production of a supportable system requires a parallel contribution from a *reverse-engineering* approach. The reverse-engineering approach aims at measuring dependencies in implemented software. The implementation may not conform to the desired architectural design. If it does not, the aim is to compare the metric values in the software with the values that the desired architecture would have delivered. The troublesome dependencies can then be pinpointed.

Measuring supportability of designs and programs cannot be done manually. Maciaszek and Liong (2003) describes a tool, called *DQ* (Design Quantifier), which is able to analyse any Java program, establish its conformance with

a chosen supportable meta-architecture, compute complete set of dependency metrics, and visualize the computed values in UML class diagrams. This section is a short description of *DQ* and its place in roundtrip architectural modeling with metrics.

DQ calculates cumulative class, message and event dependency values for each class in the subsystem, for the subsystem as the whole and for the entire system. It highlights any non-conformances that it discovers. The non-conformance is traced to an *offending data member*, *offending method* and/or *offending class*.

DQ is able to reverse-engineer any Java program and discover the code dependencies. Class dependencies represented as associations are easily noticeable both in the system design and in the code. However, *implicit dependencies* such as acquaintance and local variables of methods are difficult to see. These implicit dependencies (typically on the level of message and event dependencies) may introduce new class dependencies that need to be presented to the system designer.

DQ performs an *architectural conformance analysis* (i.e. if the design, obtained from the code or otherwise, conforms to the architectural design). Currently, *DQ* accepts the *PCMEF* meta-architecture as the reference model for conformance analysis but the tool can be configured to suit other frameworks provided the definition of the framework is given to it. The information that is required includes the notions of neighbouring subsystems, naming conventions used to identify methods or classes of significance (such as an event class), and the weights of association links and any dynamic links. With these values provided, *DQ* can calculate the dependency metrics for a system and derive its comparative supportability indicator.

DQ groups classes by their subsystems to allow subsystem-level metrics to be calculated. Information regarding neighbouring subsystems is used to identify *offending classes*. An offending class contains at least one explicit (i.e. supported by an association) or implicit dependency that does not conform to the given meta-architecture. In the case of *PCMEF*, this conformance includes the principles explained in Section 3.

DQ supports roundtrip engineering by making the retrieved information available to a visual modeling tool, such as IBM Rational Rose. *DQ* can provide a Rose script to generate UML notes containing metric values and attach them to corresponding classes in the diagram.

Figure 2 presents a model reverse-engineered from a simple Java program. The model shows (in UML notes) the cumulative metrics for the classes. The metrics are: *CCD* – cumulative class dependency; *CMD* – cumulative message dependency, and *CED* – cumulative event dependency.

Using the UML notes to display metrics is a visually unattractive technique but the only one readily supported by IBM Rational Rose. A better solution would be to use the idea of Together Control Center and introduce the fourth compartment in classes specifically for the metrics.

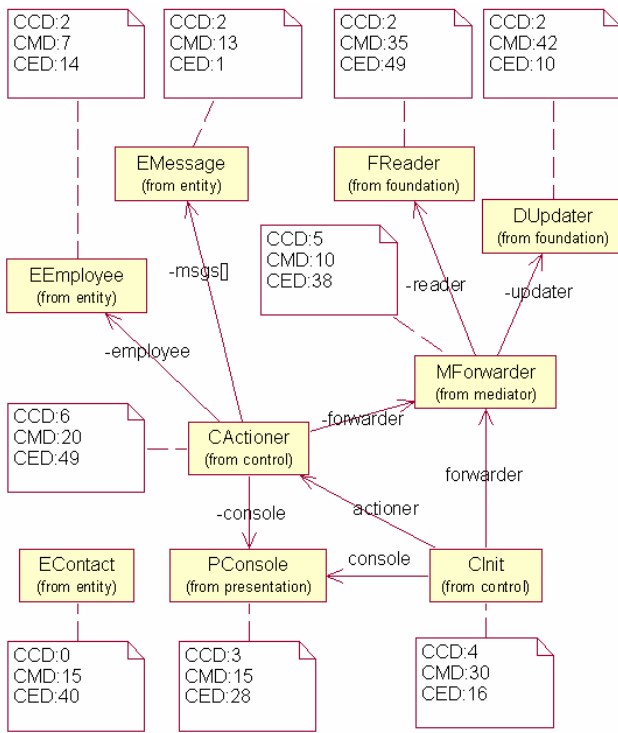


Figure 2: A model reverse-engineered by DQ

Although not supported by DQ, tools like DQ should be able to visualize dependencies by producing *call graphs*. Ideally, a call graph could be a variant of a UML sequence diagram. A call graph can be used for the change impact analysis and to answer “what-if” questions such as “which methods are affected if a particular method is modified?”.

