

Object Life-Cycle Management in a Highly Flexible Middleware System

Karl Blümlinger, Christof Dallermassl,
Heimo Haub, and Philipp Zambelli

Institute for Information Processing and Computer Supported New Media (IICM)
Graz University of Technology, Austria
Inffeldgasse 16c 8010 Graz, Austria
{kbluem, cdaller, hhaub, pzamb}@iicm.edu

Abstract. This paper describes the object life-cycle management in the Dinopolis middleware system. Advanced object composition is used to support adaptability and extensibility of the running system. Such a high degree of flexibility requires mechanisms to keep objects and the entire system in a consistent state. We show that a fine-grained definition of the object life-cycle helps controlling objects in a highly flexible fashion. Moreover, we discuss how the life-cycle of composed objects can be exploited to model different dynamic scenarios found in distributed, dynamically changing environments.

1 Introduction

Two important capabilities of modern distributed systems are adaptability and extensibility of the running system. In order to be able to react on a dynamically changing, heterogenous environment it has to be possible to reconfigure the system at runtime [15]. Design for late composition [19] and loose coupling has to be applied since developers are confronted with the problem that the concrete environment in which their software is deployed is first known at runtime. Object composition, if not done systematically, results in software which is difficult to manage and hard to understand. Composed objects have to be built up methodically and the system has to keep them in a consistent state. A well-defined life-cycle and a systematic way for restructuring composite objects is necessary to ensure integrity of the whole system. In this paper, we describe how the life-cycle of objects is controlled in *Dinopolis*. The Dinopolis middleware system is a highly modularized, distributed object system providing seamless integration and virtualization for arbitrary external systems. The system, formerly called DINO (see [2]), has been developed since 1997 as the platform for *MTP* (Medical Telematics Platform) (see [1]). MTP is the first system which manages distributed medical patient records. A first prototype of MTP, fully implemented in Java, was presented at the CeBit 1999 in Hannover and produced lot of interest among medical institution. Since then, researchers at the German Aerospace Center (DLR) and at the IICM have intensively worked on the design of Dinopolis, which will be the core of the first production release of MTP.

In this article, we focus on one of the central parts of the Dinopolis system: the *object management module*. It is responsible for the internal structure of *Dinopolis objects*, controls their life-cycle and handles structural operations on them. The object management module is part of the kernel of Dinopolis and all calls which reach the kernel are already security checked in the *Kernel Access Layer* which is the layer between kernel space and user space. The Dinopolis kernel itself consists of several kernel-modules, each of them have been assigned certain responsibilities. The Dinopolis object is the part of the system which is requested by users and/or applications (always through the kernel access layer) and its purpose is to provide the functionality of the different kernel modules to applications. For a description of the overall system architecture and all sub-modules please refer to [5]. We start with a description of the main characteristics of Dinopolis objects and describe the dynamically changing environment they reside in (section 2). Dinopolis objects are, by their nature, *long-living* objects and we show how we use advanced object composition to model their life-cycle (section 3).

2 The Dinopolis Object

Dinopolis objects are the entities which are requested by applications. As already mentioned above, Dinopolis objects are long-living objects (see [18]). Their life-cycle is not limited to their existence in memory. They are associated with a robust, globally unique handle which is kept stable even if they are removed from memory or moved around in the system (see [4] [5]). Object-mobility [18] is one of the key features of Dinopolis objects, since they encapsulate medium-sized entities as, for example, an XML file stored in a file-system. This points out another strong feature of Dinopolis: The system provides a possibility to integrate arbitrary *external systems* [5]. This includes but is not limited to file-systems, databases, servers, services in a broad sense, or legacy mainframe applications which have to be migrated to a distributed environment. Moreover, external systems can be combined to so-called *virtual systems*. A virtual system then provides the superset of functionality of a number of external systems. As an example, consider a system which combines a data-base and an x-ray equipment. A Dinopolis object represents a certain entity in a virtual system and encapsulates its functionality and data. As an example, consider an x-ray image combined with some meta-data stored in the data-base. Although Dinopolis objects can be moved around in the system, they always have a current location which is defined by a specific virtual system.

