**Lecture 2: Introduction to Problem Solving**

**English Summary**

This lecture introduces the fundamental concepts of problem-solving within the context of software engineering. A "problem" is defined as a situation that requires a solution and has certain constraints. For software engineers, problem-solving is the act of "writing an original program that performs a particular set of tasks and meets all stated constraints". The lecture then defines basic terminology such as **Computer** (an electronic device that manipulates data), **Hardware** (physical devices), and **Software** (a collection of programs).

Programming languages are categorized as **Low-level** (machine-oriented and assembly languages) and **High-level** (people-oriented and machine-independent). A **Compiler** is introduced as a language system that translates source code written in a high-level language into machine code. The core of problem-solving is the **Algorithm**, which is a finite and ordered expression of the solution steps for a problem. Algorithms can be represented visually using a **Flowchart** or textually using **Pseudocode**, which is a language between spoken and programming languages.

Finally, the lecture introduces programming paradigms, classifying them into two core types: **Imperative**, where the programmer specifies *how* to solve a problem, and **Declarative**, where the programmer only declares *what* problem to solve. The lecture focuses on **Structured Programming**, an imperative paradigm built on three fundamental control structures:

* **Sequence**: Operations are carried out one after the other.
* **Selection**: One of two options is selected depending on the result of a condition (e.g., if/then/else).
* **Repetition/Iteration**: A block of code is executed repeatedly based on a condition (e.g., while/for).

The lecture concludes by justifying the use of C/C++ for its high performance and powerful control over computer resources, and explains that starting with structured programming provides a good foundation for more advanced topics like Object-Oriented Programming (OOP).

**Türkçe Özet**

Bu ders, yazılım mühendisliği bağlamında problem çözmenin temel kavramlarını tanıtmaktadır. Bir "problem", çözüm gerektiren ve belirli kısıtlamaları olan bir durum olarak tanımlanır. Yazılım mühendisleri için problem çözme, "belirli bir görev dizisini yerine getiren ve belirtilen tüm kısıtlamaları karşılayan özgün bir program yazma" eylemidir. Ders daha sonra **Bilgisayar** (veriyi işleyen elektronik bir cihaz), **Donanım** (fiziksel cihazlar) ve **Yazılım** (programlar topluluğu) gibi temel terminolojiyi tanımlar.

Programlama dilleri **Düşük Seviyeli** (makine odaklı ve assembly dilleri) ve **Yüksek Seviyeli** (insan odaklı ve makineden bağımsız) olarak kategorize edilir. **Derleyici**, yüksek seviyeli bir dilde yazılmış kaynak kodunu makine koduna çeviren bir dil sistemi olarak tanıtılır. Problem çözmenin özü, bir problemin çözüm adımlarının sonlu ve sıralı bir ifadesi olan **Algoritma**'dır. Algoritmalar, görsel olarak bir **Akış Şeması** kullanılarak veya metinsel olarak, konuşma ve programlama dilleri arasında bir dil olan **Sözde Kod (Pseudocode)** kullanılarak temsil edilebilir.

Son olarak, ders programlama paradigmalarını tanıtır ve bunları iki temel türe ayırır: programcının bir problemi *nasıl* çözeceğini belirttiği **İmperatif** ve programcının sadece *hangi* problemi çözeceğini beyan ettiği **Deklaratif**. Ders, üç temel kontrol yapısı üzerine kurulu imperatif bir paradigma olan **Yapısal Programlama**'ya odaklanır:

* **Sıralama (Sequence)**: İşlemler birbiri ardına gerçekleştirilir.
* **Seçim (Selection)**: Bir koşulun sonucuna bağlı olarak iki seçenekten biri seçilir (örneğin, if/then/else).
* **Tekrarlama/Yineleme (Repetition/Iteration)**: Bir kod bloğu bir koşula bağlı olarak tekrar tekrar yürütülür (örneğin, while/for).

