Modeling and Engineering Adaptive Complex Systems

Leszek A. Maciaszek
Macquarie University, Department of Computing,
NSW 2109, Sydney, Australia
leszek@ics.mq.edu.au

Abstract

This paper describes a strategy for modeling and engineering complex e-business systems with built-in quality of adaptiveness. The paper explains the philosophical and scientific foundations for research findings and for the proposed strategy. After ascertaining that complexity relates to the fact that higher levels of organization manifest features not predictable from the lower ones, the paper uses a holonic approach to science that reconciles reductionism and holism. The strategy is centered on a six-layer meta-architecture (called PCBMER and described comprehensively elsewhere). The meta-architecture facilitates development of adaptive systems such that property of emergence is controlled and supplanted by the property of resultance.

Keywords: Software system, complex system, adaptive system, software engineering, system modeling, reductionism, holism, holon, holarchy.

1 Epigraph

This paper is about a strategy for the development of large software systems. It is about a scientific vision for software development (synthesized from many years of research and experience) rather than a targeted (delta) research contribution proving a theory or solving a particular problem. As such, a usual 'Introduction' is replaced here by an 'Epigraph', which should give the reader some idea of what this piece or writing is about.

This paper is set against the background of *reductionism* and *holism* as two contrasting approaches to science (Looijen 2000, Kanitscheider 2002, Jackson 2003, Capra 1982, Koestler, Smythies 1969). It makes a strong call for the importance of a middle-ground *holonic* approach (Koestler 1967, Koestler 1978, Koestler 1980) as the most promising way to understand and take control of the *complexity* of large software systems. In doing so, this paper provides further philosophical and pragmatic arguments for *holonic software architectures* advocated by the author in (Maciaszek 2007b, Maciaszek, Liong 2005, Maciaszek 2007a, Maciaszek 2006) and elsewhere.

A holonic architectural design provides a framework on which to build quality into software, and in particular to achieve the overriding quality of *adaptiveness*.

This paper takes on the task on convincing the reader that to build complex software systems, which can adapt to the ever changing requirements and environment, an intellectual freedom of 'cutting code' has to be curtailed. The studies of structure and behavior of natural systems provide guidance for construction of human-made systems. Even if this guidance is incomplete or controversial, it gives an explanation to software implementation and provides a hope that the resulting system will be understandable, maintainable and scalable. Without this assurance, and in the face of ever increasing dependence of people on software solutions, the consequences on every-day lives of people may be indeed serious.

To cover the breadth of the subject area and to address a large number of interdependent concepts and theories, the paper takes a 'build-up' approach of explaining terms in the title starting from the least descriptive last term ("systems") and going backwards to the most descriptive terms of "engineering' and "modeling". Accordingly, the following sections of the paper are titled:

- Systems
- Complex (i.e. complex systems)
- Adaptive (i.e. adaptive complex systems)
- Modeling and Engineering (of adaptive complex systems)

2 Systems

From all the terms in the paper's title, the concept of *system* is the least contentious. The general consensus is that a system is "a complex whole the functioning of which depends on its parts and the interactions between those parts" (Jackson 2003, p.3). Moreover, a system is "an integrated whole whose properties cannot be reduced to those of its parts" (Capra 1982, p.26).

While the first definition is neutral with regard to scientific methods of studying systems, the second definition places the concept firmly within the realms of *holism*. Pivoted on the idea that "the whole is greater than the sum of the parts", holism searches for new qualities and complex higher-level phenomena at the macroscopic level and it claims credibility on the basis of *emergence* alone. Holistic views in the philosophy of science have conquered various sectors in physical sciences (quantum mechanics, deterministic chaos), biological sciences (evolutionary biology) and elsewhere. Holistic views

have also made a considerable footprint in information and communication sciences (Jackson 2003).

Despite its successes, holism faces a stiff opposition from traditional scientific quarters, which base their scientific methods on the Cartesian philosophy of *reductionism* (*atomism*). Propelled by undeniable successes (most notably in the molecular biology research that has resulted in the unraveling of the genetic code), reductionists treat holism as a form of art rather than science. Theories that cannot be explained in reductionist terms (i.e. by reducing the problem to elementary parts and by synthesizing through the analysis of these parts and their interactions) are deemed unworthy of scientific investigation. Yet, the limitations of such analytic or mechanistic world view are evident across all sciences, including biology (in particular with its relation to medicine).

