**Satisfying Requirements**

Some software design decisions will involve tradeoffs in different quality attributes, such as performance, convenience, and security.

**Functional Requirements**

For software, there are functional requirements that describe what the system or application is expected to do.

**Non-Functional Requirements**

There are also non-functional requirements that specify how well the system or application does what it does. Such requirements may describe how well the software runs in particular situations.

**Collaborators**

Collaborators are other classes that the class interacts with to fulfill its responsibilities.

CRC:

Components

Responsibilities

Connections

**Components**

Components turn into collections of functions, classes, or other components. These pieces then represent a much simpler problem that the developers can individually implement.

You now have learned to take some time to think about the problem and outline the conceptual and technical design before actually implementing the solution.

Design artifacts, like conceptual mockups, help to clarify design decisions with clients and users.

Technical diagrams help to coordinate development work.

**Satisfying Requirements**

Some software design decisions will involve tradeoffs in different quality attributes, such as performance, convenience, and security.

**Abstract Data Type**

An abstract data type is a data type that is defined by the programmer and not built into the language. An abstract data type is essentially a grouping of related information that is denoted with a type. It was a way to organize data in a meaningful way.

**The goal of object-oriented design is to:**

Make an abstract data type easier to write

Structure a system around abstract data types called classes

Introduce the ability for an abstract data type to extend another

**Abstraction**

Abstraction is the idea of simplifying a concept in the problem domain to its essentials within some context. Abstraction allows you to better understand a concept by breaking it down into a simplified description that ignores unimportant details.

**Rule Of Least Astonishment**

The abstraction captures the essential attributes and behavior for a concept with no surprises and no definitions that fall beyond its scope.

You don’t want to surprise anyone trying to understand your abstraction with irrelevant characteristics.

**Encapsulation**

Encapsulation forms a self-contained object by bundling the data and functions it requires to work, exposes an interface whereby other objects can access and use it, and restricts access to certain inside details.

**Decomposition**

Decomposition is taking a **whole** thing and dividing it up into different parts. Or, on the flip side, taking a bunch of separate parts with different functionalities and combining them together to form a whole. Decomposition allows you to further break down problems into pieces that are easier to understand and solve.

Ayrıştırma, bir bütünün alınması ve farklı parçalara bölünmesidir. Ya da diğer taraftan, farklı işlevlere sahip bir grup ayrı parçayı alıp bir bütün oluşturmak için bir araya getirmek. Ayrıştırma, sorunları anlaması ve çözmesi daha kolay parçalara ayırmanıza olanak tanır.

Main reasons:

**Fixed or Dynamic**

**Lifetimes**

**Sharing**

**Generalization:**

In generalization we take repeated, common, or shared characteristics between two or more classes and factor them out into another class.

It makes code more reusable because different classes or methods can share the same blocks of code.

System become easier to maintain because we do not have repetitious code.

Generalization will help you build software that is easier to expand, easier to apply changes to and easier to maintain.

Genellemede, iki veya daha fazla sınıf arasında tekrarlanan, ortak veya paylaşılan özellikleri alır ve bunları başka bir sınıfa ayırırız.

Farklı sınıflar veya yöntemler aynı kod bloklarını paylaşabileceğinden, kodu yeniden kullanılabilir hale getirir.

Tekrar eden kodlarımız olmadığı için sistemin bakımı daha kolay hale gelir.

Genelleme, genişletilmesi daha kolay, değişiklikleri uygulaması daha kolay ve bakımı daha kolay yazılımlar oluşturmanıza yardımcı olacaktır.

UML DIAGRAMS

- ClassName

- Properties

- Operations

**Properties,** which are equivalent to Java’s member variables, are mainly composed of the variable type.

Variable types, much like in Java, can be classes or primitive types.

**Operations,**  which are equivalent to Java’s methods, are mainly composed of the operation name, parameter list and return type.

**Getter Methods**

Getter methods are methods that retrieve data, and their names typically begin with get and end with the name of the attribute whose value you will be returning.

Getters often retrieve a private piece of data.

**Setter Methods**

Setter methods change data, and their names typically begin with set and end with the name of the variable you wish to set.

Setters are used to set a private attribute in a safe way.

Encapsulation **involves** three ideas:

1. Bundle data and functions that manipulate the data, into a if-contained object.
2. Expose certain data and functions of that object, which can be accessed from other objects.
3. Restrict access to certain data and functions to only within that object.

**There are three types of relationships found in decomposition:**

Association

Aggregation

Composition

**Association**

Association is “some” relationship. This means that there is a loose relationship between two objects. These objects may interact with each other for some time.

public class Student{

public void play(Sport sport){

...

}

}

**Aggregation**

Aggregation is a “has-a” relationship where a whole has parts that belong to it. There may be sharing of parts among the wholes in this relationship.

The “has-a” relationship from a whole to the parts is considered weak. What this means is although parts can belong to the wholes, they can also exist independently.

A course has students, students have courses, and so on.

Toplama, bir bütünün kendisine ait parçalara sahip olduğu bir “vardır” ilişkisidir. Bu ilişkide bütünler arasında parçaların paylaşımı olabilir.

Bir bütünden parçalara “vardır” ilişkisi zayıf olarak kabul edilir. Bunun anlamı, parçalar bütüne ait olabilse de bağımsız olarak da var olabilirler.

Bir dersin öğrencileri vardır, öğrencilerin dersleri vardır vb.

Example:

Bookshelf-Book

Great job. A bookshelf can hold a lot of books but can also have none and it's still a bookshelf!

Here's of an example of some aggregation code. The Pet Store keeps track of many pet object but is not composed of them.

public class PetStore {

private ArrayList<Pet> pets;

public PetStore(){

pets = new ArrayList<Pet>();

}

public void add(Pet pet){

.....

}

}

Another example:

public class Airliner{

private ArrayList<CrewMember> crew;

public Airliner(){

crew = new ArrayList<CrewMember>();

}

public void add(CrewMember crewMember){...}

}

**Composition**

Composition is an exclusive containment of parts, otherwise known as a **strong** has-a relationship. What this means is that the whole cannot exist without its parts. If it loses any of its parts are destroyed too. Usually, you can only access the parts through their whole. Contained parts are exclusive to the whole.