Ders, yüksek performansı ve bilgisayar kaynakları üzerindeki güçlü kontrolü nedeniyle C/C++ kullanımını gerekçelendirerek ve yapısal programlama ile başlamanın Nesne Yönelimli Programlama (OOP) gibi daha ileri konular için iyi bir temel sağladığını açıklayarak sona erer.

**Lecture 3: Software Development Life Cycle (SDLC)**

**English Summary**

This lecture focuses on the **Software Development Life Cycle (SDLC)**, a structured process for building software. It begins by highlighting the qualities of a good software product, which include efficiency, ease of maintenance, robustness, and security. The SDLC is presented as a cyclical process consisting of five distinct phases:

1. **Requirement Analysis**: This initial phase involves understanding the problem to be solved without ambiguity and creating a detailed requirement list that outlines everything expected from the software.
2. **Modelling & Design**: An abstract model of the solution is created, which can be represented by a computer. This design directly impacts the software's quality and is often documented using tools like UML, flowcharts, or pseudocode.
3. **Development**: The design model is translated into a computer program using a suitable programming language.
4. **Testing**: The program's behavior is examined with various inputs to identify and correct different types of errors, such as Syntax, Logical, and Run-time errors.
5. **Deployment & Maintenance**: This final phase includes preparing documentation for all stages of the project, releasing the software to users, and providing ongoing support and updates.

The lecture then applies this methodology to a practical example: developing software for an **Automated Teller Machine (ATM)**. The **Requirement Analysis** for the ATM is detailed, covering user authentication, transaction types (view balance, withdraw cash, deposit funds), and the user interface (screen, keypad, cash dispenser, deposit slot). For the **Modelling & Design** phase, UML diagrams are introduced as a tool for documenting system models. Specifically, a **Use Case Diagram** is used to summarize interactions between the user and the system, while an **Activity Diagram** models the dynamic workflow of a transaction, such as a withdrawal.

**Türkçe Özet**

Bu ders, yazılım oluşturmak için yapılandırılmış bir süreç olan **Yazılım Geliştirme Yaşam Döngüsü (SDLC)** üzerine odaklanmaktadır. İyi bir yazılım ürününün verimlilik, bakım kolaylığı, sağlamlık ve güvenlik gibi niteliklerini vurgulayarak başlar. SDLC, beş ayrı aşamadan oluşan döngüsel bir süreç olarak sunulmaktadır:

1. **Gereksinim Analizi**: Bu başlangıç aşaması, çözülecek problemin belirsizlik olmadan anlaşılmasını ve yazılımdan beklenen her şeyi özetleyen ayrıntılı bir gereksinim listesi oluşturulmasını içerir.
2. **Modelleme ve Tasarım**: Çözümün, bir bilgisayar tarafından temsil edilebilecek soyut bir modeli oluşturulur. Bu tasarım, yazılımın kalitesini doğrudan etkiler ve genellikle UML, akış şemaları veya sözde kod gibi araçlar kullanılarak belgelenir.
3. **Geliştirme**: Tasarım modeli, uygun bir programlama dili kullanılarak bir bilgisayar programına çevrilir.
4. **Test Etme**: Programın davranışı, Sözdizimsel, Mantıksal ve Çalışma Zamanı hataları gibi farklı hata türlerini belirlemek ve düzeltmek için çeşitli girdilerle incelenir.
5. **Dağıtım ve Bakım**: Bu son aşama, projenin tüm aşamaları için dokümantasyon hazırlanmasını, yazılımın kullanıcılara sunulmasını ve sürekli destek ve güncellemeler sağlanmasını içerir.

