Bu ders, yazılım mühendisliğinde problem çözmenin temelini oluşturur. Bir "problem", çözülmesi gereken ve belirli kısıtlamaları olan bir durum olarak tanımlanır. Yazılım mühendisliği bağlamında problem çözme, "belirli bir dizi görevi yerine getiren ve belirtilen tüm kısıtlamaları karşılayan özgün bir program yazmak" anlamına gelir.

Ders, aşağıdaki temel terimleri tanımlayarak devam eder:

* **Bilgisayar:** Bilgiyi veya veriyi işleyen elektronik bir cihaz.
* **Donanım ve Yazılım:** Bilgisayarın fiziksel parçaları ve bu parçalar üzerinde çalışan programlar topluluğu.
* **Programlama Dili:** Geliştiricilerin bir problemi bilgisayar kullanarak çözmek için kullandıkları bir araçtır.
* **Algoritma:** Bir problemin çözüm adımlarını sıralı ve sonlu bir şekilde ifade etmektir.
* **Akış Şeması (Flowchart) ve Sözde Kod (Pseudocode):** Bir algoritmanın sırasıyla geometrik şekillerle grafiksel gösterimi ve konuşma dili ile programlama dili arasında bir ifade biçimidir.

Programlama dilleri, donanım dillerine olan yakınlıklarına göre **Düşük Seviyeli** (makine ve assembly dilleri) ve **Yüksek Seviyeli** (insan odaklı diller) olarak ikiye ayrılır. Ders, programlama paradigmalarını **İmperatif** (bir problemin *nasıl* çözüleceğini belirten) ve **Deklaratif** (sadece problemin *ne olduğunu* bildiren) olarak iki ana sınıfa ayırır. Bu ders, imperatif bir yaklaşım olan **Yapısal Programlama**'ya odaklanır. Yapısal programlamanın temel yapı taşları şunlardır:

* **Sıralama (Sequence):** Komutların birbiri ardına sırayla çalıştırılması.
* **Seçim (Selection):** Bir koşulun sonucuna göre iki seçenekten birinin seçilmesi (if/then/else).
* **Tekrarlama (Repetition/Loop):** Bir işlemin bir koşula bağlı olarak tekrar edilmesi (while/for).

Son olarak, C/C++ dilinin yüksek performansı, popülerliği ve bilgisayar kaynakları üzerinde sağladığı kontrol nedeniyle tercih edildiği belirtilir. Yapısal programlama ise programlamaya iyi bir başlangıç noktası sunduğu ve Nesne Yönelimli Programlama gibi daha ileri konular için bir temel oluşturduğu için önemlidir.**More Detailed Summary (English)**

This lecture establishes the foundation of problem-solving in software engineering. A "problem" is defined as a situation that needs to be solved and has certain constraints. In the context of software engineering, problem-solving means "writing an original program that performs a particular set of tasks and meets all stated constraints".

The lecture proceeds by defining the following basic terms:

* **Computer:** An electronic device that manipulates information, or data.
* **Hardware & Software:** The physical parts of a computer and the collection of programs that run on those parts.
* **Programming Language:** A tool used by developers to solve a problem using a computer.
* **Algorithm:** Expressing the solution steps of a problem in an ordered and finite manner.
* **Flowchart & Pseudocode:** A graphical representation of an algorithm with geometric shapes and a form of expression between spoken language and programming language, respectively.

Programming languages are classified based on their proximity to hardware languages as **Low-Level** (machine and assembly languages) and **High-Level** (people-oriented languages). The lecture divides programming paradigms into two main classes: **Imperative** (specifying *how* to solve a problem) and **Declarative** (only stating *what* the problem is). This lecture focuses on **Structured Programming**, an imperative approach. The fundamental building blocks of structured programming are:

* **Sequence:** The execution of commands one after another in order.
* **Selection:** Choosing one of two options based on the result of a condition (if/then/else).
* **Repetition/Loop:** Repeating an operation based on a condition (while/for).

**Lecture 2**

Bu ders, **Yazılım Geliştirme Yaşam Döngüsü (SDLC)** üzerine odaklanmaktadır. İyi bir yazılımın verimlilik, bakım kolaylığı, sağlamlık ve güvenlik gibi niteliklere sahip olması gerektiği vurgulanır. SDLC, bir yazılımın geliştirilmesi için izlenen beş aşamalı döngüsel bir süreç olarak sunulur:

