

Droplets: Breaking Monolithic Applications Apart

Luca Deri, IBM Zurich Research Laboratory¹

Most current systems are built around a monolithic structure that integrates all the services offered. Such a paradigm has been used for many years and is now proving to have many limitations. This is partly because, with the advent of distributed systems, there is a need to distribute the intelligence and control among applications and network entities. Also due to heterogeneous data or systems that an application has to manage, it costs too much to rebuild the application every time a new service has to be supported. It would be better to add such services while the application is running.

This paper attempts to show the benefits of a component-based application against a monolithic one and it introduces a new type of software components, called droplets based on dynamically loadable libraries. Droplets are used to implement selected application services and have the ability to be modified or added to the application while it is running, thus allowing to dynamically modify the application behavior or to extend its functionality at runtime. Finally the paper shows how to design and implement droplet-based applications and demonstrates how the droplet paradigm has been successfully integrated in an existing commercial application.

Keywords: Software Components, Object-Oriented Programming (C++), Distributed Applications, Shared Libraries.

1. Introduction

The traditional object-oriented (OO) programming vision is a bit vague on the subject of reuse. In a growing and changing world, our diverse and complex needs often vary from day to day, project to project, and role to role. Developers are supposed to write classes that may be reused in other applications, projects and platforms. Most object-oriented languages lack the means of packages that are containers for source files defining a certain entity such as a class, and often objects are distributed in binary form and are written for a specific task that makes it difficult to reuse them in another context.

Producing software is expensive because it is labor-intensive and requires highly skilled work. Thus reusability is becoming increasingly desirable. Even more important however, is to write applications that can be customized by the end-user and tailored to changing requirements. It does not make sense to pressure customers into buying an application that can do much more than the user needs. It may confuse a customer to have

¹IBM Research Division, Zurich Research Laboratory, Säumerstrasse 4, 8803 Rüschlikon, Switzerland. Email: lde@zurich.ibm.com.

to pay for an application that will be only partially used. What is certainly better is to release a basic application and allow the customer to extend it by buying or writing additional components. Consider the situation with computers: it is possible to buy a computer having a basic configuration and then add devices and software later on.

With the advent of open and distributed computing it is becoming obvious that a monolithic application can hardly survive due to the intrinsic limitations of its design. Especially in the context of network management, there is a need to build open applications able to accommodate future demands by adding new functionality. A monolithic application is by nature self-contained and cannot be easily extended.

There is another aspect that is gaining importance: mobility. Despite the fact that the market is trying to build small and powerful machines, it is sometimes impossible to install big monolithic applications on such platforms. The market demands small configurable applications that can be composed directly by customers according to their needs.

This paper will show how to break monolithic applications apart using software components. It describes a new type of software components called droplets, based on dynamically loadable libraries, used to implement selected application services and have the ability to be modified or added to the running application at runtime. This allows to extend an application or to modify its behaviour without the need to shut it down, modify it and then restart it. The ability to reload droplets makes them different from conventional software components that once loaded cannot be modified. Finally the paper shows how the droplet paradigm has been integrated in an existing commercial application and it outlines the benefits and the drawback of its usage.

2. Towards software components

Most applications are built around a block that does everything. These applications work to satisfaction and are still doing their job reasonably well. Nevertheless this paradigm is reaching its limits in terms of code reusability, extensibility, configuration, and especially speed and size. One reason is that such a paradigm specifies that the application contain the entire functionality even if much of the functionality is required only for very peculiar tasks. For example, most word-processing programs include a formula editor, a chart package, and many other tools that might never be used by the average customer. Such a customer nevertheless has to pay for unused functions that consume disk and memory space. This paradigm also requires that every extension to the initial design be integrated by the application developer, who is the only person able to access the source code. Many times the customer, and not the developer, is the only one who best knows the requirements. The natural consequence of this would be to offer an interface that allows the customer the capability to add new functionality to the monolithic block and that defines a migration path towards compound applications.

Splitting an application into basic components may be the solution. Just like a child does with Lego™, a compound application is built with many simple blocks - called components - rather than being a monolithic entity. Once the application is built, it works like a monolithic one but has the great advantage that it can be easily modified and extended, adding further pieces or replacing old components with new ones. Component software solves these and other problems by creating a system where it is easy and inexpensive to bring pieces of functionality together, allowing software to be tailored to meet individual needs. The old Latin proverb "Divide et impera" can now be changed to "Build 'n play".

2.1 Components: What are they?

In order to build a compound application we must define the application in terms of components. A building block is called a *component* if it provides a single service to the application. A component can also be regarded as the atom of the application. It is up to the developer to identify the granularity of the components. The smaller the granularity, the easier component reuse is because the component provides a generic service. This approach has certain side effects however, because very generic services have to be composed in order to provide the service required, and a large number of active components may have a negative impact on the performance. Often when an OO language is used, a component can be defined in terms of a class. The main difference between a class and a component is felt when new classes are defined. A new class usually inherits from other classes and therefore specializes the parent class.