The following base characteristics of the internal structure of Dinopolis objects can be outlined:

- *Addressing*: Every object can be asked for its globally unique handle (GUH) which localizes and identifies an object during its whole life-cycle. It is important to mention that a GUH uniquely addresses a Dinopolis object in a distributed environment even if it is moved around in the system (e.g.

across the network). In the context of object-role modelling the GUH can be seen as a globally unique, stable object-identity (see [17]) which is used to address objects across network boundaries. Please see [5] and [4] for a detailed description of the Dinopolis addressing mechanism.

- *Content*: A Dinopolis object encapsulates data in a broad sense like a document, an entry in a data-base or streaming data. Content is stored in a virtual system.
- *Meta-Data*: Objects have a hierarchically structured set of key-value pairs of meta-data. Meta-data may have different origins, as it may come from applications, from the middleware system or from a virtual system.
- *Operations and Services*: Objects provide certain functionality through their interface(s). We distinguish operations (= method calls) on objects and services such as graphical user-dialogs provided by external systems which can be used in applications [5]. Operations include methods offered by the middleware system itself, like support for distributed transactions or versioning, or they may be provided by the virtual system. If required, it is possible to pass through the full functionality of a certain external system. This is useful if applications show in-depth knowledge about a certain external system.
- *Access to Interrelations between Objects*: Interrelations are used to navigate through the system and to model relationships amongst Dinopolis objects. An object gives access to its interrelations. Interrelations may have different origins: They can be mapped from external systems or they can, for example, be explicitly set by applications. Since interrelations use the Dinopolis addressing mechanisms they are kept stable even if objects are moved around in the system.
- *Active Objects*: Dinopolis objects may represent active entities in external systems. Examples are reminders, calendars or a chat application. If applications are interested in certain events they may subscribe for different aspects as observers (see [9]).

The above definitions mainly describe *what* applications expect from a Dinopolis object. Although some parts can be considered to be part of a static interface, it is absolutely impossible to define statically *how* the implementations look like. One of the few predictions to be made is that the implementations will come from different modules in the system. For example, we know that the content is stored in any virtual system and that interrelations are managed by the interrelation module (see [5]). But we do not even know how these modules are configured in a certain running system and which functionality they actually provide. As an example, consider that every Dinopolis instance may have a different set of virtual systems. The problem becomes even worse if we consider that the objects can be moved around in the system, including that they are moved across network boundaries. Overall, it can be said that the concrete structure of a certain Dinopolis object may change during its life-cycle and that a static implementation of its functionality is impossible. As a result of the above considerations, Dinopolis objects have to be composed at runtime. This technique named *late composition* is known from component-oriented programming [18]

[19]. Hence, the decision was made to develop so-called 'in-house' components (see [18]), which are the object fragments of the Dinopolis object. They are used to build up Dinopolis objects at runtime and it allows to control their life-cycle individually.

3 The Life-Cycle of Dinopolis Objects

A life-cycle for objects is also defined in other systems, as for example for Enterprise Java Beans [10] or for CORBA objects [7]. In Dinopolis, the life-cycle of objects is more fine-grained and it can be controlled in a more flexible fashion. As already mentioned, Dinopolis objects have a long-termed life-cycle which is not limited to construction and destruction of objects and it is, for instance, possible to let them "live" as long as their persistent data exists. Moreover, it lets you define at runtime which actions are triggered for each transition in the life-cycle. As an example you can define at runtime which tasks are carried out before an object is removed from memory. Regarding the language context our approach focuses on statically-typed (see [14]), main-stream, object-oriented programming languages, namely C++ and Java. As they lack support for runtime composition we will point out which modifications/additions would be necessary in these languages as it has been introduced for example by [12].

3.1 Object Composition

As already stated in the previous section, Dinopolis objects are built up at runtime. In statically typed object-oriented programming language this leads to a delegation mechanism since it is the only possibility to change the behavior of an object at runtime [9]. The approach using delegation is not a new one: dynamicTAO (see [8]) is an implementation of a CORBA ORB [7] using the strategy design [9] pattern. It manages to reconfigure the ORB at runtime. [16] shows in what a way disciplined delegation can be used in so-called split objects which consist of several pieces, and calls are delegated among these objects. [12] analyzes type-safe integration of dynamic delegation into a class-based object model. The approach of delegation has some drawbacks which are discussed in detail in [18] and [13]. Therefore in Dinopolis we use a well-structured way for composing objects and for delegating calls among them.

