İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

NESNEYE YÖNELİK TASARIM DESENLERİ VE UYGULAMALARI

YÜKSEK LİSANS TEZİ

Müh. Mehmet Dündar AKIN
50496061711

Tezin Enstitüye Verildiği Tarih : 5 Haziran 2000
Tezin Savunulduğu Tarih : 26 Haziran 2000

Tez Danışmanı : Doç.Dr. Nadia ERDOĞAN

Diğer Jüri Üyeleri Prof.Dr. Muhittin GÖKMEN (İTÜ)

Doç.Dr. Hasan DAĞ (İTÜ)

HAZİRAN 2000

ÖNSÖZ

Bu çalışmanın hazırlanmasındaki yardım ve yönlendirmelerinden dolayı tez danışmanım Sayın Doç. Dr. Nadia ERDOĞAN'a,

Çalışmanın gerçekleştirilme safhasındaki maddi manevi desteğini benden esirgemeyen ve bu zor dönemlerde bana sabırla katlanan eşim Ganime Betül AKIN'a

Çeşitli fikir ve önerileriyle bana yol gösteren kardeşim Ahmet Afşın AKIN'a ve diğer arkadaşlarıma,

Teşekkür ederim.

Haziran 2000 Mehmet Dündar AKIN

İÇİNDEKİLER

KISALTMALAR

VCL : Visual Component Library
OMT : Object Model Technique
UML : Unified Modelling Language

ŞEKİL LİSTESİ

NESNEYE YÖNELİK TASARIM DESENLERİ VE UYGULAMALARI

ÖZET

Nesneye yönelik tasarım ve program geliştirme, özellikle kullanıcıların programlardan beklediklerinin artması ve yazılımların gelişim çevriminde kullanılan sürenin değer kazanmasıyla son yılların üzerinde en çok durulan konularından biri oldu. Yazılımların en uzun zaman alan bölümü olan test ve bakım zamanını düşürebilmek için yapısal sistemlerden modüller üzerinde programcı ve tasarımcıların daha rahat, hızlı ve az hata ile çalışabildiği nesneye yönelik sistemlere geçiş bir zorunluluğa dönüştü. Tasarım desenleri, sürekli olarak karşımıza çıkan tasarım problemlerin ve bu problemlere tecrübeli kullanıcıların yılların birikimi ile geliştirdikleri metodlarla bulunan çözümlerin isimlendirilip sınıflandırılmasını sağlamıştır. Tasarım desenleri nesneye yönelik tasarımların sorunsuz olarak tamamlanabilmesinde bir rehber ve başvuru kaynağı olarak kullanılabilir.

Tasarım desenleri adından bahsedilmeye başlanan Doksanlı yılların başından bu yana artan bir ilgi karşılanmaya devam etmektedir. Her yıl yeni tasarım desenleri ve tasarım desenlerinin uygulanması üzerine konferanslar düzenlenmekte ve çeşitli makaleler yayınlamaktadır.

Bu tez çalışmasının ilk bölümünde Tasarım desenlerinin genel olarak tanımı yapılmış, tasarım desenlerinin özellikleri, sınıfları ve kullanımı ile ilgili bilgiler verilmiştir. Bunların yanında daha sonraki bölümlerde kullanılan diyagram ve şekiller için temel bir açıklama ve nesneye yönelik tasarımın ana ilkeleri hakkında bir hatırlatma verilmektedir.İkinci bölümde ise temel bazı tasarım desenleri belirlenen bir formatta sunulmakta ve uygulama bölümlerinde de bu tasarım desenleri ile ilgili tartışmalar ve yapılabilecek değişiklikler belirtilmektedir. Gerçekleştirilen bir projede bu tasarım desenlerinin nasıl kullanıldığı ve gerçeklenmesi sırasında ortaya çıkan sorunlardan ve tasarım desenlerinin ihtiyaçlara göre nasıl değiştirilebileceğinden bahsedilmiştir.Son bölümde de gerçekleştirilen uygulama ile ilgili bilgiler, nesne altyapısı ve sonuç bölümleri yer almaktadır.

OBJECT ORIENTED DESIGN PATTERNS AND ITS APPLICATIONS

SUMMARY

Object oriented design patterns were first introduced in a book authored by Gamma E., Helm R. Johnson R. and Vlissides J. called "Design Patterns: Elements of Reuseable Object-Oriented Software" and received a worldwide acclaimation at the time. The book describes design patterns as "description of communicating objects and classes that are customized to solve a general design problem in a particular context".

Design patterns allow solutions that were developed by experienced designers and programmers to be named and catalogued. One of the short term benefits of design patterns is the fact that as a result of this naming, they act as a common vocabulary to simplify and enhance the dialog between developers. They also enable the design to be discussed at an abstract level without going into details.

Besides being a documentation tool, design patterns can provide flexible and generic solutions to problems. The outcomes of these solutions can be predetermined as to their pros and cons.

Yet, in most cases, developers face problems that have different charecteristics. Because of this, direct application of a design pattern to a problem may not provide an exact solution to the problem. In short, there is no silver bullet.

It's crucial for software to be flexible and easy to use for it to suceed. Using object oriented techniques are almost a must for such projects. Object oriented design patterns aid developers in solving recurring problems in projects.

In this study we also used some design patterns in an experimental CAD kernel GeoCAD. GeoCAD kernel is a very flexible and modular CAD arhitecture and can be expanded and modified according to needs. Use of design patterns has given us the power of reuse in important phases of design.

Here is an outline of the topics discussed in this study.

In the first part, Design Pattern concept is defined and a very brief historical backgorund is given. Then, the cpmponents of a design pattern and different classes of design patterns are introduced.

Since design patterns aims object oriented approach, an introduction to crucial object oriented concepts and their representation is also given in the first part.

In second part, several important design patterns are introduced and a detailed explanation of modifications is given.

In the last part the architecture of GeoCAD CAD kernel and conclusion is given.

1. Giriş

Yazılım alanında "Tasarım desenleri" kavramına, bu konunun ilk olarak ortaya atıldığı "Design Patterns Elements of Reusable Object-Oriented Software" kitabının önsözünü hazırlayan ve yazılım konusunda dünya çapında bir otorite olarak kabul edilen Grady Booch,

"İyi tasarlanmış tüm nesneye yönelik mimariler tasarım desenleri ile doludur. Nesneye yönelik bir sistemin kalitesi hakkında inceleme yaparken dikkat ettiğim bir nokta tasarımcıların sistem içerisindeki nesnelerin birbirleri ile olan ortak çalışma ve ilişkileri ne kadar iyi düzenledikleridir. Sistemin tasarımı aşamasında bu tür mekanizmalara odaklanmak daha küçük, etkili ve anlaşılır mimarilerin oluşmasında yardımcı olabilir."

Sözleri ile günümüz yazılım dünyasında son beş altı yıldır etkisini hala devam ettiren ve gelip geçici bir moda olmadığını gösteren tasarım desenleri konusunun önemini vurgulamıştır [2].

Nesneye yönelik tasarım, özellikle tekrar kullanılabilir nesneye yönelik tasarım zor ve zahmetli bir çalışma gerektirir. Uygun nesne yapılarının belirlenmesi, bunların sınıflandırılması, sınıflar arası haberleşme ve kalıtımla alınacak özelliklerin belirlenmesi gerekir. Tasarımınız hem o an üzerinde çalışmakta olduğunuz proje için özelleşmiş olmalı hem de gelecekteki projeler için genel ve tekrar kullanılabilir parçalardan oluşabilmelidir. Tecrübeli nesneye yönelik tasarımcılar gerçekten esnek ve tekrar kullanılabilir bir tasarım yapmanın imkansıza yakın olduğu konusunda birleşirler, bir tasarım, tamamlanmadan önce pek çok kez ciddi değişikliklere uğrar. Bunun yanında, Tecrübeli nesneye yönelik tasarımcılar iyi tasarımlar yapar, konusunda yeni olan tasarımcılar ise kendilerine verilen seçeneklerin çokluğu karşısında şaşkınlığa düşer ve çoğunlukla nesneye yönelik olmayan eski alışkanlıklarına dönme eğilimindedirler. Nesneye yönelik tasarım konusunda yeni olan kişilerin bu konuda tecrübe kazanmaları gerçekten uzun süre alabilir. Nesneye yönelik tasarım konusunda tecrübeli tasarımcıları diğerlerinden farklı kılan nedir?

Tecrübeli tasarımcılar sorunları sürekli olarak en başından itibaren çözmeye çalışmazlar, aksine, geçmişte işlerine yaramış olan çözümleri sorunlara tekrar tekrar uygularlar.

Tasarım desenleri fikri ilk olarak 1978 yılında mimar Cristopher Alexander ve Sara Ishikava tarafından yazılan "A Pattern Language" isimli kitapta ortaya atılmıştır [1]. Alexander, kitabında "Her desen, kendi ortamı içerisinde arka arkaya meydana gelen bir problemi ve problemin temelindeki çözümü tarif eder, öyle ki bu çözümü aynı şekilde tekrarlamadan milyonlarca kez kullanabilirsiniz" diyerek Tasarım deseninin genel bir tarifini yapar. Her ne kadar Alexander mimari alanındaki tasarımdan bahsetse de söyledikleri, nesneye yönelik tasarım konusunda da doğrudur. Mimarideki duvarların, pencerelerin veya kapıların yerini program tasarımında sınıflar, arayüzler ve nesneler alır. Genel olarak bir tasarım deseni dört önemli bölümden oluşur [2]:

1. Desenin adı Tasarım problemlerini ve problem için sunulan çözümleri ve çözümlerin avantaj ve dezavantajlarını isimlendirmekte kullanılır. Eğer bir desen isimlendirilmişse, proje üzerinde birlikte çalışan kişiler konu üzerinde daha rahat anlaşabilir ve detaylara dalmadan üst seviyede tartışabilirler. Bir desene iyi ve anlaşılır bir isim bulmak oldukça zor bir iş olabilir.

2. Problem, tasarım deseninin hangi durumda uygulanacağını tanımlar. Problem, bir algoritmanın nasıl bir nesne olarak ifade edilebileceği olabileceği gibi, tasarım deseninin uygulanmasından önce yapılması gereken işlerin bir listesi de olabilir.

3. Çözüm, tasarımı oluşturan elemanları, onların ilişkilerini, sorumluluklarını ve nasıl bir arada çalışmaları gerektiğini belirler. Önemli bir nokta ise çözümün kesinlikle spesifik bir tasarım sunmamasıdır, desenlerin amacı belli sınıflardaki problemler için soyut ve jenerik çözümler sumaktır.

4. Sonuç, tasarım desenin uygulanmasının sonuçları ve sisteme getirdiği fayda ve zararları belirler. Tasarım desenleri problemlere çözümler getirirken bir taraftan da bazı yönlerden sistemi zayıflatabilir, kodlama miktarını arttırabilir veya sınıf hiyerarşisinin esnekliğini azaltabilir. Tasarım desenleri anlatılırken bu tür dezavantajların muhakkak belirtilmesi gerekmektedir.

Verilen bu tanım gene olarak tasarım desenlerinin neleri içermesi gerektiğini tarif eder. Tasarım desenlerine nesneye yönelik programlama yönünden bakıldığında ise karşımıza çeşitli tarifler çıkar.

Bu noktada öncelikle neyin tasarım deseni olmadığını belirlemekte fayda var, Sınıflar içerisine konarak tekrar kullanılabilir hale getirilmiş olan bağlı listeler, ağaçlar veya çırpma (hash) tabloları tasarım deseni değildir. Tasarım deseni tanımı karmaşık, özelleşmiş bir alandaki çözümler içim de kullanılamaz. Gamma, [2] Tasarım desenleri için, "Genel bir tasarım problemini çözmek amacıyla özelleştirilmiş, birbiriyle haberleşen nesnelerin tarifidir" tanımını kullanır.

Kanadalı bir mimara üniversite kampüsünde öğrencilerin yürüyecekleri yolların yapılması işi verilir, mimar önce kar yağmasını bekler ve daha sonra karın üzerinde öğrencilerin bıraktığı izlerin yoğun olduğu yerlere yolları yapar. Bir konuda uzman kişilerin tecrübelerinden faydalanmak tasarım ve uygulama için geçen zamanı kısaltır ve sunulan çözümlerin de en iyiye daha yakın olmasını sağlar.

Frank Buschmann, Tasarım desenlerinin yetenekli yazılım mühendislerinin ortak tecrübelerinin üzerine kendi tasarımlarımızı inşa etmemize yardımcı olduğunu söyleyerek mevcut ve doğruluğu ispatlanmış tecrübelerin daha iyi tasarımların oluşmasını sağlayacağını belirtir [3]. Buschmann Tasarım desenlerini "Belirli tasarım aşamalarında sürekli ortaya çıkan problemlerin ve bu problemlere uygun çözümlerin neler olduğunun belirlenmesi" olarak tarif eder.

Her tasarım deseni belli bir nesneye yönelik problem üzerine odaklanmıştır. Tasarım desenlerinin uygulamada nasıl kullanılacağının anlaşılabilmesi için her desenin tarifinden sonra nesneye yönelik bir programlama dilinde örnekler verilmelidir.

Tasarım desenlerinin kullanılması ile elde edilecek kazançlar kısaca sıralanacak olursa;

1. Desenler tek bir sınıf veya bileşenin (component) belirlediğinden daha üst seviyede bir soyutlama sağlar. Bir tasarım deseni problemin çözümünde kullanılan nesnelerin tümünün yapısını ve aralarındaki ilişkiyi belirler.
2. Genel bir tasarım dili tanımlanmasına yardımcı olur. Desenlerin kullanımı problemlere önerilen çözümlerin uzun uzun tanımlanmadan kısaca sadece ismi söylenerek belirtilmesini sağlayabilir.
3. Tasarım desenleri tasarımcılara karmaşık ve heterojen yazılım mimarilerini kurmalarında yardımcı olacaktır. Desenler çoğu zaman binanın inşasında kullanılan yapıtaşları görevini üstleneceklerdir. Bunun yanında çoğu desenin genişleyebilme ve sonradan eklenti yapılabilmesine müsait olması yazılımın ilk versiyonlarının tamamlanmasından sonraki gelişimini rahatlatacaktır. Tabii

desenlerin , çözümün genel yapısını belirleyip özelleşmiş ve detaylı bir çözüm sunmadığını unutmamak gerekir.

Buschmann[3] tasarım desenleri için son ve kapsamlı bir tarifi de verir, buna göre yazılım mimarileri için desen, sürekli ortaya çıkan bir tasarım problemini tanımlar ve bu probleme doğruluğu ispatlanmış jenerik bir çözüm sunar ve bu çözümün elemanlarını , elemanlar arasındaki ilişkileri, sorumluluklarını ve ortak olarak nasıl çalışacaklarını belirler.

Bir tasarım deseninin öğrenilmesi ve gerçek hayatta kullanılabilmesi için tam manası ile anlaşılması gerekir. Bu çoğu zaman desenin tanımının defalarca okunması ve verilen örnek kodların tüm ayrıntıları ile incelenmesi anlamına gelir. Her yıl yapılan Tasarım deseni toplantılarının birinde izleyicilere kimlerin Gamma,Helm,Johnson ve Vlissides'in [2] kitabını satın aldığı ve okuduğu sorulduğu zaman salondakilerin hemen hepsi ellerini kaldırmış, ancak önemli bir tasarım deseni olan Ziyaretçi (Visitor) konusunu kimin anlatabileceği sorulduğunda neredeyse hiçbir el havada kalmamıştır.

Tasarım desenlerinin anlaşılabilmesi için şu yol takip edilebilir.

1. Deseni gözden geçirip genel bir fikir edinilir
2. Desende rol alan yapı ve elemanlar incelenir.
3. Verilen örnek incelenir.
4. Tasarım deseni içerisinde rol alacak nesneler için uygulamadaki rollerine uygun isimler belirlenir. Örneğin İteratör tasarım deseni kullanılacaksa Listelterator, VektorIterator gibi isimler uygun olabilir.
5. Sınıflar tanımlanır.
6. Tasarım desenleri için uygulamadaki rollerine uygun metodların ve operasyonların isimleri belirlenir.
7. Desenlerdeki işlemleri yerine getirecek kodlar yazılır

Tez içerisinde sunulacak olan tasarım desenlerinde Gamma'nın [2] tarif ettiği yol izlenecektir. Buna göre Desenlerin tanımında şu başlıklar kullanılabilir.

- Desen adı ve sınıfı : Denen için açıklayıcı iyi bir isim bulmak çok önemlidir.
- Kullanım amacı: Tasarım deseninin kısaca hangi amaçla kullanılacağını belirler.

- Diğer isimleri: Desen daha önce başka şekillerde isimlendirilmiş ise burada belirtilir.
- Motivasyon: Desenin nerede kullanılabileceğinin anlaşılması için basit bir örnek ve açıklama.
- Uygulanabilirlik: Tasarım deseni hangi şartlar mevcut olduğunda uygulanabilir , Desen ne tür zayıf tasarım parçalarının yerini alabilir.
- Yapı: Desenin grafiksel olarak yapısının belirlenmesi Burada OMT(Object Model Technique) veya UML (Unified Modelling Language) gösterimlerinden biri tercih edilebilir.
- Rol alanlar: Desenin içerisinde tanımlanan nesne ve sınıf gibi yapılar.
- Sonuçlar: Desenin kullanımında meydana açığa çıkacak olan iyi ve kötü sonuçlar açıkça belirtilmelidir.
- Dinamikler: Programın çalışma anındaki (runtime) davranışının incelenmesi
- Kod örneği: Desenin nesneye yönelik bir dilde yazılmış açık ve basit bir örneği . Bu diller C++, Java veya SmallTalk olabilir. Bu çalışmada kod örnekleri genel olarak C++ ve Java dillerinde verilecektir. Örneklerin çoğunluğu GeoCAD CAD sisteminin çekirdek yazılımı üzerinde olacaktır. Örneklerin anlaşılabilirliğini düşürmemek için C++ Kalıp sınıf (Template class) yapısının kullanılmamasına özen gösterilmiştir.
- Varyasyonlar: Desen üzerinde küçük değişiklik ve eklemeler yapılarak oluşmuş varyasyonların ve bunların çözüme neler getirdiğinin incelenmesi.
- Kullanıldığı yerler: Tanımlanan Deseni kullanan mevcut yapı veya uygulamalar listelenir.
- İlgili Desenler: Beraber kullanılabileceği veya yerini alabilecek desenlerin belirlenmesi.

Bir tasarım deseni ne zaman kullanılmamalıdır? Tasarım desenleri genellikle sisteme esneklik ve çeşitlilik getirir, ancak bunun yanında sistemde karmaşıklık ve performans kaybına da sebep olabilirler. Eğer bir desen, getireceği esnekliğe gerçekten ihtiyaç varsa kullanılmalıdır. Bunun belirlenmesinde her tasarımın açıklanmasında belirtilecek olan dezavantajlar ve sisteme getireceği değişikliklerin incelenmesi yol gösterici olacaktır.

1.1 Tasarım desenleri ile ilgili yanlış anlaşılmalar

Tasarım desenleri kısa bir süre içinde dünya çapında büyük ilgi gördüyse de, her yeni şeyde olduğu gibi bu konuda da insanların yanlış anlama ve yanlış yorumlamaları ortaya çıkmıştır.

Vlissides [6] kitabında bu yanlış anlaşılmalardan bazılarını sıralamıştır. Kısaca özetleyecek olursak;

- Bir Tasarım deseni belirli bir alandaki problemin çözümüdür : Bu fikir her ne kadar Cristopher Alexander'in [1] tanımı ile uyuşsa da, Tasarım desenlerini tam olarak anlatmaz. Tasarım desenlerinin sunduğu çözümler tekrar tekrar karşımıza çıkan problemler içindir (recurrance) , ikinci şart belirli bir konudaki çözüümü sunarken, problemin değişik varyasyonları için çözümün nasıl değiştirilebileceğinin de sunulmasıdır. Son şart ise, desenin muhakkak bir adının varolmasıdır.
- Birini görürsen, hepsini görmüş olursun: Tasarım desenleri yıllar içerisinde gelişmiş ve yeni alanlara girip çeşitlenmiştir. Dolayısıyla sadece birkaç örneği inceleyerek tasarım desenleri ile ilgili çıkarımlar yapmak zordur.
- Tasarım desenleri tekrar kullanılabilir yazılımı, verim artışını vs. arttırır: Ne yazık ki tasarım desenleri hiçbir şeyi garantilemez. Yazılım geliştirme sürecindeki insan faktörünün çıkartılması imkansızdır, dolayısıyla anahtar daima insan yeteneği olarak kalacaktır. Tasarım desenleri tasarımcılar ve geliştiricilere sunulmuş bir silah, bir araçtır. Yol gösterir ve doğru kullanılması pek çok sorunu çözebilir. Ancak yanlış kullanılması yarardan çok zarar getirebilir.
- Tasarım desenlerinin herhangi bir kişiye yardımı olduğu ispatlanmamıştır: Tasarım desenleri ilk ortaya atıldığında belki bu konuda gerçekten böyle bir iddia ortaya atılabilirdi. Ancak günümüzde pek çok yazılım tasarım desenlerinden etkili şekilde faydalanmaktadır. Zaman geçtikçe bu tür çalışmaların sayısı da artacaktır.

1.2 Tasarım desenlerinin kategorileri

Gamma, kitabında [2] tasarım desenlerini üç sınıfta incelemeyi uygun görmüştür. Bunlar sırasıyla

- Nesnelerin oluşturulması işlemi için kullanılan desenler (Creational Patterns); Sistemde tanımlı nesnelerin oluşturma (creation) aşamasında sınıf sayısı,

esneklik ve kontrol kolaylığı gibi çeşitli parametrelere göre düzenlenmesini amaçlarlar. Örneğin; Ptototip, Fabrika metodu ve Tekil

- Yapısal Desenler (Structural Patterns): Örneğin, Kompozit ve Dekorator.
- Davranışsal Desenler (Behavioral Patterns) : Örneğin: Komut, Ziyaretçi, Gözlemci

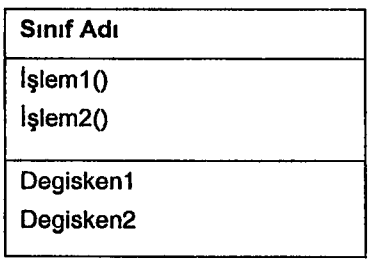
Buna karşın Buschmann, tasarım desenlerini daha farklı açıdan değerlendirmiştir [3]. Bushman'ın sıralaması aşağıdaki şekilde olmuştur.

