Erciyes Üniversitesi
Bilgisayar Mühendisliği Bölümü

BZ 313 Yazılım Mühendisliği

18. Tekrar Kullanım ve Tasarım Şablonları

Yazılımın Yeniden Kullanımı

Mevcut bileşenleri yeniden kullanmak için bir program tasarlamak genellikle iyidir. Bu, daha düşük maliyetle daha iyi yazılım elde etmeyi sağlar.

Yeniden kullanımın potansiyel faydaları

- Azaltılmış geliştirme süresi ve maliyeti
- Olgun bileşenlerin güvenilirliği artırılır
- Paylaşılan bakım maliyeti

Yeniden kullanımın potansiyel dezavantajları

- Uygun bileşenleri bulmada zorluk
- Bileşenler uygulama için uygun olmayabilir
- Kalite kontrol ve güvenlik bilinmiyor olabilir

Yazılımı Değerlendirme

İyi bilinen geliştiricilerin yazılımlarının iyi yazılmış ve test edilmiş olması muhtemeldir ancak özellikle sıra dışı uygulamalarla birleştirildiğinde yine de hatalar ve güvenlik zayıflıkları olacaktır.

Yazılımın yeni bir geliştirme ekibinin yazacağından çok daha iyi olması muhtemeldir.

Ancak bazen genel amaçlı yazılım kullanmak yerine dar tanımlı bir amaç için kod yazmak daha mantıklıdır.

Bakım

Hem ticari hem de açık kaynaklı yazılımları değerlendirirken bakıma dikkat edin. Yazılım uzun vadede bakıma devam edecek bir kuruluş tarafından destekleniyor mu?

Yeniden Kullanım: Açık Kaynak Yazılım

Açık kaynaklı yazılımların kalitesi büyük farklılıklar gösterir.

- Sorunları bildirme ve çözme süreçleri nedeniyle, Linux, Apache, Python, Lucene vb. gibi büyük sistemler çok sağlam ve sorunsuz olma eğilimindedir. Genellikle ticari eşdeğerlerinden daha iyidirler.
- Hadoop gibi daha deneysel sistemler sağlam çekirdeklere sahiptir, ancak daha az kullanılan özellikleri, en iyi yazılım ürünlerinin titiz kalite kontrolüne tabi tutulmamıştır.
- Diğer açık kaynaklı yazılımlar kalitesiz olabilir ve üretim sistemlerine dahil edilmemelidir.

Uygulama Paketlerinin Değerlendirilmesi

İş fonksiyonlarına yönelik uygulama paketleri, SAP ve Oracle gibi şirketler tarafından sağlanmaktadır. **Muazzam yetenekler** sağlarlar ve bir kuruluşu, yasalar değiştiğinde finansal sistemleri güncellemek gibi görevlerden kurtarırlar.

Çok pahalıdırlar:

- Satıcıya lisans ücretleri.
- Mevcut sistemlerde yapılan değişiklikler ve satıcıdan özel kod.
- Bunları kurarken organizasyonda kesinti.
- Uzun süreli bakım maliyetleri.
- Farklı bir satıcıya geçmenin maliyeti çok büyük.

Bir firmadan başka firmaya geçmek yüzlerce milyon dolara mal olabilir.

Eğer böyle bir karara dahilseniz, **çok kapsamlı bir fizibilite çalışması yapılması** konusunda ısrar edin. Karar vermeden önce en az bir yıl ayırmaya ve birkaç milyon dolar harcamaya hazır olun.

Değişim için Tasarım: Bileşenlerin Değiştirilmesi

Yazılım tasarımı, yaşam döngüsü boyunca sistemdeki olası değişiklikleri öngörmelidir**.**

Yeni satıcı veya yeni teknoloji

Tedarikçiler iflas ettiğinden, yeterli desteği sağlamayı bıraktığından, fiyatını artırdığından vb. veya başka bir kaynaktan gelen yazılım daha iyi işlevsellik, destek, fiyatlandırma vb. sağladığı için bileşenler değiştirilir.

Bu, açık kaynak veya satıcı tarafından sağlanan bileşenler için geçerli olabilir.

Değişim için Tasarım: Bileşenlerin Değiştirilmesi

Yeni uygulama

Orijinal uygulama, düşük performans, yetersiz yedekleme ve kurtarma, sorun gidermenin zor olması veya büyümeyi ve sisteme eklenen yeni özellikleri destekleyememesi gibi sorunlu olabilir.

Örnek. Bm.erciyes.edu.tr ilk olarak bir portal sistemi kullanılarak tamamlandı. Bu, talep edilen ve bakımı zor olan önemli uzantıları desteklemiyordu. ASP.NET/MSSQL kullanılarak yeniden yazıldı.

Değişim için Tasarım: Bileşenlerin Değiştirilmesi

Gereksinimlere eklemeler

Bir sistem üretime geçtiğinde, kullanıcı arayüzü tasarımında ekstra işlevsellik ve geliştirme için hem zayıflıkları hem de fırsatları ortaya çıkarmak olağandır.

Örneğin, veri yoğun bir sistemde, ekstra raporlar ve verileri analiz etme yolları için talepler olacağı neredeyse kesindir.

Geliştirme talepleri genellikle başarılı bir sistemin işaretidir. Müşteriler gizli geliştirme olasılıklarını bilir ve onlara sahip olmak ister.

Değişim için Tasarım: Bileşenlerin Değiştirilmesi

Uygulama alanındaki (domain) değişiklikler

Çoğu uygulama alanı, örneğin iş fırsatları, harici değişiklikler (yeni yasalar gibi), birleşmeler ve devralmalar, yeni kullanıcı grupları, yeni teknoloji vb. nedeniyle sürekli olarak değişir.

Uygulama alanı değiştiğinde tamamen yeni bir sistem oluşturmak nadiren mümkün olur. Bu nedenle, mevcut sistemleri değiştirmek daha mantıklıdır. Bu, kapsamlı bir yeniden yapılandırmayı içerebilir, ancak mevcut kodu mümkün olduğunca yeniden kullanmak önemlidir.

Tasarım Desenleri

Tasarım desenleri

Tasarım desenleri, çeşitli sistemlerde kullanılabilen şablon tasarımlarıdır. Sınıfların zaman içinde gelişen bir sistemde yeniden kullanılmasının muhtemel olduğu durumlarda özellikle uygundurlar.

Tasarım Desenleri

Kaynaklar:

E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994

The following discussion of design patterns is based on Gamma, et al., 1994, and Bruegge and Dutoit, 2004.

Wikipedia, kod örnekleriyle UML ve diğer gösterimleri kullanan birçok tasarım deseni hakkında iyi örneklere sahiptir.

Kalıtım (Inheritance) ve Abstract (Soyut) Sınıf

Tasarım desenleri, kalıtım ve soyut sınıfları kapsamlı bir şekilde kullanır.

Sınıflar, kalıtım kullanılarak diğer sınıfları içerecek şekilde tanımlanabilir. Genelleme sınıfına üst sınıf (superclass), özelleştirmeye ise alt sınıf (subclass) denir.

Soyut sınıf

Soyut sınıflar, soyut yöntemler içeren üst sınıflardır ve somut alt sınıfların soyut yöntemleri uygulayarak bunları genişleteceği şekilde tanımlanır. Soyut bir sınıftan türetilen bir sınıfın başlatılabilmesi için, üst sınıflarının tüm soyut yöntemleri için somut yöntemler uygulaması gerekir.

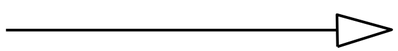
Delegation

Delegation

Standart tasarım desenlerinin pek çoğu delegation’ları kullanır. Delegate, belirli bir olay olduğunda bir nesnenin başka bir nesneye haber göndermesine olanak tanır.

