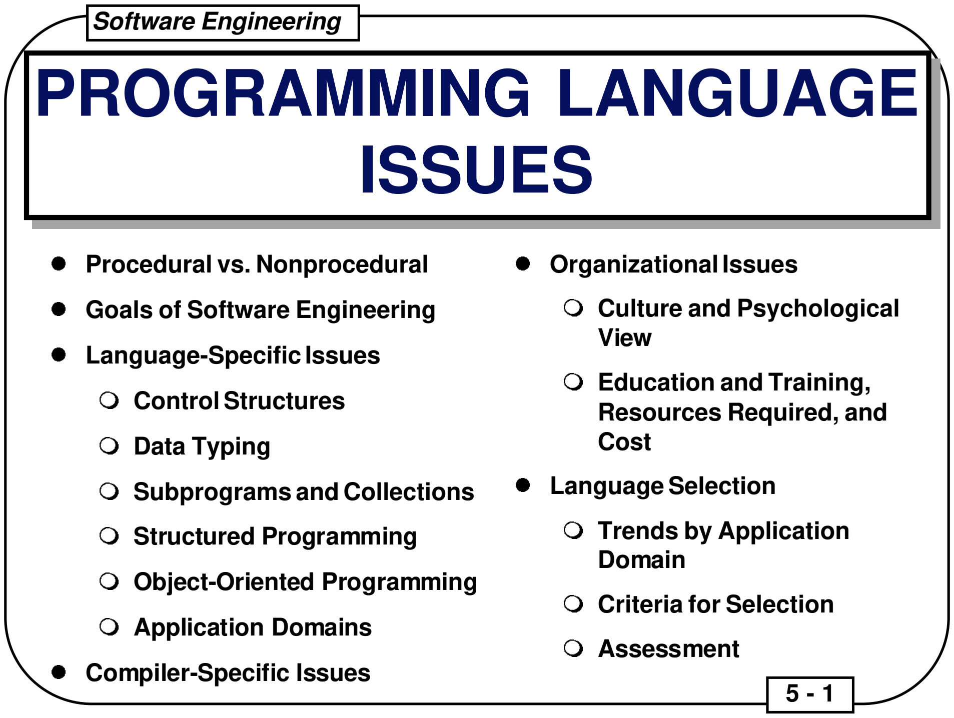


Software Engineering

PROGRAMMING LANGUAGE ISSUES

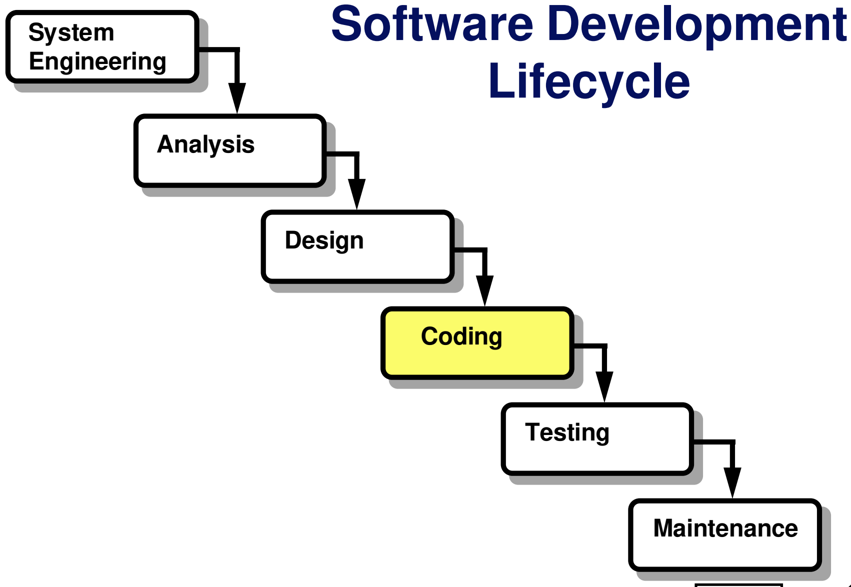
- **Procedural vs. Nonprocedural**
- **Goals of Software Engineering**
- **Language-Specific Issues**
  - ❍ **Control Structures**
  - ❍ **Data Typing**
  - ❍ **Subprograms and Collections**
  - ❍ **Structured Programming**
  - ❍ **Object-Oriented Programming**
  - ❍ **Application Domains**
- **Compiler-Specific Issues**
- **Organizational Issues**
  - ❍ **Culture and Psychological View**
  - ❍ **Education and Training, Resources Required, and Cost**
- **Language Selection**
  - ❍ **Trends by Application Domain**
  - ❍ **Criteria for Selection**
  - ❍ **Assessment**