With regard to the uncontroversial aspects of the definition of system, i.e. by looking at system as a complex whole, we have to acknowledge that systems theory is all-embracing, interdisciplinary and even transdisciplinary. Information systems, living organisms, societies, and ecosystems are all systems. From this paper's perspective, it is therefore valuable to try to derive the desired characteristics of human-made systems, such as business systems, from the studies of structures and behavior of natural systems, in particular the most complex systems that exist – living organisms.

Fig.1 shows a classification of business systems in a Venn diagram. An *e-business system* is understood as a software system (an integrated set of software applications). Most software systems are, or strive to be, network- and Internet-enabled. It, therefore, makes sense to use the concept of *e-business* to mean: "the use of the Internet and other networks and information technologies to support electronic commerce, enterprise communications and collaboration, and Web-enabled business processes, both within a networked enterprise and with its customers and business partners" (O'Brien, Marakas 2007, p.234)

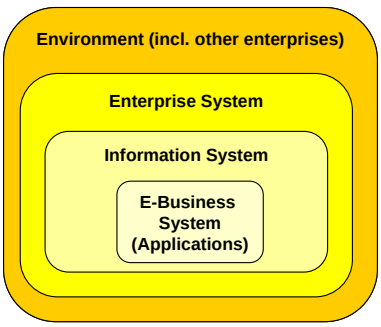

Fig. 1. Systems in the world of business.

In general, e-business refers to any commercial transactions involving the exchange of value (e.g. money) across organizations and individuals in return for products or services. It includes, therefore, consumer-to-consumer (C2C) e-business, such as an auction site, and peer-to-peer (P2P) exchanges in which products or services are openly shared. E-business includes also systems based on a service-oriented architecture (SOA). SOA systems differ from systems that merely use web services in that SOA advocates the use of a services discovery mechanism (a service broker or discovery agent).

An *information system* is concerned with generating and managing information for people. Some of this information is generated automatically by software systems. Other information is obtained manually by people. The point is that information systems are social systems that encompass and use software and other components such as: people, information, procedures, hardware, and personal and automated communications.

An implication is that the development of a software system is just an activity (albeit a fundamental one) in the development of an information system. An information system operates within the context of an *enterprise system* – an organization (perhaps a virtual organization) created under applicable laws for a business endeavor. The enterprise is further constrained by the rules and conditions of the business *environment*. The environment includes other enterprises that may become the collaborating partners in a networked organization. At this layer, a business to business (B2B) integration takes place.

Placing software development in the context of enterprise means that a software process is derived from a wider business model and it tries to support and implement a particular business process in that model. This means that a software product/service cannot be just an information service. It should also implement and assist in business actions. The design of an information system should either explicitly identify a business process it serves or, better, it should be a part of a (business) knowledge management system. One aspect of such design is to coordinate automated informational actions, manual supportive actions, and creative decision making actions.

Usually, a particular software system services a single management level – operational, tactical or strategic. The *operational level* is concerned with processing business operational data and documents, such as orders and invoices. This is the realm of OnLine Transaction Processing (OLTP) systems assisted by conventional database technology. The *tactical level* processes information obtained from the analysis of data, such as monthly trends in product orders. This is the realm of OnLine Analytical Processing (OLAP) systems assisted by data warehouse technology. The *strategic level* processes the organizational knowledge, such as rules and facts behind a highly profitable product selling. This is a realm of knowledge systems assisted by knowledge base technology.

Systems at operational management level are indispensable to the enterprise. Without them, a modern enterprise cannot function. However, operational software does not provide to the enterprise any competitive edge. Competitors already have similar systems. The business value of software increases with

increasing levels of management to which the system applies.

3 Complex

Systems are complex by their very definition. But what do we really mean by complexity? While there have been many attempts to define complexity in absolute terms, we tend to agree with propositions that '*complex*' is a primitive and relative term, which can only be given a contextual definition. As such, 'complex' can only be understood by its relation to its specific contrary notion of 'simple'. There are many primitive concepts like that, e.g. 'part' as contrary to 'whole', 'same' as the opposite of 'different' (Agazzi 2002).