Figure 3 shows the SmallWorlds’ “what-if” analysis performed on a reverse-engineered Java program. The dependencies originated in the class *D_Updater* are shown. (Note that SmallWorlds was bought by IBM and developed into a new tool called Structural Analysis for Java (SA4J)).

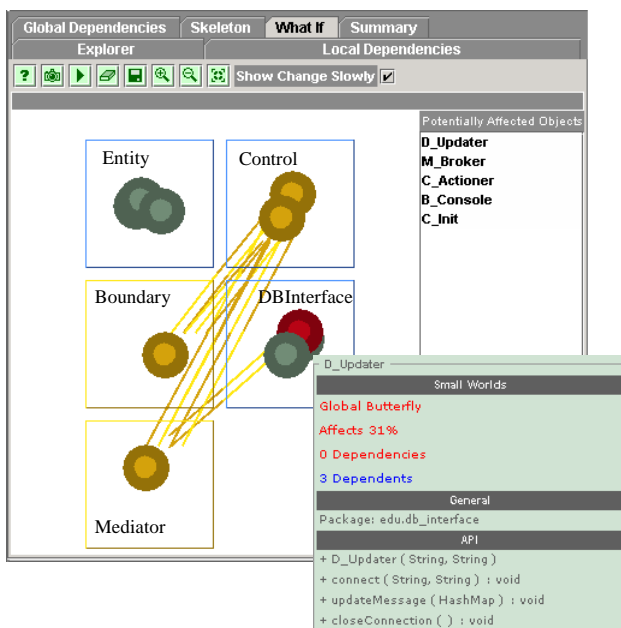


Figure 3: SmallWorlds’ “what-if” visualization

6 Process of roundtrip modeling

Figure 4 is a statechart model for the process of roundtrip architectural modeling discussed in this paper. The entry point to the roundtrip process is the existence of Meta-Architecture (the PCMEF framework in our case). The meta-architecture guides and constrains the architectural design, but it is not itself the design. A meta-architecture is a reuse technology from which the design event obtains Architectural Design Model for a system under development.

The verify conformance event supported by the modify action ensures that the supportability principles of Meta-Architecture are consistently observed by Architectural Design Model. The design transition applied on Architectural Design Model results in the next system state called Detailed Design Model. The detailed design model is a UML-based specification of system structure and behaviour. The model is subjected to the measure supportability event and the modify action to arrive at the desired “goodness” level for the supportability quality.

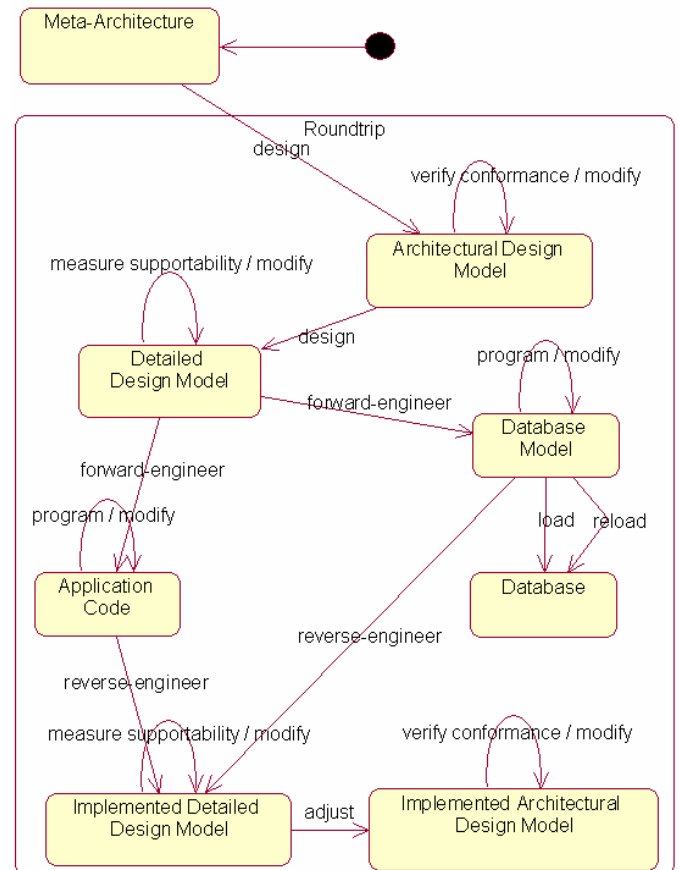


Figure 4: Statechart of roundtrip modeling

Once Detailed Design Model is sufficiently refined to be projected onto the execution platform, the forward-engineer events are “fired” to result in the Application Code and Database Model states. The Database Model state creates a database schema

according to which the load event creates the Database extension (content).