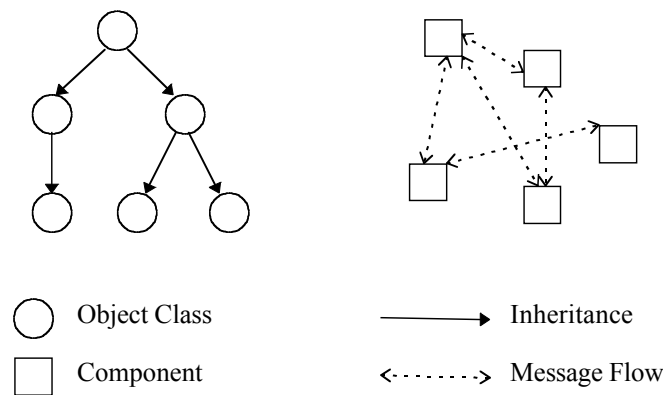


Fig. 1: Classes and Components

A component, on the other hand, adds further services and functionality to the system, often exploiting existing services provided by other components. This is done by exchanging messages containing operation requests with other components. The core idea is that a component provides new services by collaborating with peers, whereas classes do this using inheritance. Such a statement does not mean, like in some object

models², that inheritance is not supported. It means that, internally, a component can use inheritance to implement its logic but, whenever it has to interact with other entities, it does so by using the component interface and not by defining additional methods or class names. In other words the interface inheritance is supported but not the implementation inheritance. Therefore a new component provides new services by giving other components the opportunity to use its services through the component interface, whereas a new class provides new methods and a new starting point for further specialization.

A *droplet*³ is a software component having the following specific properties:

- it is not statically linked to the application but it is loaded at runtime,
- it has the ability to be replaced (i.e. a new version of the droplet can replace a previous one) at runtime while the application is running⁴,
- it has a well-defined interface that makes it possible to communicate with other droplets independently from the type of the services provided.

The droplet interface defines the services provided, the format of the messages that will be exchanged with the outside world, and additional information needed to load the droplet. Note that, whereas a class is an integral part of the application, a droplet requires an application in order to live even if the application can exist without the droplet. This is because the droplet adds functionality and services to the application but the application can exist independently of the number and the type of droplets.

3. Droplet Interface

Because a droplet-based application does not know at compilation time what services it will have to provide, its design has to be split into two parts: the core application, which provides the generic and the basic services, and the droplets. Note that the droplets, and not only the core application, can provide services that can be used by other droplets. For instance writing a WWW⁵ browser, the common services are the user interface, the routines that handle the basic communication (e.g. TCP/IP

²Microsoft's COM (Component Object Model) does not support inheritance. This choice is dictated by the belief that implementation inheritance is not an appropriate relationship between independently developed software components distributed in binary form.

³ The term droplet does not have to be confused with the MacPerl droplets, mini applications what work in conjunction with MacPerl.

⁴In order to activate a droplet it is necessary to drop it into a certain directory monitored by the running application. Hence the term droplet.

⁵The World Wide Web (WWW) is a distributed hypermedia system.

socket creation, DNS⁶ lookup), and additional general utility services such as encoding/decoding of URL⁷s or the handling of HTTP⁸ requests. Other services could be implemented by droplets. For example there could be a droplet for FTP⁹, another for GOPHER¹⁰ and so on. Each droplet exploits the basic services provided by the browser and provides additional services.

The heterogeneous nature of droplet-provided services exactly identifies the boundary between the core application and the droplets. The core application is the most important part of the whole application because it is the part that never changes. Nevertheless it has to be general enough to accommodate current and future droplets.

Every routine that can be considered to be of public interest has to be added to the core application and the data has to be accessed indirectly through the core application, or via a method or procedure call. A droplet should not be allowed to invoke functions directly or issue service requests provided by other droplets. Every request has to be sent from the droplet to the core application that invokes the requested service. This mechanism works based on the assumption that every communication has to pass through the core application because it is the only entity that knows how to locate and request services. Thanks to this, a droplet can use a service without knowing where it is really provided. Either the core application or another droplet may even act as a proxy for another remote entity that provides the real service.

3.1 Method and Function Resolution

Before discussing how the droplet interface has to be implemented, it is important to understand how droplets can invoke external methods and functions not known at compilation time. This mechanism is called method/function resolution and entails the step of determining which procedure to execute in response to a method invocation/function call. There are a few techniques for performing the resolution:

- offset resolution

⁶The Domain Name Server (DNS) is a server able to translate numeric TCP/IP address into symbolic ones and vice-versa. For further information consult RFC 1101.

⁷The Universal Resource Locator (URL) is used by the WWW to specify where a certain resource is located. Its form is <resource type>::<resource location>. For instance the URL for the IBM main WWW server is <http://www.ibm.com/>. For further information consult RFC 1738.