Ders daha sonra bu metodolojiyi pratik bir örneğe uygular: bir **Otomatik Vezne Makinesi (ATM)** için yazılım geliştirmek. ATM için **Gereksinim Analizi**, kullanıcı kimlik doğrulaması, işlem türleri (bakiye görüntüleme, para çekme, para yatırma) ve kullanıcı arayüzü (ekran, tuş takımı, para dağıtıcı, para yatırma yuvası) dahil olmak üzere ayrıntılı olarak ele alınır. **Modelleme ve Tasarım** aşaması için, sistem modellerini belgelemek için bir araç olarak UML diyagramları tanıtılır. Özellikle, kullanıcı ile sistem arasındaki etkileşimleri özetlemek için bir **Kullanım Senaryosu Diyagramı** kullanılırken, para çekme gibi bir işlemin dinamik iş akışını modellemek için bir **Etkinlik Diyagramı** kullanılır.

**Lecture 4: Algorithm, Pseudocode & Flowchart Details**

**English Summary**

This lecture delves deeper into the tools of algorithm design. An **Algorithm** is defined as the logical and symbolic description of the predicted operations for solving a problem, and it should be efficient, finite, and certain. The lecture introduces fundamental building blocks used in algorithms, such as **Identifiers** (names for variables, constants, etc.), **Variables** (symbolic names for memory areas where data is stored), and **Constants** (identifiers that keep their values throughout the program).

The lecture provides a detailed breakdown of different types of operations and structures:

* **Assignment Operation**: This is used to write data into a variable, often represented with an equals sign (=).
* **Operators**: These are symbols that indicate transactions on data. They are categorized as:
  + **Arithmetic Operators**: Used for mathematical calculations. | Operation | Mathematics | Computer | | :--- | :--- | :--- | | Addition | a+b | a+b | | Subtraction | a−b | a-b | | Multiplication | a⋅b | a\*b | | Division | a÷b | a/b | | Exponentiation | ab | a^b |
  + **Comparison Operators**: Used to check relationships between two data items (e.g., =, >, <).
  + **Logical Operators**: Used to combine conditions (e.g., AND, OR, NOT).
* **Decision/Comparison Structure**: The IF query statement is used to change the sequence of operations based on whether a condition is true or not.
* **Loop Structure**: This is used to repeat certain operations a specific number of times or while a condition holds true. A loop requires an initial value for its variable, a defined step for increment/decrement, and an end value.

**Türkçe Özet**

Bu ders, algoritma tasarım araçlarını daha derinlemesine inceler. **Algoritma**, bir problemi çözmek için öngörülen işlemlerin mantıksal ve sembolik tanımı olarak tanımlanır ve verimli, sonlu ve kesin olmalıdır. Ders, algoritmalarda kullanılan **Tanımlayıcılar** (değişkenler, sabitler vb. için isimler), **Değişkenler** (verilerin saklandığı bellek alanları için sembolik isimler) ve **Sabitler** (program boyunca değerlerini koruyan tanımlayıcılar) gibi temel yapı taşlarını tanıtır.

Ders, farklı işlem ve yapı türlerinin ayrıntılı bir dökümünü sunar:

* **Atama İşlemi**: Veriyi bir değişkene yazmak için kullanılır ve genellikle eşittir işareti (=) ile temsil edilir.
* **Operatörler**: Veri üzerindeki işlemleri belirten sembollerdir. Şu şekilde kategorize edilirler:
  + **Aritmetik Operatörler**: Matematiksel hesaplamalar için kullanılır. | İşlem | Matematik | Bilgisayar | | :--- | :--- | :--- | | Toplama | a+b | a+b | | Çıkarma | a−b | a-b | | Çarpma | a⋅b | a\*b | | Bölme | a÷b | a/b | | Üs Alma | ab | a^b |
  + **Karşılaştırma Operatörleri**: İki veri öğesi arasındaki ilişkileri kontrol etmek için kullanılır (örneğin, =, >, <).
  + **Mantıksal Operatörler**: Koşulları birleştirmek için kullanılır (örneğin, VE, VEYA, DEĞİL).
* **Karar/Karşılaştırma Yapısı**: IF sorgu ifadesi, bir koşulun doğru olup olmamasına bağlı olarak işlem sırasını değiştirmek için kullanılır.
* **Döngü Yapısı**: Belirli işlemleri belirli sayıda veya bir koşul geçerli olduğu sürece tekrarlamak için kullanılır. Bir döngü, değişkeni için bir başlangıç değeri, artırma/azaltma için tanımlanmış bir adım ve bir bitiş değeri gerektirir.