Example composition: Human - Brain

A human must have its brain to be a human.

public class Human{

private Brain brain;

public Human(){

brain = new Brain();

}

}

The human and brain are tightly dependent with one not being able to exist without the other.

Another example:

Here's an example of some code implementing a composition relationship. Since the Salary is an attribute of the employee, each employee must have a salary when they are instantiated.

public class Employee{

private Salary salary;

public Employee(Salary employeeSalary){

this.salary = employeeSalary;

}

}

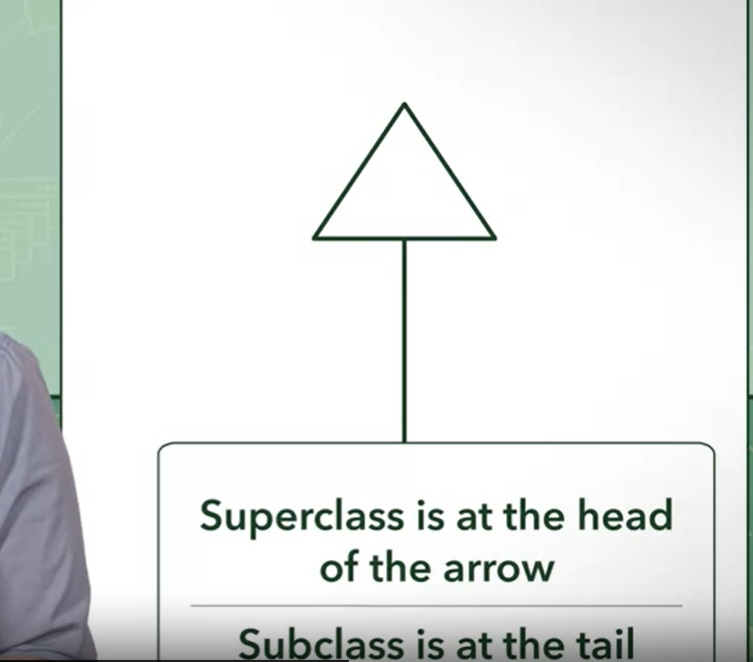
Decomposition is simply about whole objects containing part objects. Depending on your design you can relate wholes to parts in different increasingly tighter ways.

You can use association, a very loose interaction between two completely independent objects.

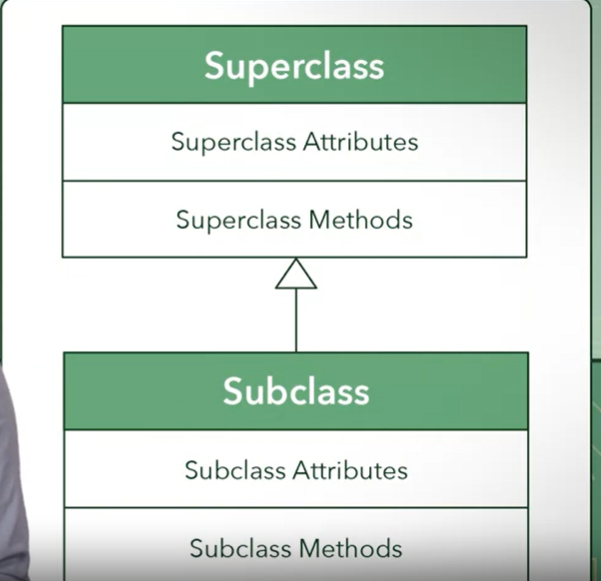
In an aggregation, one whole has a part, but both can live independently.

And finally, in composition, the whole cannot exist without its parts and vice versa.

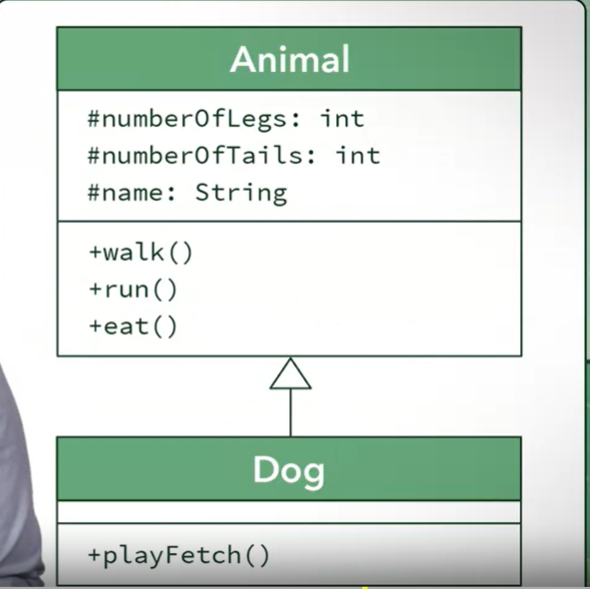
All three relationships are useful and versatile for your software designs.



This indicates, that two classes are connected by inheritance. The superclass is at the head of the arrow, and the subclass is at the tail. The standard way to draw inheritance into your UML diagrams is to have the arrow pointing upward. This means that the Superclasses are always toward the top . and Subclasses are always toward the bottom.



The superclasses are the generalized classes and the subclasses are the specialized classes.



A UMLclass diagram describes a dog class as a subclass, and Animal class as the superclass.

This means that the dog class will inherit from the Animal class.

The hash symbol is used to communicate that the animals attributes are protected.