The basic concept is that Dinopolis objects are built up from scratch starting with an empty template. The empty template provides a method to attach instances of so-called *Definitions*. Definitions are programmed statically (e.g. as a Java class) and they implement one or more *Declarations* (e.g. tagged or marked Java interfaces). Declarations and Definitions are programmed by a module programmer. Among other things, the programmer has to supplement additional information about all implemented Declarations (= the provided interfaces).

In the running system, each system module can be asked for an instance of a certain Definition. The module instantiates and initializes the Definition and

returns it to the object management module which attaches it to the Dinopolis object.

Internally, every Dinopolis object has a table similar to the *virtual method table* created by OO compilers (see [6]). This table, called *explicit virtual table* (EVT) in Dinopolis, is built up at runtime and contains a list of Definitions for each declared method. On the other hand, every object has a list of Declarations which are “shown” to applications via special proxies.

We defined the following semantic for attaching an instance of a Definition: When a Definition is attached to the Dinopolis object, first all new Declarations are added to the list of Declarations. Secondly for each declared method the Definition is appended to the list of Definitions. If a method is called on the Dinopolis object, a lookup on the EVT is done and the call is delegated to the most recently appended Definition. This way, the mechanism supports method overriding at runtime. Moreover, it allows Definitions to be detached again.

This directly leads to the closely related topic of object role modelling [17]. While the object life-cycle is determined by several states and well-defined transitions between these states, an object can play several roles and change them between object creation and object deletion. Roles can be modelled by attaching and detaching instances of Definitions. Please refer to [11] for a comparison of different technologies for object role modelling. The main difference to these technologies is that one Dinopolis object consists of several instances of components which can be added and removed at runtime. The problem of one single object-identity [17] is resolved by the Dinopolis addressing mechanism since every object is addressed by a globally unique handle (see [4]). Role-inspection is supported since every object can be asked for a list of currently attached Declarations and Definitions. Applications may apply a special cast on the object and access it in a type-safe manner since our object composition mechanisms always ensures that the type-system of the underlying object model is not violated. The strength of component-technology in this context is actually the support for late-binding since the EVT (and therefore method dispatching) can be changed at runtime by simply attaching and detaching Definitions in order to dynamically change the role of an object.

3.2 Creating a New Object

If a Dinopolis object is newly created the object management module determines the location (= a specific virtual system) of the new object and creates an empty skeleton of an object. The only methods available are those of the mechanism for attaching and detaching Definitions. The object management attaches an initial Definition which contains methods for controlling the life-cycle. They are hidden from applications since the life-cycle has to be controlled by the system. Then the Dinopolis object is added to the object management module’s internal tables and registers the new Dinopolis object at the address management module (see [5] and [4]). Next, the object management calls the *creator* method on the Dinopolis object. First, the creator determines the correct *static type*. The static type is defined by a set of Declarations and Definitions which cannot be

removed from the object and which stay the same during the whole life-cycle of the object and it is always possible to cast the object on its static type. In the context of object-role modelling the static type is often referred to as the *natural type* (see [17]). Furthermore, the creator determines the location of the object's persistent data. Object creation may include that space for the object's persistent data is allocated in an external system (e.g. a new file is created in a file-system). The creator can only be called once during a Dinopolis object's life-cycle. At the end, the creator calls the *constructor* method on the Dinopolis object. The constructor does the following: first it adds all static Definitions and marks them as static as they may not be removed from the object. According to the current system configuration, the constructor requests Definitions from the different system modules. Each system module instantiates and initializes its Definitions and returns them to the constructor which, for its part, attaches them to the object. Additionally, the constructor may add the Dinopolis object as an observer for an active entity in a virtual system. It also triggers that interrelations are attached to the Dinopolis object. Finally the constructor checks if the Dinopolis object is in a consistent state and tells the object manager that it may return the Dinopolis object to the requestor. The constructor may be called arbitrarily often during the life-cycle of a Dinopolis object. Please note that the object management module may decide at runtime which concrete implementation of the constructor is actually used.

