YILDIZ TEKNİK ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

BİR OBJECT REQUEST BROKER UYGULAMASI

Müh. Ceyhun ÖZGÜN

F.B.E Matematik Mühendisliği Anabilim Dalında
Hazırlanan

YÜKSEK LİSANS TEZİ

Tez Danışmanı : Yrd. Doç. Dr. İbrahim EMİROĞLU

Prof.Dr. Mehmet Akkaştırıoğlu

Y. Doç. Dr. Coşkun Güler

İSTANBUL, 2000

İÇİNDEKİLER

KISALTMA LİSTESİ

BOA	Basic Object Adapter
CORBA	Common Object Request Broker Architecture
DCOM	Distributed Component Object Model
DÇA	Dinamik Çağırım Arayüzü
DII	Dynamic Invocation Interface
DSI	Dynamic Skeleton Interface
DTA	Dinamik Taslak Arayüzü
DUR	Dinamik Uygulama Rutini
GIOP	General Inter-ORB Protocol
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
NA	Nesne Adaptörü
OMG	Object Management Group
ORB	Object Request Broker
POA	Portable Object Adapter
RMI	Remote Method Invocation
RPC	Remote Procedure Call
TCP/IP	Transmission Control Protocol / Internet Protocol
TNA	Temel Nesne Adaptörü

ŞEKİL LİSTESİ

ÇİZELGE LİSTESİ

ÇİZELGE LİSTESİ

ÖZET

Günümüzde, birbirlerinden farklı lokasyonlarda birimleri bulunan kuruluşların otomasyon projelerinin uygulamalarında en çok tercih edilen mimari Common Object Request Broker Architecture (CORBA) Mimarisi'dir. Çalışmanın amacı; CORBA Mimarisi'nin oluşumunu ve gelişimini sağlayan etkenleri belirleyerek CORBA'nın esnek ve güvenilir yapısının özelliklerini ortaya koymaktır.

Teknolojinin hızla büyüyen ve gelişen yapısı yazılım sektöründe de büyük değişikliklere neden olmuştur. Yazılım Geliştirme bir mühendislik dalı haline gelmiş ve yazılım projeleri uzman gruplar tarafından uygulanmaya başlanmıştır. Bunun en önemli sonucu da bu projelerde zamanın daha verimli kullanılması, projelerin daha sistemli bir şekilde yerine getirilerek çalışmalardan maksimum verim alınmasıdır.

Bütün bu gelişmeler beraberinde yazılım sektöründe sürekli gelişen ve değişen yeni mimarilerin ve teknolojilerin ortaya çıkmasını sağlamıştır.

Çalışmada ilk olarak, günümüze kadar gelen yazılım mimarileri incelenmiş ve en son adım olan Dağıtık Sistem Modeli ele alınmıştır.

Dağıtık Sistemler; uygulamayı oluşturan ve her biri kendi görevinden sorumlu olan nesnelerin, birbirleri ile olan ilişkilerini nesne arayüzleri aracılığıyla sağlayan bir yazılım mimarisidir.

CORBA; nesne arayüzlerinin tanımlanması için bir standart yöntem sunan ve nesneler arasındaki iletişimi Object Request Broker (ORB)'lar ile sağlayan bir dağıtık sistem mimarisidir. CORBA'nın temel bileşeni olan ORB'ler, CORBA'nın platformdan bağımsızlık özelliğini yerine getirir. CORBA'nın diğer temel bileşeni olan Interface Definition Language (IDL) nesne arayüzlerinin tanımlanması için kullanılan standart bir tanımlama dilidir.

Çalışmanın bir sonraki adımında CORBA Mimarisi'nin gelişimi, alternatifleri ve özellikleri ele alınmıştır.

CORBA Mimarisi hakkında en önemli kaynak, CORBA'yı uluslararası bir standart haline getiren ve uzmanlar, bilim adamları ve yazılımcılar tarafından oluşturulan bir konsorsiyum olan Object Management Group (OMG)'tur. Object Management Group'un kuruluşundan bugüne CORBA hakkında yazılan raporlar, dokümanlar ve değişik uygulama örnekleri incelenmiş, Object Request Broker'lar hakkında önemli çalışmaları olan OMG dışı uzmanların çalışmaları da dikkate alınmıştır. Ayrıca profesyonel ve amatör yazılımcıların üyesi bulunduğu AT&T Laboratories Cambridge ve Oracle Research Library tarafından oluşturulan mail-list'e üye olunmuştur.

Bu çalışmalar sonucunda gözlenen en önemli nokta; CORBA'nın programlama dili ve platform bağımsızlığı sağlaması, böylece programcılara nesneler arasındaki iletişimi kurma aşamasında meydana gelebilecek sorunlarla uğraşmak yerine, programın fonksiyonalitesine yoğunlaşma fırsatı vermesidir. CORBA'nın programlama dilinden bağımsızlığı, arayüzlerin, programlama dilinden bağımsız bir standart dil olan Interface Definition Language kullanılarak tanımlanması sayesinde sağlanır.

CORBA Mimarisi'nin diğer bir özelliği, nesneye dayalı özelliğini, C ve COBOL gibi nesneye dayalı olmayan programlama dilleri ile kullanıldığında bile sağlıyor olmasıdır. Bu sayede CORBA, alternatiflerinin aksine yeniden kullanılabilme, genişletilebilme, esneklik, güvenilirlik ve taşınabilirlik gibi yazılım geliştirme alanında önemli olan avantajları ile tercih sebebi olmaktadır.

Çalışmanın son bölümünde bir Object Request Broker uygulaması örneği bulunmaktadır. Bu uygulamada özel bir kurumun, özel bir ihtiyacını karşılamak amacıyla CORBA standardına uygun ve CORBA'nın esneklik özelliği kullanılarak hazırlanan yeni bir Object Request Broker tasarlanmış ve bu Object Request Broker'ı kullanan bir Banka Yönetim Programı geliştirilmiştir.

ABSTRACT

Nowadays, the mostly preferred architecture in the automation projects of the companies which have distributed units is CORBA. The aim of this study is to state the flexible and realiable characteristics of CORBA by determining the factors which enables CORBA's formation and improvement.

The rapid growing nature of the technology caused some important changes in the software sector. Software Development has become an engineering discipline and software projects have been started to be managed by expert groups. The most important out come of this is efficient usage of the time and the achievement of maximum effectiveness by accomplishing the projects more systematically.

With all these improvements brought together, the emergence of growing and changing software technologies and architectures continues.

In this study, the software architectures till today have been examined, and the Distributed System Model which is the last step has been inspected in detail.

Distributed Systems Model is a software architecture which regulates the commnunication of the objects that are the components of the application. Components are responsible for their own tasks, accessed through their interfaces.

CORBA is a Distributed System Model Specification which presents a standard method for defining the object interfaces and communication among objects via Object Request Brokers.

The Object Request Broker, which is the main component of a CORBA application, makes CORBA a platform independent architecture. The Interface Definition Language, which is an important specification of CORBA, is a standard definition language which is used for defining the object interfaces, independent of the implementation language.

In the next step of the study, the emergence of CORBA architecture and its alternatives and characteristics have been examined.

The most significant resource for the CORBA Specification, is Object Management Group (OMG), which is formed by experts, scientists and software engineers. OMG has played a big role in making CORBA an international standard. In this study the reports, documents and application samples about CORBA which have been prepared since the foundation of OMG have been inspected. Moreover, the studies of the experts out of OMG have also been taken into account. Besides, I subscribed to the mailing-list of the AT&T Laboratories-Cambridge and Oracle Research Library, which has both professional and amateur software developers members.

The most striking observation as a result of this work is the fact that CORBA provides platform and programming language independency and therefore lets the programmers focus on the functionality of their programs instead of dealing with the obstacles that may arise in the stage of communication of objects. CORBA is independent of programming languages, since interfaces are defined using IDL which is a standard language independent of programming languages.

Another important advantage of CORBA architecture is that it provides object-oriented features even when it is used with languages like C and COBOL which are not object-oriented. Therefore, CORBA is preferred over its alternatives for its advantages like reusability, extensibility, flexibility, reliability and portability.

In the last part of this study, there is an sample Object Request Broker application. In this application, an ORB, complying with the standards of CORBA, has been designed, making use of flexibility of CORBA. A banking application which uses that ORB, has also been developed.

1. GİRİŞ

Common Object Request Broker Architecture (CORBA), günümüzde hızla gelişmekte olan yazılım teknolojilerin sonucunda ortaya çıkan yazılım ürünleri arasındaki beraber çalışabilirlik ihtiyacını karşılayan Object Management Group'un sunduğu bir çözümdür. Basit olarak CORBA, uygulamaların nasıl yaratıldıklarını ve nerede olduklarını bilmeden birbirleriyle iletişim kurmalarına izin veren bir yöntem tanımlayan bir standarttır.

CORBA, bir Dağıtık Sistem Mimarisi'dir. Bir Dağıtık Sistem, belirli bir amaca yönelik hareket eden bir sistemin bileşenlerinin, birbirinden farklı lokasyonlarda birlikte çalışabilmelerine olanak tanır.

Object Request Broker (ORB), CORBA standardının temel bileşeni olan ve genel olarak bileşenler arasındaki iletişimi sağlayan bir ara katman yazılımıdır. Bir bileşen, bir ORB'yi kullanarak diğer bileşenin lokasyonunu bilmesine gerek kalmadan diğer bileşen ile iletişim kurabilir. ORB, ulaşılmak istenenen bileşenin lokasyonunu bulup ilgili bileşenin iletişim kurabilmesine olanak tanır.

1.1 Dağıtık Sistemlerin Tarihçesi

1.1.1 Monolitik Sistemler ve Mainframe'ler

Mainframe'ler kullanılan ilk bilgisayarlardır. Mainframe'ler aptal terminaller olarak da bilinen yeşil ekranlı terminaller ile kullanılır. Terminaller sadece kullanıcı iletişimini sağlamakta, her şey Mainframe'lerin üstünde yönetilmektedir. Bu yüzden yönetim kolaylığı sağlanabilmektedir.

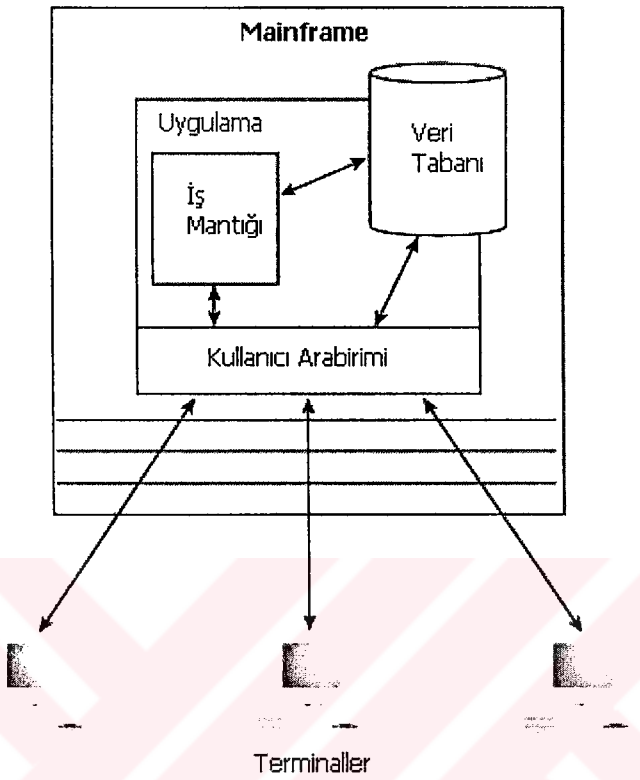
Mainframe'ler kurulum ve bakım maliyetleri çok büyük olmasına rağmen çok sayıda kullanıcıya hizmet verebilen güçlü makinalardır. Genellikle büyük yığın işler (batch jobs) ve hayati bilgilerin saklanmasında tercih edilirler*. Bu yüzden en çok banka endüstrisi, kamu hizmeti veren kurumlar ve bilgi sistemleri işletmelerinde kullanılırlar.

Mainframe'lerde kullanılmak için yazılmış programlar genellikle monolitik yapıdadırlar. Yani, kullanıcı arabirimi, iş mantığı ve veri ulaşımı fonksiyonalitesi tamamiyle Mainframe'de bulunan büyük bir program içinde bulunur. Terminaller sadece Mainframe'lere ulaşmak

---

* Software Technology Review Online Guide. Software Engineering Institute. Carnegie Mellon University. http://www.sei.cmu.edu/activities/str/descriptions/mssa.html

amacıyla kullanıldığı için terminallerde hiç bir iş yapılmamaktadır. Program Mainframe'in kendisinde çalışmaktadır. Bu nedenle programın kurulumu, bakımı kolay olmakta ve monolitik mimari yararlı ve kullanışlı olabilmektedir. Şekil 1.1'de tipik bir monolitik uygulama mimarisi gösterilmektedir.



Şekil 1.1 Monolitik uygulama mimarisi.

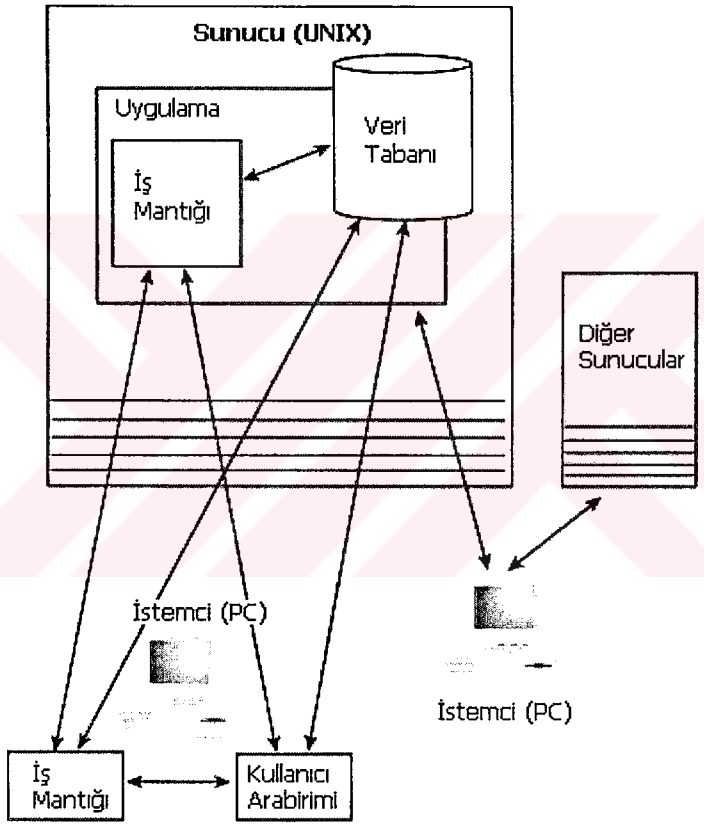
1.1.2 İstemci-Sunucu Mimarisi

Kişisel bilgisayarların yaygınlaşması ile gelişen istemci-sunucu mimarisi (Client/Server Architecture), Mainframe'lerdeki yükün masa üstü kişisel bilgisayarlara kaydırılabilmesine olanak sağladı ve UNIX tabanlı sunucuların ortaya çıkmasını sağladı. Mainframe'lerin yüksek gücüne ihtiyaç duymayan uygulamaların işlerini bu küçük UNIX tabanlı sunucu makinalarada halledebilmesi ve bu sistemin maliyetinin Mainframe'lerden çok daha az olması bu sistemin yaygınlaşmasını sağladı. İş ölçekleri büyük olmayan işletmeler için bu sistemler daha ekonomik ve kullanışlı bir hale geldi. Ayrıca büyük bir işletmenin farklı departmanlarında farklı sunucular kullanarak sadece o departmana ait veri ve uygulamaları kolaylıkla yönetme şansı doğdu. Farklı departmanlar sadece kendi özel ihtiyaçlarını karşılayan, daha kolay yönetilebilir uygulamalara sahip olabildiler. Ayrıca terminaller sadece Mainframe'lerdeki uygulamaları çalıştırabilirken, PC'ler Mainframe'lerden bağımsız olarak değişik işleri yerine

getirebilme şansına sahipti. Bu da bir masa üstü makine olan PC'nin kullanılışlılığını artırdı.

İstemci-sunucu mimarisi, uygulama bileşenlerinin sunucu ve istemciye dağıtılabilmesine olanak tanır. Kullanıcı arabirimi istemcide, veri tabanı sunucuda ve iş mantığı istemci yada sunucuda olmak üzere bir uygulamanın bileşenleri dağıtılabilir. İstemci tarafındaki bileşenlerin herhangi birinde bir değişiklik meydana geldiğinde, bu bileşenin yeni kopyalarının her bir kullanıcıya yeniden dağıtılması gerekir.

Çok katmanlı istemci-sunucu mimarisinin ortaya çıkmasıyla bu "orjinal" istemci-sunucu mimarisi "iki katmanlı" istemci-sunucu mimarisi (two-tier client/server architecture) olarak adlandırıldı. Şekil 1.2' de iki katmanlı istemci-sunucu mimarisi gösterilmektedir.



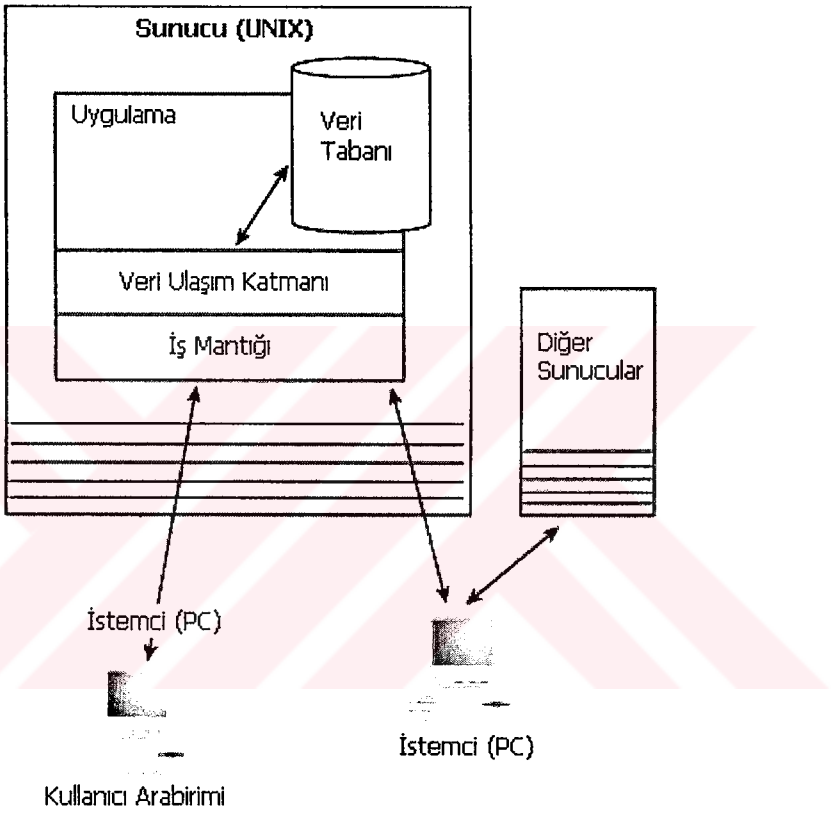
Şekil 1.2 İki katmanlı istemci sunucu mimarisi.

1.1.3 Çok Katmanlı İstemci-Sunucu Mimarisi

İstemci-sunucu mimarisi eski yöntemlere göre bir devrimdi. Mainframe tabanlı uygulamaların sorunlarını çözmesine rağmen istemci sunucu mimarisinin de eksik yönleri vardı (Rosenberger, 1998). Örneğin, genellikle veritabanı ulaşımı fonsiyonalitesi ve iş mantığı istemci bileşende bulunduğu için, iş mantığı, veri tabanı ulaşımı yada veritabanının

kendisinde bir değişiklik gerektiğinde tüm kullanıcılardaki istemci bileşenin yenilenmesi gerekir. Genellikle bu değişiklikler istemci bileşenlerin eski versiyonları ile uyuşmazlığa yol açarlar ve sonuçta uygulama tutarsız (kırılgan) bir duruma gelir.

Orjinal istemci-sunucu mimarisin problemleri çok katmanlı istemci-sunucu mimarisi (Multi-Tier Client/Server Architecture) sayesinde çözülebilmiştir (Rosenberger, 1998). En popüler çok katmanlı mimari üç katmanlı mimaridir. Bu mimaride sistem, Kullanıcı arabirimi katmanı, iş kuralları katmanı ve veritabanı ulaşım katmanı olmak üzere üç mantıksal katmana bölünmüştür. Şekil 1.3' te üç katmanlı istemci-sunucu mimarisi gösterilmektedir.



Şekil 1.3 Üç katmanlı istemci sunucu mimarisi.

Çok katmanlı istemci-sunucu mimarisi iki katmanlı mimariyi iki yönden genişletir. Birincisi ve en önemli olanı; istemci bileşen uygulamanın geri kalanında meydana gelebilecek herhangi bir değişiklikten etkilenmez. Bu nedenle uygulama daha az uyumsuzluk problemi yaşayacaktır. Ayrıca, bileşenler daha çok parçaya bölündüğünden uygulamanın bakımı ve yönetimi daha da kolaylaşmaktadır.

Çok katmanlı istemci-sunucu mimarisi katmanlar arasında daha çok ayrım sağladığı için uygulamanın kırılganlığını azaltır. Kullanıcı arabirimi katmanı direkt olarak veritabanı ulaşımı katmanıyla iletişim kurmaz, sadece iş kuralları katmanı ile iletişim kurar. İş kuralları

katmanı ise bir taraftan kullanıcı arabirimi ile iletişim kurarken bir taraftan da veritabanı ulaşımı katmanı ile ilteişim kurar. Bu nedenle veritabanı ulaşımı katmanındaki herhangi bir değişiklik kullanıcı arabirimi katmanını etkilemez, çünkü bu katmanlar tamamen birbirlerinden ayrılmıştır. Dolayısıyla kullanıcı arayüzünde yada kullanıcı arayüzünün iş kuralları katmanı ile ilişki kurduğu bağlantı noktasında herhangi bir değişiklik olmadıkça, diğer katmanlardaki değişiklikler istemci bileşenlerin kullanıcılara yeniden dağıtılmasını gerektirmeyecektir. Bu nedenle versiyon farklılığından kaynaklanan kırılganlıklar en aza indirilmiş olacaktır.

Çok katmanlı istemci-sunucu mimarisi, uygulamayı geleneksel istemci-sunucu mimarisinden daha fazla mantıksal parçaya ayırdığı için uygulamanın dağıtımında daha çok esneklik sağlar. Örneğin Şekil 1.3'te ayrı mantıksal bileşenler olmalarına rağmen iş kuralları katmanı ve veri tabanı katmanı aynı sunucu makinada yer almaktadır. Bu sunucu bileşenlerin ayrı makinalarda yer alması mümkündür. Ayrıca uygulamada iş kuralları katmanı için birden fazla iş mantığı bileşeni oluşturmak, birden fazla veritabanı kullanılıyorsa farklı veritabanı ulaşımı bileşenleri oluşturmak ve bunları ayrı ayrı makinalara yerleştirmek mümkündür. Bu sayede daha güçlü ve daha kolay yönetilebilir uygulamalar yaratmak mümkündür.

