From Hubs Via Holons to an Adaptive Meta-Architecture – the "AD-HOC" Approach

Leszek A. Maciaszek

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

Abstract. The ever increasing sophistication of software systems brings with it the ever increasing danger of humans losing control over their own creations. This situation, termed the 'software crisis', is said to have existed since the early days of software engineering and has been characterized by the inability of software developers to produce adaptive systems. This paper addresses the roots of the software crisis – the software cognitive and structural complexity and how it could be conquered through the imposition of a meta-architecture on software solutions. The meta-architecture, called PCBMER, epitomizes some important characteristics of holons and holarchies underpinning the structure and behavior of living systems.

1 Introduction

An ***adaptive*** system has an ability to change to suit different conditions; an ability to continue into the future by meeting existing functional and nonfunctional requirements and by adjusting to accommodate any new and changing requirements. *Adaptiveness* is an overriding software quality that consists of a triple of critically important sub qualities – understandability, maintainability, extensibility.

There are three principal underpinnings to achieving adaptive solutions [9]. The first underpinning is the prior existence of a ***meta-architecture*** (framework) to guide architects in doing their job of constructing architectural models for a particular software system. The second underpinning to achieving adaptive solutions is an enforcement of sound ***engineering principles***. If the architectural design *defines* adaptiveness, the engineering principles *deliver* adaptiveness. The third underpinning to achieving adaptive solutions is an enforcement of sound ***managerial practices***. Managerial practices *verify* adaptiveness.

This paper addresses the first underpinning to achieving adaptive software systems. The paper introduces and explains a meta-architecture called *PCBMER* that extends earlier meta-architectures proposed by the author, of which the last is known as the PCMEF framework (e.g. [10, 11]).

The acronym "AD-HOC" refers to our research aimed at modeling software systems on the image of living systems. This research started more than a decade ago with the papers [12, 13]. The research was then channeled to industry projects, elaborated in successive experiments and papers, applied in the textbooks [10, 11],

---

Please use the following format when citing this chapter:

Maciaszek, L.A., 2006, in IFIP International Federation for Information Processing, Volume 227, Software Engineering Techniques: Design for Quality, ed. K. Sacha, (Boston: Springer), pp. 1–13.

and it is now finding its way to a monograph still in writing during this manuscript preparation [9]. Originally, the "AD-HOC" acronym stood for Application Development – Holon-Object-Centric approach. The preferred meaning now is Application Development – Holons, Objects, Components.

2 Complexity in the wires

The ***complexity*** of modern enterprise and e-business systems is ***in the wires*** – in the linkages and communication paths between software modules rather than in the internal size of the modules. The communication paths create *dependencies* between distributed components 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).

Software ***adaptiveness*** is a function of the software *cognitive and structural complexity* (e.g. [4]). It is a function of the ease with which we can understand the software flow of logic and any resulting dependencies.

2.1 Networks

Fig.1 shows a possible system in which objects in various packages (components, subsystems) communicate indiscriminately. This creates a network of intercommunicating objects. The complexity of such systems grows exponentially with the addition of new objects. Even if the complexity within packages can be controlled by limiting the size of the packages, the complexity created by inter-package communication links grows exponentially with the introduction of more packages. The growth is exponential not necessarily because of the *actual dependencies* between objects, but because the flat network structure (with no clearly defined restrictions on communication paths between objects) creates *potential dependencies* between any (all) objects in the system. A change in an object can potentially impact (can have a "*ripple effect*" on) any other object in the system.

Assuming unrestricted origin/destination communication links (i.e. allowing both-directional dependencies between objects), the cumulative measure of object dependencies is given by a simple formula:

$$ {}_{net}CCD = n(n-1) \quad (1) $$

where n is the number of objects (nodes in the graph) and ${}_{net}CCD$ is a cumulative class dependency in a fully connected network (assuming that objects refer to classes).

The formula computes the worst *potential complexity,* where each object can potentially communicate with all other objects. For 17 classes in Fig.1, ${}_{net}CCD$ is equal to 272 (17*16). Although the worst scenario is unlikely in practice, it must be assumed in any dependency impact analysis conducted on the system (simply because real dependencies are not known beforehand). Systems permitting an indiscriminate network of intercommunicating objects are considered not adaptive.