In Java a protected attribute or method can only be accessed by:

The encapsulating class itself

All subclasses

All classes within the same package

public abstract class Animal{

protected int numberOfLegs;

public Animal(int legs){

this.numberOfLegs = legs;

}

}

Explicit constructors are used so that we can assign values to attributes during instantiation.

public abstract class Animal{

protected int numberOfLegs;

public Animal(int legs) {

this.numberOfLegs = legs;

}

}

public class Dog extends Animal{

public Dog(int legs){

super(legs);

}

}

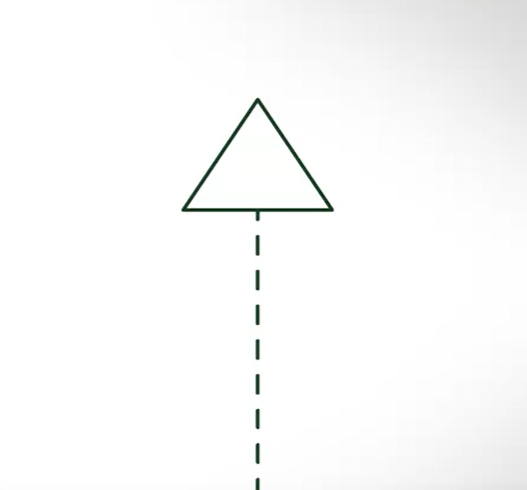
A subclass's constructor must call its superclass's constructor if the superclass has an explicit constructor

Class inheritance with the extends keyword

A JAVA interface also denotes a type. Unlike a class, however, an interface only declares method signatures, and no constructors, attributes, or method bodies.

Interfaces are used to describe behaviors.

You must have all the method signatures explicitly declared and implemented in the class.



Interface in UML diagrams.

The class touches the tail end of the arrow and the interface touches the head of the arrow.

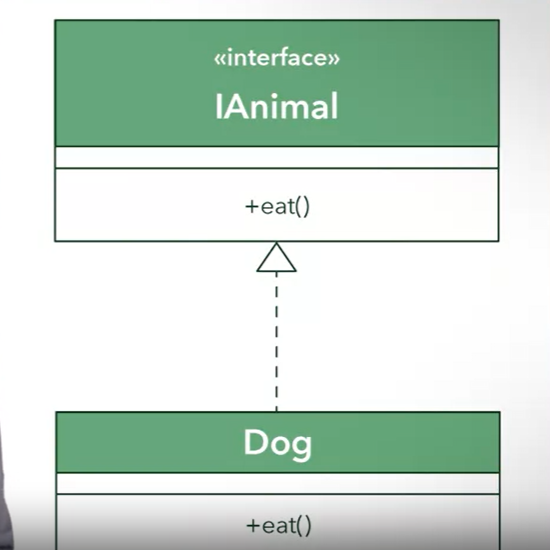
This means that the interface is always toward the top, and the classes that implement them are always toward the bottom.

**What are the three advantages that an interface will give us?**

*Interfaces can be implemented by multiple classes. Interfaces can be used by many classes just like subclass inheritance.*

*Classes can decide how to perform the behaviors described in an interface. Correct! Interfaces only specify the required methods, not how to implement them.*

*A class can implement multiple interfaces. Absolutely! This is one of the advantages that interfaces have that subclass inheritance does not.*



Like abstract classes, which are classes that cannot be instantiated,

Polymorphism is when two classes have the same description of a behavior, but the implementations of the behavior may be different.

**Select two of following statements about interfaces of which are true:**

*Interfaces can extend other interfaces. Correct*

*Yes they can! This allows you to compose different interfaces with reusable pieces.*

*Interfaces are not classes, but like abstract classes, are a means through which polymorphism can be achieved. Correct*

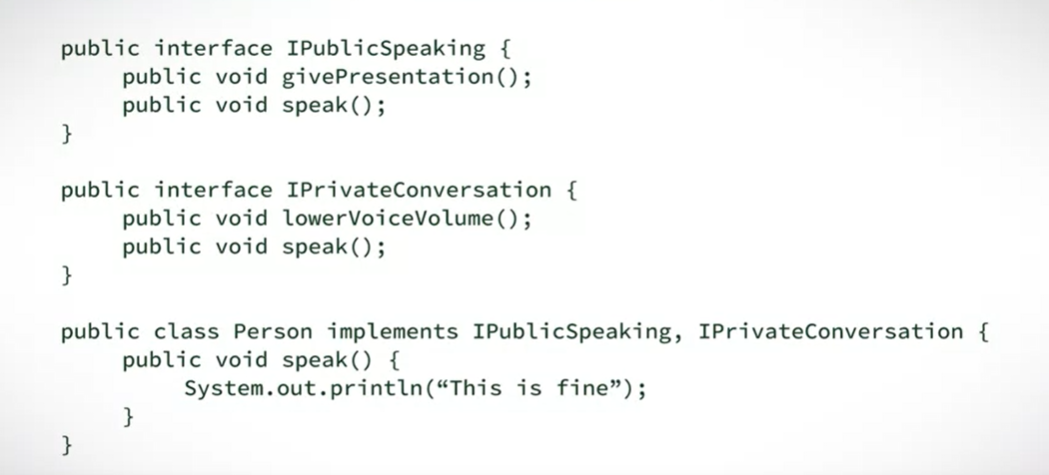
*Right! Interfaces have different properties that you can use to their advantage*.

