Distributed Information Systems Tailorability: A Component Approach

D. Theotokis, G.-D. Kapos, C. Vassilakis, A. Sotiropoulou, G. Gyftodimos
Department of Informatics
University of Athens
{dtheo,gdkapos,costas,anya,geogyf}@di.uoa.gr

Abstract

Distributed software systems need to evolve according to the ever-changing requirements on which they were built. Software systems' tailorability can be achieved in terms of component software. Atoms and molecules the basic constructs of the ATOMA framework, are the building blocks for distributed tailorable component-based software systems. These constructs can be considered as independent agents, that communicate in terms of, unanticipated, connections that are established at run-time, thus forming agent communities. System tailorability can take place at two levels. In high level tailorability whole parts of the functionality of a system, represented as agents, can be altered in order to provide new functionality. At a lower level, the tailorability of an agent itself, that is the tailorability of its functionality, is achieved through a flexible service mapping implementation for rule-based method invocation.

1. Introduction

Information systems consist of parts that co-operate in order to fulfil the users', system's and contexts' requirements. These requirements may change during the lifetime of a system. New ones may emerge, old ones may become obsolete or require modification in order to conform with the evolving nature of the contexts they describe. Called context-dependent variations, these modifications are a desirable and, in many cases, a necessary factor of information systems, particularly when the parts of an information system are distributed.

For instance, consider an application where entities representing people and their behavioural characteristics, that is roles they play in time, are modelled. It is desirable for a person entity to acquire or abandon several roles dynamically, either in conjunction or independently of each other, thus reflecting a person's behaviour within one or more contexts. It is therefore necessary to provide and accommodate in a dynamic manner multiple facets of an entity with their corresponding implementations.

In such a case, the following factors need to be taken under consideration. Firstly, in what way new entities and roles can be incrementally added in the application spaces; secondly, how is the modification of existing entities and roles achieved, so that they can be altered to reflect new behavioural aspects. Thirdly, how the entities' behaviour may be tailored, both at the end users' side and at the implementors' side. In all cases the important issue is to do so in a dynamic manner [14].

All along the evolution of software engineering methodologies and programming languages, the main concern has been the, so-called, separation of concerns principle [5]. The basic idea underlying this principle is the partition of the state space, so that it becomes possible to put the focus on the resulting parts, individually. Once partitioned, the separated pieces must somehow be "connected" to each other, preferably by a "loose" connection that maintains their autonomy as much as possible. The high degree of cohesion combined with the loose coupling of such pieces renders a system more manageable, with respect to maintainability, modifiability and extensibility, particularly when such a system is distributed [10].

We consider system tailorability as the dynamic accommodation of context-dependent behavioural variations in an existing software system. Consequently, during system development the possibility that unanticipated behavioural variations relevant to a system's parts may occur must be taken into account.

In order to deliver the features presented above, we adopt a scheme based on *atoms* and *molecules* (described in detail in the next section) and a rule-based flexible service implementation mapping. Atoma, and molecules facilitate dynamic role assignment and tailorability at the end users' side, whereas flexible service implementation mapping provides the means for customisation at the implementors' side [12]. The overall architecture is illustrated in the following figure:

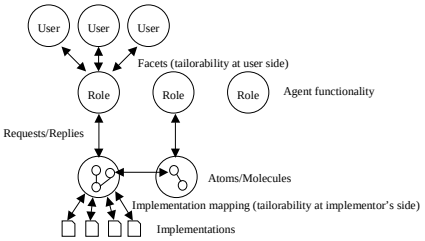

Figure 1. Overall architecture.