Yeniden kullanım gerektiğinde kalıtıma alternatif bir yapıdır.

Notation

ClassName	İtalik sınıf adı, soyut bir sınıfı belirtir
	dependency
	delegation
	inheritance

Adapter (Wrapper): Eski Kodun Etrafından Sarmalama

Sorunun açıklaması:

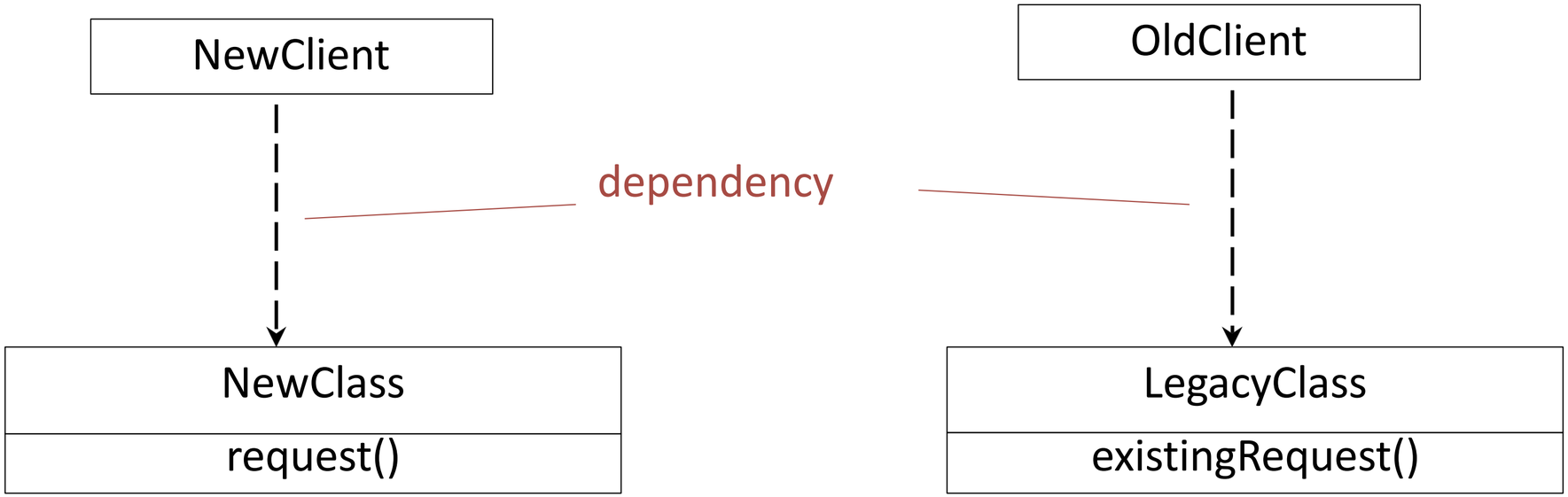
Eski bir sınıfın (legacy) interface'i client tarafından beklenen farklı bir interface'e dönüştürülür, böylece client ve eski sınıf değişiklik yapmadan birlikte çalışabilir.

Bu sorun genellikle, uzun vadeli planın eski sistemi aşamalı olarak kaldırmak olduğu bir geçiş döneminde ortaya çıkar.

Örnek:

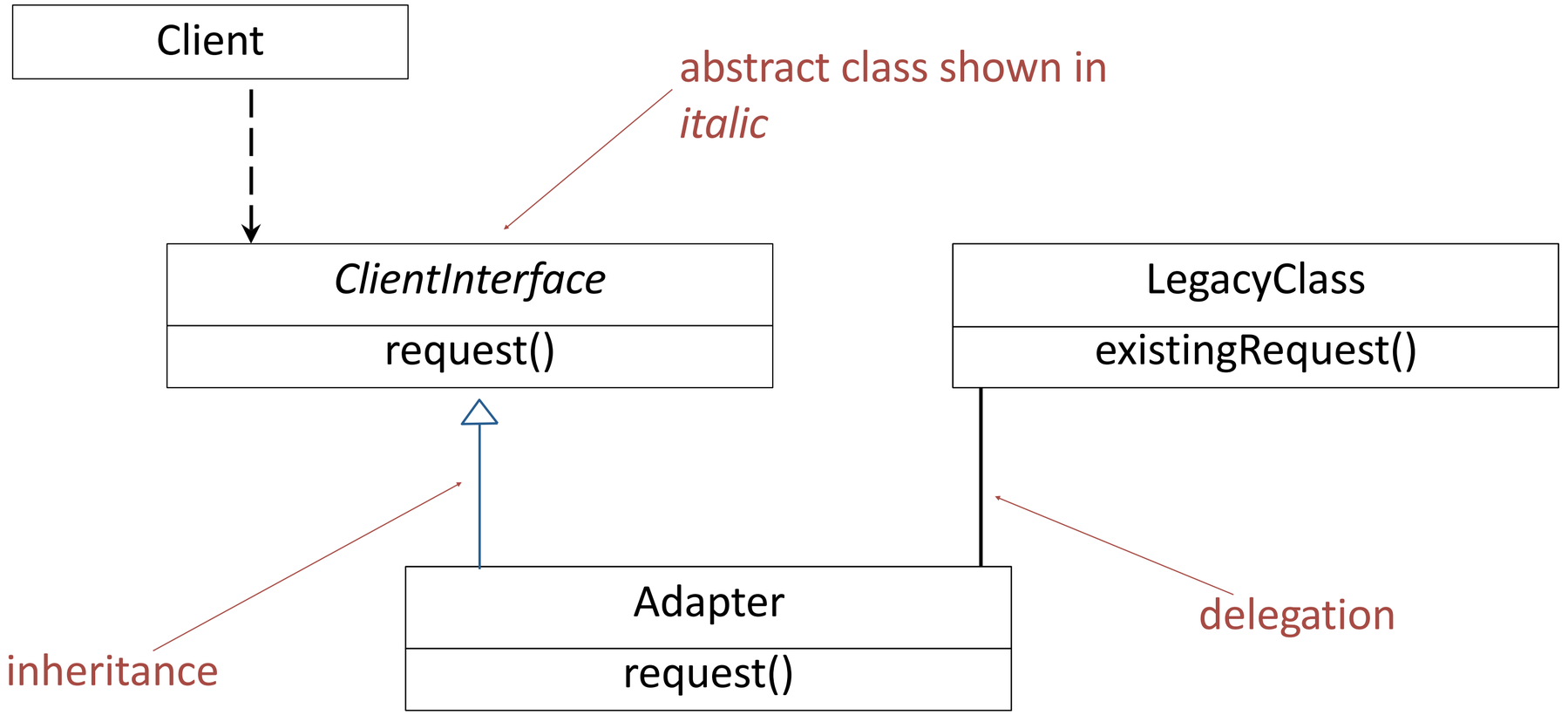
Farklı bir müşteri için tasarlanmış bir bilgi alma sistemine erişmek için bir web tarayıcısını nasıl kullanırsınız?

Adapter Tasarım Deseni: Sorun



Geçiş sırasında NewClient LegacyClass ile nasıl kullanılabilir??

Adapter Tasarım Deseni: Çözüm Sınıfı Diyagramı



Adapter Tasarım Deseni: Sonuçlar

Adapter tasarım deseni her kullanıldığında aşağıdaki **sonuçlar** geçerlidir.

- Client ve LegacyClass, ikisinde de değişiklik yapılmadan birlikte çalışır.
- Adapter, LegacyClass ve tüm alt sınıflarıyla çalışır.
- Client bir alt sınıfla değiştirilirse yeni bir Adapter yazılması gerekir.