- Mimari desenler: Sistemin içerisindeki alt birimleri, onların davranışlarını ve aralarındaki ilişkileri belirler. Genel olarak bir mimariyi belirlerler. Örneğin: Katman, Mikroçekirdek vs. [3].
- Tasarım desenleri: Alt sistemlerin ve bileşenlerin veya ilişkilerinin belirlenmesini sağlar. Belli bir konudaki genel bir tasarım problemine haberleşen bileşenlerden oluşan genel bir çözüm sunar [2].
- Deyimler: En alt seviyedeki çözümler sunan tasarım desenleridir. Tasarıma olduğu kadar uygulamaya da girerler ve platforma bağımlı olabilirler. Örneğin Tekil ve referans sayan işaretçiler.

Her iki sıralama da birbirinden farklı yaklaşımlardan çıkmaktadır. Buschmann bütünden parçaya doğru inerken, Gamma Desenleri karakteristikleri ve kullanım alanlarına göre birbirinden ayırmıştır.

1.3 Nesne yapılarının gösterimi

Bir nesnenin nasıl davranacağı onun sınıfı tarafından belirlenir. Sınıf, nesnenin iç yapısını ve yaptığı işleri belirler.



Şekil 1.1 OMT Sınıf Gösterimi

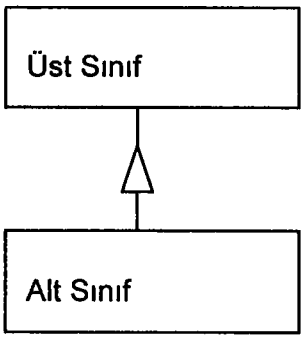
OMT (Object Model Technique)gösteriminde sınıflar dikdörtgen biçimindeki bir kutu ile ifade edilir (Şekil 1.1)

Nesneler bir sınıf şablonundan ortaya çıkan varlıklardır (instances) Bu işleme ilkleme adı verilir (instantiate) nesnenin iç verisi için bellekte yer ayrılması ve bu veri ile metodların ilişkilendirilmesinden ibarettir.

Kesikli çizgili bir ok bir sınıfın başka bir sınıfı ilklediği anlamına gelir (Şekil 1.2).

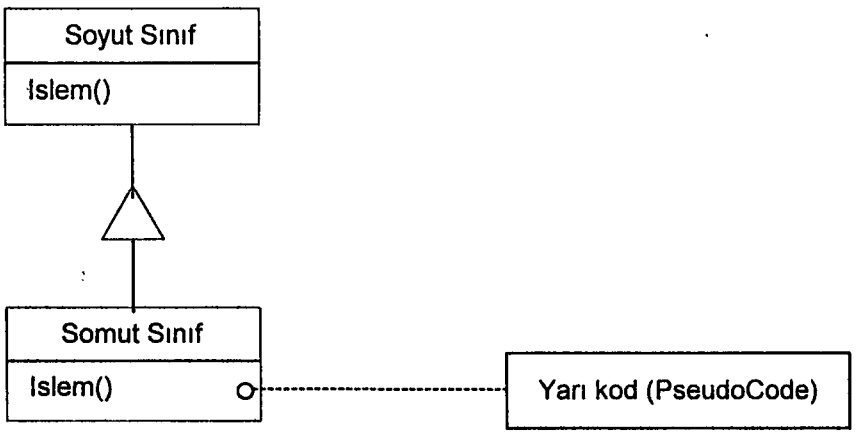
Şekil 1.2 Sınıfların ilklenişi.

Nesneye yönelik tasarımın en önemli özelliklerinden biri de sınıfların başka sınıflardan türetilebilmesi ve türedikleri sınıfın özelliklerini kalıtımla almalarıdır. Bir alt sınıf, üst sınıftan türetildiği zaman alt sınıftan oluşturulan tüm nesneler üst sınıfın özelliklerini ve fonksiyonlarını içerebilir. Bir sınıfın başka bir sınıftan türediğini göstermek için dikey bir çizgi ve bir üçgen kullanılır.



Şekil 1.3 Kalıtım.

Soyut sınıflar alt sınıflar için genel bir arayüz hazırlarlar. Soyut sınıflardan bir nesne ilklenemez, ancak soyut sınıflardan türetilmiş sınıflar ilklenebilirler. Soyut sınıflardaki



Şekil 1.4 Soyut - somut sınıflar ve sanal metodların yeniden gerçeklenmesi

işlemlere de soyut işlem adı verilir. Bu işlemler soyut sınıf içerisinde sadece boş bir kalıp iken türemiş sınıflarda anlam kazanır, buna tekrar tanımlama "override" adı verilir. Fonksiyonların alt sınıflarda tekrar tanımlama yoluyla o altsınıfa has bir davranışa dönüştürülmesi işlemi Nesneye yönelimli programlamanın temellerinden biridir.

Türetme yoluyla oluşturulan sınıflar ortak bir fonksiyonelliğe sahip sınıf ailelerini oluşturur.

Nesneye yönelik tasarımda tekrar kullanılabilirlik (reuse) konusunda çoğu zaman kompozisyon ve kalıtım metodları arasında bir seçim yapmak gerekir. Kalıtım durumunda bir sınıf diğerinin fonksiyonelliğini olduğu gibi alır ve yeni özellikler ekler. Altsınıflar oluşturarak elde edilen tekrar kullanılabilirliğe üst sınıfların özelliklerinin açık ve ortada olması sebebiyle "Beyaz Kutu" adı verilir.

Nesne Kompozisyonu kalıtıma alternatiftir. Bu durumda kompleks bir fonksiyonelliğe erişebilmek için nesneler başka bir nesnenin bünyesinde bir araya getirilerek gruplanır. Nesne kompozisyonu, seçilen nesnelerin iyi bir arayüzü olmasını gerektirir. Kullanılan nesnelerin iç yapısının bilinmemesi ve nesnelere sadece iyi tanımlanmış arayüzlerle erişilmesi sebebiyle bu metoda "Kara Kutu" adı verilmiştir.

İki metot da avantaj ve dezavantajlara sahiptir kalıtım metodunun uygulaması kolaydır ve statik bir yapısı vardır, her şey derleme zamanında halledilir. Ancak üst sınıflardan alınan işlemleri çalışma zamanında değiştirebilme şansımız yoktur. Bundan daha kötüsü, genelde üst sınıfların alt sınıflara ait bazı iç detaylar hakkında bir şeyler bilmesinin gerekmesidir. Nesne kompozisyonunda nesneler birbirlerine

sadece arayüzler vasıtasıyla eriştiğinden iç yapılarının ortaya çıkması gibi bir problem söz konusu değildir. Nesneler arayüzleri aynı olduğu sürece birbirleri ile değiştirilebilirler, bu da esnekliği ve eklenebilir (pluggable) yapıyı oldukça güçlendirir. Ortaya çıkan küçük bir başarım düşmesi de gözardı edilebilr. Bu yüzden kompozisyon, kalıtıma tercih edilmelidir.

2. Temel tasarım desenleri

Takibeden bölümlerde bazı temel tasarım desenleri ayrıntılı şekilde açıklanmıştır.

2.1 İteratör

Bu bölümde "İteratör" tasarım deseni anlatılmıştır

2.1.1 Amaç

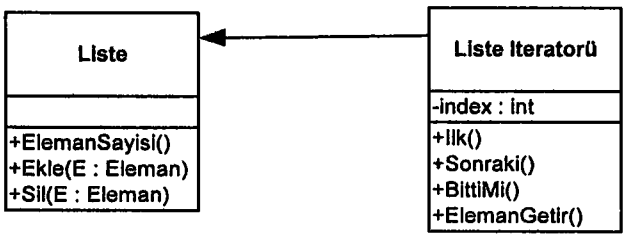
Nesne kümelerinin elemanlarına kümelerin iç yapıları ile ilgilenmeden sıra ile erişimi sağlamak

2.1.2 Diğer İsimleri

Cursor, Sorgulayici, Enumerator

2.1.3 Motivasyon

Liste ve benzeri küme nesneleri içlerindeki elemanlara kümenin iç yapısını bilmeksizin erişebilmek için bir arayüz sağlamalıdırlar. Mesela bir liste ile liste iteratörü arasındaki ilişki aşağıdaki gibi gösterilebilir.



Şekil 2.1 Liste ve Liste İteratörü

Bir Liste iteratörü oluşturmadan önce ona üzerinde tarama yapılacak listeyi temin etmek gerekir. ElemanGetir() metodu listede o an üzerinde bulunduğumuz elemanı getirir, Sonraki imleci listedeki bir sonraki elemana getirir, BittiMi() metodu, listenin bitip bitmediğini kontrol eder.

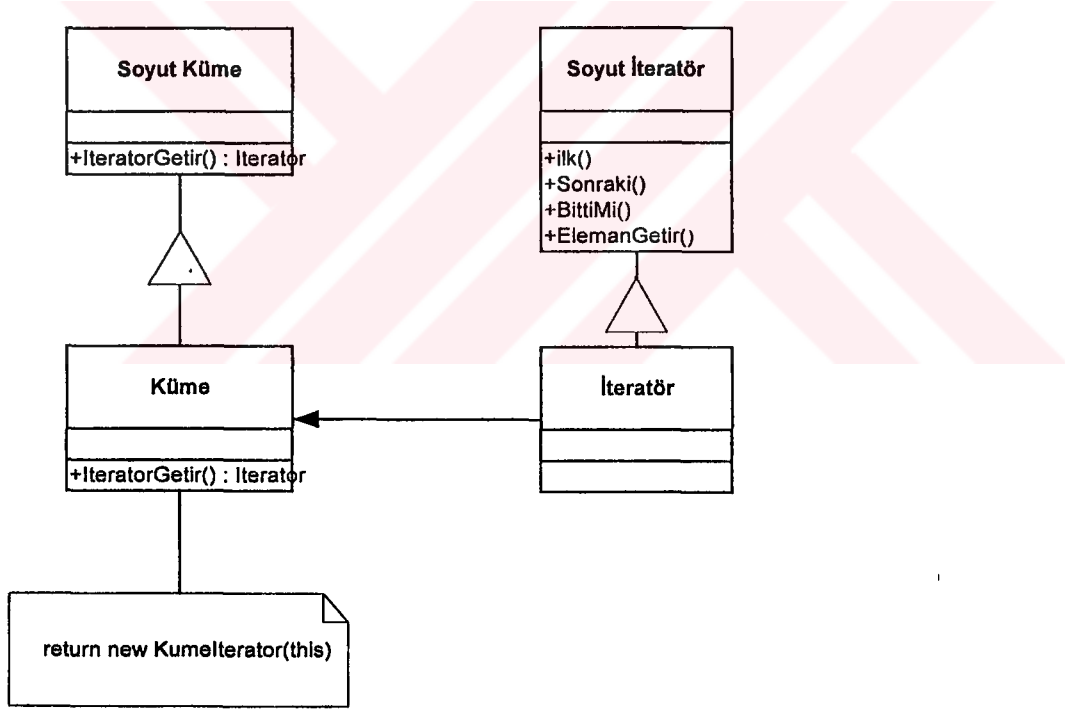
2.1.4 Uygulanabilirlik

İteratör Deseni,

- Bir küme nesnesindeki elemanlara kümenin içyapısını bilmeksizin erişimek gerektiğinde,Küme nesnelerine değişik şekillerde erişim gerektiğinde
- Çeşitli küme yapılarına erişmek için ortak bir arayüz gerektiğinde Kullanılmalıdır.

2.1.5 Yapı

İteratör tasarım deseninin sınıf yapısı ve ilişkileri Şekil 2.2'de görülmektedir



Şekil 2.2 İteratör Tasarım deseninin yapısı

2.1.6 Sonuçlar

İteratörler Kümenin taranmasında çeşitliliğe izin verir. Örneğin bir ağacın taranmasında ters veya düz sıralı taramanın yapılabilmesini sağlar. Yapılması gereken tek şey her bir tarama işlemi için ayrı ayrı iteratör alt sınıfları tanımlamaktır.

Küme yapılarının arayüzlerini basitleştirir. Bir liste, ağaç, vektör veya hash tablosuna olan erişim bu ortak arayüzden yapılabildiği için bu türden yapıları kullanan programlarda alt taraftaki algoritmanın programın geri kalanının değiştirilmesine gerek kalmaksızın olduğu gibi değiştirilebilmesini sağlar.

Bir küme üzerinde birden fazla taramanın aynı anda gerçekleştirilebilmesini sağlar.

2.1.7 Örnek

Standart bir veriyapısı olan vektörler sınırlarını otomatik olarak genişletebilen dizilerdir. Java'nın Vector sınıfı ve Borland VCL'sinin (Visual Component Library) TList sınıfı bu veri yapısının kullanımına örnek teşkil eder. Bir Vektörde temel olarak eleman ekleme, silme ve arama fonksiyonları bulunur. Eğer bir vektör'e yerleştirilmiş olan bütün elemanların herhangibir işlem için taranması gerekirse iteratör tasarım deseninin kullanılması uygun olur. GeoCAD sisteminde kullanılan iteratörler CSorgulayıcı sınıfından türerler.

```
class CSorgulayici   : public CGeoNesne{
  public:
Csorgulayici(){ }
  virtual CGeo* Sonraki()=0;
  virtual bool DahaVar()=0;
 };
```

Kod incelendiği zaman temel iteratör sınıfının iki metod ile basitçe belirlendiği görülür. Başka bir veri yapısının da iteratörü gene Bir Vektör iteratörü ise bu sınıftan türetilir ve sanal olarak belirlenen iki fonksiyonun içini uygun şekilde doldurur. CGeoVektor adındaki vektör yapısı için tanımlanan iteratörün kodu da aşağıda verilmiştir.

```
class CGeoVektorSorgulayici : public CSorgulayici
  {
  private:
   CGeoVektor *V;
   int Ptr;
  public:
   CGeoVektorSorgulayici(CGeoVektor *Vektor) {
    V=Vektor; Ptr=0;
```

```
  }
  virtual CGeoNesne* Sonraki(){
   return(V->ElemanGetir(Ptr++));
  }
  virtual bool DahaVar(){
   return Ptr < V->BoyGetir() ? true:false;
  }
 };
```

Bu tür bir iteratörün kullanımı basittir. Üzerindeki elemanlar taranacak olan kümeden iteratörü istenir, daha sonra standart bir metodla tüm elemanlar taranır.

```
Tara(CGeoVektor *V) {
Csorgulayici *Sorgulayici;
CGeoNesne *Nesne;
Sorgulayici =  V->SorgulayiciGetir();
 while(Sorgulayici->DahaVar()) {
    Nesne = Sorgulayici->Sonraki();
    ...
 }
}
```

2.1.8 Bilinen kullanımlar

C++ standart kalıp dili (STL) iteratör yapısını kullanır.

Java 1.2 sürümünde Iterator sınıfını tanımlamış ve tüm küme benzeri yapılarında kullanıma başlamıştır, Daha önceki versiyonlarında da Enumeration adı ile benzeri bir yapı kullanılmıştı.

GeoCAD sisteminin temel veri yapıları olan CGeoVektor, CGeoStack ve CGeoListe iteratör yapısını yoğun şekilde kullanır.

2.2 Tekil (Singleton)

Bu bölümde "Tekil" tasarım deseni anlatılmıştır

2.2.1 Amaç

Sistem içerisinde bir sınıfın yalnızca tek bir nesne örneği olmasının ve bu nesneye global bir erişim yöntemi sağlanması.

2.2.2 Motivasyon

Bir yazılım sisteminde bazı sınıfların yalnızca tek bir nesne örneği (instance) bulunması gerekir. Örneğin işletim sistemleri yalnızca bir adet pencere yöneticisi ve bir adet dosya sistemi bulunur. Bunun yanında genelde yazılımlarda sadece bir hata giderme mekanizması (yöneticisi) veya kütük tutma (logging) sistemi bulunur. Sistem içerisinde bulunan tüm nesneler bu sınıfın tek bir örneğine erişmeli, ve erişim metodu da aynı olmalıdır.

Bu durumda ilk akla gelen çözüm global değişkenlerin kullanılması olabilir ancak global değişkenlere farklı modüllerdeki sınıfların kolaylıkla erişmesi mümkün değildir ve sistem içerisindeki isim uzayında (namespace) karmaşaya yol açar.

Bu soruna en iyi çözüm sınıfın örneğinin oluşturulmasına ve diğer nesnelerin erişiminde kolay bir yol sağlama konusunda kendi kendisinin sorumlu olmasıdır. Bu desene Tekil (Singleton) adı verilir.

2.2.3 Uygulanabilirlik

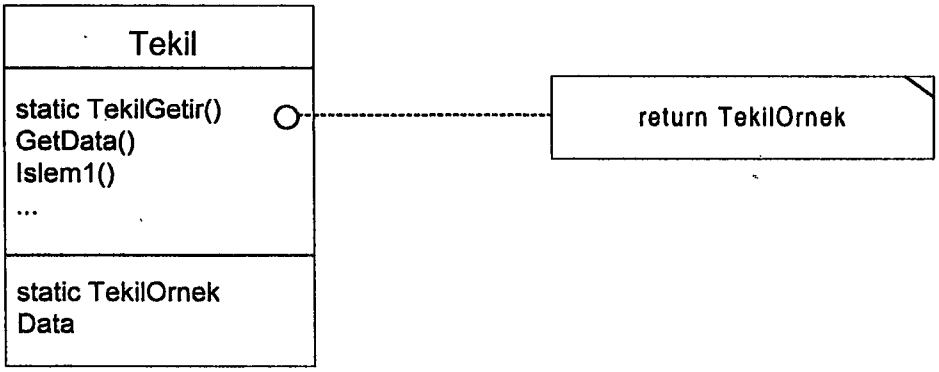
Singleton deseni,

- Sınıfın sadece tek bir örneğinin (instance) bulunması ve bu örneğe tek bir erişim noktası olması gerektiği durumlarda
- Bu tek örneğin alt sınıflar kullanılarak geliştirilmesi durumunda da kullanan sınıfların yapısında değişiklik olmaması istendiğinde

Kullanılmalıdır.

2.2.4 Yapı

Tekil tasarım deseninin yapısı Şekil 2.3'te gösterilmiştir.



Şekil 2.3 Tekil tasarım deseninin yapısı

2.2.5 Sonuçlar

Tekil tasarım deseninin çeşitli faydaları vardır

1. Nesne örneğine olan erişim kontrollüdür. Sınıf, erişimleri kendisi sağladığından kendisine erişen nesneleri kısıtlayabilir.
2. Global değişkenlerin kullanımını azaltır ve programın okunabilirliğini arttırır.

Ancak "Tekil " tasarım deseninin C++'daki kullanımında bazı zorluklar vardır ve statik sınıf metodlarının ve üyelerinin kullanımını gerektirir. Kod örneğinde C++ ve Java'daki Tekil örneklerinde bu konular izah edilecektir.

2.2.6 Uygulama

"Tekil" tasarım deseninin C++ dilinde uygulanmasında bazı dikkat edilmesi gereken noktalar bulunur. Tekil nesneye tek bir erişim noktasının bulunması için Sınıfın bir metodunun çağrılması işimizi kolaylaştıracaktır. Tanımnlanan sınıfın birden fazla örneğini oluşturmada bir kısıtlama olmadığı halde istediğimiz şeyi sağlayabilmek için statik metod (static method) ve statik sınıf üyelerinden (static class member) faydalanabiliriz.

Aşağıdaki örnekte C++'da yazılmış bir Tekil sınıfının yapısını görebiliriz

....

Ancak tekil olarak belirlenen sınıfa erişirken SınıfAdı::ErişimNoktası() ifadesini kullanmak programcı için bazı zorluklar oluşturabilir. Örneğin tekil olarak erişilen sınıfın başka bir metodu çağrılacaksa

```
SınıfAdı::ErişimNoktası()->Metod1()
```

İfadesi kullanılamaz, Bunun sebebi C++'da geriye nesne işaretçisi döndüren metodlardan döndürülen nesneye ait başka bir metodun doğrudan çağrılamamasıdır. Yani yukarıdaki ifdadeyi C++'da yazabilmek için

```
(SınıfAdı::ErişimNoktası())->Metod1()
```

Yazılması gerekirdi. Bu durumdan kurtulmanın çeşitli yolları olabilir, bir tanesi standart C++ makrolarını kullanmaktır. Örneğin

```
#define TEKILSINIF (SınıfAdı::ErişimNoktası())
```

makrosu ile istediğimiz sonuca

```
TEKILSINIF->Metod1()
```

ifadesi ile rahatlıkla varabiliriz.

2.2.7 Kod Örneği

GeoCAD sisteminde tekil sınıfı birkaç ortak nesne için kullanılmıştır. Örneğin sistemde kayıt tutma ve hata ayıklama amacı ile kullanılan konsol'a tüm sınıflardan erilşimin sağlanması gerekiyordu ve sistemde sadece bir tek Konsol bulunması isteniyordu. Bu yüzden konsol yönetici sınıfı yazılırken Tekil yapısı kullanılmıştır.

```
#define KONSOL (CKonsolYoneticisi::KonsolGetir())
class CKonsolYoneticisi : public CGeoNesne{
  private:
   static CKonsolYoneticisi *KonsolSingleton;
  public:
   void print(char *msg);
```

```
    ...
    static CKonsolYoneticisi* KonsolGetir();
};

CKonsolYoneticisi* CKonsolYoneticisi::KonsolSingleton=NULL;
CKonsolYoneticisi* CKonsolYoneticisi::KonsolGetir(){
  if(KonsolSingleton==NULL)
    KonsolSingleton = new CKonsolYoneticisi;
  return KonsolSingleton;
};
```

Bu sayede sistemde konsol'a bir mesaj yazdırmak isteyen sınıflar KONSOL->print("Mesaj") yapısını kullanarak tek bir noktadan bu sınıfa erişimi sağlamışlardır.

2.3 Gözlemci (Observer)

Bu bölümde "Gözlemci" tasarım deseni anlatılmıştır

2.3.1 Amaç

Bir nesne üzerinde oluşan durum değişikliklerini bu durum değişikliğinden haberdar olması gereken diğer nesnelere iletilmesi. Birden çokluya (one to many) ilişkilerin hazırlanabilmesi.