⁸The Hypertext Transfer Protocol (HTTP) is the protocol used by the WWW to transfer information.

⁹The File Transfer Protocol (FTP) is a protocol that allows to transfer files between two computers. For further information consult RFC 542.

¹⁰Gopher is a software system that provides on-line access to documents and other sources of information that reside on the Internet. For further information consult RFC 1436.

- name-lookup resolution
- dispatch-function resolution.

The offset resolution is the fastest technique but also the most constrained because it requires that the name of the method/function to be invoked and, in the case of OOP, the name of the class that introduces the method be known at compile time.

The name-lookup resolution, similar to resolution techniques used by Objective-C/Smalltalk, is more flexible than the offset resolution and can be used when one of these conditions is verified:

- the method/function name is not known until run time
- the method is added to the class interface at run time
- the name of the class introducing the method is not known until run time.

The dispatch-function resolution is the slowest and most flexible technique. It allows method resolution to be based on arbitrary rules associated with the class of which the receiving object is an instance.

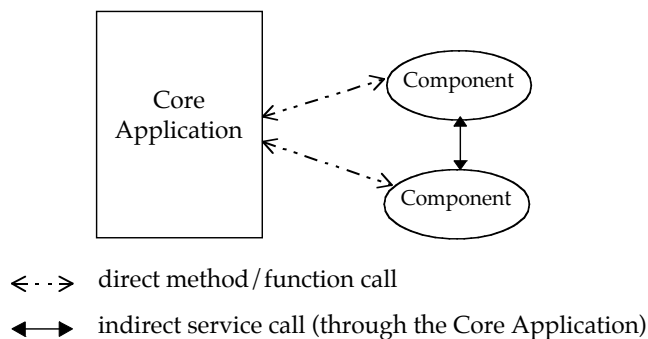


Fig. 2: Direct and Indirect Calls

The droplet issues direct (offset resolution based) and indirect (name-lookup resolution based) calls. The direct call is used when the droplet invokes functions provided by the core application. A droplet is a sort of small application linked at run time with the core application. It can use functions/methods and inherit classes from the core application and issue service calls. When a droplet calls a core application provided function/method, the offset resolution technique is used because the name of this function/method is known at compilation time, and the linker can resolve these references based upon such information. If a droplet requests a service that is not provided by the core application, the indirect call is used. In this case the droplet invokes a core application function/method that takes the service name and the parameters as input. Then the core application function/method performs the name-lookup and, if the requested service has been found, it calls the requested service and returns the results of the call to the droplet that issued the request. In practice, a service can be used by a droplet not only if the service exists but also if the droplet has access to the service. It does not make too much sense to allow droplets to use all the available services without enforcing some basic

security rules: the droplet has to have the necessary access rights in order to access critical or privileged services.

The following example shows how a droplet calls an external service named ABC:

```
if(coreAppl->LookupService("ABC") != 0)
    return SERVICE_NOT_AVAILABLE; /* Service not available */
else
{
    if(coreAppl->ExecuteService("ABC", <input params>,
                               <output params>, <access rights>) != 0)
        return SERVICE_ERROR; /* Service call returned an error */
    else
        return NO_ERROR; /* Service executed successfully */
}
```

Fig. 3: External Service call

The name-lookup resolution technique is used in the droplet-based application because method dispatching has to be done at run time for two reasons:

- it is not possible to statically link the droplet to the core application because the number and the type of the droplets are not known at compile time
- the indirect call technique decouples the droplets and makes them independent of the existence of other droplet-provided services.

Thus the indirect dispatch is of primary importance because it makes the application independent of the location and the existence of services.

Implementing the droplets using shared libraries does not have much effect on the application speed of a monolithic application. In fact in the latter case the compiler and the linker resolve the references. In a droplet-based application the resolution is performed at run time by the indirect dispatch mechanism. Every time a droplet or a service is invoked, the core application has to find the corresponding function (this search may fail if the function does not exist). This lookup has an impact on the performance but the slowdown effect can be minimized if caching or an efficient search mechanism such as hash tables is used¹¹. Note that when a virtual class or method is used in an OOPL (OO Programming Language) a similar lookup is performed; this analogy illustrates how the droplet technology is similar to the OO technology and how in some ways both techniques share similar problems.

3.2 Understanding the Droplet Interface

The object interface is the set of methods to which the object responds. The method name and its signature are defined at compilation time and

¹¹ The author has not investigated in detail how to efficiently solve the problem. The Self language uses some very interesting techniques [Ungar 87] that can be employed to significantly decrease the lookup time.

cannot be changed at run time. Hence the only way to add new methods or to modify existing ones is to shut down the application, modify it, recompile it and start it once again.

The droplet interface defines the boundary between the core application and the droplets. Unlike the object interface, the droplet interface is well defined and it is the same for every droplet independently to the provided services. It also has to be general enough to accommodate heterogeneous droplets without the need to modify it in order to accommodate different and heterogeneous services. The difference between an object interface and a droplet interface can be emphasized by means of an example.