Bridge: Alternatif Uygulamalara İzin Verme

Ad: Bridge tasarım deseni

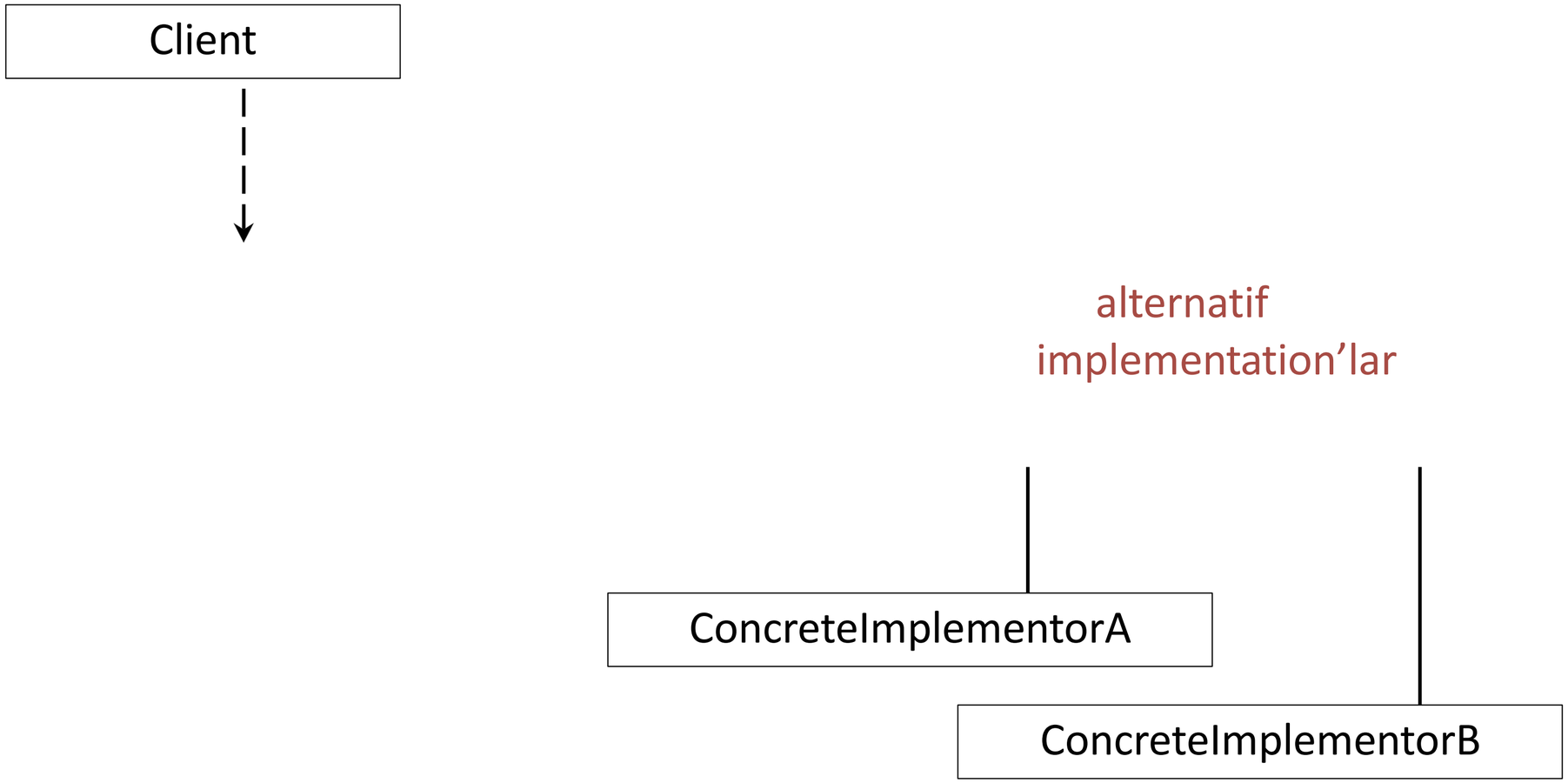
Sorunun açıklaması:

Muhtemelen çalışma zamanında farklı bir uygulamanın değiştirilebilmesi için bir arayüzü bir uygulamadan ayırın (örneğin, aynı arayüzün farklı uygulamalarını test etme).

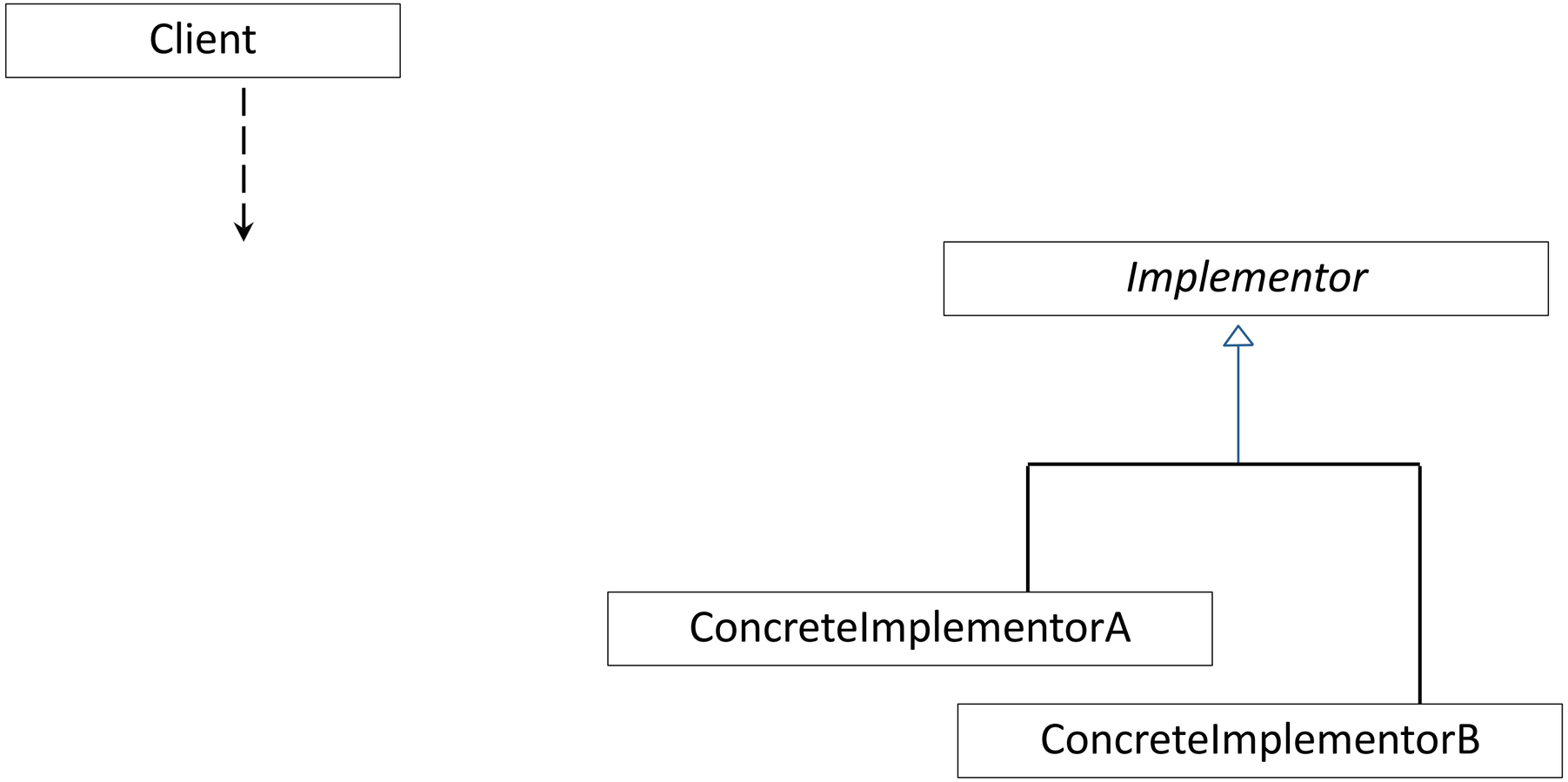
Bridge ve Adapter tasarım modellerinin karşılaştırılması