2.3.2 Motivasyon

Bir sistemi birbiri ile ilişki içerisinde bulunan, haberleşen, ortak çalışan sınıflara ayırmanın en belirgin risklerinden biri tutarlılık probleminin ortaya çıkabilmesidir. Bu problemden kurtulmak için sınıfların birbirlerine sıkı bağlarla bağlanması (tight coupling) sistemin esnekliğini azaltacaktır.

Örneğin bir tablolama yazılımında veri üzerinde bir değişiklik yapıldığı zaman veriyi grafik olarak gösteren pencereler veya veri üzerinde matematiksel işlemler yapan diğer süreçlerin bu değişimleri kullanıcıya otomatik olarak yansıtması gerekir.

Çok süreçli bir sistemde süreçler eğer belli zaman aralıklarında işlemler gerçekleştireceklerse ortak bir zamanlayıcıya bağlanmaları ve belirli aralıklarla uyarılmaları gerekebilir.

Gözlemci sisteminin pek çok kullanım alanı olabilir, bu tasarım deseni üzerinde en çok çalışma yapılmış olanlarından biridir ve genel kabul görmüştür. Java programlama dilinde bu işi gerçekleştiren sınıfın adı da "Observer" 'dir.

Gözlemci tasarım deseninde başrolü "Gözlemci" ve "Gözlenen" sınıfları paylaşır. Yukarıdaki ilk örnekte tablolama programında üzerinde değişiklik yapılan hücreler "Gözlenen" değişiklikleri yansıtan grafik pencereleri de "Gözleyen" rolünü üstlenirler.

2.3.3 Uygulanabilirlik

"Gözlemci " deseni,

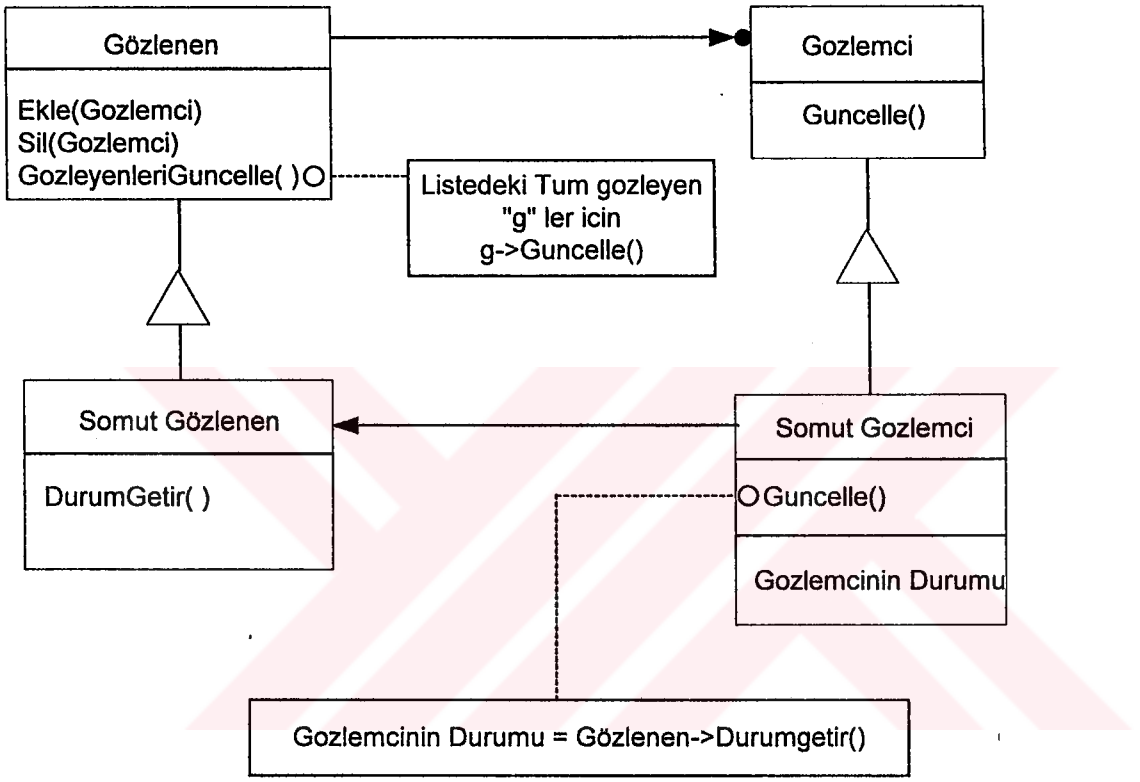
- Bir nesnenin değişikliğe uğraması halinde sayısı başlangıçta bilinmeyen sayıda bağımlı nesnenin bu değişiklikten haberdar edilmesi gerektiğinde

- Bir nesnenin kendi durum değişiklikleri yapılarını ve amaçlarını bilmedikleri diğer nesnelere bildirebilmesi gerektiğinde

Kullanılmalıdır.

2.3.4 Yapı

Gözlemci tasarım deseninin sınıf yapısı ve ilişkileri Şekil 2.4'te görülmektedir



Şekil 2.4 Gözlemci tasarım deseninin yapısı

2.3.5 Roller

- Gözlenen: Kendisini gözleyenleri bilir ve herhangi bir sayıda gözlemci bu nesneyi gözleyebilir. Java Observed sınıfı kendisini gözleyecek olan gözlemci nesneler için bir vektör sınıfını kullanır. Gözleyen sınıfı Gözlemcileri listesine ekleyebilmek, çıkarabilmek ve onları uyarabilmek için gerekli mekanizmaları taşır.
- Gözleyen: Gözlenen'deki değişiklikleri yorumlamak için bir arabirim (metod) taşır.

- Somut Gözlenen: Kendisini gözleyen somut nesneler için durum değişkenlerini sakar ve bu durum değişkenindeki değişmelerden gözleyicileri haberdar etmek gerekli mesaj gönderme sistemini taşır.
- Somut Gözleyici: Gözlediği nesneye bir referans taşır, gözlenen nesnenin durumunu saklar ve Bu durum değişkenlerini Gözleneninki ile tutarlı şekilde tutmasını sağlayacak olan güncelleme metodunu taşır.

2.3.6 Sonuçlar

Gözlemci tasarım deseni gözlenen ve gözeyen nesnelerin birbirinden bağımsız olarak çeşitlendirilebilmesini sağlar. Bunun yanında Gözleyici tasarım deseninin diğer avantaj ve dezavantajlarına göz atacak olursak,

1. Gözlemci tasarım deseninin en önemli getirisi, sınıflar arasındaki sıkı bağları ortadan kaldırabilmesidir (loose coupling). Sistem içerisinde değişik kademelerdeki ve hiyerarşilerdeki nesneler bu tasarım deseni sayesinde birbirlerini kendilerinde ortaya çıkan durum değişikliklerinden haberdar edebilirler.

2. Sınıflar arasında çoklu iletim haberleşmesine imkan tanır (Broadcast communication) Gözlemci tasarım deseninde gözlenen bir nesneye sınırsız sayıda gözleyici kendisini kaydettirip gözlenende meydana gelen değişikliklerden haberdar edilebilir. Ancak bu durumda bazen gözlenende oluşan güncelleme çağrıları istemeden gözleyenleri değiştirebilir, bu durumda gözlenenlerin gelen hangi güncelleme isteğine cevap verip vermeyeceği konusunda kontrollü davranması gerekebilir.

2.3.7 Uygulama

Gözlemci tasarım deseninin uygulanmasında çeşitli problemler karşımıza çıkmaktadır.

1. Gözlemcilerin Gözlenenlere bağlanması:

Gözlemciler normalde gözlenenin içerisindeki bir listeye yerleştirilirler. ancak sistemde çok miktarda gözlenen nesne olması bu listelerin bellekte kapladığı alanın aşırı olarak büyümesine yol açabilir. Bu yüzden gözlenecek ve gözleyecek olan nesnelerin önceden kesin bir şekilde belirlenmesinde fayda vardır.

Bunun bir yolu da sistemdeki gözlemci ve gözlenenlerin erişim hızın yüksek olan ortak bir tabloda belirlenmesidir. (örneğin çırpma tabloları).

2. Birden fazla gözlenen olması durumu:

Bu durumda gözlenen nesnenin kendi tipini veya kimliğini belirtecek bir parametreyi güncelleme sırasında gözleyicilerine göndermesi bir çözüm olacaktır.

3. Gözlenen nesnelerin silinmesiyle gözleyicilerde açıkta kalan referanslar.

Eğer gözlenen nesne silineceği zaman bunu kendisini gözleyenlere iletirse, silinen nesneler için açıkta referans kalmamış olur.

4. Gözlenen nesnelerdeki değişikliğin gözlemcilere aktarılması sırasında fazladan bazı bilgilerin de aynı metot içerisinde iletilmesi gerekebilir:

Bu durumda iki model karşımıza çıkmaktadır. Birincisi İtme (push) modeli: Bu modelde, gözlenen, tüm gözlemcilerine kendisinde meydana gelen değişiklikleri ayrıntıları ile iletir. Çekme (Pull) modelinde ise, gözlenen nesnelere sadece minimum seviyede bilgi gönderilir, eğer gözlemciler arasında değişikliğin ayrıntıları ile ilgilenenler varsa, gözlenen'den bu bilgileri ister. İtme durumunda, Gözlenen, kendisini gözleyenlerin yapısı hakkında birşeyler bilmekte, Çekme metodunda ise, kendisini gözleyenleri bilmemekte, sadece iç durum değişkenlerinin iletilmesi için metodlar taşımaktadır.

5. Güncelleme işleminin performansının arttırılması için:

Gözlemcilerin kendilerini gözlenen'e kaydettirirken, gözlenen üzerindeki ne tür değişikliklerle ilgilendiklerini de belirtmeleri sağlanabilir. Bu sayede, gözlenen , belli bir olay gerçekleştiğinde sadece o olayla ilgilenen gözlemcileri haberdar edecektir.

6. Gözlemci ve Gözlenen sınıflarının birleştirilmesi:

Çoğu durumda bazı nesneler hem gözleyen hem gözlenen rolünü üstlenebilirler, bu durumda pek tercih edilmeyen çoklu kalıtım (multiple inheritance) kullanmak yerine iki sınıfı da tek bir sınıf adı altında birleştirmek uygun olabilir.

2.3.8 Kod Örneği

GeoCAD sisteminde Gözleyici ve gözlenen sınıfları aşağıdaki biçimde tanımlanmıştır:

```
class CGeoGozleyici : public CGeoNesne{
virtual Guncelle(){ }
};
```

```
class CGeoGozlenen : public CGeoGozleyici {
 CGeoVektor *Gozleyenler;
 CGeoGozleyici(){
  Gozleyenler = NULL;
 }
 void Ekle(CGeoGozleyici* Gozleyen){
  if (Gozleyenler == NULL)
    Gozleyenler = new CGeoVektor();
  Gozleyenler -> Ekle (Gozleyen);
 }
 void Sil(CGeoGozleyici* Gozleyen){
  Gozleyenler -> Sil(Gozleyen);
 }
 void GozleyenleriGuncelle(){
  CSorgulayici *Iter;
  Iter = Gozleyen -> SorgulayiciGetir();
  while (Iter->DahaVar){
    CGozleyici *Gozleyen = (CGozleyici *) Iter->Sonraki();
    Gozleyen -> Guncelle();
   }
 }
};
```

Koddan da anlaşılabileceği gibi, GeoCAD sisteminde iki ayrı Gözlemci ve Gözleyen sınıfı olmasına karşın, gözlenen sınıfı Gözlemci sınıfında türetilmiş böylece gözlenen olarak belirlenen sınıfların da gerekirse, gözleyici konumuna geçebilmeleri sağlanmıştır.

Sistemde, gözlenen ve gözleyen olarak belirlenecek olan sınıflar, bu temel sınıflardan türerler ve türemiş gözleyen sınıfları güncelel metodunu tekrar tanımayarak gereken fonksiyonelliğe erişirler.

2.4 Komut

Bu bölümde "Komut" tasarım deseni anlatılmıştır

2.4.1 Amaç

Komut tasarım deseni özellikle etkileşimli arabirimi olan sistemlerde verilen kullanıcı komutlarını temsil eder ve geri alınabilen sistemlerin tanımlanabilmesinde yardımcı olur. (Undo, Redo işlemleri)

2.4.2 Motivasyon

Nesneye yönelik etkileşimli sistemlerde gerçekleştirilen işlemlerin içeriği ile işlemin gerçekleştirilmesine sebep olan isteğin birbirinden bağımsız olması gerekir. Örneğin bir CAD programında ekrandaki araç kutularına fare tuşlarına basarak o anda yapılmakta olan işlemin türü belirlenebilir veya herhangi bir işlemin gerçekleştirilebilir.

CAD programında projeye yeni bir çizgi veya çember eklenecekse ekranda bulunan simgelerine kliklenir ve işlem seçilmiş olur, daha sonra sistem gelen fare ve klavye istekleri o işlemi temsil eden komuta yönlendirilir. Komutun sistem üzerinde gerçekleştirdiği değişiklikler bellekte veya başka bir depolama ortamında saklanırsa daha sonradan bu işlemlerin geriye alınması (undo), veya geriye alınan işlemin tekrar gerçekleştirilmesi (redo) sağlanabilir. Bu amaç için Memento tasarım deseni de kullanılabilir. GeoCAD sisteminde de sınırsız sayıda undo ve redo işleminin gerçeklenmesine imkan tanıyan bir sistem bulunmaktadır.

Komut tasarım deseni yapılan işlemleri programın arabirim bölümünden soyutlar ve bu işlemlerin ayrık bir nesneymiş gibi hareket edebilmesine olanak tanır. Bu yüzden komut tasarım deseni sonradan eklenti yapılabilen (Plug-in enabled) yazılımlar için de bir temel oluşumunu sağlar.

Komut tasarım deseninin temelinde Soyut bir temelkomut sınıfı durur. Bu sınıf sadece sistemde kullanılacak olan gerçek komutlar için bir arayuz tanımlar (interface). Temelde komut tasarım deseni için en önemli olan arayüz metodu "işlet" 'tir (Execute) . sistemde tanımlı olan her bir işlem bu soyut komut sınıfından türetilirler ve her birinin işlet() metodu özelleşmiş bir işi gerçekleştirmek üzere gerçeklenir.

İşlemin gerçekleşmesi sırasında kullanıcıdan girişle okunuyorsa bu arabirim komutlarını karşılamak için gene özelleşmiş Olay işleyiciler de Sınıf hiyerarşisinin tepesindeki soyutkomut sınıfına eklenebilir. Bu metodun alt sınıflarda işlenmesi sırasında gelen olayın türüne göre komut'un işleyiş basamaklarında ilerlenilir.

Gamma[2] Birden fazla komutun art arda işlenmesi gereken durumlarda soyut komut sınıfından içerisinde bir komut serisi taşıyan makro komutları türetilebileceğini belirtmiştir. Bu makro komutunun işlet() metodunun görevi ise bünyesinde barındırdığı tüm komutların işlet() metodlarını çağırmaktan ibarettir.

2.4.3 Uygulanabilirlik

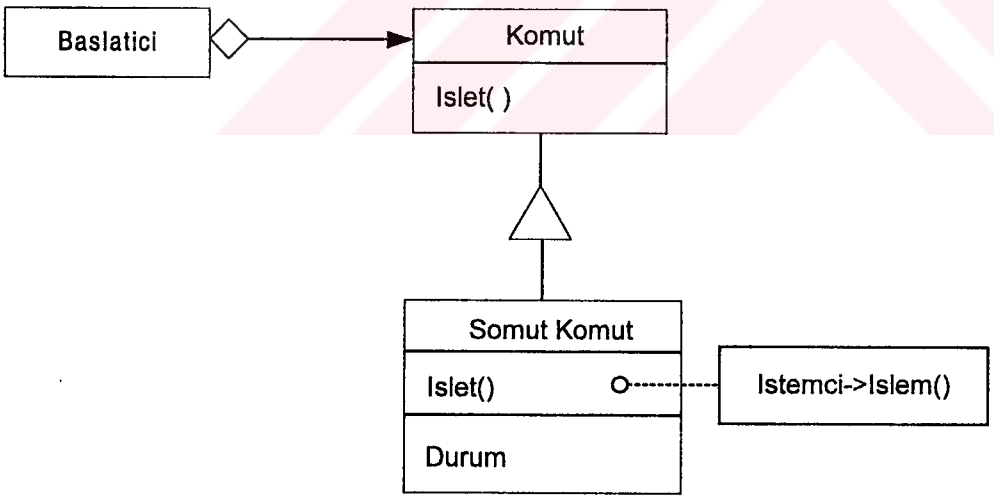
Komut tasarım deseninin kullanılabileceği yerler;

- Gerçekleştirilmesi gereken işlemlerin parametrize edilmesi gereken yerlerde kullanılabilir. Bu durumda komut tasarım deseni, yapısal dillerde benzeri bir mekanizmayı kurmakta kullanılan callback fonksiyonlarının görevini yerine getirir. Callback fonksiyonları tanımlandıktan sonra programda belli bir işlem yerine getirileceği zaman çağrılmak üzere sisteme kaydedilen (register) fonksiyonlardır. Aynı prototipe sahip fonksiyonlar yazılır ve sistem içerisinde çağrılması gereken yerlere sonradan fonksiyon işaretçileri (pointer) yardımı ile yerleştirilirler. Yapısal dillerdeki Eklenti (Plug-in) sistemleri de bu prensibe dayalıdır. Internet tarayıcıları(browser), Photoshop ve winamp gibi popüler programların plug-in yapıları da bu basit sisteme dayalıdır. Üreticiler SDK' larında (Software Developers Kit, Yazılım geliştiricisi araçları) parametreleri düzenlenmiş fonksiyon prototipleri belirlerler ve bu fonksiyonlarda kullanılmak üzere bir araç kutusu niteliğindeki çekirdek fonksiyon kütüphanesini de temin ederler. Eklenti geliştirmek isteyen kullanıcı bu prototipe uygun bir fonksiyon yazar fonksiyonunu ve sonradan yüklenebilir bir dinamik kütüphaneye dönüştürür. Bu kütüphane eklentilerin yükleneceği ana program tarafından istenirse yükleme başlangıçta, istenirse de programın çalışması anında kullanıcı tarafından yüklenir ve programa eklentinin fonksiyonelliği de katılmış olur. Komut tasarım deseni bu işlemin daha düzenli ve kontrollü şekilde gerçekleştirilmesini sağlar.
- Komut tasarım desenleri sistemdeki işleri yürüten bağımsız nesneleri ifade ettiğinden bu nesneler gerekirse uygulamalar arasında taşınabilir ve gerekirse sistemde birden fazla komut aynı anda işletimde kalabilir. Özellikle eşzamanlı sistemlerde bu özellik işe yarayabilir.

- Geri alınabilirlik (Undo) desteği. Sistemde gerçekleştirilen işlemler birbirinden bağımsız nesneler tarafından temsil edildikleri için, işlemlerin geriye alınmaları için gerekli işlemler de komut nesnesinin bir metodu olarak tanımlanabilir. Yani her komut yaptığı işi geri alabilmek için bir gerial() - undo() metodunu da taşır. Gerial metodunun işletilebilmesi için işlemin gerçekleştirilmesi sırasında ortaya çıkan verilerin bir tamponda saklanması gerekir. Ancak undo sisteminin komut tasarım deseni içerisinde işletilebilmesi konusunda bazı sorunlar ortaya çıkmaktadır, bu konu uygulama bölümünde detayları ile tekrar incelenecektir.
- Sistem olay kütüğü tutulmasında kolaylık sağlar. Eğer komut tasarım desenine yükle ve kaydet gibi metodlar eklenirse hem işletilen komutların sırası ve durumları sonradan incelenebilmek üzere kaydedilmiş olur, hem de bir çökme durumunda komutlar tekrar işletilerek kurtarma işlemi gerçekleştirilebilir. Komut tasarım deseni ardarda birden fazla komutun işletimine de olanak sağlayarak işlem (transaction) benzeri bir yapını da oluşturulabilmesine imkan tanır.

2.4.4 Yapı

Komut tasarım deseninin sınıf yapısı ve ilişkileri Şekil 2.5'de görülmektedir



Şekil 2.5 Komut tasarım deseninin yapısı

2.4.5 Sonuçlar

Komut, özellikle CAD benzeri interaktif sistemlerde kullanılması neredeyse zorunlu bir tasarım deseni olarak karşımıza çıkar.

Komut tasarım deseni bir işlemin yapılması gerektiğini belirten nesne ile işlemi gerçekleştiren nesneleri birbirinden ayırır [2].

Tüm komutlar farklı birer nesne olduklarından yeni komutlar eklemek ve esnek sistemler oluşturmak çok kolaydır.

Birden fazla komutu birleştirerek kompozit komutlar oluşturulabilir. Burada kompozit tasarım deseninden faydalanılabilir.

Komut sınıflarından yeni komutlar türetilebilir veya içerisine fazladan fonksiyonellik katılabilir.

2.4.6 Uygulama

Komut tasarım deseni için [3]'te soyut komut sınıfından türemiş olan somut komut prototiplerini taşıyan ve gerektiği zaman kullanıma sunan bir Komut kontrolcüsü de sunulmuştur.

Komut kontrolcüsü, bir olay döngüsü içerisinde kullanıcıdan gelen istekleri (bir menü veya kısayol kombinasyonu vasıtasıyla) karşılar ve gereken komutu seçer. Daha sonra seçilen komutun islet metodu çağrılır ve gerial-tekrarla (undo-redo) işlemlerinin yapılabilmesi için bir komut yığınına koyar. Komut işletim esnasında gerial işlemi için gerekli olan durum bilgisini belleğe depolar ve işletimini tamamlar. Komut kontrolcüsü bir undo işlemi isteği aldığı zaman yığının üstündeki son komutu alır ve gerial metodunu çağırır, komut, önceden depolamış olduğu durum bilgisini kullanarak gerial işlemini tamamlar.

Bu yaklaşımın CAD uygulamalarında tam olarak kullanılamayacağını çalışmalarımız sırasında gözlemledik. Bu sebepleri şöyle sıralayabiliriz.