3.3 Structural Operations

When an object is in memory the object management has to handle structural operations. Examples are, moving an object to another location, that the system configuration is changed or that the object wants to play an additional role. This may require that Definitions are attached or detached to/from the object. When the object has been returned to the requestor the object management does distributed reference counting (see [18]) and may decide whether a structural operation can be carried out. All structural operations are done in a transaction-based fashion and attaching and detaching of Definitions is restricted by several rules. First, there exist implicit rules, which, for example, avoid ambiguities if a method with the same signature is declared in different Declarations. Secondly, the object management has to regard so-called explicit dependencies. Explicit dependencies may exist among different Definitions since Definition programmers can access other Definitions in the same Dinopolis object: Definitions may issue special *scoped* and *supered* calls on other Definitions. Therefore, the programmer has to provide meta-information about these *required* Definitions. With this information, the system may check whether a required Definition is actually attached to the object at runtime. As an example of a *supered* call, consider a logging module which wants to log certain operations. The logging module triggers that its own Definition is attached to the object. Its methods are implemented in a way, so that they first submit their logging information and then they issue a *scoped* call which invokes the method on the previously

attached Definition. Another example is a Definition which does version controlling. In order to be able to react to certain changes, applications may subscribe as observers for notifications about structural changes in the Dinopolis object. The most powerful functionality provides a method simulating the *this* member variable in C++ and Java. It allows programmers to access the entire Dinopolis object. If this method is used the programmer has to provide meta-information about all *required Declarations* which have to be available in a certain Dinopolis object.

The full flexibility can be demonstrated when we consider a move operation. The system may decide upon the static type of an object and the configuration at the destination location whether the object can be moved. Generally there exist various different scenarios. If objects are moved inside a certain virtual system this might not even result in a change of the structure of the object. If an object is moved from one virtual system to another this may require that all Definitions of the source virtual system are replaced by Definitions of the new virtual system. Generally, moving an object is associated with a transfer of the content and meta-data encapsulated by a certain object (= its persistent data). The object management may therefore access the object via its static interface and request the content and its meta-data and store it at the new location. Finally, let us consider a move operation across network boundaries: the object management builds up an intermediate object at the new location and triggers all data to be read from the source and stores it at the destination. In this case the object management works together with the network management module and has a possibility to transparently replace the local object with a network proxy and vice versa (see also [5]).

3.4 Remove an Object from Memory

As already mentioned, the object management does distributed reference counting. If no more references to an object exist it may decide to remove it from memory. Therefore, it calls the *destructor* on the object. Again it is possible to define at runtime which actions are triggered before an object is removed from memory. Its default behavior is to store the object's data to its persistent location and to do necessary clean-up operations to remove the object from memory. The destructor can be called an infinite number of times during the life-cycle of an object. Storing an object is another point where flexibility comes into play: in a distributed system two different opinions about the characteristic of a store operation may exist [5]. First, it can be desired that an object's persistent data is implicitly stored by the system when it has been changed compared to its last version. The second possibility is that data storage has to be explicitly triggered by an application. If this object is not explicitly stored it is discarded without saving the changes. This behavior is known, for example, from document editors which do not store the document if the user does not explicitly trigger this action. In Dinopolis it is easy to choose the appropriate behavior by overriding the corresponding method.

3.5 Delete a Dinopolis Object

At the end of the life-cycle of an object the *deletor* is called on the object. The deletor has a different semantic than the destructor. While the destructor is used to remove objects which are in memory the deletor is semantically associated with the deletion of the object's persistent data. Since an object may be accessed simultaneously by several applications the system may react in different ways (see [5]):

- Since the object management does distribute reference counting it may forbid object deletion if any other references to the object still exist.
- The system may hide object deletion from other applications behind the scenes and may delay the operation until no more references to the object exist. At the same time it has to ensure that the object is no more requestable.
- A third possibility is to replace the object with a valid “no longer existing” object.

Again, it is possible to define at runtime how object deletion is actually carried out by simply attaching and detaching Definitions. The deletor is only called once during the life-cycle and always follows a preceding call to the destructor.

4 Summary

One of the base requirements of Dinopolis is that it has to be as flexible as possible and that the system can be reconfigured at runtime. Component software technology is applied to support late composition since class-based object-oriented programming is not flexible enough [19]. Object composition, together with a disciplined delegation mechanism, is a possibility to achieve such a high degree of flexibility. The resulting system becomes much more complex as systems using subclassing. Management of composite objects results in systems which have to ensure integrity and consistency of their objects at runtime. Verifications and checks which are normally done statically by a compiler have to be handled at runtime by the underlying component architecture. For a system using object composition and delegation, a well-defined specification of the life-cycle of objects is essential, since otherwise it is impossible to control such dynamic systems. Moreover, a life-cycle can be exploited to model highly dynamic scenarios which can be found in today's distributed systems. The work presented in this paper is a solution suited for the needs of Dinopolis but we think that the approach of modelling well-known static concepts like method overriding, polymorphism or the destructor is easier to understand for programmers familiar with class-based object-orientation.