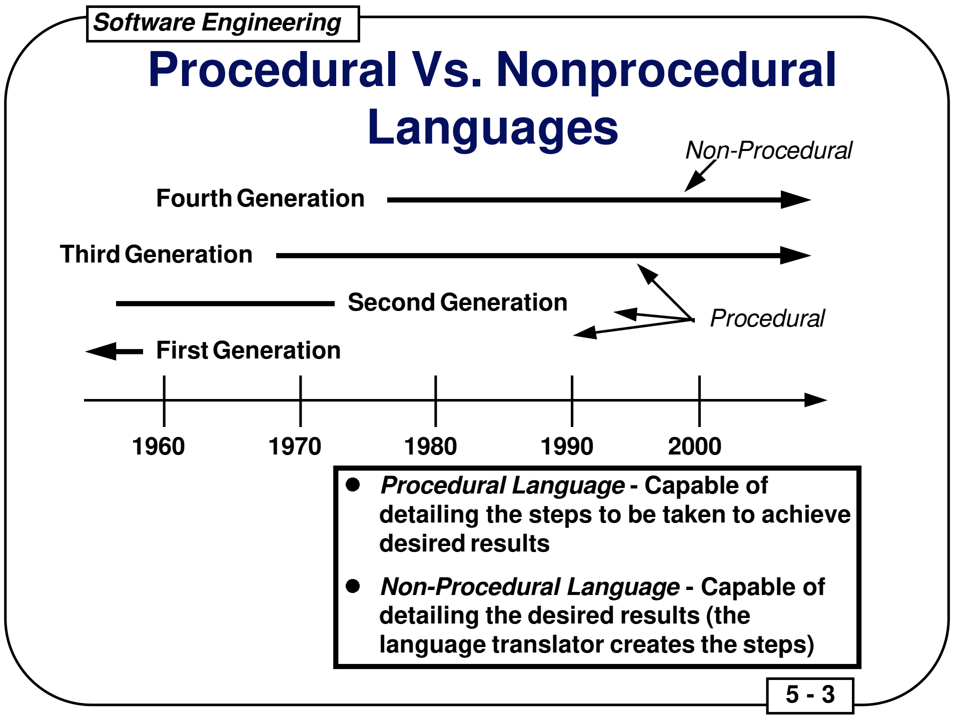
5 - 1

Objectives of Module 5

- Present and discuss the idea that languages can be classified as procedural and non-procedural. Present the four generations of programming languages.
- Review the goals of software engineering and discuss how these goals relate to the programming language.
- Discuss language-specific and compiler-specific issues. Compare and contrast language issues and compiler issues.
- Present and discuss organizational issues relating to the selection of a programming language.
- Present and discuss a method for the selection of a programming language.

Software Development Lifecycle

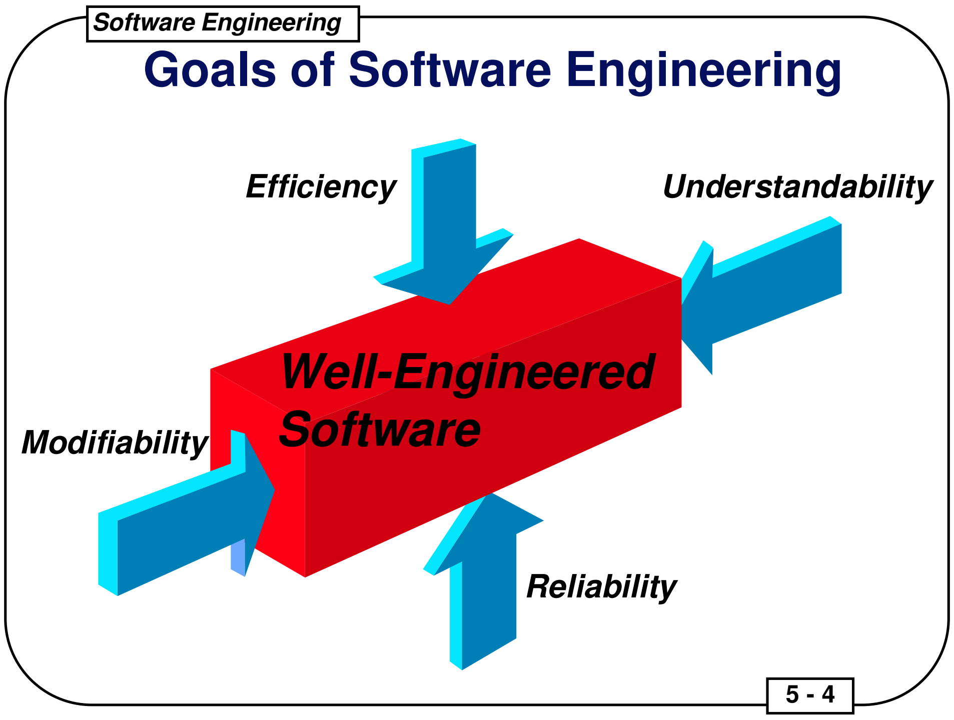




Procedural Vs. Nonprocedural Languages

- ***Procedural Language*** **- Capable of detailing the steps to be taken to achieve desired results**
- ***Non-Procedural Language*** **- Capable of detailing the desired results (the language translator creates the steps)**

- **First Generation** (Procedural Languages)
  - Machine code and assembly languages
  - Machine-dependent
- **Second Generation** (Procedural Languages)
  - FORTRAN, COBOL, ALGOL, BASIC
  - Broad usage, large software libraries, widest familiarity
  - Mature languages and compilers that lack support for current software technology (data abstractions, information hiding, strong typing, collections, structured programming, object-oriented programming)
- **Third Generation** (Procedural Languages)
  - Pascal, Modula-2, C, C++, Ada, LISP, FORTH
  - Three classes: general-purpose, object-oriented, and specialized
  - Support current software technology
- **Fourth Generation** (Non-Procedural Languages)
  - Hypercard, Oracle Query Language
  - Many classes: query, program generators, decision support, prototyping, formal specification, etc.
  - Specific to an application domain



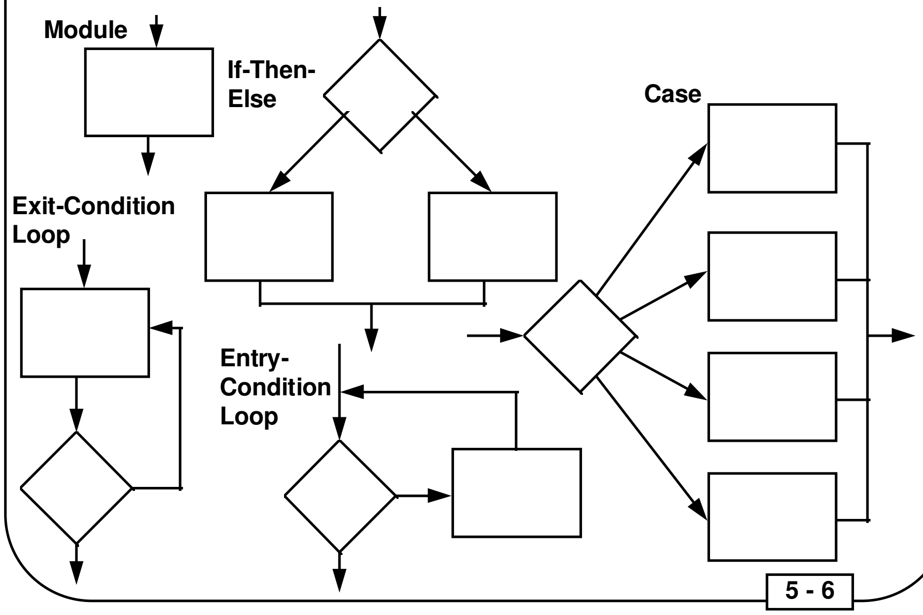
Goals of Software Engineering

- The generations of programming languages have evolved in part to support the four main goals of Software Engineering:
  - The development of software that is **efficient**, able to meet its time and space constraints
  - The development of software that is **reliable**, able to be trusted to perform its functions without error under a variety of conditions
  - The development of software that is **modifiable**, because software of any value is likely to be changed for several reasons:
    - **Corrective Maintenance** - to correct defects uncovered after its release
    - **Adaptive Maintenance** - to change it to work in new environments (such as new operating systems or new target platforms)
    - **Enhancement** - to add features not considered during its original development
  - The development of software that is **understandable**, because people in the development organization often change, and an organization cannot count on the original developer being around when changes to the software are required
- Two key ingredients are required to meet these goals:
  - a **programming language** which includes features to support these goals and
  - a **software development process** that ensures that the design of the software and the use of the programming language properly supports these goals