- Sistem, komut nesnelerini bellekte bulunan bir yığında depoladığı için gerial işlemi derinliği kısıtlıdır.
- Komut nesnesinin kendisi de yığında depolanmaktadır, komut nesnesinin kendi iç değişkenleri büyük bir fazlalık olarak görünmese de yapılacak olan toplu bir değişiklikte oluşacak binlerce komut nesnesi sisteme aşırı yük getirebilir.
- Sistemin kullanıcı etkileşimli olay işleme (event handling) mekanizması komut tasarım deseninde açıklanmamıştır.
- Sistem bir anda sadece tek bir komutu işleyebilir. İşletilen komut işletimin tamamlanmasının ardından silinmektedir.

Bu problemleri çözmek için öncelikle kullanıcıdan gelen interaktif komutlara (fare, klavye vb.) cevap verebilecek başka bir sınıf tanımlama ihtiyacı ortaya çıkmaktadır. [1]'de bu iş için kullanılabilecek araç sınıflarından özet olarak bahsedilmiştir. CAD sistemlerinin ihtiyaçlarının bilinmesi, bize problemin çözümünde de yardımcı olacaktır.

2.4.6.1 CAD sistemlerindeki durum

CAD sistemlerinde, kullanıcı etkileşimli işlemler iki aşamada gerçekleştirilirler. Birinci aşamada, kullanıcı yapacağı iş için gereken bir aracı seçer ve o işlem için gerekli olan parametreleri fare veya tuş takımı gibi bir giriş aracı yardımı ile belirler. Örneğin çember çizmek isteyen bir kullanıcı, önce menüden çember çizme işlemini seçer, daha sonra, fareyi kullanarak önce çemberin merkezini belirleyip farenin düğmesine bir kez tıklar, daha sonra farenin hareketleri çemberin yarıçapını belirlemek için kullanılır. Fare tuşunun ikinci kez tıklanması ikinci aşamasını, yani çember nesnesinin oluşturulup CAD sistemindeki gerekli veriyapılarına ve gerial sistemine eklenmesi işlemini başlatır.

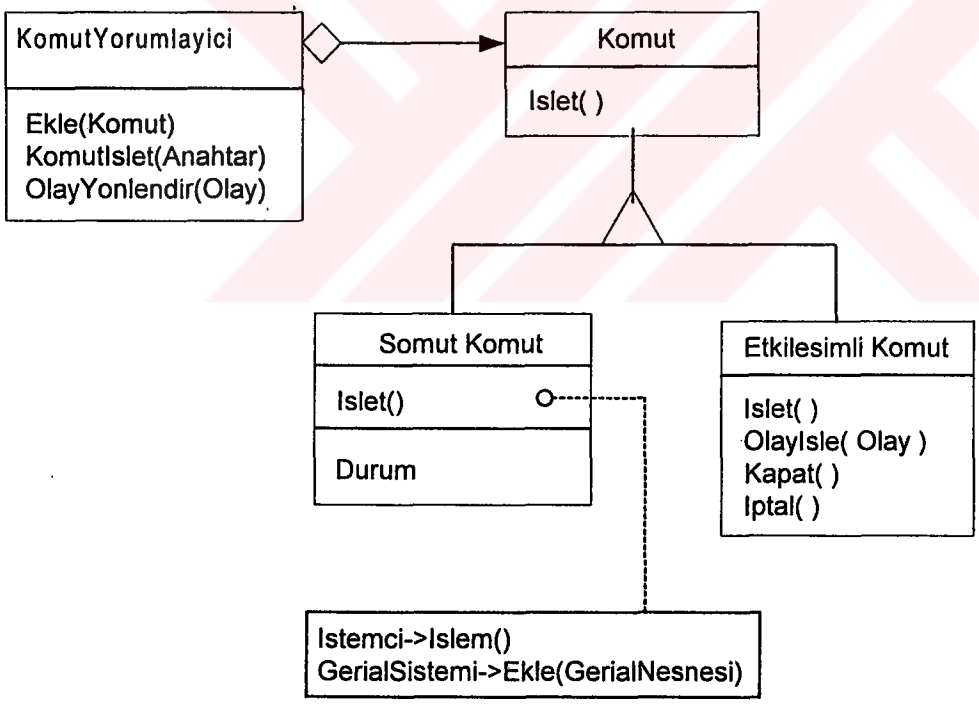
Bu iki aşamanın birbirinden ayrılması programlama kolaylığı ve sistemin karmaşıklığının azalması açısından doğru bir yaklaşım olacaktır. Vlissides de, araç sınıfları ile komutların işletimini yapan sınıfların birbirinden ayrılması gerektiğinden bahseder [6]. GeoCAD sisteminde de işlemin gerçekleştirilmesi sırasında parametrelerin elde edilebilmesi için kullanıcıdan gelen interaktif komutların işlendiği sınıflarla, komutların undo sistemine kaydedilmesini sağlayan sınıflar birbirinden ayrılmıştır. Sistemde tüm işlemler soyut bir araç sınıfından türemişlerdir.

Bir CAD sistemindeki temel işlemler çeşitli gruplarda incelenebilirler; basit veya karmaşık şekillerin oluşturulması işlemleri, nesnelerin seçilmesine yönelik işlemler, nesnelerin özelliklerinin değiştirilmesine yönelik işlemler ve projenin görsel durumunun değiştirilmesine yönelik işlemler, örneğin ekrandaki görüntünün büyütülmesi (zoom) , ekranın kaydırılması (pan) veya çizim tabakalarından bazılarının gizlenip tekrar görünür hale getirlimesi.

Büyütme ve kaydırma işlemlerinin o anda aktif olan işlemi iptal etmemesi iyi bir yaklaşım olabilir. Kullanıcı, çizim yaptığı alanı daha iyi görebilmek için o an gerçekleştirmekte olduğu işlemi kapatmadan büyütme işlemi yapmak isteyebilir. Başlatılmış bir işlemin bu nedenle kapatılması değerli kullanıcı zamanının israfı

olacaktır. Bu durum, komutların ana ve alt komutlar olarak ayrılması gerektiğine işaret etmektedir.

Bu konudaki bir başka nokta da bazı işlemlerde gerekli olan yardımcı işlemlerin gerekliliğidir. Yardımcı işlemler genelde çizim sırasında kullanıcıya fazladan seçenekler sunarlar, örneğin ekranda bir doğru çizen kullanıcı, bu doğrunun seçeceği bir başka doğru ile aynı boyda, veya o doğruya dik doğrultuda olmasını isteyebilir, bu hedef doğrunun gösterilmesi de bir başka interaktif işlemdir. Kullanıcı doğru çizme işlemine başladıktan sonra özel bir tuş kombinasyonuna basarsa bu yardımcı işlemlerden birinin başlatıldığını düşünelim. Çalışmaya başlayan yardımcı işlem kullanıcıdan gelen fare işaretlerini yorumlamaya başlayacak ve işlemi tamamlandığında (bizim örneğimizde ona dik olarak uzanmasını istediğimiz doğru seçildiğinde) önceden başlatılmış olan doğru çizme işleminin seçilen doğruya dik olarak, belirtilen koordinatlara uzaması ve işlemin tamamlanıp doğrunun sisteme eklenmesi gerekir. Bu tür yardımcı komutların yardım etmek için çağrıldığı ana komuta nasıl mesaj göndereceği gene komut sistemi tarafından belirlenmelidir.



Şekil 2.6 GeoCAD sistemi komut sınıfları

Vlissides [3]'de CAD benzeri kullanıcı etkileşimli sistemler için ziyaretçi (visitor) [2] ve komut (command) [2] tasarım desenlerini kullanan bir sistem önermiştir. Öneride, önce bir araç sınıfı belirlenir ve oluşturma, seçme, döndürme gibi grafik nesneleri üzerinde uygulanacak olan özelleşmiş araç sınıflarını bu ana soyut sınıftan türetir ve türetilen bu araç sınıflarını bir "ziyaretçi" olarak kullanır. Ziyaretçi tasarım

deseni alt sınıfların miktarının az olmasını sağlamaktadır, yani her grafik nesnesi için bir ekle, seç ve döndür işlemi olmayacaktır, onun yerine her araç içerisinde grafik nesneleri için farklı birer metod bulunacaktır. Bu yaklaşım her ne kadar alt sınıfların miktarını azaltsa da sistemin esnekliğini de sınırlar, Ziyaretçi tasarım deseninde temel sınıflar kendilerinden türeyecek olan alt sınıfların ne olduğunu bilmek zorundadır. Bu yüzden "ziyaretçi" tasarım desenini GeoCAD sisteminde kullanmamayı tercih ettik.

Her nesne için bir oluşturma, döndürme veya seçme sınıfı yapmak yerine, grafik nesnemize temelde bu metodların hepsini koyarak ve türeyecek olan sınıfların (Doğru, çember ve nokta gibi) herbirinde bu sanal (virtual) metodun grafik nesnesinin kendine has özellikleri dikkate alınarak gerçeklenmesi sağlanmıştır. Bu sayede hem alt sınıfların miktarı kısıtlanmış hem de gerekli esneklik sağlanmıştır.

2.4.6.2 Gerial – Tekrarla (Undo,Redo) işlemi ve gerçeklenmesi

Kullanıcı ile etkileşim içerisinde bulunan yazılımların hemen hepsi, yapılan hataların geriye alınabilmesi (undo) , veya geri alınan işlemin tekrar yapılabilmesini (redo) sağlayan seçenekleri kullanıcılara sunar. Özellikle kelime işleme, resim işleme ve CAD programlarında yapılan işlemin geriye alınabilir olması hayati önem taşımaktadır.

Geriye alma işleminin ayrıntılarına girmeden önce gerialma seçeneğini sunan sistemlerdeki yapıların türlerinden bahsedebiliriz. Bazı sistemler kullanıcının sadece yapılan son değişikliğin geriye almasını sağlar, biraz daha gelişmiş yazılımlar bunu bir aşama ileri götürerek sınırlı sayıda işlemin geriye veya ileriye alınmasını sağlarlar. Tam bir geri alınabilirlik için geri alınma işleminin sınırının olmaması yani sadece sistemdeki depolama alanının bu geri alınabilirlik derinliğini sınırlaması gerekir. Bazı geri alınabilir sistemler bir seferde birden fazla işlemi geri alabilmek için çoklu seçim yapılabilen bir tampon (history buffer) sunarlar.

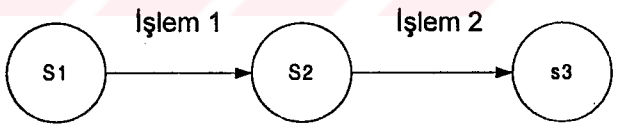
En uç geri alınabilir sistemler ise geçmişte yapılan işlemlerin belli bir bölümünü o işlemlerden sonra gerçekleşmiş işlemlere zarar vermeden iptal edebilmektir. Örneğin bir CAD uygulamasında, ekrana bir çember, daha sonra bir çizgi ve bir nokta koyduğumuzu düşünelim. eğer ilk olarak gerçekleştirilen çember çizme işlemini iptal etmek istersek üç kez gerial (undo) yapıp o çemberin oluşumunu iptal etmemiz, yani çemberin silinmesini sağlamamız gerekir. Ancak bu durumda çemberden sonra çizdiğimiz nokta ve çizgi de gerial işlemleri sırasında silinmiş olacaktır. bu noktada tekrar yap (redo) seçeneği sildiğimiz çemberi de geri getireceğinden işe yaramaz.

Önceden gerçekleştirilmiş işlemlerin daha sonrakileri etkilemeden iptal edilmesini sağlayacak mekanizmalar tutarlılık problemleri ile karşılaşabilirler. yukarıda verilen örnekteki CAD sistemin bu tür bir gerial yapısını desteklediğini varsayalım. Bu sefer gene ekrana önce bir çember, sonra bir doğru ve bir nokta çizelim ve daha sonra çemberin yarıçapını değiştirelim. Sadece ilk çember ekleme işlemini gerialmak istersek, daha sonradan o çemberin rol aldığı tüm işlemlerin de iptal edilmesi gerekir. Bu yüzden bu tür gerialma sistemleri uygulama için çok pratik değildirler.

Gerialma işlemi (undo) yazılımda o anda ve geçmişte yapılmış olan işlemlerin sitemde yaptığı değişiklikleri ters olarak tekrar yapmakla aynı şeydir. Örneğin ekrandaki görüntü %10 büyütülmüşse bu işlemin geriya alınabilmesi için ekrandaki görüntünün %10 küçültülmesi gerekir. Eğer yapılan işlem bir çemberin ekranda çizilmesi ise, o zaman bu işlemin geriye alınması o çemberin silinmesinden ibaret olacaktır.

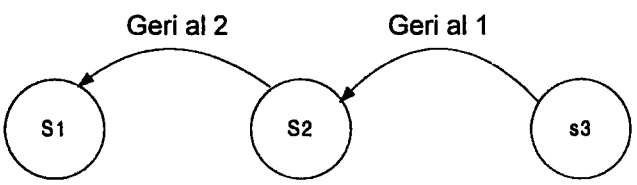
Tekrarla (redo) işlemi ise, geriye alınmış olan bir işlemin tekrar gerçekleştirilmesi, yani gerial işleminin geri alınması demektir.

Yazılımın işletilmesi sırasında gerçekleştirilen işlemlerin çoğu sistem durumunu değiştirir, yani sistemdeki parametrelerin değerini değiştirir veya yeni nesnelerin eklenmesini veya çıkarılmasını sağlar. Bu durumu Şekil 2.7 deki durum diyagramı ile gösterebiliriz.,



Şekil 2.7 Sistemin işlemlerle durum değiştirmesi

Sistemde gerçekleştirilecek bir gerial işlemi, önce şu anda bulunduğu S3 durumundan S2 durumuna getirmelidir. Daha sonra yapılacak olan başka bir gerial

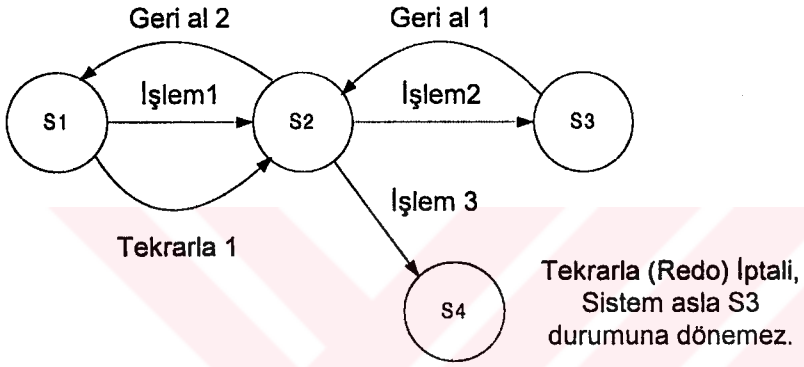


Şekil 2.8 Gerial (undo) işlemleri

işlemi de sistemi ilk durumu olan S1'e getirecektir (Şekil 2.8).

Yapılan bir hatanın düzeltilmesi için yapılan gerialınma işleminden vazgeçilmesi işlemine tekrarla (redo) dendiğini belirtmiştik. Ancak tekrarla işlemi konusunda dikkat

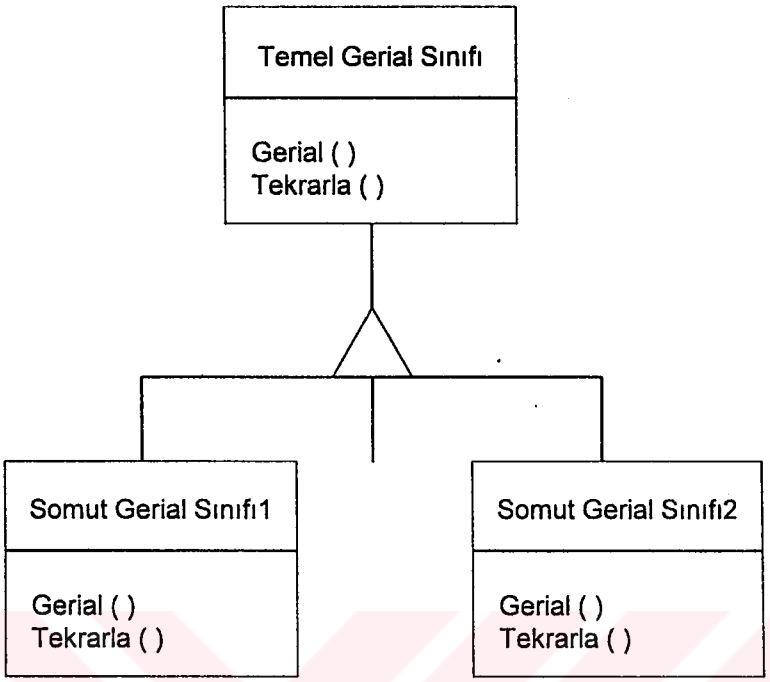
edilmesi gereken bir nokta sistemde yapılan herhangibir durum değiştirici işlemin sistemde daha önce yapılmış olan gerial işlemlerinin tekrarla işlemi ile iptal edilemeyeceğidir. Bu duruma tekrarlama iptali (redo cancellation) adı verilir. (Şekil 2.9)



Şekil 2.9 Tekrarala İptali (redo cancellation)

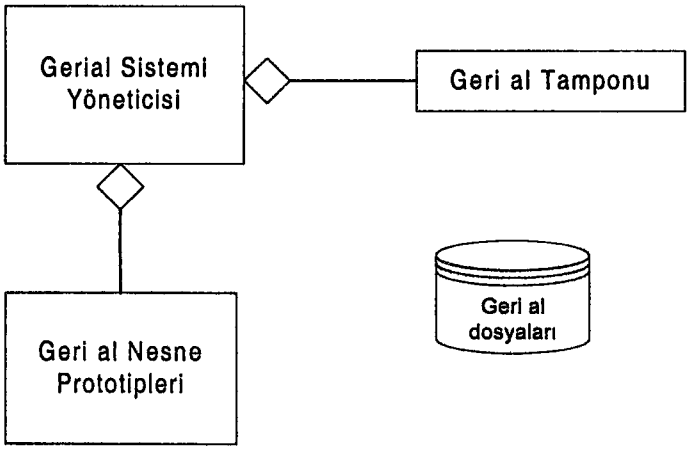
Şekil 2.9'daki durumda sistem bir gerial komutu ile S3'ten önce S2'ye daha sonra da S1 durumuna geri dönmüştür. Daha sonra son yapılan gerialma işleminden vazgeçilmiş ve sistemi daha önceden S1'den S2 durumuna getiren işlem tekrarlanmış, yani "tekrarla" işlemi gerçekleştirilmiştir. Ancak bu noktada yeni bir işlem yapılarak sistem yeni bir S4 durumuna gelmiştir. Bu noktadan sonra Sistem gerial ve tekrarla işlemleri ile S3 konumuna artık geri getirilemez.CAD yazlımları sürekli olarak kullanıcı ile etkileşimde olan sistemlerdir. Yapılan hataların gerialma komutları ile düzeltilebilmeleri çok önemlidir.

GeoCAD sisteminde gerial işleminin gerçekleştirilebilmesi için, önce temel bir Cundo nesnesi tanımlanmış ve bu nesneden sistemin durumunu değiştirecek tüm işlemler için yeni bir Undo sınıfı türetilmiştir, örneğin çember ekleme işlemi için bir CundoCemberEkle, Çember silme işleminin geriye alınması için bir CundoCemberSil, Zoom işlemi için CUndoZoom ve Seçilmiş bir grup nesnenin silinmesi işleminin geriye alınması için de CUndoGrupSil gibi.



Şekil 2.10 Gerial (undo) sınıfları

Gerialma işleminin sınırsız sayıda gerçekleştirilebilmesi için yeni işlemler yapıldıkça, yapılan işlemin geriye alınması veya o işlemin tekrarlanmasına ilişkin bilgilerin bir tampon bellekte depolanması gerekir.



Şekil 2.11 Gerial sistemi

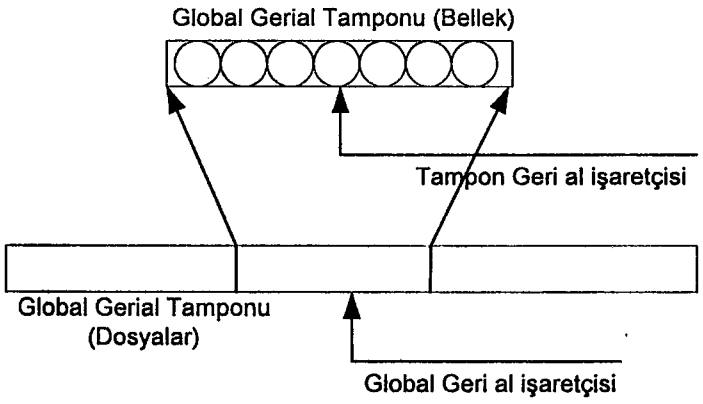
Sistemin çalışma ilkesi şu şekilde olacaktır. Programda yapılan ve programın kullandığı veri yapısı üzerinde değişiklik yapan (durumunu değiştiren) tüm işlemler

için bir GeriAl nesnesi oluşturulup Gerialma işleminden sorumlu bir tampon yöneticisine gönderilir. Tampon yöneticisi bellekte belli miktarda Gerial nesnesini tutabilmektedir. Bellekteki grial nesneleri için ayrılan alan dolduğu zaman bellekte bulunan tüm gerial nesnelerinin içeriği blok şeklinde önceden açılmış olan bir dosyaya kaydedilir ve GeriAl nesne tamponu temizlenir.

Eğer kullanıcı bir gerialma işlemi yapmak isterse, sistem önce bellekteki gerial nesneleri tamponuna bakar ve sistemin o an bulunduğu durum'u (state) gösteren işaretçinin gösterdiği Gerial nesnesini alarak "GeriAl( )" metodunu çağırır. Eğer kullanıcı arka arkaya gerial işlemleri yaparak tamponun en başına kadar gelmişse, o an tamponda bulunan tüm GeriAl nesneleri aynı dosyanın sonuna eklenir ve dosyadan en son kaydedilmiş olan gerial nesnelerine ait bloğu tekrar belleğe okunup işleme devam edilir.

Bu işlemin daha iyi anlaşılabilmesi için somut bir örnekle açıklayacak olursak; CAD programımızı kullanan kişi, projesine bir çember ,bir doğru ve bir nokta eklesin. Bu işlemlerin herbirinin ardından gerekli GeriAl nesneleri eklenen nesnelerin parametreleri ile oluşturulup GeriAl sistemine gönderilir. Parametrelerin belirlenmesi ya GeriAl nesnelerin ilklendirilmesi sırasında (Construction) ya da ilklendirildikten sonra özel olarak parametrelerin yerleştirilmesi ile yapılabilir.