1.1.4 Dağıtık Sistemler

Uygulama mimarilerindeki en son adım dağıtık sistemlerdir. Bu model nesne kavramını sunar. Bir nesne belirli bir amacı olan ve diğerlerinden ayrılabilen bir büyüklük olarak tanımlanabilir. Bir nesne diğer nesnelere hizmet sunabilir. Dağıtık Sistem Modeli, uygulamayı iş mantığı ve veri ulaşımı katmanları gibi katmanlara ayırmak yerine, uygulamanın tüm fonksiyonalitesini her biri kendi işinden sorumlu olan nesnelere paylaştırır ve bu nesneleri dış kullanıma sunar. Diğer bileşenler bu nesneleri kullanarak istedikleri hizmetlere ulaşabilirler. Bu nesneler de sistemdeki diğer nesneler tarafından kullanıma sunulan diğer hizmetleri kullanabilirler. Dağıtık Sistem mimarisi esneklik açısından en etkili mimaridir.

Dağıtık Sistem mimarisi esnekliğini bileşen arayüzleri tanımları sayesinde sağlar (Rosenberger, 1998). Bir bileşenin arayüzü diğer bileşenlere hangi hizmetlerin sunulduğunu ve nasıl kullanıldıklarını belirtir. Yani bileşenin arayüzü, bileşenin dışarıdan görülen kısmıdır ve bileşene ulaşmanın tek yolu bu arayüzü kullanmaktır. Bu arayüz değişmedikçe bileşenin içinde meydana gelen herhangi bir değişiklik diğer bileşenleri etkilemez. Örneğin, bir firmanın müşteri bilgilerine ulaşımı sağlayan bir bileşenin bilgileri belirli bir veritabanında

tuttuğunu düşünelim. Bu bileşenin bilgileri veri tabanında tutma metodu, kullandığı veritabanı, bileşenin arayüzü aynı kaldığı sürece değiştirilebilir.

Sonuç olarak Dağıtık Sistem mimarisi, nesnelerin birbirleriyle iletişim kurmalarını sağlayan arayüzler üzerine kurulmuştur.

1.2 Common Object Request Broker Architecture

CORBA, nesne arayüzlerininin tanımlanması için bir standart sunan ve nesneler arasındaki iletişimi Object Request Broker'lar ile sağlayan bir Dağıtık Sistem mimarisidir. CORBA bir teknoloji olmayıp nesne arayüzlerinin tanımlanması ve nesnelerin birlikte çalışabilmeleri için gerekli yöntemleri ve kavramları belirleyen bir standarttır. CORBA, nesnelerin arasındaki iletişim mekanizmasını ve nesnelerin lokasyonunu programcıdan soyutlayarak programcının nesneler arasındaki iletişimle uğraşmak yerine tamamiyle programın fonksiyonalitesine yoğunlaşabilmesine olanak sağlar (Vinoski, 1993). CORBA uygulamaları herhangi bir programlama dili ile yazılabilir ve herhangi bir platformda calışabilir. Yani CORBA uygulamaları programlama dili ve platformdan bağımsızdır (Chizmadia, 1999). Bir CORBA istemcisi, iletişim kurduğu sunucu hangi platformda çalışırsa çalışsın, hangi programlama dili ile yazılırsa yazılsın mutlaka düzgün bir şekilde sunucunun hizmetlerinden yararlanabilir.

1.3 CORBA'nın Alternatifleri

Bu bölümde CORBA mimarisine alternatif olarak gösterilebilecek bazı teknolojileri inceleyeceğiz.

Soket Programlama (Socket Programming)

Modern sistemlerde makinalar arası yada bazan aynı makina üzerindeki farklı uygulamalar arasındaki iletişim soketler kullanılarak yapılır. Basit olarak bir soket; iki uygulamanın birbirlerine bağlanıp iletişim kurabilecekleri bir kanaldır. Bir uçtaki uygulama yada uygulama bileşeni bu kanala bir bilgi gönderdiğinde, diğer uçtaki birim gönderilen bu bilgiyi okuyabilir. Bu işlem bir uygulamanın bilgiyi bir dosyaya yazması ve diğer uygulamanın da bu dosyaya yazılan bilgiyi okuması gibi düşünülebilir.

Sokete yazma işlemi, bilginin byte dizileri şeklinde sokete gönderilmesi işlemidir. Soketten okuma işlemi de bilginin byte dizileri şeklinde alınması işlemidir. Bu nedenle soket programlama tekniği oldukça düşük seviyeli (low-level) bir tekniktir. Bu sayede oldukça etkili ve hızlı programlar yazılabilir. Özellikle küçük ve belirli bir amaca yönelik (e-mail

istemcileri, web browserlar gibi.) programlar için kullanışlı bir tekniktir. Ancak hataya çok açık olduğu için büyük çaplı uygulamalarda kolay yönetilebilme, genişletilebilme ve bakım gibi konularda sorun çıkarabilmektedir (Schmidt vd., 1995). Ayrıca düşük seviyeli bir teknik olduğu için platformlar arası uyuşmazlıklar meydana gelebilmektedir.

Uzak Yordam Çağrımı (Remote Procedure Call – RPC)

Soket programlama üzerine kurulmuş bir teknik olan RPC, fonksiyon-tabanlı bir arayüz sunar. RPC sayesinde yazılımcı soket üzerinde akan veriyi direkt olarak yönetmek yerine, istemcinin çağırabileceği bir fonksiyon tanımlar. Bu fonksiyon herhangi bir programlama dilindeki sıradan bir fonksiyona benzer. İstemci RPC'yi kullanarak bu fonksiyonu çağırır. RPC uzak sunucu ile soketler yardımı ile iletişim kurup sunucu üzerinde belirtilen fonksiyonu çalıştırır ve yine soketler yardımıyla fonksiyonun sonucunu istemciye geri döndürür.

RPC fonksiyona dayalı bir arayüz sağladığından, genellikle soket programlamadan daha kullanışlıdır. Ancak RPC protokolü farklı platformlarda farklı şekillerde uygulandığı için platformlararası uygulamalarda yetersiz kalır.

Microsoft Distributed Component Object Model (DCOM)

DCOM, Microsoft'un dağıtık sistem teknolojisidir. DCOM, Windows 95 ve NT'ye entegre edilmiş bir nesne modelidir (Microsoft ve DCE, 1995). Farklı uygulama dilleri ile kullanılabilmesine karşın bir Microsoft teknolojisi olduğu için sadece Windows işletim sistemleri ile sınırlı kalmıştır. Bu nedenle platformlararası uygulamalarda kullanılamaz.

Java Remote Method Invocation (RMI)

RMI, güçlü bir dağıtık sistem teknolojisi olmasına rağmen sadece Java programlama diline özel bir teknolojidir. Yani RMI istemci ve sunucuları Java programlama dili kullanılarak yazılmış olmalıdır (Sun, 2000).

Yukarıda bahsedilen teknolojilerin dezavantajlarının yanında CORBA esneklik, güvenilirlik ve taşınabilirlik avantajlarına sahiptir (Gokhale ve Schmidt, 1996). Sonuç olarak CORBA mimarisi nesneye dayalı olması, programlama dili ve platformdan bağımsız olması nedeniyle en etkili dağıtık sistem mimarisidir.

1.4 CORBA'nın Tarihçesi

1.4.1 Object Management Group (OMG)

Object Management Group (OMG), içlerinde yazılım firmaları, bilgi sistemleri üreticileri gibi çeşitli üyeleri bulunan, şu anda 750'den fazla üyeye sahip uluslararası bir organizasyondur. 1989'da 8 üye ile kurulmuştur. OMG'nin amacı, yazılım geliştirme alanında nesneye dayalı teknolojileri tanıtmak ve yaymaktır (OMG, 1998). Uygulama geliştirme alanında genel bir çalışma ortamı sağlamak için nesne yönetim spesifikasyonları sunar. OMG herhangi bir yazılım üretmeyip sadece yeni teknolojiler belirleyip standartlarını sunar.

1.4.2 CORBA'nın gelişimi

CORBA 1.0, Ekim 1991'de yayınlandı. Bu versiyonu Şubat 1992'de CORBA 1.1 ve Aralık 1993'te CORBA 1.2 revizyonları izledi. Bu versiyonda Interface Definition Language (IDL), Dynamic Invocation Interfce (DII), Interface Repository ve uygulamaların bir Object Request Broker ile iletişim kurabilmelerine olanak sağlayan uygulama programlama arayüzleri (Application Programming Interface-API) tanımlanmıştır. Sadece C programlama dili dönüşümü tanımlandı.

CORBA 1.x dağıtık nesnelerin beraber çalışabilirliği alanında önemli bir adımdı ama eksiksiz değildi. ORB'lerin birbirleriyle iletişim kurmaları için standart bir protokol sunmamıştı. Sonuç olarak farklı üreticilerin ORB'leri birbirleriyle sağlıklı olarak iletişim kuramadılar ve bu ORB'leri kullanan programlar uyumlu olarak birbirleriyle çalışamadılar.

CORBA 2.0 Ağustos 1996' da tamamlanıp sunuldu. CORBA 2.0' ın en önemli başarısı farklı üreticilerin ORB'lerinin birbirleriyle iletişim kurup beraber çalışmalarına olanak sağlayan bir standart protokol sağlamasıydı. Bu protokol İnternet ORB'ler Arası Protokol (Internet Inter-ORB Protocol) olarak bilinir. ORB ürünlerinin CORBA 2.0 uyumlu olarak adlandırılabilmesi için bu protokolü kullanmaları zorunludur. Bu protokol sayesinde CORBA uygulamaları daha fazla üretici-bağımsız olabildiler.

CORBA standardı CORBA 2.0'dan sonra gelişmeye devam etti. Ağustos 1997'de 2.1 versiyonu, Şubat 1998'de de 2.2 versiyonu yayınlandı. 1998 yılı sonunda 2.3 versiyonu yayınlandı. OMG, şu anda CORBA 3.0 versiyonunu yayınlamak için çalışmaktadır.

1.5 CORBA Mimarisine Genel Bakış

CORBA, nesneye dayalı bir mimaridir. CORBA nesneleri diğer nesneye dayalı sistemlerin bir çok özelliklerini taşırlar. Örneğin arayüz miras alma (interface inheritance) ve çokşekillilik (polymorphism) gibi. CORBA'nın en önemli özelliği, bu nesneye dayalı özelliğini C, COBOL gibi nesneye dayalı olmayan programlama dilleri ile kullanıldığında bile sağlıyor olmasıdır (Rosenberger, 1998). CORBA'nın iki temel bileşeni ORB ve IDL'dir.

1.5.1 Object Request Broker

CORBA'nın temel bileşeni Object Request Broker'dır. Bir ORB, temel amacı nesneler arasındaki iletişimi sağlamak olan bir yazılım bileşenidir. ORB programcıdan nesneni lokasyonunu soyutlar. Önemli görevlerinden biri de verilen bir nesne referansı için uzak nesneyi bulmaktır. ORB'nin sağladığı bir diğer hizmet de parametrelerin uzak metoda gönderilmesi için hazırlanması (marshaling) ve uzak metodun sonuçlarının ve dönüş değerlerinin kullanılmak üzere hazırlanmasıdır (demarshaling). Ayrıca ORB dağıtık CORBA nesneleri için platformdan bağımsızlık özelliğini sağlar (OMG, 1998).

1.5.2 Arayüz Tanımlama Dili

CORBA mimarisinin bir diğer temel parçası da Arayüz Tanımlama Dili (Interface Definition Language-IDL)'dir. CORBA nesneleri arasındaki ilişkileri belirleyen arayüzleri tanımlamaya yarayan IDL, CORBA'nın programlama dili bağımsızlığını sağlayan temel bileşendir (Henning ve Vinoski, 1999). IDL herhangi bir programlama diline has bir yapıya sahip olmayan bir pseudo tanımlama dilidir. Arayüzler, IDL ile tanımlandıktan sonra dönüşüm (mapping) yoluyla bileşeni yazmak için kullanılacak programlama dilindeki eşleniklerine çevrilir. Bu dönüşüm IDL Derleyicileri (IDL Compilers) denilen programlar ile yapılır. Arayüzler, bu derleyiciler ile istenen herhangi bir programlama diline dönüştürülebileceği için, CORBA uygulamaları ve bileşenleri programlama dilinden bağımsızdır. Örneğin, C++ programlama dili ile yazılmış bir istemci bileşen Java programlama dili ile yazılmış bir sunucu bileşenle iletişim kurabilir.

IDL bir uygulama yazma geliştirme dili değildir. IDL kullanılarak bir uygulama yazılamaz. Sadece bileşenlerin arayüzlerini tanımlamaya yarayan standart bir yalancı dildir.

1.5.3 CORBA İletişim Modeli

CORBA, nesneler arasindaki iletişimi kolaylaştırmak için "*nesne referansları*" (object

references) notasyonunu kullanır. Bir uygulamanın bir bileşeni bir CORBA nesnesine ulaşmak istediğinde önce bu nesne için bir nesne referansı almak zorundadır. Nesne referansı belirli bir sunucu nesneyi temsil eden bir semboldur. Bileşen, nesne referansını kullanarak, bu sunucu nesne üzerinde, nesnenin kullanıma sunduğu metodları çağırabilir.

CORBA terminolojisinde, bir istemci bir CORBA nesnesinin sunduğu hizmetleri kullananan, yani bu nesne üzerinde metodlar çağıran bir uygulama yada uygulama bileşenidir. Benzer olarak, bir sunucu da CORBA nesneleri yaratan ve bu nesnelerin sunduğu hizmetleri diğer bileşenlerin yada uygulamaların kullanımına açan bileşen yada uygulamadır. Bir sunucu bileşen başka bir uygulama tarafından yaratılan başka bir sunucu nesnenin hizmetlerini kullanabilir. Bu durumda bir sunucu bileşen, başka bir sunucu bileşenin istemcisi olur.

CORBA ORB'leri genellikle IIOP protokolünü kullanarak iletişim kurarlar. ORB'ler arası iletişim için başka protokoller de mevcut olmasına rağmen IIOP daha hızlı popüler duruma gelmiştir. Bunun nedeni, IIOP'un standart olması ve çok popüler olan TCP/IP protokolü üzerine kurulmuş olmasıdır.

1.5.4 Stubs And Skeletons

CORBA uygulamaları geliştirirken sık karşılaşılan terimlerden ikisi *client stubs* ve *server skeletons* kavramlarıdır. Bir client stub, istemci uygulamanın, ORB'nin istemci tarafındaki parçası ile iletişimi sağlayan kod parçasıdır. Bir server skeleton da, sunucu uygulamanın, ORB'nin sunucu tarafındaki parçası ile iletişimini sağlayan kod parçasıdır (Gokhale ve Schmidt, 1997). Bu kod parçaları genellikle IDL Derleyici'leri tarafından otomatik olarak üretilirler ve kullanılan ORB'ye ve IDL Derleyicisi'ne göre değişirler. Bu kod parçaları üzerinde değişiklik yapılmaz. İstemci ve sunucu uygulamalarla beraber derlenirler.

2. CORBA MİMARİSİ

2.1 Nesne Modeli

Nesne Modeli, CORBA Mimarisi'nde kullanılan nesne kavramları ve terminolojisini tanımlar. CORBA Nesne Modeli, soyut bir model olan OMG Nesne Modeli üzerine kurulmuştur. Yani belirli bir teknoloji ile direkt olarak oluşturulmamış olup soyut olarak genel kavramlar sunulmuştur.

Bir nesne sistemi, servis sağlayıcıları ile bu servisleri kullanan istemcileri birbirinden tanımlanmış bir arayüz yardımı ile soyutlayan bir nesne yığınıdır. Bu soyutlama, özellikle veri gösterimleri ve kod seviyesindedir. Nesne Modeli, istemciler ve nesne uygulamaları ile ilgili kavramları ayrı ayrı açıklar.

Nesne Modeli, istemcilere özel olan kavramları daha ayrıntılı ve zorlayıcı bir şekilde, nesne uygulamalarına özel olan kavramları ise öneri şeklinde açıklar. Bunun nedeni nesne uygulamalarının değişik nesne teknolojileri ile yaratılmasında maksimum esneklik sağlanmasıdır.

CORBA Nesne Modeli, bir istemcinin bir nesneye mesaj gönderdiği klasik nesne modelinin bir örneğidir. Nesne, mesajı yorumlayarak hangi hizmeti yerine getireceğine karar verir.

2.1.1 Nesne Semantiği

Bir nesne sistemi istemcilere hizmetler sunar. Bir hizmetin *istemci'* si istekte bulunabilen herhangi bir mantıksal birimdir.

Bu bölümde istemcilere özel olan kavramlar ele alınacaktır.

2.1.1.1 Nesneler

Bir nesne sistemi nesneler olarak bilinen mantıksal birimleri içerir. Bir *nesne* diğerlerinden ayrılabilen ve istemciler tarafından istenebilecek bir yada daha fazla hizmet sunan bir mantıksal birimdir.

2.1.1.2 İstekler

İstemciler, hizmetleri (servisleri) istekte bulunarak isterler. Bir *istek*, bir nesnenin bir servisi yerine getirmesini talep eden bir mesajdır. Bir istek için gerekli bilgiler, bir operasyon, bir hedef nesne, ve ilgili parametrelerden oluşur.

Bir *değer* (value), bir istek için parametre olabilecek herhangi bir şeydir. Özel olarak, bir değer bir OMG IDL veri tipinin bir örneğidir (instance).

Bir *nesne referansı*, belirli bir nesneyi gösteren bir değerdir. Bir nesne referansı bir isteğin hedef nesnesini, yani servisin hangi nesne üzerinde yerine getirileceğini gösterir. Birden fazla nesne referansı aynı nesneyi gösterebilir.

Bir istek, hedef nesne üzerinde bir hizmetin yerine getirilmesini sağlar. Bir hizmetin yerine getirilmesi sonucunda, eğer hizmet için herhangi bir sonuç yada dönüş değeri tanımlanmışsa, hizmetin sonuçları istemciye dönülür.

Eğer isteğin yerine getirilmesi sırasında bir istisnai durum meydana gelirse, bir hata (exception) dönülür. Hata, bu istisnai durumu tanımlayan parametreler içerebilir.

İstek parametreleri pozisyonları ile tanımlanır. Bir parametre, bir girdi (input) parametre, bir çıktı (output) parametre yada girdi-çıktı (input-output) parametre olabilir. Bir istek ayrıca tek bir *dönüş sonuç değeri* (return result value) dönebilir. Bu dönüş değeri hizmetin sonucunu gösterir.Ayrıca istek, girdi-çıktı yada çıktı parametrelere gerekli olan bilgileri yazarak ek sonuçlar dönebilir.

2.1.1.3 Nesnelerin yaratılması ve öldürülmesi

Nesneler yaratılıp öldürülebilirler (creation-destruction). Nesneler, isteklerin bir sonucu olarak yaratılır yada öldürülürler. Nesne yaratılmasınının sonucu, istemciye yeni bir nesneyi gösteren bir nesne referansı şeklinde yansır.

2.1.1.4 Tipler

Bir *tip* (type) herhangi bir değerin formatını belirler. Belirli bir tipteki bir değer o tipin *örneği*'(instance) dir. Bu değere o tipin *üyesi* (member) denir.

Bir *nesne tipi* üyeleri nesne referansları olan bir tiptir.

Tipler, temel tipler (basic types) yada yapılandırılmış tipler (constructed types) olarak ikiye ayırılırlar.

Temel Tipler:

- 16 bit,32 bit ve 64 bit işaretli ve işaretsiz tamsayılar (integer).
- Tek yoğunluklu (32-bit single precision), çift yoğunluklu (64 bit double precision) kayan

noktalı sayılar (floating numbers).

- Sabit noktalı onluk sayılar (31 basamağa kadar).
- Karakterler.
- DOĞRU yada YANLIŞ değer alabilen Boolean tipi.
- Karakter dizisi. Değişken uzunlukta karakterlerin bir dizisidir. Bir karakter dizisinin uzunluğu bir pozitif tamsayıdır.
- "any" tipi. Hehangi bir temel yada yapılandırılmış tipi gösterebilen bir tiptir.

Yapılandırılmış Tipler:

- Kayıt tipi. (struct,record).
- Belirli bir tipin sabit uzunluktaki dizisi (arrays).
- Belirli bir tipin değişken uzunluktaki dizisi (sequences).

Bir isteğin parametreleri ancak bu tiplerden biri olabilir.

2.1.1.5 Arayüzler

Bir *arayüz* istemcilerin bir nesneden isteyebilecekleri hizmetleri tanımlar. Arayüzler OMG IDL ile tanımlanırlar.

2.1.1.6 Operasyonlar

Bir *operasyon* istenebilecek bir hizmeti gösteren bir tanımdır. Bir *Operasyon tanımlayıcı* ile tanımlanır.

Bir operasyon istek parametrelerini ve dönüş değerlerini tanımlayan bir operasyon imzasına (operation signature) sahiptir. Bir *operasyon imza* 'sı aşağıdaki bileşenlerden oluşur.

- Operasyon için gerekli parametrelerin tarifi.
- Operasyonun sonucunun tarifi.
- Operasyon tarafından dönülecek kullanıcı hatalarının (user exceptions) tanımı.
- Çalıştırma semantiği (execution semantics) tanımı.

Bir operasyon imzasının genel yapısı aşağıdaki gibidir.

```
[oneway] <op_type_spec> <identifier> (param1,…, paramL)
[raises(excet1,…,exceptN)]
```

- İsteğe bağlı **oneway** anahtar kelimesi çalıştırma semantiğini tanımlar. İstemcinin operasyonun bitmesini beklemeden diğer işlere devam edebileceğini belirtir. Bu anahtar kelime belirtilmez ise istemci operasyonun bitmesini beklemek zorundadır.
- **<op_type_spec>**, operasyonun dönüş değerinin tipini belirtir.
- **<identifier>**, operasyonun adını belirtir.
- Operasyon için gerekli parametreler. **in, out, inout** anahtar kelimeleri parametrelerin taşıdığı bilgilerin akış yönünü gösterir.
- İsteğe bağlı **raises** anahtar kelimesi kendinden sonra gelen, **exception** anahatar kelimesi ile tanımlanmış hataların operasyon tarafından döndürülebileceğini belirtir.

Parametreler

Bir parametre modu ve tipi ile tanımlanır. Parametre *mod*'u parametrenin taşıdığı bilginin akış yönünü gösterir. **in** parametrelerdeki bilgi sadece istemciden sunucuya, **out** parametrelerdeki bilgiler sadece sunucudan istemciye aktarılır. **inout** parametreler hem istemciden sunucuya hem de sunucudan istemciye aktarılacak bilgiler için kullanılır. Parametrenin tipi ise parametrenin taşıyabileceği değerin tipini belirtir.