Language-Specific Issues

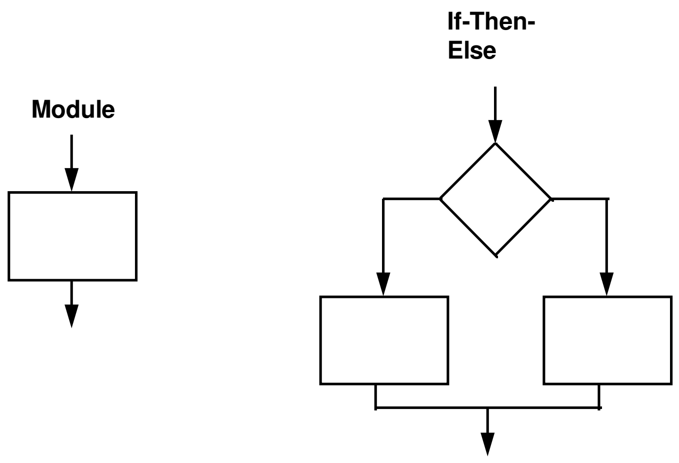
- Control Structures
- Data Typing
- Subprograms and Collections
- Structured Programming
- Object-Oriented Programming
- Application Domains

Control Structures



- Most Second and Third Generation Languages Support the Basic Control Structures:
  - **Module** - a logical collection of related operations
  - **If-Then-Else** - Perform a test and then perform one procedure or another
  - **Case** - Perform a test and then perform one of a number of procedures
  - **Entry-Condition Loop** - Perform a test; if the test succeeds, perform a procedure and go back to perform the test again; if the test fails, go on
  - **Exit-Condition Loop** - Perform a procedure; perform a test; if the test succeeds, go back to perform the procedure again; if the test fails, go on
- Modules may contain any combination of the basic control structures, including other modules

Control Structures, Continued



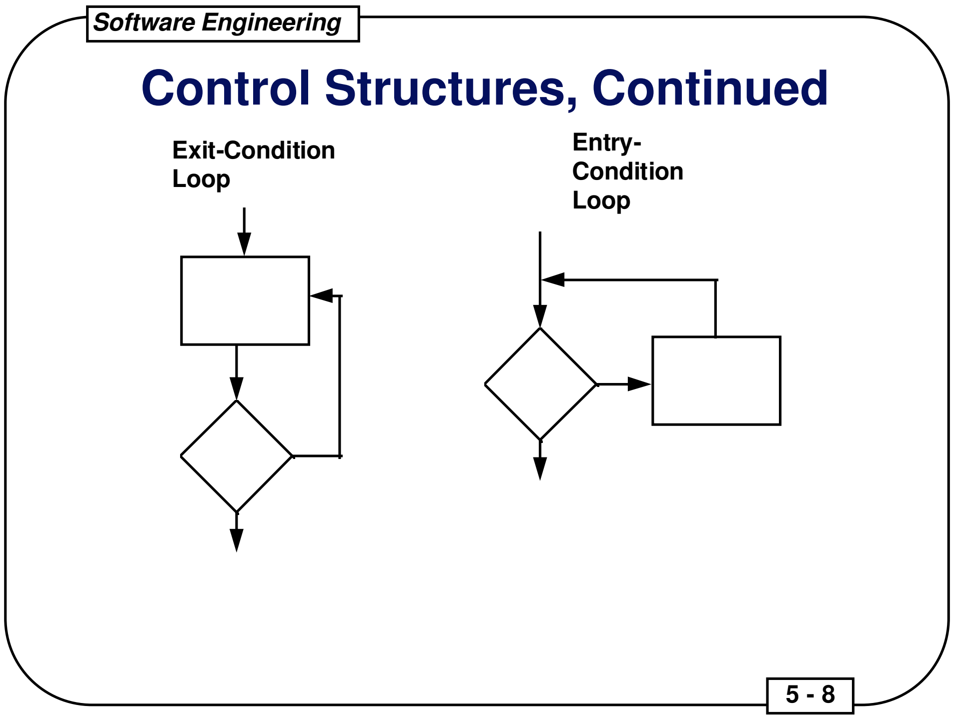
- Most Third Generation Languages support all the control structures, but the syntax may be different. The following are examples of functions (modules in the diagram above) containing if-then-else statements in C (or C++) and Ada:

C or C++ Function

```
int min(int a, int b) {
  int result;
  if (a < b) {
    result = a;
  } else {
    result = b;
  }
  return result;
}
```

Ada Function

```
function min
  (a, b: in integer)
    return integer is
  result : integer;
begin
if a < b then
  result := a;
else
  result := b;
end if;
return result;
end min;
```