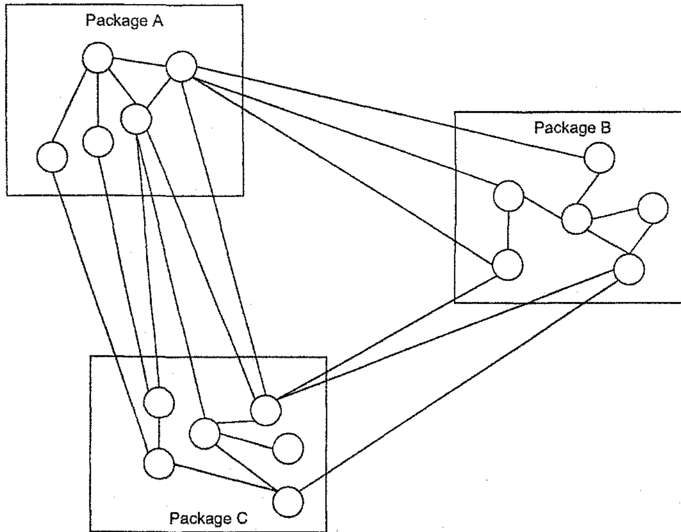

Fig. 1. Network of intercommunicating objects

2.2 Networks with hubs

The exponential growth of complexity in flat network structures is not acceptable. We need to have software architectural solutions that result in merely polynomial complexity growth when new objects/components are added. Such architectural solutions consist of meta-models, frameworks, principles, patterns, etc. At the most generic level, the reduction of complexity can be achieved through so called ***hub structures*** [3].

Fig.2 shows how the complexity of a system can be reduced by introducing hubs. Each package defines a *hub* – an interface object (this could be a Java-style interface or so called dominant class) through which all communication with the package is channeled. Despite the introduction of three extra hub objects, the complexity of the system in Fig.2 is visibly reduced in comparison with the same system in Fig.1.

More formally, the cumulative measure of object dependencies with hubs between packages but with still unrestricted origin/destination communication links within packages is given by the formula:

$$ {}_{hubnet}CCD = \sum_{i=1}^{h} (n_i(n_i - 1)) + (h(h-1)) \qquad (2)$$

where n is the number of objects in each package plus the hub object, h is the number of hubs (i.e. the number of packages) and ${}_{hubnet}CCD$ is a cumulative class dependency in a hub network. For 17 classes and 3 hubs, ${}_{hubnet}CCD$ is equal to 120.

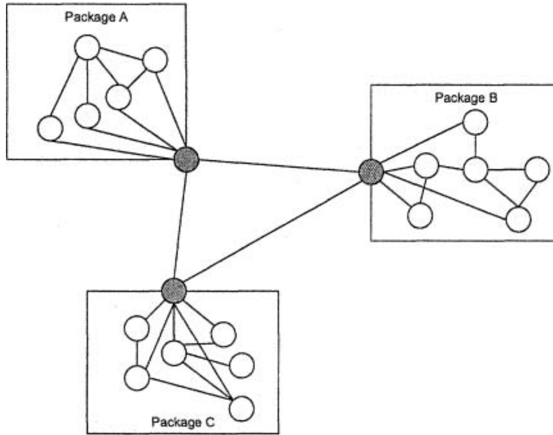

Fig. 2. Reduction of complexity owing to hubs (interfaces) between packages

2.3 Hierarchies with hubs

For the flat network structures, the best complexity values can be obtained in a *hub-spoke* structure, not discussed here [3]. However, in general, any network is a hopeless structure. All complex systems that are adaptive take the form of a *hierarchy*, or rather a *holarchy* (ref. Section 2). A hierarchy/holarchy consists of hierarchically organized layers of objects with one-way (asymmetrical) dependencies between the layers.

Fig.3 shows a *hierarchical structure with hubs* and downward only dependencies between subsystems. Objects are grouped into subsystems instead of packages (subsystems A-C mirror the structure of packages A-C in Fig.2). Subsystems are more appropriate here because the notion of the subsystem (at least in the UML sense) encapsulates some part of the intended system behavior, i.e. a client object must ask the subsystem itself (represented by a hub object) to fulfill the behavior. The notion of the package does not have such semantics [10].