Dönüş Değeri

Operasyonun dönüş değeri sunucudan istemciye dönülecek özel bir değerdir. Genellikle operasyonun başarılı olup olmadığını gösteren bir bilgidir.

Hatalar (Exceptions)

Bir hata, operasyonun yerine getirilmesi sırasında normal olmayan bir durumun ortaya çıktığını gösterir. Meydana gelebilecek farklı olası durumlar için hatalar tanımlanabilir. Bir hata, bu durumu tanımlayacak özel bilgilerin yer aldığı bir kayıt (record) şeklinde tanımlanır.

Hatalar önceden tanımlanmış sisteme özel *sistem hataları* yada operasyona özel olarak tanımlanmış *kullanıcı hataları* olarak ikiye ayrılırlar.

Çalıştırma Semantikleri (Execution Semantics)

Nesne Modeli tarafından iki çalıştırma semantiği tanımlanmıştır. Bir operasyonun semantiği oneway (tek-yönlü) olarak tanımlanmışsa, istemci sunucudan sadece bir operasyonun yerine getirilmesini ister. Bu operasyonunun bitmesini beklemez. Bu nedenle oneway operasyonlar dönüş değeri ve çıktı parametreler içeremez. Sistem oneway operasyonların sunucuya

ulaşacağını yada sadece bir kere çalışacağını garanti edemez. İstemci, oneway tanımlanmamış operasyonların bitmesini, operasyon bir dönüş değeri yada çıktı parametre içermese de beklemek zorundadır. Sistem bu tür operasyonların sadece bir defa ve emin bir şekilde çalıştırılacağını garanti eder.

2.1.2 Nesne Uygulaması

Bu bölümde nesne uygulamasına özel olan kavramlar ele alınacaktır.

Uygulama Modeli iki bölümden oluşur: Çalıştırma Modeli ve Yaratma Modeli. Çalıştırma Modeli hizmetleri yerine getiren operasyonların nasıl çalıştırıldığını tanımlar. Yaratma Modeli hizmetlerin nasıl tanımlandığını belirtir.

2.1.2.1 Çalıştırma Modeli

Yerine getirilmesi istenen bir hizmet bir veri üzerinde işlem yapan bir kodun çalıştırılması sayesinde yerine getirilir. Veri bir sistemin durumunu (state) gösterir. İstenen hizmeti yerine getiren kod sistemin durumunu değiştirebilir.

Hizmetin yerine getirilmesi için çalıştırılan koda *metod* denir. Kod bir çalıştırma motoru (execution engine) tarafından çalıştırılır. Bir çalıştırma moturu, tanımlanan hesaplamaları yapabilecek soyut bir makinadır. Bir metod, yapılacak hesaplamaları tanımlayan, bir çalıştırma motorununun anlayabileceği bilgileri içerir. Bir metodun çalıştırılması metodun aktive edilmesi (method activation) olarak adlandırılır.

Bir istemci bir istekte bulunduğunda hedef nesnenin metodu çağırılır. İstemcinin gönderdiği girdi parametreler metoda geçirilir. Metodun çağrılmasından sonra sonuç değeri ve çıktı parametreler istemciye geri gönderilir.

Bir hizmetin yerine getirilmesi nesnenin durumunu gösteren veriyi değiştirebilir.

2.1.2.2 Yaratma Modeli

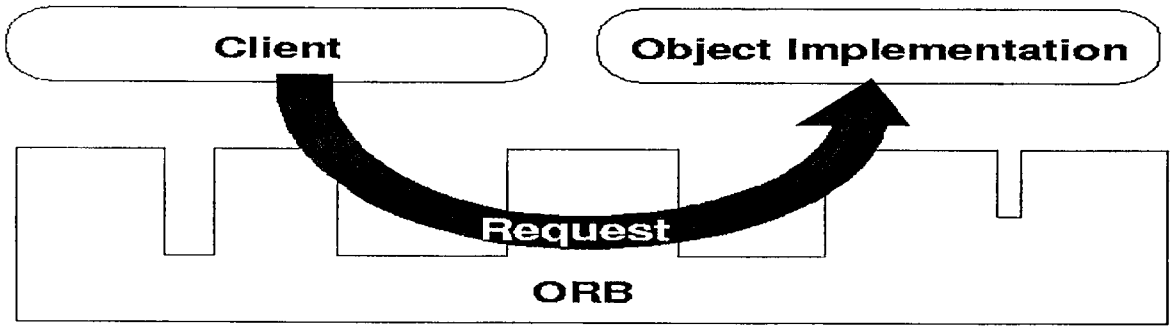
Bir nesne sistemi isteklerin davranışını yerine getirmek için mekanizmalar sağlamalıdır.Bu mekanizmalar nesnelerin yaratılması ve öldürülmesini, nesnenin durumunu, metodları ve metodların nasıl çalıştırılacağını belirten tanımları içerir.

Bir *nesne uygulaması* belirli bir nesne için bu tanımları sisteme sağlayan bir birimdir.

2.2 CORBA Mimarisi'nin Parçaları

2.2.1 Bir ORB'nin yapısı

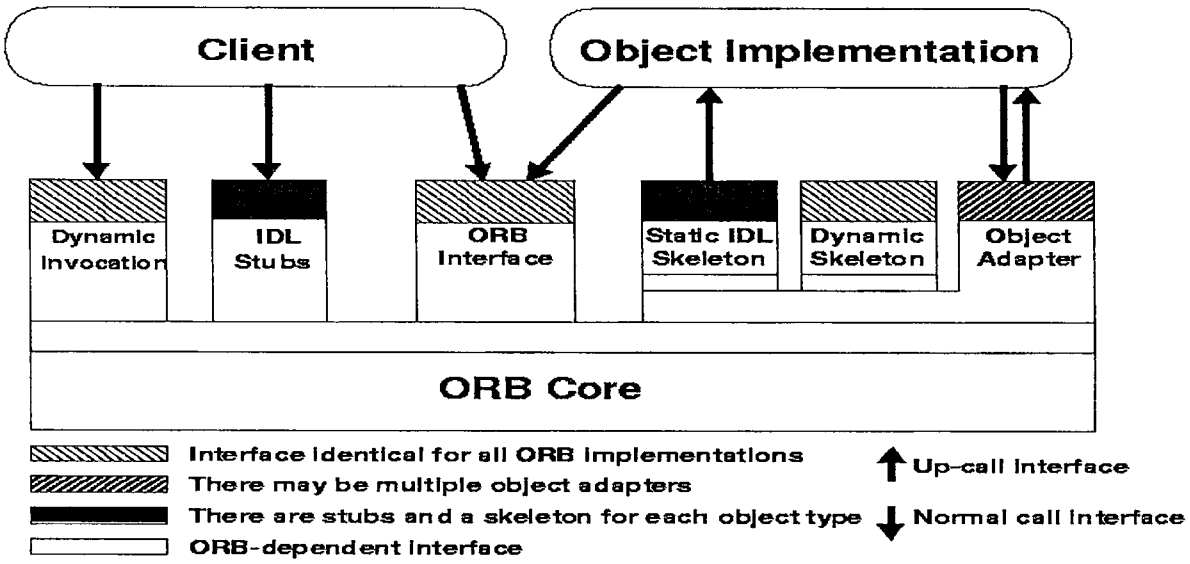
Şekil 2.1'de bir istemci tarafından bir Nesne Uygulaması'na gönderilen istek gösterilmiştir. İstemci, nesne üzerinde bir operasyon gerçekleştirmek isteyen birim, Nesne Uygulaması ise nesneyi oluşturan kod ve veri yığınıdır.



Şekil 2.1 ORB üzerinden gönderilmekte olan bir istek (OMG, 1998).

ORB, isteğin yapıldığı nesne uygulamasını bulmak, nesne uygulamasını istek için hazırlamak ve isteği oluşturan parametreleri nesne uygulamasına ulaştırmak için gereken tüm işlemlerden sorumludur. İstemcinin gördüğü arayüz, nesnenin nerede olduğundan, nesnenin hangi programlama dili ile yazıldığından yada nesnenin arayüzünde belirtilmeyen herhangi bir diğer durumdan tamamen bağımsızdır.

Şekil 2.2'de bir ORB'nin yapısı gösterilmektedir.



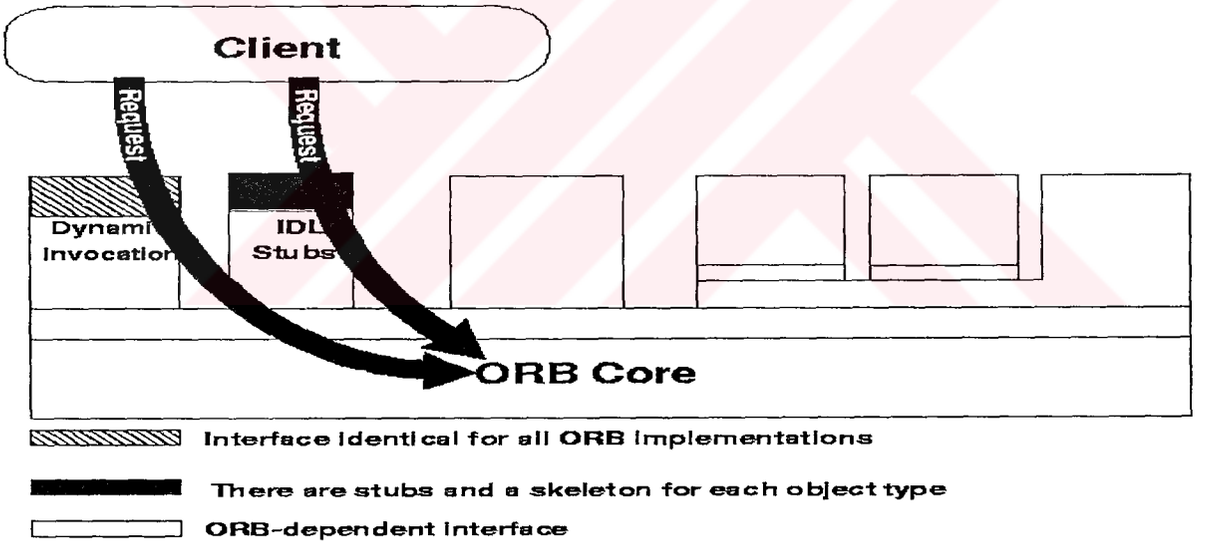
Şekil 2.2 Bir Object Request Broker'ın yapısı (OMG, 1998).

İstemci, bir istekte bulunmak için DII arayüzünü yada OMG IDL stub'ını kullanabilir. Ayrıca istemci bazı işlemler için direkt olarak ORB ile iletişim kurabilir.

Nesne Uygulaması isteği ya OMG IDL tarafından üretilmiş statik taslak üzerinden yada dinamik taslak üzerinden alır. Nesne Uygulaması herhangi bir anda Nesne Adaptörü yada ORB ile direkt iletişim kurabilir.

Nesnelerin arayüzleri iki yolla tanımlanabilir. Arayüzler OMG IDL kullanılarak statik olarak tanımlanabilir. Bu dil nesneler üzerinde uygulanabilecek operasyonlara göre nesne tiplerini ve bu operasyonlar için gerekli parametreleri tanımlar. Ayrıca arayüzler bir Arayüz Deposu (Interface Repository) servisine eklenerek tanımlanabilir. Bu sayede uygulamanın çalışması sırasında (run-time) yeni arayüzler tanımlanabilir.

İstemci bir Nesne Referansı aldıktan sonra, nesnenin tipini ve gerçekleştirilecek operasyonu bilerek bir istekte bulunur. İstemci, isteği, bu nesneye özel olarak tanımlanmış stub rutinlerini çağırarak yada Dinamik Çağırma Arayüzü (Dynamic Invocation Interface) servisini kullanıp yeni bir istek oluşturarak başlatır (Şekil 2.3).

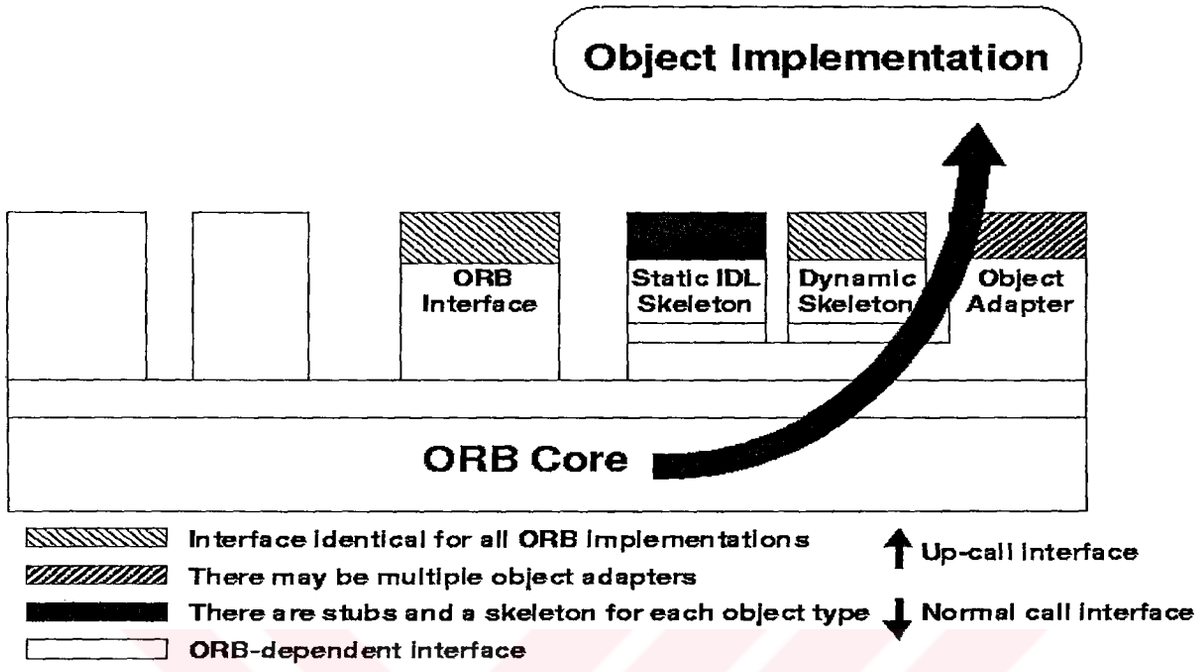


Şekil 2.3 İstemci Stub'ını yada Dinamik Çağırma Arayüzü'nü kullanan bir istemci (OMG, 1998).

Dinamik Arayüz yada stub rutinleri kullanılarak yapılan istek tamamen aynı yapıda olup isteği alan Nesne Uygulaması isteğin hangi yöntemle yapıldığını ayırt edemez.

ORB uygun nesne uygulaması kodunu bulur, parametreleri gönderir ve IDL taslağı yada dinamik taslağı kullanarak kontrolü Nesne Uygulaması'na bırakır (Şekil 2.4). Taslaklar arayüze ve nesne adaptörlerine göre değişebilirler. Nesne uygulaması isteğin yerine

getirilmesi sırasında Nesne Adaptörü aracılığıyla ORB'nin bazı servislerini kullanabilir. İstek tamamlandığı zaman kontrol ORB'ye döner. ORB çıkış parametrelerinin değerlerini ve sonuç değerlerini istemciye döndürür.



Şekil 2.4 Bir istek almakta olan bir Nesne Uygulaması (OMG, 1998).

Nesne Uygulaması hangi Nesne Adaptörü'nü kullanacağını gereksinimlerine göre kendi belirler.

2.2.1.1 Object Request Broker

Mimaride, ORB'nin tek bir bileşen olarak uygulanması zorunlu kılınmamış, ORB, arayüzleri ile tanımlanmıştır. Uygun arayüzü sunan herhangi bir ORB uygulaması kabul edilebilir. Bu arayüz üç kategoriye ayrılmıştır.

1. Tüm ORB uygulamaları için aynı olan operasyonlar.
2. Belirli nesne tiplerine özel olan operasyonlar.
3. Belirli nesne uygulaması stillerine özel operasyonlar.

Farklı ORB'ler farklı uygulama seçimleri yapabilirler. Ve IDL derleyicileri, depolar ve çeşitli nesne adaptörleri ile değişik özellikte ve değişik amaçlı servisler sunabilirler.

Farklı nesne referansı gösterimlerine ve operasyon gerçekleştirme tarzlarına sahip birden fazla ORB uygulaması olabilir. Bir istemci farklı ORB'ler tarafından yönetilen iki ayrı nesne referansına sahip olabilir. İki ORB'nin aynı anda çalışması istendiğinde her bir ORB kendi

nesne referanslarını ayırt edebilmelidir. Bu ayrım işleminin yapılması istemcinin sorumluluğunda değildir.

Çekirdek ORB (ORB Core), ORB'nin nesnelerin ve isteklerin temel gösterimini sağlayan parçasıdır. CORBA farklı nesne mekanizmalarını destekleyebilmek için tasarlanmıştır. Bu amacına Çekirdek ORB üzerine diğer ORB bileşenlerini yapılandırarak ulaşmaktadır.

2.2.1.2 İstemciler

Bir nesnenin istemcisi, bu nesnenin bir nesne referansını alıp bu nesne üzerinde operasyonlar gerçekleştirir. Bir istemci, bir nesnenin arayüzü yardımıyla sadece mantıksal yapısını bilir ve nesne, davranışını kendisine gönderilen isteklerle ortaya koyar. Genel olarak, istemci olarak kastettiğimiz bir nesne üzerinde istekte bulunan bir program yada bileşen olduğu halde istemci kavramı görecelidir. Örneğin, bir nesnenin uygulaması (sunucu) diğer bir nesnenin istemcisi olabilir.

İstemciler genellikle nesneleri ve ORB arayüzünü, bir programlama dili dönüşümü yardımıyla görürler. Dönüşüm yoluyla, kullanılan programlama diline dönüştürülen arayüzler, biraz daha somut bir yapıya kavuşurlar. İstemciler nesne uygulaması, nesne uygulamasının nasıl yazıldığı, kullanılan nesne adaptörü hakkında hiç bir bilgiye sahip değildir.

2.2.1.3 Nesne Uygulamaları

Bir nesne uygulaması, nesne için gereken veriyi ve metodları için gereken kodu içeren ve nesnenin davranışını yerine getiren bir birimdir. Nesnenin davranışını yerine getirmek için genellikle diğer nesneleri yada ek yazılımları kullanırlar.

Genel olarak, nesne uygulamaları ORB'ye ve istemcinin nesnenin metodlarını nasıl çağırdığına bağımlı değillerdir. ORB ile iletişim kurmak için istedikleri Nesne Adaptörü'nü kullanabilirler.

2.2.1.4 Nesne Referansları

Bir Nesne Referansı bir ORB'ye bir nesneyi tanıtmak için kullanılan bir bilgidir. Herhangi iki ORB'nin Nesne Referanslarını gösterim ve kullanım şekli farklı olabilir.

Bir istemcinin, bir nesne için aldığı herhangi bir nesne referansı sadece bu istemcinin çalışma süreci içinde geçerlidir. İstemci yeniden çalıştığında yeni bir nesne referansı almak zorundadır.

Bir programlama dili için bir nesne referansının dönüşümü tüm ORB'lerde aynı olmak zorundadır. Bu sayede belirli bir programlama dili ile yazılmış bir programın, kullanılan ORB' den bağımsız olarak, bir nesne referansını kullanabilmesi mümkün olur.

Ayrıca hiçbir nesneyi göstermeyen, ve diğer tüm nesne referanslarından farklı olduğu garanti edilen bir nesne referansının olması gerekir.

2.2.1.5 OMG Interface Definition Language (IDL)

OMG IDL nesnelerin arayüzlerini belirterek, nesnelerin tiplerini tanımlar. Bir arayüz, isimlendirilmiş operasyonlar ve bu operasyonlara geçilecek parametrelerin bir kümesidir.

IDL, belirli bir nesne uygulamasını kullanılabilecek olan istemcilere, bu nesnenin hangi operasyonlarının olduğunu ve nasıl çağrılacaklarını belirtir. CORBA nesneleri, IDL tanımlarından dönüşüm yoluyla belirli bir programlama diline dönüştürülürler.

2.2.1.6 OMG IDL'in programlama dillerine dönüşümü

Nesneye dayalı yada nesneye dayalı olmayan farklı programlama dilleri için, CORBA nesnelerine ulaşım metodu farklı olabilir. Nesneye dayalı programlama dillerine yapılan dönüşümlerde, CORBA nesnelerini programlama dili nesneleri olarak gösterebilmek kolaylık sağlar. Nesneye dayalı olmayan dillerde, nesne referanslarının gerçek ORB gösterimleri, metod isimleri gibi özelliklerin gizlenmesi gerekir. OMG IDL'in bir programlama diline dönüşümü tüm ORB uygulamalarında aynı olmalıdır. Bu dönüşüm dile özel veri tiplerinin ve ORB üzerinden nesnelere ulaşımı sağlayacak rutin tanımlarını içerir.

Bir dönüşüm, ayrıca istemci stub'ının yapısını, dinamik çağırma arayüzünü, uygulama taslağını, nesne adaptörlerini ve direk ORB arayüzünü ve nesne metodlarının senkron yada asenkron olup olmadığını da içerir. Genellikle senkron metod çağrımı kullanılmakta olup, istemci nesne metodunu çağırdıktan sonra programa devam etmek için bu metodun işinin bitmesini bekler. Asenkron metod çağrımında ise istemci istek çağrımını başlatır ve kontrol metodun bitmesini beklemeden istemciye döner. Metod işini yerine getirirken istemci başka bir işi yerine getirebilir. Tabii ki bu metodun işinin ne zaman bittiğini ve sonuç değerini öğrenebilmek için programlama diline özel ek rutinlere ihtiyaç olacaktır. Bu durumda dönüşüm bu tür ek rutinleri de içerecektir.

2.2.1.7 İstemci Stubları

Nesneye dayalı olmayan programlama dilleri için her arayüz tipi için bir programlama arayüzü olacaktır. Genel olarak stublar bir nesne üzerinde IDL ile tanımlanmış operasyonlara ulaşımı sağlayan kod parçalarıdır. Bu kod parçaları, nesne arayüzleri IDL derleyicileri ile derlendiğinde seçilen programlama dili için otomatik olarak üretilirler. İstemci stub kodu çağırır, stub kodu da ORB arayüzünü kullanarak operasyonlara ulaşır. Birden fazla ORB varsa her ORB için farklı stub kodları bulunabilir. Bu durumda ORB'nin ve dil dönüşümünün, belirli bir nesne referansı için uygun stub kodunun bulunması için beraber çalışması gerekir.

Nesneye dayalı programlama dillerine yapılan dönüşümlerde istemci stublara gerek duyulmaz. Çünkü ORB nesneleri direkt olarak programlama dilinin nesneleri ile eşleştirilirler.