Control Structures, Continued

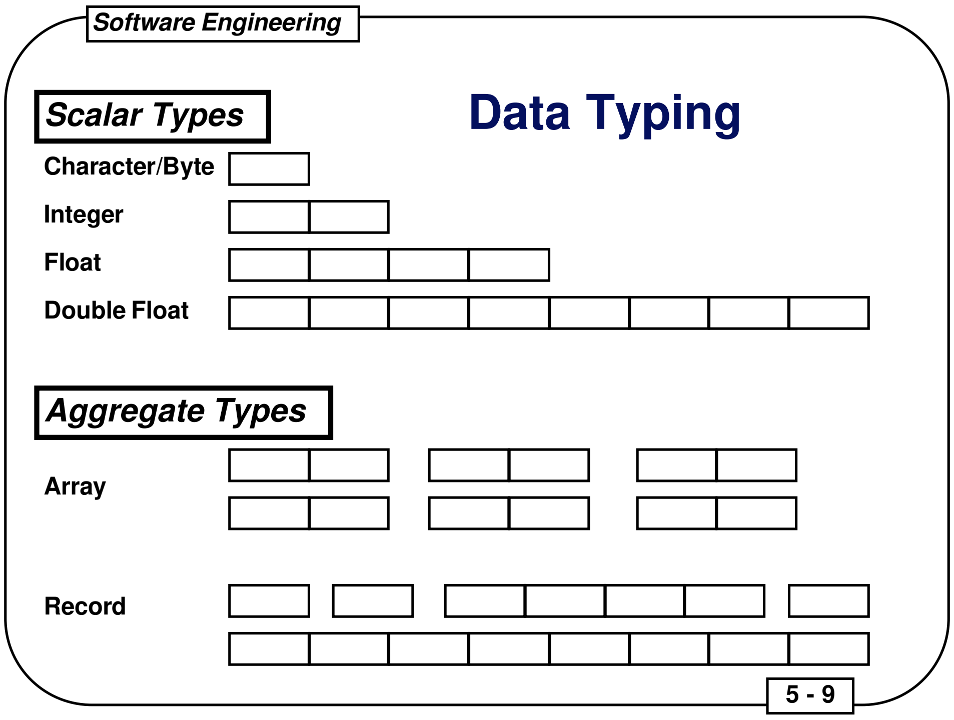
- The following code fragments show entry-condition loops in C (or C++) and Ada:

C or C++

```
sum = 0;
i = 0;
while (i <= 10) {
  sum += i;
  i++;
}
```

Ada

```
sum := 0;
i := 0;
while i <= 10 loop
  sum := sum + i;
  i := i + 1;
end loop;
```

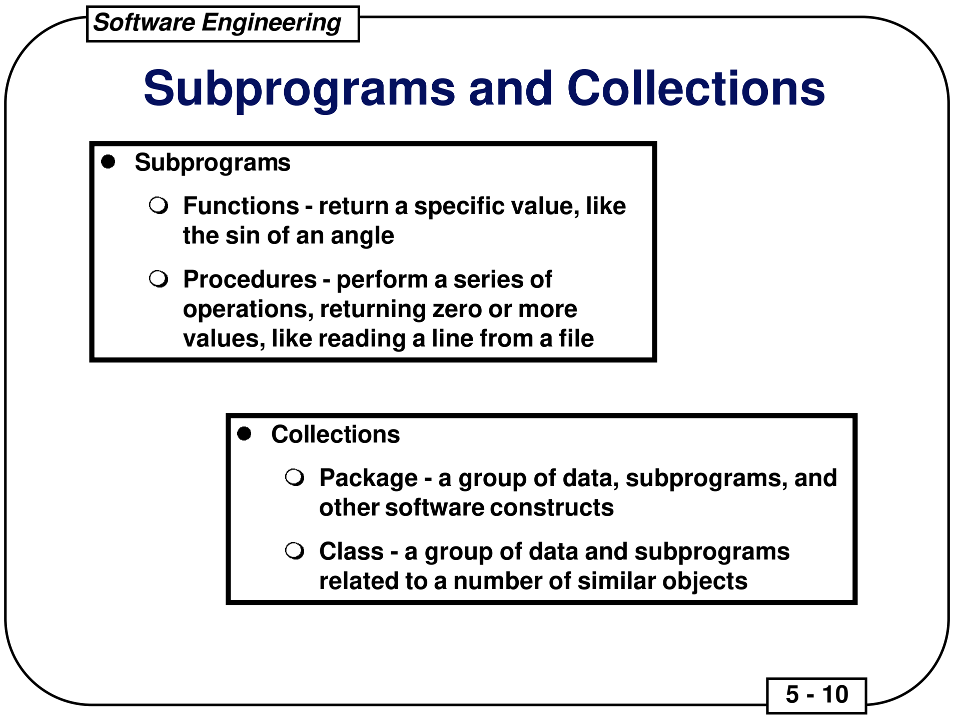


Data Typing

Scalar Types

Aggregate Types

- There are two basic kinds of data:
  - **Scalar** - used to represent a single, non-divisible entity, like a character or an integer
  - **Aggregate** - used to represent a collection of other aggregates and scalars
- In general, Second and Third Generation Languages support scalars and array aggregates and Third Generation Languages support record aggregates.
- Record aggregates are the basis for data abstraction in many methodologies and object definition in object-oriented methodologies.
- **Strong type checking** in Third Generation Languages helps to identify potential problems at compile time. Classically, interface matching has been a problem with Second Generation Languages, and newer Third Generation Languages have addressed the problem of ensuring that the interfaces are correct by putting the burdon of checking these interfaces on the compiler rather than the coder.



Software Engineering

Subprograms and Collections

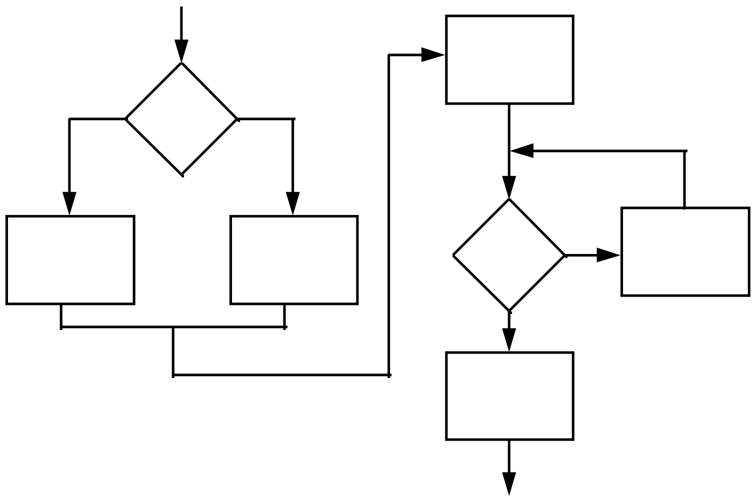
- Subprograms
  - Functions - return a specific value, like the sin of an angle
  - Procedures - perform a series of operations, returning zero or more values, like reading a line from a file

- Collections
  - Package - a group of data, subprograms, and other software constructs
  - Class - a group of data and subprograms related to a number of similar objects