Naturally enough, the generated (forward-engineered) code is subjected to further programming and refinements in the sequences of program/modify transformations. In the case of Database Model, any more significant change of the model necessitates the reload event on the Database state.

The above completes the forward part of the roundtrip modeling cycle and would normally result in the “internal delivery” of the first *iteration* within the iterative and incremental software development lifecycle. Before the iteration can be delivered to customers, the reverse-engineer events are required for abstracting models of existing implementations.

First the Implemented Detailed Design Model is obtained in a “mining” process that is hard to be fully automated. The DQ tool described in Section 5 serves the purpose of “mining” from Application Code. The “mining” process from Database Model, which includes any triggers and stored procedures, is described in Maciaszek (2002).

To close the roundtrip modeling loop, the Implemented Detailed Design Model is subjected to the measure supportability event and the modify action so that the supportability quality of the model is ensured. Finally, the adjust event results in a new Implemented Architectural Model state. This state closes the first roundtrip cycle and begins the next one.

7 Comparison of PCMEF and MDA

A careful reader would have noticed parallels between the PCMEF approach to roundtrip architectural modeling and the MDA (Model Driven Architecture) lifecycle framework from OMG (Object Management Group) (MDA 2001). It is, therefore, proper to compare the two approaches.

The MDA framework attempts to take modeling to its next natural state – *executable specifications*. The idea of executable specifications, i.e. turning formal specification models into executable code, has been entertained in software engineering from the beginning of the discipline, but the MDA takes advantage of the existing standards and modern technology to make the idea happen.

In the MDA, a system *specification* must be *formal* in order to be treated as a *model*. “A specification is said to be formal when it is based on a language that has a well-defined form (“syntax”), meaning (“semantics”), and possibly rules of analysis, inference, or proof of its constructs.” (MDA, p.3).

According to the MDA: “source code is a model that has the salient characteristic that it can be executed by a machine” and “a UML-based specification is a model whose properties can be expressed graphically via diagrams, or textually via an XML document” (MDA, p.4).

The MDA is a *transformation model* that treats systems development as a sequence of transformations from the formal specification for the domain (PIM – Platform Independent Model), via the detailed specification targeting a particular hardware/software platform (PSM – Platform Specific Model), to an executable code. If the system under development spans multiple platforms, the MDA architecture stipulates multiple PSMs linked by *PSM interoperability bridges*. Multiple PSMs transform to multiple codes, linked by *code interoperability bridges*.

Table 1 compares the PCMEF and the MDA approaches on a set of nine criteria points. The first two criteria relate to the support for the roundtrip software development. Both approaches recognize and support the combined use of forward and reverse engineering transformations. The MDA uses the name “mapping” for forward transformations and the term “refactoring” for reverse transformations.

	Criteria	PCMEF	MDA
1	Forward-engineering transformations.	Yes	Yes
2	Reverse-engineering transformations.	Yes	Yes
3	Driven by a meta-architecture aimed at complexity minimization and management.	Yes	No
4	Based on metrics to ensure system qualities, such as supportability.	Yes	No
5	Models are built, viewed, and manipulated via UML.	Yes	Yes
6	Supports other than UML formal specification technologies, including XML and UML profiles.	No	Yes
7	Supports interoperability bridges between platforms.	No	Yes
8	Supports definition at the platform independent level of pervasive services, such as directory, persistence, transaction, or security services.	No	Yes
9	Supports standardized domain specifications for specific vertical markets, such as for healthcare or telecommunication industry.	No	Yes

Table 1: Comparison of PCMEF and MDA

In the context of the roundtrip development, the MDA emphasizes the importance of *UML profiles* to ensure the production of formal platform specific models. It also

attempts to address the current weaknesses in UML to relate models at different levels of abstraction.

The next two criteria (3 and 4) capture the essence of the PCMEF approach and its objective to provide for the production of supportable systems in an architecture-driven and metrics-driven ways. These features are missing in the MDA.

The criteria 5 and 6 confirm that both the PCMEF and the MDA rely in the roundtrip tasks on models expressed using UML. However, the MDA goes further by providing support for UML profiles and for expressing models in non-graphical forms, such as in XML.