**Lecture 5: Algorithm, Pseudocode and Flowchart Examples**

**English Summary**

This lecture is dedicated to providing practical examples to reinforce the concepts of algorithm and flowchart design. The examples progress from simple to complex, demonstrating how to apply algorithmic thinking to various problems.

Key examples include:

* **Sum of Two Numbers**: A basic algorithm that takes two numbers from keyboard input, calculates their sum, and prints the result.
* **Summing Numbers Until a Condition is Met**: An algorithm that reads numbers from the keyboard and adds them to a sum variable until the sum is greater than 100. This example clearly demonstrates the use of a conditional loop.
* **Factorial Calculation**: This problem is solved using four different algorithms to illustrate that there can be multiple correct approaches. The solutions showcase loops with an incrementing counter, a decrementing counter, and two different for loop structures.
* **Estimating a Square Root**: This is a more advanced example that calculates the square root of a number based on an initial estimate and an error value. The algorithm uses an iterative method based on the formula xi+1​=xi​+b, where b≅(a−x2)/2x. The process is repeated until the absolute error value is less than or equal to a predefined threshold. A sample run is provided to trace the algorithm's execution:

**Türkçe Özet**

Bu ders, algoritma ve akış şeması tasarımı kavramlarını pekiştirmek için pratik örnekler sunmaya adanmıştır. Örnekler basitten karmaşığa doğru ilerleyerek, algoritmik düşüncenin çeşitli problemlere nasıl uygulanacağını göstermektedir.

Öne çıkan örnekler şunlardır:

* **İki Sayının Toplamı**: Klavyeden girilen iki sayıyı alan, toplamlarını hesaplayan ve sonucu yazdıran temel bir algoritma.
* **Bir Koşul Sağlanana Kadar Sayıları Toplama**: Klavyeden sayıları okuyan ve toplamları 100'den büyük olana kadar bir toplam değişkenine ekleyen bir algoritma. Bu örnek, koşullu bir döngünün kullanımını açıkça göstermektedir.
* **Faktöriyel Hesaplama**: Bu problem, birden fazla doğru yaklaşım olabileceğini göstermek için dört farklı algoritma kullanılarak çözülmüştür. Çözümler, artan sayaçlı, azalan sayaçlı ve iki farklı for döngü yapısına sahip döngüleri sergilemektedir.
* **Karekök Tahmini**: Bu, bir başlangıç tahmini ve bir hata değerine dayanarak bir sayının karekökünü hesaplayan daha gelişmiş bir örnektir. Algoritma, xi+1​=xi​+b formülüne dayalı yinelemeli bir yöntem kullanır, burada b≅(a−x2)/2x'dir. İşlem, mutlak hata değeri önceden tanımlanmış bir eşikten küçük veya ona eşit olana kadar tekrarlanır. Algoritmanın yürütülmesini izlemek için bir örnek çalışma sunulmuştur:

**Lecture 6: C++ Programming Language**

**English Summary**

This lecture provides an introduction to the C++ programming language. C++ was developed by Bjarne Stroustup in 1979 at Bell Laboratories as a continuation of the C language. The lecture details the build steps of a C++ program: writing the source code in an **Editor**, processing directives with a **Preprocessor**, creating object code with a **Compiler**, linking libraries with a **Linker**, and finally putting the program into memory with a **Loader** for the CPU to execute.