2.2.1.8 Dinamik Çağırma Arayüzü

Belirli bir nesne üzerindeki belirli bir operasyona özel olarak hazırlanmış statik stub rutinini çağırmak yerine, çağrımların programın çalışması sırasında dinamik olarak yerine getirilmesine olanak sağlayan bir arayüzdür. Bunun için istemci önce nesneyi, sonra o nesnenin çağrılacak metodunu, daha sonra da metod için gerekli parametreleri dinamik çağırma arayüzünü kullanarak belirtmelidir. Dinamik Çağırma Arayüzü (Dynamic Invocation Interface) farklı programlama dillerine göre farklılıklar taşıyabilir.

2.2.1.9 Uygulama Taslağı

Belirli bir programlama dili dönüşümü için, nesnelerin operasyonlarını yerine getiren belirli bir arayüz olacaktır. Bu arayüz, bir istemci bir operasyon isteğinde bulunduğunda, ORB'nin taslağı kullanarak çağıracağı metodları içeren bir arayüz olmalıdır. Bu taslaklar, nesne arayüzleri IDL derleyicisi ile derlendiğinde otomatik olarak oluşturulacaktır.

Bir istemci stub'ına karşılık bir uygulama taslağının olması gerekli değildir. Bir istemci Dinamik Çağırma Arayüzü'nü kullanarak bir istekte bulunabilir.

2.2.1.10 Dinamik Taslak Arayüzü

Dinamik Taslak Arayüzü (Dynamic Skeleton Interface), nesne metodlarını dinamik olarak çağırmaya yarayan bir arayüzdür. Bu arayüz istemci tarafındaki Dinamik Çağırma Arayüzü'nün uygulama (sunucu) tarafındaki eşleniğidir. Nesne Adaptörü bir operasyona, statik bir taslak yerine, nesnenin, operasyonun adı ve parametrelerine dinamik olarak

ulaşımını sağlayan bir arayüz kullanarak ulaşır.

Nesne uygulaması kodu, ORB'ye tüm parametrelerin tanımlarını vermek zorundadır. ORB de bu bilgilere göre parametrelerin değerlerini nesne uygulamasına geçirir. Nesne, operasyonu yerine getirdikten sonra çıktı parametrelerini ve dönüş değerini ORB'ye geçirir.

Dinamik Taslak Arayüzü, kullanılan programlama dili dönüşümü ve nesne adaptörüne göre değişebilir.

Dinamik Taslak Arayüzü'nün kullanılacağı, nesne uygulaması tarafından sunucu tarafındaki ORB'ye bildirilir. İstemci, çağrımı istemci stublarıyla yada Dinamik Çağırma Arayüzü ile yapmış olabilir. İstemci nesne uygulamasının operasyonu hangi yöntemle çalıştırdığını bilmez. Kullanılan yöntemin istemci için bir farkı yoktur. İki yöntem eşdeğerlidir ve aynı sonucu üretir.

2.2.1.11 Nesne Adaptörleri

Bir Nesne Adaptörü (Object Adapter), bir nesne uygulamasının ORB tarafından sunulan hizmetlere ulaşabilmesi için kullandığı arayüzlerden oluşur. Nesne Adaptörlerinin ulaşım sağladığı ORB tarafından sunulan hizmetler genellikle nesne referanslarının oluşturulması ve yorumlanması, metod çağrımı, etkileşimlerin güvenliği, nesne ve uygulama aktivasyonu ve deaktivasyonu, nesne referanslarından uygun nesne uygulamalarının bulunması ve nesne uygulamalarının ORB'ye kaydedilmesi gibi işlemlerdir.

2.2.1.12 ORB Arayüzü

ORB Arayüzü direkt olarak ORB ile iletişim sağlayan, tüm ORB'lerde aynı olan ve nesne adaptörlerine ve nesnelerin arayüzlerine bağımlı olmayan temel bir arayüzdür. ORB'nin bütün fonksiyonalitesi, nesne adaptörleri, stublar, taslaklar yada dinamik çağırma arayüzleri ile sağlandığı için bu arayüzde bütün nesneler için gerekli olan birkaç operasyon mevcuttur. Bu operasyonlar hem istemciler hem de nesne uygulamaları için gerekli olan kullanışlı operasyonlardır.

2.2.1.13 Arayüz Deposu

Arayüz Deposu (Interface Reopsitory), çalışma zamanında (run-time), nesnelerin arayüz bilgilerinin alınabilmesine imkan veren bir servistir. Arayüz Deposu'ndan alınan bu nesne arayüz bilgileri, ORB tarafından isteklerin yerine getirilmesi için kullanılır. İstemci program

derlenirken mevcut olmayan nesne arayüzleri, programın çalışması sırasında Arayüz Deposu'ndan alınabilir ve bu arayüzler kullanılarak, istenen nesne üzerinde operasyonlar gerçekleştirilebilir.

2.2.1.14 Uygulama Deposu

Uygulama Deposu (Implementation Repository), ORB'nin nesne uygulamalarını bulması ve aktif duruma geçirebilmesine olanak sağlayan bilgileri tutar. Genellikle bu bilgiler işletim sistemine yada bir ORB'ye özel olan bilgilerdir.

Örneğin, bir sistemde bir nesne uygulaması mevcut değilse, bu sisteme nesnenin kurulumu yapıldığında, nesne için gerekli bilgiler Uygulama Deposu'na kaydedilebilir. Daha sonra ORB bu bilgileri kullanarak nesne uygulamasını bulup metodlarını çağırabilir.

2.2.2 ORB türleri

CORBA Mimarisi'nde çok çeşitli ORB uygulamaları mevcut olabilir. Bu bölümde birkaç farklı seçeneği ele alacağız. Bir ORB birden fazla seçeneği destekleyebilir ve iletişim için birden fazla protokolü kullanabilir.

2.2.2.1 Kütüphane Bazlı ORB

Bir ORB, ORB metodları bir kütüphane (library) içinde yer alacak şekilde yaratılmış olabilir. İstemci ve uygulama programları bu kütüphane içindeki metodları kullanarak ORB ile ilgili işlerini yerine getirirler. Bu kütüphane, istemci ve uygulama programları ile beraber derlenir. Dolayısıyla ORB modülü, her bir CORBA programının içinde ayrı ayrı yer alır.

2.2.2.2 Sunucu Bazlı ORB

ORB'nin yönetimini merkezileştirmek ve kolaylaştırmak için, tüm istemciler ve uygulamaların bir ORB Sunucusu ile iletişim kurduğu bir sistem kurulabilir. Bu sistemde ORB Sunucusu istemciden isteği alır ve uygulamaya yöneltir. ORB, sunucuda çalışan bir program olabilir ve normal bir iletişim mekanizması kullanılabilir.

2.2.2.3 Sistem Bazlı ORB

Güvenlik ve performansı artırmak için bir ORB, bir işletim sistemine entegre edilmiş bir servis şeklinde dizayn edilmiş olabilir. Bu şekilde ilgili işletim sistemine özel optimizasyonlar yapılabilmesi ve programların gücünün artırılması mümkün olabilir.

2.2.3 Bir İstemcinin yapısı

Bir nesnenin istemcisi, bu nesneyi gösteren bir nesne referansına sahiptir. Bir nesne referansı nesneleri birbirinden ayıran anahtar bir bilgidir.

ORB, kontrolün ve bilginin nesne uygulamasına transfer edilmesi ve uygulamadan da istemciye geri döndürülmesi işlemlerini yerine getirir. ORB'nin metodu çağıramaması halinde bir hata (exception) dönülür.

İstemciler, programlar içerisinde nesne tiplerine özel stublara kütüphane rutinleri şeklinde ulaşırlar. Bu nedenle istemciler, bu rutinleri, kullandıkları programlama dilindeki normal bir rutin gibi görürler. Tüm ORB uygulamaları, nesneleri gösteren, genellikle bir göstergeç (pointer) olan dile özel bir veri tipi sunarlar. İstemci, metod çağrımını başlatmak için bu nesne referansını stub rutinlerine geçirir. Stub rutinleri nesne referansının yapısını bildiğinden, bu nesne referansının hangi nesneyi gösterdiğini bilir ve ORB'yi kullanarak metod çağrımını yerine getirir.

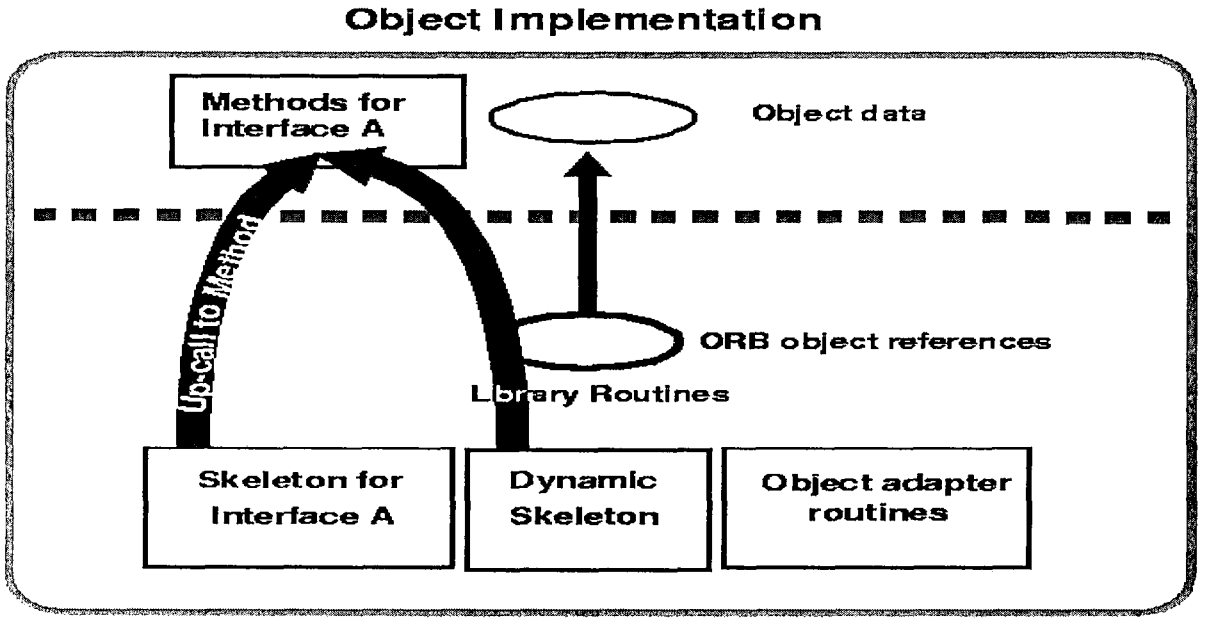
Nesnenin arayüzü istemci programın derlenmesi sırasında belirli değilse, nesneler üzerinde metod çağrımları yapabilmek için ek kütüphane rutinlerine ihtiyaç vardır. Bu durumda istemci program nesneyi ve operasyonları tanımlayabilmek için ek bilgiler sağlamak zorundadır.

İstemciler genellikle nesne referanslarını, nesne referansları bilinen nesneler üzerinde uygulanan metodların dönüş değerlerinden veya sonuçlarından alırlar. Bir istemci aynı zamanda bir nesne uygulaması ise, gerekli nesne referanslarını, metodlarına geçirilen parametreler şeklinde alır. Ayrıca bir nesne referansı, daha sonra kullanılmak amacıyla dosyalarda saklanmak yada başka nesnelerin metodlarına parametre olarak geçmek için bir karakter dizisine çevrilebilir. Daha sonra bu dizi nesne referansını ve bu diziyi oluşturan ORB tarafından nesne referansına çevrilebilir.

2.2.4 Bir Nesne Uygulaması'nın yapısı

Nesne Uygulaması, bir nesnenin durumunu ve davranışını gösterir. Nesne Uygulaması, çeşitli şekillerde yapılandırılmış olabilir. Bir nesne uygulaması, nesnenin davranışlarını yerine getiren metodları ve durumunu gösteren nesne verilerini tanımlar, nesnenin aktivasyonu ve deaktivasyonu için gerekli rutinleri sunar.

Nesne uygulaması yeni nesneleri yaratmak, kendini ORB'ye kaydetmek ve ORB bağımlı servislere ulaşmak için çeşitli yollardan ORB ile iletişim kurar (Şekil 2.5). Genellikle ORB servislerine ulaşmak için Nesne Adaptörü'nü kullanır.



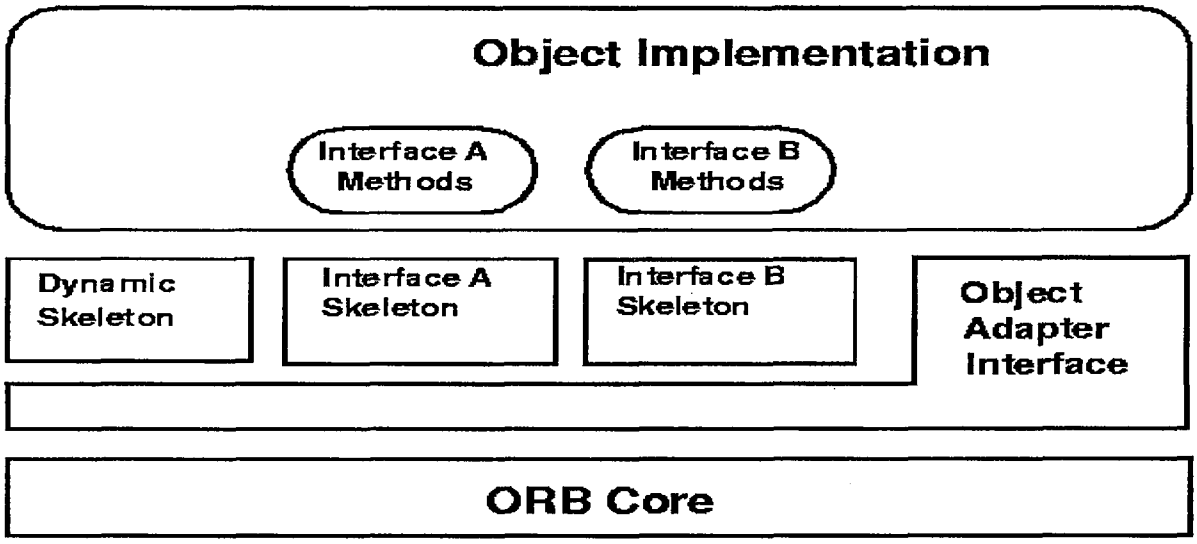
Şekil 2.5 Tipik bir Nesne Uygulaması'nın yapısı (OMG, 1998).

Bir çağrım yapıldığında, Çekirdek ORB, nesne adaptörü ve taslak, uygulamanın ilgili metodunu çağırır. Bu metoda, nesnenin verisini bulmakta kullanılacak, nesneyi tanımlayan bir parametre (genellikle bir göstergeç) geçilir. Ayrıca metodun arayüz tanımında belirtilen gerekli parametreler geçirilir. Metodun çalışması sırasında, nesne uygulaması çeşitli amaçlarla Nesne Adaptörü ve ORB ile iletişim kurabilir. Metodun çalışması bittiğinde kontrol uygulamadan, istemciye geri aktarılır ve dönüş değerleri istemciye gönderilir.

Yeni bir nesne yaratıldığında, ORB'ye nesnenin yaratıldığı bildirilir. Böylece ORB bu nesnenin uygulamasını nerede bulacağını bilir. Uygulama ORB'ye, hangi arayüzün uygulaması olduğunu ve uygulama aktif durumda değilse ORB'nin uygulamayı nasıl aktif duruma geçireceğini bildirerek kendini kaydeder.

2.2.5 Bir Nesne Adaptörü'nün yapısı

Bir Nesne Adaptörü, bir nesne uygulamasının, nesne referansı oluşturma gibi hizmetler için ORB'ye ulaşmasının temel yoludur (Şekil 2.6). Bir Nesne Adaptörü, nesne uygulamasına açık (public) olarak sunulan, bir taslağa gizli (private) olarak sunulan bir arayüzdür. Gizli bir ORB-bağımlı arayüz üzerine kurulmuştur.



Şekil 2.6 Tipik bir Nesne Adaptörü'nün yapısı (OMG, 1998).

Nesne Adaptörleri aşağıdaki işlemlerden sorumludur (Vinoski, 1993):

- Nesne Referanslarını yaratılması ve yorumlanması
- Metod çağrımı
- Etkileşim güvenliği
- Nesne ve uygulama aktivasyonu ve deaktivasyonu
- Nesne referanslarından nesne uygulamalarının bulunması
- Uygulamaların kaydedilmesi

Bu işlemler Çekirdek ORB ve gerekli ek bileşenler kullanılarak yerine getirilir. Bir nesne adaptörü bir yada bir kaç işini üzerine kurulduğu Çekirdek ORB'ye yönlendirebilir.

Bir nesne uygulamasının ihtiyaç duyduğu servis Çekirdek ORB' de bulunmuyorsa, bu servisi Nesne Adaptörü sağlar. Bu hizmeti Çekirdek ORB sağlıyorsa Nesne Adaptörü kendine gelen servis isteğini Çekirdek ORB'ye yönlendirir. Bir Nesne Adaptörü, tüm ORB'ler için aynı temel arayüzü sağlamalıdır. Ama belirli ORB'ler için, ORB'ye özel ek arayüzler sağlanabilir.

2.2.6 Nesne Adaptörü türleri

Nesne Adaptörlerinin bir çok çeşidi vardır. Ancak nesne uygulaması nesne adaptörlerine bağımlı olduğundan az sayıda olması daha yararlıdır. Nesne adaptörleri genellikle geniş bir nesne uygulaması yelpazesine hitap ettikleri için ancak bir nesne uygulaması farklı bir servise ihtiyaç duyduğunda yeni bir nesne adaptörü gerekebilir.

2.2.6.1 Temel Nesne Adaptörü

CORBA'nın 2.2 versiyonundan önceki sunumlarında, bir çok ORB nesnesi için kullanılabilecek genel amaçlı bir Temel Nesne Adaptörü (Basic Object Adapter-BOA) tanımlanmıştır. Bu Nesne Adaptörü için nesne uygulamaları ayrı programlardır. Her metod için ayrı bir program çalıştırılmasına, her bir nesne için ayrı bir program çalıştırılmasına yada bir nesne tipine ait tüm nesneler için aynı programın sunduğu nesne uygulamasının kullanılmasına olanak tanır. TNA bir uygulama aktif değilse kullanılan yönteme göre uygulamayı içeren programı otomatik olarak çalıştırır.

2.2.6.2 Taşınabilir Nesne Adaptörü

Temel Nesne Adaptörü genel amaçlı ORB nesneleri için kullanılabiliyordu. Ancak Temel Nesne Adaptörü'nün tanımı eksiksiz olarak yapılmamıştı. Örneğin bir nesne uygulamasının Temel Nesne Adaptörü'ne kaydedilme metodu standart olarak belirlenmemişti. Bu nedenle farklı ORB uygulamaları, ORB'lerini tamamlamak için bu işlemi farklı şekillerde yerine getiriyorlardı. Sonuçta bir ORB için yazılmış CORBA programları diğer bir ORB ile çalışamıyordu.

Bu nedenle OMG, CORBA 2.2 versiyonunda Taşınabilir Nesne Adaptörü'nü tanımladı. Taşınabilir Nesne Adaptörü (Portable Object Adapter – POA) tam anlamıyla eksiksiz bir tanıma sahiptir. Taşınabilir Nesne Adaptörü, Temel Nesne Adaptörü'nün eksikliklerini tamamen kapatmasının yanında yeni özellikleri ile daha çok kullanılır hale geldi.

POA'nın en önemli özelliği adından da anlaşıldığı gibi taşınabilir olmasıdır. Yani bir ORB için yazılmış bir CORBA programı hiç bir değişiklik yapmadan CORBA 2.2 uyumlu başka bir ORB ile çalışabilir.

POA' nın bir diğer önemli özelliği, BOA'dan daha fazla nesne uygulaması yaratma yöntemi sunmasıdır. POA ile nesne uygulamasının tipi ve davranışı üzerinde daha fazla kontrol sağlanabilmekte ve bu sayede çok değişik tipte nesne uygulamasının yaratılabilmesi mümkün olmaktadır (Schmidt ve Vinoski, 1997).

2.3 Temel Nesne Adaptörü

2.3.1 Giriş

Bir nesne adaptörü, bir nesne uygulamasının ORB fonksiyonlarına ulaşmak için kullandığı

ana yoldur. Temel Nesne Adaptörü, çeşitli nesne uygulamalarını destekleyebilmesi için tasarlanmış genel bir arayüzdür. TNA nesne referanslarının yaratılması, uygulamaların kaydedilmesi, uygulamaların aktivasyonu ve deaktivasyonu ve isteklerin yetki kontrolü gibi hizmetleri sunar.

TNA'nın arayüzlerinin çoğu OMG IDL ile gösterilebilir, çünkü bu arayüzler nesne adaptörü üzerindeki operasyonları tanımlar.

2.3.2 TNA'nın rolü

TNA, her ORB uygulamasında mutlaka yer almalıdır. ORB bağımlı bir yapıya sahip olmasına rağmen, TNA'yı kullanan nesne uygulamaları her ORB uygulamasında çalışabilmelidir.

Nesne Uygulamaları, TNA'dan başka nesne adaptörlerini de kullanabilirler. Genel olarak nesne uygulamasının hangi nesne adaptörünü kullandığı, bir nesnenin istemcisini ilgilendirmez.

Temel Nesne Adaptörü aşağıdaki fonksiyonları sunar:

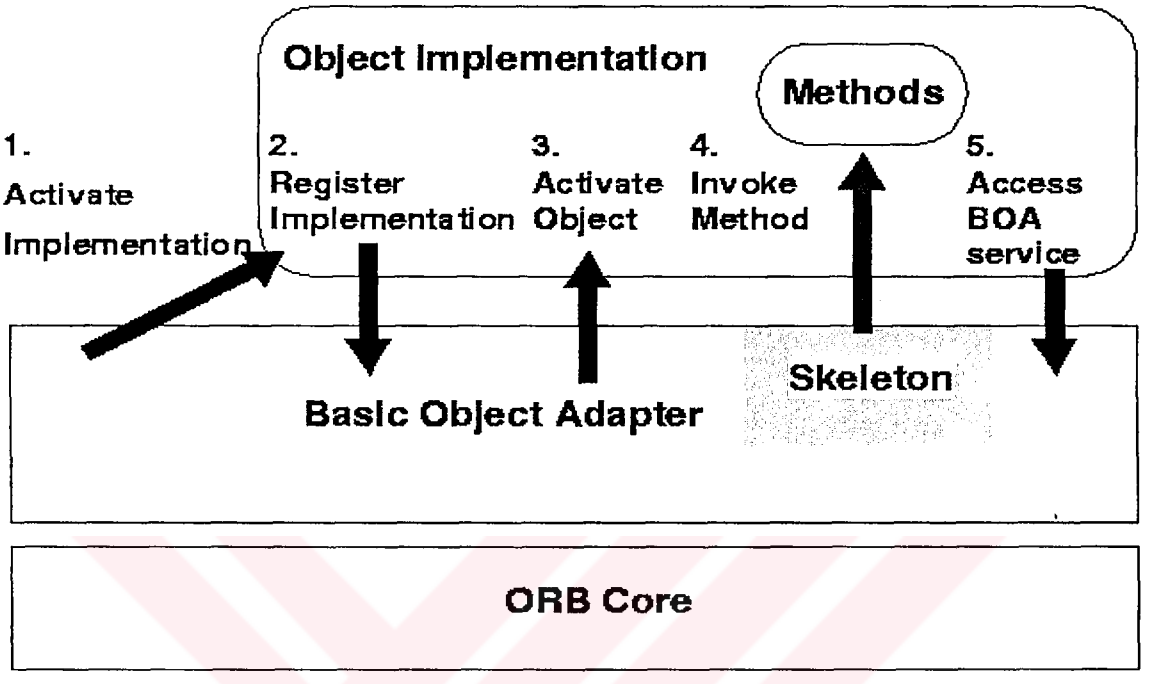
- Nesne referanslarının yaratılması ve yorumlanması
- Çağrımı yapan istemcinin yetki kontrolü
- Nesne uygulamasının başlatılması, aktivasonu ve deaktivasyonu
- Taslaklar kullanılarak metodların çağırımı

TNA bir yada daha fazla programdan oluşan nesne uygulamalarını destekler. TNA, ORB'nin parçası olmayan sistem fonksiyonlarını kullanarak bu programları aktive eder ve bu programlarla iletişim kurar. Bu nedenle TNA nesne uygulamalarını içeren programları bulmak ve çalıştırabilmek için, üzerinde çalıştığı sisteme özel, taşınabilir olmayan bazı bilgilere ihtiyaç duyar (programın bulunduğu sürücü ve dizin gibi). TNA, bu bilgileri kendisi tutup kontrol etmek yerine Uygulama Deposu'ndan alır.