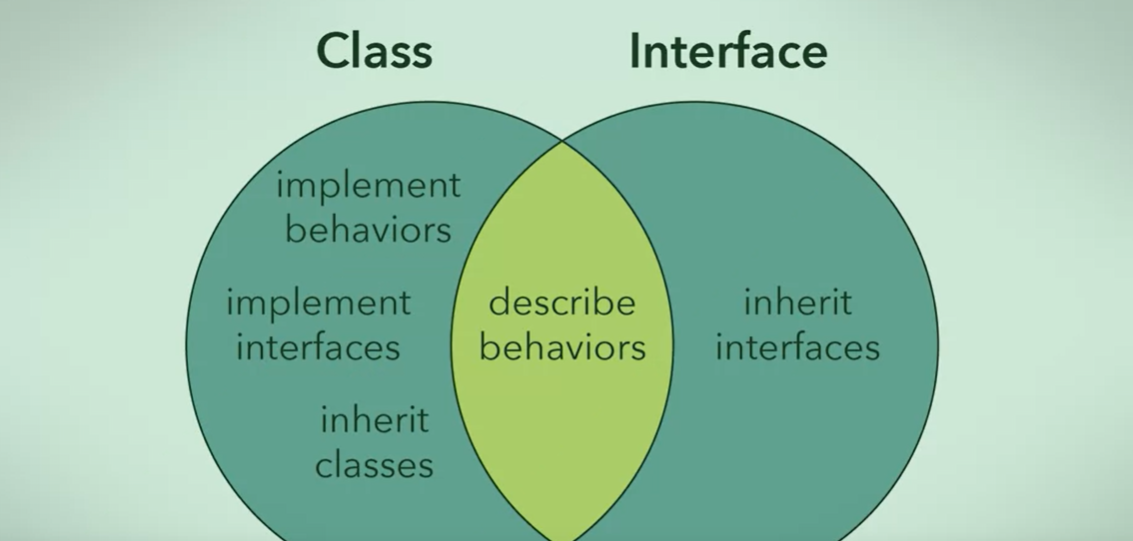
**Multiple Inheritance:**

This is when a subclass has two or more super classes. While this is possible to do with other object oriented languages, like C++. JAVA doesn’t support Multiple Inheritance. Since JAVA can not tell which one you would be referencing, it does not allow for multiple inheritance so that data ambiguity is not an issue. Interfaces do not run into this issue.

In Java, a class can implement as many interfaces as we want. This is because of the nature of interfaces. Since they are only contracts and do not enforce a specific way to complete these contracts, overlapping method signatures are not a problem.

A single implementation for multiple interfaces with overlapping contracts is acceptable.





Interface A should only inherit from interface B if the behaviors in interface A can fully be used as a substitution for interface B.

**WEEK 3**

**Separation of Concerns**

**Concern**

A concern is a very general notion, basically it is anything that matters in providing a solution to a problem. ( Endişe çok genel bir kavramdır, temelde bir soruna çözüm sağlamada önemli olan her şeydir. )

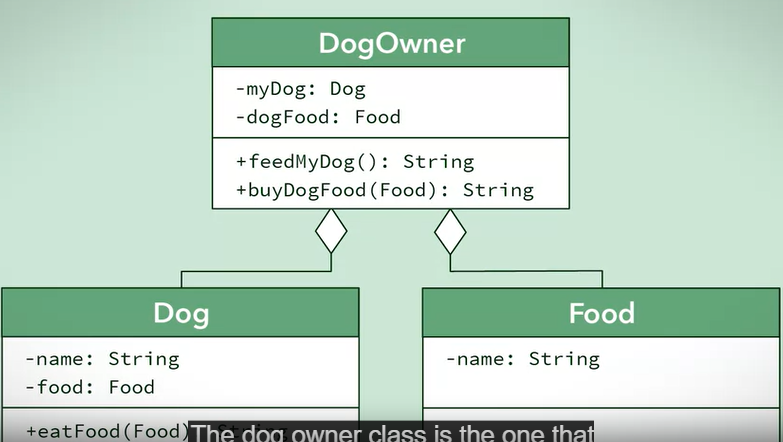
**Concerns**

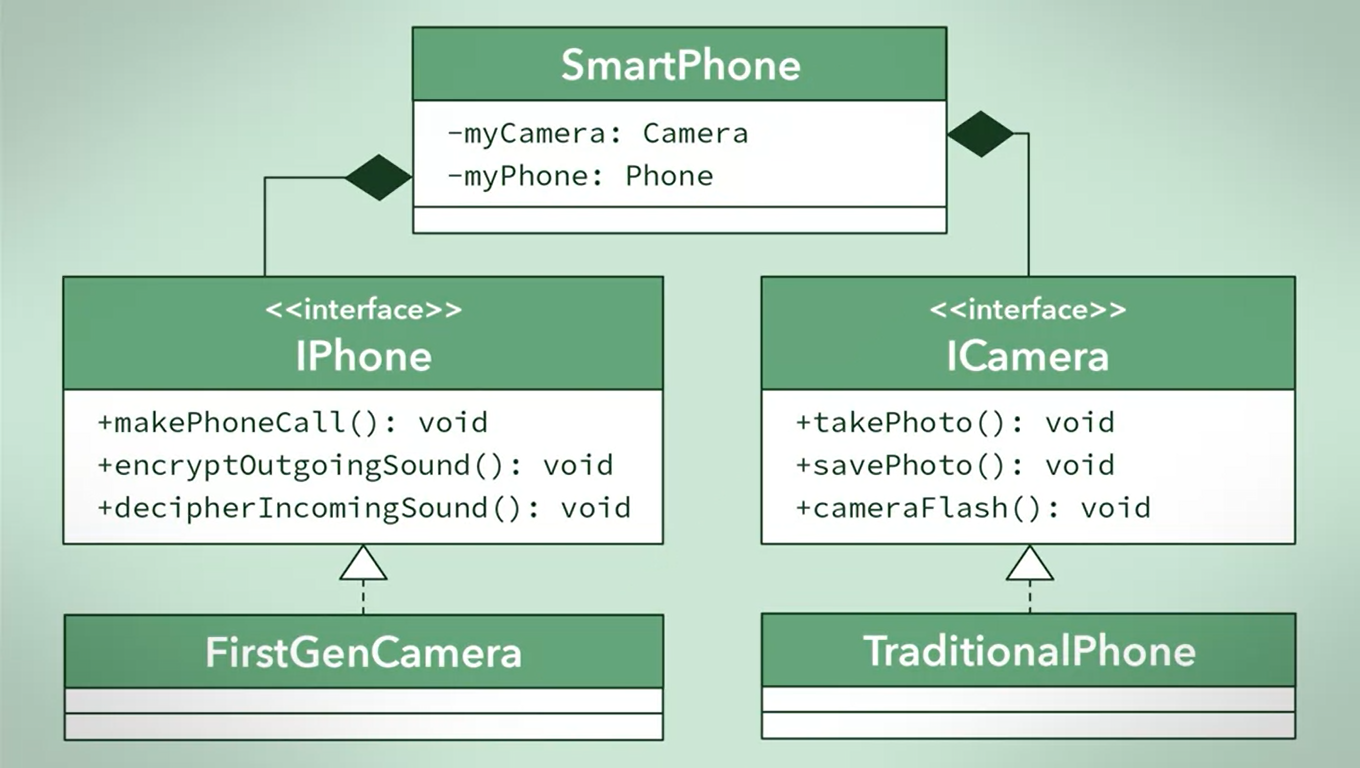
How these abstractions are implemented in the software can lead to more concerns. Some of these concerns may involve:( Bu soyutlamaların yazılımda nasıl uygulandığı daha fazla endişeye yol açabilir. Bu endişelerden bazıları şunları içerebilir: )