The dependencies between subsystems are only downwards and the dependencies within subsystems have no cycles [11]. Any upward communication between subsystems is realized by a "dependency-less" loose coupling facilitated by interfaces placed in lower subsystems but implemented in higher-level subsystems and/or by event processing instead of message passing and/or by the use of XML-based meta-level technologies. Similarly, cycles within subsystems are eliminated by using interfaces, but also through refactoring techniques that extract circularly-dependent functionality into separate objects/components.

The complexity formula for hierarchies with hubs is:

$$_{hubhier}CCD = \sum_{i=1}^{root} \frac{o_i(o_i - 1)}{2} + \sum_{j=1}^{root} p_{j+1} \tag{3}$$

where:

- o is the number of objects in each subsystem i including any hub objects,
- p_{j+1} is the number of objects in each directly adjacent subsystem above any leave subsystem minus any hub object (this computes the number of potential downward paths to all hub objects in the adjacent subsystems),
- and $_{hubhier}CCD$ is a cumulative class dependency in a hub hierarchy (and assuming as before that objects refer to classes).

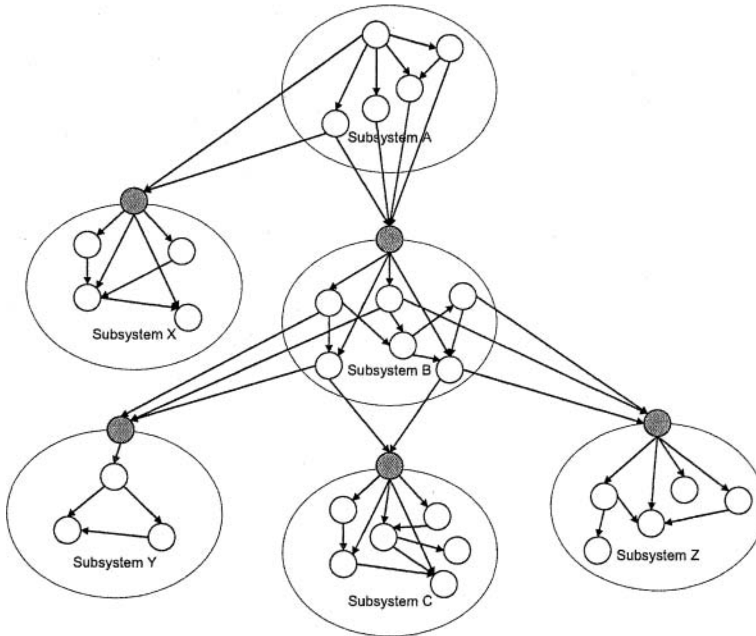

Fig. 3. Reduction of complexity in a hierarchy with hubs

Comparing between Fig.3 and Fig.2, $_{hubhier}CCD$ is equal 63 whereas $_{hubnet}CCD$ is equal 120. The overall $_{hubhier}CCD$ for the model in Fig.3 is equal 111 (63 for subsystems A-C plus 48 for the remaining subsystems).

3 Holons and holarchies

The complexity of living systems by far exceeds the complexity of any man-made system. This observation is easily validated by a simple fact that many intricacies of living organisms escape human understanding. Despite of, or rather owing to, this complexity – living systems are able to *adapt* to changing environments and evolve in

the process. Therefore, it seems sensible to study the structure and behaviour of living organisms in search for paradigms of use in the construction of software solutions.

Living systems are organized to form multi-leveled structures, each level consisting of subsystems which are wholes in regard to their parts, and parts with respect to the larger wholes. Thus molecules combine to form organelles, which in turn combine to form cells. The cells form tissues and organs, which themselves form larger systems, like the digestive system or the nervous system. These, finally, combine to form the living person; and the 'stratified order' does not end there. People form families, tribes, societies, nations. All these entities - from molecules to human beings, and on to social systems - can be regarded as wholes in the sense of being integrated structures, and also as parts of larger wholes at higher levels of complexity.