Aktive edilen programı TNA'ya ve ORB'ye bağlamak için kullanılan mekanizma belirtilmemiştir. Çünkü bu mekanizma işletim sistemine ve programlama diline bağımlıdır. TNA herhangi bir mekanizma kullanarak taslaklara bağlanır. Aktivasyon işleminden sonra TNA uygulama içindeki rutinleri çağırabilir ve uygulama da TNA üzerindeki rutinleri çağırabilir.

Şekil 2.7'de TNA'nın yapısı ve TNA ile Nesne Uygulama'sının bazı etkileşimleri gösterilmiştir. TNA Nesne Uygulaması'nın içinde bulunduğu programı çalıştırır (1). Nesne

Uygulaması TNA'ya hazırlığının bittiğini ve istekler için hazır olduğunu bildirir (2). Belirli bir nesne için ilk istek geldiğinde uygulamaya nesneyi aktive etmesi için bir mesaj gönderilir (3). TNA taslağı kullanarak uygun metodu çalıştırır (4). Uygulama nesne yaratılması ve deaktive edilmesi gibi işlemler için TNA üzerindeki metodları çağırabilir (5).



Şekil 2.7 Temel Nesne Adaptörü'nün yapısı ve çalışma şekli (OMG, 1998).

TNA, Nesne Uygulaması için gereken operasyonları kullanıma sunar (export). Nesne uygulamasının gerek duyduğu hizmetleri Çekirdek ORB ile TNA ortaklaşa çalışarak sağlarlar.

2.3.3 TNA Arayüzü

TNA Arayüzü OMG IDL ile tanımlanmıştır, bu sayede Nesne Uygulaması'nın yazıldığı programlama diline dönüşüm yapılarak Nesne Uygulaması'nın TNA' ya ulaşması sağlanır.

Pratikte, TNA'nın bir parçası ayrı bir bileşen olarak bir parçası da Nesne Uygulaması tarafından kullanılan bir kütüphane olarak uygulanır. TNA'nın ayrı bileşen olarak kullanılan bölümü, nesne uygulamasının aktive edilmesi için kullanılır. Kütüphane olarak kullanılan bölümü ise metodlar ve taslak arasındaki iletişimi kurmak için kullanılır. Bu parçaların taşıdığı TNA fonksiyonalitesi, TNA uygulamasına göre değişebilir.

Aşağıda TNA arayüzünün tanımı verilmiştir.

```
module CORBA {                                    // Pseduo IDL
  interface InterfaceDef;
  interface ImplementationDef;
  interface Object;

  typedef sequence< octet, 1024> ReferenceData;

  interface BOA {
    Object create(
      in ReferenceData id,
      in InterfaceDef intf,
      in ImplementationDef impl
                );

    void          dispose( in Object obj );
    ReferenceData get_id( in Object obj );

    void change_implementation( in Object obj,  in ImplementationDef impl );

    void impl_is_ready( in ImplementationDef impl );
    void deactivate_impl( in ImplementationDef impl );

    void obj_is_ready( in Object obj,  in ImplementationDef impl );
    void deactivate_obj( in Object obj );
  };
};
```

TNA'dan istenebilecek isteklerin çeşitleri aşağıdaki gibidir.

- Nesne referansları yaratmak yada öldürmek, yada TNA'nın bir nesne referansı için tuttuğu bilgilerin sorgulanması ve güncellenmesi için gerekli operasyonlar
- Belirli bir istekle ilişkilendirilmiş operasyonlar
- Aktif nesne ve uygulamaların kaydı için gerekli operasyonlar.

TNA'nın bir uygulamadan taslaklar yoluyla isteyebileceği istekler de aşağıdaki gibidir.

- Bir uygulamanın aktivasyonu
- Bir nesnenin aktivasyonu
- Bir operasyonun çağırılması (taslak yolu ile).

2.3.3.1 Uygulamaların kaydedilmesi

TNA, uygulamaları tanımlayan bilgilerin bir Uygulama Deposu'nda tutulmasını zorunlu kılar. Genellikle uygulamalara ait bilgiler, uygulama sisteme kurulduğunda Uygulama Deposu'na yazılır. OMG IDL'de bu gerekli bilgileri tutmaya yarayan **ImplementationDef** adında bir arayüz tanımlanmıştır.

2.3.3.2 Uygulamaların Aktivasyonu ve Deaktivasyonu

TNA, bir nesnenin operasyonunun yerine getirilmesi sırasında iki çeşit aktivasyona ihtiyaç duyar. İlki, bir nesne için istenen operasyonu yerine getirecek nesne uygulamasının hazır olmadığı durumda ortaya çıkar. Uygulama Aktivasyonu olarak adlandırılır. İkincisi de

operasyon için gerekli olan nesnenin hazır olmadığı durumda ortaya çıkar. Nesne Aktivasyonu olarak adlandırılır.

Nesne Aktivasyonu, TNA ve uygulamayı içeren program yada programlar arasındaki iletişime ihtiyaç duyar. Bu bölümde 'sunucu' terimi, TNA'nın başlatabileceği, nesne uygulamasını içeren program yada programlar yerine kullanılacaktır.

TNA, aktivasyon işlemini, genellikle işletim sistemine bağımlı bir yolla uygun sunucuyu çalıştırarak başlatır. Uygulama kendini hazırlar ve **impl_is_ready** yada **obj_is_ready** fonksiyonlarını kullanarak TNA'ya istekleri yerine getirmek için hazır olduğunu bildirir.

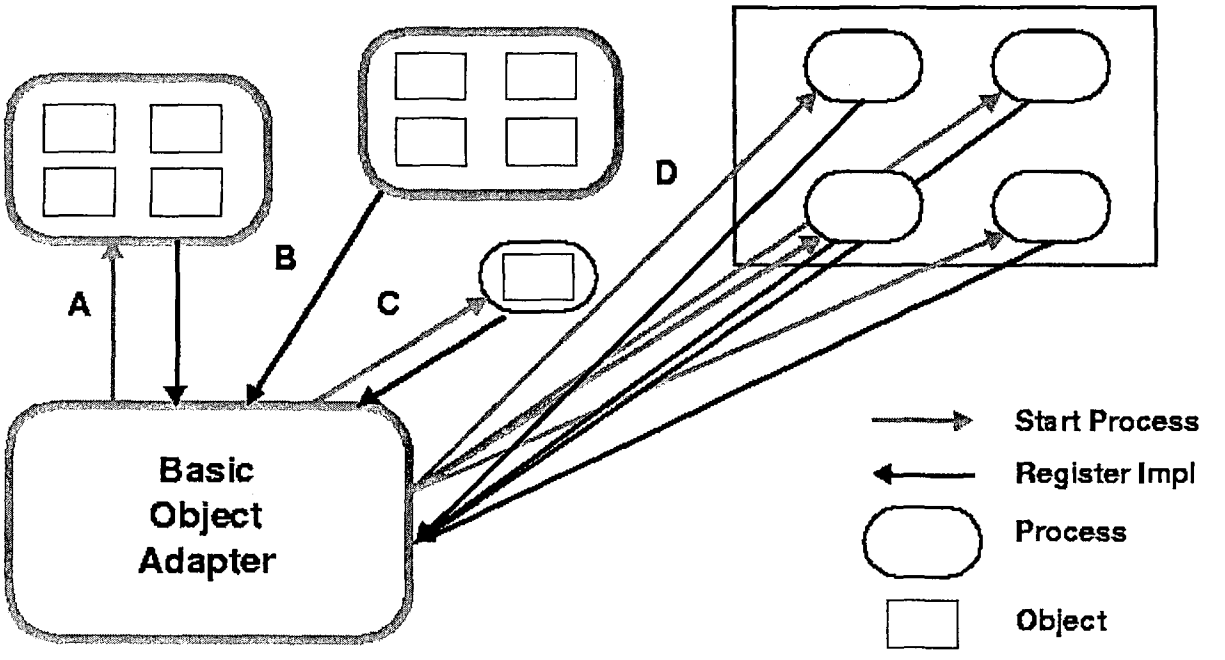
Sunucunun başlatılmasından, uygulamanın hazır olmasına kadar geçen zamanda TNA sunucuya başka isteklerin gelmesini engeller. Bu noktadan sonra, TNA taslaklar yardımı ile uygulamanın metodlarını çağırabilir.

```
void impl_is_ready( in ImplementationDef impl );                    // PIDL
void obj_is_ready( in Object obj, in ImplementationDef impl );
```

Bir *aktivasyon politikası (activation policy)*, birden fazla nesne ve uygulamanın aktif olmadığı durumlarda belirli bir uygulamanın uyması gereken kuralları tanımlar. Uygulama aktivasyonu için TNA uygulamalarının desteklediği dört politika vardır.

- *Ortak Sunucu Politikası (Shared Server)*; bir uygulamanın birden fazla aktif nesnesi aynı sunucuyu paylaşır
- *Ayrı Sunucu Politikası (Unshared Server)*; her bir sunucu için belirli bir anda sadece bir uygulamaya ait bir nesne aktif olabilir.
- *Metod Başına Sunucu Politikası (Server per-method)*; bir metodun her çağrılışında ayrı bir sunucu başlatılır ve metodun çalışması bittiğinde sunucu durdurulur.
- *Kalıcı Sunucu Politikası (Persistent Server)*; Sunucuyu TNA başlatmaz. Sunucu istekleri alabilmek için kendini TNA'ya kaydeder. Bir kalıcı sunucu aynı anda birden fazla aktif nesneye sahip olabilir.

Şekil 2.8'de bu aktivasyon politikaları gösterilmiştir. Durum A, bir ortak sunucuyu gösterir. TNA bir sunucuyu başlatır ve sunucu hazırlığı bitince TNA'ya kendini kaydeder. Durum B bir kalıcı sunucuyu gösterir. Sunucuyu TNA başlatmaz, sunucu kendini TNA'ya kaydeder. Durum C bir ayrı sunucuyu gösterir. TNA, sadece bir nesneyi yönetebilecek bir sunucuyu başlatır. Durum D ise bir metod başına sunucuyu gösterir. Her bir metod çağırımında (istek geldiğinde) TNA yeni bir sunucu başlatır.



Şekil 2.8 Nesne Uygulaması Aktivasyon Politikaları (OMG, 1998).

Ortak Sunucu Politikası

Ortak sunucu politikasında, bir program birden fazla nesneyi yönetebilir. Bu en çok karşılaşılan sunucu tipidir. Sunucu, sunucu içinde bulunan herhangi bir nesnenin metodu çağrılması gerektiğinde başlatılır. Sunucu hazırlığını tamamlar ve **impl_is_ready** fonksiyonunu çağırarak TNA'ya hazır olduğunu bildirir. Ardından TNA, bu sunucu içinde yer alan nesneler için gelen istekleri sunucuya gönderir. Sunucu **deactivate_impl** fonksiyonunu çağırıncaya kadar aktif halde kalır ve istekleri alır. TNA, sunucu aktif ise yeni bir sunucu başlatmayacaktır.

Ayrı Sunucu Politikası

Ayrı sunucu politikasında, her nesne ayrı bir sunucu içinde hizmet verir. Nesne üzerinde bir metod çağrımı isteği geldiğinde sunucu aktif hale getirilir. Sunucu hazırlığını tamamlar ve **obj_is_ready** fonksiyonunu çağırarak TNA'ya hazır olduğunu bildirir. TNA bu nesne için gelen istekleri nesne uygulamasına yönlendirir. Sunucu **deactivate_obj** fonksiyonunu çağırıncaya kadar aktif halde kalır ve istekleri almaya devam eder.

Aktif olmayan bir nesne için bir istek geldiğinde, aynı nesne uygulamasının başka bir nesnesi aktif olsa bile yeni bir sunucu başlatılır.

Metod Başına Sunucu Politikası

Bu politikada, her bir istek için daima yeni bir sunucu başlatılır. Sunucu sadece belirli bir isteğin yerine getirilmesi sırasında çalışır. Aynı anda aynı nesne için hatta aynı nesnenin aynı metodu için aktive edilmiş birden fazla sunucu bulunabilir. Her istek için yeni bir sunucu başlatıldığı için, sunucunun TNA'ya bir nesnenin hazır olduğunu yada deaktive edildiğini bildirmesine gerek yoktur.

Kalıcı Sunucu Politikası

Kalıcı sunucular, TNA tarafından aktive edilmeyen sunuculardır. Bu sunucular genellikle TNA dışında, işletim sistemine özel bir şekilde (bir kullanıcı, bir yönetici tarafından yada bir işletim sistemi servisi olarak sistemin açılışı sırasında otomatik olarak) başlatılır. Sunucu **impl_is_ready** fonksiyonunu çağırarak TNA'ya nesne uygulamalarının hazır olduğunu bildirir. TNA bu sunucuyu bir paylaşılmış sunucu olarak kabul eder ve tüm nesneler ve istekler için aktivasyon mesajlarını bu sunucuya yöneltir.

2.3.3.3 Nesne Referanslarının yaratılması ve yorumlanması

Nesne referansları, nesne uygulamaları talepte bulunduğunda Çekirdek ORB kullanılarak TNA tarafından yaratılır. TNA ve Çekirdek ORB belirli bir nesne referansına bazı bilgiler eşlemektedir. Bir nesne aktif duruma geçirildiğinde bu bilgiler nesne uygulamasına bildirilir. Bu bilgiler, nesne uygulamasının farklı nesne referanslarını birbirinden ayırabilmesine olanak sağlayan tek taşınabilir (portable) yoldur. Aşağıda yeni bir nesne referansı yaratmak için kullanılan TNA operasyonu gösterilmiştir.

```
Object create(
  in ReferenceData id,
  in InterfaceDef intf,
  in ImplementationDef impl
);
```

id parametresi, nesneyi tek olarak tanımlayan, nesne uygulaması tarafından nesne yaratılırken seçilmiş ve nesnenin yaşam süresi (life time) boyunca değişmeyecek olan bir bilgidir. **intf** parametresi, nesne tarafından uygulanmış arayüzler kümesini belirten Arayüz Deposu nesnesidir. **impl** parametresi ise nesne için kullanılacak uygulamayı belirleyen Uygulama Deposu Nesnesi'dir. **Object** değeri **create** operasyonu tarafından üretilen nesne referansını gösterir.

Nesne referansı farklı ORB'ler tarafından farklı şekilde ele alınabilir ancak **id** değeri tüm ORB'lerde mevcut olmalıdır. Genellikle bu **id** değerini nesne uygulaması belirlediği için

sadece nesne uygulaması yorumlayabilir. Bir nesne referansının **id** değerini almak için kullanılan TNA operasyonu **get_id** dir.

```
ReferenceData get_id( in Object obj );
```

Bir nesne referansı ile ilişkilendirilmiş uygulamayı değiştirmek mümkündür. Bu durumda, yeni istekler yeni uygulamaya yönlendirilecektir. Uygulamanın değiştirilmesini sağlayan TNA operasyonu aşağıda gösterilmiştir.

```
void change_implementation( in Object obj,  in ImplementationDef impl );
```

Bir uygulama, bir nesnenin işi bitince onu öldürebilir. Bir nesne öldürüldükten sonra, TNA ve Çekirdek ORB o nesne hiç yaratılmamış gibi davranır. Bu nedenle herhangi bir nesne öldürüldükten sonra bu nesneyi gösteren nesne referansları kullanılarak yapılan istekler hata ile sonuçlanır. Nesneleri öldürmek için aşağıdaki TNA operasyonu kullanılır.

```
void dispose( in Object obj );
```

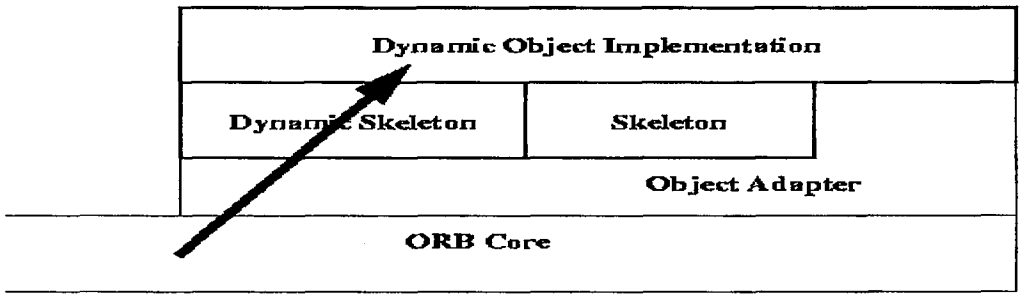
2.4 Dinamik Taslak Arayüzü

DTA (Dynamic Skeleton Interface), nesne metod çağırımlarının dinamik olarak ele alınmasına izin verir.Yani belirli bir operasyona bir taslak üzerinden ulaşmak yerine, DTA üzerinden ulaşılır. Nesne uygulaması, bu arayüzü kullanarak, yerine getirilmesi gereken operasyonun adı ve parametreleri gibi bilgilere ulaşabilir.

DTA, uygulamanın kullanıma sunduğu, derleme zamanında (compile-time) yapısı bilinmeyen bir nesne üzerinde, istemcilerin yaptığı isteklerin ORB tarafından uygulamaya gönderilmesine olanak sağlayan bir arayüzdür. Bu yapı, tipe özel, statik OMG IDL tabanlı taslaklara tamamen zıt olup, aynı mantıksal rolü üstlenir.

DTA, istemci tarafındaki DÇA'nın sunucu tarafındaki benzeridir. Bir nesne uygulaması, istemcinin isteği stub'lar yoluyla mı DÇA yoluyla mı yaptığını ayırdedemediği gibi, istemci de ORB'nin uygulama ile iletişim kurmak için taslakları mı DTA'yı mı kullandığını ayırdedemez.

DTA'nın temel prensibi, ORB'nin belirli bir nesne üzerindeki tüm istekleri, nesne uygulaması üzerinde tanımlanmış, Dinamik Uygulama Rutini-DUR (Dynamic Implementation Routine-DIR) olarak adlandırılan bir rutine yönlendirmesidir. Bir DUR'nin yapısı ve parametreleri belirli olduğu ve genel amaçlı olduğu için, değişik arayüze sahip birden fazla nesne için bir uygulama olarak kullanılabilir.



Şekil 2.9 Dinamik Taslak Arayüzü üzerinden istek alan bir Nesne Uygulaması (OMG, 1998).

ORB, DUR'ne, üzerinde operasyon gerçekleştirilecek nesneyi, operasyonun adını ve operasyona ait parametreleri geçirir. DUR, ilgili nesneyi, nesne adaptörünü ve Arayüz Deposu'nu kullanarak ilgili operasyon hakkında daha fazla bilgi alabilir.

DUR, nesne uygulaması aktif hale geçtiğinde, nesne uygulaması tarafından ORB'nin istekleri yönlendirebilmesi için ORB'ye kaydedilir. Bu işlem, hangi nesne için bu rutinin çağırılması gerektiğini belirten parametreleri içerir. DTA tamamen TNA tarafından sağlandığı için bu kayıt işlemi TNA uygulamalarına göre değişebilir niteliktedir.

DUR'a geçirilen **ServerRequest** tipindeki parametre, yerine getirilecek operasyon hakkında gerekli bilgileri içerir.

```
module CORBA {
  interface ServerRequest {
    readonly attribute Identifier operation;
    void arguments( inout NVList nv );
    void set_result( in Any val );
    void set_exception( in Any val );
  };
};
```

operation değeri operasyonun adını içerir. **arguments** operasyonu çağırılarak istek parametreleri **nv** adındaki listeye aktarılabilir. Bu listeye aktarılan bilgiler kullanılarak operasyon yerine getirildikten sonra uygulama **set_result** operasyonunu kullanarak operasyonun dönüş değerini belirtmelidir. Operasyonun çalıştırılması sırasında herhangi bir hata olması durumunda uygulama **set_exception** operasyonunu kullanarak hata ile ilgili bilgileri belirtmelidir.

2.5 Dinamik Çağırım Arayüzü

DÇA (Dynamic Invocation Interface), nesneler üzerinde talep edilen isteklerin dinamik olarak yaratılmasına ve gerçekleştirilmesine olanak sağlar. Bir istemci, bir nesneye bir isteği göndermek için bu arayüzü yada istemci stub'larını kullanabilir. Bu iki yol tamamen aynı

işleve sahiptir. DÇA'nın avantajı, istemci programın derlenmesi sırasında yapısı belli olmayan nesneler üzerinde istekte bulunmaya olanak sağlamasıdır.

Bir istek, bir nesne referansı, bir operasyon ve bir parametre listesinden oluşur. DÇA'da bir operasyonun parametreleri, bir parametre listesi şeklinde gösterilir. Bu parametre listelerinin her elemanı, bir **NamedValue** tipinin bir örneğidir.

2.5.1 Genel Veri Yapıları

NamedValue tipi OMG IDL de sık rastlanan bir veri tipidir. Kullanıcı tarafından tanımlanmış herhangi bir nesnenin bir metodunun bir parametre tipi olarak yada DÇA ile yapılan isteklerde parametre listesini belirtmek amacıyla kullanılabilir. **NVList**, parametre listeleri oluşturmak için kullanılan bir yalancı nesnedir. Bu tiple OMG IDL ve C programlama dilinde aşağıdaki gibi tanımlanmışlardır.

```
typedef unsigned long Flags;
struct NamedValue {
  Identifier name;
  Any        argument;
  long       len;
  Flags      flags;
};

CORBA_NamedValue *CORBA_NVList;
```

NamedValue ve **Flags** CORBA Modülünde tanımlanmıştır.