If the notion of complexity (as well as simplicity) is not absolute but relative, then what is complex from one point of view may not be complex from another point of view. Short of stating that the complexity is in the eye of the beholder, we can identify four kinds of software complexity:

- *Problem complexity* – the complexity of the problem domain itself. This is also known as *computational complexity*. Problem complexity is an offshoot of the Brooks' essential characteristics of software, i.e. the four difficulties of software production that are not amenable to breakthroughs or 'silver bullets' (the inherent software complexity, conformity, changeability, and invisibility) (Brooks 1987).
- *Algorithmic complexity* – aiming at measuring the efficiency of software algorithms. This is a kind of complexity with diminishing relevance due to the shift of computing paradigm from algorithms to interactions. Unlike algorithmic systems, interactive systems can learn and adapt and as a result can produce outputs that are only partially determined by their inputs (Wegner 1997).
- *Cognitive complexity* – measuring the effort required to understand the software.
- *Structural complexity* – aiming at establishing the relationship between the structure of the software and the ease of its maintenance and evolution. The measurements are applied to control flow structures, hierarchical structures, modular structures, etc.

If the problem complexity is an 'essence' of software production and the algorithmic complexity is an old hat, then the last two kinds of complexity must take priority in considerations related to modeling and engineering of e-business systems. A closer look at these two kinds of complexity reveals that the cognitive complexity is a necessary condition of structural complexity. The structural complexity subsumes cognitive complexity. Accordingly, in what follows *complexity* is understood as structural complexity.

The complexity of software systems is *in the wires* – in the linkages and communication paths between software objects. The "wires" create *dependencies* between distributed objects that may be difficult to understand and manage (a software object A depends on an object B, if a change in B necessitates a change in A).

The realization that the ways objects are interconnected and integrated are more important than the objects themselves places software systems on the holistic end of scientific investigation. The resulting whole is more than the sum of its parts. This also places software systems firmly within the context of general systems theory. "Systems theory looks at the world in terms of the interrelatedness and interdependence of all phenomena, and in this framework an integrated whole whose properties cannot be reduced to those of its parts is called a system." (Capra, p.26).

These observations bring us to another point about the nature of complexity – the difference between 'compound' and 'complex'. "… in a *compound* we have a plurality of components, but are not concerned about their relations, whereas a *complex* is a compound in which the relations among its constituents are significant, since they make of this compound a *whole* endowed with an identity and evincing an analytical complexity." (Agazzi 2002, p.7).

Another way of looking at this point is by distinguishing between analytic and synthetic simplicity (Dilworth 2001), and then by counter-supposition distinguishing between analytic and synthetic complexity. An object is analytically simple if it has no *internal relations* and it is synthetically simple if it has no *external relations*. Vice versa, an object is analytically complex if it has internal relations and it is synthetically complex if it has external relations.

Regarding the definition of system (Section 2), a *whole* is the effect of synthetically complex parts. Thus, a *part* is anything that is either analytically and synthetically simple or analytically complex but synthetically simple. In the context of a software system, the "wires" express the relations, both internal and external. An analytically complex object is considered a part if its internal relations are *encapsulated* (as per the object-oriented software engineering paradigm). Otherwise it is a whole. We can say that a part becomes a whole when its internal relations (if any) are externalized (un-encapsulated).

This line of reasoning, when applied to natural systems, has led Arthur Koestler (Koestler 1967, Koestler, Smithies 1969, Koestler 1978, Koestler 1980) to the notion of *holon* (from the Greek word: 'holos' = whole and with the suffix 'on' suggesting a part, as in neutron or proton). A holon is an object that is both a whole and a part, and which exhibits two opposite tendencies: an integrative tendency to function as part of the larger whole, and a self assertive tendency to preserve its individual autonomy. Looking downward, a holon is something complete and unique, a whole. Looking upward, a holon is an elementary component, a part.

Like the entire notion of complexity, the notion of holon is placed within the context of an *order* or a *structure*. "A living organism is not an aggregation of elementary parts, and its activities cannot be reduced to reeling off a chain of conditioned responses. In its bodily aspects, the organism is a whole consisting of "sub-wholes", such as

the circulatory system, digestive system, etc., which in turn branch into sub-wholes of a lower order, such as organs and tissues - and so down to individual cells. In other words, the structure and behaviour of an organism ... is a multi-levelled, stratified hierarchy of sub-wholes, ... where the sub-wholes form the nodes, and the branching lines symbolise channels of communication and control." (Koestler 1980, p.447). "Generally speaking, a holon on the /n/ level of the hierarchy is represented on the /n+1/ level as a unit and triggered off as a unit. Or, to put it differently: the holon is a system of relations which is represented on the next higher level as a unit, i.e., a relatum." (Koestler 1967, p.72).

