ARGUMENTA OECONOMICA
No 1-2 (19) 2007
PL ISSN 1233-5835

Leszek A. Maciaszek *

AN INVESTIGATION OF SOFTWARE HOLONS – THE 'adHOCS' APPROACH

This paper overviews our approach to application development using Holons, Objects, Components and Services (adHOCS) process, it explains the relevance of some modern software technologies to the adHOCS approach, and it presents a case study to measurably illustrate the adHOCS benefits. The holon abstraction – introduced by Arthur Koestler to interpret the structures and processes in living systems – is used to restrain software complexity. The adHOCS approach addresses the adaptiveness issue – namely that the software can adapt to changes and growth provided that better structuring and behavioural abstractions are used for the development and description of software systems. The paper shows that by superimposing the holon abstraction on application development methods, the resulting software systems display readily-understood structures that can accommodate the future growth while managing the underlying complexity.

Keywords: Complex system, adaptive system, holon, holarchy, software architecture, composite pattern, aspect-oriented programming, multi-agent systems, single-subject experimental design, software dependencies, software metrics.

1. INTRODUCTION

This paper investigates a *holon* notion for describing software systems. This notion is based upon Arthur Koestler's interpretation of the structure of natural systems (Koestler 1967, 1978 and 1980; The Alpbach Symposium 1969), which has been adopted by a growing number of scientists in many different fields of research. The central concept of holon is interpreted as an object that is both the part and the whole. More precisely, holons are self-regulating objects which exhibit both the interdependent properties of parts and the independent properties of wholes. Furthermore, these holons form naturally into stratified layers organized in so-called *holarchy*, which happens to be the most promising paradigm to be used (amenably or not) by today's computer industry to model software designs.

The main reason why one should use the holon abstraction is that it restrains *complexity*, which is after all an essential property of software. The

* Department of Computing, Macquarie University, Sydney, Australia

complexity of software derives from various elements of which the intricacies of the problem domain and the inadequacies of development methods are most significant. While the former is an inherent, – and therefore invariant – property of software, the latter could and should be addressed. The point is that the inherent complexity of the problem domain must not be further exacerbated by poor development abstractions and models. The development methods and designs should explain and clarify the underlying problem domain.

The *adaptiveness* of software systems remains a great challenge of contemporary computing. Software solutions tend to be characterized by incomprehensible networks of objects/components responding to random events that invoke a real jungle of interrelated operations. While in small systems such architectures can be controlled, the exponential growth of possible execution paths among operations become quickly uncontrollable in large applications. This is in sharp contrast with living systems that are more complex than any software systems, yet are able to adapt to the changing environment. The holonic view of the world offers an explanation to this adaptation process. Hence, in this paper, we advance the hypothesis that holonic principles can be used to understand and describe the software problem domains and, as a consequence, can be used as an abstraction underpinning the software development methods.

While the holon hypothesis emerged in the context of natural systems, it has also been used to describe and interpret man-made (engineered or designed) systems and social (human activity) systems. In particular, the realm of manufacturing systems has been influenced, or even dominated, by the holonic thinking (e.g., Tharumarajah *et al*. 1996; Babiceanu 2006). In our work, we have used the holon hypothesis to propose a meta-architectural framework together with modelling principles and patterns to guide software architects and engineers in their responsibility to produce complex systems with the built-in quality of adaptiveness.

We call our approach adHOCS (application development – Holons, Objects, Components and Services). The adHOCS approach is not a new application development method and it is not a substitute for other well-known object-oriented analysis and design methods (Maciaszek 2007). It is rather a modeling philosophy that can be applied with any method to produce adaptive software systems.

This research started more than a decade ago with the papers by Maciaszek *et al.* (1996a and 1996b). The approach was then called AD-HOC and stood for Application Development – Holon-Object-Centric approach.

The research was then channeled to industry projects, elaborated in successive experiments and papers, applied in the books (Maciaszek 2005a; Maciaszek and Liong 2005) and addressed in more recent papers. In Maciaszek (2006b) the meaning of the acronym was changed to Application Development – Holons, Object, Components (to reflect the growing significance of software re-use via components). In Maciaszek (2006a), our approach to managing adaptive complex systems was extended to include application integration and interoperability based on web services. The importance of web services is formally acknowledged in this paper in the extended adHOCS acronym.

The paper is structured as follows. In the next section we explain the holon hypothesis as expressed by its architect, Arthur Koestler. Then we go on to describe in Section 3 the adHOCS approach and its representative meta-architecture called PCBMER. Section 4 refers to some of the most important properties of holonic systems and it identifies those major contemporary software technologies that, quite coincidentally, can be used to support those properties. Section 5 presents a simple case study to demonstrate how the adHOCS approach can bring about adaptiveness in a complex enterprise and e-business systems. The Summary Section completes the paper.

2. THE HOLON HYPOTHESIS

“Living systems are organised in such a way that they form multi-levelled 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 woman or man; 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 has coined the word `holons' for these subsystems which are both wholes and parts, and he has emphasised that each holon has two opposite tendencies: an integrative tendency to function as part of the larger whole, and a self assertive tendency to preserve its individual autonomy. In a biological or social system each holon must assert its individuality in order

to maintain the system's stratified order, but it must also submit to the demands of the whole in order to make the system viable. These two tendencies are opposite but complementary. In a healthy system an individual, a society, or an ecosystem there is a balance between integration and self assertion. This balance is not static but consists of a dynamic interplay between the two complementary tendencies, which makes the whole system flexible and open to change." (Capra 1982, p. 27)

This quote from the book 'The Turning Point' (Capra 1982) is where the holon concept was first encountered by the author. It suggests that most of the successful systems are arranged according to a stratified order that hides complexity in successively lower layers, whilst providing greater levels of abstraction within the higher layers of its structure. For example, stratified layers in living systems are:

- Organism (e.g., animal and humans),
- Organ Systems (e.g., nervous, circulatory and lymphatic systems),
- Organs (e.g. brain, heart and lungs),
- Tissues (e.g., epithelial, i.e. skin, and connective, i.e. bone, tissue),
- Cells (e.g., nerve muscle and blood cells),
- Organelles (e.g., mitochondria, ribosomes),
- Molecules (e.g., nucleic acids, i.e. DNA, which are known as the building blocks of life),
- Atoms (e.g., hydrogen, carbon and oxygen).

Each of the layers hides its complexity from the layer above. There are many layers with which the system must progress through before a complex action is completed, like a movement within the human body. We do not consciously know of all the operations that are performed when we move any of our body parts such as when we walk. The complexity of this task is hidden in the various stratified layers within the human body. If we had to worry about where every cell must be any particular moment then we would probably stop breathing while trying to do even the most simplest of operations.

A *complex system* can be defined as "a large network of relatively simple components with no central control, in which emergent complex behavior is exhibited" (Mitchell 2006, p. 1194). Accordingly, and in the graph-theoretic sense, a holonic system is a network of holons, where a *network* is understood as "a collection of nodes (vertices) and links (edges) between nodes" (Mitchell 2006, p. 1196). What makes holonic systems special is the layering of holons, i.e. the 'network' is not a flat chain of links but a *hierarchy*

of stratified holon layers (i.e., holarchy). In Koestler's words: "The chain is a hopeless model; we cannot do without the tree" (Koestler 1978, p. 296).

Koestler defines the holon principles together with a point summary on some of the general properties of Self-regulating Open Hierarchic Order (SOHO). The holon/SOHO concept is an attempt at reconciling atomism/reductionism and holism as two opposite ways of interpreting the world. *Atomism* is a doctrine according to which all higher structures of matter (complex systems) can be explained by (reduced to) their components, i.e., explained by the emergent properties of components acting on each other. By contrast, *holism* can be defined by the statement that the whole is greater than the sum of its components; i.e., the whole is determined on the basis of emergence and interconnectedness of components. Atomism is derived from the Greek word *atomos* = anything that cannot be divided into smaller pieces, holism from *holos* = whole, all, entire, total.

The word *holon* is also derived from the Greek word *holos*, but with the suffix *on* suggesting a part, as in neutron or proton. Holon refers to these entities which are both wholes and parts, and which exhibit two opposite tendencies: an integrative tendency to function as a part of a larger whole, and a self assertive tendency to preserve its individual autonomy.

The holonic view of the world forms a middle-ground between atomism and holism, and the holonic structures form a middle-ground between network and hierarchic structures. The *stratified order* of holonic layers resembles a hierarchy of layers and allows flat networks within layers, but it is different from both. The stratified order is not about a rigid transfer of control or about free interconnectedness of nodes, but it is rather about the self-organization of complexity and adaptation. 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 a stratified order. As a matter of fact, there seem to be no records of survival of any others." (Capra 1982, p. 304)

The *SOHO properties* are broken into ten key points (quoted below from Koestler 1978, pp. 304-311):

1. The Holon

- The organism in its structural aspect is not an aggregation of elementary parts, and in its functional aspects not a chain of elementary units of behaviour.

- The organism is to be regarded as a multileveled hierarchy of semi-autonomous sub-wholes, branching into sub-wholes of a lower order, and so on. Sub-wholes on any level of the hierarchy are referred to as *holons*.

2. Dissectibility

- Hierarchies are 'dissectible' into their constituent branches, on which the holons form the nodes; the branching lines represent the channels of communication and control.
- The number of levels which a hierarchy comprises is a measure of its 'depth', and the number of holons on any given level is called its 'span' (Simon).