- Bridge: Değişmeyen tanımlı arayüzler
- Adapter: Birlikte çalışması gereken eski ve yeni arayüzler

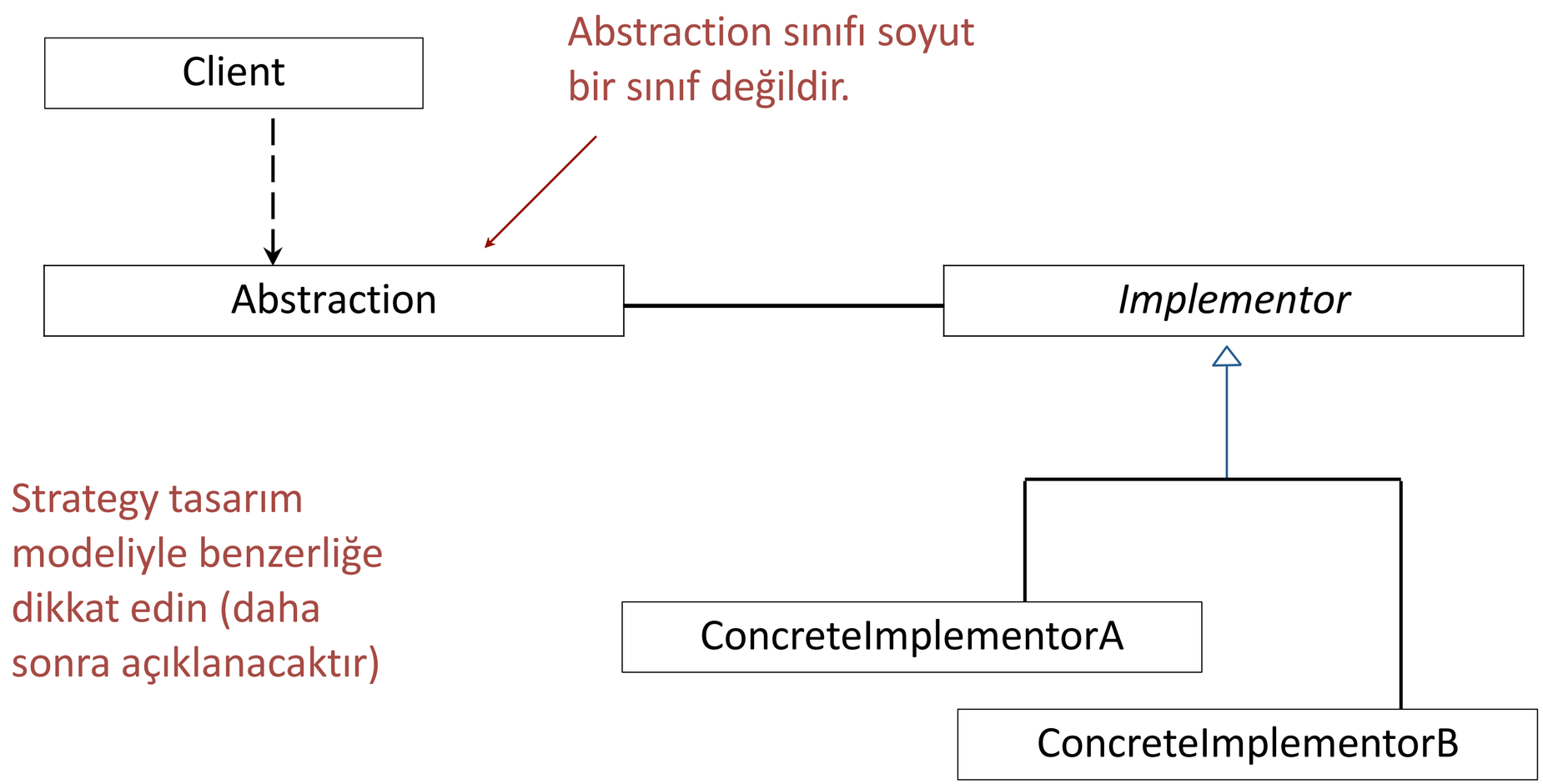
Bridge: Sınıf Diyagramı



Bridge: Sınıf Diyagramı



Bridge: Sınıf Diyagramı



Strategy tasarım modeliyle benzerliğe dikkat edin (daha sonra açıklanacaktır)

Bridge: Alternatif Uygulamalara İzin Verme

Çözüm:

- Abstraction sınıfı, istemci tarafından görülebilen arayüzü tanımlar.
- Implementor, Abstraction için kullanılabilen alt düzey yöntemleri tanımlayan soyut bir sınıftır.
- Bir Abstraction örneği, karşılık gelen Implementor örneğine bir referans tutar.
- Abstraction ve Implementor bağımsız olarak geliştirilebilir.

Bridge: Sonuçlar

Sonuçlar:

- Client soyut ve somut uygulamalardan korunur.
- Arayüzler ve uygulamalar ayrı ayrı test edilebilir

Strategy: Encapsulating Algorithms

Name: Strategy Tasarım deseni

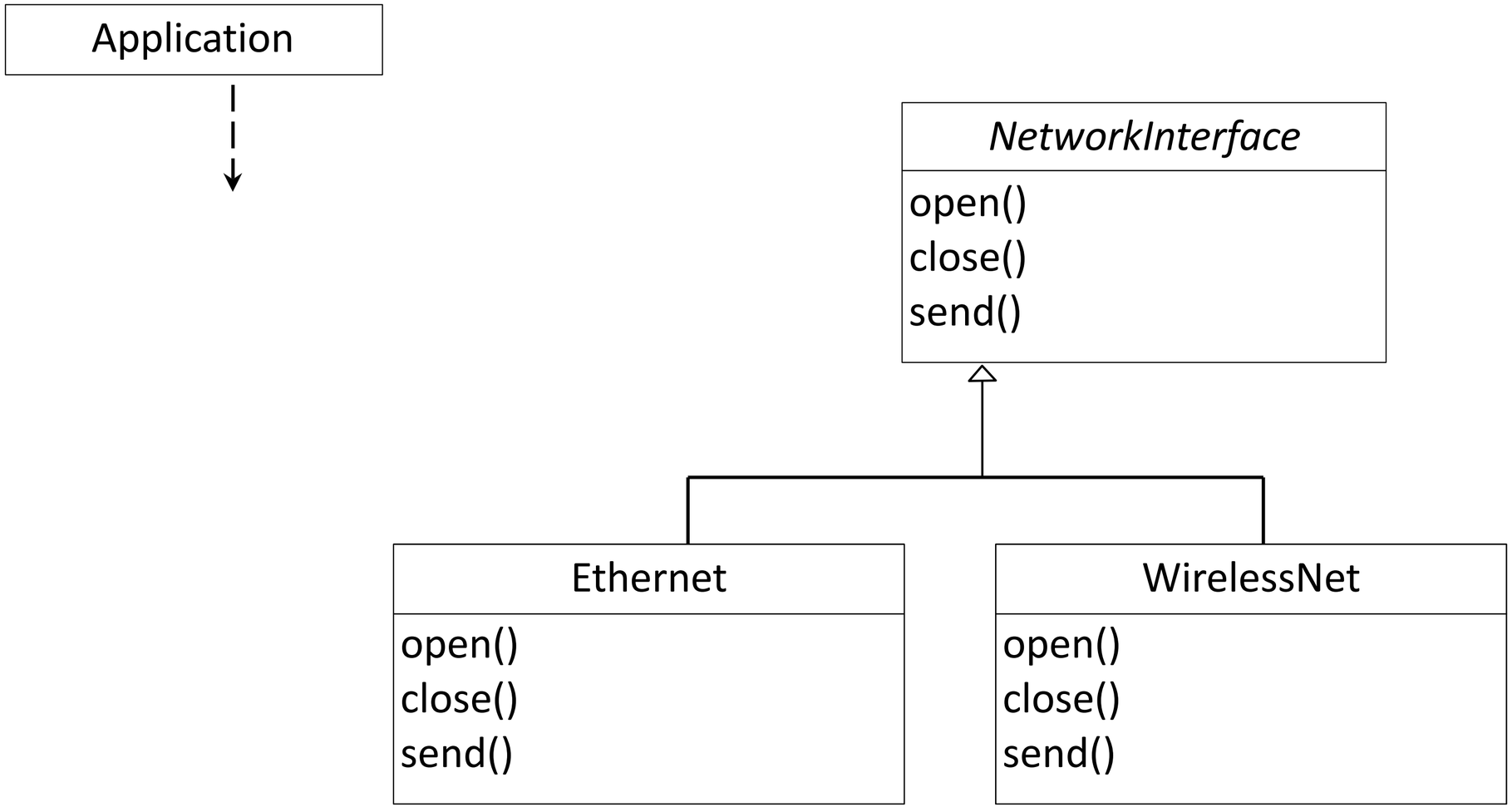
Örnek:

Bir mobil bilgisayar, kablosuz bir ağ ile kullanılabilir veya konum ve ağ maliyetlerine bağlı olarak ağlar arasında dinamik geçiş ile bir Ethernet'e bağlanabilir.

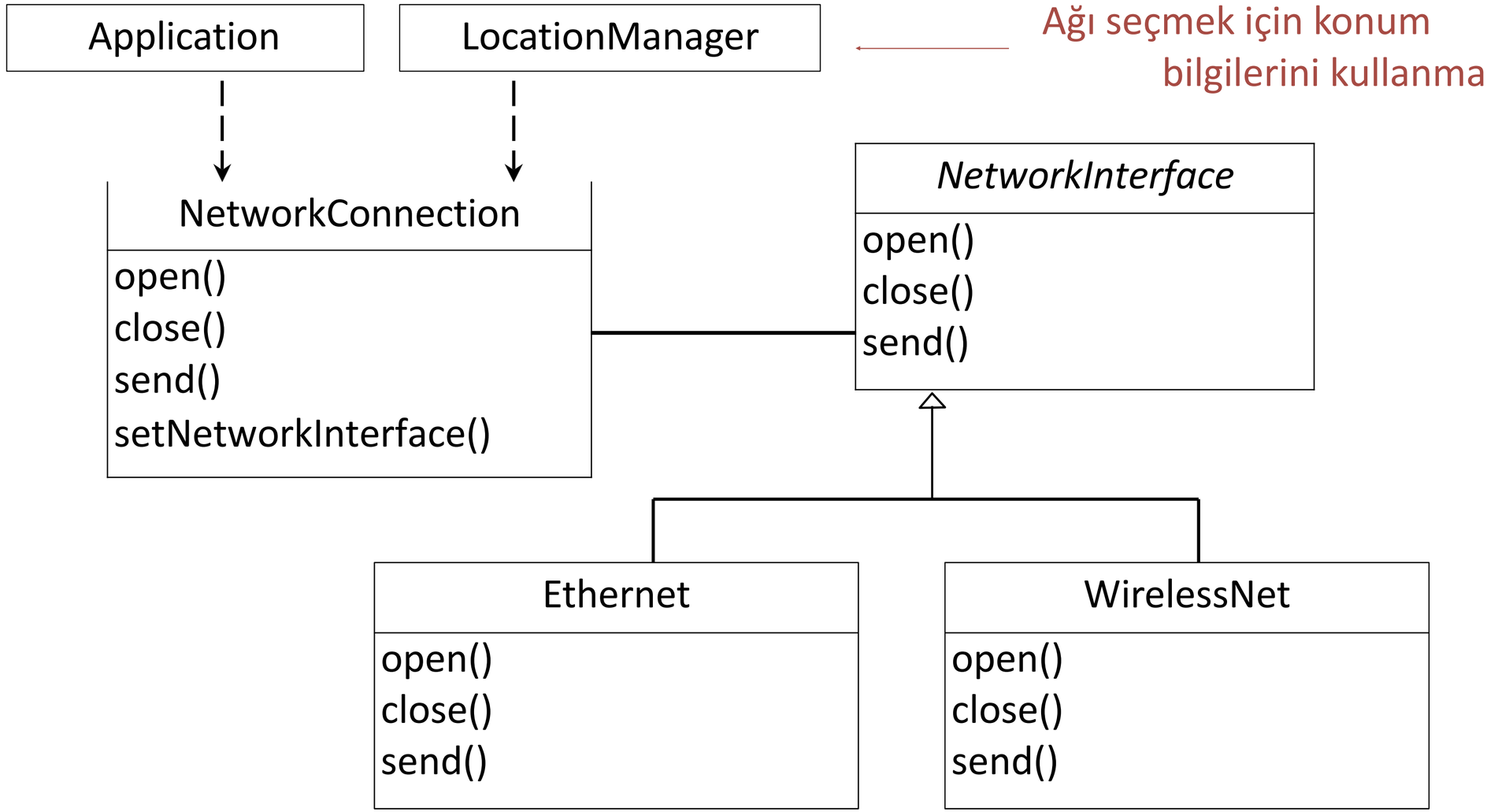
Sorunun açıklaması:

Farklı mekanizmaların şeffaf bir şekilde değiştirilebilmesi için policy-deciding class'ın bir dizi mekanizmadan ayrılması.