Suppose we have a droplet with this interface:

```
class Class {
    public:
        int PerformAction(char *serviceName, char **returnValue);
}
```

Fig. 4: Droplet Interface

Imagine that the droplet recognizes only the "Apple" and "Pear" services, and that we have to extend it by adding a service called "Orange". The way to do this is to add the functions/methods needed to handle it. When the `serviceName` parameter has the value "Orange", this new logic is used. Thus a droplet does not redefine its interface but releases some constraints on the parameters by accepting a wider set of values. The interface being untouched, it prevents the application from being recompiled and but just the droplet itself whose new version can be reloaded by the running application.

Instead a class will be defined as follows:

```
class ParentClass {
    public:
        int Apple();
        int Pear();
};

class Class: public ParentClass {
    public:
        int Orange();
};
```

Fig. 5: Class Definition

Subclassing is necessary because the `ParentClass` class may have been shipped as a library and thus it cannot be modified but only subclassed. This prevents one from using the new class and the new methods from a class that is not derived from `ParentClass`. Also in order to make the new service available, the application has to be recompiled and restarted, whereas the droplet-based one has simply to reload the droplet. Nevertheless it is important to note that a droplet can be written either in a OOPL or in a non OOPL. Thus the difference between droplets and classes is that the droplet adds new services by releasing limitations on the values of the interface parameters (the interface being unchanged), whereas a class provides new

functionality by subclassing and defining new methods. This difference is shown in the following example:

```
int Class::dropletInterface(char *serviceName, char &returnValue)
{
    if(strcmp(serviceName, "Child") == 0)
        strcpy(&returnValue, GetChildName());
    else if(strcmp(serviceName, "Father") == 0)
        strcpy(&returnValue, GetFatherName());
    else
        return SERVICE_NOT_HANDLED; /* Service not handled by this droplet */

    return NO_ERROR; /* No error */
}
```

Fig. 6: Droplet: service handling

Nevertheless the droplet paradigm does not have to be considered class competitor. It has been shown that they can fit together and that, thanks to this, the droplet paradigm can be introduced into existing applications written in an OOPL like a transition path towards a pure compound application.

3.3 Droplet Interface Definition

A droplet is not statically linked to the core application but is loaded at run-time. Therefore a mechanism to load a droplet dynamically and to bind it to the core application has to be identified. Many operating systems provide facilities for the creation and use of dynamically bound shared libraries. Dynamic binding allows external symbols referenced in user code and defined in a shared library to be resolved by the loader at run time. The shared library code is not present in the executable image on disk. Shared code is loaded into memory once in the shared library segment and shared by all processes that reference it. Considering the facilities offered by a shared library and its great versatility, it makes sense to use it for droplet implementation.

A droplet is seen by the core application as a shared library. The core application can load droplets on demand or at start-up time. The entry point of a shared library can either be a function/method or a variable. Given the entry point the core application has to be able to access all the services provided by the droplet. It is therefore necessary to define a new type that contains all the information about the droplet and to define a static variable in the droplet code containing this information. Such a variable will be the entry point of the library.

This is the basic information that has to be specified by a droplet:

```
typedef void(*DropletInitTermFunct)(<function parameters>);
typedef void(*DropletInterface)(<interface parameters>);
```

```
typedef struct {
    short version;
    char *toolName, *toolInfo;
    short toolId;
    void* additionalInfos;
    DropletInitTermFunct startFunct, endFunct;
    DropletInterface toolFunct;
} DropletInfo;
```

Fig. 7: Droplet Information

The `DropletInterface` is defined as a function but it can be also defined as a (static) method. The parameters are not specified because they can vary from application to application. The `version` field is useful to check the version in order to prevent inconsistencies. The droplet interface or `DropletInfo` may have to be changed; therefore this will cause errors and possibly they may crash the entire application. The `toolName` and `toolInfo` are used to pass the droplet-related information to the core application. The `toolName` is used to find droplets and the `toolInfo` to characterize the droplet by specifying what the droplet does and how it expects to be invoked (the technique of putting comment information into the item itself has already been applied in languages such as Smalltalk). Suppose a droplet that intends to use a service XYZ is provided by another droplet, then the `toolName` can contain the service name (in this case XYZ) and the core application may exploit such information to localize the service. The `toolId` is a unique identifier that identifies the tool. It is used in case a service (e.g. XYZ) is mapped to a number and in this case the retrieval is based upon the `toolId` rather than on the `toolName`. The `additionalInfos` field contains optional information in an unspecified form. The core application is not responsible for this information but it is the droplet itself that has to manage it. The `startFunct`, `toolFunct` and `endFunct` are pointers to droplet functions. The `startFunct` and `endFunct` are optional (they can be NULL) and when present they are called when the droplet is loaded/unloaded. The `toolFunct` identifies the core droplet function. It is invoked by the core application when an operation concerning the droplet has been requested.