Daha sonra tampona yerleştirilen bu nesneler GeriAl işlemi nesnelerin içindeki gerekli metod (GeriAl( )) çağrılarak gerçekleştirilir.

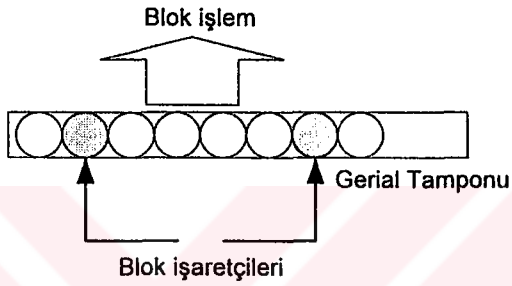


Şekil 2.12 Global gerial tamponu

Özetlemek gerekirse Geri al sistemi o an hafızada bulunan GeriAl nesneleri ile önceden diske yazılmış olan gerial nesnelerinin tamamını dev bir tampon olarak görmektedir. Gerialma ve tekrarlama işlemlerinde kullanmak amacıyla disk

tamponu üzerinde evrensel bir işaretçi kullanılmıştır. Bu işaretçinin değeri sisteme her yeni GeriAl nesnesi eklendiğinde artmakta, işlemler Geri Alındıkça da azalmaktadır.

Bu sayede sistem elinde sadece disk alanı ile sınırlandırımış miktarda bir gerial nesne tamponunu görmekte ve işlemler çoğu zaman bellekteki tampon üzerinden gerçekleştiği için çok miktarda gerial – tekrarla işlemi yapılsa bile sistem performansına olan etki minimuma indirilmiş durumdadır. Yapılan testlerde 128 veya 256 GeriAl nesnesinin bellekte saklanmasının en uygun sonuçları verdiği görülmüştür.



Şekil 2.13 Blok işlemlerin geri alınması

Ancak Gerialma işlemi ile ilgili iki sorun daha vardır. Bunlardan birincisi bazı işlemlerin birden fazla işi birden yapmasıdır, örneğin ekranda seçilmiş olan birden fazla nesnenin silinmesi ve sonra bu ilemin geri alınmak istenmesi durumunda çoklu GeriAl işleminin yapılması gerekecektir. Bu amaç için GeoCAD programında gene temel CGeriAl sınıfından türetilen CGrupIsaretcisi nesnesi kullanılmıştır. Çoklu bir işlem yapılacağı zaman, GeriAl sistemi önce bu boş işaretçi nesneden oluşturup sisteme kaydeder, daha sonra da diğer işlemler için GeriAl nesnelerini oluşturup sisteme ekler. İş tamamlandığı zaman tekrar bir işaretçi nesne oluşturup sisteme kaydeder ve işlemi tamamlar. Herhangibir gerial işlemi GeriAl Tamponunun gösterdiği yerde işaretçi nesneye rastlarsa, başkabir işaretçi nesne gelene kadar tampondaki nesneleri alıp onların GeriAl( ) metodlarını çağırır, kısacası ardarda yapılmış olan tüm işlemleri geri alır.

GeriAl işlemi ile ilgili bir başka sorun da sistemde yapılan bazı işlemlerde işlemin geriye alınması için gerekli veri miktarının belirsiz olmasıdır. Örneğin programda ekrana uzunluğu her seferinde değişen yazılar yazılması veya kenar sayısı başlangıçta belli olmayan poligonlar çizilmesi gibi. Bu durumlarda GeriAl

nesneleri'nin boyları da sabit olmayacağından biraz önce anlatılan şekilde depolanamayacaklardır. Bu yüzden Temel GeriAl sınıfından ve belirsiz uzunluklu nesneler için temel sınıf görevi yapacak yeni bir soyut sınıf türetilir. Bu yeni sınıfı kullanabilmek için, boyutu belli olmayan veriler için GeriAl nesneleri oluşturulmadan önce veri paketlenir ve adresi nesnenin sağladığı bir metod yardımı ile belirlenir. Daha sonra nesne GeriAl tamponuna veya diske yazılırken, nesnenin kendi iç bilgilerinin hemen ardından bu veri paketi de yazılır. Nesnenin içerisindeki alanlardan birinde bu alanın boyu bilgisi bulunduğundan okuma ve yazma sırasında herhangibir sorun da ortaya çıkmaz.

2.4.6.3 Eklenti Yapılabilir Uygulamalar

Sonradan eklenti yapılabilirlik (pluggable architecture), özellikle son dönemde yazılım dünyasında neredeyse tüm uygulamalar için vazgeçilmez bir özellik olmaya başlamıştır.

Bir yazılıma sonradan eklenti yapmak için yorumlanan (script) diller veya özel olarak eklenti programcıları için hazırlanan yazılım geliştirme kiti (SDK) kütüphaneleri kullanılarak hazırlanmış dinamik olarak yüklenen kütüphanelerdir. Dinamik kütüphanelerin çalışma zamanında (run-time) yüklenebilmesi Unix ve Windows işletim sistemlerinde mümkündür.

Hazırlanan kütüphane eklenti (plug-in) için bir standart giriş fonksiyonu içerir. Sistem eklentiyi yüklerken ilk önce bu giriş fonksiyonuna girer ve programlar arasında önceden belirlenmiş olan bir protokole göre ya yeni fonksiyonun bir işaretçisi ana programda belirlenir ya da GeoCAD sisteminde olduğu gibi önceden ana programda belirlenmiş olan soyut bir sınıftan türeyen yeni sınıf Ana sistemdeki bir prototip listesine kaydedilir.

2.5 Prototip

Bu bölümde "Prototip" tasarım deseni anlatılmıştır

2.5.1 Amaç

Oluşturulacak nesnelerin türlerini belirlemek için bir prototiplerinin kullanılması ve bu nesnelerin oluşturulması için prototiplerinin bir kopyesinin alınması.

2.5.2 Motivasyon

Bir CAD programının temel öğeleri ekrana çizilen ve kullanıcının üzerinde çalıştığı nesnelerdir. Eğer, hem bu nesne türlerinin genişleme ve sonradan sisteme eklenme ihtimalleri varsa, hem de nesnelerin oluşturulması sırasında oluşum işleminin belli bir merkezden kontrol edilmesi (fabrika benzeri bir yapı) isteniyorsa, klasik fabrika benzeri tasarım desenleri işimize yaramayacaktır.

[2]'de bir müzik editör programında nota sınıflarından ve notaları ekrana yerleştirmekte kullanılan grafik aracı sınıflarından bahsedilir. Grafik aracı sınıfının her nota sınıfı için alt sınıflara bölünmemesi için nota nesnelerinin prototiplerinin kullanılabileceği belirtilmiştir. Grafik aracı sınıfı o anda ekrana yerleştirdiği nesnenin nasıl oluşturulacağını bilmez, sadece o anda çalışmakta olduğu grafik nesnesinin Kopyalama (Cloning) metodunu çağırarak bir örneğini oluşturur.

Oluşturulacak olan nesne türlerinin sabit olmadığı durumlarda da prototip kullanımı gerekli esnekliği sağlayacaktır. Örneğin CAD programlarında o anda sistemde tanımlı olmayan temel nesne türleri, bu tür bir yapı ile, gerek programcının kendisi, biraz fazladan çaba ile de program ortaya çıktıktan sonra, eklenti olarak kullanıcılar veya üçüncü parti programcılar tarafından sonradan sisteme eklenebilir.

Java programlama dili diğer tüm nesnelerin türetildiği en temel Object nesnesinde clone metodunu tanımlamıştır. Ancak bu işlemin C++ veya SmallTalk benzeri dillerde bazı sorunlara yol açabilmektedir.

2.5.3 Uygulanabilirlik

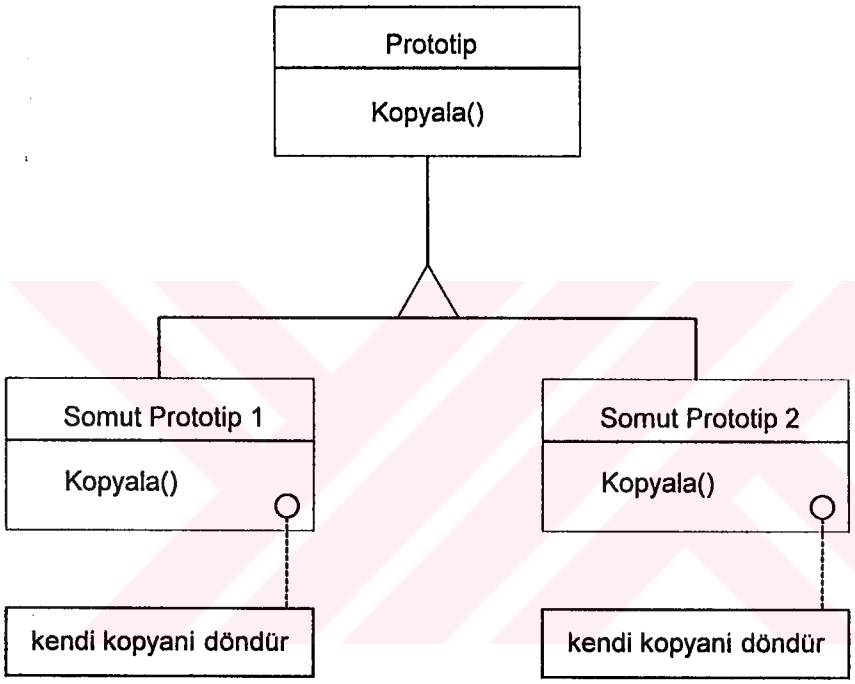
Prototip tasarım desenini,

- Sistemin kullandığı nesnelerin nasıl oluşturulacağından bağımsız olması gerektiğinde

- Kulanılacak sınıfların çalışma zamanında (run-time) belirlenmesi gerektiğinde
- Eğer bir sınıfın oluşabilecek kombinasyonları çok az ise o zaman bu sınıfın, farklı kombinasyonlarda oluşturulmuş örnekleri de sisteme bir prototip olarak kaydedilebilir.

2.5.4 Yapı

Prototip tasarım deseninin sınıf yapısı ve ilişkileri Şekil 2.14'te görülmektedir



Şekil 2.14 Prototip tasarım deseninin yapısı

2.5.5 Kullanım

Prototip sınıfı "Soyut fabrika"(Abstract Factory) ve "Kurucu" (Builder) [2] oluşturucu tasarım desenleri ile pek çok ortak avantajı paylaşır, örneğin üretim mekanizmasının ürettiği sınıfların iç yapısını bilmemesi gibi. Bunun yanında Prototip tasarım desenini kullanmanın avantajları şöyle sıralanabilir.

1. Sınıfların programın çalışma zamanında eklenebilmesi ve çıkarılabilmesi: Prototip kullanımı, sisteme bilinen temel bir sınıftan türeyen farklı sınıfların sonradan eklenebilmesini sağlar. Yeni sınıfın bir örneği üretilir ve yazılımdaki bir prototip gurubuna kaydedilmesi sağlanır. Daha sonra bu nesneden bir tanesinin

elde edilmesi istendiğinde, sınıfta tanımlı olan kendisini kopyalama (cloning) metodu çağrılır.

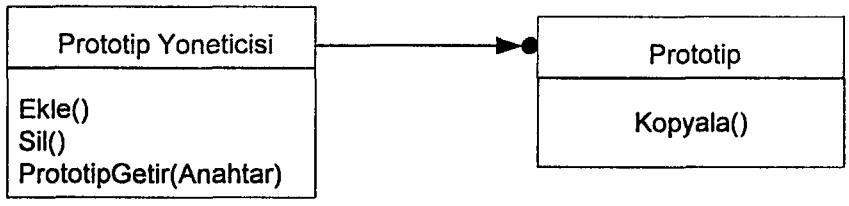
2. Altsınıfların azalması: Üretim tasarım desenleri genelde her nesne türü için farklı bir fabrika sınıfı ortaya çıkarır. Ancak prototip sisteminde nesneler kendi kendileri için bir fabrika olarak görev yaparlar.

3. Yeni bir yapıya sahip nesnelerin tanımlanabilmesi: sistemdeki ilkel nesnelerin bileşiminden oluşan kompozit nesneler, [2,3] listelerindeki ilkel nesnelerinin kopyala metodunu da kullanarak kendilerinin bir kopyasını oluşturabilirler. Bu sayede karmaşık, kullanıcı tanımlı blok nesneler de sisteme sonradan eklenebilir hale getirilebilirler. [2]'de bahsedilen bir devre tasarım programında kullanılan alt devre blokları buna bir örnek olabilir. GeoCAD sisteminde kullanılan kompozit nesneler de kendi kendilerini kopyalama yeteneğine sahiptirler.

Prototip sınıfının en önemli dezavantajı ise desenin kalbinde yer alan kopyala (clone) metodunun C++ gibi dillerdeki uygulamasının oldukça zor olmasıdır. Bu durumda metodu uygulayacak olan sınıflara bağlı olan diğer sınıfların da nasıl kopyalanacağı veya çembersel referanslar için ne yapılacağı belirlenmelidir.

2.5.6 Uygulama

Prototip sınıfının uygulanması Java ve Smalltalk gibi dillerde sorun olmasa da, C++ için halledilmesi gereken problemler karşımıza çıkmaktadır. Prototip tasarım desenini kullanırken aşağıdaki maddelerdeki konuların akılda tutulması yararlı olacaktır.



Şekil 2.15 Prototip Yöneticisi

1. Prototip yöneticisinin kullanımı:

Eğer uygulamada kullanılacak olan nesnelerin sayısı belirgin değilse, bir prototip yöneticisini kullanılması uygun olabilir. Bu durumda bir prototip yönetici sınıfı oluşturulur sınıfın prototip eklemek, çıkarmak ve belli bir anahar kullanarak

kendisine daha öncden kaydedilmiş olan prototipleri bulmak için bir bulma metodunu da içermesi gerekebilir.

GeoCAD sisteminde de ilkel nesneler, komut nesneleri ve undo nesneleri prototip tasarım desenine uygun olacak şekilde kullanılmışlardır. Temel görsel nesne sınıfları (nokta, çember vs.) bu nesneleri tutan bir Nesne yöneticisi'ne kaydedilirler. Bir proje oluşturulduğu zaman, sistem bu listeye göre listedeki her temel nesne için bir nesne kümesi oluşturur. Herhangi bir nesne oluşturulacağı zaman Nesne yöneticisine oluşturulacak olan nesnenin anahtarı gönderilir ve nesne yöneticisi bulduğu prototipe kendi kendisinin bir Kopyasını oluşturmasını söyler, oluşan kopyayı da istekte bulunan sınıfa gönderir. Sisteme istenirse sonradan yeni nesne türleri eklenebilir.

2. Kopyala (clone) işleminin gerçeklenmesi:

Kopyalama işlemi, genelde kopyalanacak olan sınıfın kendisinin bir örneğini (instance) oluşturup geri döndürmesinden ibarettir. Bu iş için gerekli olan çmetodların Java ve Smalltalk gibi dillerde gerçeklenmiş durumda olduğunu daha önce belirtmiştik. Java'da cloneable arayüzünü gerçekleyen tüm sınıflar temel Object sınıfının clone metodunu da tekrar gerçeklerler (override). Bu sınıflardan oluşturulan nesnelerin clone metodunu çağırmak o nesnelerin tam bir kopyasının hafızada oluşmasını sağlar. Dönüş değeri de Object cinsindendir, o yüzden oluşan kopyayı kullanabilmek için bir altdönüşüm (downcasting) işlemi gerekebilir, örneğin bir vektörü kopyalamak için aşağıdaki küçük java kod örneği kullanılabilir,

```
Vector v = new Vector(10);
Vector clonedvector;
Clonedvector = v.clone();
```

Java dilindeki clone metodu nesneyi ve o anda nesnenin taşıdığı referanslara bağlı tüm nesneleri kopyalar. Yani yukarıdaki örnekteki v vektörüne bağlı tüm nesnelerin de bellekte bir kopyası alınır. C++ dilinde clone işlemini gerçekleştirilirken ne tür bir kopyalama işlemi yapacağımıza karar vermemiz gerekir. Eğer bir nesnenin kopyası alınırken sadece o nesneyi kendi iç değişkenlerinin o andaki değerleri ile beraber kopyalarsak bu boş (shallow) bir kopyalama olacaktır. GeoCAD sistemindeki nesneler temelde bu boş kopyalama işlemini desteklerler, örneğin Cnokta, Ccember ve Chat sınıfları kendi kopyalarını CGeoNesne cinsinden bir işaretçi üzerinden döndüren Clone() metodlarına sahiptirler. Aşağıdaki örnekte CNokta sınıfı için Clone metodunun uygulanışını ve kullanım örneğini görüyoruz.

```
CGeoNesne* CNokta::Clone(){
```

```
 return new CNokta(parametreler);
}
Nokta1 = new CNokta();
CNokta *Kopya = (Cnokta *)Nokta1.Clone();
```

Ancak kopyalama işlemi yapılırken, nesnenin içerisinde barındırılan diğer nesnelerin de kopyala metodu çağrılırsa o zaman bir derin (deep) kopyalama işlemi gerçekleşmiş olacaktır. Bu tür kopyalama işlemlerinin kontrolü oldukça zor olabilir, örneğin bir Vektör temel nesne türünden türeyen çeşitli nesnelerin işaretçilerini üzerinde taşıyan dizidir. Bir vektörün kopyalanması sadece bu işaretçilerin kopyalanmasından ibaret olursa, diğer vektörden bir nesne silinmesi halinde oluşturulan kopya da doğrudan etkilenecek, ve bu silme işleminden haberdar edilmediği için kendi listesinde silinen nesne halen var imiş gibi görünecektir. Bu durum istenmeyen bazı hataların oluşmasına sebep olabilir. Bu yüzden, bu tür nesnelerde derin kopyalama işleminin en azından kısmen gerçeklenmesi gerekebilir.

3.Prototiplerin ilklenmesi:

Prototip tasarım deseni her ne kadar nesnelerin oluşturulmasında kolaylıklar sağlasa da oluşturulan nesnelere ilk değerlerinin nasıl kazandırılması gerektiği konusunda bir fikir vermez. Normalde bir nesne oluşturulduğu zaman bazı ilk değerleri sınıfın constructor metodunda verilir. Ancak prototip sınıfında bu tür bir şansımız olmayacağından, nesnenin prototipinin kopyalanmasının ardından bir Degerleri ilkleme metodunun çağrılması yerinde olabilir.

2.5.7 Kod Örneği

GeoCAD sisteminde prototip mekanizması kullanılmıştır. Özellikle nesnelerin oluşturulmasında Ana çekirdekteki bir nesne yöneticisine önceden kaydedilmiş olan nesne prototipleri kullanılmıştır. Nesnelerdeki clone metodu da GeoCAD sisteminin temel nesnesi olan CgeoNesne'de sanal (virtual) bir metod olarak belirleniş, ve isteyen sınıf bu metodu tekrar uygulayarak istenen sonucun elde edilmesi sağlanmıştır. Clone metodu eğer tekrar uygulanmamışsa (overrided), tüm nesneler için NULL döndürür. CGeoNesne sınıfına bir göz atarsak;

```
class CGeoNesne {
  public:
   virtual CGeoNesne* Clone(){return NULL;};
   virtual bool Karsilastir( CGeoNesne * Karsilastirilacak){
     return (this==Karsilastirilacak);
   }
```

```
    virtual unsigned int HashDegeri(){ return 0; }
};
```

Bu sınıftan türeyen CNokta nesnesi ise Clone metodunu gerçeklerken kendi kendisinin bir kopyasını alarak geri döndürmüştür.

```
CGeoNesne* CNokta::Clone()
 {
  CNokta *ClonedNokta = new CNokta(this->X,this->Y,this-
>Z,this->NoktaNo,this->Tabaka,this->Durum);
  return (ClonedNokta);
 }
```

Nesne prototipleri Cekirdek sınıfındaki bir Vektörde saklanırlar. Bu açıdan bakılınca Cekirdek sınıfı bizim Prototip yöneticimiz sayılır. Cekirdek sınıfının aşağıda sıralanan metodları gereken işlevselliği sağlamıştır.

```
bool NesneTuruKaydet(CNesne *YeniNesneTuru);
void NesneleriKaydet();
CGeoVektor* TanimliTurleriGetir();
CNesne* PrototipGetir(int NesneID);
```

NesneTuruKaydet metodu yeni bir nesnenin sisteme eklenmesini sağlarken, TanimliTurleri getir, bize o ana kadar kaydedilmiş olan nesnelerin sınıf bilgilerinin bulunduğu bir vektörü geri getirir. PrototipGetir metodu ise, istenen nesneden bir kopya oluşturup geriye döndürür. PrototipGetir metodu, NesnelD anahtarı ile aranan nesneyi daha önceden kaydedilmiş olan prototipler arasından bulur ve o nesnenin Clone metodunu çağırır. Eğer aranan nesne listede bulunamamışsa geriye NULL döndürülür.

```
CNesne* CCekirdek::PrototipGetir(int NesneID){
 CNesne* Prototip=NULL;
 Prototip = (CNesne*)KayitliNesneler->ElemanGetir(NesneID);
 if(Prototip == NULL) {
   KONSOL->println("Prototip isteği geçersiz, Nesne Prototipi
kaydedilmemis!");
   return NULL;
 }
 else return (CNesne*)Prototip->Clone();
}
```

2.6 Fabrika Metodu

Bu bölümde "Fabrika metodu" tasarım deseni anlatılmıştır

2.6.1 Amaç

Bir nesne oluşturmak için bir arayüz tanımlamak, ama hangi sınıfın ilklendirileceğine alt sınıfların karar vermesine izin vermek. Fabrika metodu bir sınıfın ilklendirmeyi (construction) alt sınıflara ertelemesini sağlar.

2.6.2 Motivasyon

Çerçeveler, (Frameworks) nesneler arasındaki ilişkileri tanımlamak ve bunları idare etmek için soyut sınıfları kullanırlar. Bir çerçeve genellikle nesnelerin oluşturulmasından da sorumludur.