3. Rules and Strategies

- Functional holons are governed by fixed sets of rules and display more or less flexible strategies.
- The rules – referred to as the system's '*canon*' – determine its invariant properties, its structural configuration and/or functional pattern.
- While the canon defines the permissible steps in the holon's activity, the strategic selection of the actual step among permissible choices is guided by the contingencies of the environment.

4. Integration and Self-Assertion

- Every holon has the dual tendency to preserve and assert its individuality as a quasi-autonomous whole; and to function as an integrated part of an (existing or evolving) larger whole. This polarity between the self-assertive and integrative tendencies is inherent in the concept of hierarchic order; and a universal characteristic of life.

5. Triggers and Scanners

- Output hierarchies generally operate on the trigger-releaser principle, where a relatively simple, implicit or coded signal releases complex, preset mechanisms.
- Input hierarchies operate on the reverse principle; instead of triggers, they are equipped with 'filter'-type devices (scanners, 'resonators', classifiers) which strip the input of noise, abstract and digest its relevant contents, according to that particular hierarchy's criteria or relevance. 'Filters' operate on every echelon through which the flow of information must pass on its ascent from periphery to center(...)

6. Arborization and Reticulation

- Hierarchies can be regarded as 'vertically' arborizing structures whose branches interlock with those of other hierarchies at a multiplicity of levels

and form a 'horizontal' network: arborization and reticulation are complementary principles in the architecture of organisms and societies.

7. Regulation Channels

- The higher echelons in a hierarchy are not normally in direct communication with "lowly" ones, and vice versa; signals are transmitted through 'regulation channels', one step at a time.

8. Mechanization and Freedom

- Holons on successively higher levels of the hierarchy show increasingly complex, more flexible and less predictable patterns of activity, while on successive lower levels we find increasingly mechanized, stereotyped and predictable patterns.
- (…) New or unexpected contingencies require decisions to be referred to higher levels of the hierarchy, an upward shift of controls from 'mechanical' to 'mindful' activities.

9. Equilibrium and Disorder

- An organism or society is said to be in dynamic equilibrium if the self-assertive and integrative tendencies of its holons counter-balance each other.
- The term 'equilibrium' in a hierarchic system does not refer to relations between parts on the same level, but to the relation between part and the whole (the whole being represented by the agency which controls the part from the next higher level).
- If the challenge to the organism exceeds a critical limit, the balance may be upset, the over-excited holon may tend to get out of control, and to assert itself to the detriment of the whole, or monopolize its functions. (…) The same may happen if the coordinating powers of the whole are so weakened that it is no longer able to control its parts (Child).
- The opposite type of disorder occurs when the power of the whole over its parts erodes their autonomy and individuality.

10. Regeneration

- Critical challenges to an organism or society can produce degenerative or regenerative effects.
- The regenerative potential of organisms and societies manifests itself in fluctuations from the highest level of integration down to earlier, more primitive levels, and up again to a new, modified pattern.

By Koestler's own admission, some of the propositions listed above may appear trivial, some rest on incomplete evidence, others may need correcting and qualifying. However, they have provided a sound basis for discussion among kindred spirits in both atomism and holism cultures, in search of an

alternative to the mechanistic image of a living system. They have also provided a basis for better understanding of man-made systems and, in the context of this paper, for better understanding of how adaptive complex systems should be modeled.

3. THE 'adHOCS' APPROACH

The holon concept offers a new perspective on software and a guidance on how software should be constructed. The kind of software that we are most interested in is modern enterprise and e-business systems. These are complex systems that, sadly in current practice, are not adaptive (or at least not adaptive enough). Our hypothesis is that to accomplish an adaptive complex system, its structure ought to resemble a holarchy of holons and its behaviour ought to adhere to the holonic principles.

In any given layer of such a holarchy, a *software holon* ('H' in the adHOCS acronym) could be described by the *object* that provides a specific service to the next higher layer and by the services it uses 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 the lower level, 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 to 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.

The *complexity* of software systems is *in the wires* – in the linkages and communication paths between software objects. This places software systems on the holistic end of holonic structures, which states that the ways holons are interconnected and integrated are more important than the holons

themselves. 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 1982, p. 26)

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). System *adaptiveness* is then 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. However, to ensure adaptiveness the number of dependencies must be manageable to start with and grow at most polynomially with the evolutionary 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 the 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 2006a, 2006b, 2007).

Figure 1 presents the holonic view of a PCBMER system. 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).

By contrast with more traditional top-down presentations of software architectural layers (including PCBMER visualizations in earlier works by this author), the presentation in Figure 1 is a bottom-up tree-like structure. The *tree* emphasizes here the changing levels of complexity within a holonic software system 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 a 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).

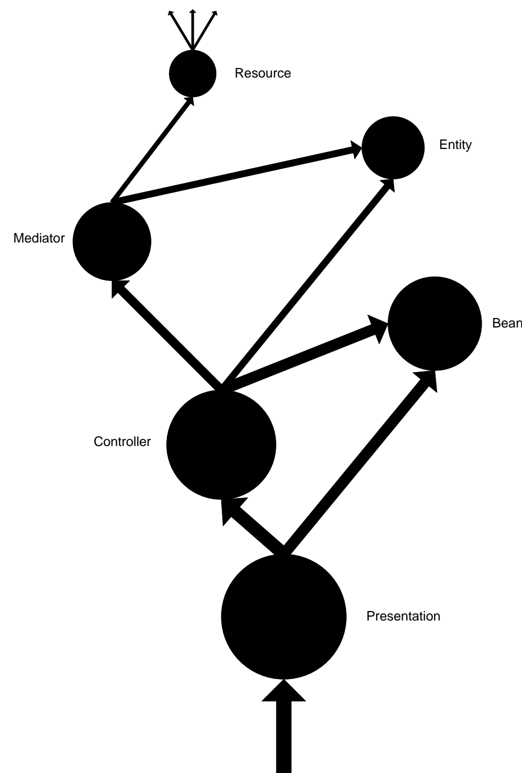

Figure 1. Holonic view of a PCBMER system.
Source: author's own

In general, the tree is "…a more appropriate symbol for the ecological nature of stratification in living systems. As a real tree takes its nourishment through both its roots and its leaves, so the power in a systems tree flows in both directions, with neither end dominating the other and all levels interacting in interdependent harmony to support the functioning of the whole." (Capra 1982, p. 305).

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 Figure 1), 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 Figure 1 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, instantiating 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.

Figure 2 illustrates the PCBMER meta-architecture modeled in UML and showing layers as UML packages. The diagram reverts from the bottom-up visualization in Figure 1 to the top-down engineering view. The system is presented as a *utility service* to emphasize its readiness to integrate with other systems, hence this variant of the meta-architecture is referred to as PCBMER-U (Maciaszek 2006a).

The PCBMER-U meta-architecture is explicitly extended with an interoperability automation package called Orchestration, included in Controller. Orchestration is responsible for discovering web services, providing service binding information to Mediator, and 'orchestrating' an exchange of information through web service interactions. The dependency called service discovery is normally realized through WSDL descriptions. The dependency service binding is normally realized through SOAP invocations, but REST (Representational State Transfer) and Events Triggers can also be used as the invocation means of the service.

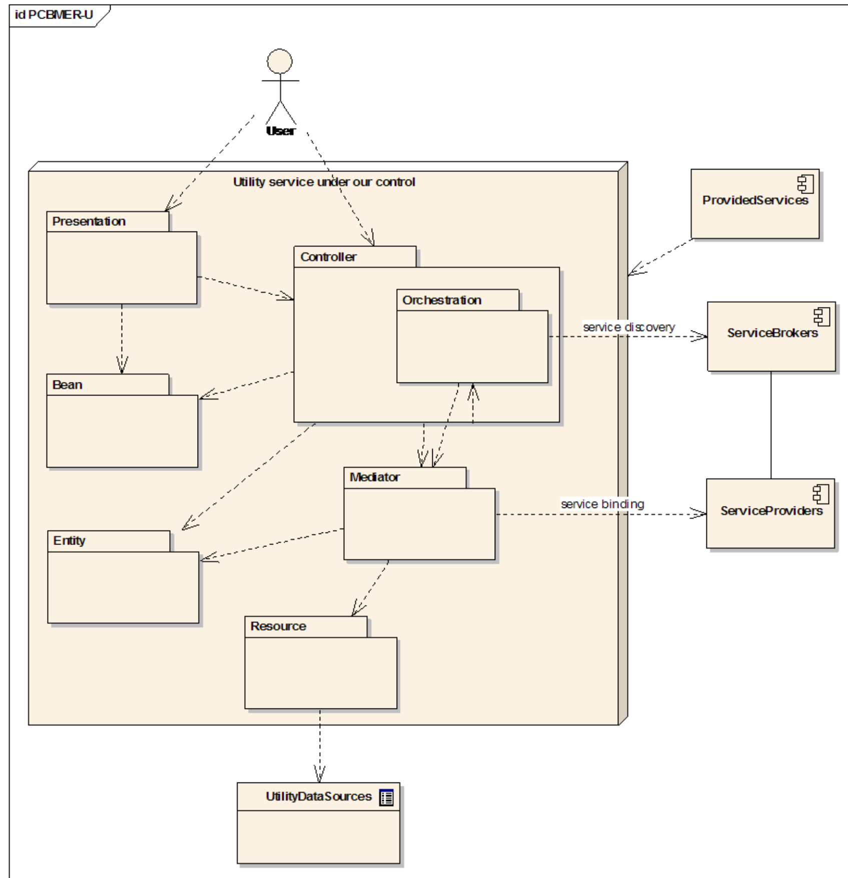

Figure 2. The PCBMER-U meta-architecture.