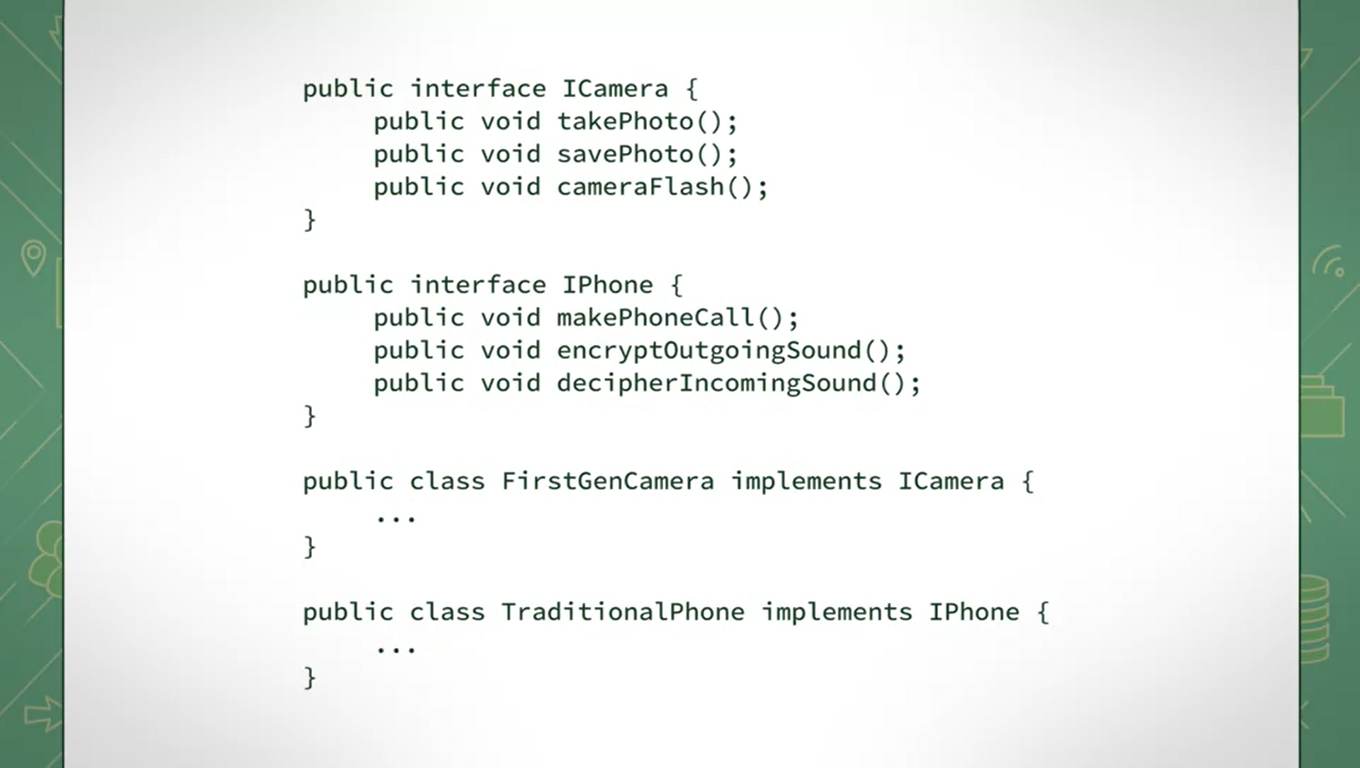
What information the implementation represents ( Uygulamanın temsil ettiği bilgiler )

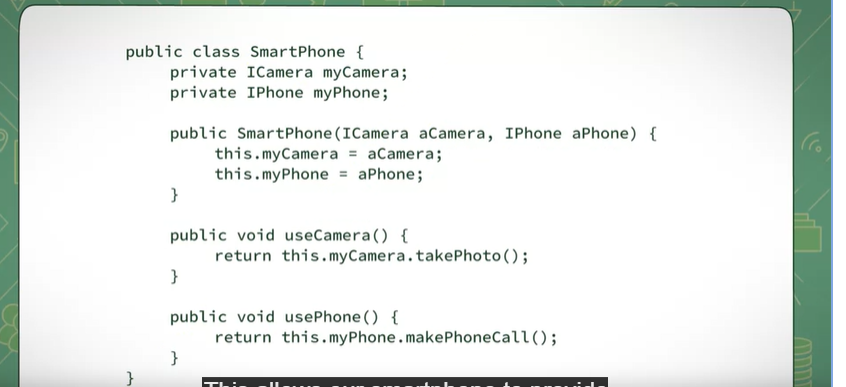
What it manipulates (Neyi manipüle eder)

What gets presented at the end (Sonunda sunulan şey)









Cohesion of the system is increased by encapsulating related functions into separate, and distinct classes. (Sistemin uyumu, ilgili işlevleri ayrı ve farklı sınıflara dahil ederek artırılır.)

The increase in modularity helps with creating a flexible system where additional classes and functionality can be added with ease or reused in other systems. ( Modülerlikteki artış, ek sınıfların ve işlevlerin kolaylıkla eklenebildiği veya diğer sistemlerde yeniden kullanılabildiği esnek bir sistem oluşturmaya yardımcı olur. )

The system becomes easier to maintain because classes are more specialized and contain less code.( Sınıflar daha özel ve daha az kod içerdiğinden sistemin bakımı daha kolay hale gelir. )

**Information Hiding**

Information hiding allows modules of our system to give others the minimum amount of information needed to use them correctly and "hide" everything else.( Bilgi gizleme, sistemimizin modüllerinin başkalarına onları doğru kullanmak için gereken minimum miktarda bilgi vermesini ve diğer her şeyi "gizlemesini" sağlar. )

**Information Hiding Through Encapsultion**

We use encapsulation to bundle attributes and behaviors into their appropriate class, and expose an interface to provide access.(Nitelikleri ve davranışları uygun sınıflarında bir araya getirmek için kapsülleme kullanıyoruz ve erişim sağlamak için bir arayüz ortaya koyuyoruz.)