5 - 10

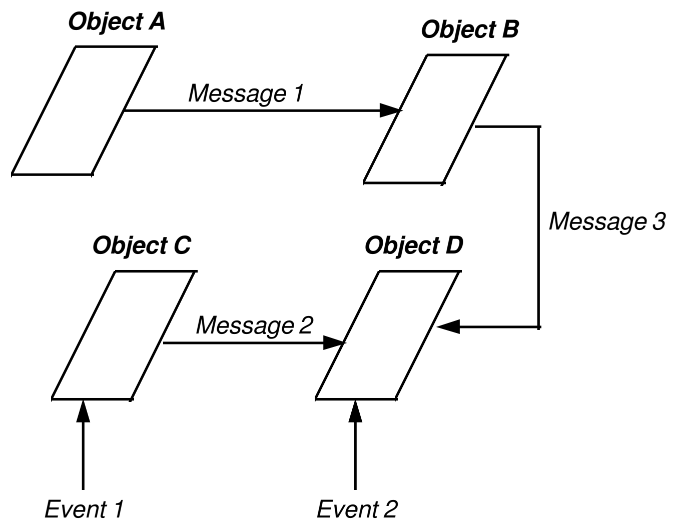
- A subprogram is a body of code that performs a specific action or set of actions. Subprograms are supported by Second and Third Generation Languages.
- Collections are groups of subprograms and data which are related to each other in some way. In certain languages, a collection may have more than just subprograms and data associated with it. Collections are supported by some Third Generation Languages (such as Ada, C++, and Modula-2).
- Subprograms and collections support both structured and object-oriented programming. In particular, modularity, information hiding, data abstraction, and object-oriented structuring are directly supported.

Structured Programming

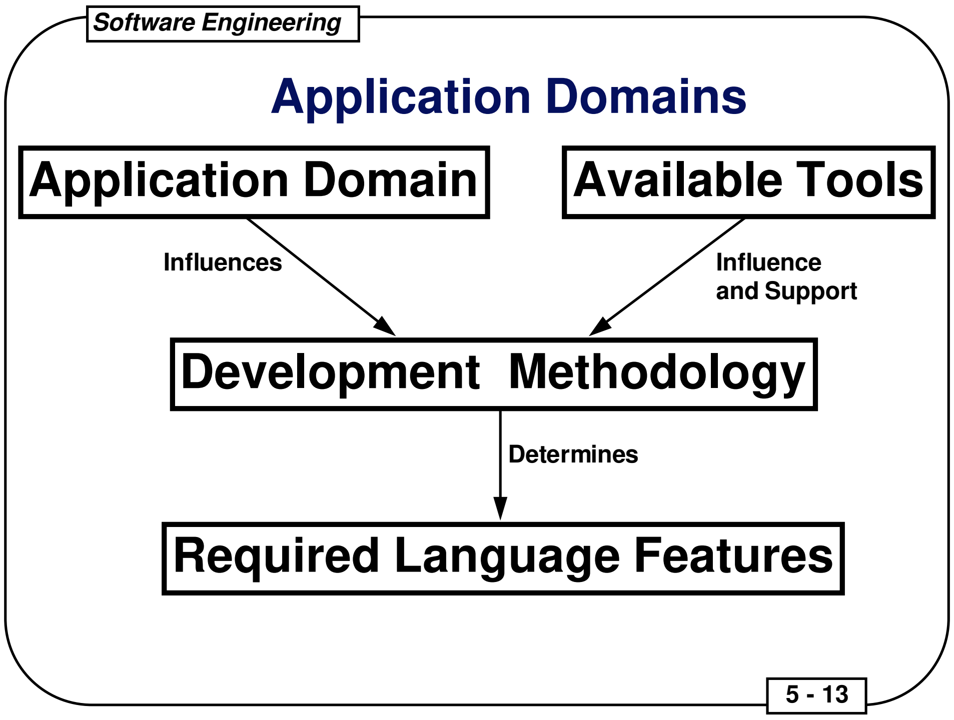


- **Structured programs** are those programs composed of combinations of the basic control structures. Some Second Generation and all Third Generation Languages provide control structures to the extent that all programs written in these languages may be structured.
- Structured programs tend to be more readable, understandable, and maintainable than unstructured programs.
- The desire for structured programming grew out of situations in which the application software could meet its time and space constraints, so there was an opening to consider designing the code with less regard to the time and space constraints and more regard to other concerns, particularly the life cycle cost of the software. The cost of the software became more pronounced in the maintenance phase, so efforts were made to make the code easier to maintain.
- Two main attributes of the software were shown to impact its maintenance significantly: the degree of **cohesion** within a module and the degree of **coupling** between modules. It was found desirable to make modules highly cohesive (so they could be viewed and tested as autonomous units of the software system) and loosely coupled (so testing of one module did not require another module to be involved).