A stratified hierarchy of holons is called by Koestler a *holarchy* to distinguish it from a network, but also from a hierarchy. Clearly, a holarchy is a kind of hierarchy for otherwise the very containment of a part in any whole cannot be defined and understood. What is special about a holarchy is dispensing with any traces of ranking or dominance between holons. A holarchy is not linear in nature. It is rather a nested conception, a composition or containment. As observed by Wilber (1995), a whole contains parts in a way reminiscent of what can be seen in one mirror in a house of mirrors.

A holarchy seems to be a hint given by nature for how to develop and manage complex human-made systems. The various stratified layers are stable holons of differing complexities and with a degree of autonomy that enables them to adapt to new circumstances and to changes in the environment. "Nonstratified systems, on the other hand, would totally disintegrate and would have to start evolving again from scratch. Since living systems encounter many disturbances during their long history of evolution, nature has sensibly favored those which exhibit stratified order. As a matter of fact, there seem to be no records of survival of any others." (Capra 1982, p.304).

4 Adaptive

In the Epigraph Section, we alluded to the definition of *adaptiveness* as a trio of concepts – understandability, maintainability, and scalability (evolution). In the previous Section, we concentrated on the notion of complexity in its interpretation of structural complexity (that in turn subsumes cognitive complexity). Complexity in this sense refers to the level of ease (or difficulty) associated with the same trio of concepts as in the definition of adaptiveness.

Clearly, adaptiveness and complexity are two sides of the same coin. Adaptiveness is a desirable quality of a complex system. A quality that should be first built into the system and then managed. An *adaptive system* has an ability to change to suit different conditions; an ability to continue into the future by meeting existing expectations (requirements) and by adjusting to accommodate any new and changing requirements. Adaptiveness is a complexity management notion.

As for the complexity issue, to understand the various dimensions of adaptiveness (or adaptation), a reference to natural systems is proper. Living organisms seem to possess three levels of adaptation: reversible, somatic, and geno-typic (Capra 1982).

A *reversible* adaptation is a temporal change due to a short-term stress on an organism. Hangovers after drinking too much alcohol or initial symptoms associated with an ascending to a high altitude are examples of reversible adaptations. In software terms, any "stress' on the program resulting in error or exception conditions is a reversible adaptation.

A *somatic* adaptation is a change in an organism due to a long-term stress. Although still reversible, a somatic change is a physiological response of an organism aimed at absorbing the environmental impact. Addiction and acclimatization are somatic changes. In software terms, all forms of maintenance (including 'perfective maintenance') are somatic adaptations.

A *geno-typic* adaptation refers to the change in the genetic makeup of an organism. Such a change is irreversible within the lifetime of an organism. It is a change to the lowest levels of a holarchy and to the most 'stable' holons – cells, organelles, molecules. Adaptation of the species (evolution) is a geno-typic adaptation. In software terms, a re-design and re-implementation of the system reaching to the majority of its smallest components while retaining its architectural backbone is a form of the geno-typic adaptation.

In passing, we described the notion of a complex system in terms of emergent complex behaviour that they exhibit. Therefore, a question to be asked is how 'adaptive' relates to 'emergence'. In complexity theory, *emergence* is used "to indicate the presence of properties that can *not* be explained as the consequence of the properties of the analytic simples." (Agazzi 2002, p.9).

If something cannot be explained then it cannot be managed. Yet, 'adaptive complexity' refers to a complex system that is managed to exhibit the quality of being adaptable. Emergence is a feature of a complex system, but it is a feature that needs to be controlled and suppressed in adaptive complex systems. A permitted feature in adaptive systems is *resultance* defined as the "properties of the whole that are produced by properties of the analytic simples by virtue of certain internal relations of the whole." (Agazzi 2002, p.9).

Allowing resultance and disallowing emergence in software systems excludes certain more advantageous systems from consideration. Most notably it excludes *multi-agent systems* in which dynamic agent interactions can result in potentially unpredictable (emergent) patterns and outcomes (Maciaszek 2007a). Multi-agent systems are designed as sets of autonomous software entities (agents) that are embedded in an organizational structure (the environment). Agents perform tasks by acting in the environment and interacting with one another. Being autonomous, agents have control over their internal state as well as over their behavior.

Having run-time control over their behavior distinguishes agents from objects as normally implemented in object-oriented systems. Objects encapsulate state and some of their behavior (through private and protected visibility

modifiers). However, most object services are public and do not (in typical implementations) discriminate how these services are used by other objects. This means that objects do not have control over their choice of action and they only become active when requested by other objects. We stress, however, that this prevalent computational model for objects is merely the implementation issue. A system could be implemented to allow computations at the knowledge level such that the software entities (whether called objects, components, agents or holons) exert autonomy over their run-time choice of actions based on the definition of the organizational context in which the system executes.