The fundamental structure of a C++ program includes preprocessor directives (e.g., #include <iostream>), namespace declarations (e.g., using namespace std;), and the main() function, which is the program's entry point. Key concepts are explained:

* **Variables**: These refer to memory locations where data is stored. Variable names must consist of letters, numbers, and underscores, and cannot start with a digit. The lecture provides a table of variable types and their value ranges. | Data Type | Byte | Value Range | | :--- | :--- | :--- | | signed char | 1 | -128 to +127 | | unsigned char | 1 | 0 to 255 | | signed int | 4 | -2,147,483,648 to +2,147,483,647 | | float | 4 | +/- 10E38 (7 digits of precision) | | double | 8 | +/- 10E308 (15 digits of precision) |
* **Constants**: These are identifiers whose values remain the same throughout the program, defined using either the #define preprocessor directive or the const keyword.
* **Operators**: A comprehensive set of operators is presented, including Arithmetic, Relational, and Logical operators, along with a table defining their priority.
* **Namespaces**: These are used to solve name conflict problems in larger programs by allowing programmers to define their variables and functions within their own named scope.
* **Library/Header Files**: These files provide access to pre-written functions. For example, #include <iomanip> is used for the setw() function to format output, and #include <cmath> is used for mathematical functions like sqrt().

**Türkçe Özet**

Bu ders, C++ programlama diline bir giriş yapar. C++, 1979 yılında Bjarne Stroustrup tarafından Bell Laboratuvarları'nda C dilinin bir devamı olarak geliştirilmiştir. Ders, bir C++ programının derleme adımlarını ayrıntılı olarak açıklar: kaynak kodun bir **Düzenleyici**'de yazılması, direktiflerin bir **Ön İşlemci** ile işlenmesi, bir **Derleyici** ile nesne kodu oluşturulması, kütüphanelerin bir **Bağlayıcı** ile bağlanması ve son olarak programın CPU'nun yürütmesi için bir **Yükleyici** ile belleğe konulması.

Bir C++ programının temel yapısı, ön işlemci direktiflerini (örneğin, #include <iostream>), isim alanı bildirimlerini (örneğin, using namespace std;) ve programın giriş noktası olan main() fonksiyonunu içerir. Anahtar kavramlar açıklanmıştır:

* **Değişkenler**: Bunlar, verilerin saklandığı bellek konumlarına atıfta bulunur. Değişken adları harflerden, sayılardan ve alt çizgilerden oluşmalı ve bir rakamla başlayamaz. Ders, değişken türleri ve değer aralıkları tablosu sunar. | Veri Tipi | Bayt | Değer Aralığı | | :--- | :--- | :--- | | signed char | 1 | -128 ila +127 | | unsigned char | 1 | 0 ila 255 | | signed int | 4 | -2,147,483,648 ila +2,147,483,647 | | float | 4 | +/- 10E38 (7 basamak hassasiyet) | | double | 8 | +/- 10E308 (15 basamak hassasiyet) |
* **Sabitler**: Bunlar, değerleri program boyunca aynı kalan ve #define ön işlemci direktifi veya const anahtar kelimesi kullanılarak tanımlanan tanımlayıcılardır.
* **Operatörler**: Aritmetik, İlişkisel ve Mantıksal operatörler dahil olmak üzere kapsamlı bir operatör seti, önceliklerini tanımlayan bir tablo ile birlikte sunulur.
* **İsim Alanları (Namespaces)**: Bunlar, programcıların kendi değişkenlerini ve fonksiyonlarını kendi adlandırılmış kapsamlarında tanımlamalarına izin vererek daha büyük programlardaki isim çakışması sorunlarını çözmek için kullanılır.
* **Kütüphane/Başlık Dosyaları**: Bu dosyalar, önceden yazılmış fonksiyonlara erişim sağlar. Örneğin, çıktıyı biçimlendirmek için setw() fonksiyonu için #include <iomanip> ve sqrt() gibi matematiksel fonksiyonlar için #include <cmath> kullanılır.