The elements of this figure are described in detail in the following sections

2. Component software

Key to distributed tailorable information systems are components, the building units of such systems. In Theotokis et al. [16] components are defined as follows: "An atom (the proposed term for components) is an independent, standalone, customisable, executable software unit of independent production, acquisition, and deployment, that fulfils some specific requirements of a given system." Given this definition, an atom can be thought of as an agent and subsequently the implementation of an IS is a multi-agent system, where agents may be distributed. The combination of atoms, at run-time, results in the formation of molecules, that is, agent communities. Customisability and tailorability are achieved in terms of establishing or abolishing connections between agents and agent communities. Roles, atoms and molecules are the fundamental structures of a framework for component software, called ATOMA, presented in [6,16, 17].

2.1 Roles

A role is a structure that provides a new facet to an entity, i.e. an atom, in terms of additional behaviour. This means that when an atom obtains a new role, it immediately attains the role's context. Roles are not self-existent entities. They can only exist if they are assigned to atoms or other roles (a role can also be assigned to another role, as a new sub-facet of the facet specified by the parent role). An atom (or a role) may have more than one roles of different types or even the same type (qualified roles). Thus, underneath an atom there may exist a whole tree structure of roles. Each of them provides a facet of different context and varying level of detail to the atom. Within this work, we approach the concept of roles, as it is defined in the Component Focused Role Model. A detailed description of the model in question can be found in [6].

2.2. Atoms

From a conceptual perspective, an atom, as a loosely coupled and highly cohesed executable software unit, realises the behavioural aspects (functionality) of a system's part. One could consider atoms and molecules as being the objects in a UML collaboration diagram. Although, the interactions between two objects are made explicit when constructing the diagram, in the ATOMA framework such "interactions" become explicit only when atoms are connected to each other. The behavioural aspects encapsulated by an atom are modelled in terms of the mechanisms provided by the object-oriented programming paradigm (i.e. classes, interfaces, inheritance, etc.) [2,3,13].

In practice, an atom is a wrapper class that encapsulates the behavioural aspects, of a part of the system being modelled, as a component, according to the aforementioned atom definition. A wrapper class realises a wrapper function taking two objects as parameters and returning a new one. The first object is the parameter for the self reference. The second object is the parameter for the super reference. The notion of a wrapper class is depicted in figure 2, where the $\oplus$ operator is the left-referential combination of records defined, as follows:

$$(m \oplus n)(x) = \begin{cases} m(x) & \text{if} \quad x \in dom(m) \\ n(x) & \text{if} \quad x \in dom(n) - dom(m), \\ & \text{where} \quad dom(m) = \{x \mid m(x) \neq \perp\} \\ \perp & \text{otherwise} \end{cases}$$

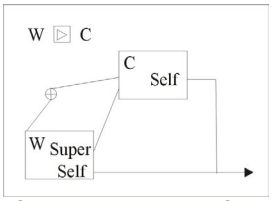

Figure 2.Wrapper Class

We assume that a class can be represented as a record as follows: A record is a finite mapping from a set of labels to a set of values. A record is denoted by

$$\begin{bmatrix} x_1 & \rightarrow & v_1 \\ \cdots & \cdots & \cdots \\ x_n & \rightarrow & v_n \end{bmatrix}$$

with labels x_i and values v_i. Labels not present in the list are mapped to $\perp$. The empty record where all labels are mapped to $\perp$ is denoted by [ ]. Atoms communicate with each other by exchanging messages. Modelling a system in terms of atoms requires its separation into well defined parts that describe a concrete behavioural aspect of the system that can exist independently of, but needs to co-operate with, the remaining parts.

The structure of an atom is presented in figure 3. The thick thunder-like lines (a and g in figure 3) represent the

virtual pins (virtual connection points) of an atom. An atom in reality does not have any such pins. Instead, an atom defines the abstract data types on which the methods of the public interface of its constituent parts operate, as well as the data types of the messages it receives (boxes 1 and 2). Based on this information an atom accepts or rejects the messages it receives. Rejection of messages can only take place when an atom becomes momentarily unavailable due to behavioural changes.