The ability of the core application to load the droplet at run-time makes it possible to load droplets on demand. This technique can be used to save system resources such as memory and to make distributed applications more flexible. For example a remote application can provide a service, implemented by a droplet, used frequently by a local application. In order to reduce the network traffic, it makes sense to copy the droplet from the remote host to the local host and to attach it to the local application. This way every time this service is requested, a local request is issued instead of a remote request. Similar techniques can be used in many other cases where the performance, the network traffic or the resources in use have to be optimized.

3.4 The Service Interface

Droplets do not have to be seen as mere parts of an application. They are separate entities that provide and use services. In order to specify which

services are provided by a droplet, the droplet interface needs to be extended. A mechanism that allows a droplet to issue service requests and receive responses must be defined too.

In a monolithic application the names of classes and methods are known at compilation time. In a droplet-based application, each droplet knows about itself and the basic services provided by the core application. A droplet cannot make any hypothesis about the presence of a certain service or about its location.

One way to address this problem is to add a service interface to the droplet interface. The droplet specifies the services that it intends to make available to peers, and the core application uses such information to locate and invoke the services on behalf of the droplets. This interface must also mask how the service is implemented and other details that should not be of public interest, just like a class of an OOPL does. The interface can be specified as follows:

```
typedef int(*ServiceFunction)(void* in_data, void** out_data);

typedef struct {
    ServiceFunction servPtr;
    char *servName,      /* Name of the service      */
        *servInfo,      /* Info about the service  */
        *servInParam,   /* Info about input parameter */
        *servOutParam,  /* Info about output parameter */
        *servRetValue;  /* Info about return value  */
} Service;

typedef struct {
    ...
    Service* availableServices;
} DropletInfo;
```

Fig. 8: Service Interface

The `servPtr` is the pointer to the function that provides the service. It is very important that this function has a well-defined and general interface like the one shown above. There are other possibilities to define `ServiceFunction` such as imposing no constraints concerning the number and type of the parameters. This requires the use of a variable argument list that may reduce robustness due to the limited checks that can be performed on the parameters. The `servName` identifies the name of the service. To improve the lookup speed, a numerical index can be added as well. The other fields are used to provide additional information about the service: what it does, which input/output parameters it expects, and how to interpret the return value. These fields are very important whenever a droplet has to call services provided by other droplets and whenever it has to find out how it is supposed to issue requests. They do not have to be identified as information of secondary importance but they have to be considered part of the dynamic service call interface. For example suppose that a graphic application displays a pop-up menu containing the names of all the external tools such as circle, line or rectangle. The application can exploit `servName` to add the tool names to the menu and `servInfo` to draw an icon for each entry. Note that certain services may be available only when

specific conditions are verified (e.g. only when some resources are available); for this reason the core application offers a way to dynamically register/deregister services other than via `DropletInfo`.

The services are identified with a precise name just like for methods. Nevertheless it is important to include a description of the service. It has to be remembered that the droplets may have been written by different persons for very different purposes and therefore may not have a description, so these droplets are useless for anybody but the droplet-writer.

3.5 Handling External Events

The droplet interface allows the core application to invoke droplets prior to a certain request. A droplet is not able to invoke external droplet-provided services or functions asynchronously even if the `DropletInterface` and `DropletFunc` may invoke functions external to the droplet itself. In order to overcome this problem, the interface must be extended once more. Every event that the droplet intends to handle (e.g. time event, external event, key down/up event etc.) has to be mapped to a function. Therefore the interface has to contain an additional field:

```
typedef struct {
    short eventId;
    DropletFunc eventFunction;
} EventEntry;

typedef struct {
    ...
    EventEntry *handledEvents;
} DropletInfo;
```

Fig. 9: External Event Handler Definition

The `handledEvents` is a NULL-terminated list of events (it can be empty) that are handled by the droplet. An entry could be:

```
EventEntry dropletEntry[] = {
    { EXTERNAL_EVENT, compExtEventFunc },
    { INCOMING_DATA, incomingDataFunc },
    NULL };;
```

Fig. 10: An example of droplet handled events

In this case the droplet tells to the core application to invoke the `compExtEventFunc` prior to an external event or to invoke `incomingDataFunc` when incoming data is available.

4. Applying Droplet Concepts

IBM has recently introduced a product for managing telecommunication networks (TMN). A key component of this product is the OSI agent technology developed in the Zurich Research Laboratory and currently used by many customers world wide.

In the OSI network management world, an agent is an application able to maintain information about the state of a part of the network that it is responsible for. A manager sends protocol requests to the agent using a protocol named CMIP. The agent process such requests and send replies back to the manager.