Arthur Koestler [5] has coined the word ***holon*** (from the Greek word: *holos* = whole and with the suffix *on* suggesting a part, as in neutron or proton) for these entities which are both wholes and parts, and which exhibit two opposite tendencies: an integrative tendency to function as part of the larger whole, and a self assertive tendency to preserve its individual autonomy. Koestler uses the term ***holarchy*** (or *holocracy*) to name a hierarchy of holons from one point of development to another.
Fig.4 represents a possible mental picture of a holarchy. Looking downward, a holon is something complete and unique, a whole. Looking upward, a holon is an elementary component, a part. The diagram captures the essence of holons as defined by Koestler: "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." [5, p.72].

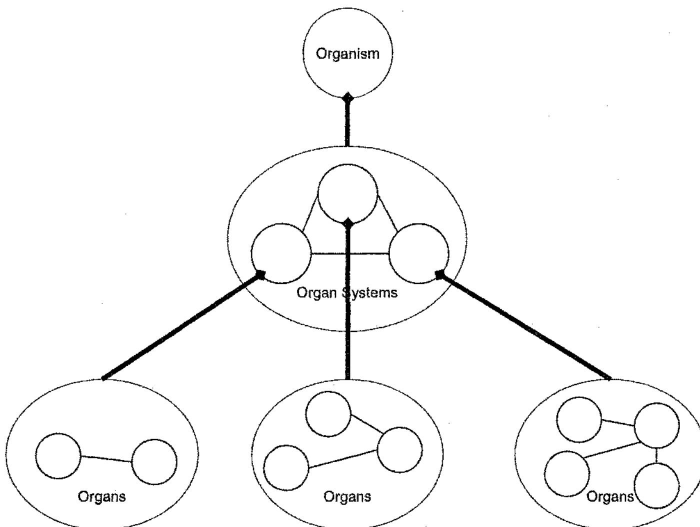

Fig. 4. A holarchy

Individual holons within a holarchy are represented by four main characteristics: (1) their internal charter (interactions between them can form unique patterns), (2)

self-assertive aggregation of subordinate holons, (3) an integrative tendency with regard to superior holons, and (4) relationships with their peer holons.

Holarchies do not operate in isolation, but interact with others. "Thus the circulatory system controlled by the heart and the respiratory system controlled by the lungs function as quasi-autonomous, self-regulating hierarchies, but they interact on various levels." [7, p.463]. Koestler uses the term ***arborization*** for vertical structures and ***reticulation*** for horizontal net formations between holarchies.

Behavior of holarchies is defined by ***fixed rules*** and ***flexible strategies***. The rules are referred to as the system's *canon* that determines its invariant properties – its structural configuration and/or functional pattern. "*The canon represents the constraints imposed on any rule-governed process or behaviour*. But these constraints do not exhaust the system's degrees of freedom; they leave room for more or less *flexible strategies*, guided by the contingencies in the holon's local environment. ... In *acquired skills* like chess, the rules of the game define the permissible moves, but the strategic choice of the actual move depends on the environment – the distribution of the chessmen on the board." [6, pp.293-294].

Since the concept of holon was introduced by Koestler in [5], it has been used by various branches of science ranging from biology via communication theory to more practical uses for implementation of holonic manufacturing systems [16]. Holons and holarchies offer great architectural and other solution ideas for implementing software systems. Successful systems tend to resemble holarchies in many of their aspects, including the ability to hide complexity in successively lower layers, whilst providing greater levels of abstraction within the higher layers of their structures.

The space limitations do not allow us to discuss software technologies (some established, other emerging) that parallel various holon ideas [9]. Most interesting parallels seem to be:

1. Arborization → object composition (e.g. the GoF composite pattern).
2. Reticulation → weaving in aspect-oriented programming.
3. Fixed rules → meta-architectures.
4. Flexible strategies → autonomous agents in multi-agent systems.

4 Dependencies

Our goal is to minimize code dependencies through skillful architectural design. A necessary condition to understand a system behavior is to identify object dependencies and measure ripple effects that they may cause. A ***ripple effect*** of a dependency is a chain reaction that a change to a supplier object may cause on all client objects that directly or indirectly depend on the supplier.

In simple systems, the ripple effect can be determined by the analysis of ***actual dependencies*** in the code. But even in simple systems, finding all actual dependencies may be difficult if some suppliers of services are chosen dynamically at ***run-time*** and are, therefore, unknown at ***compile-time*** (i.e. not directly visible in the source code). It follows that the ripple effect, for all but very simple systems, needs to be determined by the analysis of all ***potential dependencies*** in the code, i.e. dependencies that are allowed by the architectural design of the system, whether or

not they actually exist (and assuming that architectural design is adhered to in the implemented code). Fig.5 provides a classification of dependency relationships relevant to the discussion in this chapter.