Object-Oriented Programming

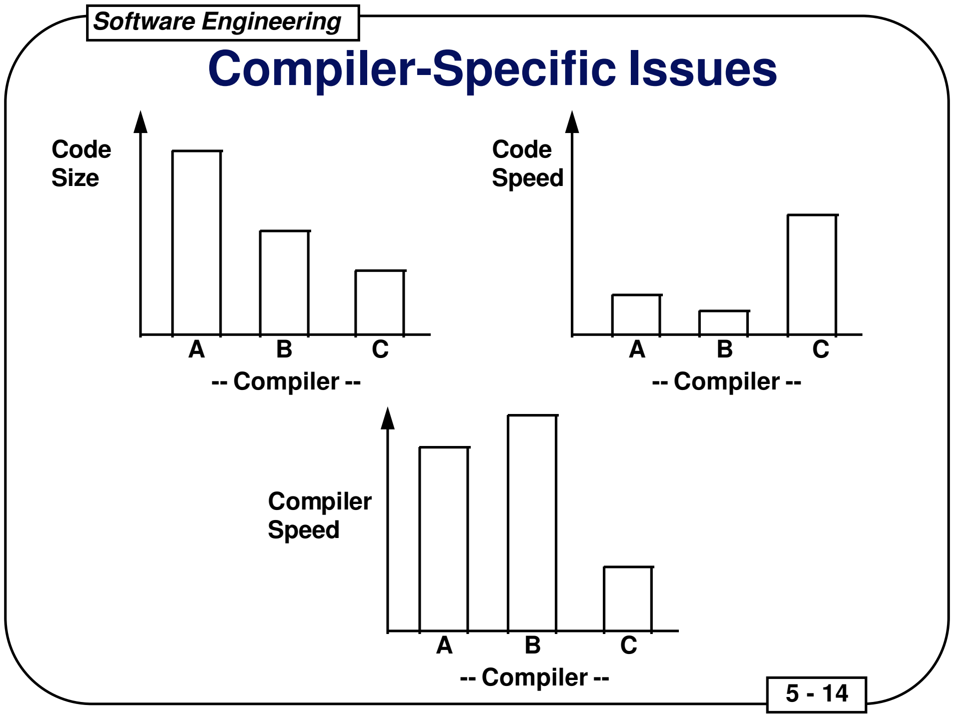


- **Object-Oriented Programming** arose from a need to manage very complex software systems -- systems which were so complex that Structured Programming was inadequate in and of itself. Object-Oriented Programming is a logical extension of Structured Programming, adding to the structured programming paradigm the concepts of processing asynchronous events (such as mouse clicks and interrupts) and performing concurrent operations (using messages to communicate between the concurrent processes).
- Object-Oriented Programming allows the program to become a model of the real-world problem. The objects in the real world are modelled as objects in the program, and the programmatic object models define the key attributes and behaviors of the real-world objects.
- Constructs beyond the control structures and data typing described previously are needed to support object orientation:
    - A means to relate objects as members of a **class**, which is a common definition which groups objects of related attributes and behaviors together
    - A means to derive new classes from existing classes, inheriting all the attributes and behaviors or a selected subset of them from the existing classes in the new classes
    - A means to model concurrency, since real-world objects often operate in parallel with each other
    - A means to handle events, since real-world objects often cause or react to events



- **Development Considerations**
  - Structured or Object-Oriented?
  - Textual or Graphical?
  - Individual or Team?
  - Code Hacking or Code Engineering?
- **Language Features**

	Available In		
Feature	*C*	*C++*	*Ada*
Control Structures	x	x	x
Data Abstraction	x	x	x
Packages			x
Classes		x	x
Inheritance		x	
Dynamic Binding		x	
Concurrency			x
Event Detection and Processing	x	x	x
Information Hiding			x
Strong Type Checking		x	x


Software Engineering

Compiler-Specific Issues

5 - 14

EFFICIENCY and OPTIMIZATIONS

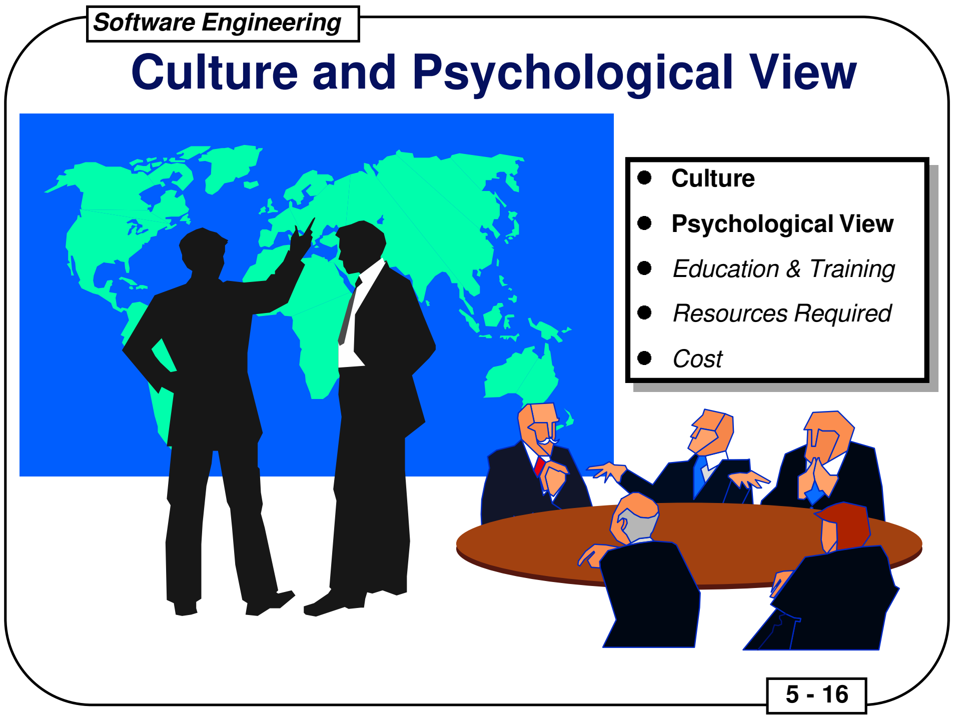
- The size and speed of the code produced by a given compiler is a feature of the compiler -- *not* a feature of the language.
- The speed of the compiler and the tools that come with it to aid the developer are also features of the compiler rather than the language.

ENVIRONMENT ACCESS

- Access to the environment of the target is often an issue which is dependent on the compiler rather than the language.
- The environment of the target includes the following:
  - The command line, the system variables, and the process parameters
  - The required libraries and bindings, such as X Windows and SQL
  - The target operating system
  - Operating system and applications software with which the software under development must interface

Organizational Issues

- **Culture and Psychological View**
- **Education and Training, Resources Required, and Cost**



Software Engineering

Culture and Psychological View

- **Culture**
- **Psychological View**
- *Education & Training*
- *Resources Required*
- *Cost*