**Access Modifiers**

Access modifiers change which classes are able to access attributes and behaviors. They also determine which attributes and behaviors a superclass will share with its subclasses. ( Erişim değiştiricileri, hangi sınıfların niteliklere ve davranışlara erişebileceğini değiştirir. Ayrıca bir üst sınıfın alt sınıflarıyla hangi nitelikleri ve davranışları paylaşacağını da belirlerler. )

**Public Methods**

Public methods are also accessible by any class in your system, but this access does not allow other classes to change the implementation of the behavior for the method. A publicly accessible method simply allows other classes to call the method and recieve any output from it.

( Genel yöntemlere sisteminizdeki herhangi bir sınıf tarafından da erişilebilir, ancak bu erişim, diğer sınıfların yöntemin davranışının uygulamasını değiştirmesine izin vermez. Genel olarak erişilebilir bir yöntem, diğer sınıfların yöntemi çağırmasına ve ondan herhangi bir çıktı almasına izin verir. )

**Protected Behaviors**

Protected behaviors and attributes are not accessible to every class in the system. They are only available to the encapsulating class itself, all subclasses and classes within the same package. ( Korunan davranışlar ve nitelikler, sistemdeki her sınıf tarafından erişilebilir değildir. Yalnızca kapsülleyen sınıfın kendisi, aynı paket içindeki tüm alt sınıflar ve sınıflar tarafından kullanılabilirler. )

**Package**

Packages are the means by which Java organizes related classes into a single namespace.

**Default Access Modifier**

The default access modifier will only allow access to attributes and methods to subclasses or classes that are part of the same package or encapsulation. This access modifier is also called the **no modifier** access because you do not need to explicitly declare it. ( Varsayılan erişim değiştiricisi, yalnızca aynı paketin veya kapsüllemenin parçası olan alt sınıflara veya sınıflara özniteliklere ve yöntemlere erişime izin verir. Bu erişim değiştiriciye, açık bir şekilde bildirmeniz gerekmediğinden değiştiricisiz erişim de denir. )

**Private Access Modifier**

Private attributes and methods are not accessible by any class other than by the encapsulating class itself. This means these attributes cannot be accessed directly and these methods cannot be invoked by any other classes.

( Özel niteliklere ve yöntemlere, kapsülleyen sınıfın kendisi dışında herhangi bir sınıf erişemez. Bu, bu özniteliklere doğrudan erişilemeyeceği ve bu yöntemlerin başka herhangi bir sınıf tarafından çağrılamayacağı anlamına gelir. )

**Conceptual Integrity**

"It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas."

( "Bir sisteme sahip olmak, bazı anormal özellikleri ve iyileştirmeleri atlamak, ancak bir dizi tasarım fikrini yansıtmak, birçok iyi ancak bağımsız ve koordine olmayan fikir içeren bir sisteme sahip olmaktan daha iyidir." )

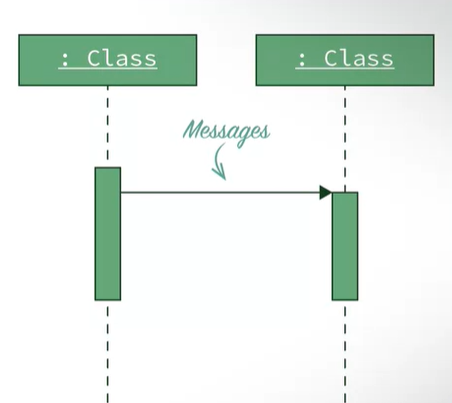
**Liskov Substitution Principle**

This principle states that a subclass can replace a superclass, if and only if, the subclass does not change the functionality of the superclass. ( Bu prinsip bildirir ki, alt sinif supersinifi əvəz edə bilər, o halda ki, alt sinif supersinfin funksionallığını dəyişməz. )

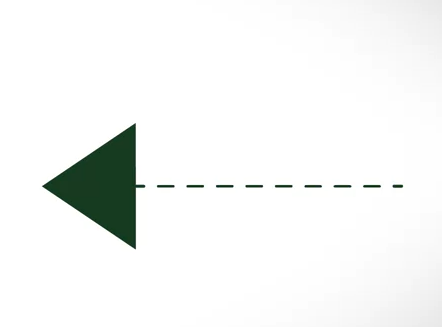
**Sequence Diagram**

A Sequence Diagram describes how objects in your system interact to complete a specific task.

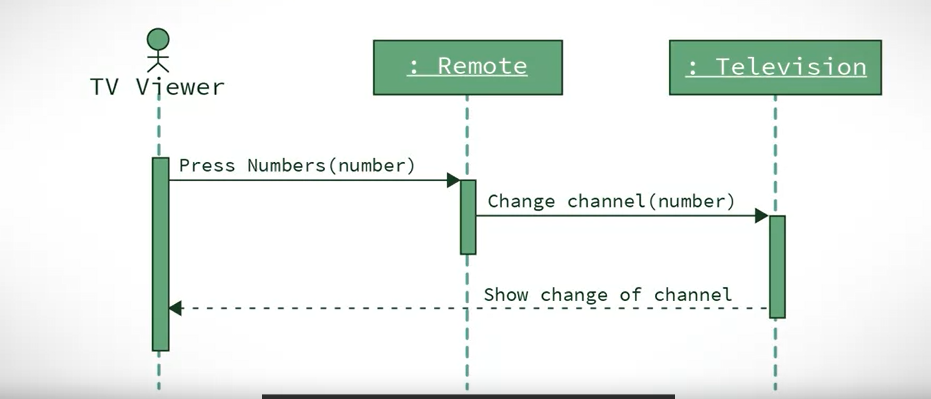
( Sıra Diyagramı, sisteminizdeki nesnelerin belirli bir görevi tamamlamak için nasıl etkileşimde bulunduğunu açıklar. )