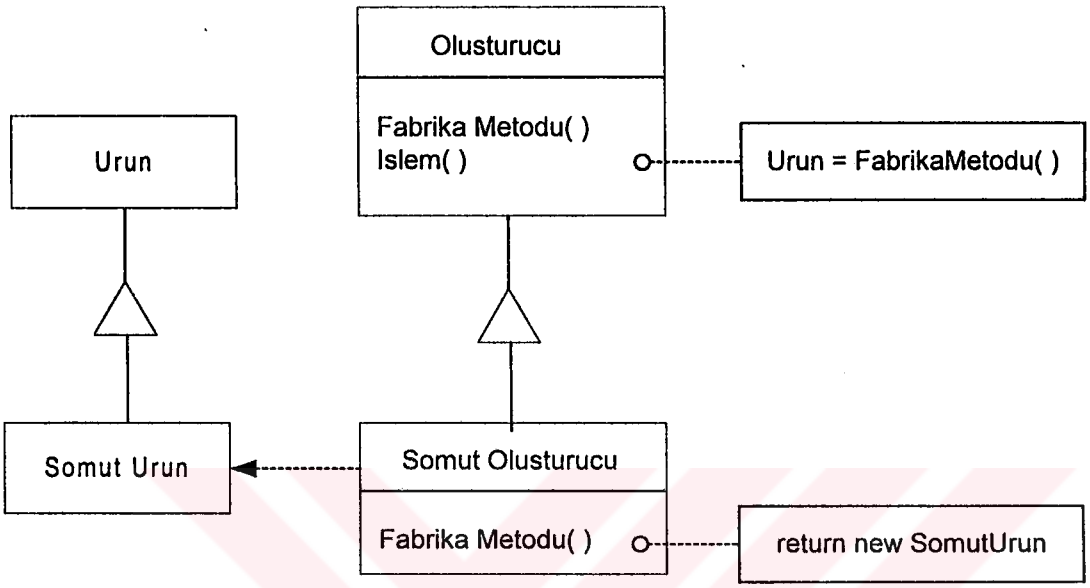
Bir kullanıcıya birden fazla doküman sunabilen uygulamalar için bir çerçeveyi düşünelim. Bu çerçevedeki iki anahtar soyutlama Uygulama (Application) ve Doküman (Document) sınıflarıdır. İki sınıf da somuttur ve istemciler (client) bu sınıfların uygulamaya özel gerçeklemelerini yapmak için bunların alt sınıflarını oluşturmalıdır. Örneğin bir çizim uygulaması yapmak için ÇizimUygulaması (DrawingApplication) ve ÇizimDokümanı (DrawingDocument) sınıflarını tanımlarız. Uygulama sınıfı Dokümanların yönetiminden sorumludur eve bunları gerektikçe – örneğin bir kullanıcı bir mönüden "Aç" ya da "Yeni"'yi seçtiğinde – oluşturacaktır.

İlklendirilecek "Doküman" alt sınıfı uygulamaya özel olduğu için Uygulama sınıfı "Doküman"'ın hangi alt sınıfının ilklendirileceğini tahmin edemez – Uygulama sınıfı ne tür bir Doküman oluşturulacağını değil, yalnızca yeni bir dokümanın ne zaman oluşturulacağını bilir. Bu bir ikilem yaratır: Çerçeve sınıfları ilklendirmelidir ama yalnızca soyut sınıflar hakkında bilgisi vardır, ama soyut sınıfları ilklendiremez.

Fabrika metodu deseni bu soruna bir çözüm sunmaktadır. Bu desen hangi Doküman alt sınıfının ilklendirileceği bilgisini kapsar ve bu bilgiyi çerçevenin dışına taşır.

2.6.3 Yapı

Fabrika Metodu tasarım deseninin sınıf yapısı Şekil 2.16'da görülmektedir



Şekil 2.16 Fabrika Metodu tasarım deseninin yapısı

2.6.4 Katılımcılar

- Ürün (Doküman)

  fabrika metodunun oluşturacağı nesnelerin arayüzünü tanımlar

- Somut Ürün (Dokümanım)

  Ürün arayüzünü gerçekler

- Oluşturucu (Uygulama)

- Ürün tipinde bir nesne döndüren fabrika metodunu tanımlar Oluşturucu ayrıca fabrika metodunun varsayılan (default) bir SomutÜrün nesnesi döndüren bir varsayılan gerçeklemesini de tanımlayabilir. Bir Ürün nesnesi oluşturmak üzere fabrika metodunu çağırabilir.

- SomutOluşturucu (Uygulamam)

  fabrika metodunu SomutÜrünün bir instance'ını döndürecek şekilde tekrar tanımlar (override) eder.

2.6.5 Sonuçlar

Fabrika metotları program koduna uygulamaya özel sınıfları bağlama (bind) gereğini ortadan kaldırır. Kod yalnızca Ürün arayüzü ile ilgilenir; böylece kullanıcı tarafından tanımlanmış herhangi bir SomutÜrün sınıfı ile birlikte çalışabilir.

Fabrika metotlarının potansiyel bir dezavantajı istemcilerin sadece belli bir SomutÜrün nesnesi oluşturmak için Oluşturucu sınıfının alt sınıfını oluşturma zorunda kalmalarıdır. Eğer istemci zaten Oluşturucu sınıfının alt sınıfını oluşturacaksa bunun zararı yoktur ama aksi takdirde istemci şimdi sınıflardaki evrimin bir başka aşaması ile uğraşmak zorunda kalacaktır.

Aşağıda Fabrika Metodu deseninin iki sonucu daha verilmiştir:

1. Altsınıflar için açık uçlar sağlar. Fabrika metodu ile bir sınıfın içerisinde nesneler oluşturmak her zaman bir nesneyi doğrudan oluşturmaktan daha esnektir. Fabrika metodu alt sınıflara bir nesnenin genişletilmiş bir versiyonunu oluşturma imkanını verir.

   Doküman örneğinde , Doküman sınıfı mevcut bir dokümanı açmak için varsayılan bir dosya dialoğu nesnesi oluşturan bir DosyaDiyaloğuOluştur (CreateFileDialog) adlı bir fabrika metodu tanımlamayabilir. Dokümanın bir altsınıfı bu fabrika metodunun üstüne yazarak (override) uygulamaya özel bir dosya diyaloğu tanımlayabilir. Bu durumda fabrika metodu soyut değildir ama kabul edilebilir bir varsayılan gerçekleme yapılmasını sağlar.

2. Paralel sınıf hiyerarşilerini birbirine bağlar.

Şimdiye kadar incelediğimiz örneklerde fabrika metodu yalnızca Oluşturucular tarafından çağrılmaktaydı. Fakat bu bir zorunluluk değildir; istemciler de özellikle paralel sınıf hiyerarşilerinin oluğu durumlarda fabrika metotlarından faydalanabilirler.

Paralel sınıf hiyerarşileri bir sınıf sorumluluklarından bir kısmını baka bir sınıfa aktardığında ortaya çıkar. Etkileşimli olarak işlenebilen grafiksel figürleri düşünelim; bunlar fare hareket ettirilerek uzatılabilir, yerleri değiştirilebilir, ya da döndürülebilirler. Bu tür etkileşimler gerçekleştirmek her zaman kolay değildir çünkü

genellikle verilen bir zamandaki değişikliğin durumunu kaydetmek ya da güncellemek gerekecektir. Bu duruma yalnızca değişiklik sırasında ihtiyaç duyulur; bu yüzden bunun figür nesnesinde tutulmasına gerek yoktur. Dahası farklı figürler kullanıcı kendilerini değiştirildiklerinde farklı davranırlar. Örneğin bir çizgi figürünü uzatmak sonlanma noktasının yer değiştirmesi etkisini oluştururken bir metin figürünü uzatmak satır aralarında değişikliğe yol açabilir.

Bu kısıtlarla,etkileşimi gerçekleyen ve değiştirmeye özel herhangi bir durum bilgisini takip eden ayrı bir Değiştirici nesnesi kullanmak daha iyidir. Farklı figürler belli etkileşimleri gerçekleştirmek için farklı Değiştirici altsınıfları kullanırlar. Sonuçta oluşan Değiştirici sınıf hiyerarşisi (en azından kısmen) Figür sınıfı hiyerarşisine paralel olacaktır.

2.6.6 Gerçekleme

Fabrika Metodu desenini uygularken aşağıdaki noktalar dikkate alınmalıdır.

1. İki ana varyasyon vardır:

Fabrika Metodu desenin iki varyasyonu vardır. (1) Oluşturucu sınıfının soyut bir sınıf olduğu ve tanımladığı fabrika metodu için bir gerçekleme sağlamadığı durum (2) Oluşturucu sınıfının somut bir sınıf olduğu ve fabrika metodu için varsayılan bir gerçekleme sağladığı durum. Varsayılan bir gerçekleme sağlayan bir soyut sınıfın olduğu durum da mümkündür ama buna çok sık rastlanmaz.

Birinci durum alt sınıfların bir gerçekleme tanımlamasını gerektirir, çünkü geçerli bir varsayılan yoktur. Önceden görülemeyen sınıfları ilklendirme ikilemini çözmeyi sağlar. İkinci durumda, somut Oluşturucu, fabrika metodunu en başta esneklik sağlamak amacıyla kullanır. Bu durumda "Nesneleri ayrı bir işlemle oluştur böylece alt sınıflar kendilerinin oluşturulma şeklini değiştirebilsinler" kuralı takip edilmektedir. Bu kural alt sınıfların tasarımcılarının atalarının ilklendirdiği nesneler sınıfını gerektiğinde değiştirebilmelerini garanti altına alır.

2. Parametreleştirilmiş fabrika metotları:

Desenin bir başka varyasyonu fabrika metodunun çoklu tipte ürünler oluşturmasına izin verir. Fabrika metodu oluşturulacak nesneleri tanımlayan bir parametre alır. Fabrika metodunun oluşturduğu tüm nesneler Product arayüzünü paylaşacaktır. Doküman örneğinde Uygulama farklı türde Doküman'ları destekleyebilir. DokümanOluştur'a hangi türde bir doküman oluşturulacağını belirten fazladan bir parametre yollanır.

Unidraw grafik düzenleme çerçevesi diskte saklanmış nesneleri yeniden yapılandırmak için bu yaklaşımı kullanır. Unidraw, argüman olarak bir sınıf belirteci alan Oluştur adlı bir fabrika metodu olan bir Oluşturucu sınıfı tanımlar. Unidraw bir nesneyi diske sakladığında önce sınıf belirtecini, ardından ilklenmiş değişkenlerini yazar. Nesneleri diskten yeniden oluştururken ise önce sınıf belirtecini okur.

Sınıf belirteci okunduktan sonra çerçeve belirteci parametre olarak göndererek Oluştur'u çağırır. Oluştur, ilgili sınıfın ilklendiricisini (constructor) bulur ve nesneyi ilklendirmek için kullanır. Son olarak, Oluştur nesnenin Oku işlemini çağırır. Oku geri kalan bilgileri diskten okur ve nesnenin insance değişkenlerini ilklendirir.

Parametreleştirilmiş bir fabrika metodu genel olarak aşağıda gösterilen yapıdadır. Burada Ürünüm ve Ürünün Ürün sınıfının altsınıflarıdır.

```
Class Olusturucu {
Public:
      virtual Urun* Olustur(UrunId);
};
Urun* Olusturucu::Olustur (UrunId id) {
      İf (id==MINE) return new Urunum;
İf (id==YOURS) return new Urunun;
// diğer Urunler için tekrarla…
return 0;
}
```

Parametreleştirişmiş bir fabrika metodunun üstüne yazmak (override) bir Oluşturucu'ın ürettiği ürünleri kolayca ve seçerek değiştirme imkanı verir. Yeni türde ürünler için yeni belirteçler tanımlanabilir ya da mevcut belirteçleri farklı ürünlerle ilişkilendirilebilir.

Örneğin, bir Oluşturucum altsınıfı Ürünüm ve Ürünün ile yer değiştirtip yeni bir Ürünleri altsınıfını destekleyebilir:

```
Urun* Olusturucum::Olustur (UrunID id) {
İf (id==YOURS) return new Urunum;
İf (id==MINE) return new Urunun;
```

```
İf (id==THEIRS) return new Urunleri;
      Return Olusturucu::Olustur(id);
}
```

3. Dile özel farklılıklar ve sorunlar.

Farklı diller başka ilginç varyasyonlar için elverişlilik gösterebilirler.

Smalltalk programları ilklendirilecek nesnenin sınıfını döndüren bir metodu sıklıkta kullanır. Bir Oluşturucu fabrika metodu bu değeri bir ürün oluşturmak için kullanabilir ve bir SomutOlusturucu bu değeri saklayabilir hatta hesaplamada kullanabilir. Bunun sonucunda ilklendirilecek SomutÜrün tipinin belirlenmesinin daha da sonraya bırakılabilmesidir.

Parametreleştirilmiş fabrika metotlarına akraba olan daha da esnek bir yaklaşım ise oluşturulacak sınıfı Uygulama'nın bir sınıf değişkeni olarak kullanmaktır. Bu şekilde ürünü değiştirmek için Uygulamayı türetmek gerekmez.

2.7 Dekoratör

Bu bölümde "Dekoratör" tasarım deseni anlatılmıştır

2.7.1 Amaç

Bir nesneye dinamik olarak ek sorumluluklar yüklemektir. Dekoratörler işlevselliği artırmak için yapılan türetmelere esnek bir alternatif sağlarlar.

2.7.2 Diğer Adı

Kaplayıcı, Wrapper

2.7.3 Motivasyon

Bazen sınıfın tümüne değil bireysel nesnelere sorumluluklar eklemek isteriz. Örneğin bir grafik kullanıcı arabirimi araç kutusu (toolkit) kullanıcı arayüz öğelerine sınır gibi özellikler ya da kaydırma gibi davranışlar eklememize izin vermelidir.

Sorumluluklar eklemenin bir yolu kalıtımdır. Başka bir sınıftan bir sınır kalıtım her alt sınıf örneğinin (instance) çevresine bir sınır ekler. Ancak sınır seçimi statik olarak yapıldığı için bu esnek değildir. Bir istemci öğeyi ne zaman ve nasıl bir sınıfla dekore edeceğini kontrol edemez.

Öğeyi sınırı ilave eden başka bir nesneyle kapsamak daha esnek bir yaklaşım olacaktır. Kapsayan nesneye dekoratör adı verilir. Dekoratör dekore ettiği öğenin arayüzüne uyumlu olduğu için böyle bir nesnenin varlığı öğenin istemcileri açısından şeffaftır. Dekoratör istekleri nesneye yönlendirir ve yönlendirmeden önce ya da sonra ek işlemler (bir sınır çizmek gibi) gerçekleştirebilir. Şeffaflık dekoratörleri içiçe (recursive) yazmaya izin verir böylece sınırsız sayıda sorumluluk eklemeyi mümkün kılar.

Örneğin bir pencerede bir metni gösteren bir YazıKutusu nesnesini ele alalım. YazıKutusu her zaman gerekli olmadığı için başlangıçta varsayılan bir kaydırma çubuğuna sahip değildir. Gerektiğinde bir KaydırmaCubuguDekoratoru ile kaydırma çubukları ekleyebiliriz. Ayrıca YazıKutusu'nun çevresine siyah kalın bir sınır çizmek istediğimizi farz edelim. Bunun için bir SınırDekoratoru kullanabiliriz. İstenen sonucu elde etmek için YazıKutusu etrafında dekoratörler oluştururuz.

2.7.4 Uygulanabilirlik

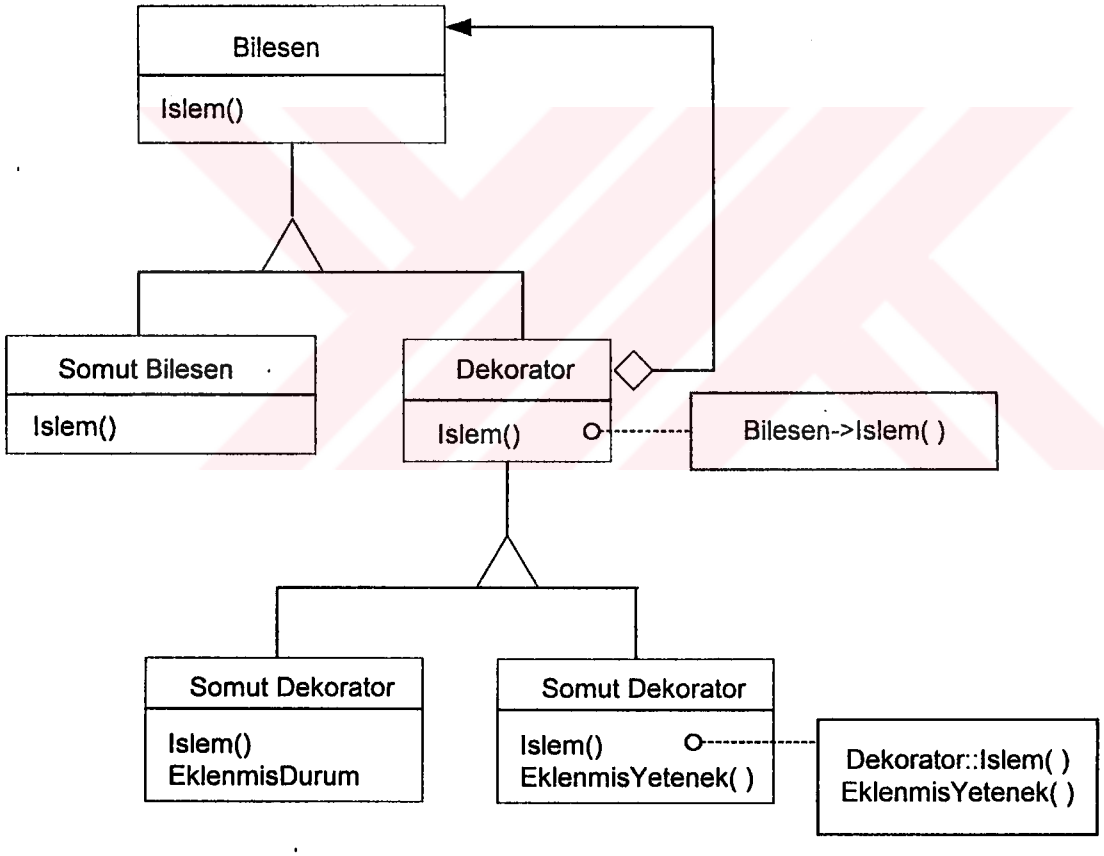
Dekoratörler aşağıdaki durumlarda kullanılır.

Bireysel nesnelere dinamik ve şeffaf bir şekilde, yani diğer nesneleri etkilemeden sorumluluklar yüklemek için,

Geri alınabilecek sorumluluklar için,Türetme ile işlevsellik genişletme pratik olmadığı zaman. Bazen çok sayıda bağımsız genişletmeler olması mümkündür ve her kombinasyonu desteklemek için çok fazla alt sınıf türetilmesi gerekebilir. Ya da bir sınıf tanımı gizli tutulabilir veya türetmeye uygun olmayabilir.

2.7.5 Yapı

Dekoratör tasarım deseninin sınıf yapısı ve ilişkileri Şekil 2.17'te görülmektedir



Şekil 2.17 Dekoratör tasarım deseninin yapısı

2.7.6 Sonuçlar

Dekoratör deseninin avantaj ve dezavantajları:

1. Statik kalıtımdan daha esnektir: Dekoratör deseni nesnelere sorumluluk eklemede statik (çoklu) kalıtım ile elde edilebilecek olandan daha esnek bir yol sağlar. Dekoratörlerle, sorumluluklar dekoratörlerin eklenmesi ya da çıkartılması suretiyle çalışma zamanında (run-time) eklenip geri alınabilir. Kalıtım ise eklenecek her sorumluluk için yeni bir sınıf tanımlamayı gerektirir. Bunun sonucu olarak sınıf sayısı ve sistemin karmaşıklığı artar.

Dekoratörler ayrıca bir özelliğin iki kez eklenmesini de kolaylaştırır. Örneğin, bir YazıKutusu nesnesine iki sınır çizmek için yalnızca iki tane sınır dekoratörü eklemek yeterlidir.

2. Hiyerarşinin en tepesinde çok özellikli sınıfların olmasını engeller: Dekoratörler sayesinde ilerde ortaya çıkabilecek tüm ihtiyaçları destekleyecek karmaşık, özelleştirilebilir bir sınıf yazmak yerine basit bir sınıf tanımlanabilir ve Dekoratör nesneleriyle azar azar sorumluluk eklenebilir.

3. Bir dekoratör ve bunun öğeleri birbirinin aynı değildir: Bir dekoratör şeffaf bir kılıf görevi yapar. Ancak nesne kimliği açısından bakıldığında dekore edilmiş bir nesne öğenin kendisiyle tıpatıp aynı değildir. Dolayısıyla programcıyı dekoratörler kullanırken nesne kimliğine güvenmemelidir.

4. Çok sayıda küçük nesne olması: Dekoratörleri sıkça kullanan bir tasarım birbirine benzeyen çok sayıda küçük nesneden oluşan sistemler oluşturur. Nesneler sınıfları ya da değişkenlerinin değerleri açısından değil yalnızca bağlanış şekilleri açısından birbirlerinden faklılık gösterirler. Bu sistemler kendilerini anlayanlar tarafından kolayca özelleştirilebilirse de bunları öğrenmek ve hata ayıklamak zordur.

2.7.7 Gerçekleme

Dekoratör desenini uygulamaya koyarken çeşitli sorunlar göz önüne alınmalıdır.

1. Arayüze uyumluluk: Bir dekoratör nesnesinin arayüzü, dekore ettiği öğenin arayüzüne uyumlu olmalıdır.

2. Soyut dekoratör sınıfının atlanması: Yalnızca bir sorumluluk eklemek gerektiğinde bir soyut Dekoratör sınıfı eklemeye gerek yoktur. Genellikle bu durum yeni bir sınıf hiyerarşisinin oluşturulmayıp mevcut bir hiyerarşinin üzerinde çalışıldığı durumlarda ortaya çıkar.