It turns out that by and large the reality of enterprises is (and must remain) much more deterministic and, hence, the behavior of e-business systems is more prescriptive. They operate within the context of prescribed business rules. Biological and agent-like features, such as dynamic (execution-time) learning and emergence, are only required in more strategic e-business applications associated with decision-making, data mining, knowledge discovery and artificial intelligence domains. E-business systems need rather to be adaptive in the sense that the required changes are made as a software development effort (i.e. at compile-time, not at run-time).

Restricting adaptiveness to resultance rather than emergence places software systems on the reductionist end of scientific investigation. This is exactly opposite to what we stated in Section 2 when we argued that complexity places software systems on the holistic end of scientific investigation. However, there is no contradiction here. Complex systems can be classified into those that show resultant properties reducible to analytic simples and those that (also) show emergent properties. It is just that for a system to be truly adaptive, emergent properties must be limited and controlled.

By correlation, the holonic view of scientific investigation gains additional credence as the middle ground between holonic and reductionist views. In some ways, the holonic view offers the middle ground between intuitive and rational knowledge, between ecological and mechanistic view of the world. It also acknowledges that: "Scientific theories can never provide a complete and definitive description of reality. They will always be approximations of the true nature of things. To put it bluntly, scientists do not deal with truth; they deal with limited and approximate descriptions of reality." (Capra 1982, p.33).

Adaptiveness offers an important distinction to the characterization of complexity as related to the *thing* as opposed to complexity as related to its *description*. This is precisely the distinction that motivated Kolmogorov in his search to make the notion of complexity precise and in defining some forty years ago his measure of complexity known as Kolmogorov complexity or K-complexity (Mosterin 2002). The premise of K-complexity was to reduce the qualitative notion of complexity to the quantitative notion of size – the minimum length of a Turing machine program needed to generate the whole description of the thing. Similarly, in our complexity metrics (Maciaszek 2007a, Maciaszek 2006), we measure complexity in terms of the minimum number of dependencies between objects (holons) in the system.

5 Modeling and Engineering

Software is a product of *engineering* as a branch of knowledge aiming at solving practical problems for the needs of humanity. It uses scientific principles to design and construct structures and machines (e.g. software systems). In engineering, goal or target comes first (Endres, Rombach 2003). Solving a problem is equivalent to developing an artifact by using certain methods and following particular process. To verify if the artifact meets its goal, metrics are defined and measurements are taken both during the development and on the artifact delivery. If the measurements indicate departures from the goals, the method and process need to be changed and re-applied.

Engineering can be considered as part of *modeling*. Models are abstract representations of reality. Short of putting forward an untenable argument that a software program is a reality, software production (including engineering) is all about modeling and working with abstraction. A software program is the final and most detailed model that executes on a computer.

Our approach to modeling and engineering of adaptive complex systems is called adHOCS (Maciaszek 2007a). It derives from the holon hypothesis and is centered on a 'holarchical' *meta-architecture*. In any given layer of such a holarchy, a software holon ('H' in the adHOCS acronym) is defined as an object that provides a specific service to the next higher layer and that uses services from the next lower layer. A holon is a recursive concept, i.e. a holon can contain other holons. Likewise, an object is a recursive concept, as per the dominant contemporary programming paradigm – the object-oriented paradigm. At run-time, an object ('O') is an instance of a class.

The inclusion of the component 'C' concept in the adHOCS acronym refers to objects as components, i.e. units of object composition with contractually specified interfaces and which need to be loaded, installed, composed, deployed and initialized before they can be run. In general, a software holon can represent a holarchical layer or a set of layers in any given system. Accordingly, an object can refer to a subsystem representing a layer or to the entire system. However, 'S' in the adHOCS acronym is chosen to stand for web services rather than subsystem/system (but the broader interpretation of 'S' would have its merits as well).

Services are running software instances. In adHOCS, they account for 'societies' of software holons akin of societies in nature, such as ant colonies, human social networks or economic markets. In software systems, 'S' refers to e-business systems created by orchestrating services of various business partners, suppliers and customers.