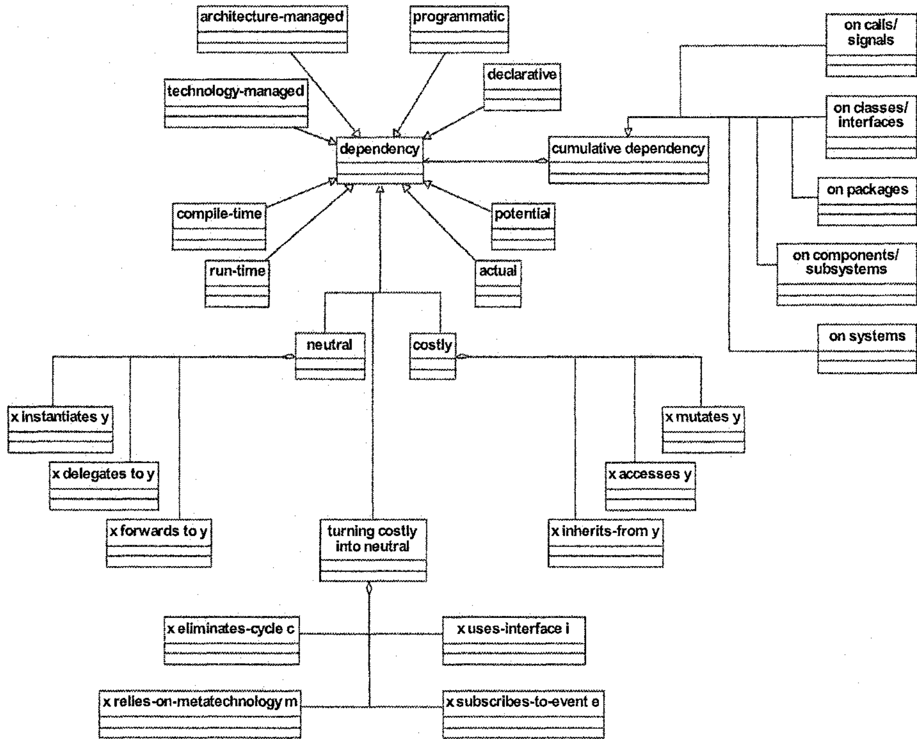

Fig. 5. Dependencies

A hierarchical architectural structure is undefined unless we determine precisely what dependencies are allowed between hierarchy layers and within the layers, and what their potential ripple effect is. These are ***architecture-managed*** dependencies that are under complete control of system developers.

However, *application software* is implemented using particular *system software* and applying particular development technology (application servers, databases, libraries, etc.). The system software takes then responsibility for some important functionality, which otherwise would have to be implemented in the application software. Clearly, application software *depends on* system software, but these are dependencies that cannot be really managed by application developers. These are meta-level ***technology-managed*** dependencies.

Ideally, the integration of application and system software should be based on ***declarative*** dependencies documented in various configuration files, preferably XML files. Configuration files act as agents able to determine actions to be taken (planners), selecting between different possible actions (decision makers), managing execution requests (mediators), etc.

Increasingly, XML-style declarative dependencies replace hard-coded ***programmatic*** dependencies not only in *integration development* (including integration of application and system software) but also within *application development*. As compared with declarative dependencies, programmatic dependencies introduce tight-coupling between client and supplier objects and are significantly more difficult to manage. Sometimes, programmatic dependencies are a sign of weaknesses in the technology applied, but in general they are just a way of making programming objects to communicate in order to make the application perform required tasks.

As mentioned, the main purpose of measuring dependencies is to define their ripple effects so that their impact on system complexity and adaptiveness can be quantified. However, not all dependencies are equally ***costly***. Some categories of dependencies are relatively ***neutral*** in the calculations aiming at establishing cumulative dependencies for the system. There is also an important category of dependencies under the name ***turning costly into neutral*** – they can be used as a way of enforcing the agreed principles of the architecture so that the complexity of the system can be managed.