Sorce: author's own

The definition of the PCBMER meta-architecture includes seven overriding principles:

1. Downward Dependency Principle (DDP)

The DDP states that the main dependency structure is top-down from more to less complex layers. Objects in layers of higher complexity depend on objects in layers of lower complexity. Consequently, lower complexity layers should be more *stable* (should change less) than higher complexity layers. Interfaces, abstract classes, dominant classes and similar devices should encapsulate stable layers so that they can be extended when needed.

2. Upward Notification Principle (UNP)

The UNP promotes low coupling in a bottom-up communication between layers. This can be achieved by using *asynchronous communication* based on *event processing*. Objects in layers of higher complexity act as subscribers (*observers*) to state changes in layers of lower complexity. When an object (*publisher*) in a lower complexity 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.

3. Neighbour Communication Principle (NCP)

The NCP demands that a layer can only communicate directly with its neighbour layer as determined by direct dependencies between layers. This principle ensures that the system does not disintegrate to a network of intercommunicating layers, but it maintains its stratified order. To enforce this principle, the message passing between non-neighbouring objects uses *delegation* or *forwarding* (the former passes a reference to itself; the latter does not). In more complex scenarios, a special *acquaintance* layer can be used to group interfaces to assist in collaboration that engages distant layers.

4. Explicit Association Principle (EAP)

The EAP visibly documents permitted message passing between objects. This principle recommends that *associations* are established on directly collaborating classes of objects. Provided the design conforms to *PCBMER*, the downward dependencies between classes (as per DDP) are legitimized by corresponding associations. Associations resulting from DDP are unidirectional (otherwise they would create circular dependencies). It must be remembered, however, that not all associations between classes are due to message passing. For example, both-directional associations may be needed to implement referential integrity between classes in the entity layer.

5. Cycle Elimination Principle (CEP)

The CEP ensures that *circular dependencies* between layers and classes within layers are resolved. Circular dependencies violate the separation of concerns guideline and are the main obstacle to reusability. Cycles can be resolved by placing offending classes in a new layer created specifically for that purpose or by forcing one of the communication paths in the cycle to communicate via an interface (Maciaszek and Liong, 2005).

6. Class Naming Principle (CNP)

The CNP makes it possible to recognize in the *class name* the layer to which the class belongs. To this aim, each class name is prefixed in *PCBMER* with the first letter of the layer name (e.g. EVideo is a class in the Entity layer). 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 layer (e.g. ICVideo is an interface in the Controller layer).

7. Acquaintance Package Principle (APP)

The APP is the consequence of the NCP. The acquaintance layer consists of *interfaces* that an object passes, instead of concrete objects, in arguments to method calls. The interfaces can be implemented in any *PCBMER* layer. This effectively allows communication between non-neighboring layers while centralizing dependency management to a single acquaintance package.

4. HOLONIC SOFTWARE TECHNOLOGIES

The PCBMER meta-architecture together with its seven main principles and various matching design patterns and programming practices demands enabling software technologies to produce adHOCS compliant systems. Perhaps not surprisingly many existing and emerging technologies are quite suitable for adHOCS development. The main challenge is to use them skillfully and synergistically. In this Section, four technologies are addressed in the context of the 3rd and the 6th SOHO property (Section 2).

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" (Koestler 1980, p. 463). Koestler uses the term *arborization* for vertical structures and *reticulation* for horizontal net formations between holarchies (the 3rd SOHO property – Section 2).

Behaviour of holarchies is defined by *fixed rules* and *flexible strategies* (the 6th SOHO property). 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." (Koestler 1978, pp. 293-294).

Associated with the four SOHO concepts above are the four technologies discussed next:

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.

4.1. Arborization via object composition

The Merriam-Webster OnLine dictionary (http://www.m-w.com/) defines *arborization* as "formation of or into an arborescent figure or arrangement" and *arborescent* as "resembling a tree in properties, growth, structure, or appearance". Koestler used the term arborization to emphasize that holarchies are vertical structures. However, these vertical structures are not isolated but entwined with other vertical structures. To emphasize this entwining, Koestler used the term *reticulation* for horizontal network formations between holarchies.

"One obvious point is that hierarchies do not operate in a vacuum, but interact with others. This elementary fact has given rise to much confusion. If you look at a well-kept hedge surrounding a garden like a living wall, the rich foliage of the entwined branches may make you forget that the branches originate in separate plants. The plants are vertical, *arborising* structures. The entwined branches form horizontal networks at *numerous* levels. Without the individual plants there would be no network. Without the network, each plant would be isolated, and there would be no hedge. *Arborisation* and *reticulation* (net-formation) are complementary principles in the architecture of organisms and societies." (Koestler 1980, p. 463)

Thus, life forms are hierarchically ordered with each layer in the hierarchy being a whole/part holon. Each holon is autonomous, but not spontaneous. The behaviour of each holon is a result of the composition of behaviours provided by the lower-level holons. This implies a top-down vertical communication between hierarchical layers of holons.

"Biologically, a cell is separated from outside by a membrane, through which material enters and exits. Inside a cell there is an organelle, which creates the cell's functions. Cells are immersed in an environment which is chemical. Substances are taken in from this environment.(…) When several distributed units operate in an autonomous manner, there is potential for conflicts and ultimate disintegration of community behaviour. Therefore, some form of coordination is essential. This function in biological systems is executed by enzymes.(…) When specification is given at the top-layer, it passes down layer-by-layer to the bottom. In the bottom-up process, units' actions cumulate and manifest in an operation of the whole system. Coordination is required for the layers to perform in harmony. Biologically this function is fulfilled by enzymes which mediate between layers." (Tharumarajah *et al*. 1996, p. 218)

The feedback loops between arborization and reticulation are realized by a nested nature of holons enabling a *composition* (containment) of behaviour. The issue here refers back to a considerable debate among scientists aimed at distinguishing holarchies from hierarchies (Wilber, 1995). 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. As Wilber observes, a whole contains parts in a way reminiscent of what can be seen in one mirror in a house of mirrors.

The composition processes are both vertical and horizontal. Vertical composition finds a counterpart in the object-oriented notion of *software composition* (discussed in this subsection). Horizontal composition finds a counterpart in *weaving* of the base code and the aspect-code in aspect-oriented programming (discussed in the next subsection).

Vertical *object composition* is governed by the *composite pattern* (Gamma *et al*., 1995). The composite pattern represents whole-part hierarchies of objects and allows treating parts and wholes in a uniform way (by narrowing the difference between the components and compositions of components). Object composition represents a very nature of holons and holarchies. Figure 3 is a UML graphical representation of the composite pattern, as made available by Sparx Systems' Enterprise Architect (http://www.sparxsystems.com.au/uml_patterns.html).

As stated in Gamma *et al.* (1995, p. 163): "The key to the Composite pattern is an abstract class that represents *both* primitives and their containers." To be precise, the Component class is a partially implemented abstract class that declares an interface for objects in the composition. The

behaviour of primitive objects is defined (implemented) in the Leaf class and the behaviour of components having children is defined in the Composite class. Default implementations of the behaviours common to all classes in the composition can be provided in the Component class (hence, this class is a partially implemented *abstract class*, not an *interface* (in Java or UML sense) or not even a *pure abstract class*). Client programs use the *public interface* of the Component class to communicate with objects in the composition. If a target object is an instance of the Composite class, then the client's message will usually trigger (delegate) requests to its child components before the requested operation can be completed.

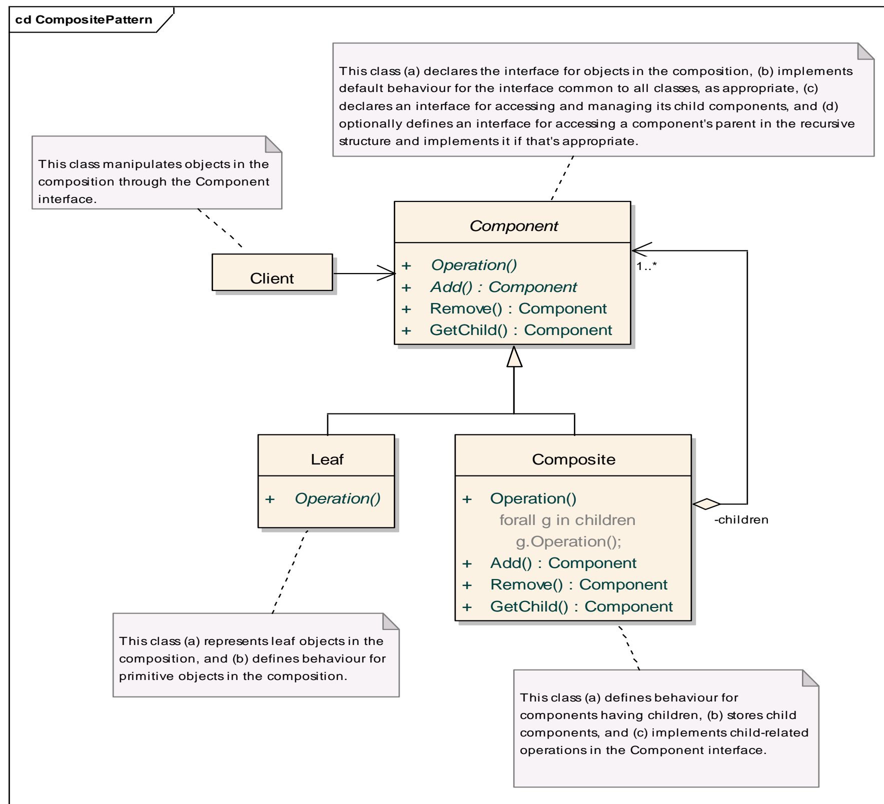

Figure 3. The Composite pattern.