An atom declares also an initially empty list of other atoms that may eventually be the recipients of the atom's messages (box 3 in figure 3). The irregular shapes within the atom represent the roles describing the behavioural aspects of the atom (the self references of roles are passed as parameters). There is no limit as to the number of roles encompassed by an atom. The solid bi-directional lines (d in figure 3) represent the interaction between an atom's constituent parts. Dotted lines (c and e in figure 3) indicate the communication channels that exist between the atom and its roles.

Assuming that atom X is being connected to atom Y, atom Y is the receiver of X's messages. Given a message M sent by X, Y's dispatcher receives it via channel a (phase a). The dispatcher then searches inside the data type and event type structures (boxes 1 and 2 respectively) to determine which of its constituent roles will be the receiver of the message (phase b). Once the recipient is determined, the message is forwarded to it (phase c). The receiver evaluates the message while it is possible to interact with other roles of the atom (phase d). Once processing is completed the recipient dispatches the outcome of the evaluation to the dispatcher (phase e), which in turn determines the recipient(s) of this new message by searching the list of receivers (phase f). Finally, the message is dispatched to the recipient(s) (phase g). Determining the recipients of the message is again based on which atoms are able to process the data type(s) encapsulated in the message. Consequently, an atom's virtual pins can be seen as channels of information flow, entry and exit points. For reasons of efficiency atoms under the ATOMA framework are multi-threaded entities.

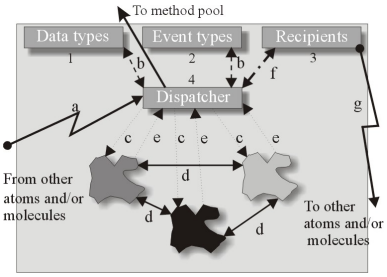

Figure 3: The structure of an atom

Within an atom the self and super references of its constituent parts are bound during construction time, at source code level. This is not however the case with the contents of its list of recipients. An atom is added to another atom's list of recipients when the two of them are connected together at run-time. An important point to make is that apart from their original super reference (their superclass) an atom's constituent parts define yet another super reference, namely wrapped-by, which is bound at an atom's instantiation time to the wrapper class, that acts as their wrapper.

2.3. Molecules

The composition of atoms at run-time results in the construction of molecules: an aggregation of the participating atoms. It must be noted that the term customisation does not only refer to the static properties of an atom, e.g. the value of one of its fields or the background colour of its Graphical User Interface (GUI), if such a GUI exists, but also takes under consideration its behavioural aspects that may vary, that is the context-dependent behavioural variations that may take place during an atom's lifetime. Such variations are modelled as roles. A role is an atom whose purpose is to define the behavioural aspects of a context-dependent variation that may take place for a given atom. The association between atoms and roles is established via the variation-of relationship that assigns roles to atoms.

Molecules are containers. Their constituent parts are atom instances and/or other molecules. These are added to a molecule once they are connected together during run-time. The structure of a molecule is the same as that of an atom. In fact, a molecule is an variation of an atom, the variation being that a molecule's contents and their intercommunication as well as their communication with other atoms and/or molecules is determined after a molecule is generated at run-time. This is achieved through reflection, that is the data and event types of its constituent parts are acquired dynamically during run-time and the structure underlying box 1 and 2 in figure 3 are filled. The list of recipients, as well as the messaging and intercommunications mechanisms, are the same as those of an atom. Consequently, a molecule displays the same structure as that of an atom. The difference is that a molecule's dispatcher dispatches messages to the atoms contained in the molecule.

3. Atom and molecule connectability and communication protocol

Atoms and molecules communicate by exchanging messages once connections between them are established. Such connections can occur at run-time only, by means of scripting techniques (visual and script code based scripting). The important point to notice is that atom and molecule connectivity is not an ad-hoc process. There are three rules governing how atoms and molecules are connected.

3.1. Message and data type compatibility

Given two atoms A and B, if any of the types of the messages produced by atom A are the same as the types of any of the messages accepted by atom B, then A and B can be connected.