Strategy Örneği: Mobil Bilgisayar için Sınıf Diyagramı



Strategy Örneği: Mobil Bilgisayar için Sınıf Diyagramı



Strategy: Encapsulating Algorithms

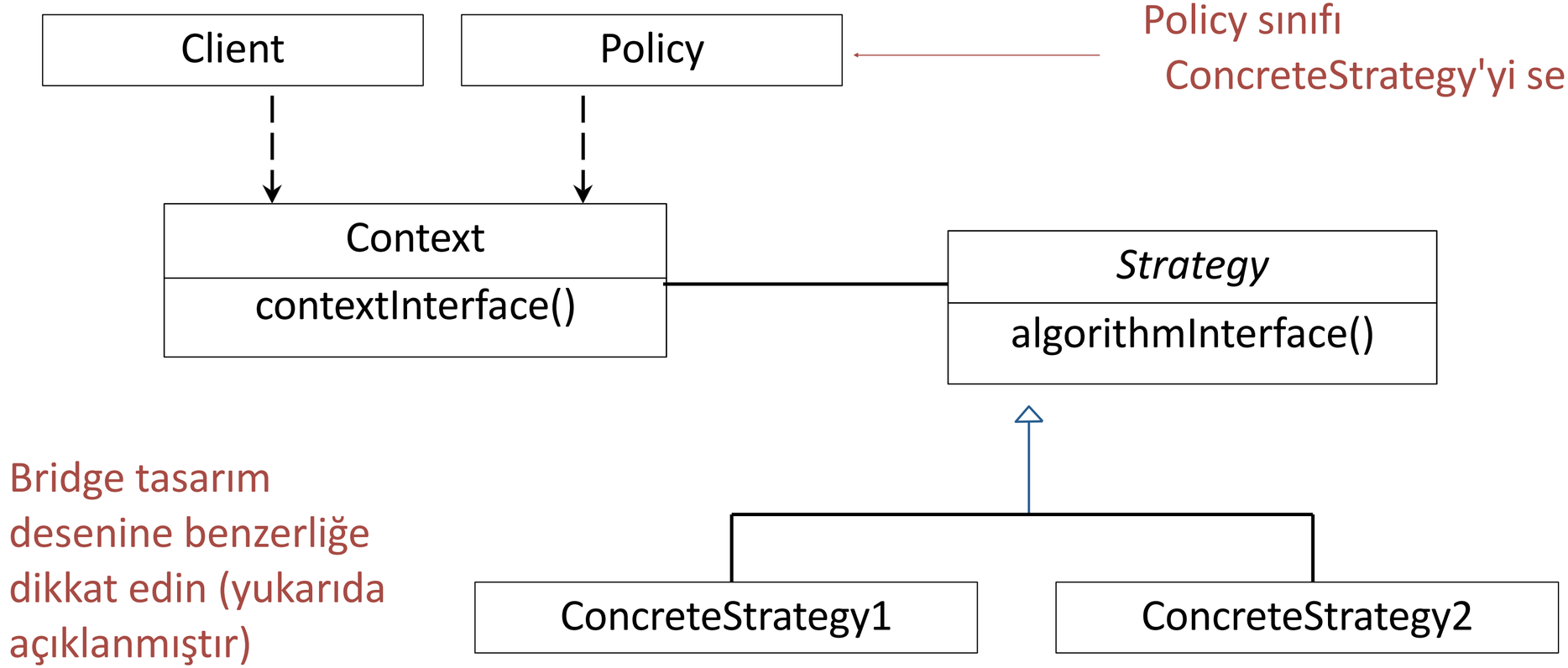
Çözüm:

Bir Client, bir Context tarafından sağlanan hizmetlere erişir.

Context hizmetleri, bir Policy nesnesi tarafından kararlaştırılan çeşitli mekanizmalardan biri kullanılarak gerçekleştirilir.

Strategy soyut sınıfı, Context'in kullanabileceği tüm mekanizmalar için ortak olan arayüzü açıklar. Policy sınıfı bir ConcreteStrategy nesnesi oluşturur ve Context'i bunu kullanacak şekilde yapılandırır.

Strategy: Sınıf Diyagramı



Bridge tasarım desenine benzerliğe dikkat edin (yukarıda açıklanmıştır)

Strategy: Sonuçlar

Sonuçlar:

- ConcreteStrategies, context'ten şeffaf bir şekilde ikame edilebilir.
- Policy, mevcut koşullar göz önüne alındığında hangi Strategy'nin en iyi olduğuna karar verir.
- Context veya Client değiştirilmeden yeni policy algoritmaları eklenebilir.

Facade: Encapsulating Subsystems

Name: Facade tasarım deseni

Sorunun açıklaması:

Bir dizi ilgili sınıf ile sistemin geri kalanı arasındaki bağlantı azaltılır.

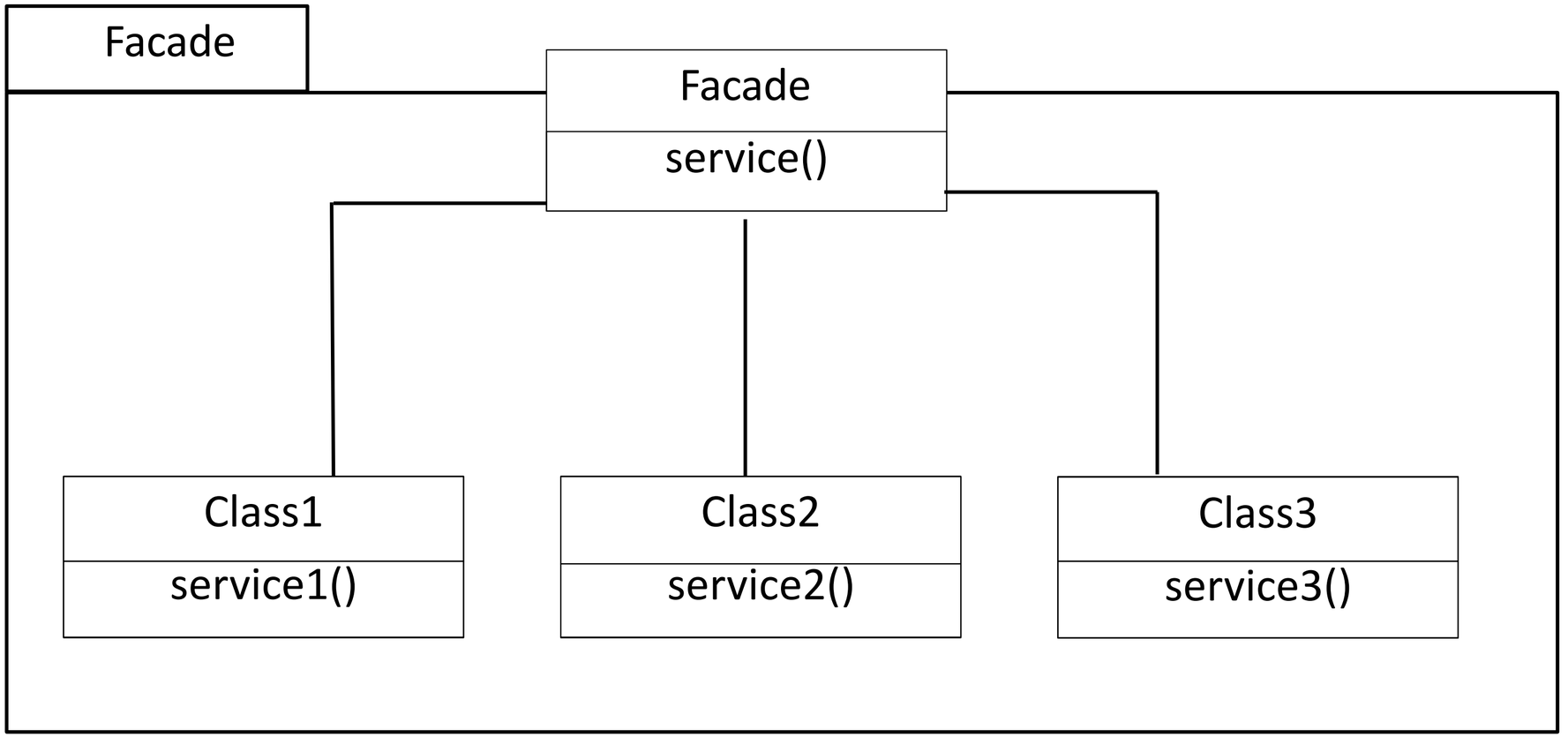
Örnek:

Bir Compiler birkaç sınıftan oluşur: LexicalAnalyzer, Parser, CodeGenerator, vb. Bir caller yalnızca, kapsanan sınıfları çağıran Compiler (Facade) sınıfını çağırır.

Çözüm:

Tek bir Facade sınıfı, alt düzey sınıfların yöntemlerini çağırarak bir alt sistem için üst düzey bir arabirim uygular.

Facade: Sınıf Diyagramı



Facade: Sonuçlar

Sonuçlar:

- İstemciyi bir alt sistemin alt düzey sınıflarından korur.
- Daha üst düzey yöntemler sağlayarak bir alt sistemin kullanımını basitleştirir.
- Alt düzey sınıfların istemcilerde değişiklik yapılmadan yeniden yapılandırılmasını sağlar.