As proof of the concept, a prototype of the object life-cycle management presented in this article has been implemented in Java using a preprocessor written in *OpenJava*. Since meta-information about a certain Definition (e.g. required Declarations) still has to be provided explicitly, it would be desirable

for current statically typed object-oriented languages such as Java to support advanced object composition within their language, as introduced, for example, by [20].

References

1. Aly F., Bethke K., Bartels E., Novotny J., Padeken D., Schmaranz K., Schwartzmann D., Wilke D., Wirtz M.: *Medical Intranets for Telemedicine Services: Concepts and Solutions*, Proceedings G7 Meeting “The Impact of Telemedicine on Health Care Management”, Regensburg (1998), available online at <http://www.uni-regensburg.de/Fakultaeten/Medizin/Uch/g7/program/mon.htm>.
2. D. Freismuth, D. Helic, G. Meszaros, K. Schmaranz K., B. Zwantschko B.: DINO - Distributed Interactive Network Objects - The Java Approach, Proceedings Ed-Media '97, Calgary (1997), available online at http://www.iicm.edu/liberation/iicm_papers/edmed97/dino.html
3. Klaus Schmaranz: On Second Generation Distributed Component Systems, J.UCS Vol.8, No.1, 97–116 (2002).
4. Klaus Schmaranz: DOLSA - A Robust Algorithm for Massively Distributed, Dynamic Object-Lookup Services, submitted to J.UCS.
5. Klaus Schmaranz: Dinopolis – A Massively Distributable Componentware System, Habilitation Thesis, June (2002)
6. Timothy Budd: An introduction to object-oriented programming. Second Edition. Addison Wesley Longman Inc., 1998.
7. Object Management Group (OMG): Common Object Request Broker Architecture: Core Specification, Version 3.0.2, December 2002.
8. Manuel Romn and Fabio Kon and Roy H. Campbell: Reflective Middleware: From Your Desk to Your Hand, University of Illinois at Urbana-Champaign, 2001.
9. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software, Addison Wesley (1995)
10. Sun Microsystems: Enterprise Java Beans Specification Version 2.1. Proposed Final Draft, August 2002.
11. G. Kappel, W. Retschitzegger and W. Schwinger: A comparison of role mechanisms in object-oriented modeling. in: Modellierung '98 Bericht Nr. 6/98-I (Angewandte Mathematik und Informatik, Universität Münster 1998) 105–109.
12. Günther Kniesel: Type-safe Delegation for Dynamic Component Adaptation - Position Paper for ECOOP 98 Workshop on Component-Oriented Programming, 1998.
13. Henry Liebermann: Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems, in Proceedings of First ACM Conference on Object-Oriented Programming Systems, Languages and Applications, Portland, OR, SIGPLAN Notices, September 1986.
14. Bertrand Meyer: Object-Oriented Software Construction - Second Edition. Prentice Hall, New York, 1997.
15. Nikos Parlavantzas, Geoff Coulson, Mike Clarke, and Gordon Blair. Towards a Reflective Component Based Middleware Architecture. In Walter Cazzola, Shigeru Chiba, and Thomas Ledoux, editors, On-Line Proceedings of ECOOP'2000 Workshop on Reflection and Metalevel Architectures, June 2000.
16. D. Bardou and C. Dony. Split Objects: A Disciplined Use of Delegation Within Objects. In Proceedings of OOPSLA'96, Sans Jose, California. Special Issue of ACM SIGPLAN Notices (31)10, pages 122137, 1996.

17. Friedrich Steimann: On the representation of roles in object-oriented and conceptual modelling. In *Jornal: Data Knowledge Engineering* Vol. 35/1 p. 83-106, 2000.
18. Clemes Szyperski: *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1997.
19. Wolfgang Weck: Inheritance Using Contracts Object Composition, *Proceedings ECOOP Workshops*, (1997).
20. Matthias Zenger: Type-safe prototype-based component evolution. Technical report, *Ecole Polytechnique Federale de Lausanne, Switzerland*, April 2002.