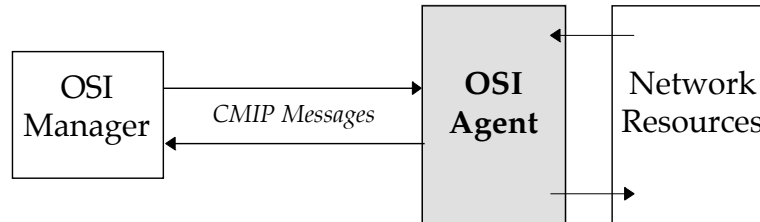


Fig. 11: OSI Agent/Manager architecture

Information to be managed by an agent is modelled as managed objects. A managed object (MO) represents either a logical resource such as a user account, or a real resource such as a router. A MO contains attributes, i.e. actions that can be performed on the object and notifications that can be issued by the object prior to a certain event.

The agent technology developed at the Zurich laboratory is based on a tool that takes as input the description of the MOs and of their relative attributes¹² and that generates a C++ class for each attribute and MO part of the agent. The generated code is then linked with the core agent functions in order to generate the final application.

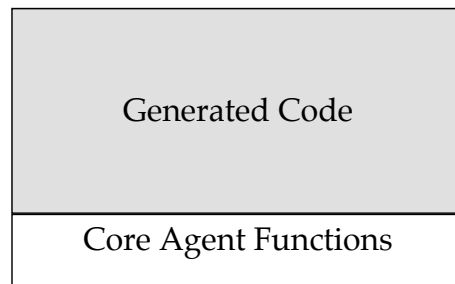


Fig. 12: Internal structure of the Agent built using the Zurich technology (ZSME)

The scope of the agent has been extended by applying the droplet technology to the initial object-oriented design. Each generated class, both MO and attribute, is no longer compiled and statically linked with the core agent function but is contained in a droplet that will be loaded by the agent at runtime. The goal is allow to add/replace specific MOs or attributes without having to rebuild the entire agent.

¹²The GDMO (Guidelines for the Definition of Managed Objects) is the notation used to define MOs. The type of attributes within a MO is defined using the ASN.1 (Abstract Syntax Notation One). These notations are defined in the international standards ISO/IEC 10165-4 and ISO/IEC 8824 respectively.

A new element, called *droplet manager* (DM), has been added to the core agent. This service is responsible for load/unload droplets - stored in a subdirectory of the agent directory - and for their management.

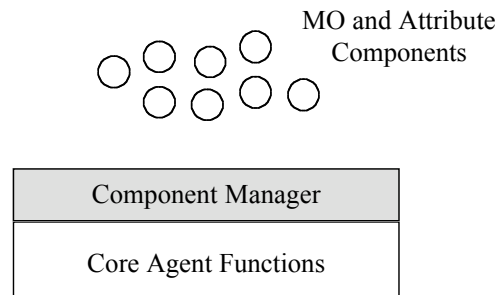


Fig. 13: Architecture of the droplet based ZSME Agent

At start-up time, the DM loads all the droplets and then enables the MOs that can actually be instantiated. A MO can be enabled only if it has access to all the necessary resources.

The DM is also used to decouple the implementation of a MO from the implementation of its attributes. Each MO class is derived from the `MCinstance` abstract class, and each attribute from the `attribute` abstract class.

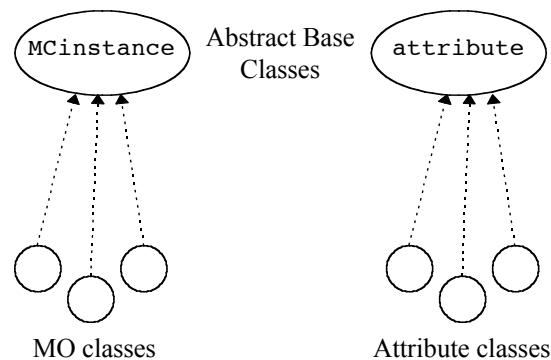


Fig. 14: Agent abstract and concrete classes

In the generated code of each MO, all the references to the attributes - class names and include files - have been replaced with a service provided by the DM. This is shown in the following example where a reference to the `ATTR_systemId` attribute present in the MO `system` has been replaced with the `CreateAttribute` method call.

```
#include "ATTR_systemId.h"
[....]
new_attr = new ATTR_systemId;
```

Fig. 15: Creation of the `ATTR_systemId` attribute from within the MO `system`

```
DropletManager *compMgr;
#define ATTR_systemId_OI "2.9.3.2.7.4"
new_attr = compMgr->CreateAttribute(ATTR_systemId_OI13);
```

Fig. 16: Use of the DM for the creation of the ATTR_systemId attribute

The CreateAttribute method, given its unique object identifier, has to locate the droplet that implements the requested attribute and then call the toolFunct function (Fig. 7).

```
attribute* DropletManager::CreateAttribute(char* attrOID)
{
    DropletInfo *theEntry = RetrieveDropletbyOID(attrOID);

    if(theEntry)
        return(theEntry->toolFunct());
    else
        return NULL; /* Attribute NOT found */
}
```