NamedValue ve **NVList** yapıları DÇA'da dinamik olarak yaratılan isteklere geçirilen parametreleri belirtmek için kullanılır. **NVList** tipi yukarıda genel olarak tanımlanmasına rağmen ORB bağımlı bir yapıya sahiptir. Bu nedenle sadece ORB Arayüzündeki **create_list** operasyonu kullanılarak yaratılabilir.

Bir **NamedValue** yapısı argümanın adı, argümanın verisi, argümanın uzunluğu ve argüman modunu içeren bayraklar kümesinden oluşur.

arg_modes alanı aşağıdaki bayrak (flag) değerlerini alabilecek bir bit maskesi (bit-mask) olarak tanımlanmıştır.

CORBA::ARG_IN	İlgili parametre değeri sadece girdi parametredir.
CORBA::ARG_OUT	İlgili parametre değeri sadece çıktı parametredir.
CORBA::ARG_INOUT	İlgili parametre değeri girdi-çıktı parametredir.

Bu bayrak değerleri verinin aktarılış yönünü gösterir.

2.5.2 Dönüş durumları ve Hatalar

DÇA'da bir çok rutinin dönüş değeri **Status** tipindedir. **Status** tipi durum kodlarını içerir ve CORBA Modülünde aşağıdaki gibi tanımlanmıştır.

```
typedef unsigned long Status;
```

2.5.3 İstek operasyonları

İstek operasyonları **Request** yalancı nesnesinin arayüzü ile tanımlanmıştır.

```
module CORBA {
  interface Request {
    Status add_arg(
      in Identifier   name,
      in TypeCode     arg_type,
      in void         *value,
      in long         len,
      in Flags        arg_flags
    );
    Status invoke( in Flags invoke_flags);
    Status delete();
    Status send( in Flags invoke_flags);
    Status get_response( in Flags invoke_flags);
  };
};
```

2.5.3.1 create_request

Bu operasyon bir yalancı nesne yarattığı için **Object** arayüzünde tanımlanmıştır. **create_request** operasyonu dinamik olarak istek yapılacak nesnenin nesne referansı üzerinde çağırılır.

```
Status create_request(
  in Context       ctx,
  in Identifier    operation,
  in NVList        arg_list,
  inout NamedValue result,
  our Request      request,
  in Flags         req_flags
);
```

Bu operasyon bir ORB isteği yaratır. Bu istek **invoke** yada **send/get_response** operasyonları çağırılarak yerine getirilir.

operation parametresi yerine getirilmesi istenen operasyonun adını belirtir. **arg_list** parametresi istek parametrelerini içeren bir listedir. Eğer **arg_list** parametresi verilmezse yaratılan **Request** nesnesinin **add_arg** metodu kullanılarak parametreler belirtilebilir. İstek parametrelerinin hepsi ya **arg_list** argümanı ile yada **add_arg** metodu ile verilmelidir. Bir kısmı **arg_list** argümanıyla bazıları **add_arg** metoduyla belirtilemez.

Argüman listesi belirtilirken her bir argüman için **value** ve **len** parametreleri belirtilmelidir. **value** parametresi argümanın değerini, **len** parametresi de argümanın veri uzunluğunu gösterir.

İsteğin sonucu, istek yerine getirildikten sonra **result** argümanına aktarılır.

2.5.3.2 add_arg

```
Status add_arg(
  in Identifier name,
  in TypeCode   arg_type
  in void      *value,
  in long       len,
  in Flags      arg_flags
);
```

add_arg metodu her çağrılışında isteğe bir argüman ekler.

Her bir argüman için en azından **value** ve **len** değerleri belirtilmelidir.

İsteğe eklenen argümanlar istekle ilişkilendirilirler. Bu nedenle isteği yaratan programın, istek yerine getirilinceye kadar bu argümanların değerlerinin değiştirilmemesi gerekir.

2.5.3.3 invoke

```
Status invoke( in Flags invoke_flags);
```

Bu metod ORB'yi kullanarak isteğin gösterdiği operasyonun bulunması ve yerine getirilmesi için kullanılır. Eğer operasyonu başarılı bir şekilde çalışırsa, operasyonun dönüş değeri **create_request** metoduna geçirilen **result** argümanına aktarılır. Bu metod ilgili operasyonun işi bitinceye kadar dönmez.

2.5.3.4 delete

```
Status delete();
```

Bu metod isteği öldürür. İstek için ayrılmış hafıza serbest bırakılır. Bir **Request** nesnesi, bu metod çağrılarak öldürüldükten sonra bu nesne üzerinde herhangi bir işlemin yapılmaması gerekir.

2.5.4 Ertelenmiş istek operasyonları

2.5.4.1 send

```
Status send( in Flags invoke_flags);
```

send metodu ilgili operasyonun çalıştırılması işlemini başlatır. Ama **invoke** metodunda olduğu gibi operasyonun işinin bitmesini beklemez, işlemi başlatıp hemen döner. Operasyonun sonucunu öğrenmek için **get_response** metodu kullanılır. Operasyonun dönüş değeri ve çıktı parametreleri operasyonun işi bitinceye kadar kullanılmamalıdır.

send metodu nesne referansının geçerli bir nesne referansı olmaması gibi durumlarda hata

dönebilir. Bunun dışındaki hatalar **get_response** metodu çağırıldığında dönülür.

Eğer operasyon **oneway** olarak tanımlanmış yada **INV_NO_RESPONSE** bayrağı **send** metoduna parametre olarak geçirilmişse **get_response** metodunun çağırılmasına gerek yoktur.

2.5.4.2 get_response

```
Status get_response( in Flags response_flags);
```

get_response metodu operasyonun bitip bitmediğini kontrol etmek amacıyla kullanılır. Eğer **get_response** metodu operasyonun bittiğini gösterirse, çıktı parametreleri ve operasyonun dönüş değeri geçerli bilgileri içerir.

Eğer **RESP_NO_WAIT** bayrağı metoda parametre olarak geçirilmişse **get_response** metodu operasyonun işi bitmese bile hemen döner. Aksi durumda **get_response** metodu operasyon bittikten sonra döner.

2.6 ORB Arayüzü

ORB Arayüzü, kullanılan nesne adaptörüne bağlı olmayan ORB operasyonlarına ulaşmayı sağlar. Bu operasyonlar tüm ORB'ler için aynıdır ve nesnelerin istemcileri yada nesne uygulamaları tarafından kullanılabilirler. Bu operasyonların bazıları ORB, diğerleri de nesne referansları ile ilgili işlemler için kullanılırlar.

```
module CORBA {
  typedef unsigned short ServiceType;
  typedef unsigned long  ServiceOption;
  typedef unsigned long  ServiceDetailType;

  const ServiceTypeSecurity = 1;

  struct ServiceDetail {
    ServiceDetailType service_detail_type;
    sequence< octet > service_detail;
  };

  struct ServiceInformation {
  sequence<ServiceOption> service_options;
  sequence<ServiceDetail> service_details;
  };

interface ORB {
  string object_to_string( in Object obj);
  Object string_to_object( in string str);
  Status create_list( in long count, out NVList new_list);
  Boolean get_service_information(in  ServiceType        service_type,
                                  out ServiceInformation service_information);
 };
};
```

2.6.1 Nesnelerin karakter dizilerine çevirilmesi

Bir nesne referansı farklı ORB'lerde farklı şekilde gösterilebileceği için nesne referanslarının bir kalıcı saklama ortamına aktarılması güvenilir bir yöntem değildir. Burada iki problem

ortaya çıkar. Birincisi, bir istemcinin bir nesne referansını bir saklama ortamına aktarılabilecek bir değere çevirmesi işlemi, ikincisi de bu değerin daha sonra uygun nesne referansına dönüştürülebilmesi işlemidir.

Bir nesne referansı **object_to_string** operasyonu ile bir karakter dizisine çevrilebilir. Bu karakter dizisi saklanmak yoluyla yada başka bir yolla bir istemciye iletilmek için kullanılabilir. Daha sonra, **object_to_string** operasyonunun ürettiği bu değer **string_to_object** operasyonuna geçirilerek uygun nesne referansına dönüştürülebilir.

Bu çevirme işlemi tüm ORB'ler tarafından anlaşılabilir bir yapıda olmalıdır. Bir ORB'de bir nesne referansı için **object_to_string** operasyonu ile üretilen bu karakter dizisi, başka bir ORB'de **string_to_object** operasyonuna geçirildiğinde aynı nesneyi gösteren bir nesne referansına dönüştürülebilmelidir.

2.6.2 Servis bilgisinin alınması

```
boolean get_service_information(
        in ServiceType          service_type,
        out ServiceInformation service_information
);
```

get_service_information operasyonu kullanılan ORB'nin desteklediği servisler hakkında bilgi almak için kullanılır. Operasyonu geçirilen **servis_type** parametresi bilgi alınacak servisi tanımlar. Bu parametrenin alabileceği değerler CORBA modülünde tanımlanmıştır. Eğer istenen servis kullanılan ORB tarafından destekleniyorsa operasyon DOĞRU (TRUE) döner ve servisin bilgileri **service_information** çıktı parametresine aktarılır. Eğer servis desteklenmiyorsa operasyon YANLIŞ (FALSE) döner.

2.6.3 Nesne Referansı operasyonları

ORB tarafından yerine getirilen, nesne referansları üzerinde uygulanabilecek operasyonlar OMG IDL **Object** arayüzü kullanılarak ulaşılabilir. Bu operasyonlar aşağıda gösterilmiştir.

```
module CORBA {
  interface Object {
    ImplementationDef get_implementation();
    InterfaceDef      get_interface();

    boolean           is_nil();
    Object            duplicate();
    void              release();

    boolean           is_a( in string logical_type_id);
    boolean           is_equivalent( in Object other_object);

    Status create_request (
      in Context       ctx,       in Identifer     operation,
      in NVList        arg_list,  inout NamedValue result,
      out Request      request,   in Flags         req_flags);
  };
};
```

2.6.3.1 Nesne arayüzünün belirlenmesi

Nesne referansı üzerindeki **get_interface** operasyonu, nesnenin arayüzünü tanımlayan bir Arayüz Deposu nesnesi, **get_implementation** operasyonu da nesnenin ait olduğu nesne uygulamasını tanımlayan bir Uygulama Deposu nesnesi döner.

```
InterfaceDef       get_interface();
ImplementationDef  get_implementation();
```

2.6.3.2 Nesne Referanslarının kopyalanması ve kopyaların silinmesi

Nesne Referansları'nın yapısı ORB bağımlı olduğu için, istemcilerin ve uygulamaların nesne referansları için hafıza ayırmaları mümkün değildir. Bu nedenle bir nesne referansının kopyalanması ve silinmesi için operasyonlar tanımlanmıştır.

```
Object duplicate();
void   release();
```

Bir nesne referansının kopyası gerektiğinde istemci **duplicate** operasyonunu kullanarak yeni bir kopya oluşturabilir. Bir nesne referansının kopyası orjinal nesne referansının gösterdiği nesne uygulamasını gösterir, yeni bir nesne uygulaması yaratılmaz. Hangi kopya kullanılırsa kullanılsın talep edilen istekler aynı nesne uygulaması üzerinde yerine getirilir.

Bir nesne referansının kopyasına ihtiyaç kalmadığında **release** operasyonu kullanılarak kopya nesne referansı hafızadan silinebilir. Bu operasyon nesne uygulamasını yada diğer nesne referanslarını etkilemez.

2.6.3.3 Boş Nesne Referansları

Değeri **OBJECT_NIL** olan bir nesne referansı hiç bir nesneyi göstermez. **is_nil** operasyonu kullanılarak bir nesne referansının bu değeri taşıyıp taşımadığı kontrol edilebilir. Nesne uygulaması bu operasyon tarafından dikkate alınmaz.

```
boolean is_nil();
```

2.6.3.4 Eşitlik testi operasyonu

Bir nesne referansının gösterdiği nesnenin istenen tipte olup olmadığını kontrol etmek için **is_a** operasyonu kullanılır.

```
boolean is_a ( in RepositoryID logical_type_id );
```

logical_type_id parametresi nesne tipini tanımlayan bir karakter dizisidir.

2.6.4 ORB ve NA'nın başlatılması ve İlk Referanslar

Bir programın CORBA ile çalışmaya başlayabilmesi için önce aşağıdaki işlemleri yerine getirmesi gerekir.

- ORB ve nesne adaptörünün başlatılması
- ORB ve NA operasyonlarına ulaşabilmek için ORB ve NA yalancı-nesnelerinin (pseudo-object) referansların alınması.

CORBA, uygulamalarının bu nesne referanslarını alabilmesi için gerekli, yalancı IDL ile tanımlanmış operasyonlar sunar.

- ORB'ye ulaşımı sağlayan operasyonlar. Bu operasyonlar CORBA modülünde yer alır, ORB Arayüzünde yer almaz.
- TNA'ya ve diğer nesne adaptörlerine ulaşımı sağlayan operasyonlar. Bu operasyonlar ORB Arayüzünde yer alır.
- Arayüz Deposuna, İsimlendirme Servislerine ve diğer servislere ulaşımı sağlayan operasyonlar. Bu operasyonlar ORB Arayüzünde yer alır.

2.6.4.1 ORB'nin başlatılması

Bir CORBA programı, CORBA ortamı ile çalışabilmek için ORB ve NA yalancı nesne referanslarını almak zorundadır. Bunun için önce ORB ve NA'yı başlatmalı ve sonra ORB ve NA için yalancı nesne referanslarını almalıdır. ORB ve NA'nın başlatılma sırası önemlidir. ORB'nin NA'dan önce başlatılması gerekmektedir.

ORB'yi başlatmak için gerekli operasyon bir nesne üzerinde uygulanamaz. Çünkü program ORB'yi başlatmadan önce herhangi bir CORBA nesnesine sahip değildir. Bu nedenle ORB'nin başlatılması için gereken **ORB_init** operasyonu herhangi bir nesnenin arayüzünde değil her programa açık olan CORBA modülünde tanımlanmıştır.

Bir program ORB yalancı nesnesinin referansını almak için **ORB_init** operasyonunu çağırır.

```
module CORBA {
  typedef string ORBid;
  typedef sequence<string> arg_list;
  ORB ORB_init( inout arg_list, in ORBid orb_identifier);
};
```

ORB_init operasyonu, **ORB** tipinde ORB Arayüzünü gösteren yalancı bir nesne referansı döner. **ORB_init** operasyonu her bir programda bir kere çalıştırılır. Birden fazla çalıştırılması durumunda her seferinde aynı yalancı nesne referansını dönmesi gerekir.

Hem istemci hem de sunucu programlar ORB operasyonlarına ulaşabilmek için ORB'yi başlatmalıdırlar.

2.6.4.2 NA ve TNA'nın başlatılması

Bir programın Nesne Adaptörü operasyonlarına ulaşabilmesi için NA'yı başlatması gerekmektedir. NA sadece sunucu programlar tarafından başlatılmalıdır. Çünkü istemci programlar herhangi bir nesneyi kullanıma sunmazlar. Bunun için istemci programların NA'yı başlatmasına gerek duyulmaz.

NA'nın başlatılması ve NA yalancı nesne referansı için gerekli operasyonu ORB Arayüzünün bir parçası olan **<OA>_init** operasyonudur. Örneğin CORBA 2.2 versiyonundan önceki ORB uygulamalarında kullanılan TNA' nın başlatılması için **BOA_init** operasyonu mevcuttur.

```
module CORBA {
  interface ORB {
    typedef string BOAid;
    typedef sequence<string> arg_list;
    BOA BOA_init( inout arg_list,  in BOAid boa_identifier);
  };
};
```

BOA_init fonksiyonu **BOA** tipinde Temel Nesne Adaptörü arayüzünü gösteren yalancı bir nesne referansı döner.

2.6.4.3 İlk Referansların alınması

Arayüz Deposu, İsimlendirme Servisi gibi servislere ihtiyaç duyan programlar bu servislerin arayüzlerini gösteren bir referans almak zorundadır. Bu referansların alınması taşınabilir bir yapıda olmalıdır. Yani her ORB'de aynı şekilde alınmalıdır. Bu nedenle, bu referanslar ORB Arayüzünde tanımlanmış operasyonlar kullanılarak alınır.

```
module CORBA {
  interface ORB {
    typedef string ObjectId;
    typedef sequence<ObjectId> ObjectIdList;
    exception InvalidName {};
    ObjectIdList list_initial_services();
    Object resolve_initial_references(in ObjectId identifier)
           raises (InvalidaName);
  };
};
```

resolve_initial_references operasyonu istenen servis için bir nesne referansı döner. **ObjectId** tipi bir karakter dizisi olup hangi servisin nesne referansının alınacağını belirtir. Bu yöntemle alınacak referanslar sadece belli başlı servislere aittir. Bir kullanıcı tarafından yaratılan bir nesneyi gösteren bir nesne referansı almak için bu operasyon kullanılmamalıdır. **OMG** tarafından **resolve_initial_references** operasyonu ile ilk referansı alınabilecek servisler Arayüz Deposu ve İsimlendirme Servisi olarak sınırlandırılmıştır. Bu servisler için geçerli **ObjectId** değerleri sırasıyla **InterfaceRepository** ve **NameService** 'dir.

İlk referansları alınabilecek servislerin listesini almak için **list_initial_services** operasyonu kullanılır. **list_initial_services** operasyonu **ObjectIdList** tipinde bir değer döner. **ObjectIdList** tipi **ObjectId** tipinin bir dizisidir.

resolve_initial_references operasyonu genel tipte bir nesne referansı (**Object**) döner. Yani bu operasyonun döndüğü nesne referansı herhangi bir nesne tipini gösterebilir. Bir programın, **resolve_initial_references** operasyonunun gerçekten istediği servisi gösteren bir nesne referansını dönüp dönmediğini kontrol etmesi gerekir.

3. OMG ARAYÜZ TANIMLAMA DİLİ

OMG Arayüz Tanımlama Dili (OMG IDL), nesne arayüzlerini nesne uygulamalarından ayırmaya yarayan, CORBA mimarisinin temel soyutlama mekanizmasıdır. OMG IDL, istemci ve sunucu arasında, kullanılacak veri tipleri ve nesne arayüzlerini tanımlayan bir sözleşme ortaya koyar. Bu tanım nesne uygulamasının yazıldığı programlama dilinden bağımsızdır.

IDL kullanılarak yazılan arayüzler '**.IDL**' uzantılı dosyalara kaydedilirler. Bu dosyalar IDL Dosyaları (IDL Files) olarak adlandırılır. IDL Dosyaları IDL Derleyicisi ile derlenerek, tanımlanan arayüzler istenen programlama diline dönüştürülür.

Bu bölümde Arayüz Tanımlama Dili'ni detaylı olarak inceleyeceğiz.

3.1 Yazım Kuralları

3.1.1 Açıklamalar (Comments)

IDL, /* ve */ işaretleri ile yada // işaretleri ile gösterilen açıklamaların IDL dosyasına yazılmasına izin verir. /* ve */ işaretleri birden fazla satıra taşan açıklamalar için, // işaretleri ise sadece bulundukları satırın sonuna kadar olan açıklamalar için kullanılırlar.

```
/*
* This is legal IDL Comment
*/
// This IDL comment is up to end of this line
```

3.1.2 Anahtar Kelimeler (Keywords)

IDL için özel anlamı olan bazı anahtar kelimeler vardır. Bu kelimeler kullanıcı tarafından tanımlanan değişken, operasyon yada veri tipi adı olarak kullanılamazlar. Bu anahtar kelimeler sadece IDL tarafından tanımlandığı şekilde ve uygun yerlerde kullanılmalıdır. IDL anahtar kelimeleri aşağıdaki tabloda verilmiştir.

Çizelge 3.1 OMG IDL Anahtar Kelimeleri (OMG, 1998).

any	double	interface	readonly	unsigned
attribute	enum	long	sequence	union
boolean	exception	module	short	void
case	FALSE	object	string	wchar
char	fixed	octet	struct	wstring
const	float	oneway	switch	
context	in	out	TRUE	
default	inout	raises	typedef	

Bu anahtar kelimeler mutlaka yukarıda gösterildiği gibi yazılmalıdır. Örneğin **string** anahtar

kelimesi **String** şeklinde yazılırsa IDL derleyicisi hata verecektir.

3.1.3 Tanımlayıcılar (Identifiers)

Tanımlayıcılar IDL ile tanımlanmış herhangi bir birimi adlandırmak için kullanılırlar. Örneğin bir değişkenin adı, bir arayüzün adı, yeni tanımlanan bir veri tipinin adı bir tanımlayıcıdır. Tanımlayıcılar bir alfabetik karakterle başlayıp, alfanümerik karakterler ve alt tire ile devam eden karakter dizileridir.

```
long Status;

interface FileServer
{
      ...
};
```

Yukarıdaki örnekte **Status** ve **FileServer** kelimeleri bir tanımlayıcıdır.

Tanımlayıcılar belirtilirken kullanılan karakterlerin küçük yada büyük harf olması önemsizdir. (case-insensitive). Örneğin **Dosya** ve **DOSYA** tanımlayıcıları IDL tarafından eşit kabul edilirler.

3.2 Temel IDL Veri Tipleri

IDL birkaç temel veri tipini destekler. Bu veri tipleri aşağıdaki tabloda gösterilmiştir.

Çizelge 3.2 IDL veri tipleri ve alabilecekleri değerler (OMG, 1998).

Tip	Alabileceği değer	Uzunluğu
short	[-215 , 215 - 1]	>= 16 bit
long	[-231 , 231 - 1]	>= 32 bit
unsigned short	[0 , 216 - 1]	>= 16 bit
unsigned long	[0 , 232 - 1]	>= 32 bit
float	Tek yoğunluklu	>= 32 bit
double	Çift yoğunluklu	>= 64 bit
char	[0, 255]	>= 8 bit
string	[1, 255]	Değişken zunlukta
boolean	TRUE yada FALSE	Belirtilmemiştir
octet	[0, 255]	>= 8 bit
any	çalışma anında belirlenen bir tipi içerir	

CORBA standardı, dil dönüşümlerinin yukarıda belirtilen veri tiplerinin, yukarıda belirtilen tip uzunluklarına sadık kalmasını zorunlu kılmıştır.

CORBA standardı, veri tipi uzunlukları için yukarıda görüldüğü gibi sadece alt sınırı belirlemiştir. Bunun nedeni bazı programlama dillerinin yada CPU mimarilerinin veri tiplerini

değişik yapılarda göstermesidir. Örneğin bir **char** tipinin en az 8 bit uzunluğunda olması gerekmektedir. Bazı CPU' larda **char** tipi, CPU mimarisinin izin verdiği en küçük veri uzunluğu olan 16 bitlik bir veri tipine dönüştürülebilir.