1. **Gereksinim Analizi (Requirement Analysis):** Çözülecek problemin belirsizlik olmadan anlaşılması ve yazılımdan beklenenlerin ayrıntılı bir listesinin oluşturulmasıdır.
2. **Modelleme ve Tasarım (Modelling & Design):** Problemin bilgisayar tarafından temsil edilebilecek soyut bir modelinin oluşturulmasıdır. Bu aşamada UML, akış şemaları veya sözde kod gibi araçlar kullanılır.
3. **Geliştirme (Development):** Tasarım aşamasında oluşturulan modelin bir programlama dili ile bilgisayara aktarılmasıdır.
4. **Test Etme (Testing):** Programın olası girdi değerleri için nasıl davrandığının incelenmesi ve hataların (sözdizimsel, mantıksal, çalışma zamanı) ayıklanmasıdır.
5. **Dağıtım ve Bakım (Deployment & Maintenance):** Projenin her aşamasında yapılan işleri açıklayan belgelerin hazırlanması, yazılımın yayınlanması ve sonrasında destek sağlanmasıdır.

Bu teorik bilgiler, bir **Otomatik Vezne Makinesi (ATM)** yazılımı geliştirme örneği üzerinden pratikleştirilir. ATM için gereksinim analizi ayrıntılı olarak ele alınır:

* **Temel İşlevler:** Kullanıcıların hesap bakiyelerini görüntülemesi, para çekmesi ve para yatırması gerekmektedir.
* **Kullanıcı Arayüzü:** Sistem bir ekran, tuş takımı, para dağıtıcı ve para yatırma yuvası içermelidir.
* **İş Akışları:** Kullanıcı doğrulama (hesap numarası ve PIN girişi), ana menü (bakiye sorgulama, para çekme, para yatırma, çıkış), para çekme menüsü (önceden tanımlanmış tutarlar) ve para yatırma süreci gibi adımlar detaylandırılır.
* **Modelleme:** Sistemin davranışını modellemek için UML diyagramları tanıtılır. **Kullanım Senaryosu (Use Case) Diyagramı**, kullanıcı ile sistem arasındaki etkileşim senaryolarını özetlerken , **Etkinlik (Activity) Diyagramı**, para çekme işlemi gibi bir iş akışının dinamik görünümünü modeller.

**More Detailed Summary (English)**

This lecture focuses on the **Software Development Life Cycle (SDLC)**. It emphasizes that good software should possess qualities such as efficiency, ease of maintenance, robustness, and security. The SDLC is presented as a five-stage cyclical process for developing software:

1. **Requirement Analysis:** Understanding the problem to be solved without any uncertainty and creating a detailed list of what is expected from the software.
2. **Modelling & Design:** Creating an abstract model of the problem that can be represented by a computer. Tools like UML, flowcharts, or pseudocode are used in this phase.
3. **Development:** Transferring the model created during the design phase to the computer using a programming language.
4. **Testing:** Examining how the program behaves for possible input values and debugging errors (syntax, logical, run-time).
5. **Deployment & Maintenance:** Preparing documents that explain the work done at every stage of the project, releasing the software, and providing subsequent support.

This theoretical information is put into practice through the example of developing an **Automated Teller Machine (ATM)** software. The requirement analysis for the ATM is covered in detail:

* **Core Functions:** Users must be able to view their account balances, withdraw money, and deposit funds.
* **User Interface:** The system must include a screen, a keypad, a cash dispenser, and a deposit slot.
* **Workflows:** Steps such as user authentication (account number and PIN entry), main menu (balance inquiry, withdrawal, deposit, exit), withdrawal menu (predefined amounts), and the deposit process are detailed.
* **Modeling:** UML diagrams are introduced to model the system's behavior. A **Use Case Diagram** summarizes the interaction scenarios between a user and the system , while an **Activity Diagram** models the dynamic view of a workflow, such as the withdrawal transaction.

**Lecture 3**

Bu ders, algoritma, sözde kod ve akış şeması kavramlarını derinlemesine inceler. **Algoritma**, bir problemin çözüm adımlarını düzenli ve sonlu bir şekilde ifade eden bir talimatlar kümesi olarak tanımlanır. İyi bir algoritmanın özellikleri şunlardır: Verimli, Sonlu, Kesin, Girdi/Çıktı tanımlı ve Performanslı. Algoritmalar, metin olarak, yarı kod yarı metin olan **Sözde Kod (Pseudocode)** ile veya geometrik şekillerle görselleştirilen **Akış Şemaları (Flowcharts)** ile ifade edilebilir.