Since atoms are considered as executable units that exhibit some behaviour, it follows that this behaviour operates on some input data and produces some output data. Given two atoms A and B, these can be connected to each other if the data types of any of the output data produced by atom A are the same as the data types of any of the input data that atom B expects.

3.2. Method mapping

Atoms may become connected by explicitly mapping the implementation of some behavioural aspect (method) of atom A to the implementation of some behavioural aspect (method) of atom B. Mapping refers to the direct association between two methods belonging to two different atoms. What this really means is that a method or behavioural aspect of an atom may invoke another atom's method without having to encompass such information in either atom during their construction. Method mapping is achieved by generating, at run-time, what we call adjustment units, that is software units similar to adapter classes, that provide the desired association in terms of an invocation mechanism.

The order of these three rules is ascending by the degree of abstraction they embody and is descending by the degree of flexibility they provide. That is, messages' compatibility provides a more abstract and thus generic way to connect atoms, while method based connections is far more flexible, but requires a thorough understanding of an atom's behavioural aspects.

The development of applications that can adapt to their evolving requirements, in other words that can incorporate context-dependent behavioural variations, requires a framework under which such applications may be built in terms of atoms, molecules and roles. In this section we provide the fundamental architectural elements of the atoma framework. In order to establish the requirements for this architecture we provide a more detailed description of what an atom, molecule and role is.

3.3. Atom and molecule communication

As already mentioned, there are three ways for establishing such connections. At the moment let us consider the ones that involve message and data type compatibility. Under the object-oriented paradigm, two objects communicate with each other by exchanging messages. In standard object-oriented programming, an object delegates a message to another object requesting from the latter to perform some action. In doing so, the message source (the object that delegates the message) needs to know the recipients' types at its creation time[7]. In class-based systems, in particular, this has to be explicitly defined at class definition. This however, leaves little or no room for dynamically altering a source's recipients. An alternative to this approach is that of broadcasting [4, 9, 15], were the message source broadcasts, i.e. makes public for all other objects in the object space, its message. Any interested recipients pick this message up. The details on how broadcasting is realised are beyond the scope of this discussion. The use of this approach permits unanticipated recipients of a message to acquire it, although they were never defined as such during the construction of an object, at the expense however of system performance. Broadcasting poses an overhead on the system. Every object in the object space of such a system, apart from performing its behavioural aspects (intrinsic functionality), has to determine whether a message relevant to its functionality has been broadcasted.

In the atoma model we proposed a message passing technique where messages are delivered to specific recipients, the ones contained in an atom's list of recipients, not known at design or development time. Selective forwarding, as we call this technique, enables atoms to exchange messages without having to have prior knowledge of each other. In doing so atoms are divided into two categories: Event listeners and Event sources.

Event listeners are capable of processing any messages they may receive, when they do so, from the atoms they are connected to, provided that the connections were established in accordance to the rules presented above. Event sources are responsible for generating messages and forwarding them to their event listeners which are established during run-time via connections. It may be the case that an atom is both an event source and an event listener at the same time. Under the atoma model, each atom, when constructed, defines an empty list of event listeners, that is message recipients. During its lifetime, event listeners may be added or removed from this list according to the required functionality and behavioural variation that may occur.

Moreover, atoms are also characterised as observers and observables. An observer is an atom that "monitors" behavioural changes (e.g. addition of roles) performed on an observable. When a modification on the behavioural aspects of an observable takes place its observers are notified. Embodied in the structure of an observer is a mechanism through which it can "veto" a given modification, in which case the modification does not take place. Furthermore, if an alteration in an observer's behavioural aspects occur its observables are notified. The semantics of observers and observables in this context are beyond the scope of this paper. What is important to note

though is that, unlike event listeners and sources, an atom cannot be an observer of itself.

4. Elements of the ATOMA architecture.

The realisation of tailorable applications is carried out using the ATOMA development system. This system provides the necessary architectural elements for creating, loading, installing atoms and composing molecules and applications. The architecture of this system is illustrated in figure 4. There are two main parts in this architecture. The registry, and the atom-molecule loader.