3.3 Kullanıcı Tarafından Tanımlanmış Tipler

IDL, önceden tanımlanmış temel veri tiplerinin yanında kullanıcı tarafından tanımlanmış kompleks veri tiplerinin de kullanılmasına izin verir.

3.3.1 İsimlendirilmiş Tipler

typedef anahtar kelimesi kullanılarak daha önceden tanımlanmış bir tipe yeni bir isim verilebilir. **typedef** anahtar kelimesi yeni bir tip oluşturmaz. Yeni isim verilen tip bir IDL temel veri tipi yada kullanıcı tarafından tanımlanmış bir tip olabilir.

```
typedef unsigned long Status;
typedef short         Year;
```

Örnekte **short** veri tipine **Year** adı verilmiştir.

3.3.2 Listeler (Enumerations)

IDL sadece belirtilen değerleri alabilen yeni bir veri tipi tanımlanmasına izin verir.

```
enum Color { red, green, blue, black, white, orange, yellow };
```

Yukarıdaki tanımla sadece listede belirtilen değerleri alabilen **Color** adında bir veri tipi tanımlanmıştır. Bu tipler en az 32 bit uzunluğundaki veri tiplerine dönüştürülürler.

Bir liste tanımında yer alan herhangi bir değer başka bir liste tanımında yer alamaz. Aşağıdaki örnek bir derleyici hatası üretecektir.

```
enum BasicColor { red, green, blue };
enum LightColor { black, white, red };
```

Bu örnekte **red** değeri iki ayrı liste tanımında yer aldığı için IDL derleyicisi bu tanıma hata verecektir.

3.3.3 Kayıtlar (Structures)

IDL, kullanıcının çeşitli tipte elemanlar içeren yeni kayıt yapıları oluşturabilmesine izin verir. Bu yapılara **kayıt** denir. Kayıtların elemanları isimleri ile ayırt edilir. Kullanıcı, kayıt tanımı ile belirli bir bilgiyi gösterecek yeni bir veri tipi yaratır. Kayıtlar **structure** anahtar kelimesi ile tanımlanır.

```
structure FileInfo {
  long   size;
  string name;
};
```

Örnekte **FileInfo** adında bir dosyaya ait bilgileri tutmak için bir kayıt tanımlanmıştır. Bu kayıtın elemanları **long** ve **string** tipindedir. **FileInfo** tipindeki herhangi bir değişken, **size** adında **short** tipli bir değişken ve **name** adında **string** tipli bir değişken içerecektir.

3.3.4 Bileşimler (Unions)

Bileşimler, kayıtlar gibi değişik tipte elemanlar içeren yeni bir tip tanımlarlar. Ancak kayıt tipinde bir değişken her zaman elemanlarının toplam uzunluğu kadar uzunluğa sahiptir ve her zaman tüm elemanları geçerli bilgiye sahiptir. Bileşimler ise belirli bir anda elemanlarından sadece birinin geçerli bilgi taşıdığı bir yapı oluştururlar. Dolayısıyla bileşimlerin toplam uzunluğu, uzunluğu en fazla olan elemanının uzunluğu kadardır.

Bileşimler, elemanlarından hangisinin geçerli bilgiye sahip olduğunu tanımlayan bir ayırıcı (discriminator) içerirler. Bileşim ayırıcıları ancak **char**, tamsayı tipler, **boolean** yada liste tipinde olabilir.

Bileşimler **union** anahtar kelimesi ile tanımlanırlar.

```
enum InfoKind { text, numeric, none };
union Info switch (InfoKind) {
  case text:
    string description;
  case numeric:
    long   index;
};
```

Örnekte **Info** adında bir bileşim tanımlanmıştır. Bileşimin ayırıcısı **InfoKind** adındaki bir liste (enumeration) tipidir. **Info** tipindeki herhangi bir değişkenin ayırıcısının değeri **text** ise **description** adındaki elemanı, **numeric** ise **index** adındaki elemanı geçerli bilgiyi taşımaktadır.

3.3.5 Sabit Uzunluktaki Diziler (Arrays)

IDL, belirli bir tipin dizilerinin tanımlanmasına izin verir. Bu diziler bir yada çok boyutlu olabilir. Bu diziler sabit uzunluktadır. Yani dizi tanımlandığı sırada dizinin kaç elemana sahip olduğu bellidir. Dizi tanımlarının **typedef** anahtar kelimesi ile başlaması zorunludur.

```
typedef Color ColorArray[10];
typedef string Names[10][20];
```

Örnekte **ColorArray** adında, **Color** tipindeki elemanları içeren ve uzunluğu 10 olan bir dizi tanımlanmıştır. **Names** adındaki dizi **string** tipli elemanlar içermekte olup iki boyutlu bir

dizidir.

3.3.6 Değişken Uzunluktaki Diziler (Sequences)

IDL, eleman sayısı dizinin tanımı sırasında belirli olmayan dizilerin tanımlanmasına izin verir. Değişken uzunluktaki diziler sınırlı yada sınırsız olarak tanımlanabilirler. Değişken uzunluktaki diziler **sequence** anahtar kelimesi ile tanımlanırlar.

```
typedef sequence < Color > Colors;          // sınırsız dizi
typedef sequence < long, 100 > Numbers;     // en fazla 100 eleman
```

Sınırı belirlenmemiş dizilerin boyu eleman eklendikçe dinamik olarak büyüyebilir. Eleman sayısı ancak programa ait hafıza miktarı ile sınırlıdır. Sınırlı diziler ise en fazla belirtilen sayı kadar eleman içerebilirler. Değişken uzunluktaki bir dizi herhangi bir anda boş olabilir.

Dizi elemanları başka diziler olan diziler tanımlanabilir. Bu şekilde eleman sayısı sabit olmayan çok boyutlu diziler tanımlanabilir.

```
typedef sequence < Color , 100 > Colors;
typedef sequence < Colors, 100 > ColorList;    // en fazla 100 eleman
```

3.4 Arayüzler ve Operasyonlar

IDL, arayüzleri ve arayüz operasyonlarını tamımlamak için kullanılır. Arayüzler **inteface** anahtar kelimesi kullanılarak tanımlanır.

```
interface Thermostat {
            // sıcaklığı oku
  short get_temp ( );
            // maksimum sıcaklığı değiştir, bir önceki maksimum sıcaklığı döndür
  short set_max_temp ( in short new_temp );
};
```

Yukarıdaki tanım **Thermostat** adında yeni bir CORBA arayüz tipi tanımlar. Arayüz iki operasyon sunar : **get_temp** ve **set_max_temp**. Herhangi bir istemci bu arayüze arayüzdeki operasyonları kullanarak ulaşabilir. Bir istemci, sıcaklığı öğrenmek için **get_temp**, termostatın maksimum sıcaklığını değiştirmek için de **set_max_temp** operasyonunu kullanmalıdır.

Bir istemcinin bir nesne uygulaması ile iletişim sağlayabilmesi için tek yol ilgili arayüzün operasyonlarını kullanmaktır.

Her CORBA nesnesi bir tek arayüze sahiptir. Fakat aynı anda aynı arayüze sahip birden fazla nesne bulunabilir.

IDL ile tanımlanan arayüzler yeni bir tip tanımlar. Tanımlanan arayüzler bir operasyona parametre olarak geçirilebilirler.

3.4.1 Operasyon tanımları

Bir operasyon tanımı sadece bir arayüz tanımı içerisinde yer alabilir. Bir operayon tanımı

- bir dönüş değeri tipi
- bir operasyon adı
- sıfır yada daha fazla parametre tanımı

içermelidir.

Aşağıda en basit şekliyle bir operasyon tanımlanmıştır.

```
interface A {
      void op( );
};
```

op operasyonu herhangi bir parametre almamakta ve bir dönüş değeri dönmemektedir. Bir operasyon dönüş değerine sahip değilse **void** anahtar kelimesinin belirtilmesi zorunludur. Aşağıdaki gibi dönüş değeri tipi boş bırakılamaz.

```
interface A {
 op( );    // Hata. Dönüş değeri tipi belirtilmemiş.
};
```

Aşağıda başka bir arayüz tanımı gösterilmiştir.

```
interface Numbers {
  typedef unsigned long Number;
  Number next_number  (in long n);
  void   next_number2 (in long n, out Number x);
  void   next_number3 (inout long n);
};
```

Yön Belirteçleri

Yukarıda tanımlanan operasyonların parametreleri, aşağıda tanımlanan, parametrelerin verilerinin taşınma yönünü gösteren yön belirteçleri ile tanımlanmıştır.

- **in**

  in belirteci parametrenin verisinin istemciden sunucuya gönderildiğini gösterir.

- **out**

  out belirteci parametrenin verisinin sunucudan istemciye gönderildiğini gösterir.

- **inout**

  inout belirteci parametrenin istemciden sunucuya veri taşıdığını ve sunucunun isterse bu değeri değiştirebileceğini gösterir. Operasyon tamamlandıktan sonra istemci, sunucunun değiştirdiği değeri kullanabilir.

Yön belirteçleri aşağıdaki sebeplerden dolayı gereklidir.

- **Performans**

  Yön belirteçleri verilerin iletilmesinde bazı yüklerden kurtulmak için gereklidir. Yön belirteçleri olmadan IDL derleyicisi hangi parametrelerin hangi amaçla kullanıldığını ayırt edemez. Bu durumda tüm parametrelerin istemciden sunucuya ve sunucunda da istemciye taşınması gerekir. Bu işlem, fazla sayıda parametresi olan ve parametrelerinin veri uzunlukları büyük olan operasyonların çalıştırılması sırasında performans kayıplarına neden olur.

- **Hafıza Yönetimi**

  Yön belirteçleri parametreler için hafıza ayrılması ve bu hafızanın serbest bırakılması işlemlerinin hangi tarafta (istemci veya sunucu tarafında) yapılması gerektiğini gösterir.

3.5 Kullanıcı Tarafından Tanımlanmış Hatalar (User Exceptions)

IDL, hata durumlarını göstermek için standart yöntem olarak hataları (exceptions) kullanır. IDL, belirli hata durumlarına özel hataların kullanıcı tarafından tanımlanmasına izin verir. Hataların tanımı kayıtların tanımına benzer. Bir hata, belirli bir hata durumuna özel ek bilgileri taşıyan elemanlar içerebilir. Hatalar **exception** anahtar kelimesi ile tanımlanırlar.

```
exception Failed { };

exception RangeError {
  long supplied_value;
  long minimum_value;
  long maximum_value;
};
```

Hata tanımları yeni bir tip tanımlarlar. Fakat hatalar kullanıcı tarafından tanımlanmış tiplerin veri elemanları olarak kullanılamazlar.

```
struct ErrorReport {
  Object     obj;
  RangeError exc;  // Hata. Hatalar veri elemanı olarak kullanılamazlar.
};
```

Hatalar sadece bir operasyonun düşebileceği olası bir hata durumunu belirtmek ve hatayı açıklayıcı bilgileri tanımlamak için kullanılırlar. Bir operasyonun düşebileceği olası bir hata durumu **raises** anahtar kelimesi ile belirtilir.

```
interface Failable {
  void can_fail ( )  raises ( Failed );
  void can_also_fail ( )  raises ( Failed, RangeError);
};
```

Örnekte de görüldüğü gibi bir operasyon birden fazla hata üretebilir. Bir operasyonun üretebileceği tüm hatalar operasyonun tanımında belirtlilmelidir. **raises** anahtar kelimesi ile belirtilen hata listesi boş olamaz. Bir operasyon hata üretmiyorsa **raises** anahtar kelimesi hiç

kullanılmamalıdır.

3.6 Nitelikler (Attributes)

Arayüzler operasyonların yanında çeşitli tipte veri elemanları içerebilirler. Bu elemanlara nitelik (attribute) denir. Nitelikler **attribute** anahtar kelimesi ile tanımlanırlar.

```
interface Thermostat {
  readonly attribute short temperature;
  attribute          short max_temp;
};
```

attribute anahtar kelimesi sadece bir arayüz tanımı içerisinde kullanılabilir. Nitelikler herhangi bir tipte olabilirler. Bir nitelik, IDL derleyicisi tarafından, bir istemcinin bir değeri alması ve değiştirmesi için kullanacağı bir çift operasyona dönüştürülür. **readonly** niteliklerin sadece değeri okunabilir, değeri değiştirilemez. Yukarıdaki arayüz tanımı aşağıdaki tanımla tamamen aynıdır.

```
interface Thermostat {
  short get_temperature ( );
  short get_max_temp ( );
  void  set_max_temp ( in short t );
};
```

Nitelikler, kayıt yapılarının elemanları gibi görünseler de yukarıda belirtildiği gibi sadece IDL derleyicisi tarafından operasyon çiftleri şeklinde dönüştürülürler. Bir arayüzün bir niteliği arayüz içerisinde tanımlanmış bir veri elemanı değildir. Ayrıca bir nitelik tanımından üretilen **get** ve **set** operasyonları hata üretemezler ve normal operasyon tanımı ile tanımlanmış operasyonlara göre hiç bir üstünlükleri yoktur. Bu nedenle nitelikler genellikle az kullanılırlar.

3.7 Arayüz Miras Alma (Inheritance)

IDL arayüzleri başka bir arayüzden türeyebilirler, yani başka bir arayüzün özelliklerini miras alabilirler (Henning ve Vinoski, 1999).

```
interface Thermometer {
  typedef short TempType;
  readonly attribute TempType temperature;
};

interface Thermostat : Thermometer {
  void set_max_temp( in TempType t );
};
```

Örnekte **Thermostat** arayüzü **Thermometer** arayüzünden türetilmiştir. Bir **Thermostat** arayüzü **Thermometer** arayüzünden miras aldığı **temperature** niteliğine ve kendine özel **set_max_temp** operasyonuna sahiptir. Ayrıca **set_max_temp** operasyonunun aldığı parametre tipi **TempType** tipindedir. **TempType** tipi de **Thermometer** arayüzünde tanımlanmıştır. Miras yoluyla **Thermostat** arayüzü, **Thermometer** arayüzünde tanımlanmış

her şeyi içermektedir.

Thermostat arayüzü **Thermometer** arayüzünün niteliklerine ve operasyonlarına sahip olduğu için **Thermometer** arayüzü bir **Thermometer** arayüzü gibi kullanılabilir.

```
interface Logger {
  long add ( in Thermometer t, int n );
  void remove ( in long id );
};
```

Örnekte verilen **Logger** arayüzü belirli aralıklarla sıcaklıkları kaydedilecek termometrelerin bir listesini tutmaktadır. Termometreler **add** operasyonuna nesne referansları geçirilerek kaydedilebilir.

Thermostat arayüzü **Thermometer** arayüzünden türediği için **add** operasyonuna bir **Thermometer** arayüzü gibi parametre olarak geçirilebilir. **add** operasyonu kendisine geçirilen parametrenin gerçekten hangi arayüz olduğunu bilemez.

4. CSORB: YENİ BİR YÖNTEM ARAYIŞI

Araştırma-Geliştirme Uzmanı olarak çalışmakta olduğum cyberSoft Enformasyon Teknolojileri Ltd. Şti. şu anda Maliye Bakanlığı'nın Vergi Dairesi Otomasyon Projesi (VEDOP)'ni yürütmektedir. Bu proje ile Türkiye çapında Tek Vergi Kimlik Numarası Uygulaması'na geçilmiştir. Proje kapsamı içinde vergi dairelerinde bulunan terminallerde çalışan VEDOP Programı'nın Merkez'deki uygulamalarla iletişim içinde olması gerekmiştir. Bu iletişimi sağlamak için bir yöntem bulunması gerekliliği doğmuş ve bu görev tarafıma verilmiştir. Şu anda mevcut bulunan ve 1. Bölüm'de de ele alınan yöntemleri inceledikten sonra CORBA'nın uygun bir yöntem olduğuna karar verilerek piyasadaki CORBA ORB uygulamaları incelenmiştir.

Çeşitli yazılım üreticilerinin ORB uygulamaları incelendikten sonra AT&T Bell Laboratuvarları'nın ürettiği omniORB ürünü seçildi. Bu seçimde MLC Systeme GmbH ve Charles Universitesi Yazılım Mühendisliği Bölümü Dağıtık Sistemler Araştırma Grubu'nun birlikte yaptıkları "CORBA Comparison Project" adlı peformans testi etkili oldu. Bu test ücretsiz olarak kaynak kodu ile birlikte dağıtılan omniORB'nin yüksek fiyatlara sahip diğer ticari ORB'lerden kesinlikle daha iyi performans ortaya koyduğunu gösteriyordu (MLC, 1999).

omniORB iyi performansına rağmen sadece *Kalıcı Sunucu Politikası*'nı destekliyordu. Yani CORBA nesnelerinin uygulamalarını içeren programların sunucu makinalar üzerinde elle çalıştırılması gerekiyordu. VEDOP Projesi için *Ortak Sunucu Politikası* daha iyi bir yöntemdi. Bu yöntemde TNA, sunucuları sadece herhangi bir istek geldiğinde otomatik olarak başlatmaktadır. Dolayısıyla hiç kullanılmayacak bir nesne için bir sunucu programın çalıştırılmasına gerek kalmamaktadır.

Ayrıca CORBA nesneleri örneğe özel (istemci'ye özel) veri tutamadıkları (stateless objects) için bir nesne tipine ait bir nesnenin tüm istemcilerine aynı nesne örneği (object instance) hizmet veriyordu. Bu nedenle nesnelerin istemcilere özel verileri nesne üzerinde değil istemci üzerinde tutulmaktaydı. Veriyi istemciler tuttuğu için, nesnenin ihtiyaç duyduğu bilgiler sunucu nesnenin metodlarına parametre olarak geçirilmekte ve değişen veri yine istemcilere geri aktarılmaktaydı. Bu da her metod çağrımında daha fazla verinin taşınması nedeniyle performans kaybına yol açmaktaydı.

Halbuki bir istemciye karşılık gelen bir sunucu nesne örneği yaratılması sayesinde her bir istemciye özel verilerin nesne üzerinde tutulması mümkündür. Her bir istemci için hizmet

verecek yeni bir nesne örneği yaratılır ve istemci işini bitirdiğinde sunucu nesne öldürülür. Dolayısıyla istemci ve nesne örneği arasında bire-bir ilişki kurulur. Bir nesne örneğini başka bir istemci kullanamayacağı için istemcinin verisini sunucu nesne tutabilir. Dolayısıyla nesneni metodlarına istemcinin verisini parametre olarak gönderme yükü ortadan kalkmakta ve yukarıda bahsedilen performans kaybı ortadan kalkmaktadır.

Bahsedilen nedenlerden dolayı Proje Yönetimi ile VEDOP Projesi'nin yapısına daha yatkın ve daha uygun olan bir ORB uygulaması yazılmasına karar veridi. Ve adı CSORB (cyberSoft Object Request Broker) olan ORB uygulamasının yazılması için araştırmalara başlandı.

CSORB'un aynı anda birden fazla istemciye hizmet verebilmesi için *multi-threaded* olması zorunluydu. Bu nedenle "Multi-Threaded Programming" hakkında araştırmalar yapıldı. Daha sonra "TCP/IP ve Socket Programming" hakkında bilgilerin toplanması ve network iletişimi bileşenlerinin yazılması gerekti.

omniORB, ORB'nin sunucu ve istemci tarafındaki kısımlarını aynı kütüphane içinde yer alacak şekilde uygulayan bir Kütüphane Bazlı ORB şeklinde uygulanmıştı. Yani omniORB çalışma zamanı kütüphaneleri (run-time DLLs) ORB'nin hem istemci hem de sunucu tarafındaki kısımlarını içermektedir. Yazılan CORBA programları bu kütüphanelerle derlendiğinden sunucu programlar aynı zamanda ORB'nin istemci tarafındaki kısımları, istemci programlar da ORB'nin sunucu tarafındaki kısımlarını içermektedir. Bu nedenle programların boyutları büyümekte ve programların dağıtımı zorlaşmaktadır.

Bu nedenle CSORB'un istemci ve sunucu kısımlarının ayrı ayrı uygulanmasına karar verildi. CSORB üç bölümden oluşmaktadır. İstemci tarafı,sunucu tarafı ve IDL Derleyicisi.

İstemci tarafı, CSOrbProxy.DLL adında bir kütüphane olup oldukça küçük bir boyutta yaratılmış dolayısıyla istemci terminallere dağıtımı kolaylaştırılmıştır. CSORB'un sunucu tarafında bulunan nesnelere ulaşımı sağlayan bileşendir.

Sunucu tarafı, CSORBManager.EXE adında, bir NT Service'i olarak çalışacak bir program olarak yaratılmıştır.

CSORB, bir Sunucu Bazlı ORB'dir. Yani CSORB'un istemcileri CSORB'un istemci tarafını kullanarak CSORB Sunucu tarafı ile iletişim kurmakta ve sunucu isteği ilgili nesne uygulamasına yönlendirmektedir. Bu nedenle CSORB'un yönetimi tek bir merkezden yürütülebilmekte ve yönetim kolaylığı sağlanmaktadır.

CSORB'un istemci ve sunucu tarafı kısımları yazıldıktan sonra CSORB IDL Derleyicisi'nin

yazılmasına başlandı. Önce IDL Derleyisi için *lexical analysis* ve *syntax analysis* konularında araştırmalar yapıldı. Daha sonra uygulama geliştirme dili olarak kullandığımız C++ diline yapılacak dönüşümler için CORBA Standardı'nın tanımladığı C++ dili dönüşümü incelendi.

CSORB IDL, CORBA standardının tanımladığı IDL üzerine kurulmuş olmakla birlikte VEDOP Projesi'nin özel ihtiyaçları nedeniyle IDL'in tüm özelliklerini taşımamakta ve bazı özel eklere sahip bulunmaktadır. Örneğin CSORB server nesnelerinin ayrı kütüphanelerde yer alması kararlaştırıldığından dolayı, nesne arayüzü tanımlarına nesne uygulamasının içinde bulunacağı kütüphaneleri tanımlayan özel ekler konulmuştur.

Örneğin aşağıdaki CSORB IDL tanımıyla tanımlanan **Bank** nesnesinin uygulaması Bank.DLL adlı kütüphanede yer alacaktır.

```
interface Bank in "Bank.dll"
{
    BOOL OpenBank(in BANK_ID id);
    ...
```

CSORB IDL tarafından desteklenen temel veri tipleri aşağıda gösterilmiştir.

Çizelge 4.1 CSORB IDL Temel Veri Tipleri.

DOUBLE	INT64	FLOAT	ULONG	DWORD	CHAR
BOOL	UINT	WORD	LONG	BYTE	INT

CSORB IDL bu temel veri tiplerinin yanında **struct** anahtar kelimesi ile kullanıcı tarafından yeni veri tiplerinin tanımlanmasına izin verir. Ayrıca **typedef** anahtar kelimesi ile isimlendirilmiş tiplerin tanımlanması da mümkündür.

Sonuç olarak ortaya çıkan CSORB, omniORB'ye göre performansı daha düşük olmasına rağmen VEDOP Projesi'nin tüm ihtiyaçları karşıladı.