Ders, algoritmaların yapı taşları olan temel terimleri ve sembolleri ayrıntılı olarak açıklar:

* **Tanımlayıcı (Identifier), Değişken (Variable), Sabit (Constant):** Program içinde veri depolamak ve adlandırmak için kullanılan yapılar ve bunlarla ilgili kurallar.
* **Atama İşlemi (Assignment Operation):** Bir değişkene değer atama eylemi.
* **Operatörler:**
  + **Aritmetik Operatörler:** Toplama, çıkarma, çarpma, bölme gibi matematiksel işlemler.
  + **Karşılaştırma Operatörleri:** Değerleri karşılaştırmak için kullanılır (==, !=, >, <).
  + **Mantıksal Operatörler:** Koşulları birleştirmek için kullanılır (VE-AND, VEYA-OR, DEĞİL-NOT).
* **Karar/Karşılaştırma Yapısı (Decision/Comparison Structure):** Bir koşula bağlı olarak program akışını değiştiren IF ifadeleri.
* **Döngü Yapısı (Loop Structure):** Belirli işlemleri belirli sayıda veya bir koşul sağlanana kadar tekrarlayan yapılar.

Son olarak, **Akış Şeması Sembolleri** tanıtılır. Bunlar arasında Başlat/Bitir (elips), Girdi (paralelkenar), İşlem (dikdörtgen), Karar (baklava dilimi) ve Döngü (altıgen) gibi temel şekiller bulunur. Ders, bu kavramları pekiştirmek için N'e kadar olan sayıların toplamını hesaplama ve matris/vektör toplama gibi örnekler sunar.

**More Detailed Summary (English)**

This lecture provides an in-depth look at the concepts of algorithms, pseudocode, and flowcharts. An **Algorithm** is defined as a set of instructions that expresses the solution steps of a problem in an orderly and finite manner. The characteristics of a good algorithm are: Efficient, Finite, Certain, having defined Input/Output, and being Performant. Algorithms can be expressed as text, with **Pseudocode** (half code, half text), or with **Flowcharts**, which are visualized using geometric shapes.

The lecture details the fundamental terms and symbols that are the building blocks of algorithms:

* **Identifier, Variable, Constant:** Structures used to store and name data within a program, along with their associated rules.
* **Assignment Operation:** The act of assigning a value to a variable.
* **Operators:**
  + **Arithmetic Operators:** Mathematical operations such as addition, subtraction, multiplication, and division.
  + **Comparison Operators:** Used to compare values (==, !=, >, <).
  + **Logical Operators:** Used to combine conditions (AND, OR, NOT).
* **Decision/Comparison Structure:** IF statements that change the program flow based on a condition.
* **Loop Structure:** Constructs that repeat certain operations a specific number of times or until a condition is met.

Finally, **Flowchart Symbols** are introduced. These include basic shapes like Start/End (ellipse), Input (parallelogram), Operation (rectangle), Decision (diamond), and Loop (hexagon). To reinforce these concepts, the lecture presents examples such as calculating the sum of numbers up to N and performing matrix/vector addition.

**Lecture 4**

Bu ders, önceki derslerde öğrenilen teorik bilgileri pekiştirmek amacıyla tamamen pratik örneklere ayrılmıştır. Algoritma ve akış şeması oluşturma becerilerini geliştirmek için basitten karmaşığa doğru sıralanmış çeşitli problemler sunulur ve her biri için adım adım çözümler verilir.

İşlenen örnekler şunlardır:

* **Örnek #1, #2, #3:** "Merhaba Dünya" yazdırma, önceden tanımlanmış iki sayıyı toplama ve klavyeden girilen iki sayıyı toplama gibi temel programlama görevlerini içerir.
* **Örnek #4:** Klavyeden girilen x ve y değerlerine göre x2+y matematiksel ifadesini hesaplayan bir programın algoritması ve akış şemasıdır.
* **Örnek #5:** Bir döngü ve koşul yapısı gerektiren bir problemdir. Klavyeden girilen sayıların toplamı 100'ü geçene kadar sayı girişine devam edilir.
* **Örnek #6:** Bir sayının faktöriyelini hesaplamak için dört farklı çözüm sunar. Bu çözümler, farklı döngü mantıklarını göstermektedir: artan sayaçlı bir döngü, azalan sayaçlı bir döngü ve iki farklı for döngüsü uygulaması. Bu, tek bir probleme birden fazla algoritmik yaklaşımla yaklaşılabileceğini gösterir.
* **Örnek #7:** Daha ileri düzey bir problem olan bir sayının karekökünü yinelemeli bir yöntemle tahmin etmektir. Bu örnek, problemin arkasındaki matematiksel formülü (xi+1​=xi​+b ve b≅(a−x2)/2x) açıklar, sözde kodu sunar, örnek bir girdiyle (a=31.8, x=5.0, ε=0.005) adımların nasıl işlediğini bir tabloyla gösterir ve son olarak çözümün akış şemasını çizer.

#### More Detailed Summary (English)

This lecture is entirely dedicated to practical examples to reinforce the theoretical knowledge learned in previous lectures. It presents various problems, ordered from simple to complex, to develop skills in creating algorithms and flowcharts, providing step-by-step solutions for each.