Source: author's own

The composite pattern represents the arborization principle in a holonic system. Complex software systems consist of several layers of abstraction. Each one of them is a quasi-autonomous construction visible as a part that contributes behaviour to the higher layer of complexity and at the same time counting on behaviour from lower layers of complexity. A layer has its own semantics which is complete according to its specification and which is independent from the semantics of the higher layers. The composite pattern determines all transitions between different layers of abstractions that make sense in such a system.

The Component objects conform to the following observation by Koestler with regard to holons: they "...should operate as an autonomous, self-reliant unit which, though subject to control from above, must have a degree of independence and take routine contingencies in its stride, without asking higher authority for instructions. Otherwise the communication channels would become overloaded, the whole system clogged up, the higher echelons would be kept occupied with petty detail and unable to concentrate on more important factors" (Koestler 1967, p. 55)

Composition (or aggregation) has been used as a major semantic modeling technique rivaling inheritance (or generalization) since the seminal work by Smith and Smith (1977). We have been emphasizing the relative advantages of composition over inheritance since 1985 originally by implementing both concepts in a commercial CASE workbench (IDDK), then in a book (Maciaszek, 1990), and in a few research contributions (e.g., Maciaszek *et al*. 1992a; Maciaszek *et al*. 1992b). Some of our earlier work was consistent with the proposals of other researchers to use composition as a method of stratifying entity-relationship diagrams to improve their readability (Teorey *et al.* 1989). In Maciaszek *et al*. (1996a) we contrasted composition (aggregation) with inheritance (generalization) and showed that composition is a better abstraction when it comes to supporting the understanding and evolution of large systems.

4.2. Reticulation via aspect weaving

Reticulation expresses horizontal net formations between holarchies. Without reticulation, each holarchy would be isolated, and there would be no *integration* of functions. Clearly, any *application integration* project (Maciaszek 2006a) can be seen as a reticulation process. However, even within a single application development there is a need for reticulation. That

need has led to a relatively recent paradigm of *Aspect-Oriented Programming (AOP)* (Kiczales *et al*. 1997).

AOP is not a revolutionary idea – few truly useful ideas are. As discussed at the end of this subsection, most concepts underpinning AOP have been known and used before, although frequently under different names and using different technologies. The main objective of AOP is to produce more modular systems by identifying so-called *crosscutting concerns* and producing separate software modules for these concerns. The modules are called *aspects* (akin of holonic arborizations). The aspects are integrated through the process called *aspect weaving* (akin to holonic reticulation).

A starting point for AOP is a realization that a software system consists of many vertical modules. The pivotal module contains classes that implement the functional requirements of the system. However, each system must also obey nonfunctional requirements that determine such software qualities as correctness, reliability, security, performance, concurrency, etc. (Maciaszek and Liong 2005). These qualities need to be addressed from various (or even most) classes/components/services responsible for the system's functions. In "conventional" object-oriented programming, the code implementing these qualities would be duplicated (scattered) in many classes. These nonfunctional qualities are known in AOP as *concerns* – goals that the application must meet. Because the object-oriented implementation of these concerns would cut across many classes, they are known as *crosscutting concerns*.

To avoid code scattering due to crosscutting concerns, AOP advocates the gathering together of such code into separate modules called *aspects*. Although aspects tend to be units implementing nonfunctional requirements, in general they could also be units of the system's functional decomposition. In particular, they could implement various enterprise-wide *business rules* that need to be enforced by classes responsible for the application's *program logic*.

Thus, AOP decomposes systems into *aspects* built around the core functional classes, interfaces and other object-oriented constructs. The classes constitute the *base code*. The aspects constitute the separate *aspect code*. A form of reticulation is required for such a system to work. Classes have to be composed with aspects or, to put it the other way around, aspects have to be weaved into the program logic flow. Such a process of software composition is called *aspect weaving*. Some aspects can be weaved into the system at compile time (*static weaving*), others can only be weaved at runtime (*dynamic weaving*).

Aspect weaving applies to *join points* in the program's execution. Join points are pre-defined points of software composition, such as a call to a method, an access to an attribute, an instantiation of an object, pointing out an exception, etc. A particular action that needs to be taken for a join point is called an *advice* (e.g., checking security permissions of the user or starting a new business transaction). An advice can run before the join point it operates over (*before advice*), after a join point completes (*after advice*), or it can replace the join point it operates over (*around advice*).

In general, the same advice may apply to many join points in the program. A set of join points related to a single advice is called a *pointcut*. Pointcuts are often defined programmatically with wildcards and regular expressions. Compositions of pointcuts may also be possible (and desirable).

With all its good intentions to improve software modularization (and therefore adaptiveness), AOP can be a potential source of emergent or even incorrect behaviour (e.g., Murphy and Schwanninger 2006). This is because aspects modify behaviour at join points in a way that may be oblivious to the developer responsible for the application's functional logic. Interactions of aspects with the base code can be quite far reaching and result, for example, in an instantiation of an object, invoking a method in the public interface of a class, or even introducing a new method or constructor declaration into a class. Moreover, aspects themselves are not necessarily independent and multiple aspects can affect each other in a subtle way resulting in emergent behaviour.

There is a clear need for AOP development practices to ensure that the aspect code and the base code evolve gracefully together and the crosscutting concerns are well-documented and known to application developers at all times. A resolution to this dilemma is particularly difficult in the presence of dynamic weaving (e.g., Hirschfeld and Hanenberg 2005). A necessary condition to be able to tackle these issues is the developer's awareness of the mutual impact of changes in the base and aspect code. This difficulty is similar (although, arguably, easier to deal with) to that faced by developers of multi-agent systems in which dynamic agent interactions can also result in potentially unpredictable patterns and outcomes (Section 4.4).

Reticulation is a necessity in adaptive complex systems. Hence, AOP's implementation of reticulation by the use of aspect weaving is not a radically new approach, but rather a neat evolution and generalization of previous concepts, approaches, and technologies. Johnson and Hoeller (2004) identify the following established object-oriented ideas on which AOP has built its framework:

- *Metaclasses* in languages such as Python, which offer the way to modify the behaviour of classes and help to reduce code duplication across multiple methods.
- *Before, after* and *around constructs* in the CLOS (Common Lisp Object System) language.
- The *Decorator pattern* (Gamma *et al.* 1995) that allows the addition of a custom behaviour when a method is invoked. The pattern uses a Decorator class that implements the same interface as the "base" class but is able to extend the implemented interface with a custom code. By doing so, the Decorator pattern can be seen as a variation of the *Composite pattern* allowing distribution of responsibilities among objects and modularization of concerns.
- The *Observer pattern* that defines "a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically" (Gamma *et al.* 1995, p. 293). Also known as the *Publish-Subscribe pattern*, the Observer pattern makes subscriber objects perform particular process ("advice") when a publisher object notifies of an event. Under this scenario (and unlike in AOP), the publisher objects are not oblivious of the advice composition that applies to them.
- The *Chain of Responsibility pattern* (Gamma *et al.* 1995), known also as *delegation*, that allows a client object to request a service without knowing which of the supplier objects in a "chain of responsibility" is able to supply the service ("advice"). However, as in the Observer pattern, the client object is not oblivious of the advice composition because it is necessary to explicitly set up a chain of responsibility for each method requiring one.

Apart from object-oriented influences on AOP, one can find influences from database programming. For example, the before, after and around constructs are reminiscent of the *before, after* and *instead triggers* (e.g., Silberschatz *et al*. 2006). Triggers are programs that fire automatically when an underlying table is subject to change by SQL operations of insert, update or delete. Comparing with AOP, triggers provide "advice" for join points defined as method invocations (and the methods are mutator methods, i.e., setters, not getters).

4.3. Fixed rules via architectural frameworks

Associated with holons and holarchies are fixed rules and flexible strategies. *Fixed rules* impose constraints and controls on the holon's activities. In Koestler's terminology, they define the holon's *code* or *canon* –

holon's structural configuration and functional pattern; the permissible steps in the holon's activities. "… every level in a holarchy of any type is governed by a set of fixed, *invariant rules*, which account for the coherence, stability, and the specific structure and function of its constituent holons." (Koestler 1980, p.454)

Flexible strategies enable selection of the actual step among permissible choices. The step that will be taken is guided by the contingencies of the holon's current environment. "The canon determines the rules of the game, strategy decides the course of the game." (Koestler 1978, p.305). The following example illustrates the relationship between fixed rules and flexible strategies.

"The common spider's web-making activities are controlled by a fixed inherited canon (which prescribes that the radial threads should always bisect the laterals at equal angles, thus forming a regular polygon); but the spider is free to suspend his web from three, four or more points of attachment – to choose his strategy according to the lie of the land." (Koestler 1980, p.455)

Not surprisingly, there is a direct mapping from the canon of holons to the canon of software systems. This canon of software systems is represented by *meta-architectures* (*frameworks*) that are used to design system architectures. Most modern architectural frameworks, including the PCBMER framework (Section 3), are underpinned by the *Model-View-Controller* (MVC) framework developed as part of the Smalltalk-80 programming environment (Krasner and Pope 1988). In Smalltalk-80, MVC forces the programmers to divide application classes into three groups that specialize and inherit from three Smalltalk-provided abstract classes – Model, View and Controller.