4.1. The registry and special purpose lookup tables

The registry is the main component of the system. It consists of three special purpose lookup tables (a1, a2, a3), the adapter class module (a4), the script code parser (a5) and finally the method pool used by the rule processing system.

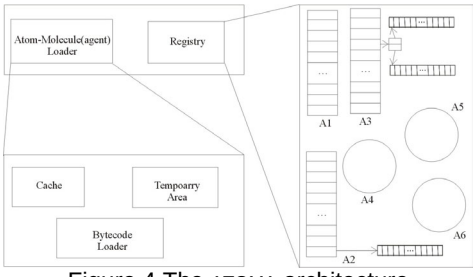

Figure 4 The ATOMA architecture

Special purpose lookup tables (SPLT) are used to store information regarding the atoms and molecules an application consists off, when the application is constructed. The first lookup table, the atom and molecule lookup table, stores the names and references to the classes of the atoms and molecules used in the application. This table's key is the class's name, represented as a string.

The second lookup table, the instance lookup table, contains all atom and molecule instances in the application space. The key here is the class reference. The instance of each class is stored in a skip list [11].

The third lookup table, the data and event type lookup table, exhibits a more complex structure. As its name implies it is used to store information about data and event types supported by atoms and molecules in the application space. This table's key is again the reference to the atom or molecule classes. Its contents are arrays of two elements each holding a skip list. The first skip list is used to store the data types and the other one the event types.

4.2. Adapter classes module and script code parser

The adapter classes module (ACM) contains all adapter classes created by the mapping of methods. These too are organised in terms of special purpose lookup tables, so that efficient indexing and searching can be achieved.

The script code parser's (SCP) purpose is to validate and evaluate script code, used to connect atoms and molecules together. Script code is used to achieve "lazy" evaluation and active/temporal functionality.

4.3. Atom-molecule loader

The atom-molecule loader (AML) is the part of the system responsible for loading into the registry the bytecodes of the atoms and molecules. The bytecodes are retrieved either from the local store or the network space of the application. The latter realises some form of mobility. The AML consists of three parts, the bytecode loader (BL), the temporary area (TA) and the cache (CH).

5. Flexible service implementation mapping

Agent-specific functionality is realised by means of rule-based method invocation mechanism. Within each registry there exists a container of methods. When the functionality of an agent's method changes, the original method is not removed. Instead, the new method is placed in the method container and a "hook" from the original method to its replacement is made.

A services' administrator provides the ability to offer different facets of the IS-provided services, in order to tailor these services to the specialised needs of user groups or situations, facilitating thus end-user customisation. The flexible service implementation mapping, on the other hand, offers customisation facilities at the side of the service provider. Through flexible service implementation mapping, service implementors may develop independently multiple code fragments that realise the service, and service administrators may designate which code fragment will be executed in reply to each request, depending on the request context. The realisations of the service may differ in the quality delivered, in the processes they model etc.

For example, a trading company may provide the order service to its customers, which is advertised as follows:

```
int order(Item item, int amount) raises
sales_denied, not_approved;
```

Through this service, the company's customers may order an amount of some item and be informed of the final price. The company wants to deny any sales to designated customers, while orders worth more than $100.000 must be approved by the manager; thus these

three different cases of ordering, model different business processes, that must be followed. The three different business processes may be coded independently within the following methods:

```
int deny_sales(void) raises sales_denied;
int order(Item item, int amount);
int order_with_approval(Employee authoriser,
Item item, int amount) raises not_approved;
```

The flexible service implementation mapping scheme provides the means to map each call to the advertised (or external) interface to the appropriate code fragment, within the actual (or internal) interface. The separation of an object's interfaces resembles, in some aspects, to that employed by Sena and composition filters [1]. In order to determine which code fragment must be invoked, the flexible service implementation mapping scheme employs a rule-base and a rule processing system. Each rule has the following form:

(object_type, invoked_method_signature, condition, executed_method_signature)

where:

1) *object_type* is the type of the receiver object. Inheritance rules apply as usual , i.e. methods defined for a supertype are eligible for invocation from instances of its subtypes.
2) *invoked_method_signature* is comprised of the method name, the names and types of the formal parameters to the method invocation.
3) *condition* is a Boolean expression which may reference:
   i) *the object state*, i.e. the values of the receiving object's instance properties. For instance, the algorithm used to compute an employee's payroll check may depend on the date of hiring. The receiving object may also be asked to execute some method that returns a result. Normally, such a method should be "read-only"; however, this is not mandatory, and if the method has side effects, these are considered as part of the object's response to the invocation. Any part of the object's state or behavior may be referenced using the notation *self.part_name*.
   ii) *the actual values of the invocation parameters*. Any such value may be referenced using the name of the respective formal parameter, included in the *invoked_method_signature*. In the example presented above, *item* and *amount* may be used to reference the ordered item and the ordered quantity, respectively.
   iii) *environmental data*, such as user credentials, connection security level etc. Each such piece of information may be obtained by supplying the appropriate parameter to the *env_info* function, which is supported by the rule processing system. For example, in a weather forecasting system certain users may be allowed to use a highly accurate -but computationally expensive- method for weather forecasting only during the night or when the machine is idle, whereas in all other cases they are forced to use a less accurate method that saves machine resources.
4) *executed_method_signature* is the signature of the method that will actually be executed. Besides the method name, this signature may include parameters whose values are results of expressions. These expressions are allowed to reference all information available to the third part of the rule (the condition).

All method invocations are intercepted by the rule processing system. Upon receiving a method invocation request of the form object.method(param1, param2, ...), the rule processing system scans the portion of the rule database pertaining to the type of the object and its supertypes to locate a rule for the specific method, having a compatible invoked method signature. When such a rule is located, the condition field of the rule is evaluated and, if the result of the evaluation is "true", the method designated in the executed_method_signature rule field is invoked. Failure to locate a rule for the specific method having a compatible signature and a condition evaluating to true results in an exception, that is forwarded to the requesting object. During rule scanning, the rules defined in subclasses are considered before superclass rules, so a class may override the rules defined by its parents. Additionally, the order in which the rules are listed is significant, since rule processing stops when a matching rule with a condition evaluating to true is found. Rule lists pertaining to a specific method may contain a rule with a true condition as their last element, in order to catch all requests that will not be handled by the preceding rules. Figure 55 presents a sample rule base for the order method.

```
/* Cut off user "Scott" */
(company,   order(Item   item,   int   amount),
            env_info("USER")      =      "Scott",
            deny_sales());
/* Get the necessary permission for orders above
100000 */
(company, order(Item item, int amount),
item.price * amount > 100000,
            order_with_approval(self.manager,
            item, amount));
/* Pass through all other orders, using a "true"
condition */
(company, order(Item item, int amount), true,
order(item, amount));
```

Figure 5 – Sample rule base

The proposed scheme may be employed to implement security (requests that shouldn't be honoured may be mapped to a method that raises an appropriate exception), and user profiles (different code fragments may be available to different users). Moreover, it eases service development, since programmers may focus only on the aspects related to the implementation of the services – staying in line with the aspect-oriented programming paradigm [8]– limits the need for recompilation only to the cases that the modelled processes themselves change,

and facilitates service administration, given that changes to code fragment selection policies require simply the modification of the rule database.

6. Conclusions and future work

The ATOMA architecture with the extensions described in this work provides a flexible and effective approach towards the tailorability of distributed software systems. Roles, atoms, and molecules enable variations-oriented programming, which in turn provides the means for altering a system's functionality at runtime. Considered as agents, atoms and molecules, when connected together form agent communities that allow a system to be modelled as independent, yet co-operating parts. Agent functionality can be customised through the use of rule based method invocation.