Fig. 17: DropletManager::CreateAttribute implementation

The droplet interface has been defined as:

```
typedef attribute* (*DropletInterface)();
```

Fig. 18: Droplet interface definition

and then toolFunct in the case of the MO system is:

```
typedef attribute* (*DropletInterface)();
attribute* toolFunct() { return((attribute*)new ATTR_systemId); }
```

Fig. 19: Droplet interface definition and implementation (ATTR_systemId)

In this way the core agent is not aware that the attribute returned by toolFunct is actually an instance of the ATTR_systemId class; it instead considers the attribute as it would be an instance of the attribute class. In case the core agent calls a virtual method x on this instance, the method ATTR_systemId::x is called (if it exists) even if the core agent is not aware of the existence of that method. This is because the virtual table associated with the instance returned by toolFunct is actually a pointer to the ATTR_systemId virtual table and then the methods are dispatched correctly (ATTR_systemId::x is called instead of attribute::x).

This technique just shown for the attributes is used for the MO too. The core agent is then able to manage MO and attribute instances without having access to their definition and implementation. The DM is responsible for creating instances of the actual classes (e.g. CLASS_system and ATTR_systemId) on behalf of the core application. The C++ virtual method dispatch mechanism then does the rest of the work by calling the correct methods and not just the ones known by the core application.

At runtime it is possible to replace a droplet with a new version of it. It is then possible to change the behavior of the application dynamically while

¹³In GDMO each entity such as MOs and attributes are identified with a unique object identifier that is usually represented by a set of digits separated by dots.

the application is running. In many cases it is not acceptable to shut the agent down and then restart the new version because other applications may have to be constantly connected to the agent. The DM then has to unload the old droplet and replace it with the new version. In order to allow this, the new droplet is free to change the method implementation but not to add/remove class attributes. In other words the implementation of the class can change but not its definition. This is because the image in the memory of the class instances must remain unchanged, otherwise the new class implementation will not be able to manage both old and new instances transparently.

An additional aspect of the dynamic droplet reload has to be considered. In some operating systems, the reload of a shared library may cause a new memory area to be used instead of reusing, and probably extending, the old memory area. If the memory area is reused, then the virtual table for a certain class remains at the same location. Therefore instances created with the old droplet version will automatically call the new methods. If the new shared library is loaded at a different location, the virtual table of instances created with the old droplet version will point to a location not longer in use and then the application will crash. A solution to this problem is to remove all the virtual methods (then the virtual table is no longer created) and implement them using software techniques instead of relying on the virtual table mechanism provided by C++.

Thanks to the use of droplets, parts of the agent can now be modified or even new functionality added without recompilation and relinking. This is value-added functionality in cases where the agent must remain operational or where new resources, not handled by the initial agent, have to be managed. The integration of droplets with the agent did not require the initial design to be modified significantly, and it has exploited and enforced the initial object-oriented design. The code has been slightly changed in order to use the DM to create the instances. This modification is very limited and it has affected not more than 5-10% of the generated code. The performance of the droplet-based agent with respect to the old version is almost the same. We have noticed only a minimal slow down when an instance has to be created because of the method `DropletManager::CreateAttribute` has been used. This is because the method has to locate the requested attribute and then create the new instance. The performance degradation is very limited (in our tests it is less than 3%) and it affects only the creation of the instances. Once the instance has been created, the usual method to dispatch via a virtual table is used and then the performance is not affected. Implementing the droplets with shared libraries allowed the memory used by the agent at runtime to be significantly reduced (up to 60%). This memory saving is very useful on small machines or when multiple agents run on the same host because the memory used by shared libraries is shared among the agents.

In spite of these advantages, droplets have two significant drawbacks related to their dynamic nature that are not present in monolithic applications because it is assumed that the code cannot change unless the

application is recompiled. In order to reload a droplet while an application is running, a droplet must:

- not store any data because the data will be lost when the droplet is updated;
- not change the class definition (i.e. adding new attributes) of the classes contained in a droplet, if any, but just the implementation.

The first drawback can be overcome by defining a storage service in the core application: data is always stored in the core application and the droplet contains just pure code. From another perspective, the separation between data and code is the mechanism that allows droplets to be replaced while the application is running without losing data.

The second drawback is due to the fact that a new class implementation has to deal with instances created with the previous implementation. Because of this, the instance shape in memory must not change otherwise the new implementation will not be able to handle the old instances or it will try to access attributes that are not at the expected position because their definition has changed. This behaviour is very dangerous because it may cause unexpected application termination.

The droplet-based agent offers several benefits with respect to the initial version:

- it is now possible to manage new resources by adding droplets at runtime or changing the agent behavior by modifying existing droplets and reloading them;
- memory usage is now more efficient and the compilation time has been significantly reduced since a modification of a class does not require the entire agent to be rebuilt, but just a droplet;
- link time is reduced because the core agent never changes and therefore it has to be built only once;
- tailoring has been simplified since it can be done simply by modifying the set of droplets that have to be loaded by the core agent.