Model objects represent data objects – the business entities and the business rules in the application domain. Changes to model objects are notified to view and controller objects via event processing. This uses the publisher/subscriber technique. Model is the publisher and it is therefore unaware of its views and controllers. View and controller objects subscribe to the Model, but they can also initiate changes to model objects. To assist in this task, Model supplies necessary interfaces, which encapsulate the business data and behavior.

View objects represent user interface (UI) objects and present the state of the model in the format required by the user and rendered on the user's graphical interface. View objects are decoupled from model objects. View subscribes to the Model so that it gets notified of Model changes to update

its display. View objects can contain subviews, which display different parts of the Model. Typically, each view object is paired with a controller object.

Controller objects represent mouse and keyboard events. Controller objects respond to the requests that originate from View and that are the results of user interactions with the system. Controller objects give meaning to keystrokes, mouse clicks, etc. and convert them into actions on the model objects. They mediate between view and model objects. By separating user input from visual presentation, they allow changing system's response to user actions without changing the UI presentation, and vice versa – changing UI without changing system behavior.

Figure 4 illustrates an actor's (user's) perspective on communication between MVC objects (Maciaszek and Liong 2005). The arrows represent communication between objects (they are not really meant to represent dependencies as discussed in Section 2). The user GUI events are intercepted by view objects and passed to controller objects for interpretation and further action. Mixing the behavior of View and Controller in a single object is considered a bad practice in MVC.

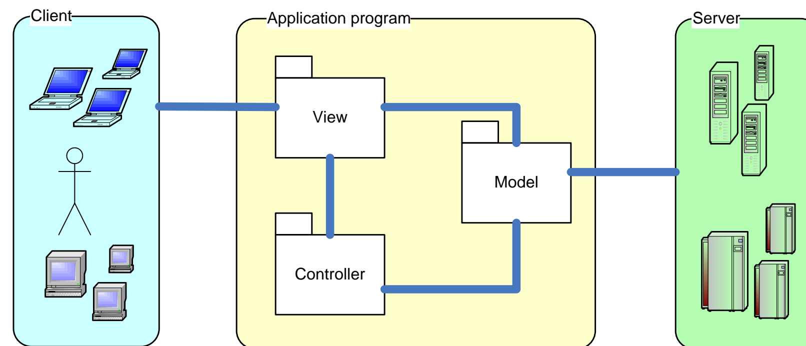

Figure 4. The MVC architectural framework.

Source: author's own

MVC is a backbone of virtually all modern frameworks, which then extend the framework to enterprise and e-business systems. The Core J2EE architecture is one such framework (Alur *et al.* 2003; Roy-Faderman *et al.* 2004). As shown in Figure 5, the J2EE model consists of tiers – three embracing the application program components (Presentation, Business, and Integration), and two external to the application (Client and EIS – Enterprise Information System).

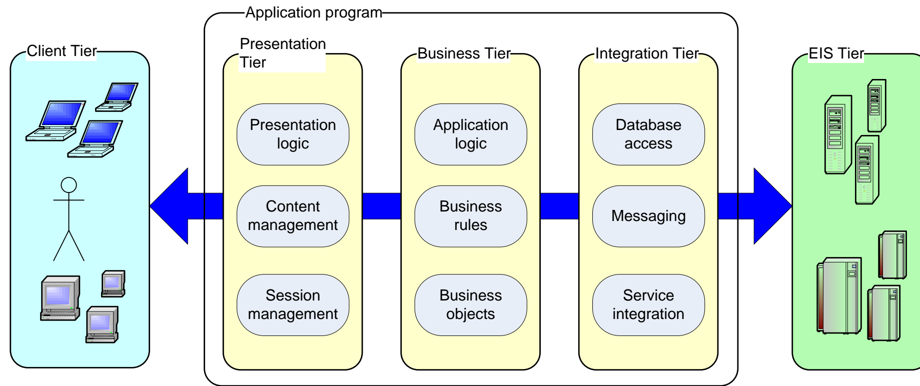

Figure 5. The Core JEEE architectural framework.

Source: author's own

The user communicates with the system from the *Client* tier. This tier can be a programmable client (e.g. a Java Swing based client or applet), a HTML web browser client, a WML mobile client or even an XML-based web service client. The process that presents a user interface can execute on the client machine (a programmable client) or can execute from a web or application server (e.g. a Java JSP/servlet application).

The *EIS* tier (called also the *Resource* tier) is any persistent information delivery system. This could be an enterprise database, an external enterprise system in an e-business solution, or an external SOA service. The data can be distributed across many servers.

The user accesses the application via the *Presentation* tier (known also as the *Web* tier or a *Server-Side Presentation* tier). In a web-based application, this tier contains user interface code and processes that run on a web and/or application server. With reference to the MVC framework, the Presentation tier contains both the view and controller components.

The *Business* tier contains parts of the application logic not already implemented as controller components in the Presentation tier. It is responsible for validation and enforcement of enterprise-wide business rules and business transactions. It also manages business objects, previously loaded from the EIS tier to the application memory cache.

The *Integration* tier has sole responsibly for establishing and maintaining connections to data sources. This tier knows how to communicate with databases via JDBC (Java Database Connectivity) and how to integrate with external systems via JMS (Java Messaging Service).

The Core J2EE framework is generic and explanatory, rather than regulatory. It introduces “separation of concerns” between the three application programs’ tiers. It also dictates that Presentation components can only communicate with Integration components via a Business tier, and vice versa. But it does not, for example, enforce a strict stratified order because it does allow both-way communication (invocation of methods) and it does therefore allow invocation cycles.

There are, however, multiple technologies developed for J2EE that are quite regulatory and address various holonic-like complexity concerns in the development and integration of enterprise and e-business systems. They start with technologies such as Jakarta Struts to allow proper implementation of the MVC pattern. They extend to enterprise services with technologies such as the Spring Framework and application servers (e.g. JBoss or Websphere Application Server). They further extend to e-business integration with implementations of JMS within applications servers.

4.4. Flexible strategies via autonomous agents

The notion of *flexible strategies* establishes another connection between Koestler’s holons and holarchies and the IT technologies that seem suitable for building adaptive complex systems. One such promising technology is that of agents and multi-agent systems in agent-based computing (Ferber 1999; Jennings 2000). Arguably, the multi-agent computing model together with the associated multi-agent software engineering can be used to realize holarchical systems with a sufficient dose of flexible strategies (Pichler 2000).

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.

After Ferber (1999) (as cited in Pichler 2000), agents "can realize different organizational function such as being a *supplier* (servicing customers), a *mediator* (managing execution requests), a *planner* (determining actions to be taken), a *coordinator* (distribution of actions and execution requests), a *decision maker* (making the choice between different possible actions) or an *executive* (realizing actions)". As further observed by Pichler, similar functions can be attributed to holons.

Like agents, holons have the ability to interact with their environment (organizational context). Unlike agents, holons are whole-parts with built-in feedback channels to their master holons above in the holarchy and with communication channels to their sub-holons below them. By contrast, agents interact within a flat (but flexible) organizational context in which explicit organizational relationships provide agents with a decision-making framework. The organizational relationships are themselves the subject of ongoing changes (due to varying social interactions, flow of time, etc.). Accordingly, protocols need to be specified to enable forming and dismantling of organizational groupings.

The connection between multi-agent systems and holonic biological systems is not accidental. After all, the ultimate goal is to be able to construct artificial (mechanical) systems that are "naturally" complex and need to be adaptive. One (albeit dominant) category of such artificial systems is enterprise and e-business systems. There is a need to determine the degree to which the nature of enterprise and e-business systems is compatible with the run-time self-regulation powers of holons and agents.

It turns out that by and large the reality of enterprises is (and must remain) much more deterministic and, hence, the behavior of enterprise and 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, self- adaptiveness, etc. are only required in more strategic enterprise and e-business applications associated with decision-making, data mining, knowledge discovery and other artificial intelligence domains. Enterprise and e-business systems need rather to be *adaptive*, i.e. to be understandable, maintainable and scalable in the sense that the required changes are made as a software development effort (i.e. at a compile-time, not at a run-time).

Clearly, enterprise and e-business systems change in structures and in behavior (after all, company structures, product and employee classifications, plant operation specifications, and many other aspects of business are in a constant state of flux). Enterprise and e-business systems need to be *designed for change* but they cannot typically allow unpredictable patterns and outcomes of the interactions characteristic of multi-agent systems. To become a mainstream technology for enterprise and e-business systems, the multi-agent computing model needs to be equipped with social level characterizations (i.e. an organizational context) that would counteract any emergent behavior (Jennings 2000). Interestingly, and as an aside, the changes in enterprise and e-business systems seem to be more far reaching than changes in biological systems. Biological organisms change behavior and their internal state, but they are unlikely to acquire new behavior or new structures (outside, of course, of the biological evolution).

Secondly, and as a related issue, the holon hypothesis explains the structure and behavior of an *"implemented"* system (e.g. living organism), but it does not explain the abstractions needed for the *development* of a system. Short of further analyzing the evolution of living organisms, we follow the suggestions of Koestler and others that a large system that works is always a result of an evolution of a small system that was built using an adaptive structure reminiscent of a holarchy.