System adaptiveness is a function of *dependencies* in the software. A necessary but not sufficient condition for an adaptive system is that dependencies are explicit, i.e. readily visible and discoverable from the code. To ensure

adaptiveness the number of dependencies must be manageable to start with and grow at most polynomially with the growth of the system. This second condition can be achieved by a holonic organization of the system, i.e. by constructing it according to some meta-architecture that conforms to the adHOCS model. Over years we have advanced a number of adHOCS conformant meta-architectures. The latest and most elaborate one is called PCBMER and consists of six main layers – Presentation, Controller, Bean, Mediator, Entity, and Resource (Maciaszek 2007b, Maciaszek 2007a, Maciaszek 2006).

Fig. 2 presents the holonic view of a PCBMER system (Maciaszek 2007b). The arrowed lines represent dependency relationships between PCBMER layers. Hence, for example, Presentation depends on Controller and on Bean, and Controller depends on Bean. Note that the PCBMER hierarchy is not strictly linear and a more complex layer can have more than one adjacent layer above it (and that adjacent layer may terminate within the scope of the presented system, i.e. it may have no layers above it, although in general the open-ended property of holons allows creating new dependencies as the system grows or integrates with other systems).

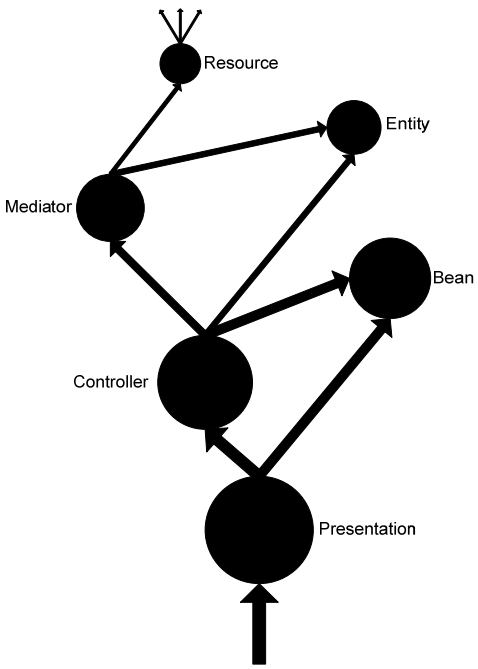

Fig. 2. The PCBMER meta-architecture.

By contrast with more traditional top-down presentations of software architectural layers, the presentation in Fig. 2 is a bottom-up tree-like structure. The *tree* emphasizes here the changing levels of complexity within the holarchy and it de-emphasizes the domination and control aspect of traditional top-down hierarchies (for which the pyramid is a typical symbol). The trunk of the tree signifies that the software system can be connected to or integrated with other software systems, which have similar holonic organization. Each layer has a degree of independence and may, therefore, provide its services to other software systems (it can be re-used).

The relationships between layers are those of composition or containment. Each layer is a whole for layers with lower levels of complexity (i.e. higher in the tree in Fig. 2), and also a part for larger wholes at higher levels of complexity (i.e. lower in the tree). The relative sizes of the circles in Fig. 2 capture the nature of these relationships.

The *Presentation* layer represents the screen and user interface (UI) objects on which the data (beans) from the Bean layer can be rendered. It is responsible for maintaining consistency in its presentation when the beans change. So, it depends on the Bean layer. This dependency can be realized in one of two ways – by direct calls to methods (message passing) using the pull model or by event processing followed by message passing using the push model (or rather push-and-pull model)

The *Bean* layer represents the data classes and value objects that are destined for rendering on UI. Unless entered by the user, the bean data is built up from the entity objects (the Entity layer). The Core PCBMER framework does not specify or endorse if access to Bean objects is via message passing or event processing as long as the Bean layer does not depend on other subsystems.

The *Controller* layer represents the application logic. Controller objects respond to the UI requests that originate from Presentation and that result from user interactions with the system. In a programmable GUI client, UI requests may be menu or button selections. In a web browser client, UI requests appear as HTTP Get or Post requests.

The *Entity* layer responds to Controller and Mediator. It contains classes representing “business objects”. They store (in the program’s memory) objects retrieved from the database or created in order to be stored in the database. Many entity classes are container classes.

The *Mediator* layer establishes a channel of communication that mediates between Entity and Resource classes. This layer manages business transactions, enforces business rules, instantiates business objects in the Entity layer, and in general manages the memory cache of the application. Architecturally, Mediator serves two main purposes. Firstly, to isolate the Entity and Resource layers so that changes in any one of them can be introduced independently. Secondly, to mediate between the Controller and Entity/Resource layers when Controller requests data but it does not know if the data has been loaded into memory or it is only available in the database.