Ayrıca CSORB'un sadece VEDOP Projesi'ne özel olmadığını göstermek için CSORB ile bir bankacılık uygulaması geliştirilmiştir.

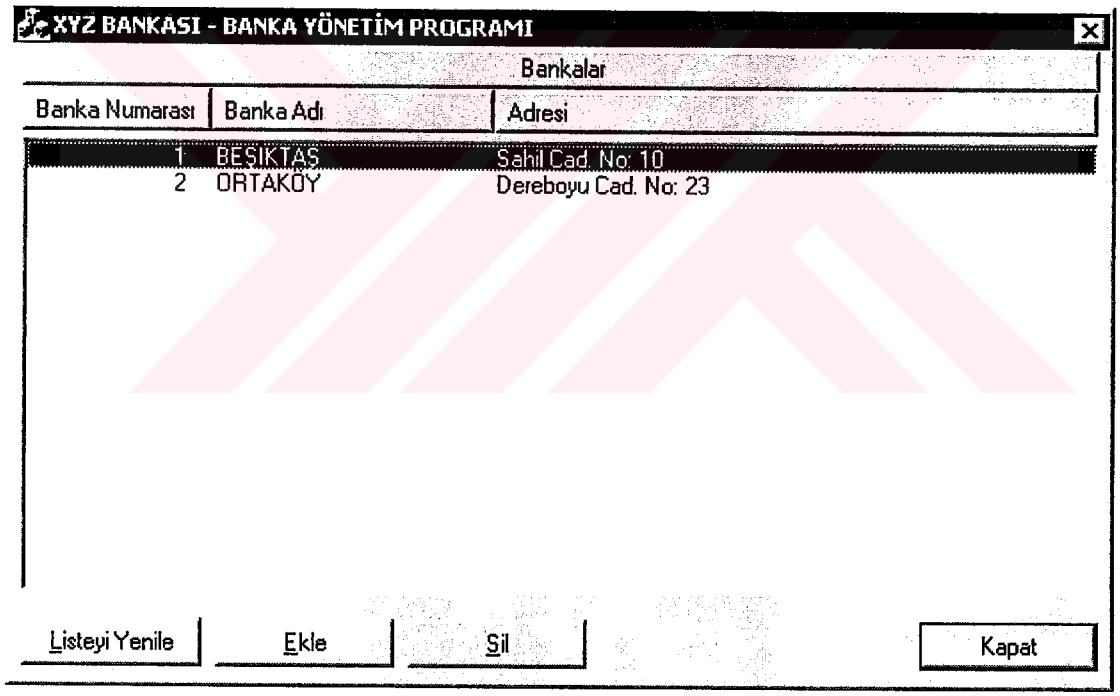
Bu uygulamanın CSORB IDL ile tanımlanmış arayüzleri Ek 1'de verilmiştir.

Bu IDL dosyasında **BankManager, Bank** ve **Account** adında üç adet arayüz tanımlanmıştır. **BankManager** arayüzü banka şubelerini yönetecek olan CORBA nesnesinin arayüzünü tanımlar. **BankManager** CORBA nesnesi, yeni şube ekleme, şube silme ve şube bilgilerinin görüntülenmesi gibi işlemleri yerine getirecektir. **Bank** arayüzü ise bir banka şubesinde yapılacak işlemleri idare edecektir. Yeni hesapların açılması, hesapların görüntülenmesi ve

hesapların kapatılması gibi işlemler **Bank** arayüzü içinde tanımlanmış operasyonlar ile yerine getirilecektir.Örneğin **CreateAccount** operasyonu ilgili şubede yeni bir hesap açmak için kullanacaktır. **Account** arayüzü ise bir mevduat sahibinin para çekme, para yatırma ve bakiye görüntüleme gibi işlemlerini yerine getirecektir.

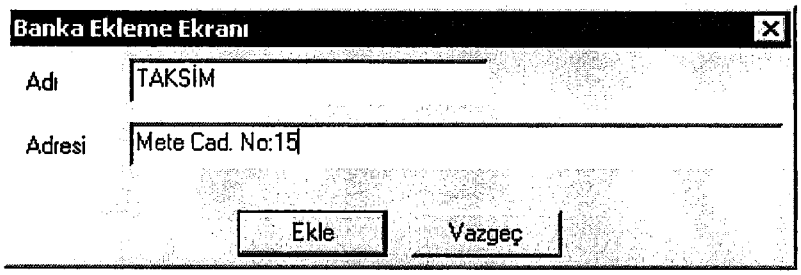
Bu nesnelerin uygulaması ise IDL arayüz tanımlarında belirtildiği gibi Bank.DLL adındaki kütüphane içinde yer alacaktır. Bu kütüphane içinde yer alan nesne uygulamaları, CSORB Sunucu tarafı tarafından ilgili operasyonların yerine getirilmesi için kullanılacaktır.

Uygulama bir Banka Yönetim Programı içermektedir. Bu program bankanın merkezinde çalışacaktır. CSORB Sunucu tarafı bankanın merkezine kurulduktan sonra bu program çalıştırılmalıdır. Bu programla mevcut şubelerin görüntülenmesi, yeni şubelerin açılması, mevcut şubeleri silme gibi işlemler yerine getirilebilir. Bir şubenin çalışmaya başlaması için önce Banka Yönetim Programı kullanılarak şube kaydının eklenmesi gerekir. Şekil 4.1'de Banka Yönetim Programı'nın ana ekranı gösterilmiştir.



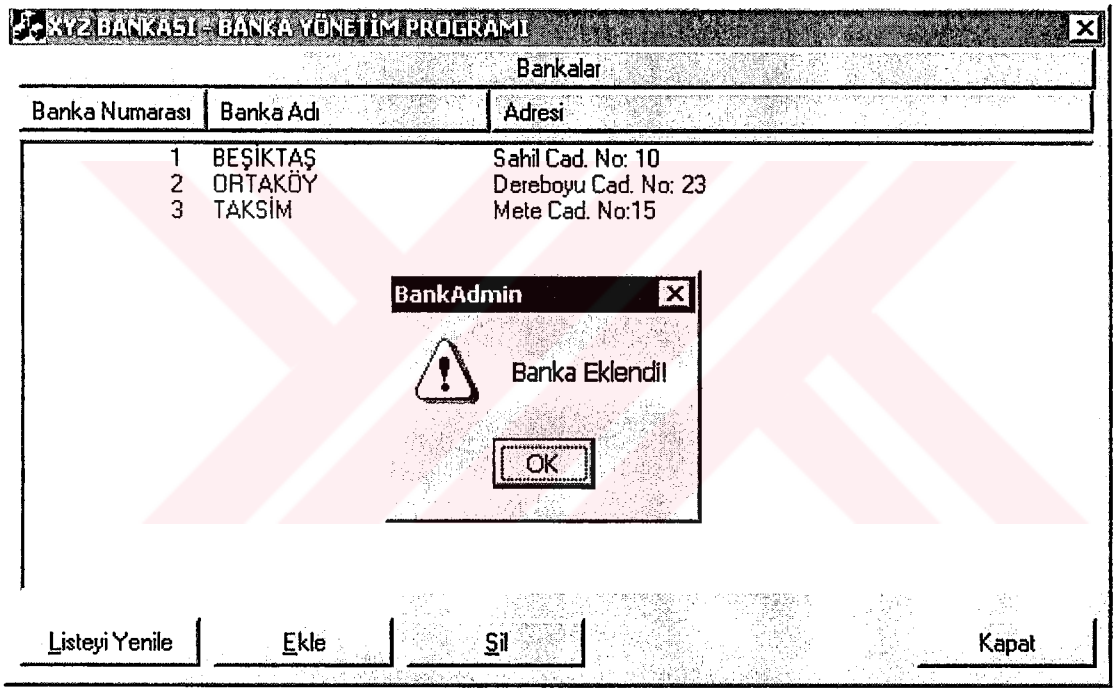
Şekil 4.1 Banka Yönetim Programı Ana Ekranı.

Şekil 4.1'de de görüldüğü gibi Ana Ekran'da Listeyi Yenile, Ekle, Sil ve Kapat düğmeleri bulunmaktadır. Listeyi Yenile düğmesi banka listesinin yeniden alınmasını sağlar. Ekle düğmesi ise yeni bir şube eklemek için kullanılır. Ekle düğmesine basıldığında Banka Ekleme Ekranı açılır (Şekil 4.2).



Şekil 4.2 Banka Ekleme Ekranı.

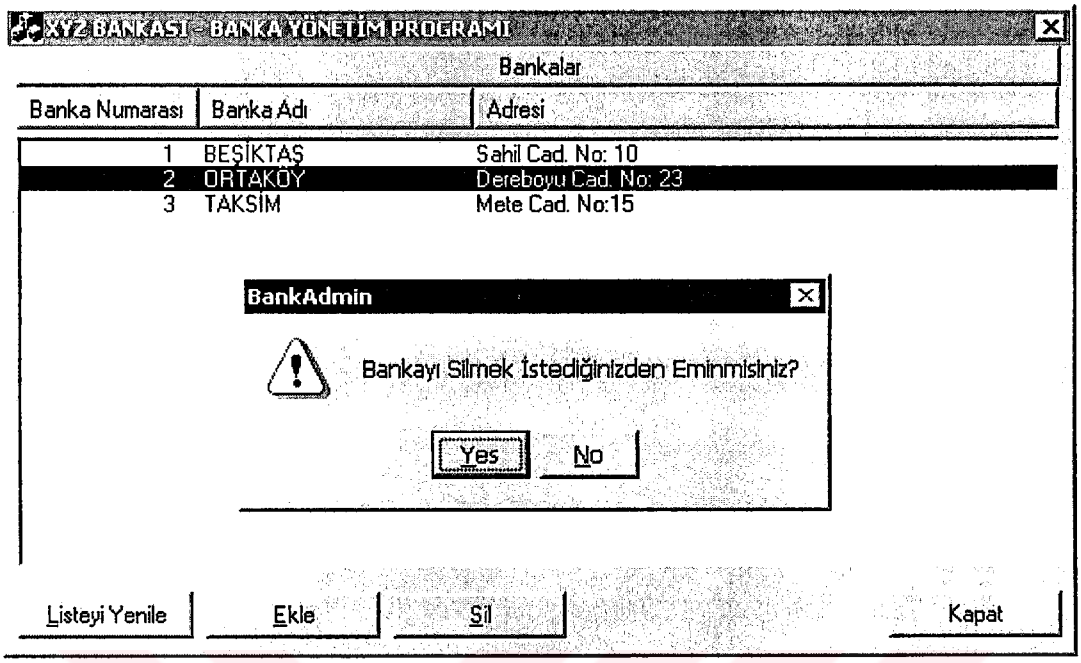
Banka Ekleme Ekranı'ndaki alanlar eklenecek yeni şubenin bilgilerini göstermektedir. Yeni şubenin adı ve adresi girildikten sonra Ekle düğmesine basılarak yeni şube sisteme eklenir. Şube eklendikten sonra ekrana,şubenin başarılı bir şekilde eklendiğini gösteren bir uyarı penceresi gelir (Şekil 4.3).



Şekil 4.3 Yeni şubenin eklendiğini gösteren uyarı ekranı.

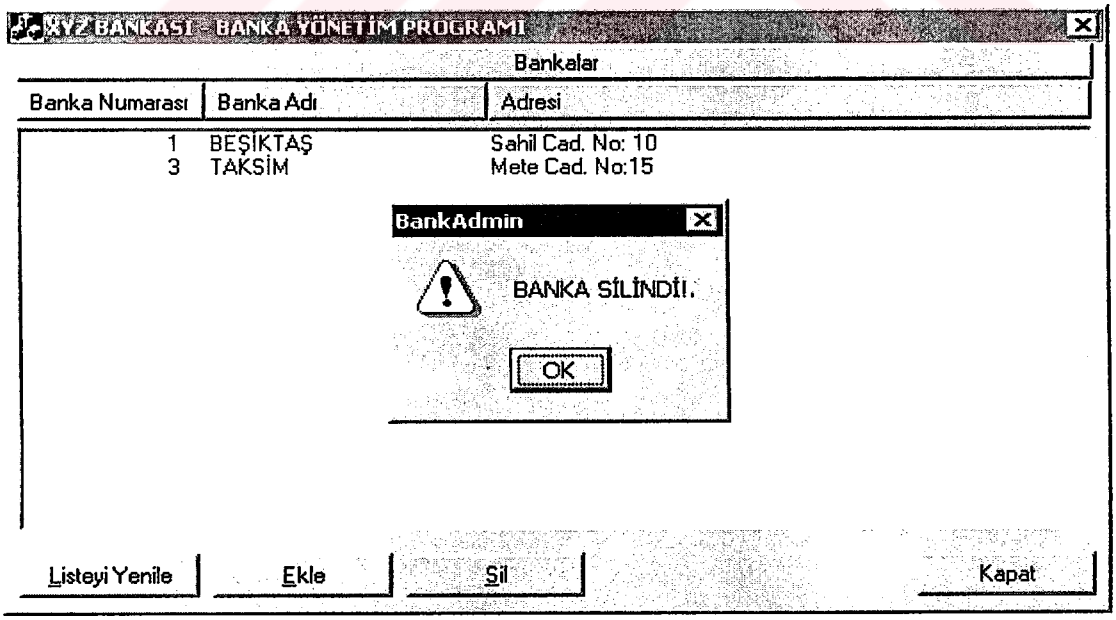
Eklenen şube Banka Yönetim Programındaki Bankalar Listesi'nde otomatik olarak yer alacaktır.

Ana ekrandaki Sil düğmesi banka listesinde seçili bulunan banka şubesini silmek için kullanılır. Bu düğmeye basıldığında, kazara bankanın silinmesini engellemek amacıyla bir onay penceresi ekrana gelir (Şekil 4.4).



Şekil 4.4 Banka Silme Onay Ekranı.

Bu ekranda YES düğmesine basıldığında banka silinecek, NO düğmesine basıldığında ise silme işleminden vazgeçilecektir. YES düğmesine basıldığında bankanın silindiğine dair bir uyarı penceresi ekrana gelecektir (Şekil 4.5).



Şekil 4.5 Bankanın silindiğini gösteren uyarı ekranı.

Bu şekilde banka şubeleri kullanıma açıldıktan sonra banka şubelerine Banka Şube Programı ile CSORB'un istemci tarafı olan CSORBProxy.DLL adlı kütüphane dağıtılır. Banka Şube Programı, CSORB'un istemci tarafını kullanarak merkezdeki CSORB Sunucu tarafı ile iletişim kurarak **Bank** ve **Account** arayüzlerinde gösterilen, şubelere özel işlemleri yerine getirir.

5. SONUÇLAR

CORBA Mimarisi, günümüzde gerçek zamanlı uygulamalarda, sektöre özel yazılımlarda ve çeşitli araştırma projelerinde kullanılmaktadır. CORBA'nın önemli avantajları nesneye dayalı olması, platformdan ve kullanılan programlama dilinden bağımsız olması ve nesne lokasyonunun programcıdan soyutlanmasıdır. CORBA bu avantajları birlikte sunması ile diğer yazılım mimarileri ve teknolojilerinden belirgin bir şekilde ayrılmaktadır.

CORBA, aralarında Nokia, Ericsson, Nike, Boeing, American Airlines, Lufthansa, CNN, Harvard Üniversitesi gibi büyük kurum ve kuruluşların projelerinin de bulunduğu değişik amaçlı projelerde kullanılmaktadır.

CORBA'nın diğer teknolojileri geride bıraktığı bir diğer özelliği de esnekliğidir. Herhangi bir teknolojinin bir projede yetersiz kalması durumunda yapılacak tek şey, o teknoloji değiştirilemeyeceği için yeni bir teknoloji yada yöntem bulmaktır. Halbuki CORBA, belirli bir teknoloji ile uygulanmış bir yazılım yada yazılım kütüphanesi değil, genel olarak ORB'lerin hangi kurallara uyması gerektiğini belirten bir standart olduğu için, bir ORB'nin belirli bir projenin ihtiyaçlarını karşılamaması durumunda, bu projenin özel ihtiyaçlarını karşılayan ve CORBA standardına uygun olan yeni ORB'lerin edinilmesine ya da geliştirilmesine olanak tanır. Sonuçta ortaya çıkan farklı bir amaca hizmet eden, farklı özelliklere sahip ama CORBA uyumlu olan yeni bir ORB'dir. Dolayısıyla mimari değiştirilmeden, farklı ORB uygulamalarından duruma uygun olanı seçilerek farklı ihtiyaçlar karşılanabilir. Örneğin, projenin belirli bir aşamasında bir işlem görüntüleyici (transaction-processing monitor-TPM) özelliğine ihtiyaç duyulduğunda, uygulama, TPM hizmeti sağlayan yeni bir ORB edinilerek yada geliştirilerek uygulamada değişiklik yapılamdan bu işlevi de sağlar hale getirilebilir.

CORBA'nın bu önemli özelliği, çalışma kapsamında geliştirilen Maliye Bakanlığı'nın Vergi Dairesi Otomasyon Projesi'nin özel ihtiyaçlarına yönelik olarak tasarlanıp geliştirilen cyberSoft ORB (CSORB) ile gösterilmiştir.

Bütün bu avantajlarının yanında, CORBA, nesne lokasyonu konusunda çok yüksek bir soyutlama sağladığı için heterojen ortamlarda çalışan uygulamalarda performans kaybına yol açabilir. Bunun nedeni, farklı ortamlarda aynı davranışı sergileyebilmek için yapılan marshalling ve demarshalling işlemleridir. Ama bu sadece yazılım teknolojileri alanında değil her alanda karşımıza çıkan, bir konuda avantajlı olabilmek için diğer konulardan ödün vermek zorunluluğundan kaynaklanır (trade-off).

KAYNAKLAR

Chizmadia, D., (1999), "CORBAsec: Securing Distributed Systems", Software Development Online, http://www.sdmagazine.com/.

Gokhaleve, A.S. ve Schmidt, D.C., (1996), "The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks", GLOBECOM Conference, 1:15-17, 12-18 Kasım 1996, Londra.

Gokhaleve, A.S. ve Schmidt, D.C., (1997), "Evaluating CORBA Latency and Scalability Over High-Speed ATM Networks", Proceedings of ICDCS'97, 1:3, 27-30 Mayıs 1997, Baltimore,Maryland.

Henning M. ve Vinoski S., (1999), Advanced CORBA Programming with C++, Addison-Wesley Publishing.

Inprise, (2000), Inprise Case Study. http://www.inprise.com/cgi-bin/cstudy.pl.

IONA, (2000), IONA Company Customers. http://www.iona.com/info/aboutus/customers/index.html.

Microsoft ve Digital Equipment Corporation, (1995), "Distributed Component Object model Specification. Technical Report", Ekim 1995.

MLC, (1999), "CORBA Comparison Project", MLC Systeme GmbH ve Distributed Systems Research Group, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, 16 Ağustos 1999.

OMG, (1997), "The Common Object Request Broker: Architecture and Specification", 2.2 Versiyonu, Şubat 1998.

OMG, (2000), CORBA Success Stories. http://www.corba.org/success.htm.

Rosenberger, J., (1998), Sams Teach Yourself CORBA in 14 Days, Sams Publishing.

Schmidt, D.C. ve Vinoski, S., (1995), "Comparing Alternative Client-Side Distributed Programming Techniques", C++ Report Magazine, 5:1-18.

Schmidt, D.C., Harrison, T. ve Al Shaer, E., (1995), "Object-Oriented Components for High-Speed Network Programming",USENIX Object-Oriented Technologies Conference, 1:8-12, Haziran 1995, Monterey, CA.

Schmidt, D.C. ve Vinoski, S., (1995), "Comparing Alternative Server Programming Techniques", C++ Report Magazine, 10:1-21.

Schmidt, D.C. ve Vinoski, S., (1996), "Comparing Alternative Programming Techniques for Multi-Threaded CORBA Servers", C++ Report Magazine, 8:1-13.

Schmidt, D.C ve Vinoski, S., (1997), "Object Adapters: Concepts and Terminology", C++ Report Magazine, 11: 1-5.

Sun, (2000), Java Remote Method Invocation (RMI), Sun Microsystems Inc. http://www.javasoft.com/products/jdk/rmi/index.html.

Vinoski, S., (1993), "Distributed Object Computing With CORBA", C++ Report Magazine, 7:2-8.

EKLER

Ek 1 Bank.IDL Dosyası.

Ek 1 Bank.IDL Dosyası

```
// Bank IDL File
//

typedef LONG BANK_ID;
typedef LONG ACCOUNT_ID;
typedef CHAR NAME[32];
typedef CHAR ADDRESS[128];
typedef CHAR ERROR_STRING[255];

typedef struct
{
    CHAR            creation_date[16];
    NAME            name;
    NAME            surname;
    ADDRESS         address;
    CHAR            tel[12];
    DOUBLE          remaining;
} NEW_ACCOUNT_INFO;

typedef struct
{
    ACCOUNT_ID      id;
    BANK_ID         bank_id;
    NEW_ACCOUNT_INFO info;
} ACCOUNT_INFO;

typedef struct
{
    NAME            name;
    ADDRESS         address;
} NEW_BANK_INFO;

typedef struct
{
    BANK_ID         id;
    LONG            AccountCount;
    NEW_BANK_INFO   info;
} BANK_INFO;

interface BankManager in "Bank.dll"
{
    BOOL CreateBank(in NEW_BANK_INFO bi, out BANK_ID id);
    BOOL DeleteBank(in BANK_ID id);
    BOOL GetBankCount(out LONG BankCount);
    BOOL GetBankInfo(in LONG Index, out BANK_INFO BankInfo);

    BOOL GetLastError(out ERROR_STRING errStr);
}

interface Bank in "Bank.dll"
{
    BOOL OpenBank(in BANK_ID id);

    BOOL CreateAccount(in NEW_ACCOUNT_INFO ai, out ACCOUNT_ID id);
    BOOL DeleteAccount(in ACCOUNT_ID id);
    BOOL GetAccountCount(out LONG AccountCount);
    BOOL GetAccountInfo(in LONG Index, out ACCOUNT_INFO AccountInfo);
    BOOL GetLastError(out ERROR_STRING errStr);
}

interface Account in "Bank.dll"
{
    BOOL OpenAccount(in ACCOUNT_ID id);

    BOOL GetRemaining(out DOUBLE amount);
    BOOL Deposit(in DOUBLE amount);
    BOOL WithDraw(in DOUBLE amount);
    BOOL GetInfo(out ACCOUNT_INFO AccountInfo);
    BOOL GetLastError(out ERROR_STRING errStr);
}
```

ÖZGEÇMİŞ

Doğum tarihi	05.06.1975	
Doğum yeri	Karabük	
Lise	1989-1992	İstanbul Kabataş Erkek Lisesi
Lisans	1992-1996	Yıldız Üniversitesi Kimya ve Metalurji Fakültesi Matematik Mühendisliği Bölümü

Çalıştığı kurum(lar)

1996-1999	Link Bilgisayar Ltd Şti.
1999-Devam ediyor	cyberSoft Enformasyon Teknolojileri Ltd Şti.