Not. Facade desenlerinin tekrar tekrar kullanılması, katmanlı bir sistem sağlar.

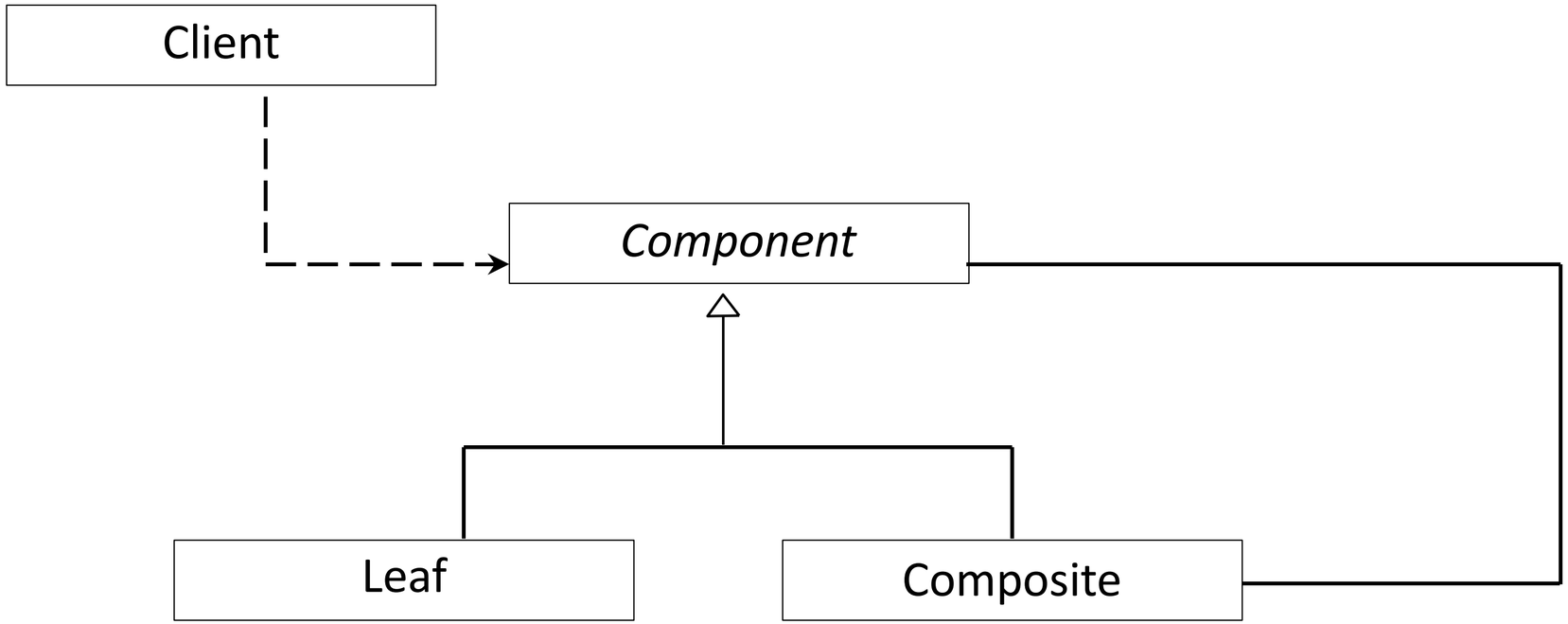
Composite: Özyinelemeli Hiyerarşileri Temsil Etme

Name: Composite tasarım deseni

Sorunun açıklaması:

Değişken genişlik ve derinlikte bir hiyerarşiyi temsil eder, böylece yapraklar (leaf) ve kompozitler (composite) ortak bir arayüz aracılığıyla eşit şekilde işlenebilir.

Composite: Sınıf Diyagramı



Composite: Özyinelemeli Hiyerarşileri Temsil Etme

Çözüm:

- Component arayüzü, Leaf ve Composite arasında paylaşılan hizmetleri belirtir.
- Composite, Component ile bir toplama ilişkisine sahiptir ve kapsanan her Component üzerinde yinelenerek her hizmeti uygular.
- Leaf hizmetleri asıl işi yapar.

Composite: Sonuçlar

Sonuçlar:

- Client, Leaf veya Composite ile ilgilenmek için aynı kodu kullanır.
- Leaf-specific davranış, hiyerarşiyi değiştirmeden değiştirilebilir.
- Hiyerarşiyi değiştirmeden yeni Leaf sınıfları eklenebilir.

Tartışma

Tartışma

Wikipedia'daki ilginç tartışmaya bakın (2019):

"Bu desenin kullanılması, çalışma zamanında bile bunları kullanan kodu değiştirmeden somut sınıfları değiştirmeyi mümkün kılar. Ancak, benzer tasarım desenlerinde olduğu gibi bu desenin kullanılması, kodun ilk yazımında gereksiz karmaşıklığa ve fazladan çalışmaya neden olabilir."

Örnek Sınav Sorusu

Spor malzemeleri üreten bir şirket, çevrimiçi spor malzemeleri satmak için bir sistem oluşturmaya karar verir. Şirket, ürettiği ekipmanın özelliklerini, pazarlama bilgilerini ve fiyatlarını içeren bir ***ürün veritabanına*** *zaten sahiptir.*

Çevrimiçi ekipman satmak için şirketin şunları oluşturması gerekir: bir müşteri veritabanı ve çevrimiçi müşteriler için bir ***sipariş sistemi.***

Plan, sistemi iki aşamada geliştirmektir. Aşama 1 sırasında, müşteri veri tabanının ve sipariş sisteminin basit versiyonları üretime getirilecektir. Aşama 2'de, bu bileşenlerde büyük geliştirmeler yapılacaktır.

Örnek Sınav Sorusu

Aşama 1 sırasındaki dikkatli tasarım, Aşama 2'de yeni bileşenlerin geliştirilmesine yardımcı olacaktır.

(a) Sipariş sistemi ve müşteri veri tabanı arasındaki arayüz için:

i Aşama 1'den Aşama 2'ye kademeli geçişe izin verecek bir tasarım deseni seçin.