The notion of *object* can refer to a programming element of any granularity (call (message), signal (event), class, package, component, subsystem, or the entire system). Accordingly object dependencies can be specified on any of these object types. Object dependencies of lower granularity need to be then considered when determining dependencies of higher granularity. Because classes are the main programming modules in contemporary systems, ***class dependencies*** are the focal point of all modern complexity metrics, such as the CK metrics [2].

5 PCBMER meta-architecture

There is no one unique or best meta-architecture that could provide a framework for constructing adaptive complex system. Also, depending on the application domain, the system characteristics and the category of development/integration project, various variations of a particular meta-architecture can be determined and used. The *pivotal meta-architecture*, which we advocate, is called ***PCBMER***. The *PCBMER* framework defines six hierarchical layers of software objects – *Presentation, Controller, Bean, Mediator, Entity* and *Resource*.

5.1 PCBMER layers

Fig.6 illustrates the *Core PCBMER* architectural framework. The framework borrows the names of the external tiers (the Client tier and the EIS tier) from the Core J2EE framework [1]. The tiers are represented as UML nodes. The dotted arrowed lines are *dependency relationships*. 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 higher-layer can have more than one adjacent layer below (and that adjacent layer may be an *intra-leaf*, i.e. it may have no layers below it).

Fig.6 presents two variants of the *Core PCBMER* framework – one defined on UML packages and the other on UML subsystems. As opposed to the variant with packages, the services that components/subsystems provide are fully encapsulated and exposed as a set of *ports* that define the provided and required *interfaces*.

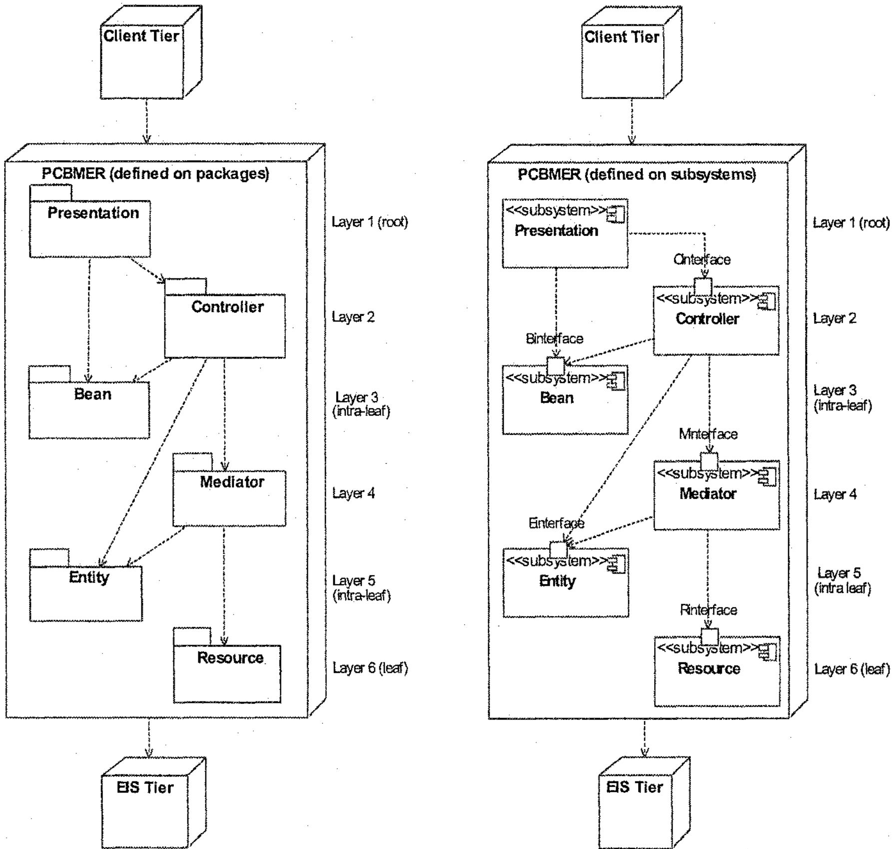

Fig. 6. The Core PCBMER meta-architecture

The emphasis that the notion of component places on encapsulation, ports and interfaces makes components directly applicable for modeling hub structures. Therefore, architectural frameworks presented in the context of subsystems may imply lower cumulative class complexity than those presented with the notion of package.