3. Bileşen sınıflarının hafifsıklet tutulması: Uyumlu bir arayüzün oluşmasını garanti etmek için bileşenler ve dekoratörler ortak bir Bileşen sınıfından türemelidirler. Bu

ortak sınıfın hafifsıklet tutulması önemlidir. Yani sınıf veriyi saklamaya değil bir arayüz tanımlamaya odaklanmalıdır. Veri gösteriminin tanımı altsınıflara ertelenmelidir. Aksi halde Bileşen sınıfının karmaşıklığı dekoratörlerin kullanımını çok ağırlaştırabilir yapabilir. Bileşen'e çok fazla sorumluluk yüklemek somut altsınıfların ihtiyaç duymayacakları özellikler için bedel ödemeleri ihtimalini artırır.

4. Bir nesnenin iç yapısını değiştirmek yerine dışını değiştirmek: Dekoratörü bir nesnenin üzerinde onun davranışını değiştiren bir katman olarak düşünebiliriz. Bir alternatif ise nesnenin iç yapısının değiştirilmesidir.

Bileşen sınıfının ağırlaştığı olduğu durumlarda stratejiler iyi birer çözümdür. Strateji deseninde bileşen davranışlarının bir kısmını ayrı bir strateji nesnesine yönlendirir. Strateji deseni strateji nesnesini değiştirerek bileşenin fonksiyonelliğini artırmamıza ya da değiştirmemize imkan sağlar.

Örneğin, bileşenin sınır çizme işini ayrı bir Sınır nesnesine ertelemesiyle farklı sınır stillerini destekleyebiliriz. Sınır nesnesi, sınır çizme stratejisini kapsayan (encapsulation) eden bir Strateji nesnesidir. Strateji sayısını birden ucu açık bir genişleterek dekoratörleri iç içe çağrılarla birbirlerine bağlayarak aynı sonucu elde ederiz.

Strateji tabanlı yaklaşım bileşeni yeni genişletmeleri destekleyecek şekilde değiştirmeyi gerektirebilir. Diğer yandan, dekoratörün arayüzü bileşenin arayüzüne uymalıyken bir stratejinin kendi özelleştirilmiş arayüzü olabilir.

2.7.8 Bilinen Kullanımlar

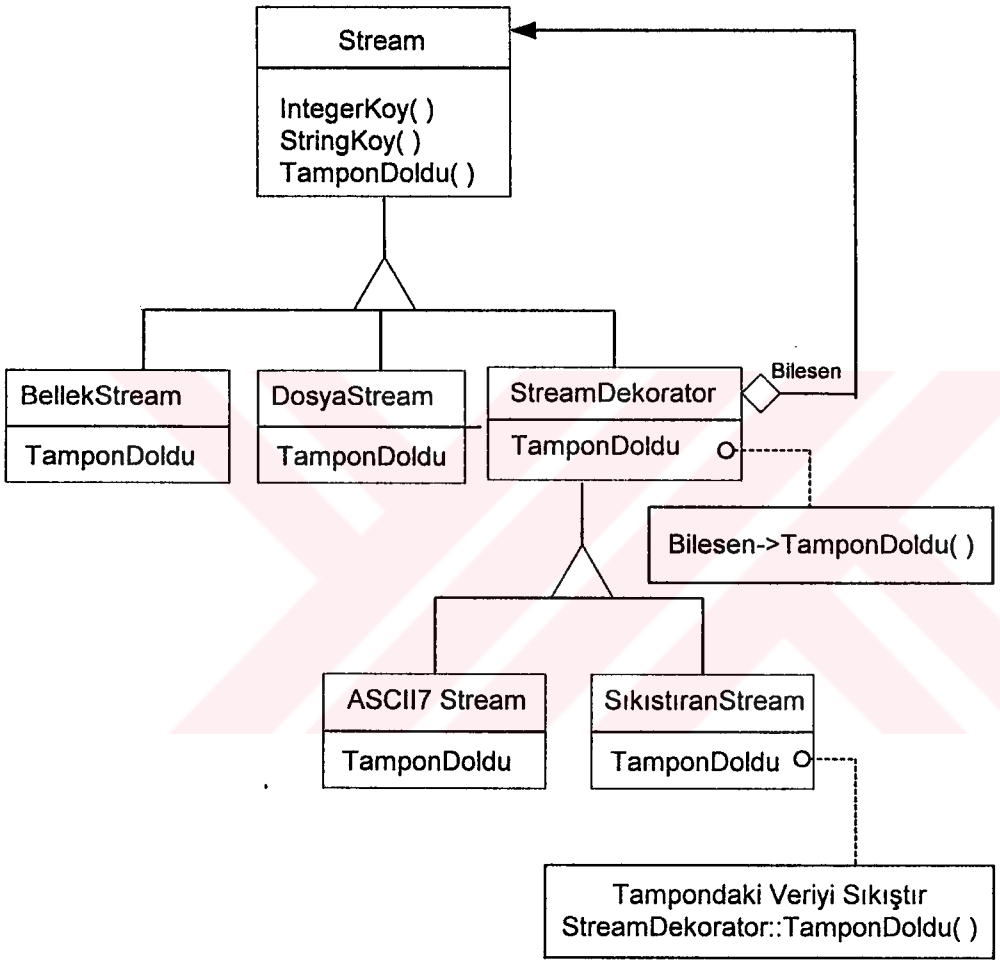
Birçok nesneye yönelik kullanıcı arayüz araçlara grafik süslemeler eklemek için dekoratörleri kullanır. Ancak Dekoratör deseninin kullanım alanı grafik kullanıcı arabirimleri ile sınırlı değildir.

Stream'ler birçok G/Ç aracında temel bir soyutlamadır. Bir stream nesneleri bir karakter ya da sekizli sekansına dönüştürmek için bir arayüz sağlarlar. Bu bizim bir nesneyi daha sonra ulaşabileceğimiz bir dosyaya ya da bir katara dönüştürmemizi sağlarlar. Bunu yapmanın bir yolu bellek Stream ve FileStream adında iki altsınıfı olan soyut bir Stream sınıfı tanımlamaktır. Ama varsayalım aşağıdakileri de yapmak istiyoruz:

Veri stream'ini farklı sıkıştırma algoritmaları kullanarak sıkıştırmak,

Stream'i 7 bitlik ASCII karakterlerine indirgeyerek bir ASCII iletişim kanalından iletilmesini sağlamak.

Dekoratör deseni bize bu sorumlulukları zarif bir şekilde eklemek için bir yol sunar.



Şekil 2.18 Dekoratör tasarım deseninin bir uygulaması

Stream soyut sınıfı bir iç tampon tutar ve verileri stream'de saklamak için işlemler sağlar. (PutInt, PutString) Tampon dolduğunda Stream TamponDoldu soyut işlemini çağırır. Bu işlem asıl veri transferi işini yapar. Bu işlemin FileStream versiyonu bu işlemi tamponu bir dosyaya transfer edecek şekilde tekrar tanımlar (override) eder.

Burada anahtar sınıf bir component stream'ine bir referans tutan ve istekleri buna yönlendiren StreamDecorator sınıfıdır. StreamDecorator altsınıfları

HandleBufferFull'u tekrar tanımlar eder ve StreamDecorator'ın TamponDoldu işlemini çağırmadan önce ek işlemler gerçekleştirir.

Örneğin CompressingStream altsınıfı veriyi sıkıştırır ve ASCII7Stream veriyi 7 bit ASCII'ye dönüştürür. Veriyi sıkıştıran ve sıkıştırılmış ikili düzendeki verileri 7 bitlik ASCII'ye dönüştüren bir FileStream oluşturmak için bir FileStream'i bir CompressingStream ve bir ASCII7Stream ile dekore ederiz.

2.8 Sorumluluk Zinciri

Bu bölümde "Sorumluluk zinciri" tasarım deseni anlatılmıştır

2.8.1 Amaç

Bir isteği gönderenin alıcıya bağlı (coupling) olmasını birden fazla nesneye istekle ilgilenme şansı vererek önlemek. Alıcı nesneler bir zincir haline getirilir ve bir nesne isteği yerine getirinceye kadar istek zincir boyunca geçirilir.

2.8.2 Motivasyon

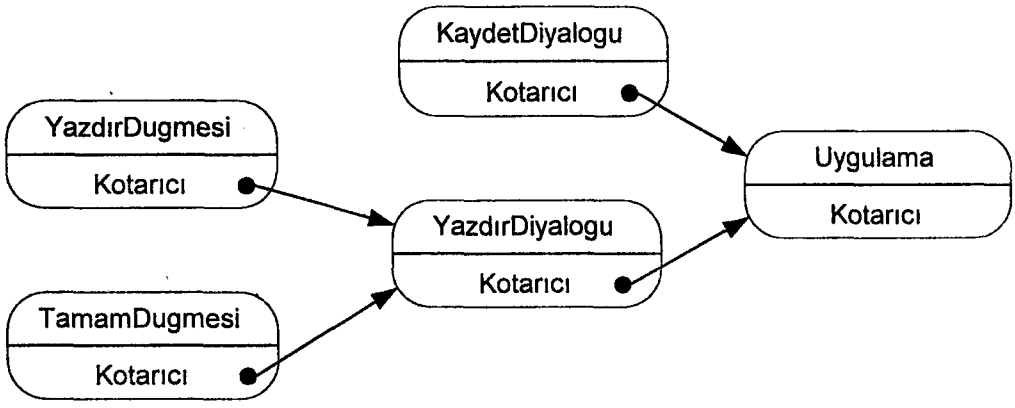
Bir grafik kullanıcı arabiriminin bağlama hassas yardım aracını ele alalım. Kullanıcı arayüzün herhangi bir yerine tıklayarak ilgili yardım bilgisine ulaşabilir. Sağlanan yardım arayüzün seçilen kısmına ve bağlamına bağlıdır; örneğin bir diyalog üzerindeki bir düğmenin yardım bilgisi ana penceredeki benzer bir düğmeden farklı olabilir. Eğer arayüzün bu kısmı için özel bir yardım bilgisi mevcut değilse yardım sistemi eldeki bağlam hakkında, örneğin diyalog kutusunun tamamı hakkında daha genel bir yardım mesajı görüntülemelidir.

Yardım bilgisini genelliğine göre - en özelden en genele doğru - organize etmek doğaldır. Dahası, bir yardım isteği çeşitli kullanıcı arayüzü nesnelerinin biri tarafından ele alınmaktadır. Hangi nesne olduğu bağlama ve yardımın ne kadar özelleştirilmiş olduğuna göre değişir.

Buradaki sorun sonuçta yardımı sağlayan nesnenin başlangıçta yardım isteğini ilklendiren nesne tarafından (düğme) açık olarak bilinmemesidir. Bize gereken yardım isteğini ilklendiren düğmeyi yardım bilgisini sağlayan nesnelerden ayırmak için bir yoldur. Sorumluluk zinciri deseni bunun nasıl yapılacağını tanımlar.

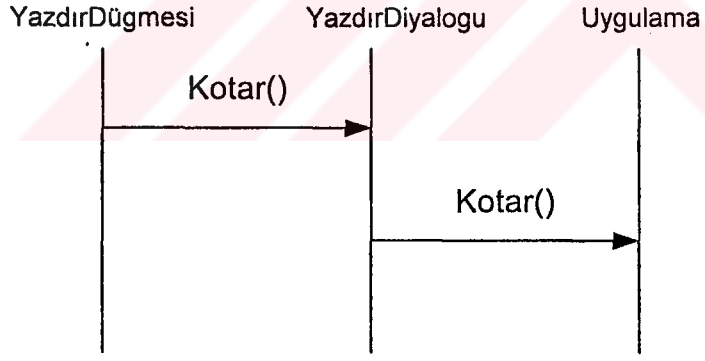
Bu desen birden fazla nesneye bir isteği karşılama şansını vererek göndericileri ve alıcıları birbirinden ayırmaktır. İstek nesnelerden biri bunu karşılayana kadar bir nesne zinciri boyunca iletilir.

Zincirdeki ilk nesne isteği alır ve ya isteği yerine getirir ya da zincirdeki bir sonraki aday nesneye iletir, sonraki aday da aynısını yapar. İstekte bulunan nesne kimin isteği yerine getireceği konusunda bilgiye sahip değildir - bu durumda isteğin açık olmayan bir alıcısı olduğu söylenir.



2.19 .Yardım işlemi için sorumluluk zinciri

Kullanıcının "Yazdır" yazılı bir düğme üzerine yardım için tıkladığını varsayalım. Düğme YazdırDiyalogu'nun bağlı olduğu uygulama nesnesini bilen bir örneği tarafından içerilmektedir. Aşağıdaki diyagram yardım isteğinin zincir boyunca nasıl iletildiğini gösterir.



2.20 Sorumluluk zincirinde emrin iletimi

Bu durumda ne YazdırDugmesi ne de YazdırDiyalogu isteği yerine getirir; istek Uygulama'da durur. Uygulama, isteği göz ardı edebilir ya da yerine getirebilir. İsteği başlatan istemci sonuçta isteği yerine getirene doğrudan bir referans içermez.

İsteği zincir boyunca iletmek ve alıcıların gizli kalmalarını garanti etmek için zincirdeki her nesne isteklerle ilgilenmek ve zincirdeki halefine erişmek için ortak bir arayüzü paylaşır. Örneğin bir yardım sistemi bir YardımKotarıcı (HandleHelp) işlemine sahip bir YardımKotarıcı sınıfı tanımlayabilir. YardımKotarıcı aday nesne sınıfları için üst (parent) sınıf olabilir ya da bir melez (mixin) sınıfı olarak

tanımlanabilir. Bundan sonra yardım isteklerini yerine getirmesini isteğimiz sınıflar Kotarıcı'yı bir üst sınıf yapabilir:

Dugme, Diyalog ve Uygulama sınıfları YardımKotarıcı işlemlerini yardım isteklerini karşılamak için kullanırlar. YardımKotarıcının Kotar işlemi isteği doğrudan sonradan gelene (successor) yönlendirir. Altsınıflar doğru durumlarda yardım sağlamak için tekrar tanımlayabilir.; aksi takdirde isteği yönlendirmek için varsayılan gerçeklemeyi kullanabilirler.

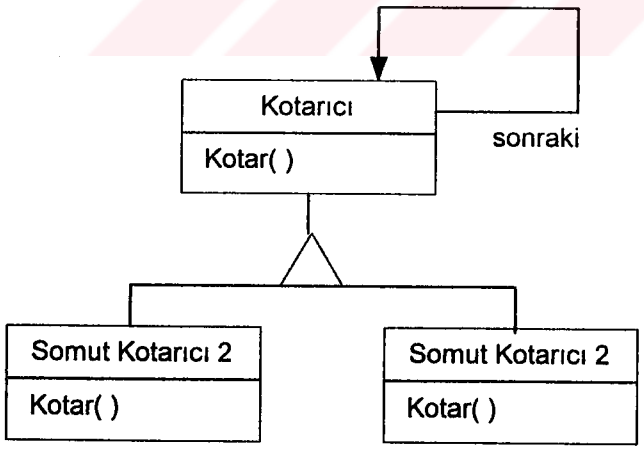
2.8.3 Uygulanabilirlik

Sorumluluk zinciri aşağıdaki durumlarda kullanılır:

istek birden fazla nesne tarafından karşılanabilirse ve kimin isteği yerine getirebileceği önceden bilinmiyorsa. Yerine yetirecek olan otomatik olarak kesinleşmelidir. Alıcıyı açık olarak belirtmeden çeşitli nesnelere bir istek gönderileceğinde.ve bir isteği yerine getirebilecek nesneler kümesi dinamik olarak belirlenecekse sorumluluk zinciri kullanılabilir.

2.8.4 Yapı

Sorumluluk zinciri tasarım deseninin sınıf yapısı Şekil 2.21'de görülmektedir



Şekil 2.21 Sorumluluk zincirinin yapısı

2.8.5 Katılımcılar

Kotarıcı (HelpHandler)

- İstekleri karşılamak için bir arayüz tanımlar.
- Sonragelen bağlantısını gerçekler.

SomutKotarıcı (YazdırDugmesi, YazdırDiyalogu)

- Sorumlu olduğu istekleri karşılar
- Sonragelenine erişebilir
- Eğer SomutKotarıcı isteği karşılayabilirse, karşılar; aksi takdirde isteği sonragelenine yönlendirir.

2.8.6 Sonuçlar

Sorumluluk zincirinin avantaj ve dezavantajları aşağıdadır:

1. İngirdenmiş bağımlılık (decreased coupling): Desen bir nesneyi başka hangi nesnenin isteği yerine getireceğini bilmekten kurtarır. Bir nesnenin yalnızca isteğin "uygun şekilde" yerine getirileceğini bilmesi yeterlidir. Alıcı ve gönderici birbirleri hakkında açık bilgiye sahip değildirler ve zincirdeki bir nesne zincirin yapısını bilmek zorunda değildir.

Bunun sonucu olarak Sorumluluk Zinciri nesne iç bağlaşımlarını (interconnection) basitleştirebilir. Nesnelerin tüm aday alıcılara referans saklamaları yerine songelenlerine tek bir referans bulundurmaları yeterlidir.

2. Nesnelere sorumluluk atama konusunda ek esneklik: Sorumluluk Zinciri sorumlulukları nesneler arasında dağıtmak konusunda ek esneklik getirir. Bir isteği yerine getirme konusundaki sorumlulukları zincire çalışma zamanında ekleme yaparak ya da zinciri değiştirerek ekleyebilir ya da değiştirebilirsiniz.

3. Alındı garanti değildir: Bir isteğin açık bir alıcısı olmadığı için isteğin yerine getirileceğine dair bir garanti yoktur - istek yerine getirilmeden zincirin ucundan düşebilir. Zincirin doğru şekilde ayarlanması da isteklerin yerine getirilmeden iletilmesinin bir başka sebebidir.

2.8.7 Gerçekleme

Sorumluluk zincirinin gerçeklenmesinde aşağıdaki noktalar göz önüne alınmalıdır:

1. Sonragelen zincirinin gerçeklenmesi: Bunun iki yolu vardır:

(a) Yeni linkler tanımlamak (genelde Handler'da ama Somutkotarıcıları da tanımlayabilir)

(b) Mevcut linkleri kullanmak.

Şimdiye kadarki örneklerimiz yeni linkler tanımlamaktaydı ama genelde sonragelen zincirini oluşturmak için mevcut nesne referansları kullanılabilir. Mevcut linklerin kullanımı eğer linkler istenen zinciri destekliyorsa iyi çalışır. Linkleri açık olarak tanımlamaktan kurtarır ve yerden kazandırır. Fakat eğer yapı uygulamanızın gerektirdiği sorumluluk zincirini yansıtmıyorsa gereksiz (fazla) linkler tanımlanmak zorunda kalınır.

2. Sonragelenlerin bağlanması: Eğer bir zinciri tanımlamak için önceden mevcut olan referanslar yoksa bunların tanımlanması gerekir. Bu durumda Kotarıcı yalnızca isteklerin arayüzünü tanımlamakla kalmaz, genelde sonrageleni de tutar. Bu, Kotarıcının isteği sonragelene (eğer varsa) yönlendiren IstegiKotar için varsayılan bir gerçekleme sağlamasına izin verir. Eğer bir ConcreteHandler altsınıfı istekle ilgilenmiyorsa yönlendirme işlemini tekrar tanımlamak (override) etmek zorunda değildir, çünkü varsayılan gerçekleme şartsız olarak yönlendirmeyi yapmaktadır.

Aşağıda bir sonragelen linkini tutan bir YardimKotarici sınıfı verilmiştir.

```
class YardimKotarici {
public:
      YardimKotarici(YardimKotarici* s):_successor(s) {}
      virtual void HandleHelp();
private:
      YardimKotarici* _successor;
};
void YardimKotarici::YardimKotar() {
      if (_successor) {
            _successor->YardimKotar();
      }
}
```

3. İsteklerin ifade edilmesi: İstekleri ifade etmek için farklı seçenekler vardır

Bir alternatif, parametre olarak bir istek kodu (bir tamsayı sabiti ya da bir katar) alan tek bir kotarıcı fonksiyonu kullanmaktır. Bu ucu açık (open ended) bir istekler kümesini destekler. Tek gereken gönderici ve alıcının isteğin nasıl kodlanacağı konusunda anlaşmaya varmış olmalarıdır.

Bu yaklaşım daha esnektir ancak isteği koduna göre göndermek için şartlı ifadelerin kullanılmasını gerektirir. Ayrıca, parametreleri geçirmek için tipler açısından güvenli bir yol olmadığı için parametrelerin paketlenmesi ve paketlerin açılması elle yapılmalıdır. Tabii ki bu bir işlemi doğrudan başlatmaktan daha az güvenlidir.

Parametre geçirme sorununu çözmek için istek parametrelerini paketleyen ayrı istek nesneleri kullanabiliriz. Bir Istek sınıfı istekleri açık olarak ifade edebilir ve yeni istek türleri türetme yoluyla tanımlanabilir. Altsınıflar farklı parametreler tanımlayabilir. Kotarıcı'lar bu parametrelere ulaşmak için gerekli istek türlerini (yani hangi Istek altsınıfın kullandıklarını) bilmelidirler.

İsteği tanımlamak için Istek, sınıf için bir tanımlayıcı döndüren bir erişim fonksiyonu tanımlayabilir. Alternatif olarak, eğer gerçekleme dilleri destekliyorsa alıcı bir çalışma zamanı tip bilgisi kullanabilir.

2.9 Kompozit

Bu bölümde "Kompozit" tasarım deseni anlatılmıştır

2.9.1 Amaç

Nesneleri Ağaç benzeri bir yapıda, bir bütün ve bütünü oluşturan parçaları hiyerarşik şekilde ifade etmek. Kompozit nesneleri oluşturan diğer nesne parçaları ile kompozit'in kendisi aynı nesne arayüzüne sahiptir ve bu nesneleri kullananlar her iki türe de aynı şekilde çağrlar yapabilirler.

2.9.2 Motivasyon

Özellikle rafik ve CAD uygulamalarında karmaşık şekiller çoğu zaman daha basit ve ilkel nesnelerin birleştirilmesi ile oluşturulur. Bazı basit şekiller bile birden fazla ilkel nesneden oluşabilir, örneğin CAD uygulamalarında sıklıkla karşımıza çıkan ve mesafe göstermede kullanılan ölçü nesnesi bunun güzel örneklerindendir. Ölçü nesnesi çizgiler ve aradaki mesafeyi gösteren bir yazı nesnesinden oluşur.