The *Resource* layer is responsible for all communications with external persistent data sources (databases, web services, etc.). This is where the connections to the database and SOA servers are established, queries to persistent data are constructed, and the database transactions are instigated.

The PCBMER meta-architecture provides a model upon which to engineer a specific instance of an adaptive complex system. The development of such an instance assumes adherence to the meta-architecture and the

conformance to related engineering principles and patterns (Maciaszek 2007b, Maciaszek 2007a, Maciaszek 2006, Maciaszek, Liong 2005). It also assumes that the development takes the form of *roundtrip* engineering consisting of cycles of forward- and reverse-engineering activities (Maciaszek 2005).

The forward-engineering process is from design to implementation. The aim is to implement a software product that minimizes dependencies by imposing an architectural solution on programmers. A related aim is to disallow exponential growth of complexity with the introduction of more objects and with the modifications to relations between objects.

This process must be monitored by the reactive approach that aims at measuring dependencies in implemented software. This starts a reverse-engineering process – from implementation to design. The implementation may or 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 need to be pinpointed and addressed for the implemented system in order to reach the quality of adaptiveness.

In software systems, dependencies can be identified for objects of varying granularity – components, packages, classes, methods. The dependencies between more specific objects at lower levels of granularity propagate up to create dependencies at higher level of granularity. Accordingly, dependency management necessitates a detailed study of the program code to identify all relationships between data structures and code invocation between software objects.

Graph-theoretically, the holarchy representing a PCBMER-compliant system (Fig. 2) is a DAG (Directed Acyclic Graph) in which the nodes are ordered (parent and child) and there are no cycles (no path returns to the same node).

Let the PCBMER layers be l_1, l_2 … l_n. For any layer l_i, let:

- $s(l_i)$ be the number of objects in l_i
- l'_I be the number of parents of l_i
- $p_j(l_i)$ be the j[th] parent of l_i

Then, the cumulative object dependency *COD* for a PCBMER holarchy as in Fig. 2 is calculated according to Equation 1 (Maciaszek 2007a):

$$_{PCBMER}COD = \sum_{i=1}^{n} \frac{s(l_i) * (s(l_i) - 1)}{2} + \sum_{i=1}^{n} \sum_{j=1}^{l'_i} (s(l_i) * s(p_j(l_i))) \quad \textbf{(1)}$$

Metrics to measure complexity compute *actual dependencies* in the code as long as the code shows adherence to the meta-architecture. However, as soon as the metrics reveal any violation of the meta-architecture, the complexity must be measured in terms of *potential dependencies* between objects. This is because the system is not a holarchy (is not adaptive) any more; it has degenerated to a random network of intercommunicating objects. Therefore, a change in an object can potentially impact (can have a "ripple effect" on) any other object in the system. To account for potential dependencies, metrics formulas, such as Equation 1, need to be modified.

Measuring adaptiveness of designs and programs cannot be done manually. A tool called DQ (Design Quantifier) is described in (Maciaszek, Liong 2003). It is able to analyze any Java programs, establish its conformance with a chosen adaptive meta-architecture, compute dependency metrics, and visualize the computations in UML class diagrams.

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?"

6 Epilogue

We started this paper with the epigraph. It is therefore proper to conclude with the epilogue which would allude to what might happen to the "story" next. The research reported here has a long history dating back more than ten years. It had its own ups and downs, but it has been always regaining momentum based on generated interests and hopeful practical experiences in university laboratories and within the IT industry.

The problem domain is large and diverse. There are at least four aspects of it: ontological, epistemological, and methodological. All these aspects require further intensive research to unfold the "story".

Ontological aspects relate to the question of the nature of being and reality. They relate to the study of what actually a complex system is, what it is made up of, what properties are assigned to it, and what functions or relations exist within it. Within the broad agreement that entities of higher levels of organization are complex composites of entities of lower levels, we have opted for the holonic view of the world as the basis for an ontological conceptual model.

Epistemological aspects relate to the question of the nature of human knowledge and cognition (how do we know what we know?). They relate to the study of how to obtain (and assess) knowledge of what a complex system is, how this knowledge is embodied in theories, what are the assumptions for a theory, and what are the logical relations between theories. Epistemological aspects border on empiricism and rationalism in the quest to obtain knowledge. From this perspective we admit an initial bias to empiricism - the motivation for our studies of complex systems came from experience and observation of many hardly-maintainable or unmaintainable e-business systems. However, reason and factual analysis have guided us when looking for criteria

and metrics to allow judgments of comparative complexity.