The ***Bean*** *subsystem* represents the data classes and value objects that are destined for rendering on user interface. Unless entered by the user, the bean data is built up from the entity objects (the Entity subsystem). 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 subsystem does not depend on other subsystems.

The ***Presentation*** *subsystem* represents the screen and UI objects on which the beans can be rendered. It is responsible for maintaining consistency in its presentation when the beans change. So, it depends on the Bean subsystem. 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 ***Controller*** *subsystem* represents the application logic. Controller objects respond to the UI requests that originate from Presentation and that are results of 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*** subsystem 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*** subsystem 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 subsystem, and in general manages the memory cache of the application. Architecturally, Mediator serves two main purposes. Firstly, to isolate the Entity and Resource subsystems so that changes in any one of them can be introduced independently. Secondly, to mediate between the Controller and Entity/Resource subsystems when Controller requests data but it does not know if the data has been loaded to memory or it is only available in the database.

The ***Resource*** subsystem 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 *Core PCBMER* framework has a number of immediately visible advantages resulting in minimization of dependencies. One noticeable advantage is the ***separation of concerns*** between subsystems allowing modifications within one subsystem without affecting the other (independent) subsystems or with a predictable and manageable effect on the other (dependable) subsystems. For example, the Presentation subsystem that provides a Java application UI could be switched to a mobile phone interface and still use the existing implementation of Controller and Bean subsystems. That is, the same pair of Controller and Bean subsystems can support more than one Presentation UI at the same time.

The second important advantage is the ***elimination of cycles*** between dependency relationships and the resultant six-layer hierarchy with downward only dependencies. Cycles would degenerate a hierarchy into a network structure. Cycles are disallowed both between *PCBMER* layers and within each *PCBMER* layer.

The third advantage is that the framework ensures a significant degree of ***stability***. Higher layers depend on lower layers. Therefore, as long as the lower layers are stable (i.e. do not change significantly, in particular in interfaces), the changes to the higher layers are relatively painless. Recall also that lower layers can be extended with new functionality (as opposed to changes to existing functionality), and such extensions should not impact on the existing functionality of the higher layers.

5.2 PCBMER structural complexity

To compute *cumulative dependencies* between program's objects we use ***structural complexity*** metrics and apply them to a particular design and to a resulting implementation. The metrics can apply to objects of various granularities, from methods in classes to subsystems and systems. However, in the structural complexity argument, the most indicative is a cumulative dependency computed on classes.

In traditional software engineering sense, structural complexity metrics reveal the classic tension between cohesion and coupling of objects (e.g. [15]). ***Coupling*** is really another name for dependency between objects. Two objects are coupled if they collaborate with one another. In good designs, coupling is minimized so that collaboration is just enough to perform required tasks. As opposed to our stance on dependencies, coupling allows both-ways collaboration, i.e. cycles are permitted.

If coupling is a relationship between objects, ***cohesion*** defines the internal responsibilities of each object. "A class with low (bad) cohesion has a set of features that don't belong together. A class with high (good) cohesion has a set of features that all contribute to the type abstraction implemented by the class." [14, p.246].

The objective is to have low coupling and high cohesion, but unfortunately these two concepts contradict each other. For any system, the challenge is to define a right balance between coupling and cohesion. The best known strategy to balance coupling and cohesion in object-oriented designs is known as the ***Law of Demeter (LoD)*** (Lieberherr and Holland, 1989). The LoD is known in the popular formulation as "talk only to your friends" principle. It aims at minimizing coupling by prescribing what targets are allowed for messages within class methods. Note that the LoD has a direct counterpart in the *PCBMER*'s NCP principle.

We believe that a starting point to achieve proper balance between the system-wide cohesion and coupling is to ensure that the initial definition of each class is determined alone on the basis of its cohesiveness. We, therefore, assume that – for comparisons of structural complexities in various designs for the same system – the cohesion of classes is constant (with reason, of course; i.e. classes cannot be grouped together to achieve lower coupling, but extra classes may be created to ensure architectural conformance or to take advantage of a particular technology).