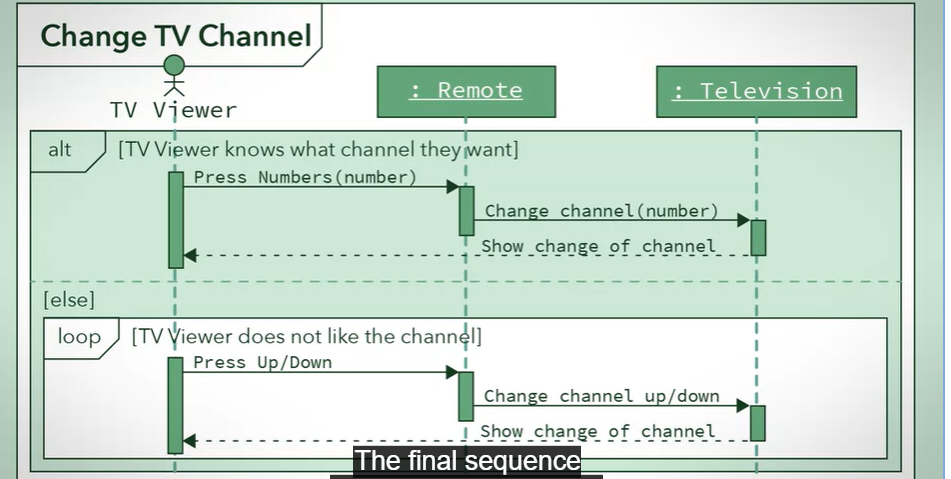


Solid draw



Sequence diagram for change TV channel:





Sequence Diagrams are used to show your team how objects in your program interact with each other to complete tasks. ( Ardıcıllıq Diaqramları komandanıza tapşırıqları yerinə yetirmək üçün proqramınızdakı obyektlərin bir-biri ilə necə qarşılıqlı əlaqədə olduğunu göstərmək üçün istifadə olunur. )

Simply put, think of sequence diagram like a map of conversations between different people, where this map follows all the messages sent from person to person.

( Basitçe söylemek gerekirse, dizi diyagramını farklı insanlar arasındaki konuşmaların bir haritası gibi düşünün, bu harita kişiden kişiye gönderilen tüm mesajları takip eder. )

**UML State Diagram**

A State diagram is a technique that you can use to describe how your system behaves and responds.

( Durum diyagramı, sisteminizin nasıl davrandığını ve yanıt verdiğini açıklamak için kullanabileceğiniz bir tekniktir. )

**State Diagrams**

State diagrams can describe a single object and illustrate how that object behaves in response to a series of events in your system.

( Durum diyagramları tek bir nesneyi tanımlayabilir ve bu nesnenin sisteminizdeki bir dizi olaya yanıt olarak nasıl davrandığını gösterebilir. )

**State ( Belirtmek, bildirmek )**

A state is the way an object exists at a particular point in time. The state of an oject is determined by the values of its attributes.

( Durum, bir nesnenin belirli bir zamanda var olma şeklidir. Bir nesnenin durumu, niteliklerinin değerleri ile belirlenir. )

First, I'll indicate the start of this diagram with a filled circle, Every state diagram has a filled circle to indicate which is the starting state, the vending machine starts in a state named idle.

( İlk olarak, bu diyagramın başlangıcını dolu bir daire ile göstereceğim, Her durum diyagramında, başlangıç durumunun hangisi olduğunu belirtmek için dolu bir daire vardır, otomat boşta adlı bir durumda başlar. )



States in more detail, each state has three important sections,

a state name,

state variables,

and activities.

A state name is as it sounds, the name of the state, these names should be meaningful for the states of your object.

(Bir durum adı göründüğü gibi, statein adıdır, bu adlar nesnenizin durumları için anlamlı olmalıdır.)

For example, a car in reverse would have a state named reverse.

State variables are data relevant to the state of the object.

( Durum değişkenleri, nesnenin durumuyla ilgili verilerdir. )

The course would be in state full if this variable was at the course capacity.

( Bu değişken kurs kapasitesinde olsaydı, kurs dolu durumda olurdu. )

Activities are actions that are performed when in a certain state, and they're displayed at the bottom. There are three types of activities for each state, entry, exit, and do.

Entry activities are actions that occur when the state is just entered from another state.

( Etkinlikler, belirli bir durumdayken gerçekleştirilen eylemlerdir ve altta görüntülenirler. Her durum için giriş, çıkış ve yap olmak üzere üç tür etkinlik vardır.

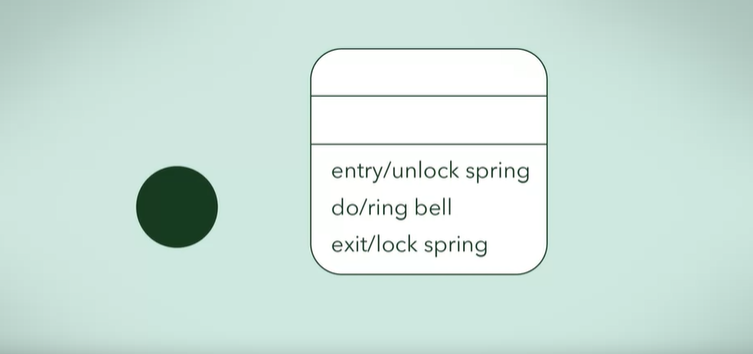
Giriş faaliyetleri, duruma başka bir durumdan yeni girildiğinde gerçekleşen eylemlerdir. )

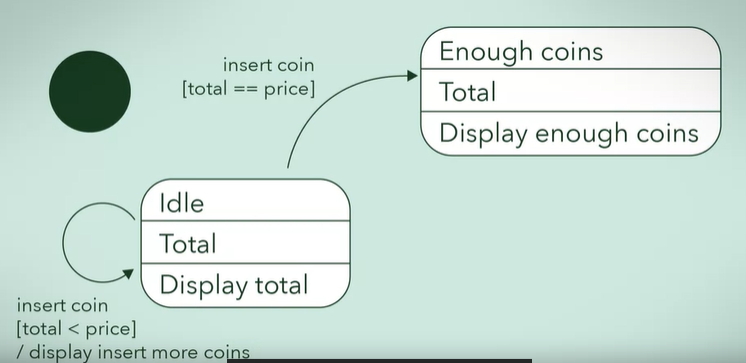
Exit activities are actions that occur when the state is exited and moves on to another state. And do activities are actions that occur once, or multiple times while the object is in a certain state.