Therefore, the development process must start with an architectural design that proposes stratified layers of abstractions, such as in the PCBMER meta-architecture (Section 2). The layering structure that governs that higher layers depend on lower layers but lower layers should be independent from the higher layers. This in turn results in a requirement that lower layers should be designed to be more *stable* than higher layers, i.e. lower layers must not be subject to changes or the changes should be very infrequent and carefully controlled. Note that the fact that a layer is not susceptible to changes does not mean that it is difficult to extend (Martin 2003). Interfaces, abstract classes, dominant classes and similar devices should encapsulate stable packages so that they can be extended when needed.

There is a direct relevance of this notion of stable layers to the interplay between fixed rules and flexible strategies in holarchies:

"Generally we find on successively higher levels of the hierarchy increasingly complex, more flexible and less predictable patterns of activity with more degrees of freedom (a larger variety of strategic choices); while conversely every complex activity, such as writing a letter, branches into sub-skills which on successively lower levels of the hierarchy become

increasingly mechanical, stereotyped and predictable (Cf. the ethnologist's "fixed action patterns")." (Koestler 1980, p.462)

This reflection by Koestler implies that the more *stable* the architectural layers are (or need to be), the less *strategic choices* they present (or allow). On closer investigation, the layer stability and the scope for strategic choices are two sides of the same adaptiveness coin. (Incidentally, in Koestler's holarchies, the relevance of this distinction is taken further to the analysis of such fundamental problems in living systems as *free will* versus *determinism*.)

It is therefore crucial that an architectural intent is obeyed in the implemented system. This is the pivotal assumption of the adHOCS approach and the PCBMER framework, which not only dictates the fixed rules of the game, but also provides (as many as possible) flexible strategies defining the course of the game. Unsurprisingly, the latter task is much more formidable than the former. Like in the game of chess, the rules are simple and define the permitted moves, but the strategy that decides the choice of the actual move is not available in the rule-book. There are only some partial strategies suggesting which move to take in some well-defined situations, such as strategies for game opening, for checkmate attacks, for endgames. There are also more elaborate winning plans, such as French Defense, Vienna Gambit, Sicilian Grand Prix Attack, etc. The knowledge of these strategies turns a social player into a professional whose moves are no longer *random*. Well, this is the best we can hope for in the "software game" as well.

5. CASE STUDY

Associated with the adHOCS approach for constructing holonic software systems are various design patterns that software developers can use to solve specific implementation problems and ensure PCBMER conformance (Maciaszek and Liong 2005; Maciaszek 2007). Also associated with adHOCS are various controlling mechanisms to measurably verify (using complexity metrics) that any specific implementation is PCBMER compliant and that it minimizes dependencies to facilitate software adaptiveness (e.g. Maciaszek and Liong 2003; Maciaszek 2006b). Rather than re-iterate these issues in this paper, we present here a small case study that illustrates the benefits of the adHOCS approach, and the PCBMER meta-architecture in particular, to constructing adaptive software systems.

The research method applied in this case study is *single subject experimental design* popularized by clinical psychology (Heffner 2004), experimental medicine (e.g. McReynolds and Thompson 1986), education, social sciences, and also advocated for using in software engineering (Harrison 2005). As a consequence of B.F. Skinner's research known as operant behavior (ref. BFSkinner 2006), single subject designs perform and measure behavioral modifications by the comparison of treatment effects on a single subject (rather than comparing groups of subjects).

The most common application of single subject methods is known as the *A-B-A-B design* (Heffner 2004). The design starts with the gathering of pretest information sufficient to see a trend, often called a *baseline measurement* (the first *A*). For the purpose of the study of adaptive complex systems, a baseline is a system design that satisfies the user functional requirements. The baseline may or may not be implemented and deployed when the measurements are taken. The measurements take advantage of metrics for computing structural complexity of software (Maciaszek and Liong 2003; Maciaszek 2006b).

The baseline is the starting point to which a *treatment* is applied. The change due to treatment is *measured* leading to the first *B* of the *A-B* design (known as the *treatment measurement* step). For adaptive complex systems, the treatments take the form of architectural principles, engineering patterns, etc. The subject is then left alone for a period of time. In our case of the subject being an enterprise or e-business system, the time is allowed for changes and extensions to the system.

The next step is the *withdrawal of treatment* (the second *A* of the *A-B-A* design). The purpose is to determine if the subject returns to original behavior or if the behavioral changes due to the first treatment continue. Accordingly, the *second baseline measurement* is taken to see the effects of withdrawing the positive reinforcer on behavior. In the case of software, withdrawal of treatment signifies a likely deterioration of system structure and behavior due to maintenance activities and due to normal iterative and incremental evolution of the system's required functionality. At this step, we can observe if the treatment applied for the *A-B* design is robust, extensible, well-understood and accepted by developers, etc.

The second *B* in the *A-B-A-B* design is the *re-introduction of treatment.* The treatment is once again applied and the *second treatment measurement* is taken to establish the effects of spontaneous recovery. In the case of human-made systems, the "spontaneous recovery" should rather be called a "controlled recovery", as it has to do with applying even more rigorous

architectural and engineering measures and with applying stricter controls to ensure that the subject is brought again to good health.

It is interesting to note that single subject experimental design corresponds nicely with the software development practice of *round-trip engineering* (Maciaszek 2005b). The *A*-*B*-*A*-*B* design is compatible with the first cycle of round-trip engineering, in which forward engineering stage accomplishes the *A*-*B* design, the reverse engineering establishes the second baseline measurement and the next forward engineering push re-introduces the treatment and completes the *A*-*B*-*A*-*B* design. Continuing round-trip engineering activities may be seen as more complex variants of single subject methods, such as an *A*-*B*-*A*-*B*-*A*-*B*-*A*-*B* design.

The case study relates to a simple application called "Foreign Exchange Calculator" (FEC). The application consists of just two web pages (Figures 6 and 7). The first page enables the user to enter the amount of money to be converted, to select from combo boxes the "from" and "to" currencies, and then to press Calculate or a similar button. The second page shows the results of the calculation.

Foreign Exchange Calculator

This amount	of this type of currency	into this type of currency
Enter amount 250	AUD Australia Dollars	USD United States Dollars

Perform Currency Conversion

Figure 6. The FEC web application – entry page.

Source: author's own

From	AUD
To	USD
Amount	250.0
Rate	0.742052
Exchange Total	185.513

NextConversion

Figure 7. The FEC web application – result page.

Source: author's own

The method used is the A-B design. The *A* design establishes a *baseline measurement* for the subject. The baseline could be any implementation that delivers the FEC functionality. It is expected that the implementation will have symptoms of a disease in the A design, for otherwise no treatment of the B design would be necessary (a healthy subject does not need a treatment).

Let us assume that the processing logic in the baseline FEC application is as in the UML sequence diagram in Figure 8. A Request web page accepts user input, instantiates a Calculator object and asks it to calculate() the exchange amount. To be able to do the calculation, Calculator needs to getRate() from the database. This task is performed by a Query object. After obtaining the exchange rate, Calculator instantiates a Bean object and sets its content with data ready for display in a Result web page. Calculator instantiates a Result object and passes to it the reference to Bean. This enables Result to get Bean's data and render it to the screen. A user's request to startOver() results in passing the control back to the Request web page.

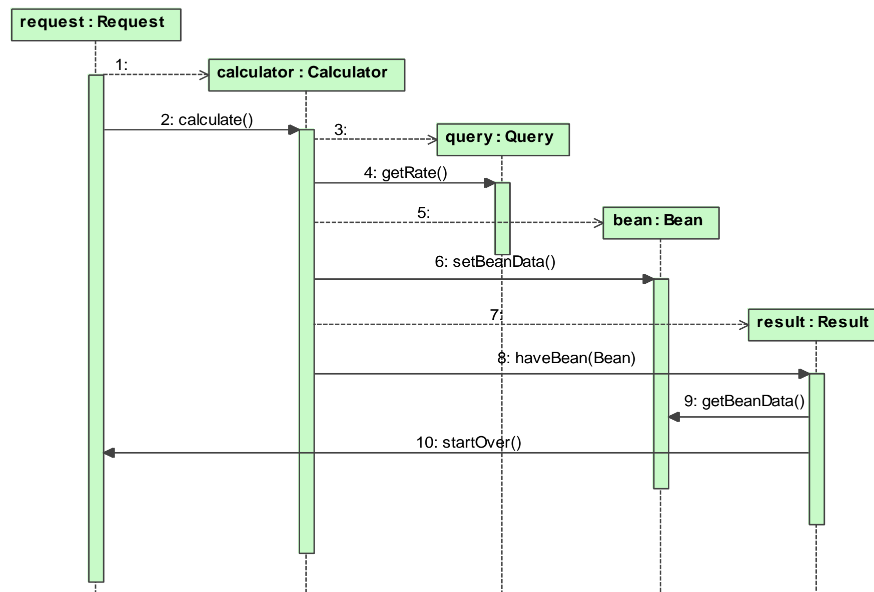

Figure 8. Sequence diagram for the A design (FEC).

Source: author's own

Figure 9 is a class dependency diagram for the sequence diagram in Figure 8. The diagram shows that all classes have been allocated to the same package (as could be expected in the case of an ad-hoc design without an

architectural vision). The diagram also shows a worrying cyclic dependency between Request, Calculator and Result.