With the above in mind, the generic ***cumulative class dependency*** formula for the *Core PCBMER* defined *on subsystems* is the same as Formula 3 for hierarchies with hubs (this is a *generic* formula and other formulas may apply to specific *PCBMER* architectures derived from the *Core* framework). Strictly speaking, there is a difference in the way the formula is applied because the *PCBMER* framework permits a lower-layer subsystem to be communicated from more than one higher-layer subsystem. However, these higher-layer subsystems are considered to be "directly adjacent", thus the formula applies as stands. Note that because only downward dependencies are allowed, the communication from higher-layer subsystems retains the hierarchical properties of the *PCBMER* framework.

Formula 3 ensures *polynomial* growth of dependencies between architectural layers represented as subsystems, while allowing *exponential* growth of class dependencies within layers. However, the exponential growth can be controlled by grouping classes within a layer into *nested subsystems* (as subsystems can contain other subsystems). The communication between nested subsystems can then be performed using hubs.

6 Summary

In this paper we: (1) explained the interplay between software complexity and adaptiveness, (2) showed that hierarchical structures with hubs minimize complexity in the wires (and mentioned, but not elaborated, that hub-spoke structures can provide further minimization), (3) talked about the structure and behaviour of living systems in terms of holons and holarchies and linked these concepts to software systems, (4) provided an elaborated classification of object dependencies, (5) introduced the *PCBMER* meta-architecture and defined its layering structure, architectural principles, and structural complexity.

The lack of space did not permit to address software engineering practices and technologies that could guarantee the compliance of an implemented software system with the *PCBMER* meta-architecture and its principles. Similarly, no reverse-engineering verification procedures were defined to substantiate in metrics the level of compliance in the implemented system. Many of these issues have been addressed in other "AD-HOC" papers and are being compiled into a book [9].

References

1. Alur, D. Crupi, J. and Malks, D.: Core J2EE Patterns: Best Practices and Design Strategies. 2nd edn. Prentice Hall (2003)
2. Chidamber, S.R. and Kemerer, C.F. A Metrics Suite for Object Oriented Design. IEEE Tran. od Soft. Eng. 6 (1994) 476-493
3. Daskin, M.S.: Network and Discrete Location. Models, Algorithms and Applications John Wiley & Sons (1995)
4. Fenton, N.E. and Pfleeger, S.L.: Software Metrics. A Rigorous and Practical Approach. PWS Publ. Comp. (1997)
5. Koestler, A.: The Ghost in the Machine. Hutchinson (1967)
6. Koestler, A.: Janus. A Summing Up. Hutchinson (1978)
7. Koestler, A.: Bricks to Babel. Random House (1980)
8. Lieberherr, K.J. and Holland, I.M.: Assuring Good Style for Object-Oriented Programs. IEEE Soft. 9 (1989) 38-48
9. Maciaszek, L.A.: Development and Integration of Adaptive Complex Enterprise and E-business Systems. Pearson Education (2007) (in preparation)
10. Maciaszek, L.A.: Requirements Analysis and System Design. 2nd edn. Addison-Wesley, Harlow England (2005)
11. Maciaszek, L.A. and Liong, B.L.: Practical Software Engineering. A Case-Study Approach. Addison-Wesley, Harlow England (2005)
12. Maciaszek, L.A. De Troyer, O.M.F Getta J.R. and Bosdriesz, J: Generalization versus Aggregation in Object Application Development - the "AD-HOC" Approach. Proc. 7th Australasian Conf. on Inf. Syst. ACIS'96., Hobart, Tasmania, Australia (1996) 431-442
13. Maciaszek, L.A. Getta, J.R. and Bosdriesz, J.: Restraining Complexity in Object System Development - the "AD-HOC" Approach. Proc. 5th Int. Conf. on Inf. Syst. Development ISD'96, Gdansk, Poland (1996) 425-435
14. Page-Jones, M.: Fundamentals of Object-Oriented Design in UML. Addison-Wesley (2000)
15. Pressman, R.S.: Software Engineering. A Practitioner's Approach, 6th edn. McGraw-Hill (2005)
16. Tharumarajah, A. Wells, A.J. and Nemes, L.: Comparison of the Bionic, Fractal and Holonic Manufacturing System Concepts. Int. J. Comp. Integr. Manufact. 3 (1996) 217-226