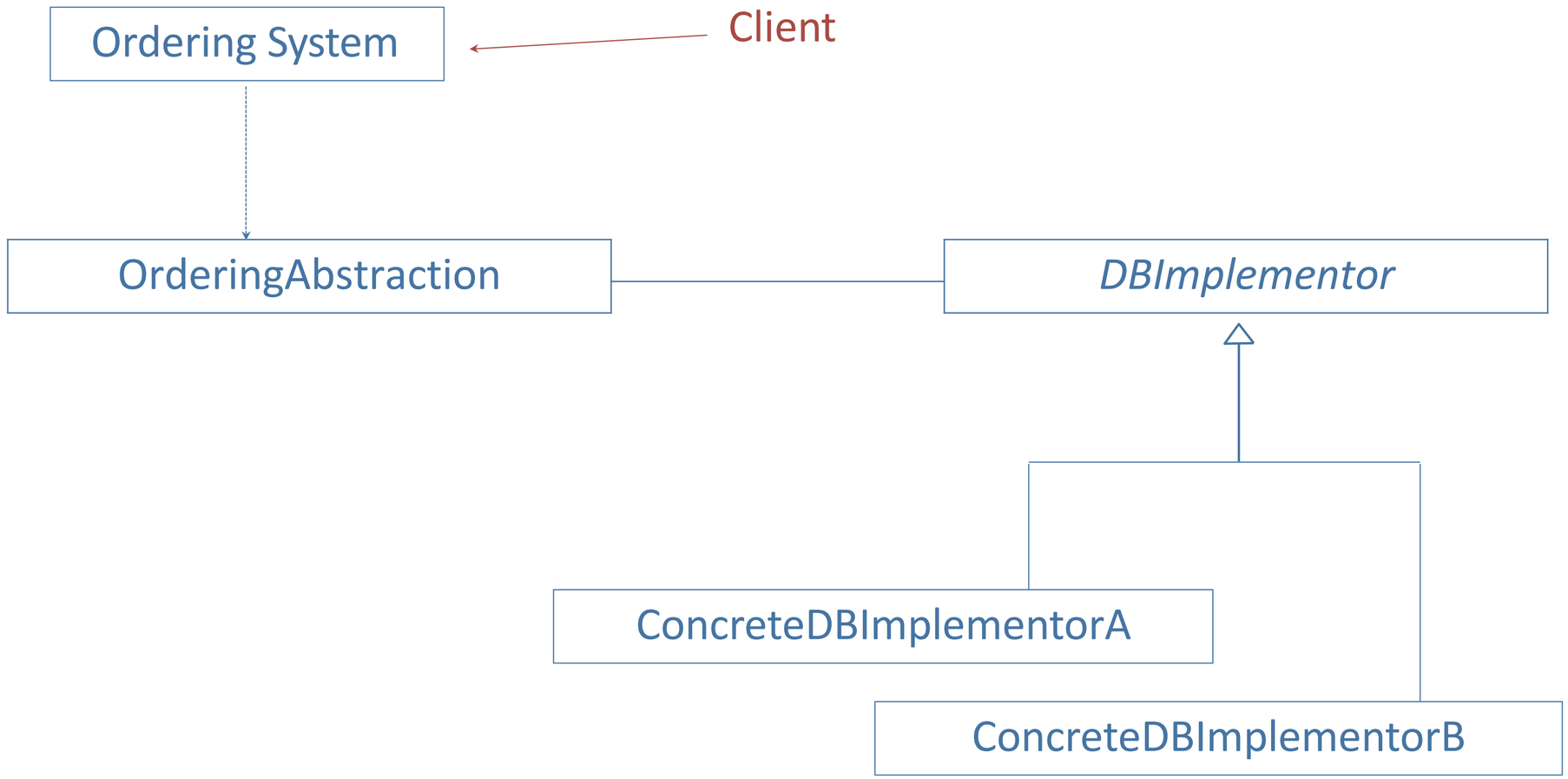
Bridge tasarım deseni

ii Bu tasarım deseninin Aşama 1'de nasıl kullanılacağını gösteren bir UML sınıf diyagramı çizin.

Diyagramınız abstract sınıflara, inheritance, delegasyona veya benzer özelliklere dayanıyorsa bunun diyagramınızda açıkça belirtildiğinden emin olun.

[Sonraki slayta bakın]

Örnek Sınav Sorusu



Örnek Sınav Sorusu

(c) Bu tasarım deseni nasıl destekliyor?:

i 2. Aşama'da sipariş sisteminde yapılan geliştirmeler?

OrderingAbstraction alt sınıflarına ayırarak

ii 2. Aşamada müşteri veritabanının olası bir değişimi?

Birkaç ConcreteDBImplementor sınıfına izin vererek

Eski (legacy) Sistemler

Bankalar, üniversiteler vb. tarafından kullanılanlar gibi birçok veri yoğun sistem eski sistemlerdendir. Kırk yıl önce toplu işleme, ana dosya güncelleme sistemleri olarak geliştirilmiş ve sürekli olarak değiştirilmiş olabilirler.

- Son değişiklikler, web, akıllı telefonlar vb. için müşteri arayüzlerini içerebilir.
- Sistemler bilgisayardan bilgisayara, işletim sistemleri arasında, farklı veritabanı sistemlerine vb. taşınmış olacaktır.
- Kuruluşlar birleşmeler vb. yoluyla değişmiş olabilir.

Bu tür eski sistemler için tutarlı bir sistem mimarisini sürdürmek çok büyük bir zorluktur, ancak yeni sistemler oluşturmanın karmaşıklığı o kadar büyüktür ki nadiren denenir.

Eski Sistemler

En Kötü Senaryo

Birkaç on yıl önce geliştirilen büyük, karmaşık bir sistem:

- Büyük bir kuruluşta veya bilinmeyen sayıda müşteri tarafından yaygın olarak kullanılır.
- Tüm geliştiriciler emekli olmuş veya ayrılmıştır.
- Gereksinim listesi bulunmamaktadır. Sistemin hangi işlevleri sağladığı ve kimin hangi işlevleri kullandığı belirsizdir.
- Sistem ve program dokümantasyonu eksik ve güncel tutulmamaktadır.
- Programlama dillerinin güncel olmayan sürümlerinde, aynı zamanda güncel olmayan sistem yazılımı kullanılarak yazılmıştır.
- Yıllar boyunca orijinal sistem mimarisini ve program tasarımını göz ardı eden çok sayıda yama yapılmıştır.
- Kapsamlı kod çoğaltma (duplication) ve artıklık (redundancy).
- Kaynak kodu kütüphaneleri ve üretim binary dosyaları uyumsuz olabilir.

Eski Gereksinimler

Planlama

Müşteri ile birlikte sistemi yeniden inşa etmek için bir plan geliştirin.

Müşteriler ve kullanıcılar tarafından görülen gereksinimler

- Kullanıcılar kimlerdir?
- Sistemi gerçekte ne için kullanıyorlar?
- Sistem, önemli olan belgelenmemiş özelliklere veya kullanıcıların güvendiği hatalara sahip mi?
- Sistemin kenarda kalmış kısımlarını kaç kişi kullanıyor? Nerede esnekler?

Sistem tasarımının ima ettiği gereksinimler

- Herhangi bir sistem belgesi varsa, gereksinimler hakkında ne diyor?
- Kaynak kodu, gereksinimler hakkında herhangi bir ipucu içeriyor mu?
- Eski donanım veya hizmetleri desteklemek için kod var mı? Eğer öyleyse, hala onları kullanan var mı?

Eski Kod

Kaynak kod yönetimi

- Kaynak kodun başlangıç sürümünü ve bu kaynak kodundan oluşturulan binary dosyaları oluşturmak için bir kaynak kodu yönetim sistemi kullanın.
- Yeniden oluşturulan sistemin mevcut sistemle karşılaştırılabilmesi için bir test ortamı oluşturun. Test senaryolarını toplamaya başlayın.
- Tüm satıcı yazılımlarının lisanslarını kontrol edin.

Eski Kod

Yazılımı yeniden oluşturma

Artımlı bir yazılım geliştirme süreci genellikle uygundur ve her artış tamamlandığında yayınlanır.

Aşağıdaki görevler, kodun durumuna bağlı olarak uygun herhangi bir sırayla ele alınabilir. Genellikle strateji, sistemin farklı bölümleri üzerinde bir dizi aşamada çalışmak olacaktır.

- Özgün sistem mimarisini ve program tasarımını anlayın.
- Kodun çoğu mimariyi ihlal etse ve bağdaştırıcılara ihtiyaç duysa bile, tanımlı arayüzlere sahip bir bileşen mimarisi oluşturun.
- Programlama dillerinin ve sistem yazılımlarının güncel sürümlerine geçin.
- Kaynak koduna sahip olmayan herhangi bir alt sistem varsa, gereksinimleri uygulayan yeni kod oluşturmak için bir geliştirme döngüsü gerçekleştirin.
- Yinelenen kod varsa, tek bir sürümle değiştirin.
- Gereksiz kodu ve eski gereksinimleri kaldırın.

İlerledikçe temizleyin.

Erciyes Üniversitesi
Bilgisayar Mühendisliği Bölümü

BZ 313 Yazılım Mühendisliği

18. Tekrar Kullanım ve Tasarım Şablonları

Ders Sonu