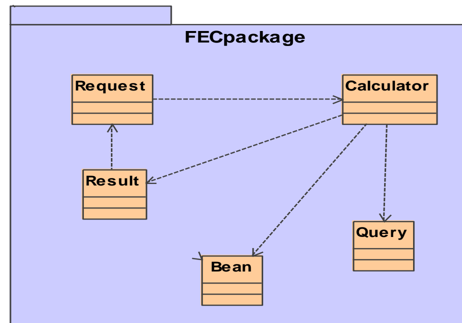

Figure 9. Class dependency diagram for the A design (FEC).

Source: author's own

Without any guidelines and restrictions from a meta-architecture, there are many possible *A* designs. Also, the *A* design in Figures 8 and 9 is not influenced by any technology and may not even be technologically-viable. For the particular case in Figure 8, the actual number of origin/destination communication links, which defines the actual cumulative measure of class dependencies, is 6.

However, the actual measure cannot be used as an indicator of software complexity and software adaptiveness (as they are measured based on potential, not actual, dependencies). The *A* design is a network structure for which the cumulative class dependency *CCD* is given by Equation 1:

$$ {}^{Adesign}_{FEC}CCD = n(n-1) = 5*4 = 20 \qquad \text{Equation 1} $$

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 first *B* design provides a *treatment measurement* for the subject. The treatment applies the architectural framework onto the design and it chooses the technology that conforms as closely as possible to the architectural framework. The FEC is a web based application that significantly depends on the applied programming environment and the related technology. The technology has some impact on how the architectural principles can be applied.

In an ideal scenario, a Presentation object would submit a user's request, together with any data entered by the user, to Controller. This makes Presentation depend on Controller. Controller would instantiate Bean objects and would maintain their state (thus, changes of the Bean's definition affect its Controller; hence, Controller depends on the Bean). Once the Controller completes its computation, it could return (to the original Presentation object) a reference to the Bean object. Presentation could then instantiate another object to display the Bean data. That new Presentation object will get the data from the Bean object (thus Presentation depends on Bean) and render them.

Being a simple application, FEC requires only two classes (JSP pages, to be precise) in the Presentation subsystem, one Controller class, one Bean class, and one Resource class. There is no need for any Entity objects because no memory caching of exchange rates is necessary or desirable.

We will discuss two possible B designs – the first using basic JSP/servlet/JavaBean technology, the second using Jakarta Struts. The processing logic for the basic solution is documented in the sequence diagram in Figure 10. The design uses the Class Naming Principle (CNP) according to which each class name is prefixed with the first letter of the package/subsystem name (hence, e.g., PInput to indicate that the class belongs to the Presentation package/subsystem).

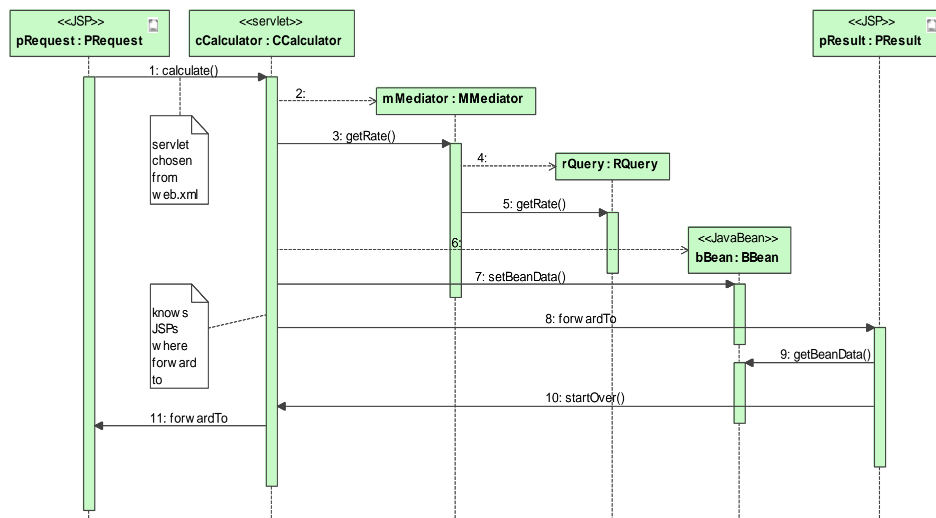

Figure 10. Sequence diagram for the B design using basic technology (FEC).

Source: author's own

The narrative for the sequence diagram in Figure 10 is as follows. PRequest.jsp sends a post (or get) request to the servlet identified for it in the web.xml configuration file – the CCalculator class in our case. CCalculator asks MMediator to getRate() and MMediator delegates this request to RQuery. RQuery gets the exchange rate from the database and the rate value is returned all the way back to CCalculator. CCalculator can now instantiate and populate a BBean object. CCalculator also knows (declares) a JSP page (PResult in our case) to which the response information should go to. PResult can then access the BBean data to render it in the web browser. Finally, CCalculator gets any startOver() messages from PResult and directs them to PRequest.

Setting response information, and other "details" such as accessing and using the user's session and fetching request details in the first place, are the responsibility of the web container. In effect, CCalculator merely communicates to the web container that PResult will need to obtain response details and that the control needs to go back to PRequest when the user wants to start over.

Figure 11 is a class dependency diagram for the sequence diagram in Figure 10. The diagram shows that the FEC design does not comply with the PCBMER framework because of the upward (and cyclic) dependencies between Presentation and Controller. These undesired dependencies are due to the shortcomings of the basic technology used. Fortunately, these shortcomings are addressed by various more powerful technological frameworks. One such framework is Jakarta Struts.

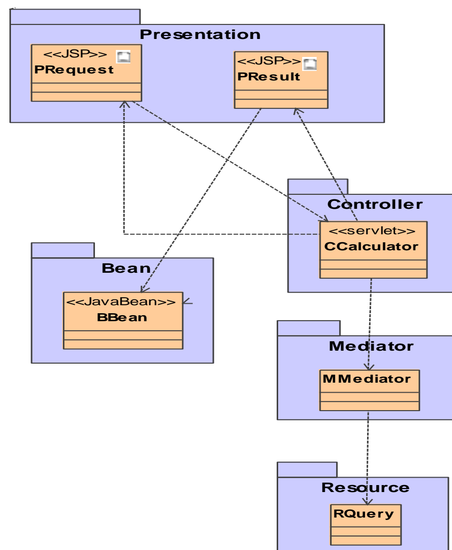

Figure 11. Class dependency diagram for the B design using basic technology (FEC). Source: author's own

Figure 12 represents a PCBMER-compliant Struts design for FEC. Struts provides a brokerage service between Presentation and Controller, which effectively replaces «architecture-managed» dependencies by «technology-managed» dependencies (Maciaszek 2006b). While the former category of dependencies is controlled by developers, the latter is not. All 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. Application developers need to understand them and trust that technology suppliers have control over their software and guarantee no ripple effects on the application code due to changes and upgrades of their system software.

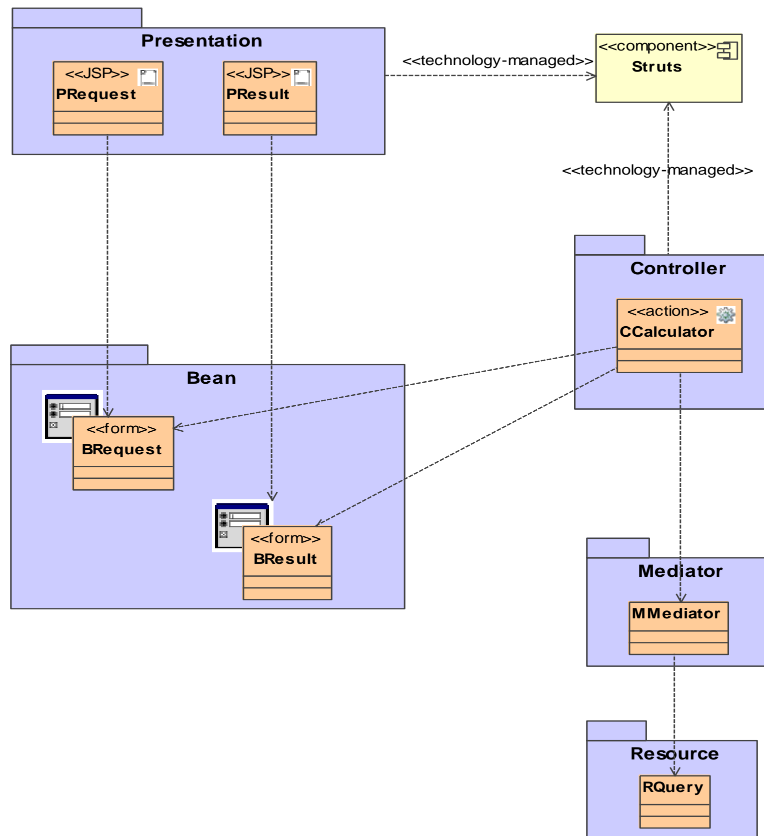

Figure 12. Class dependency diagram for the B design using Struts technology (FEC).
Source: author's own

The design in Figure 12 conforms to the holonic assumptions present in the PCBMER architecture (Figures 1 and 2). The design represents a non-linear holarchy of layers such that a layer can depend on more than one layer and it can provide services to more than one layer. Graph-theoretically, the holarchy is a DAG (Directed Acyclic Graph) network in which the nodes are ordered (parent and child) and there are no cycles (no path returns to the same node).