5 - 16

- **Culture**

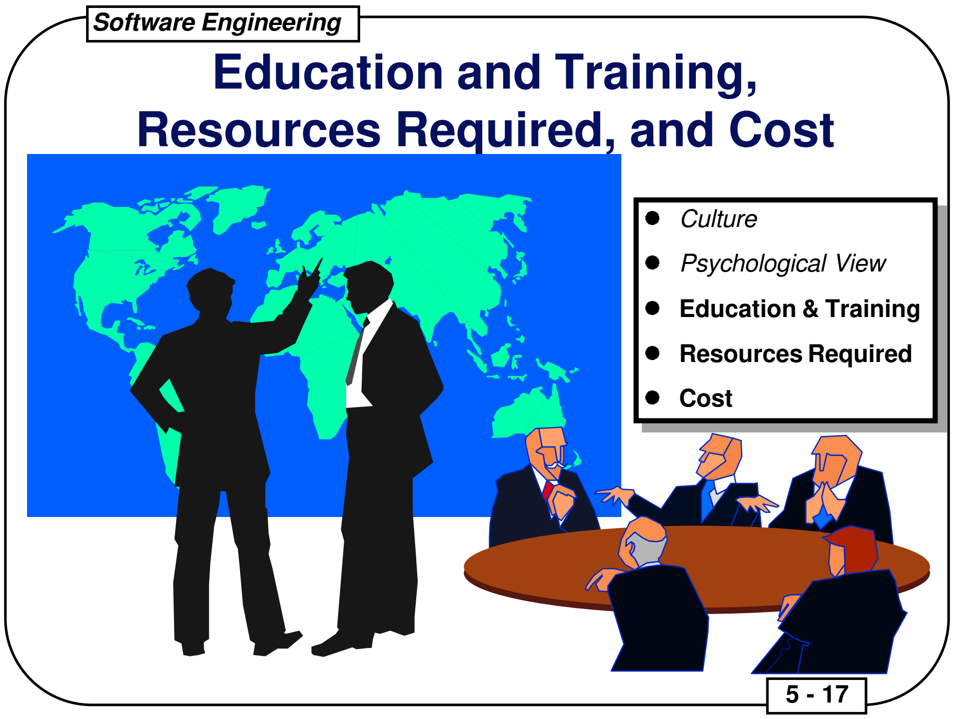
  The culture of an organization can impact the software development process significantly. An NIH (Not Invented Here) mind set can inhibit software reuse, for example. An environment in which the projects are in a mode of "putting out fires" can inhibit technology development. Culture is one of the easiest things to observe and one of the hardest to change.

- **Psychological View**

  The psychological view of a programming language focusses on human concerns such as ease of use, simplicity in learning, improved reliability, reduced error frequency, ease to maintain, and enhanced user satisfaction. Simultaneously, an awareness of machine efficiency, software capacity, and hardware constraints must be maintained.

  A number of psychological characteristics are evidenced in the design of a programming language:

  - **Uniformity** - the degree to which a language uses consistent notation, applies seemingly arbitrary restrictions, and supports syntactic or semantic exceptions to the rule
  - **Ambiguity** - how the compiler interprets a statement as opposed to how a human naturally interprets the same statement
  - **Compactness** - the amount of code-oriented information that must be recalled from human memory at a time



Education and Training, Resources Required, and Cost

- *Culture*
- *Psychological View*
- **Education & Training**
- **Resources Required**
- **Cost**

- **Education and Training**

  Most education and training for the introductory use of a programming language can usually be accomplished in a few weeks, although it may take a few years of on-the-job use of a language and supplementary advanced courses in it before a programmer masters the language. The initial training is just a starting point, however. The people must apply what they have learned as quickly as possible afterward the initial training. In addition, access to a consultant during this initial application period can be quite beneficial.

- **Resources Required**

  Software tools to support the designers and developers, disk space to support those tools and their data files, CPU speed to allow the tools to run quickly for the designers and developers, and disk space to provide room for intermediate files and different versions of the software under development are all issues to be considered. Huge impacts can be made to productivity if any of these items are lacking.

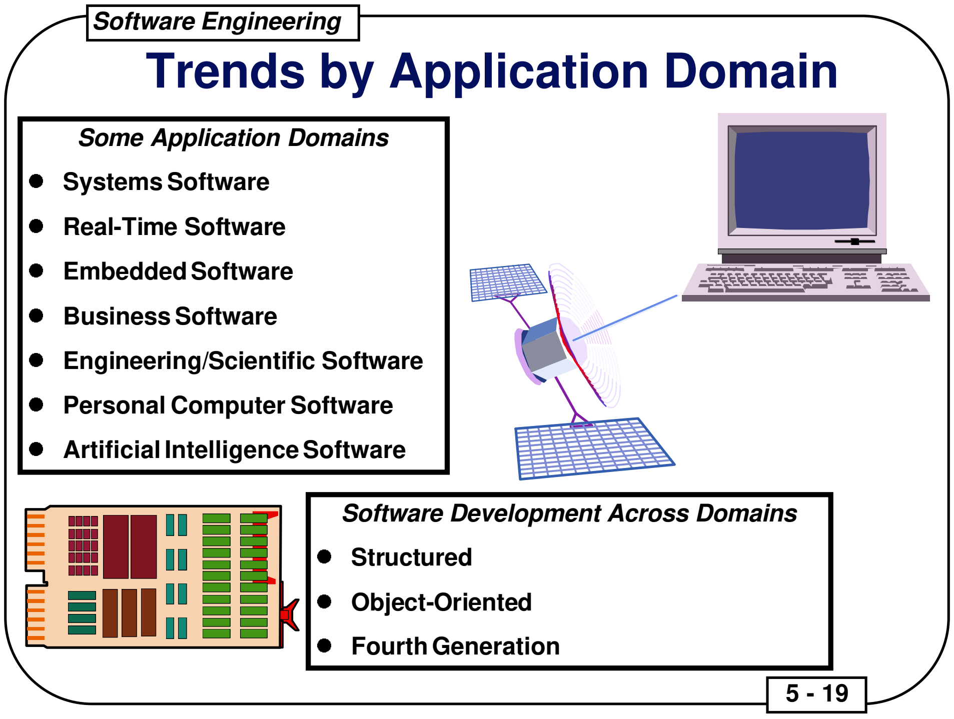
  Modern Third Generation Languages often require more CPU power to do their work than previous languages. The compiler is relieving the programmer of many tasks relating to checking his code for validity before it runs, so the compiler is much more complex, requiring more CPU power to run in a reasonable amount of time.

- **Cost**

  The cost of education, training, resources, software development, and software maintenance can be significantly impacted by the selection of the language. Cost in this sense refers to both dollars and time.