Methodological aspects relate to the methods, procedures, and techniques used to collect and analyze information in order to gain new knowledge. In this context, it is worthwhile to make a distinction between methodology as method of research and methodology as strategy of research (Looijen 2000). In the former sense, every scientist is a reductionist, and we are not an exception. In the latter sense, our approach has been rather holistic, phenomenological, and 'top-down'.

7 References

Agazzi, E. (2002): What is. Complexity? In: *Complexity and Emergence. Proceedings of the Annual Meeting of the International Academy of the Philosophy of Science*. pp. 3-11. Agazzi, E., Montecucco, L. (eds) World Scientific

Brooks, F.P. (1987): No Silver Bullet: Essence and Accidents of Software Engineering, *IEEE Software*, 4, pp. 10-19

Capra, F. (1982): *The Turning Point. Science, Society, and the Rising Culture*. Flamingo, 516p.

Dilworth, C. (2001): Simplicity, *Epistemologia* **24**(2) pp.173-201

Endres, A., Rombach, D. (2003): *A Handbook of Software and Systems Engineering. Empirical Observations, Laws and Theories*, Addison Wesley, 327p.

Fenton, N.E., Pfleeger, S.L. (1997): *Software Metrics. A Rigorous and Practical Approach*, PWS Publ. Comp., 638p.

Jackson, M. (2003): *Systems Thinking: Creative Holism for Managers*. John Wiley & Sons, Ltd., 352p.

Looijen, R.C. (2000): *Holism and Reductionism in Biology and Ecology. The Mutual Dependence of Higher and Lower Level Research Programmes*. Kluwer Academic Publishers, 350p.

Kanitscheider, B. (2002): Beyond Reductionism and Holism. The Approach to Synergetics. In: *Complexity and Emergence. Proceedings of the Annual Meeting of the International Academy of the Philosophy of Science*. pp. 39-44. Agazzi, E., Montecucco, L. (eds). World Scientific

Koestler, A. (1967): *The Ghost in the Machine*. Hutchinson, 384 pp.

Koestler, A. (1978): *Janus. A Summing Up*. Hutchinson, 354 pp.

Koestler, A. (1980): *Bricks to Babel*, Random House. 697p.

Koestler, A., Smythies, J.R. (1969): *The Alpbach Symposium 1968. Beyond Reductionism. New Perspectives in the Life Sciences*. Hutchinson of London, 438p.

Maciaszek, L.A. (2005): Roundtrip Architectural Modeling. In: *Second Asia-Pacific Conference on Conceptual Modelling (APCCM2005)*, pp.17-23. Hartman, S., Stumper, M. (eds). Australian Computer Science Communications 27 (6)

Maciaszek, L.A. (2006): From Hubs Via Holons to an Adaptive Meta-Architecture – the "AD-HOC" Approach. In: *IFIP International Federation for Information Processing, Volume 227, Software Engineering Techniques: Design for Quality*. pp.1-13. K. Sacha (ed.), Boston: Springer.

Maciaszek, L.A. (2007a): An Investigation of Software Holons – the 'adHOCS' Approach. *Argumenta Oeconomica*, **19** (1-2), pp.1-40

Maciaszek, L.A. (2007b): *Requirements Analysis and System Design*. 3rd ed. Addison-Wesley, 642p.

Maciaszek, L.A. and Liong, B.L. (2003): Designing Measurably-Supportable Systems. In: *Advanced Information Technologies for Management.* pp.120-149. Niedzielska, E., Dudycz, H., Dyczkowski, M. (eds). Wroclaw University of Economics Research Papers 986

Maciaszek, L.A., Liong, B.L. (2005): *Practical Software Engineering*. A Case-Study Approach. Addison-Wesley, 864p.

Mosterin, J. (2002): Kolmogorov Complexity. In: *Complexity and Emergence. Proceedings of the Annual Meeting of the International Academy of the Philosophy of Science*. pp. 45-56. Agazzi, E., Montecucco, L. (eds). World Scientific

O'Brien, J.A., Marakas, G.M. (2007): *Enterprise Information Systems*. 13th ed. McGraw-Hill, 543p.

Wegner, P. (1997): Why interaction is more powerful than algorithms, *Comm. ACM* **40** (5) pp.80–91

Wilber, K. (1995): *Sex, Ecology, Spirituality: The Spirit of Evolution*, Shambhala Publ. Inc., Boston, MA.