Let the layers be $l_1, l_2 \ldots l_n$. For any layer l_i, let:

- $size(l_i)$ be the number of objects in l_i
- l'_I be the number of parents of l_i
- $p_j(l_i)$ be the jth parent of l_i

Then, the cumulative class dependency *CCD* for a PCBMER holarchy as in Figure 12 is calculated according to Equation 2:

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

Equation 2

Specifically for the FEC design that uses the Struts technology in Figure 12, the CCD formula of Equation 2 evaluates to 12, as in Equation 3:

$$^{A-Bdesign}_{FEC}CCD = 2 + 8 = 12$$ Equation 3

The first constituent value 2 relates to *potential* dependencies within layers, which are equal 0 for Controller and 1 for Presentation and Bean (even though there are no *actual* dependencies in Presentation and Bean). The second constituent value 8 consists of 4 possible dependencies from Presentation to Bean, 2 possible dependencies from Controller to Bean, 1 dependency from Controller to Mediator and 1 dependency from Mediator to Resource.

The *CCD* (treatment measurement) for the *A-B* design improves by a factor of two the *CCD* (baseline measurement) of the *A* design despite introducing extra bean classes into the design. To know if the applied treatment is long-lasting and to know if the complexity measure is not going to deteriorate once we allow time for changes and extensions to the system, we could continue with the next step in single subject experimental design,

known as the *withdrawal of treatment*, which calculates the second baseline measurement for the subject resulting in the *A-B-A* design. Then we could continue into the *A-B-A-B* design, etc. However, the simplicity of the FEC example does not warrant further "treatments".

Measuring adaptiveness of designs and programs cannot be done manually. Maciaszek and Liong (2003) describe a tool, called *DQ* (Design Quantifier), which is able to analyze any Java program, establish its conformance with a chosen adaptive meta-architecture, compute a complete set of dependency metrics, and visualize the computed values 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. SUMMARY

Research reported in this paper extends our work on adaptive complex systems in a number of important directions. Firstly, it is our most comprehensive attempt so far to align the development of adaptive complex systems with the holonic structures of matter. Secondly, the paper goes beyond software development and addresses also software integration and interoperability (by adding 'S' (services) to the adHOCS acronym). Thirdly, the paper describes how software technologies can contribute to fulfilling adHOCS properties of arborization, reticulation, fixed rules and flexible strategies. Fourthly, the paper illustrates with a case study how the adHOCS' PCBMER meta-architecture reduces software complexity and brings about adaptive software. Fifthly, the paper introduces an improved formula (Equation 2) for computing CCD in PCBMER holarchies without so called hub objects in layers on which more complex layers depend (Maciaszek, 2006b)

Acknowledgement

The author thanks Dr Abhaya Nayak for helpful discussions concerning the PCBMER metrics and his help in formulating Equation 2.

REFERENCES

Alur D., Crupi J. and Malks D., *Core J2EE Patterns: Best Practices and Design Strategies*, 2/e, p. 528, Prentice Hall, 2003.

Babiceanu R. F., Chen, F. F., *Development and Applications of Holonic Manufacturing Systems: a Survey*, “Journal of Intelligent Manufacturing”, 17, pp. 111-131, 2006.

BFSkinner: *A Brief Survey of Operant Behavior*, http://www.bfskinner.org/briefsurvey.html, 2006 (last accessed March 2007).

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

Ferber J., *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*, Addison Wesley, 1999.

Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns. Elements of Reusable Object-Oriented Software*, pp. 396, Addison-Wesley, 1995.

Ganek A. G., Corbi T. A., *The Dawning of the Autonomic Computing Era*, “IBM Systems Journal”, Vol. 42, No. 1, pp. 5-18, 2003.

Harrison W., *Skinner Wasn’t a Software Engineer*, “IEEE Software”, May/June,, pp.5-7, 2005.

Heffner, C. L., *Research Methods*, “AllPsych Online”, http://allpsych.com/researchmethods/index.html, 2004 (last accessed March 2007).

Hirschfeld R., Hanenberg S., *Open Aspects*, “Computer Languages, Systems & Structures”, 32, pp. 87-108, 2005.

Jennings N. R., *On Agent-Based Software Engineering*, “Artificial Intelligence”, 117, pp. 277-296, 2000.

Johnson R., Hoeller J., *Expert One-On-One. J2EE Development without EJB*, Wrox, p. 552, 2004.

Kiczales G. et al, *Aspect-Oriented Programming*, Proc. European Conf. on Object-Oriented Programming (ECOOP 97), LNCS 1242, pp. 220-242, Springer, 1997.

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

Koestler A., *Janus. A Summing Up*, p. 354, Hutchinson of London, 1978.

Koestler A., *The Ghost in the Machine*, p. 384, Hutchinson, 1967.

Krasner G. E., Pope S. T., *A Cookbook for Using the Model View Controller User Interface Paradigm in Smalltalk-80*, “J. Object-Oriented Prog.”, pp.26-49, Aug-Sept, 1988.

Maciaszek L. A., *Database Design and Implementation*, pp. 384, Prentice-Hall, 1990.

Maciaszek L. A., *Development and Integration of Adaptive Complex Enterprise and E-business Systems*, Pearson Education (2007) (in preparation).

Maciaszek L.A., *Adaptive Integration of Enterprise and B2B Applications*, Proceedings of International Conference on Software and Data Technologies (ICSOFT 2006), Setubal, Portugal, Sept 11-14, INSTICC, 2006 a, pp.IS-3–IS-12. (keynote paper; to appear also in a book by Springer).

Maciaszek L. A., *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, ed. K. Sacha, Boston: Springer, 2006b, pp.1-13. (keynote paper at the IFIP Working Conf. on Soft. Eng. Techniques SET 2006, Warsaw, Poland, Oct. 17-20).

Maciaszek L. A., *Requirements Analysis and System Design*, 2/e, p. 504, Addison-Wesley, Harlow England, 2005a.

Maciaszek L. A., *Roundtrip Architectural Modeling*, “Second Asia-Pacific Conference on Conceptual Modelling (APCCM2005)”, Newcastle, Australia, January 30 – February 4, 2005, eds. S. Hartmann and M.Stumper, Australian Computer Science Communications, Vol. 27, No. 6, 2005b, pp. 17-23 (invited paper).

Maciaszek L. A., *Requirements Analysis and Systems Design*, 3/e, Addison-Wesley, Harlow England, 2007 (to appear in June 2007).

Maciaszek L. A., Dampney C. N. G., Getta J. R., *Behavioural Object Clustering*, Future Databases'92, Proc. 2nd Far-East Workshop on Future Database Systems, Kyoto, Japan, eds. Q. Chen, Y. Kambayashi, R. Sacks-Davis, pp. 186-193, World Scientific, 1992a.

Maciaszek L. A., De Troyer O. M. F, Getta J. R., 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, pp. 431-442, 1996a.

Maciaszek L. A., Getta J. R., Bosdriesz J., *Restraining Complexity in Object System Development the "AD-HOC" Approach*, Proc. 5th Int. Conf. on Inf. Syst. Development ISD’96, Gdansk, Poland, pp. 425-435, 1996b.

Maciaszek L. A., Getta J. R., Dampney C. N. G., *From Data Flows to Object Clusters*, Proc. 3rd Int. Conf. on Information Systems Developers Workbench, Sopot, Poland, pp. 349-364, 1992b.

Maciaszek L. A., Liong B. L., *Designing Measurably-Supportable Systems*, Advanced Information Technologies for Management, Research Papers No 986, ed. by E. Niedzielska, H. Dudycz, M. Dyczkowski, pp.120-149, Wroclaw University of Economics, 2003.

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

Martin R. C., *Agile Software Development, Principles, Patterns, and Practices*, p. 529, Prentice-Hall, 2003.

McReynolds L. V., Thompson C. K., *Flexibility of single-subject experimental designs. Part I: Review of the basics of single-subject designs*, “J Speech Hear Disord.”, 51(3), pp. 194-203, 1986.

Mitchell M., *Complex Systems: Network Thinking*, “Artificial Intelligence”, 170, pp. 1194-1212, 2006.

Murphy G., Schwanninger C., *Aspect-Oriented Programming*, “IEEE Soft.”, January-February, pp. 20-23, 2006.

Pichler F., *On the Construction of A. Koestler’s Holarchical Networks*, “EMCSR” April 25-28, www.cast.uni-linz.ac.at/Department/Publications/Pubs2000/PIEMCSR2000.doc, 2000, 10p. (last accessed March 2007).

Roy-Faderman A. *et al*., *Oracle JDeveloper 10g Handbook*, p. 802, McGraw-Hill/Osborne, 2004.

Silberschatz A., Korth H. F., Sudershan S., *Database System Concepts*, 5th ed., pp. 1142, McGraw-Hill, 2006.

Smith J. M., Smith D. C. P., *Database Abstractions: Aggregation and Generalization*, "ACM Trans. Database Syst.", 2, pp. 105-133, 1977.

Teorey T. J., Wei G., Bolton D. L., Koenig J. A., *ER Model Clustering As an Aid for User Communication and Documentation in Database Design*, "Comm. ACM", 8, pp. 975-987, 1989.

Tharumarajah A., Wells A. J., Nemes L., *Comparison of the Bionic, Fractal and Holonic Manufacturing System Concepts*, "Int. J. Comp. Integr. Manufact.", 3, pp. 217-226, 1996.

The Alpbach Symposium: *The Alpbach Symposium 1968. Beyond Reductionism. New Perspectives in Life Sciences*, ed. A. Koestler and J.R. Smythies, Hutchinson, p. 438, 1969.

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

Received: March 2007