The examples covered are:

* **Examples #1, #2, #3:** These include basic programming tasks such as printing "Hello World", adding two predefined numbers, and adding two numbers entered from the keyboard.
* **Example #4:** The algorithm and flowchart for a program that calculates the mathematical expression x2+y based on x and y values entered from the keyboard.
* **Example #5:** A problem requiring a loop and a conditional structure. Number input continues until the sum of the numbers entered from the keyboard exceeds 100.
* **Example #6:** This presents four different solutions for calculating the factorial of a number. These solutions demonstrate different loop logics: a loop with an increasing counter, a loop with a decreasing counter, and two different for loop implementations. This shows that a single problem can be approached with multiple algorithmic strategies.
* **Example #7:** A more advanced problem of estimating the square root of a number using an iterative method. This example explains the mathematical formula behind the problem (xi+1​=xi​+b and b≅(a−x2)/2x), presents the pseudocode, shows how the steps work with sample input (a=31.8, x=5.0, ε=0.005) in a table, and finally, draws the flowchart for the solution.

**Lecture 5**

Bu ders, **C++ programlama diline** kapsamlı bir giriş yapar. C++'ın, Denis Ritchie tarafından geliştirilen C dilinin bir devamı olduğu ve 1979'da Bjarne Stroustrup tarafından geliştirildiği belirtilir. Ders, bir C++ programının geliştirme ve derleme adımlarını ayrıntılı olarak açıklar: Düzenleme (kaynak kod yazımı), Ön İşleme, Derleme (kaynak kodun nesne koduna çevrilmesi), Bağlama (kütüphanelerle birleştirilip çalıştırılabilir dosya oluşturma) ve Yükleme/Çalıştırma.

Bir C++ programının genel yapısı şu bileşenlerden oluşur:

* **Kütüphane Dosyaları:** #include <iostream> gibi ön işlemci direktifleriyle eklenirler.
* **İsim Alanı (Namespace):** using namespace std; komutu, standart kütüphanedeki tanımlayıcıların (örn. cout, cin) doğrudan kullanılmasını sağlar. İsim alanları, büyük programlarda ad çakışmalarını önlemek için kullanılır.
* **main() Fonksiyonu:** Programın başlangıç noktasıdır.
* **Değişkenler ve Sabitler:** Programda veri tutmak için kullanılırlar. Ders, int, char, float gibi temel veri türlerini, değişken adlandırma kurallarını ve const ile sabit tanımlamayı açıklar.
* **Operatörler:** Aritmetik (+, -, \*, /, %), ilişkisel (==, !=, >), mantıksal (&&, ||, !) ve özel C++ operatörleri (+=, ++) gibi operatörler ve işlem önceliği tablosu sunulur.
* **Kütüphane Fonksiyonları:** Farklı başlık dosyalarının (<iomanip>, <cmath>) nasıl dahil edileceği ve setw() (alan genişliği ayarlama) ile sqrt() (karekök hesaplama) gibi fonksiyonların nasıl kullanılacağı örneklerle gösterilir.

Ders, klavyeden veri almak için cin ve ekrana veri yazdırmak için cout komutlarının kullanımını gösteren pratik kod örnekleriyle doludur.

#### More Detailed Summary (English)

This lecture provides a comprehensive introduction to the **C++ programming language**. It is stated that C++ is a continuation of the C language, developed by Denis Ritchie, and was developed in 1979 by Bjarne Stroustrup. The lecture explains the development and build steps of a C++ program in detail: Editing (writing source code), Preprocessing, Compiling (translating source code to object code), Linking (combining with libraries to create an executable file), and Loading/Executing.

The general structure of a C++ program consists of the following components:

* **Library Files:** They are included with preprocessor directives like #include <iostream>.
* **Namespace:** The using namespace std; command allows for the direct use of identifiers (e.g., cout, cin) from the standard library. Namespaces are used to prevent name collisions in large programs.
* **The main() Function:** This is the starting point of the program.
* **Variables and Constants:** They are used to store data in the program. The lecture explains basic data types like int, char, float, variable naming rules, and defining constants with const.
* **Operators:** Operators such as arithmetic (+, -, \*, /, %), relational (==, !=, >), logical (&&, ||, !), and special C++ operators (+=, ++) are presented, along with a table of operator precedence.
* **Library Functions:** How to include different header files (<iomanip>, <cmath>) and use functions like setw() (to set field width) and sqrt() (to calculate square root) is demonstrated with examples.

The lecture is filled with practical code examples that show how to use the cin command to get data from the keyboard and the cout command to print data to the screen.

**Lecture 6**