The last three criteria (7, 8, and 9) capture the essence of the MDA approach. The MDA is an industry-strength approach with the objective to set standards for software production in the future. Accordingly, it addresses such weighty issues as software interoperability, pervasive services, and solutions for vertical industries.

8 Conclusion

As software systems grow in complexity, the software developers have been learning valuable lessons about building complex enterprise information systems. As observed in the seminal paper by Brooks (1987), software is *complex by its essence*. The only thing that we can do is to ensure that we do not add to that inherent complexity through bad software development practices.

This paper defined what we believe is a good software development practice. The practice described in this paper (the PCMEF approach) results in building *supportable software systems* with manageable complexity. This is achieved by a set of integrated measures referred to as the *roundtrip architectural modeling*.

The PCMEF approach is backed by years of the author's consulting experience. The approach has been validated on multiple software projects and has been used in case studies presented in the author's books (Maciaszek 2005; Maciaszek and Liong 2005). Despite this, the roundtrip architectural modeling of the PCMEF approach is predominantly a *conceptual know-how* rather than a fully instrumented technology. Yet, as a widely recognized truth says – "good designs come from good designers". Good designers have the know-how. The technology takes always the background stage.

9 References

- Alur, D. Crupi, J. and Malks, D. (2003): *Core J2EE Patterns: Best Practices and Design Strategies*, 2/e, Prentice Hall, 528p.
- Brooks, F.P. (1987): No Silver Bullet: Essence and Accidents of Software Engineering, *IEEE Software*, 4, pp.10-19; reprinted in: *Software Project Management. Readings and Cases* (1997), ed. C.F.Kemerer, Irwin, pp.2-14.
- Fowler, M. (2003): *Patterns of Enterprise Application Architecture*, Addison-Wesley, 531p.
- Gamma, E. Helm, R. Johnson, R. and Vlissides, J. (1995): *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison-Wesley, 395p.
- Krasner, G.E. and Pope, S.T. (1988): A Cookbook for Using the Model View Controller User Interface Paradigm in Smalltalk-80, *J. Object-Oriented Prog.*, Aug-Sept, pp.26-49.
- Liong, B.L. and Maciaszek, L.A. (2003a): Computing Message Dependencies in System Designs and Programs, *ICEIS'2003 Fifth Int. Conf. on Enterprise Information Systems*, Volume III, ICEIS Press, Angers, France, April, pp.619-622
- Liong, B.L. and Maciaszek, L.A. (2003b): Computing Event Dependencies in System Designs and Programs, *CAiSE'03 Forum Proceedings Short Papers*, University of Maribor Press, Velden, Austria, June, pp.189-192
- Maciaszek, L.A. (2002): Process Model for Round-Trip Engineering with Relational Database, in: *Software Reengineering*, ed. S. Valenti, IRM Press, pp.76-91
- Maciaszek, L.A. and Liong, B.L. (2003): Designing Measurably-Supportable Systems, *Advanced Information Technologies for Management, Research Papers* No 986, ed. by E. Niedzielska, H. Dudycz, M. Dyczkowski, Wroclaw University of Economics, pp.120-149
- Maciaszek, L.A. (2004a): Architectural Patterns for Designing and Refactoring Information Systems for Large Enterprises, *VII Forum of Information Technology, XX Autumn Meetings of Polish Society of Information Processing*, November 2004, Mragowo, Poland, 11p. (invited paper)
- Maciaszek, L.A. (2004b): Managing Complexity of Enterprise Information Systems, *ICEIS'2004 Sixth Int. Conf. on Enterprise Information Systems*, Volume I, INSTICC, April 2004, Porto, Portugal, pp.IS-9-13 (invited paper)
- Maciaszek, L.A. Cheng, A.M.K. and Sabloniere, P. (2004): Evolution of Enterprise Information Systems in the Internet Era: Contributions and Limits of New Technologies and Architectures, *Enterprise Information Systems V*, eds.: O. Camp, J.B.L. Filipe, S. Hammoudi, M. Piattini, Kluwer Academic Publishers, pp.1-4 (invited paper)
- Maciaszek, L.A. (2005): *Requirements Analysis and System Design*, 2nd ed., Harlow England, Addison-Wesley, 504p.
- Maciaszek, L.A. and Liong, B.L. (2005): *Practical Software Engineering. A Case-Study Approach*, Harlow England, Addison-Wesley, 864p.
- MDA: Model Driven Architecture (MDA), Document number ormsc/2001-07-01, Architecture Board ORMSC, http://www.iosoftware.com/as_support/brochures/executive_overview.pdf Accessed 3 November 2004