Evolution of some part (e.g. some atoms) can result to the evolution of the whole system (the molecule using these atoms). Thus, it is an easy, flexible way for the developer to produce new versions of the software.

The rule processing system is an area, which requires further investigation so that method selection can be achieved in a more intelligent way. In particular fuzzy logic may be employed in order to enhance the rule processing system's capabilities. Mobility of atoms and molecules is also an important matter that requires further research. Finally, persistency needs also to be considered in order to track the behavioural changes that a software system undergoes.

7. References

[1] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa, "Abstracting Objects Interactions Using Composition Filters", vol. 791 of *LNCS*, Springer-Verlag, R. Guerraoui, O. Nierstrasz, and M. Riveill edition, 1993, chapter Object-Based Distributed Processing, pp. 152-184.

[2] G. Booch, *Object-Oriented Design with Applications*, The Benjamin-Cummings Company, 1992.

[3] G. Booch, *Object-Oriented Analysis and Design with Applications*, Addison-Wesley, 1994.

[4] G. F. Coulouris, and J. Dollimore, *Distributed Systems Concepts and Design*, Addison-Wesley, 1988.

[5] D. O. Dijkstra, and C. A. R. Hoare, *Structured Programming*, Academic Press, London, 1972

[6] N. Dragios, G.-D. Kapos, S. Mantzouranis, D. Theotokis, G. Gyftodimos, "Designing Tailorable Information Systems Using a Component Focused Role Model", in *Proc. of the 7th Hellenic Conference on Informatics*, D. I. Fotiadis, and S. D. Nikolopoulos, The University of Ioannina, Ioannina, Greece, August 1999, pp. III.20-III.28.

[7] A. Eliens, *Principles of Object-Oriented Software Development*, Addison-Wesley, 1995.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, and J. Irwin, "Aspect-Oriented Programming", invited talk in M. Aksit, and S. Matsuoka, Proc. of the *11th European Conference on Object-Oriented Programming (ECOOP '97)*, vol. 1241 of LNCS, Springer-Verlag, 1997, pp. 220-243.

[9] M. Mezini, "Supporting evolving objects without giving up classes", in Proc. of the *18th TOOLS Conference*, B. Meyer, C. Minings, and R. Duke, eds., Prentice-Hall, 1995, pp. 183-197.

[10] O. Nierstrasz, and D. Tsichritzis, *Object-Oriented Software Composition*, The Object Oriented Series, Prentice-Hall, 1995.

[11] W. Pugh, "Skip Lists: A Probabilistic Alternative to Balanced Trees", *Communications of the ACM*, 33(6) pp. 668-676, 1990.

[12] T. Reenskaug, E. P. Andersen, A. J. Berre, A. Hurlen, A. Landmark, A. O. Lehne, E. Nordhagen, E. Neww-Ulsteth, G. Oftedal, A. L. Skaar, and P. Stenslet, "OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems", *Journal of Object-Oriented Programming*, October 1992.

[13] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modelling and Design*, Prentice-Hall, 1991.

[14] C. Szyperski, *Component Software. Beyond Object-Oriented Programming*, ACM Press – Addison-Wesley, 1998.

[15] A. S. Tanenbaum, *Modern Operating Systems*, Prentice-Hall, 1992.

[16] D. Theotokis, G. Gyftodimos, and P. Georgiadis, "ATOMS: A Methodology for Component Object Oriented Software Development in the Education Context", in Proc. *International Conference on Object Oriented Information Systems OOIS 96*, Springer-Verlag, South Bank University, London, UK, December 1996, pp. 226-242.

[17] D. Theotokis, G. Gyftodimos, P. Georgiadis, and G. Philokyprou, "ATOMA: A Component Object-Oriented Framework for Computer Based Learning", in Proc. Of the *3rd International Conference on Computer Based Learning in Science (CBLIS '97)*, De Montford University, Leicester, UK, June 1997, pp. G5-G15