5. Conclusions

This paper has shown that the major advantages of the droplets are:

- Coexistence with the OO paradigm: this is very important because the existing applications can benefit from the use of the droplets without having to be rewritten.
- Cleaner code design: the boundaries defined by the droplet interface help the developer specify the information and the services. Often OO programmers merge services and information in a (virtual base) class without trying to define it properly.
- Simple tailoring: an application has to provide only the services that are actually used. This can be done by providing only the droplets that implement such services and the core application.
- Low cost and resource usage: the customer buys only the droplets that he actually uses without having to pay for something that he will never run and that may complicate the configuration and use of the

application. Memory and disk space are saved because there is no need to load/store all the droplets. General performance is improved because only the necessary resources are used.

- Easy extensibility: due to the droplet interface and to the services provided by the core application or by other droplets, it is not difficult to add new functionality by creating new droplets.
- Path to distributed programming: because the services are bound by the droplets and have a well-defined interface, it is possible for remote applications to use these services. It is also possible, if a service is heavily used, to copy the droplet that provides the service from the remote application and attach it to the local application, thus speeding up the processing significantly.

It has been shown that OO programming and droplets are two complementary and yet compatible technologies. The droplet is a promising technology, different flavours of which have already been applied in part in some specific applications. It has several advantages and a few drawbacks due to its dynamic nature. For these characteristics, it can be considered an interesting and novel technique for breaking monolithic applications apart.

6. References

- [Apple 95] Apple Computer
Components Made Easy
OpenDoc Technical White Paper, March 1995.
- [Ban 95] B. Ban and L. Deri
Object Factory Revised: a Design Pattern
IBM Zurich Research Laboratory, September 1995
- [Booch 91] G. Booch
Object-Oriented Design
Benjamin-Cummings, Redwood City, CA, 1991.
- [Bourne 83] S. R. Bourne
The UNIX System
Addison-Wesley, Reading, MA, 1983.
- [Chen 93] D. J. Chen and S. K. Huang
Interface for Reusable Software Components
Journal of OO Programming Lang., Jan. 1993.
- [Chen 94] D. J. Chen and D. T.K. Chen
An Experimental Study of Using Reusable Software Design Frameworks to Achieve Software Reuse
Journal of OO Programming Lang., May 1994.
- [Cox 91] B. Cox and A. Novobilski
Object Oriented Programming: An Evolutionary Approach
Addison-Wesley, Reading, MA, 1991.
- [Ellis 91] M. Ellis and B. Stroustrup
The Annotated C++ Reference Manual
Addison-Wesley, Reading, MA, 1990.
- [Gamma 94] E. Gamma, R. Helm, R. Johnson and J. Vlissides
Design Patterns: Elements of Reusable OO Software.
Addison-Wesley, Reading, MA, 1994.
- [Goldberg 83] A. Goldberg and D. Robson
Smalltalk-80: The Language and its Implementation
Addison-Wesley, Reading, MA, 1983.
- [IBM 94] SOM Dev. T.lkit: *An Introductory Guide to the System Object Model and its Accompanying Frameworks*
IBM, October 1994.
- [Kernighan 88] B.W. Kernighan and D. M. Ritchie
The C Programming Language
Prentice Hall, Englewood Cliffs, NY, 1988.
- [Mathews 90] D. C. Mathews
Static and Dynamic Type Checking
Bancilhon & Buneman, 1990.
- [Meyer 88] B. Meyer
Object Oriented Software Construction
Prentice Hall, Englewood Cliffs, NY, 1988.
- [Micallef 88] J. Micallef
Encapsulation, Reusability and Extensibility in Object Oriented Programming Languages
Journal of OO Programming Lang., Apr./May 1988.

- [Microsoft 93] *Object Linking and Embedding v. 2 (OLE2): Programmer's Ref.*
Vols. 1 and 2, Microsoft Press, Redmond, WA, 1993.
- [Nackman 94] L. R. Nackman and J. J. Barton
Base-Class Composition with Multiple Derivation and Virtual Bases
IBM T.J. Watson Research Center, 1994
- [Nierstrasz 92] O. Nierstrasz, S. Gibbs and D. Tsichritzis
Component-Oriented Software Development
Communications of the ACM, Vol. 35, No. 9, Sept. 1992.
- [OMG 93] *The Common Object Request Broker: Architecture and Specification*
1.2 edition, Object Management Group, 1993.
- [Pinson 91] L. J. Pinson and R. S. Wiener
Objective-C: Object Oriented Programming Techniques
Addison-Wesley, Reading, MA, 1991.
- [Udell 94] J. Udell
ComponentWare
Byte, May 1994.
- [Ungar 87] D. Ungar and R. Smith
Self: The Power of Simplicity
OOPSLA '87 Conference Proceedings, October, 1987.