( Çıkış faaliyetleri, durumdan çıkıldığında ve başka bir duruma geçildiğinde gerçekleşen eylemlerdir. Do etkinlikleri, nesne belirli bir durumdayken bir veya birden çok kez gerçekleşen eylemlerdir. )

To explain these activities, let's use a traditional alarm clock that uses a bell as an alarm.

( Bu etkinlikleri açıklamak için, alarm olarak zili kullanan geleneksel bir çalar saat kullanalım. )





Termination

One element of state diagrams that's not shown in the one we just created is termination. Termination represents an object being destroyed, or the process being completed, and is drawn as a circle with a filled circle inside.

( Sonlandırma

Az önce oluşturduğumuz şemada gösterilmeyen durum diyagramlarından biri sonlandırmadır. Sonlandırma, yok edilen bir nesneyi veya tamamlanan süreci temsil eder ve içi dolu bir daire olan bir daire olarak çizilir.

)

State diagrams are useful for describing the behavior of a system or of a single object.

( Durum diyagramları, bir sistemin veya tek bir nesnenin davranışını tanımlamak için kullanışlıdır. )

For example, it can help you determine the different events that might occur during an object's lifetime. Like different user inputs, and how that object should behave when these events occur, like checking conditions and performing actions.

(Örneğin, bir nesnenin ömrü boyunca meydana gelebilecek farklı olayları belirlemenize yardımcı olabilir. Farklı kullanıcı girdileri gibi ve koşulları kontrol etme ve eylemleri gerçekleştirme gibi bu olaylar meydana geldiğinde bu nesnenin nasıl davranması gerektiği gibi.)

State diagrams can also help you to find issues in your system. Like discovering a condition that you did not plan for or help you to create tests. Knowing the different states of a system can help to make sure that your tests are complete and correct. State diagrams are a very powerful design technique that you can use to model your software's behavior. Using them, along with the other UML diagrams you've learned, will help you to create well designed software.

(Durum diyagramları, sisteminizdeki sorunları bulmanıza da yardımcı olabilir. Planlamadığınız bir durumu keşfetmek veya testler oluşturmanıza yardımcı olmak gibi. Bir sistemin farklı durumlarını bilmek, testlerinizin eksiksiz ve doğru olduğundan emin olmanıza yardımcı olabilir. Durum diyagramları, yazılımınızın davranışını modellemek için kullanabileceğiniz çok güçlü bir tasarım tekniğidir. Öğrendiğiniz diğer UML diyagramlarıyla birlikte bunları kullanmak, iyi tasarlanmış bir yazılım oluşturmanıza yardımcı olacaktır.)

A State diagram shows you how a component, system, or object changes its state when events occur.

( Durum diyagramı, olaylar meydana geldiğinde bir bileşenin, sistemin veya nesnenin durumunu nasıl değiştirdiğini gösterir. )

**Model Checking**

Model checking is a systematic check of your system's state model in all its possible states. Basically, you check all the various states of your software and find that there are any errors by simulating different events that would change the states and variables of your software. So you check if the state model of your software contains some flaw. These checks are done using model checking software.

( Model kontrolü, sisteminizin durum modelinin tüm olası durumlarında sistematik bir kontrolüdür. Temel olarak, yazılımınızın tüm çeşitli durumlarını kontrol eder ve yazılımınızın durumlarını ve değişkenlerini değiştirecek farklı olayları simüle ederek herhangi bir hata olup olmadığını bulursunuz. Böylece, yazılımınızın durum modelinin bazı kusurlar içerip içermediğini kontrol edersiniz. Bu kontroller, model kontrol yazılımı kullanılarak yapılır. )

**Deadlock**

A deadlock is a situation where your system cannot continue because two tasks are waiting for the same resource.

( Kilitlenme, iki görevin aynı kaynağı beklediği için sisteminizin devam edemediği bir durumdur. )

**State Model**

A state model is an abstract state machine that can be in one of various states. The model checker then checks that the state model conforms to certain behavioral properties.

( Durum modeli, çeşitli durumlardan birinde olabilen soyut bir durum makinesidir. Model denetleyicisi daha sonra durum modelinin belirli davranış özelliklerine uygun olup olmadığını kontrol eder. )

The three different phases to performing model checking:

The Modeling phase

The Running phase

The Analysis phase

During the modeling phase, you can perform some Saturday checks. These are quick checks that should be easy to do because they usually come from very clear and simple logic. It's like testing if a light bulb is working by flipping a switch on and off. It's clear to see when the light bulb is malfunctioning. It's beneficial to get rid of these simpler errors before using any model checkers so that you can focus on specifying the more complex properties to check. The second phase is the running phase, this is when you run the model checker to see how your model conforms to the desired properties that you've wrote in the modeling phase.

( Modelleme aşamasında bazı Cumartesi kontrolleri yapabilirsiniz. Bunlar, genellikle çok açık ve basit bir mantıktan geldikleri için, yapılması kolay olması gereken hızlı kontrollerdir. Bu, bir anahtarı açıp kapatarak bir ampulün çalışıp çalışmadığını test etmek gibidir. Ampulün ne zaman arızalı olduğunu görmek açıktır. Kontrol edilecek daha karmaşık özellikleri belirlemeye odaklanabilmeniz için herhangi bir model denetleyicisini kullanmadan önce bu daha basit hatalardan kurtulmak faydalıdır. İkinci aşama çalıştırma aşamasıdır, bu, modelinizin modelleme aşamasında yazdığınız istenen özelliklere nasıl uyduğunu görmek için model denetleyicisini çalıştırdığınız zamandır. )