Language Selection

- Trends by Application Domain
- Criteria for Selection
- Assessment



Trends by Application Domain

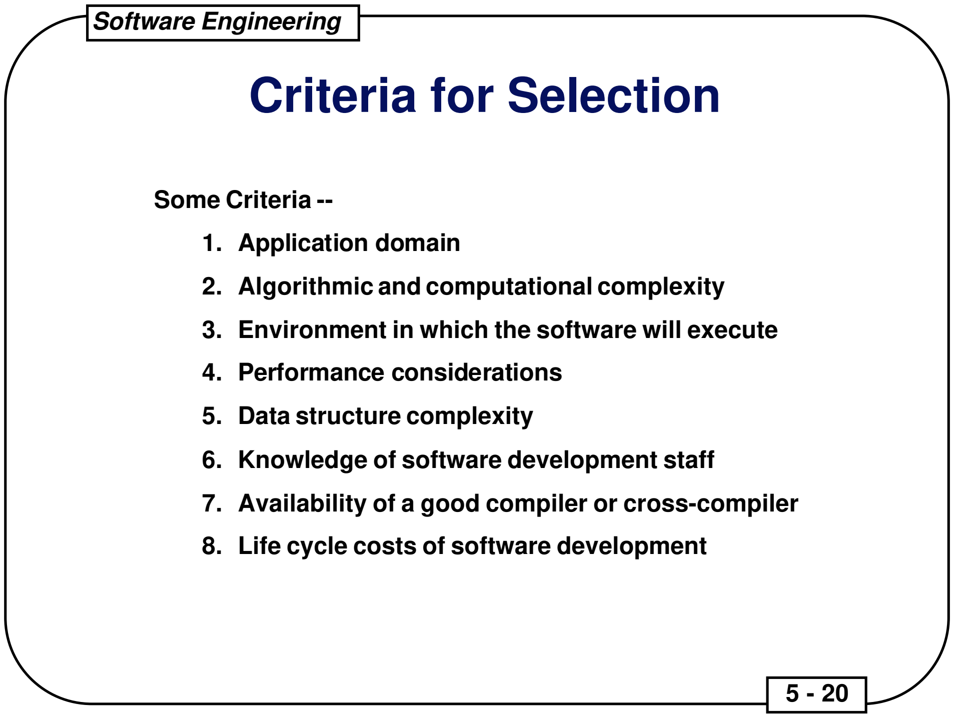
Some Application Domains

- Systems Software
- Real-Time Software
- Embedded Software
- Business Software
- Engineering/Scientific Software
- Personal Computer Software
- Artificial Intelligence Software

Software Development Across Domains

- Structured
- Object-Oriented
- Fourth Generation

- Systems Software - C, C++
- Real-Time Software - Ada, C, Modula-2, FORTH, FORTRAN, assembly
- Embedded Software - Ada, C, assembly
- Business Software - COBOL, 4GLs
- Engineering/Scientific Software - FORTRAN, C, Pascal, Ada
- Personal Computer Software - C, assembly, BASIC, Pascal
- Artificial Intelligence Software - LISP, PROLOG, OPS5



Software Engineering

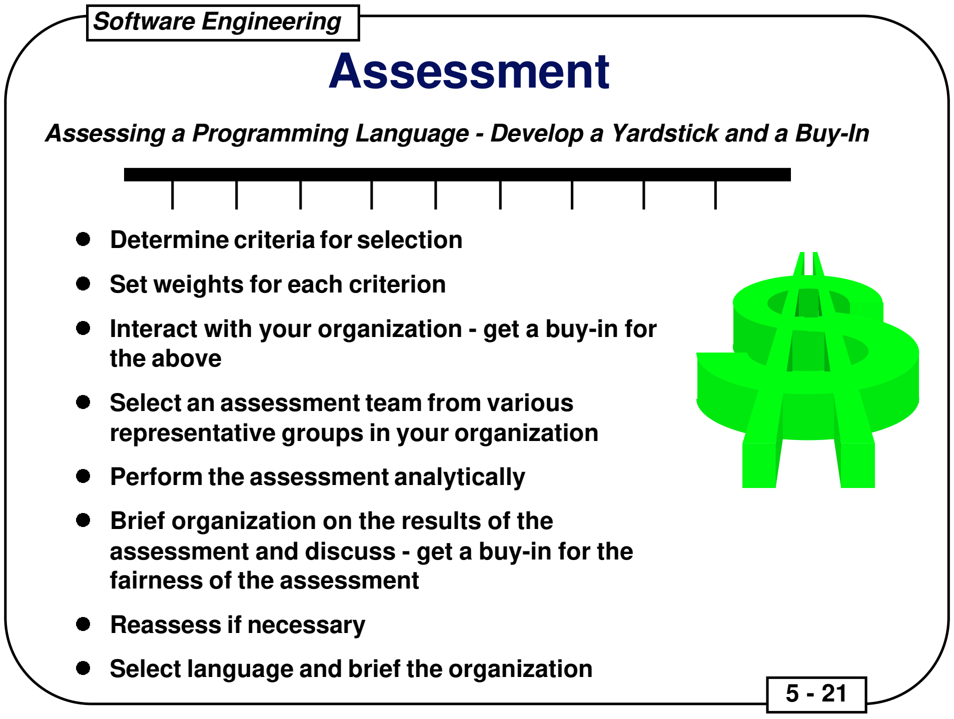
Criteria for Selection

Some Criteria --

1. **Application domain**
2. **Algorithmic and computational complexity**
3. **Environment in which the software will execute**
4. **Performance considerations**
5. **Data structure complexity**
6. **Knowledge of software development staff**
7. **Availability of a good compiler or cross-compiler**
8. **Life cycle costs of software development**

5 - 20

- The applications domain of a project is a criterion that is applied most often during language selection.
- The proliferation of "new and better" programming languages continues at a significant pace. There are now over 4,500 high-order programming languages, not counting dialects (such as FORTRAN-66 and FORTRAN-77).
- It is sometimes better to choose a "weaker" (old) programming language:
  - Support software (compilers and other tools) is solid and reliable
  - Documentation is well-written
  - Everyone on the software development team knows the language
  - The language has been applied in the past with good results
  - There may be a psychological resistance to change
- It is sometimes better to choose a newer programming language:
  - The desired methodology is supported only by the newer languages
  - The software's environment does not support the older languages
  - A newer language may better support the overall goals of software engineering -- being able to generate efficient, reliable, modifiable, and understandable code



Assessment

Assessing a Programming Language - Develop a Yardstick and a Buy-In

- **Determine criteria for selection**
- **Set weights for each criterion**
- **Interact with your organization - get a buy-in for the above**
- **Select an assessment team from various representative groups in your organization**
- **Perform the assessment analytically**
- **Brief organization on the results of the assessment and discuss - get a buy-in for the fairness of the assessment**
- **Reassess if necessary**
- **Select language and brief the organization**

- This slide shows the recommended steps (in the opinion of the author) for the selection of a programming language.
- An important feature of this process is the buy-in of the process, the intermediate results, and the final result by the organization which will be affected by this assessment.
- Another important feature of this process is the detached method of assessing by using a set of weighted criteria. This generate numbers, and people emotionally and logically buy into numbers better than emotions and hand waiving.