Kompozit tasarım deseni, hem ilkel nesnelere hem de bu ilkel nesnelerin gruplanması ile oluşturulan daha karmaşık nesnelerin aynı nesne etkileşim arayüzüne sahip olmasını sağlayarak, her kompozit nesne için farklı bir uygulama yapma gereğini ortadan kaldırır ve programcıya kolaylık sağlar.

Kompozit tasarım desenindeki anahtar nokta, hem ilkel nesnelerin hem de bu ilkelleri taşıyan kompozitlerin aynı soyut sınıftan türemeleridir.

Harita CAD programlarında kullanılan şev nesnesi de kompozitlerin en iyi örneklerinden biridir. Şev, arazi üzerinde tabiatın veya insanların oluşturduğu ani yükselti değişikliklerinin gösteriminde kullanılan ve kapalı bir poligon ve belirli bir düzende sıralanmış çizgilerden oluşan şekildir (dere yataklarının kenarı, yol kenarındaki ani eğim değişiklikleri gibi) . Bu nesne oluşturulurken bir poligon ve pek çok çizgi nesnesi bir kompozitte toplanır. kompozit çizileceği zaman (Ciz() metodu) Kompozit kendi içerisinde bir listede tutulan tüm nesnelerin Ciz metodunu çağırarak işlemi tamamlar.

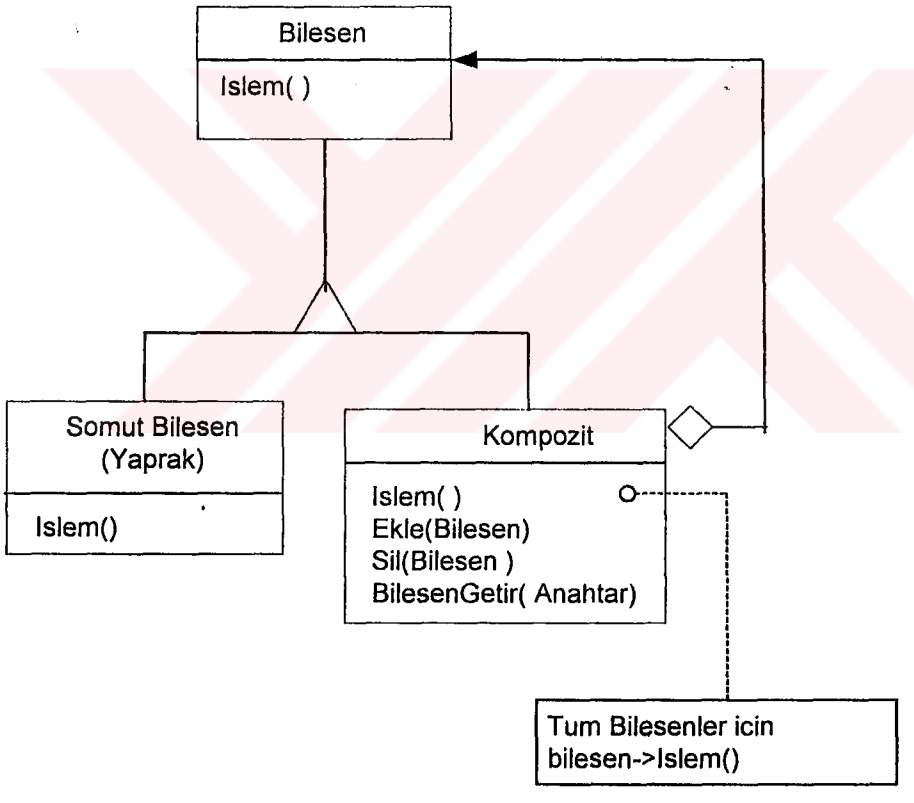
2.9.3 Uygulanabillirlik

Kompozit tasarım deseni,

- Bir bütün ve bütünü oluşturan parçalar şeklindeki yapıları ifade etmek için
- Karmaşık nesneleri ve ilkel nesneleri kullanırken aynı şekilde davranmak ve aynı arayüzü kullanılmak istendiği zaman kullanabilir.

2.9.4 Yapı

Kompozit tasarım deseninin sınıf yapısı ve ilişkileri Şekil 2.22'de görülmektedir



Şekil 2.22 Kompozit tasarım deseni

2.9.5 Avantaj-Dezavantajlar

Sınıf hiyerarşisini kompozit ve ilkel nesnelerden oluşur, kompozitleri birleştirerek daha büyük kompozitler oluşturulabilir. Kompozitleri ve ilkel nesneleri kullanan istemciler hep aynı arayüzü kullanırlar.

Yeni nesne türlerinin eklenmesini kolaylaştırır. Ağaç şeklindeki yapıdaki uçlarda bulunan ilkel nesnelerin tüm tanımları zaten yapılmış olduğundan çoğu zaman kompozitler için ekstra bir iş yapılması gerekmez.

Tüm nesneler için aynı arayüz kullanıldığından , bunları kullanan istemci tarafının yapısı da basitleşir ve her nesne için farklı bir işlem yapmak için büyük case blokları kullanılması gerekmez.

Kompozitlerle ilgili sorunlardan biri kompoziti oluşturan ilkel nesnelerin sistem içerisindeki diğer ilkellerle aynı veriyapısının içinde taşınmamasının doğurabileceği sorunlardır. Eğer bir işlem sistemde tanımlı tüm ilklerler üzerinde bir işlem gerçekleştirecekse, ilkellerle beraber tüm kompozitlerin de taranması gerekir.

Kompozitlerin kendisini oluşturan parçalara ayrılabilmesi için sistemdeki nesneler için tanımlı olan Patlat() metodu kullanılabilir.

2.9.6 Gerçekleme

1. Kompozit nesnelerin içindeki ilkellerhangi kompozite ait olduklarına dair bir sahip göstericisi tutabilirler. Bu, taramalar sırasında işe yarayacaktır.

Kompoziti oluşturan parçaların yönetimi; Kompozit nesneler parçaların eklenebilmesi ve çıkarılabilmesi için Ekle ve Çıkar gibi metodlara ihtiyaç duyarlar. Bu metodların en üstteki soyut sınıfta mı, yoksa sadece temel kompozit sınıfında mı tanımlanacağı bir problem olarak karşımıza çıkabilir.

Birinci durumda, yani tüm görsel nesnlerin türediği sınıfta bu gerçekleme yapılırsa, gereken şeffaflık sağlanabilir, ancak bu sefer de ağacın yapraklarını oluşturan nesnelere ekleme ve çıkarma işlemleri yapılabilir, bu da istenmeyen bir durumdur.

İkinci durum ise sadece kompozite özel metodları tanımlayacaktır ve temel nesnelerle kompozitlerin erişim arayüzleri farklı olacaktır.

- Parçaların depolanabilmesi için kullanılacak olan veri yapısı, sistemin ihtiyaçlarına göre seçilebilir. Bu bir bağlı liste, ağaç, vektör, çırpma tablosu (hash table) veya basit bir dizi de olabilir. Iterator tasarım deseninin parçalara erişimde kullanımı,

Kompozit'in nesnelere erişiminde alt tarafta bulunan veri yapısından bağımsız olmasını ve ihtiyaçlara göre kolaylıkla değişitirilebilmesini sağlayacaktır. GeoCAD sisteminde kullanım kolaylığı ve basitliği göz önüne alınarak vektör yapısı kullanılmıştır.

2.9.7 Kod Örneği

Aşağıda, GeoCAD sisteminde kullanılan temel kompozit nesnesinin kısmi kodu verilmiştir.

```
class CKompozitNesne : public CNesne
{
 private:
 CGeoVektor *NesneListesi;

 public:
 CKompozitNesne(){
  NesneListesi = new CGeoVektor(5);
 }

 void NesneEkle(CNesne *yeniNesne){
  NesneListesi->Ekle(yeniNesne);
 }

void NesneSil(CNesne *yeniNesne){
  NesneListesi->Sil(yeniNesne);
 }

 // Listedeki tüm nesneler için Ciz metodunu çağır
 virtual void Ciz(CEkranParam *Ekr){
  CSorgulayici *Iter = NesneListesi->SorgulayiciGetir();
  for(;Iter->DahaVar();){
   CNesne *N = (CNesne*)Iter->Sonraki();
   N->Ciz(Ekr);
  }
 };
```

```
// listedeki tüm nesneler için kaydır metodunu kullan
virtual void Move(double dx,double dy){
 CSorgulayici *Iter = NesneListesi->SorgulayiciGetir();
 for(;Iter->DahaVar();){
  CNesne *N = (CNesne*)Iter->Sonraki();
  N->Move(dx,dy);
 }
};
};
```

Görüldüğü gibi kompozit nesneler üzerlerinde kendisini oluşturan birimleri taşıyan bir liste ve bu listeye yeni elemanlar eklemek veya varolan elemanları silmek için metodlar taşırlar.

Diğer metodlar ise özel bir şey içermez, sadece gelen emrin listesindeki ilkel nesnelere aynen iletilmesinden ibarettir. Bir kompozitin ekrana çizilmesi için Ciz metodu çağrıldığı zaman, Kompozit listesindeki tüm elemanların çiz metodunu art arda çağırır.

3. SONUÇLAR VE TARTIŞMA

Tasarım desenleri iyice incelenip anlaşıldığı zaman nesneye yönelik tasarımı hem kolaylaştırır hem de gelişim sürecinin otomatikleşmesine edilmesine yardımcı olur. Özellikle büyük projelerin gerçeklenmesinde tasarım desenlerinin kullanılması hem hata oranını azaltacak hem de genişleyebilir ve bakımı kolay bir sistem ortaya çıkmasına yardımcı olacaktır. Tasarım desenleri çok yeni bir konu olduğundan üzerinde hala pek çok tartışma ve araştırma yapılmaktadır. Sunulan desenlere hemen her gün yenileri eklenmektedir. Bu desenlerin çoğu temel desenlere eklemeler yapılarak veya bir kaç desen birleştirilerek elde edilmektedir.

Uzman programcılarla beraber yapılan çalışmalar, bilgi ve tecrübenin sentaks üzerinde değil daha çok kavramsal yapılar, algoritmalar ve yapılar üzerinde yoğunlaştığını göstermiştir. Tasarımcılar çoğu zaman bu tür yapıları hangi gösterimde saklayacakları hakkında ek düşünmezler. Tasarım desenleri , tasarımcıların birbirleriyle haberleşmesi, bilgilerini paylaşması için ortak bir tasarım sözlüğü sağlar. Bu sayede daha üst bir soyutlama seviyesinde fikirler ortaya atılabilmektedir.

Tasarım desenlerinin ortaya çıkan bir başka faydası da mevcut sistemlerin anlaşılabilmesinde kolaylık sağlamasıdır. Büyük nesneye yönelik sistemlerin çoğu bu desenlerden en az birini kullanırlar. Bu sistemleri inceleyen kişiler tasarım desenleri konusuna aşina olmadıkları için genelde sınıf hiyerarşisinin ve kalıtımın değişik kullanımlarını anlamakta güçlük çekmektedirler. Tasarımların öğrenilmesi kullanıcıyı daha iyi bir tasarımcı yapar. Nesneye yönelik programlarla uzun süre haşır neşir olan kişiler uzun bir süre sonra bu tasarım desenlerini kendi kendilerine keşfedeceklerdir, ancak mevcut seçenekleri, bu seçeneklerin avantaj ve dezavantajlarını, varyasyonlarını bilmek tasarımcıya büyük bir avantaj sağlar.

Nesneye yönelik tasarımlarım metodları, kullanıcılara nasıl daha iyi tasarım yapacaklarını öğretmek, tasarımların yapılış aşamasını standartlaştırmak amacına yöneliktirler. Tasarım metodları tasarım sırasında ortaya çıkabilecek problemleri ve bunlara nasıl çözüm bulunabileceği konusundadır. Ancak tasarım metodları uzman kullanıcıların tecrübelerini tasarımlara aktarma işini yapmaz. Bu noktada tasarım desenleri bu konudaki eksikliği gidermektedir. Tasarım desenleri kalıtım,

(inheritance), tekrar tanımlama (polymorphism) ve dinamik harmanlama (dynamic binding) gibi temel nesneye yönelik yapıları kullanarak, çözümler üretir.

Tasarım desenlerinin bir başka iyi yönü ise, analiz modellerini uygulamaya geçirmede kullanıcıya yüksek hız kazandırmasıdır.

Tasarım desenlerinden daha iyi anlaşılması ve faydalarının gösterilebilmesi amacıyla, GeoCAD CAD çekirdeğinin yazılmasında da faydalanılmıştır.

GeoCAD bir CAD uygulaması değildir, sadece sistemin çekirdeğini tanımlar. Bu yüzden geniş bir fonksiyonellik sunmaz, temel veri yapılarını ve nesne hiyerarşisini belirler ve üzerine daha büyük ve karmaşık yapıların oturtulması için gerekli olan temelleri ve mimariyi sunar.

Verilen örnekte GeoCAD Cad çekirdeği ve en az seviyede fonksiyonellik belirlenmiştir. GeoCAD'ın bazı özelliklerine göz atacak olursak;

- Birden fazla proje üzerinde çalışabilme
- Bir projede farklı ekran parametreleri ile pencere açabilme (Tasarım deseni: Gözlemci)
- Hızlı 2D temel grafik algoritmaları
- Sistem log'u ve debug özellikleri (Tasarım Deseni : Tekil)
- Esnek Palet işlemleri
- Temel iki boyutlu standart CAD nesneleri (Tasarım desenleri: Fabrika, Prototip, Tekil)
    1. Nokta (Koordinatlı, hızlı, çemberüzeri, grup...)
    2. Doğru (Paralel, Normal)
    3. Çember (üç noktadan, iki noktadan, merkezden serbest)
    4. Yay (üç noktadan, serbest)
    5. Poligon
    6. Sembol

- Nesne özellik değiştirme işlemleri
    1. Yerdeğiştir
    2. Döndür

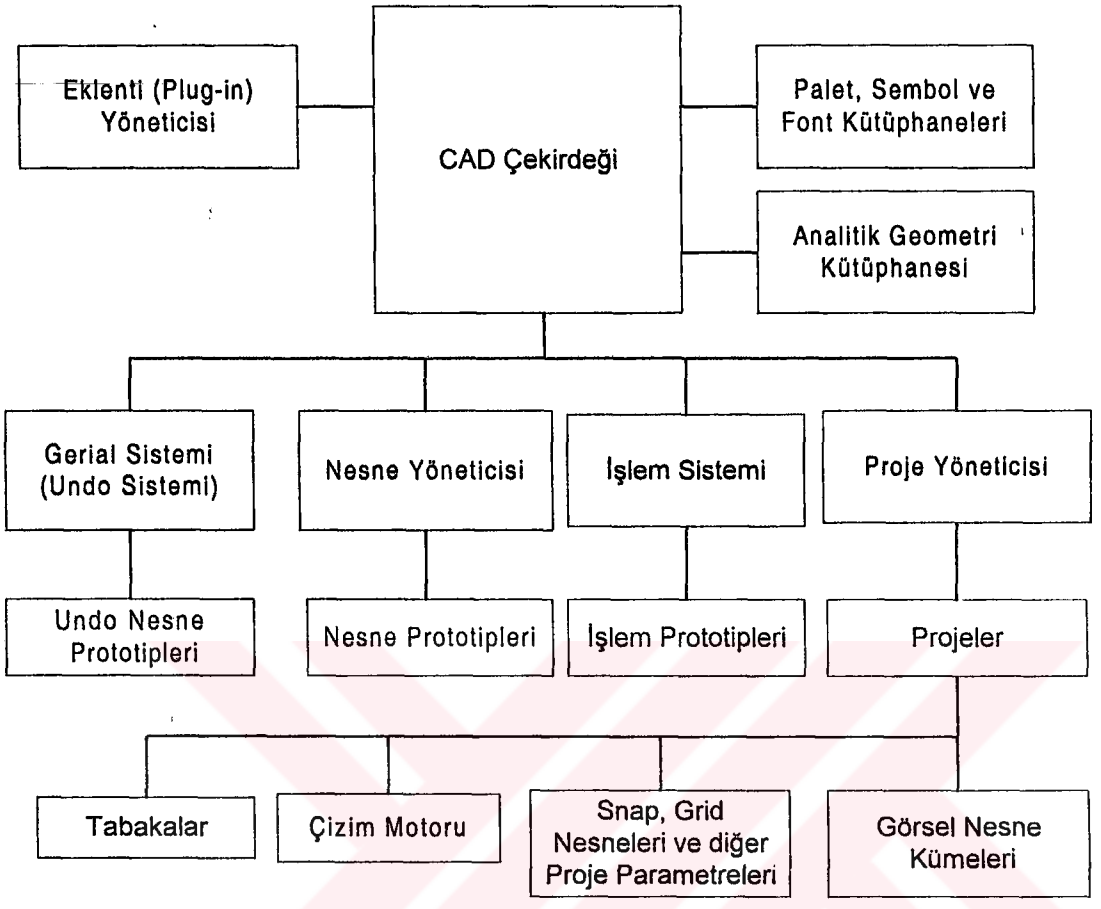
3. Ölçekle
4. Sil
5. Kopyala
6. Yapıştır
7. Özellikleri listele (Tasarım deseni: Memento)

- Snap (Yakala)
  1. Nesne üzeri
  2. uç
  3. cember ustu
  4. Cember kenarı
  5. merkez
  6. kesisim
  7. teğet
  8. Grid
- Nesne seçim fonksiyonları
  1. Tekil
  2. Lasso
  3. Dikdörtgen
- Sınırsız Undo-Redo işlemi (Tasarım deseni: Memento, Tekil)
- Undo temel sınıfları
- Tampon bellekli hızlı undo-redo
- Çok esnek komut özelliği (Tasarım deseni: Command)
- Komutlar ayrı sınıflar şeklinde tanımlanır
- Komut prototiplerini tutan Komut yorumlayıcı (Tasarım deseni: Prototip)
- Sonradan komut ekleyebilme (Plug-in)
- Kullanıcı tanımlı nesne ve grup nesne imkanı (Tasarım deseni: Kompozit)
- Proje başına 256 tabaka ve Fonksiyonel Tabaka yöneticisi (Tasarım deseni: Gözlemci)
  1. 256 farklı tabaka ile çalışma imkanı

2. Her tabaka için ayarlanabilen özellikler,
3. Renk
4. Tabaka Adı
5. Görünür/Görünmez
6. Kilitli/Açık
7. Alan Boyama açık/ Alan Boyama Kapalı
8. Aktif Tabaka
9. Tabaka Numarası
10. Birden fazla tabakayı seçip üzerinde işlem yapma imkanı
11. Silme
12. Seçime ekleme
13. Seçimden çıkarma
14. Filtreleme özellikleri, sadece belirlenen özelliklerdeki tabakaların görünür olması kolaylıkla sağlanabilir
15. Tabaka menüsü ekranda dururken kullanıcı çalışmasına devam edebilir.
16. Her proje kendi tabakalarına sahiptir, çalışma anında projeden projeye geçildiği anda tabaka ekranı da o projenin tabakalarını gösterecek şekilde değişir.
17. Projeler arasında tabaka aktarım işlemi ve bunun için sürükle-bırak desteği
18. Proje içindeki Tabaka seçici ile bağlantılı çalışma
19. Tabakaları adlarına gore sıralayabilme imkanı
20. Tabakaları kaydedebilme ve okuyabilme

- Projeyi dosyaya kaydedebilme ve okuma imkanı (Tasarım deseni: Eklenebilir Fabrika).
- Binary format
- Text formatı
- Thumbnail görüntü desteği
- Diğer formatlardaki dosyaları okumak ve yazmak için kolaylıklar.

Sistemin ve modüllerinin kabaca bir şemasını Şekil 3.1'de görülmektedir



Şekil 3.1 GeoCAD sistemi

GeoCAD Sistemi ileride yapılacak çalışmalarla daha da zenginleştirilebilir ve komple bir CAD sistemi haline getirilebilir.

Tasarım desenleir konusu ülkemizde oldukça az tanınmaktadır. Tanınmasının yazılım endüstrimize önemli faydalar getireceğine inanıyorum

KAYNAKLAR

[1] **Alexander, C., Ishikawa S., and Silverstein M.** 1977. *A Pattern Language - Towns-Buildings-Construction.* Oxford University Press

[2] **Gamma E., Helm R., Johnson R, Vlissides J.** 1995 *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley

[3] **Buschmann F., Meunier R., Rohnert H., Sommerland P., Stal M.** 1996 *A System of Patterns* John Wiley & Sons Ltd.

[4] **Pree W.** 1994 *Design Patterns for Object-Oriented Software Development* Addison-Wesley

[5] **Larman C.** 1997 *Applying UML and Patterns* Prentice Hall

[6] **Vlissides J.** 1999, *Tooled Compozite* C++ Tech. Report, September, Vol. 11,No. 8.

[7] **Tepfenhart W., Cusick J.** *1997 A Unified Object Topology* IEEE software January 31-35

[8] **Monroe R. Kompanek A., Melton R., Garlan D.** 1997 *Architectural styles, design Patterns, and Objects* IEEE Software January 43-52

[9] **Kerth N, Cunningham W.** 1997 *Using Patterns to improve our architectural Vision* IEEE Software January 53-59

[10] **Coad P.** 1992 *Object Oriented Patterns* Communications of the ACM September Vol. 35 No 9

[11] **Mellor S.** , Johnson R. 1997 *Why explore Object Methods, Patterns and Architectures?* IEEE Software January 27-30

[12] **Coplien J.** 1997 *Idioms and Patterns as Architectural Literature* IEEE Software January 36-42

ÖZGEÇMİŞ

Mehmet Dündar AKIN 1974 yılında Karaman'da doğdu. İlk ve ortaokulu Karaman'da okudu ve 1985 yılında Karaman Lisesi'ne girdi. 1991 Yılında Yıldız Teknik Üniversitesi Bilgisayar bilimleri ve mühendisliği bölümünü kazandı. Temmuz 1996'da lisans eğitimini tamamlayan Mehmet Dündar akın'ın lisans tezi görüntü işleme fonksiyonları hakkındadır.Ekim 1996'da İstanbul Teknik Üniversitesi Fen Bilimleri Enstitüsü, Kontrol ve Bilgisayar Mühendisliği Anabilimdalı, Kontrol ve Bilgisayar Mühendisliği bölümüne giren Akın'ın ilgi alanları Nesneye yönelik tasarım, tasarım desenleri ve grafik uygulamalarıdır.